Laboratorio de Métodos Numéricos

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

TP2 : Reconocimiento de dígitos

Integrante	LU	Correo electrónico
Enzo Samuel Cioppettini	405/15	tenstrings5050@gmail.com
Tomás Ariel Pastore	266/15	pastoretomas96@gmail.com
Tomás Jaratz	59/15	tomyjara1@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Resumen

El trabajo de investigación consiste en introducirnos al campo de machine learning por medio de un interesante contexto como lo es el reconomiento de dígitos. Poniendo en evidencia la cercanía de este terreno recién mencionado con los métodos numéricos que se estudian en la materia. Se considerará una base de datos de imágenes de dígitos etiquetadas. Esta se particionará en un conjunto "train" y otro "testing" con el fin de poder entrenar a la computadora para que reconozca patrones que implican los dígitos, y luego, poder comprobar sobre el conjunto testing si se predicen bien las etiquetas de estos. Para la resolución del problema se emplean métodos numéricos como **k-NN**(k vecinos más cercanos), **PCA**(Análisis de Componentes Principales) y **PLS-DA**(Análisis discriminante con cuadrados mínimos parciales). Finalmente, por medio de ciertos experimentos se analiza la calidad de los resultados obtenidos mediante el uso de los diferentes métodos; variando el contexto de uso(parámetros y base de datos de entrenamiento). Se generan conclusiones, y se propone criterios para sugerir en que contexto cada método es conveniente ante los demás.

Índice general

1. Introducción **1**

2. Desarrollo **2**

2.1. Implementación de los métodos utilizados 2

2.1.1. k-NN 2

2.1.2. PCA 3

2.1.3. PLS-DA 3

2.2. Métricas consideradas 4

3. Experimentación **5**

3.1. Experimento 1: Calidad de resultados 5

3.2. Experimento 2: Calidad de resultados 13

3.3. Experimento 3: Estudio de particiones 19

3.4. Experimento 4: Optimalidad 21

3.5. Experimento 5: Optimización del tiempo de cómputo 22

3.6. Experimento 6: Overfitting 25

3.7. Resultados en Kaggle 28

4. Conclusiones finales **29**

5. Optimizaciones y puntos opcionales: **30**

6. Bibliografía **32**

Introducción

En este trabajo, estudiaremos el campo de machine learning bajo el contexto de uso de reconocimiento de dígitos.
Se considerará una base de datos de imágenes de dígitos etiquetados. Esta se particionará en un conjunto "train" y otro "testing" con la finalidad de:

- 1) Entrenar a la computadora para que reconozca patrones que implican los dígitos.
- 2) Evaluar sobre el conjunto testing si se predicen bien las etiquetas de estos.

Realizar toda la experimentación sobre una única partición de la base podría resultar en una incorrecta estimación de parámetros, como por ejemplo el conocido overfitting. Luego, se considera la técnica de cross validation, en particular el K- fold cross validation, para realizar una estimación de los parámetros del modelo que resulte estadísticamente más robusta.

Como instancias de entrenamiento, se tiene una base de datos de n imágenes de M $\times$ M pixeles, las cuales se encuentran etiquetadas con el dígito, 0-9, al que corresponden. Definimos como $m = M \times M$ al número total de pixeles de una imagen. Asumiremos también que las imágenes se encuentran en escala de grises (cada pixel corresponde a un valor entre 0 y 255, indicando la intensidad del mismo) y que el etiquetado no contiene errores. El objetivo del trabajo consiste en utilizar la información de la base de datos para, dada una nueva imagen de un dígito sin etiquetar, determinar a cuál corresponde teniendo en cuenta factores de calidad y tiempo de ejecución requeridos. Una primera aproximación es utilizar el conocido algoritmo de k Vecinos más Cercanos (k-NN, por su nombre en inglés). Sin embargo, este es sensible a la dimensión de los objetos a considerar y, aún aplicando técnicas de preprocesamiento, puede resultar muy costoso de computar. Teniendo en cuenta esto, una alternativa interesante de preprocesamiento es buscar reducir la cantidad de dimensiones de las muestras para trabajar con una cantidad de variables más acotada y, simultáneamente, buscando que las nuevas variables tengan información representativa para clasificar los objetos de la base de entrada. En esta dirección, consideraremos dos métodos de reducción de dimensiones: Análisis de Componentes Principales (PCA, por su sigla en inglés) y Análisis discriminante con cuadrados mínimos parciales (PLS-DA, por su sigla en inglés).

Desarrollo

Implementación de los métodos utilizados

k-NN

Descripción del método: En su versión más simple, este algoritmo considera a cada objeto de la base de entrenamiento como un punto en el espacio, para el cual se conoce a qué clase corresponde (en nuestro caso, qué dígito es). Luego, al obtener un nuevo objeto que se busca clasificar, simplemente se buscan los k vecinos más cercanos y se le asigna la clase que posea el mayor número de repeticiones dentro de ese subconjunto, es decir, la moda. Con este objetivo, podemos representar a cada imagen de nuestra base de datos como un vector $x_i \in \mathbb{R}^m$, con i = 1, . . . , n, y de forma análoga interpretar las imágenes a clasificar mediante el algoritmo k-NN .

Implementación: La primera implementación de k-NN consistió en calcular las n distancias entre las imagenes de la base train y la nueva imagen, almacenando los resultados en un vector de tuplas del tipo <distancia, label>. Luego ordenamos el vector de forma creciente por primera coordenada con la función qsort de la std, seleccionamos las primeras k labels del vector y las pusimos en un vector de 10 posiciones representando estas los dígitos posibles. Retornamos finalmente la posición del vector tal que el elemento que contuviera fuese el máximo del vector. Es decir, la moda. Este algoritmo utilizaba doubles para realizar el cálculo de las distancias entre vectores, pero sus parámetros de entrada eran vectores de unsigned char.
Luego realizamos una versión de k-NN análoga pero con parámetros de entrada de tipo vector<double>ya que los vectores transformados tras los métodos de PCA y PLS-DA no son necesariamente casteables a unsigned char sin perdida de presición.
Finalmente implementamos optimizaciones para estas dos versiones, que se detallarán en el capítulo "Optimizaciones y puntos opcionales".

PCA

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

$$M = X^t X$$

a 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 del dígito x_i como el vector $tcPCA(x_i) = (v_1^t x_i, v_2^t x_i, ..., v_\alpha^t x_i) \in \mathbb{R}^\alpha$, donde $\alpha \in 1, \ldots, m$ es un parámetro de la implementación. Este proceso corresponde a extraer las α primeras componentes principales de cada imagen. La intención es que tcPCA(x_i) resuma la información más relevante de la imagen, descartando los detalles o las zonas que no aportan rasgos distintivos.

Implementacion: Para implementar este método se siguieron los pasos propuestos en el informe y en la presentación. Se decidió hacer una clase en la cual guardar la matriz a la cual se le aplicó la transformación característica, y los autovectores, como miembros. Esto fue para encapsular la información en un solo objeto, para no utilizar variables globales "sueltas". Además, aunque no se implementó una clase matriz como tal, se sobrecargaron la mayoría de los operadores para la representación de matriz que escogimos, en busca de que el código fuese más parecido a las definiciones matemáticas, lo cuál consideramos que facilita la lectura y razonamiento sobre los algoritmos implementados. Para calcular los autovectores se implementó el método de la potencia basándonos en el pseudocódigo que estaba en la presentación del trabajo práctico, utilizando la función *std::rand* para generar la estimación inicial, utilizando como semilla el reloj del sistema.

PLS-DA

Descripción del método: En el caso de PLS-DA, la idea es similar al método de componentes principales con la diferen- cia de la información original que se utiliza para realizar la transformación. En este caso, en vez de utilizar solamente las imágenes de entrenamiento, se utilizan también las clases a las que pertenecen dichas imágenes para influenciar la transformación característica. Definimos X de manera análoga al algoritmo PCA. preY $\in \mathbb{R}^{nx10}$ como una matriz que tiene un 1 en la posición preY i,j si la muestra i de la base de entrenamiento corresponde al dígito j, y -1 en el caso contrario. Por último, definimos Y como la matriz resultante de tomar preY , sustraer el promedio de todas las filas a cada una de ellas, y dividir por la raíz cuadrada de la cantidad de muestras menos 1. Luego, se utiliza el siguiente pseudocódigo para generar los vectores w_j .

PLS(**Inout** X matriz, **Inout** Y matriz, **In** γ int)

for i = 1 . . . γ do

1: $M_i \leftarrow X^t Y Y^t X$

2: calcular wi como el autovector asociado al mayor autovalor de M_i

3: normalizar w_i, definir t_i como Xw_i

4: normalizar t_i

5: actualizar X como X - $t_i^t t_i$ X

6: actualizar Y como Y - $t_i^t t_i$Y

end for

return $w_1, w_2, ..., w_\gamma$

En el anterior pseudocódigo, γ es un parámetro para controlar la cantidad de dimensiones a utilizar. Una vez obtenidos los vectores w_j , la transformación característica de este método se define de forma análoga como $tcPLS(x_i) = (w_1^t x_i, w_2^t x_i, ..., w_\gamma^t x_i) \in \mathbb{R}^\gamma$ Los métodos PCA y PLS-DA recientemente presentados no son métodos de clasificación en sí, sino que sirven para realizar una transformación de los datos de entrada. Dada una nueva imagen x de un dígito, se calcula tc_m(x), siendo tc_m la transformación caracterítica del método que se esté utilizando, y se compara con tc_m (x_i), para i = 1, . . . , n, utilizando algún criterio de clasificación adecuado, como por ejemplo k-NN.

Implementacion: En este caso, no se implemento una clase, sino que se guardó la matriz de cambio de base en el main como variable global, así como la media, para poder ser utilizadas para transformar cada dígito sin necesidad de recalcularlas para cada dígito dentro de un mismo K fold. Básicamente seguímos el pseudo-código entregado por la cátedra y reutilizamos la función del método de la potencia implementada previamente en PCA. Implementamos además dos funciones que utilizamos bastante que son transpuesta(calcula la transpuesta de una matriz) y producto2, que realiza el producto matricial de 2 matrices. También utilizamos así como se hizo en PCA, los operadores definidos en operadores.hpp, que facilitaron en gran medida la implementación, al menos considerando la reducción de cantidad de código. Nos dimos cuenta de que el orden en el que realizabamos la multiplicación de matrices era relevante. Pues en una parte del algoritmo de la función pls teniamos un producto triple de matrices, el cual si asociabamos de forma (AB)C, AB resultaba ser una matriz de doubles tal que AB $\in \mathbb{R}^{nxn}$ lo que en nuestro caso es 37800x37800. Cada double ocupa 8 bytes en memoria, por lo que necesitariamos $(37800)^2 * 8$ bytes de memoria RAM para poder almacenar dicha matriz. La cuenta es equivalente a 10.6 Gigabytes. Por lo que forzosamente, al tildarse las computadoras, nos dimos cuenta de que convenía asociar de forma A(BC); de esta forma las matrices eran de un tamaño soportado.

Métricas consideradas

Para evaluar los resultados de los experimentos, se consideraron todas las métricas mencionadas en el enunciado, estas se implementaron en el archivo metricas.hpp. A su vez, se implementó una función que realiza una matriz de confusión. En ciertos casos, para condensar la información del recall y la precisión se utilizó el F1-Score.

Experimentación

Experimento 1: Calidad de resultados

Estudio bajo distintos parámetros

En este experimento nos enfocamos en estudiar los resultados obtenidos al combinar PCA y PLS-DA con Knn para un rango amplio de combinaciones de valores de k, α y γ sobre distintas métricas y tiempo de cómputo.

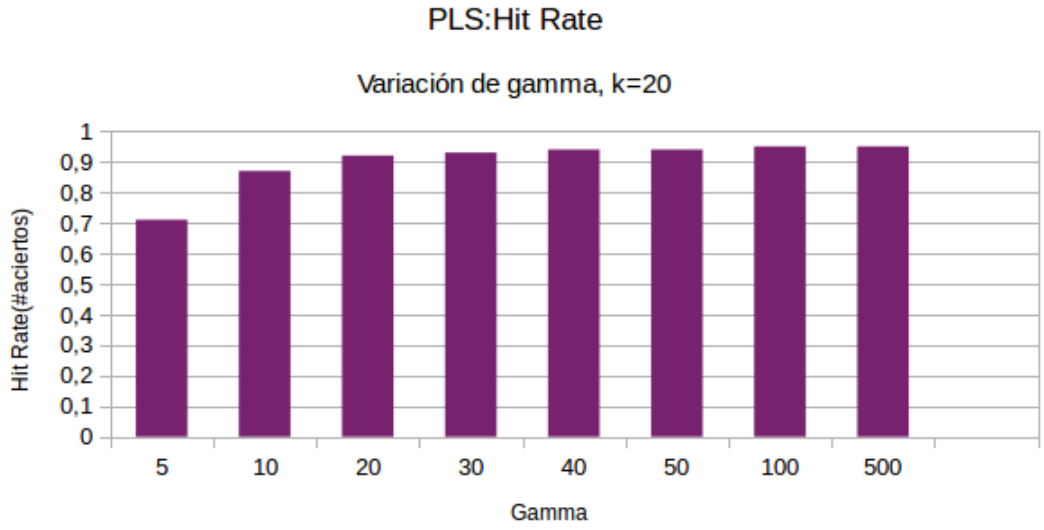
Descripción : Para realizar este experimento utilizamos una sola partición del conjunto de entrenamiento y para ambos métodos(PCA y PLS-DA) realizamos dos pruebas, en la primera fijamos k en 20 y variamos α y γ respectivamente en el siguiente rango de valores:5-10-20-30-40-50-100-500. Procedimos a evaluar como esto influenciaba el tiempo de cómputo, el hit rate y las distintas métricas. Después nos enfocamos en fijar los valores de α y γ donde observamos buenos resultados y estudiamos como afectaba a los dos métodos variar el valor de k, el rango de valores utilizados fue el siguiente:1-5-10-20-30-40-50-100. Utilizamos una pc con las siguientes carácteristicas en un entorno controlado:

-Sistema Operativo: GNU/Linux Ubuntu
-CPU: Intel(R) Core(TM)2 Duo CPU T6500 @ 2.10GHz
-RAM: 3GB
-DISCO: 242GB

Hipótesis: En el caso de ambos métodos consideramos que al aumentar los valores de α y γ esto generará una mejora considerable respecto del hit rate, la precisión y las demás métricas lo que por otro lado implicará un aumento sustancial en el tiempo de cómputo. Es por esto que resaltaremos cuales son los parámetros que mejor se ajustan segun nuestro criterio para el sistema. En cuanto a la variación de k creemos que mientras se mantenga en valores bajos al ir aumentando su valor se observará una mejora en los resultados, sin embargo para valores demasiado grande esto puede no ser así. Las métricas utilizadas fueron precisión, y recall. Se optó tomar como valor estándar k = 20 para la primera prueba pues consideramos que era un valor medianamente razonable.

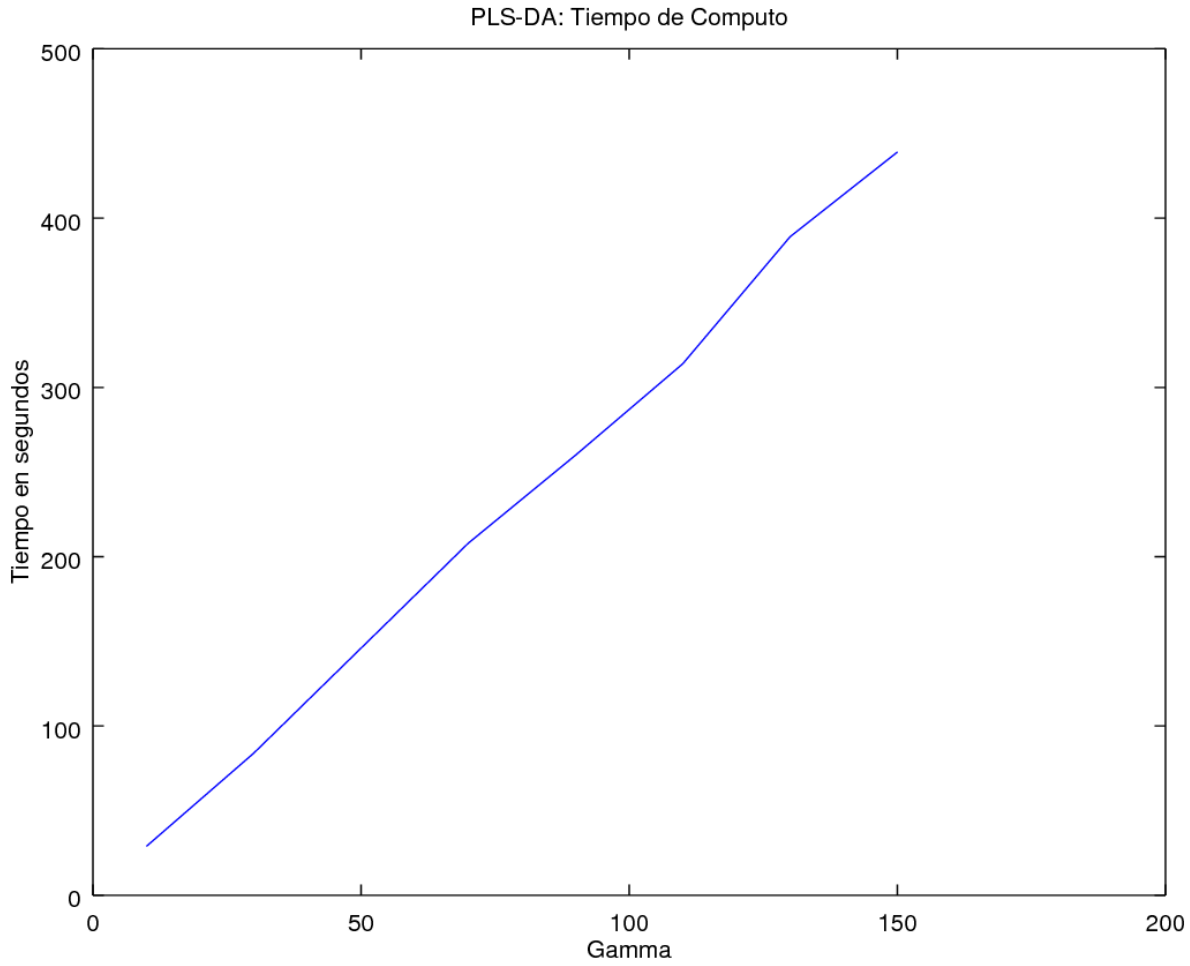
Resultados: PLS-DA

En el siguiente gráfico podemos ver el hit rate obtenido al dejar k fijo en 20 y variar γ:



Como podemos observar para valores chicos como 5 y 10 el hit rate se mantiene realtivamente bajo pero al ir aumentano se puede ver una notable mejora, ya que a partir de 20 siempre se supera el 90 % de efectividad alcanzando el 94 % en 40 y el 95 % en 100 y 500.

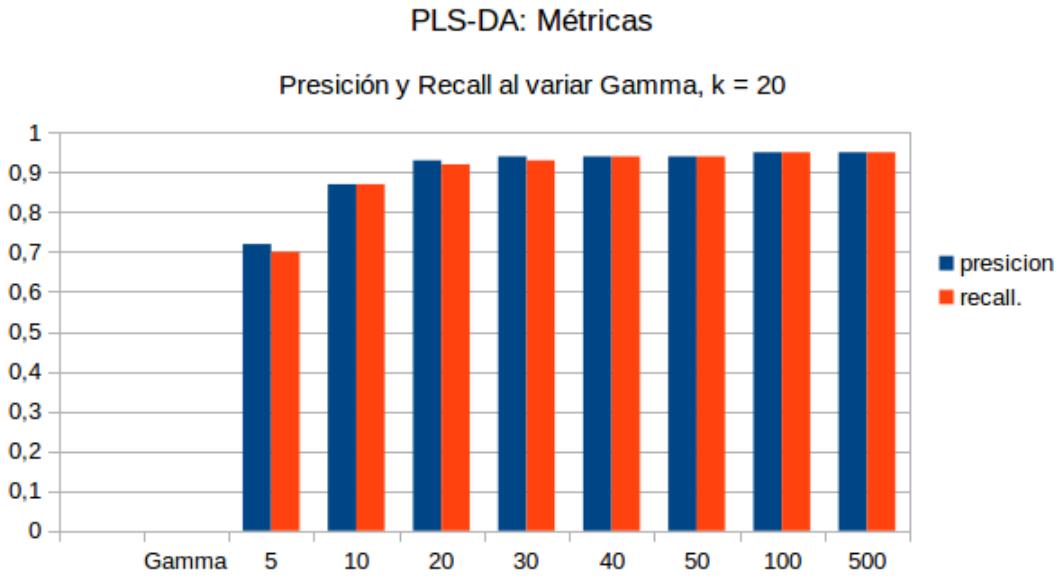
A continuación veremos como aumentar γ influye en el tiempo de cómputo del algoritmo:



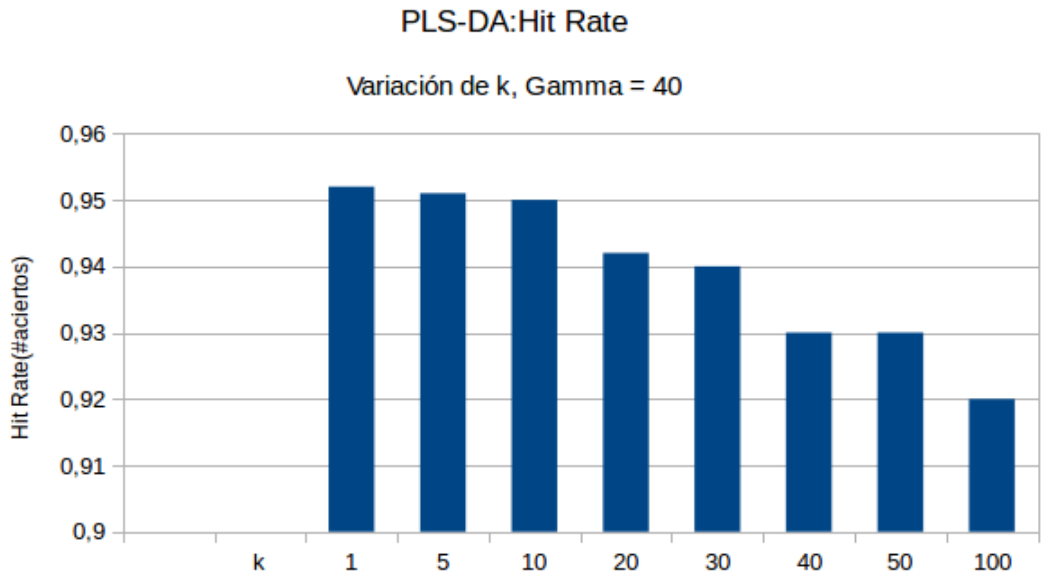
Se observa que para valores de γ entre 5 y 30 el tiempo no supera practicamente los dos minutos mientras que para valores intermedios como 40, 50 y 100 se necesitan entre 3 y 7 minutos para correr el programa, ya para valores mas extremos como 500 puede llegar a tardar 30 minutos o mas.

En el siguiente gráfico se muestran los resultados obtenidos para las métricas presición y recall al variar γ:

Ambas métricas resultan ser muy similares para todos los casos y además su relación con el hit rate es prácticamente 1 a 1.



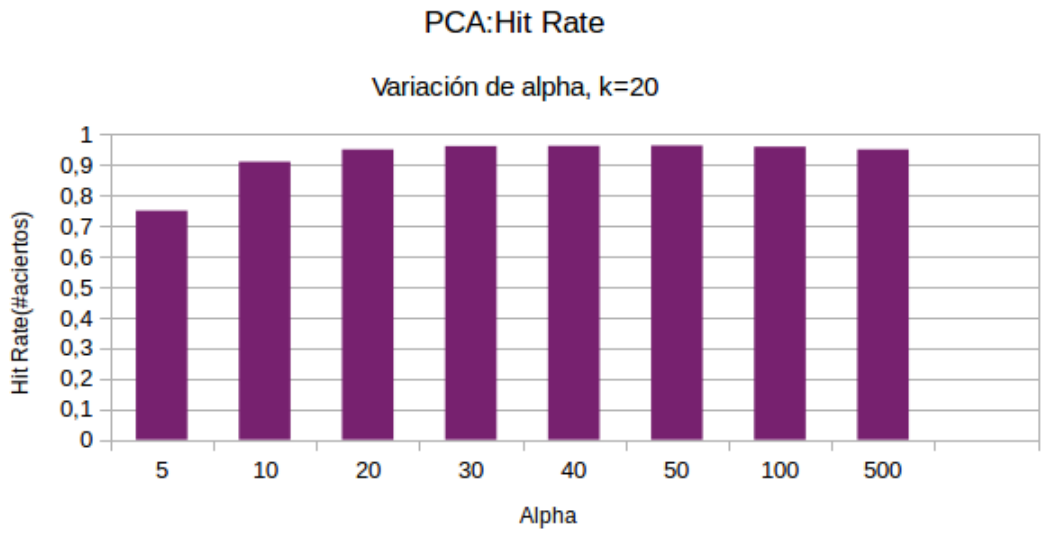
Por último tenemos el gráfico que representa la segunta parte de este experimento donde nos enfocamos en dejar fijo γ en 40 y variar k. Como los tiempos de cómputo resultaron prácticamente constantes en todos los casos nos limitamos a mostrar la efectividad mediante el hit rate, siendo los tiempos nunca mayores a los 4 minutos.



El mejor hit rate fue de 95 % con k = 1, luego para otros valores la diferencia era menor a 0.1 y por eso en el gráfico no llega a apreciarse, para valores mayores de k el hit rate baja hasta llegar a los 92 % con k = 100.

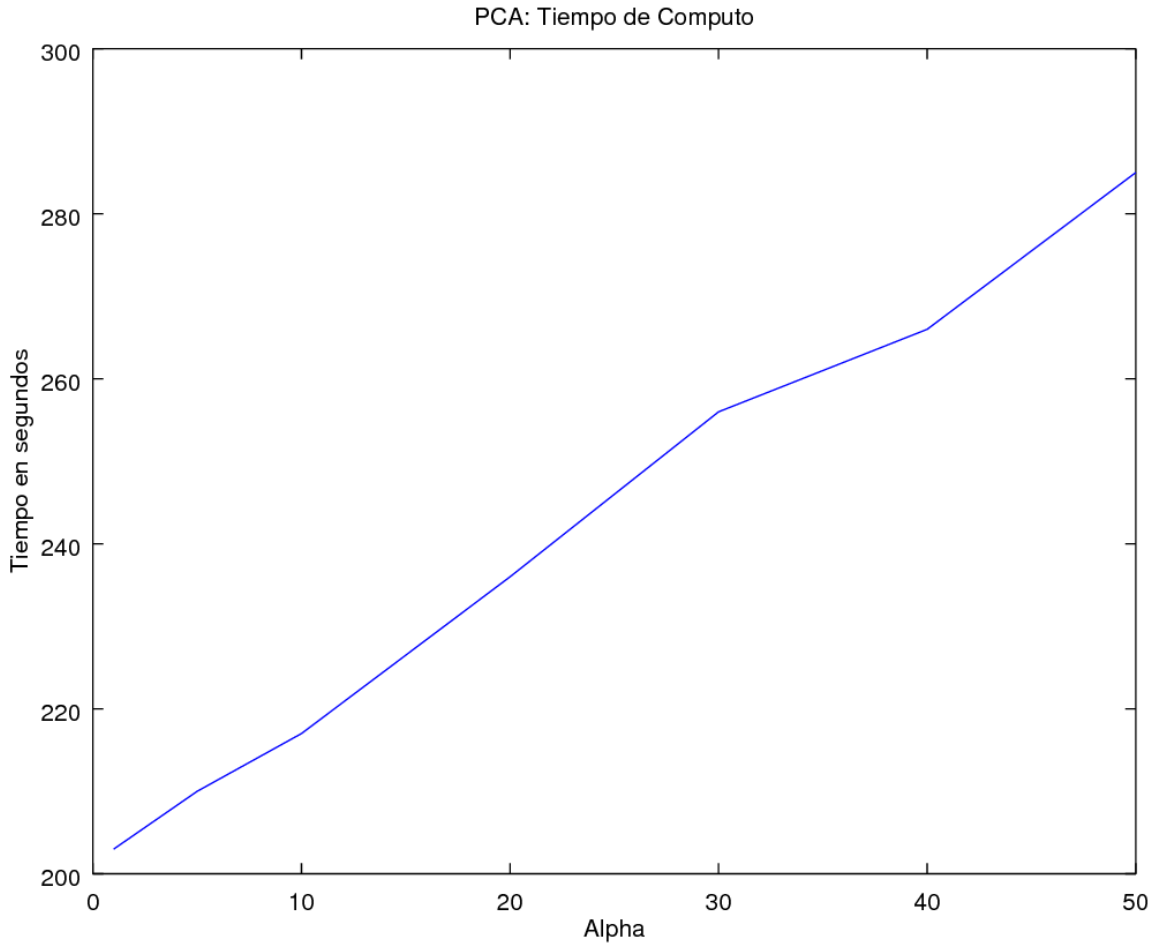
Resultados: PCA

En el siguiente gráfico podemos ver el hit rate obtenido al dejar k fijo en 20 y variar α:



Como podemos ver para valores chicos como 5 y 10 el hit rate se mantiene bajo, al ir aumentano se puede ver una notable mejora y a partir de 20 siempre se supera el 90 % de efectividad alcanzando el 96 % en 50 mientras que para valores muy grandes el hit rate baja nuevamente un poco en 100 y 500 tomando un valor de 95 %.

A continuación veremos como aumentar α influye en el tiempo de cómputo del algoritmo:

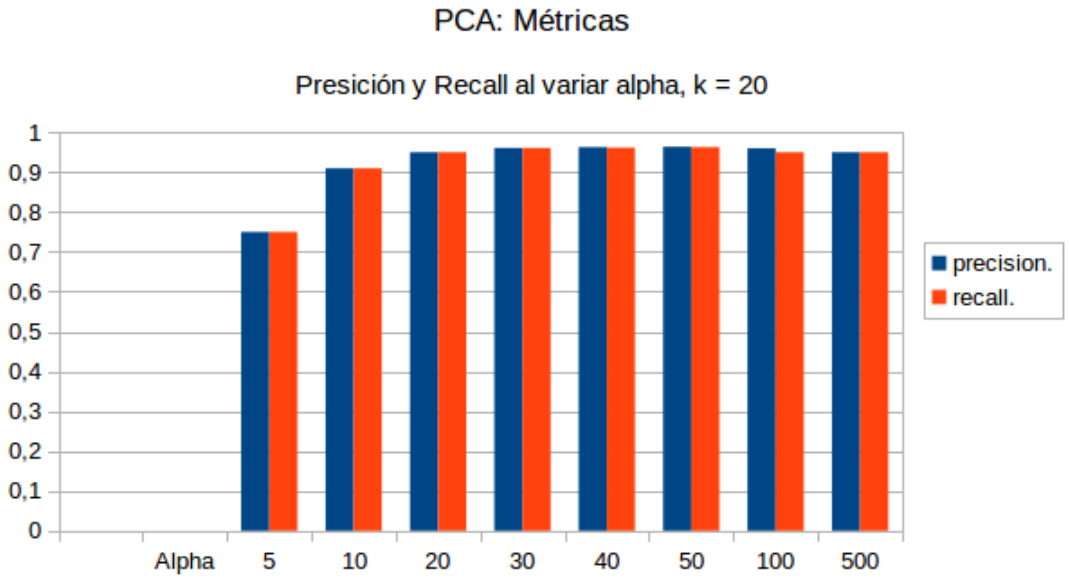


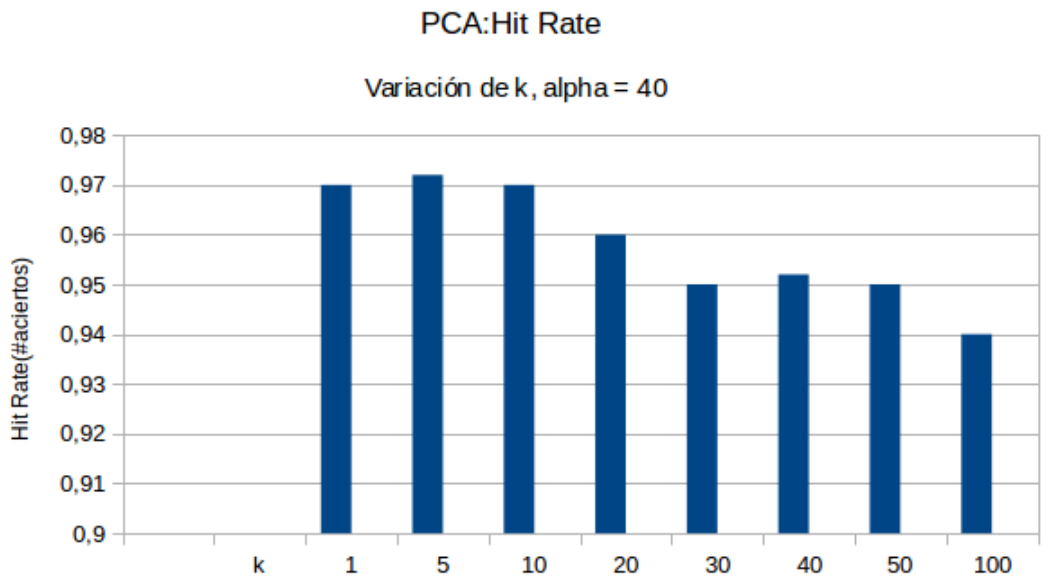
Como podemos ver los tiempos de cómputo resultan mucho mayores que en PLS-DA en casi todos los casos superando siempre los 15 minutos.

En el siguiente gráfico se muestran los resultados obtenidos para las métricas presición y recall al variar α:

Ambas métricas resultan ser muy similares para todos los casos y además su relación con el hit rate es prácticamente 1 a 1.

Por último tenemos el gráfico que representa la segunta parte de este experimento donde nos enfocamos en dejar fijo α en 40 y variar k. Como los tiempos de cómputo resultaron prácticamente constantes en todos los casos nos limitamos mostrar la efectividad mediante el hit rate.





El mejor hit rate fue de 97.2 % con k = 5, luego para los valores menores la diferencia era menor a 0.1, para valores mayores de k el hit rate baja hasta llegar a los 94 % con k = 100

Conclusión:

En ambos experimentos obtuvimos resultados similares como se puede observar, la principal conclusión es que con valores de α y γ relativamente medianos, entre 30 y 50 obtuvimos muy buenos porcentajes de acierto mientras que al aumentar mucho estos valores la ganancia resulta ser mínima y en ciertos casos empeora. Para valores chicos obtenemos hit rates muy bajos y por lo tanto no los recomendamos, en este sentido se confirma nuestra hipótesis del experimento pero por otro lado

observamos que al aumentar los k vecinos mas cercanos la ganancia fue mínima y los mejores resultados se obtuvieron con valores chicos como 1 o 5 lo cual resultó algo que no esperabamos ya que se esperaba que para valores mayores como 20, 30 o 40 la mejora fuese considerable.

Experimento 2: Calidad de resultados

Estudio variando la cantidad de imágenes de entrenamiento

Objetivos: Analizar la calidad de los resultados obtenidos al combinar PCA y PLS-DA con k-NN, para un rango amplio de cantidad de imágenes de entrenamiento. Considerar en el análisis también el tiempo de ejecución.

Hipotesis: La calidad de los resultados es directamente proporcional a la cantidad de imágenes de entrenamiento utilizadas. Es decir, a mayor cantidad de imágenes de entrenamiento, se obtendrá mejor calidad de resultados para ambos métodos.

Descripción : Se realizaron particiones con distintas cantidades de imágenes de entrenamiento mediante el algoritmo "generarParticion", implementado en el archivo utils.cpp. Las cantidades elegidas fueron 100, 420(1 %), 2100(5 %), 4200(10 %), 10500(25 %), 21000(50 %), 31500(75 %), 35700 (85 %), 37800(90 %), 41580(99 %), 42000(100 %) imágenes de entrenamiento. Esto permite abarcar todo el rango de la base de datos poniendo enfasis en los bordes, generando más particiones con mayor granularidad en estos. El archivo de entrada que se utilizó para el experimento tenía los parametros k=5, $\alpha = 50$, $\gamma = 50$, K = 11(para realizar el experimento en una corrida), seguidas de las 11 lineas correspondientes de las particiones antes mencionadas.

Este el algoritmo para generar la partición responde al siguiente pseudo-código:

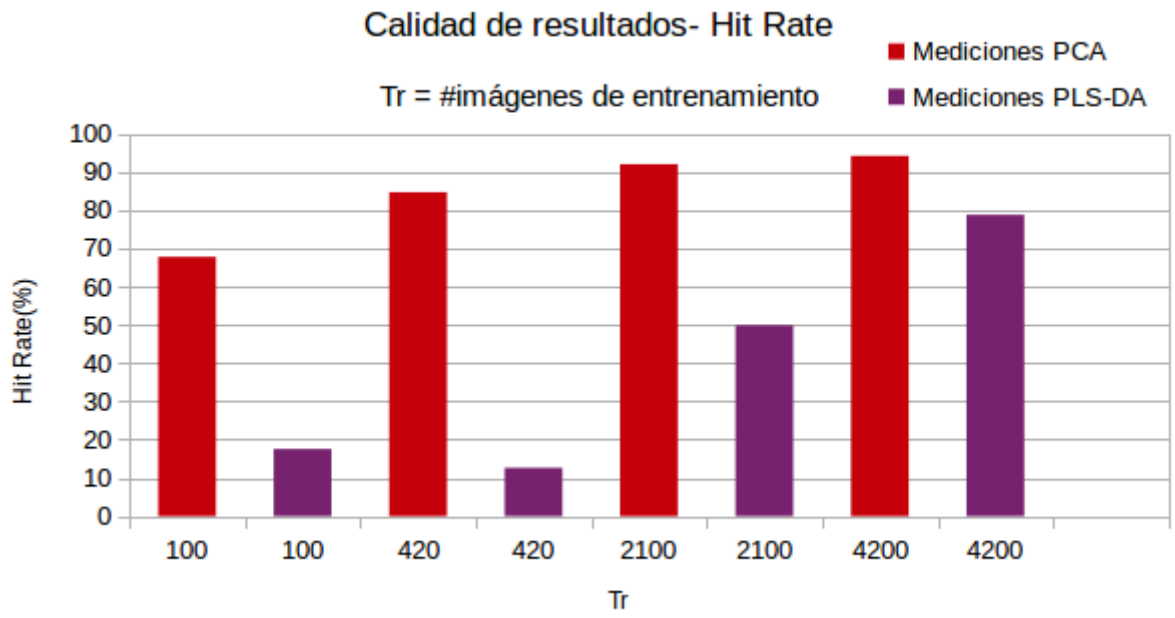
```
generarParticion(In cantTrain int, In archivoSalida string)
 1: particion ← crearVector(42000)
 2: if cantTrain<21000 then
 3:     for i = 1 . . . 42000 do
 4:         particion[i]← 0
 5:     end for
 6:     for i = 1 . . . cantTrain do do
 7:         pos ← random(0..42000)
 8:         while particion[pos]==1 do
 9:             pos ← random(0..42000)
10:         end while
11:         particion[pos] ← 1
12:     end for
13: else
14:     for i = 1 . . . 42000 do
15:         particion[i]← 1
16:     end for
17:     for i = 1 . . . 42000-cantTrain do do
18:         pos ← random(0..42000)
19:         while particion[pos]==0 do
20:             pos ← random(0..42000)
21:         end while
22:         particion[pos] ← 0
23:     end for
24: end if
25: escribir en archivoSalida el vector particion con espacios entre componentes.
    Fin generarParticion
```

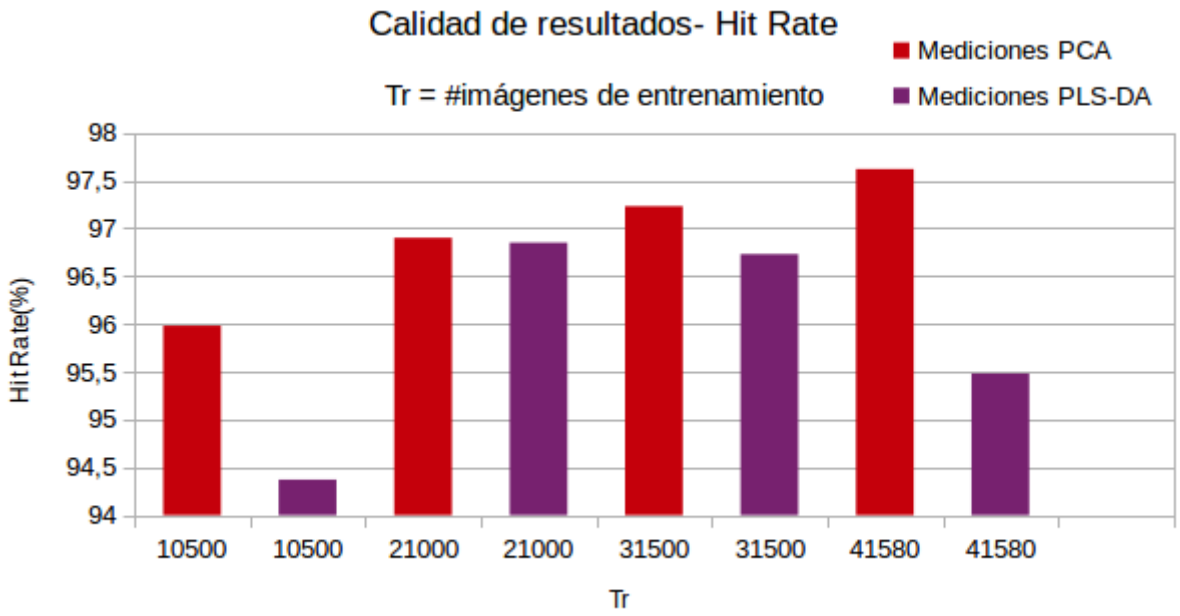
Nota: Utilizamos random porque sólo usaríamos una partición para cada cantidad de imágenes de entrenamiento.

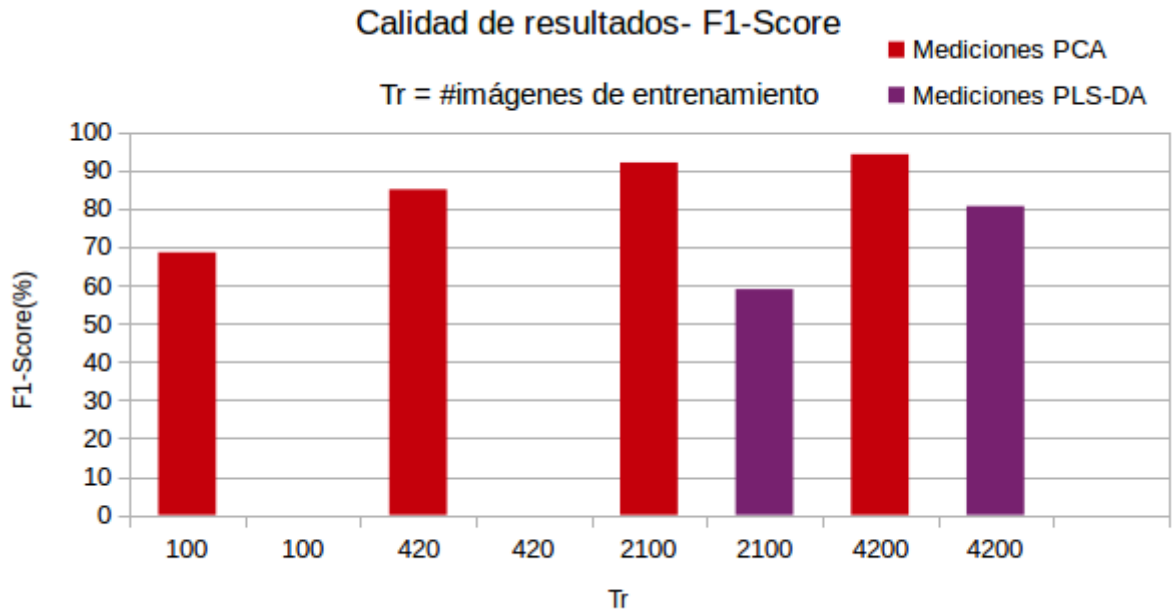
El experimento fue realizado en una computadora con las siguientes especificaciones:

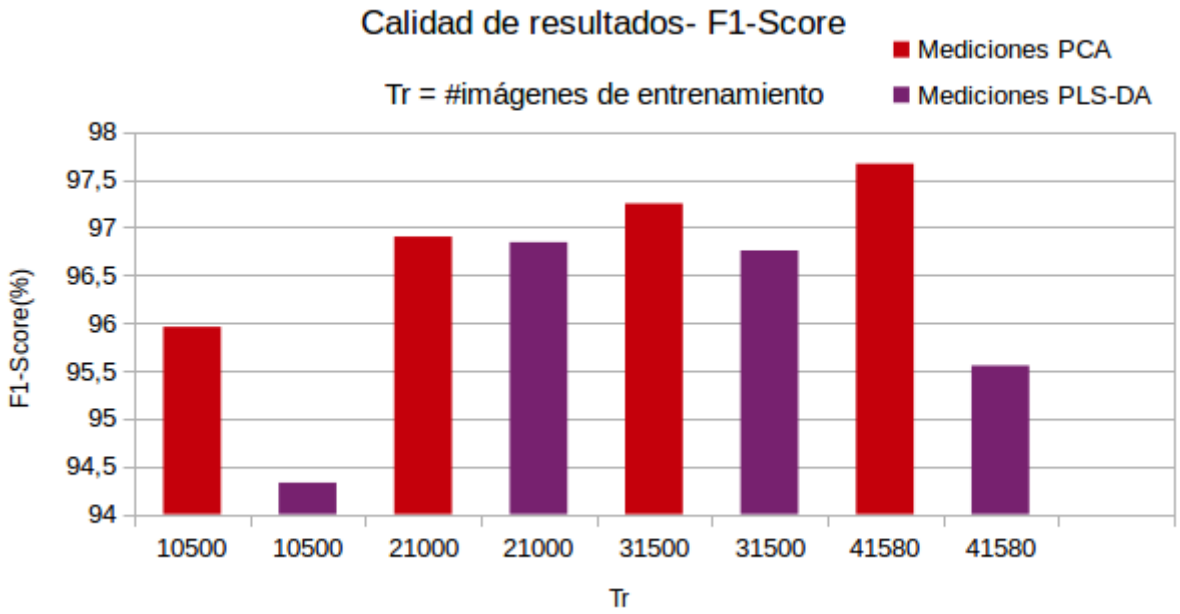
-Sistema Operativo: GNU/Linux Ubuntu
-Procesador: AMD Phenom x4 840 CPU @ 3.00GHz
-Memoria: 4GB DIMM DDR3 1333 MHz
-Disco: 500 Western Digital caviar blue, 7200 rpm, SATA 2.0, 3.0 Gb/s

Resultados y conclusiones: Los resultados totales se encuentran en la carpeta Experimentos. A continuación expresamos un resumen de lo más relevante.

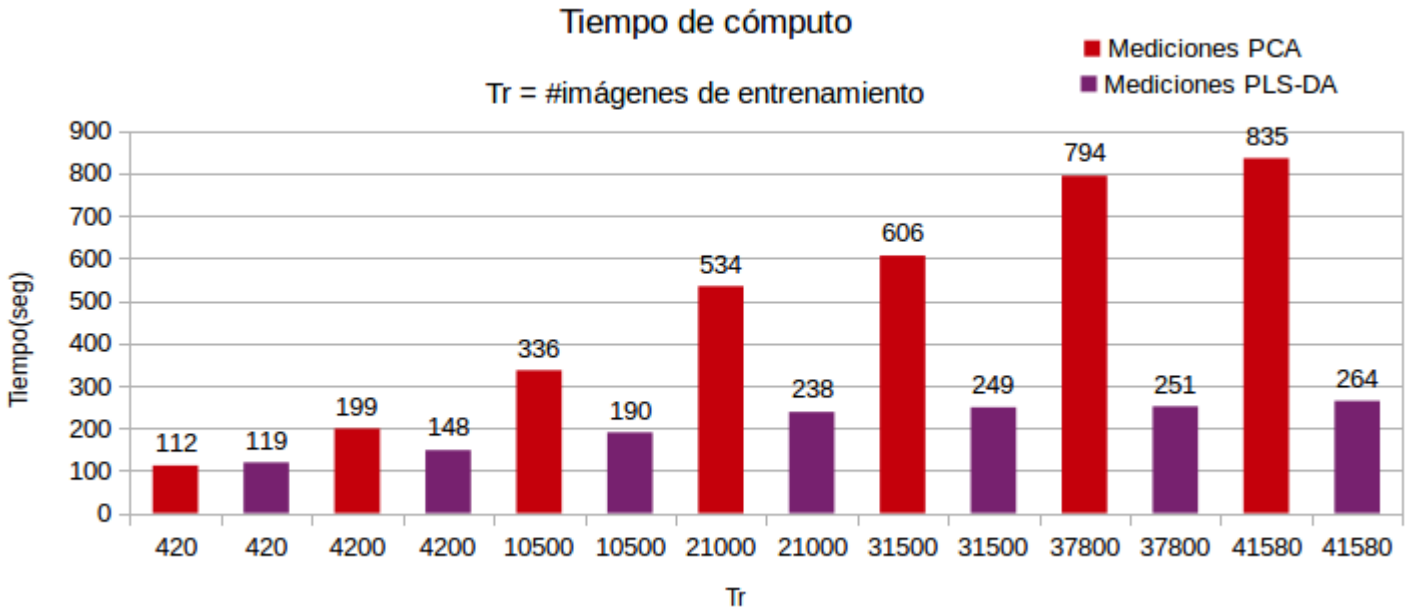


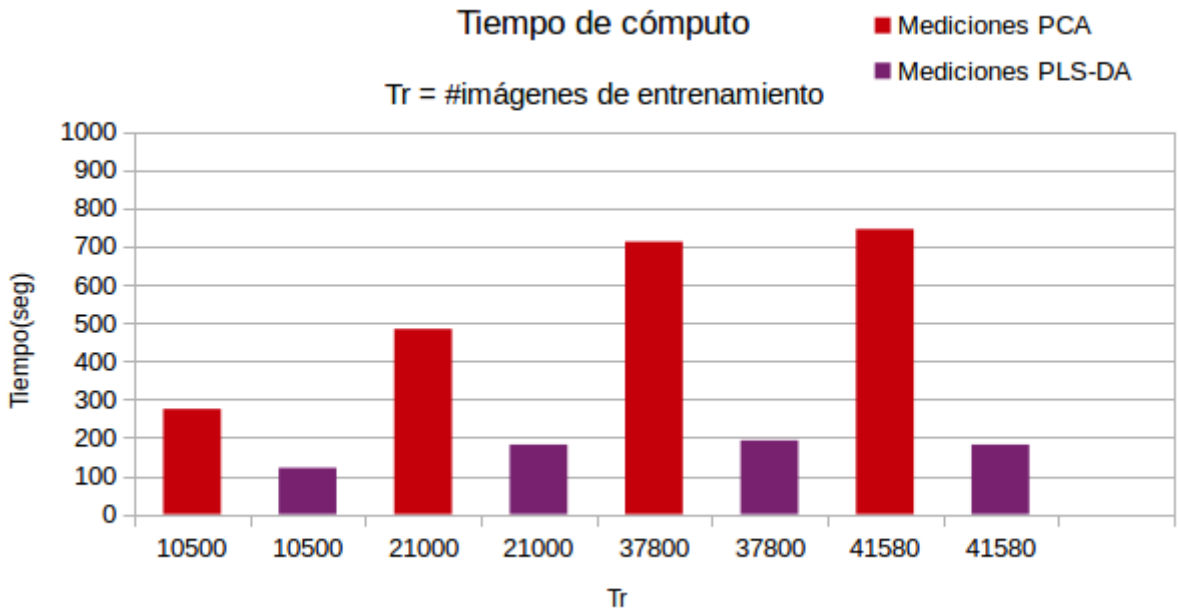


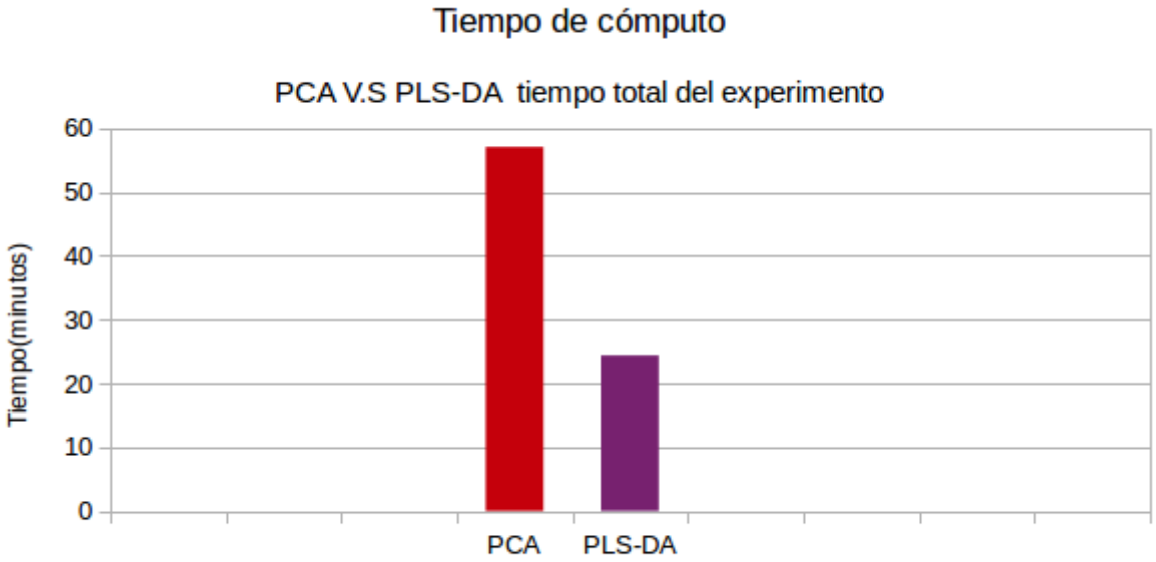




En cuanto a la calidad de los resultados, podemos decir que PCA+ kNN responde mucho mejor en cuanto a hit rate que PLS-DA, ante escenarios donde la cantidad de imágenes de entrenamiento es reducida. Además, el hit rate en PCA demostró crecer de manera proporcional a la cantidad de imágenes de entrenamiento, mientras que en PLS-DA se alcanzó un pico en un punto de 21000 imágenes y luego comenzó a bajar nuevamente. En cuanto al F1-Score, el cuál involucra al recall y precisión de los métodos, podemos realizar la misma observación que con el hit rate.


Tiempo de cómputo
Tr = #imágenes de entrenamiento
Mediciones PCA
Mediciones PLS-DA
Tiempo(seg)
900
800
700
600
500
400
300
200
100
0
112
119
199
148
336
190
534
238
606
249
794
251
835
264
420
420
4200
4200
10500
10500
21000
21000
31500
31500
37800
37800
41580
41580
Tr


Tiempo de cómputo
Tr = #imágenes de entrenamiento
Mediciones PCA
Mediciones PLS-DA
Tiempo(seg)
1000
900
800
700
600
500
400
300
200
100
0
10500
10500
21000
21000
37800
37800
41580
41580
Tr



En referencia al tiempo de cómputo, inducimos que la eficiencia de PLS-DA por encima de PCA puede deberse a la cantidad de operaciones requeridas para computar la matriz de covarianza. Esta en PCA se calcula realizando el producto matricial X^tX con $X \in \mathbb{R}^{nxm}$, por lo que los productos internos para calcular la posición i,j de la matriz de covarianza iteran hasta n. Mientras que en PLS-DA es calculada realizando $(A)^t(A)$ donde $A = Y^tX$ y $A \in \mathbb{R}^{10xm}$, es decir, los productos internos iteran hasta la constante 10. Esto se analizará un poco más en detalle en la conclusión del experimento 5.

Conclusión final: PCA es más eficiente en términos de calidad de resultados, en especial en contextos de pocas imágenes de entrenamiento. Sin embargo se debe considerar el factor tiempo de cómputo si no se cuenta con la matriz de transformación preprocesada y la cantidad de imágenes de entrenamiento es grande, ya que PLS-DA resultó ser mucho más veloz en estos casos.

Experimento 3: Estudio de particiones

Objetivos: Realizar los experimentos 1 y 2 para al menos dos valores distintos de K. Esto conduce a un estudio más fiable ya que considera una variación en la cantidad de dígitos de cada clase, cambiante en cada partición.

Descripción: Procedimos a realizar el experimento 1 con 3 valores de K, estos fueron 1, 5 y 10. Estos fueron elegidos junto a una cantidad de entrenamiento de 90 % de imágenes, luego 10 sería el límite de K folds para variar el 10 % de test, además de esta manera podríamos aprovechar las particiones de los test1 y test2 otorgadas por la cátedra. Para esto realizamos 3 corridas para los valores de k y α en PCA y para k y γ en PLS-DA. Por cada una de ellas se calculó el promedio de los hit rates obtenidos para cada k/alpha/gamma entre las particiones según el K actual, luego se retornó como parámetro óptimo aquel que tuviese mayor promedio entre los distintos valores de k/alpha/gamma. Para un mayor entendimiento dejamos los main.cpp con los que realizamos el experimento en la carpeta Experimento 3.
En cuanto al experimento 2, decidimos utilizar 4 porcentajes de cantidad de imagenes de entrenamiento, estos son 50, 75, 90 y 99. Decidimos empezar en 50 ya que con menos no tendría sentido hacer más de 2 K folds, ya que se solaparía el conjunto de test. Para cada uno de estos porcentajes contamos con los resultados del experimento 2 para K=1. Luego pensamos en usar el K óptimo para cada porcentaje para comparar con este otro resultado. Este K óptimo sería igual a 100/(100-p). Donde p es el porcentaje a tomar como train. Luego al finalizar el experimento tendríamos para:
50 %, K = 1 y K = 2
75 %, K = 1 y K = 4
90 %, K = 1 y K = 10
99 %, K = 1 y K = 10
En el caso de 99 % decidimos cambiar el valor 100/(100-p) por 10, ya que 100 implicaría un tiempo de cómputo inaceptable.
Los K folds fueron construidos con la función KCrossValidation, implementada en utils.cpp . Realizamos un shuffle previo de la base de datos para asegurar que no hubiese ningún patrón de clases. Dejamos también el main utilizado para automatizar las mediciones del experimento para realizar todo en una sola corrida, en el achivo *mainVariaCantTrain.cpp* que se encuentra en la carpeta Experimento 3.

Hipótesis: Consideramos que al utilizar una mayor cantidad de particiones es posible que obtengamos un resultado diferente al de los experimentos anteriores, probablemente un resultado mas aproximado

Resultados de experimento "1"**:**

A continuación presentamos una tabla con los valores óptimos del experimento 1 obtenidos para PLS-DA sobre las distintas particiones:

Como vemos en la tabla al utilizar más particiones obtuvimos otros valores óptimos sin embargo

K	Gamma óptimo	k óptimo
1	100	1
5	500	5
10	500	5

muy cercanos al primer experimento.

Luego, adjuntamos a continuación la tabla representativa para PCA con sus parámetros óptimos del experimento 1:

K	Alpha óptimo	k óptimo
1	50	5
5	50	5
10	50	5

Se observa que para todas las particiones los valores óptimos se mantuvieron constantes y generaron los mismos resultados que en el experimento 1

Nota: Nos reservamos a poner la tabla con los valores que representan los mejores parámetros, para mayor información consultar la sección 'Experimentos', el tercero corresponde a este y allí se pueden encontrar todos los valores obtenidos para cada corrida del experimento.

Resultados de experimento "2"**:**

HIT RATE(%)								
	PCA				PLS-DA			
	K				K			
Porcentajes	1	2	4	10	1	2	4	10
50	96,9	96,9	* * *	* * *	96,8	96,7	* * *	* * *
75	97,2	* * *	97,4	* * *	96,7	* * *	96,9	* * *
90	97	* * *	* * *	97,5	95,9	* * *	* * *	96,9
99	97,6	* * *	* * *	97,4	95,4	* * *	* * *	96,4

Conclusión: Como conclusión final de este experimento podemos sugerir como párametros óptimos para utilizar PCA $\alpha = 50$ y k = 5, con una cantidad de imágenes de entrenamiento lo mayor posible. Mientras que para PLS-DA si bien obtuvimos mejores resultados con $\gamma = 500$ este resulta casi idéntico a utilizar $\gamma = 100$, superándolo por menos de un decimal en porcentaje, por lo que consideramos que si el tiempo de cómputo es algo importante en la utilización del algoritmo lo mejor es utilizar $\gamma = 100$ y k = 5 con una cantidad de imágenes de entrenamiento cercana a 31500(en nuestro caso 75 % de la base). Por últomo respecto a los resultados del experimento 2 podemos concluir que al aumentar la cantidad de imágenes de entrenamiento en PCA obtuvimos una mejora respecto del hit rate mientras que en PLS-DA esto no fue necesariamente así ya que si vemos el ejemplo mas claro en el que se usa una sola partición, los portencajes del hit rate tienden a bajar asi también como ocurre con 10 particiones.

Experimento 4: Optimalidad

Objetivos: Comparar los 2 métodos con entradas que consideramos óptimas para cada uno.

Hipotesis: PCA obtendrá mayor calidad de resultados, mientras que PLS-DA predominará en cuanto a tiempo de cómputo.

Descripción: En base al experimento anterior, elegimos para PCA los parametros k = 5 y α=50. Utilizaremos 90 % de la base de datos como entrenamiento, realizando K fold cross-validation con K=10. Aunque con 99 % el experimento anterior haya dado un poco mejor de hit rate, cabe considerar que solo habría 420 imagenes para testear en este caso, por lo que el error relativo del hit rate podría ser mayor que al utilizar una cantidad mayor como lo es 4200(10 %).
Para PLS-DA los parámetros serán k = 5 y γ = 100, y se utilizara el 75 % de la base de datos como train realizando K cross-validation con K = 4, ya que este fue el óptimo obtenido en el experimento anterior.

Resultados:

Conclusión:

Experimento 5: Optimización del tiempo de cómputo

Aproximación de la matriz de covarianza en PCA

Objetivos: Probar que la matriz de covarianza en PCA puede ser aproximada de forma muy eficaz considerando muchas menos imagenes que "n". Resultando en un hit rate casi idéntico y sin embargo ahorrando un considerable tiempo de cómputo.

Hipotesis: Aproximar la matriz de covarianza con una cantidad de imagenes cercana a 1000 dará una aproximación eficaz, resultará en un hit rate muy similar al que se obtendría utilizando n imagenes, y sin embargo disminuirá considerablemente el tiempo de cómputo utilizado para crear la estructura.

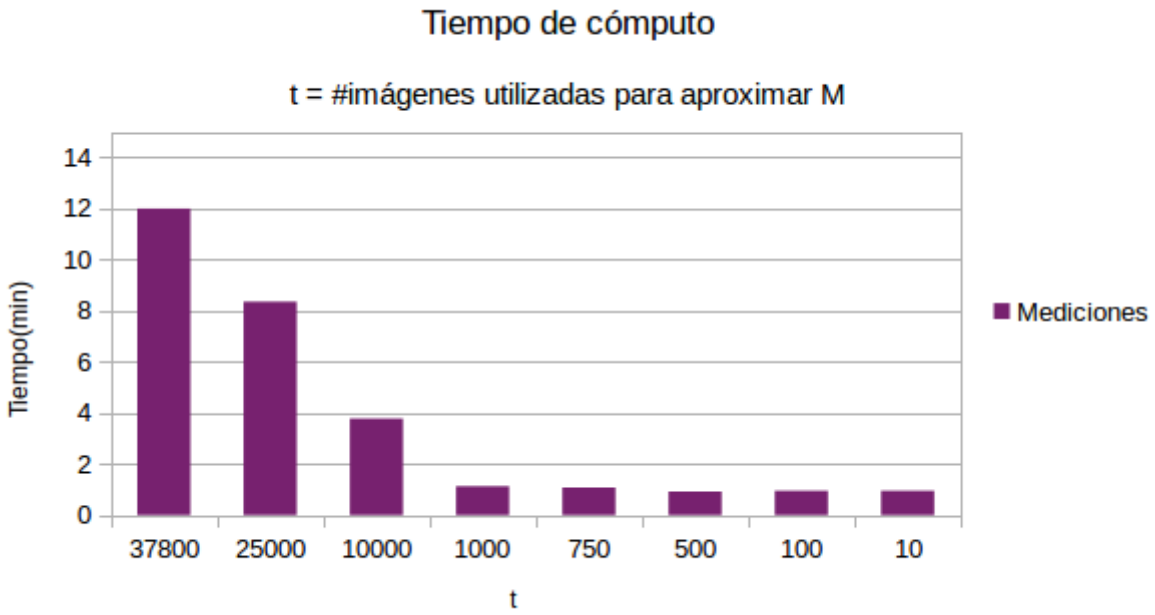
Descripción: En el algoritmo que crea la estructura de PCA, se calcula la matriz de covarianza utilizando la función prodTraspuesta, que toma por parámetro una matriz X, calcula el producto X^tX y lo deja en la matriz M(también parámetro). Si se mira el código de la función, se ve que el for interno que tiene como variable k, debería iterar hasta X.size(), es decir, tal que $M_{ij} = col_i * col_j$. Esto se traduce a utilizar n = 37800 (en el caso de test1.in) componentes i,j para aproximar la covarianza entre las variables i,j (es decir, esos pixeles). En este experimento se estudia que ocurre si en vez de ciclar hasta n se cicla hasta un k mucho menor, es decir, se aproxima la covarianza entre las componentes i,j con menos imagenes que n, con el objetivo de reducir el tiempo de cómputo. Notar que esto no afecta la cantidad de imágenes a utilizar como train, ya que la transformación será luego aplicada a todas ellas.

Resultados:

Se observaron los siguientes resultados, siendo k = # imágenes utilizadas para aproximar la matriz de covarianza.

Podemos observar en la tabla que el mejor hit rate ni si quiera se obtuvo con un gran k, es decir, bastó tomar 500 imagenes para aproximar la matriz de covarianza para alcanzar un hit rate de 0.9726. Por otro lado podemos ver que de 37800 a 500 imágenes el tiempo de cómputo es considerablemente menor para una simple partición(el experimento se hizo para un único K fold). Con 37800 imágenes requirió 12 minutos, mientras que para 500 imágenes ni si quiera 1. En general, se tiene que utilizar 500 imágenes resulta ser un 1204 % más rápido que utilizar 37800.

Por otro lado, una vez que fijamos la cantidad de imágenes para aproximar la matriz de covarianza en 500. Se realizó un experimento que consistió en aumentar la cantidad de imágenes de entrenamiento de las particiones, con los mismos valores que se utilizaron en el experimento 2(con alpha fijo = 50). Con la idea de que al saturar a 500 la cantidad de imágenes en una función central del algoritmo para crear la estructura, los tiempos de cómputo tendrían un pico cuando el tiempo requerido por k-NN para predecir la cantidad de imágenes a testear, superara el tiempo implicado



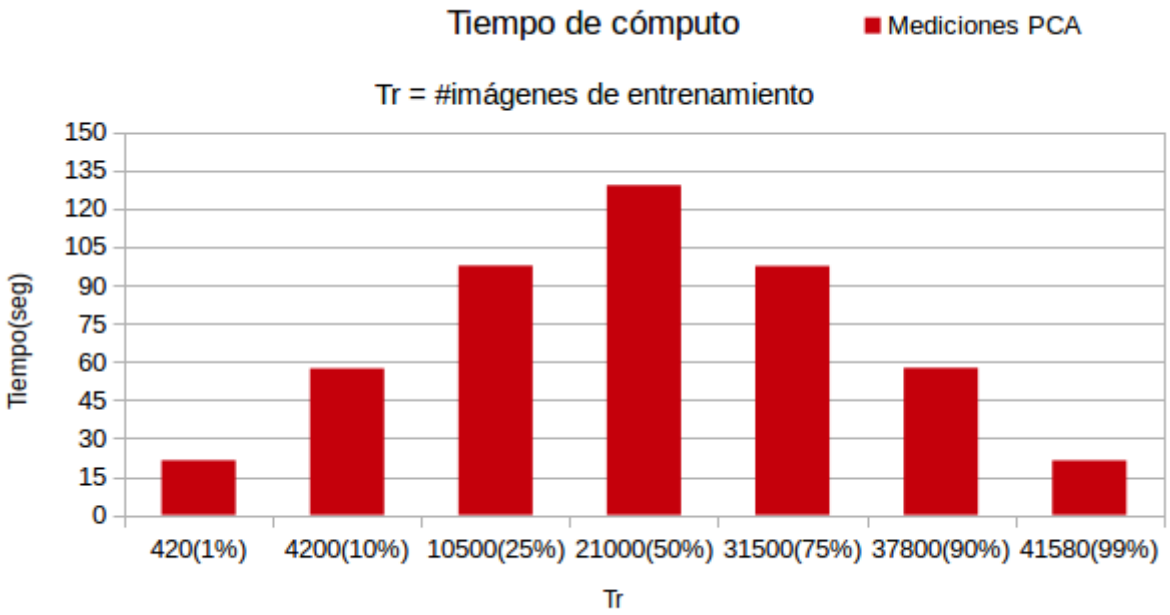
Resultados para alpha = 50								
t (#img)	37800	25000	10000	1000	750	500	100	10
Hit Rate	0,971	0,9702	0,9712	0,9702	0,9705	0,9726	0,9704	0,911
Tiempo(s)	721,1	501,4	226,7	68,1	64,7	55,4	56,7	56,7
Tiempo(min)	12	8,36	3,78	1,13	1,08	0,92	0,95	0,95

para generar la estructura. En efecto, se obtuvieron los siguientes resultados.

-Sistema Operativo: GNU/Linux Ubuntu
-Procesador: AMD Phenom x4 840 CPU @ 3.00GHz
-Memoria: 4GB DIMM DDR3 1333 MHz
-Disco: 500 Western Digital caviar blue, 7200 rpm, SATA 2.0, 3.0 Gb/s

Conclusión:

Se avaló nuestra hipótesis, en ciertos casos conviene evaluar el error que se obtiene al aproximar por una muestra menor, si esto implica una mejora considerable en tiempo de cómputo. Analizando a la par la calidad de los resultados. A su vez, se observó que al realizar esto, se obtenía un pico en el tiempo de cómputo, para una cantidad de imágenes de test igual al 50 % de la base de datos. Con este resultado, además se puede tener un mayor entendimiento de por qué PLS-DA resultó más rápido a mayor cantidad de imágenes de entrenamiento, en el experimento 2. Ya que la matriz de covarianza en PLS-DA se calcula realizando $(A)^t(A)$ donde $A = Y^tX$ y $A \in \mathbb{R}^{10xm}$. Recordando que este producto matricial realiza para cada posición i,j el producto interno entre la columna i y la



columna j de A, se tiene que, como las columnas tienen solo 10 componentes, esto sería el equivalente a en PCA saturar el k a 10. Por lo que PLS-DA es muchisimo más eficiente en términos de tiempo de cómputo en este punto.

Experimento 6: Overfitting

Evasión en probabilidad del sobreajuste

Objetivos: Mejorar el Hit Rate en PCA utilizando una gran muestra de particiones aleatorias.

Hipotesis: La calidad de los resultados puede aumentar si se considera una aproximación estadística a los parámetros óptimos, consecuente de tomar distintas particiones aleatorias para una gran cantidad de imágenes de entrenamiento.

Descripción: Utilizaremos PCA con la optimización descripta en el experimento anterior, para reducir considerablemente el tiempo de cómputo. Ya que incluso así la idea de este experimento requerirá uno considerable. Se toma una base de entrenamiento de 40000 imágenes. Se realizan 100 particiones diferentes por medio de la función generarParticiones, implementada en utils.cpp. Para cada partición se varian los parámetros de entrada como se hizo en el experimento 1, primero fijando α y γ, y luego fijando k. Para cada instancia, por ejemplo, para la que varía el k de vecinos más cercanos, se realizan las 100 corridas para el i-esimo valor de k a estudiar y se promedian los hit rates, tomando ese promedio como el correspondiente al k_i. Finalmente se escoge como k óptimo al k tal que su correspondiente k_i sea el máximo, es decir, al de mayor hit rate en promedio. Para la instancia de optimizar el alpha se realiza un procedimiento análogo. La idea reside en utilizar un gran número de particiones distintas para que el procentaje de cada clase sea variable.

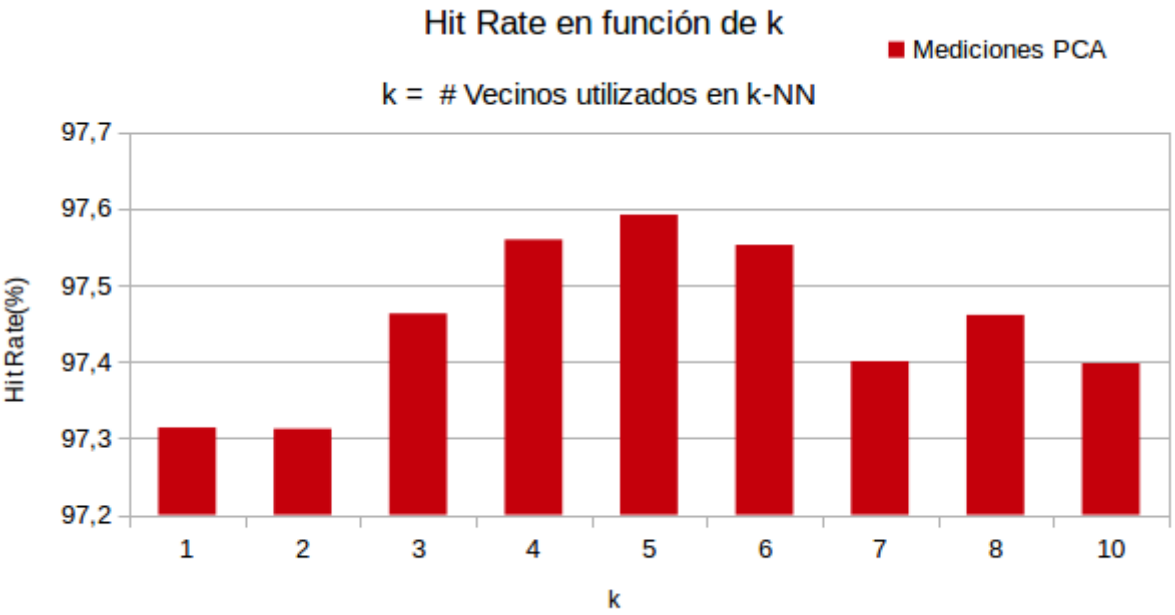
Instancia 1: Se busca el k óptimo mediante el método mencionado en la descripción. Se utiliza la siguiente sucesión para valores de k.

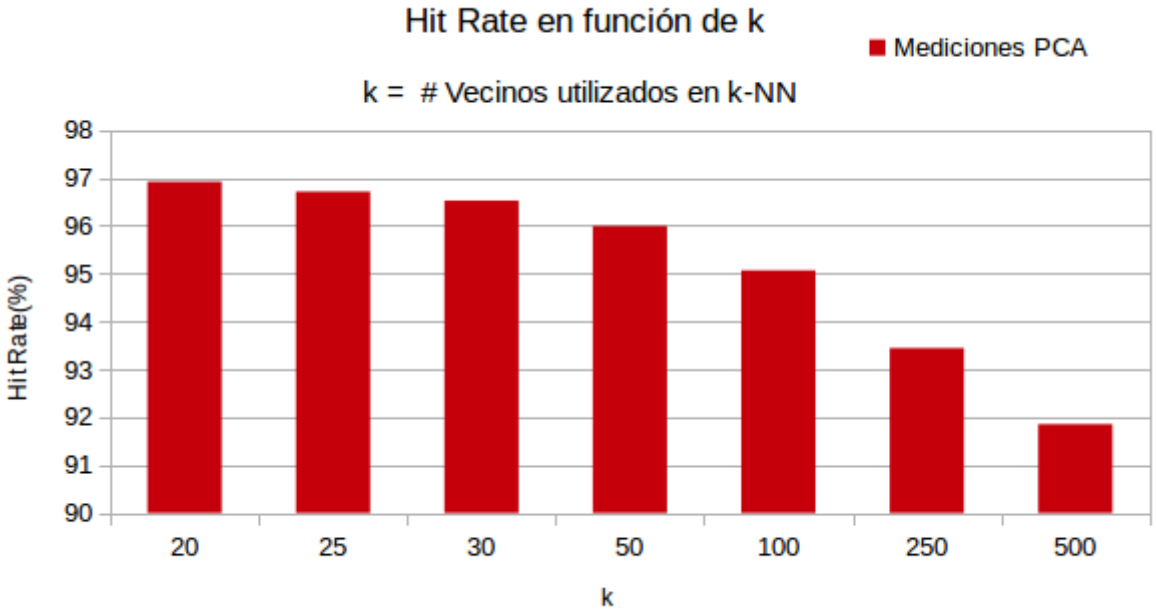
k = 1, 2, 5, 10, 20, 25, 30, 50, 100, 250, 500.

Se fija $\alpha = 40$.

Resultados de la Instancia 1:

El tiempo de corrida 1 del experimento con el PCA óptimizado del experimento 5, fue de 26196.5 segundos, es decir, aproximadamente 7 horas, 27 minutos, 40 segundos(si no hubiesemos usado la versión del experimento 5 hubiese tardado 3 dias y medio según los resultados de rendimiento del expermento 5). El k óptimo resultó ser el mismo que en obtenido en el experimento 1, y 3. k = 5. Luego se realizó una segunda corrida con los valores k = 3, 4, 6, 7, y 8. El óptimo de esta tanda fue el 4, pero siguió siendo inferior a k = 5. Luego el resultado final es que el k óptimo es tomar 5 vecinos. Los resultados completos están en la carpeta del Experimento 6.



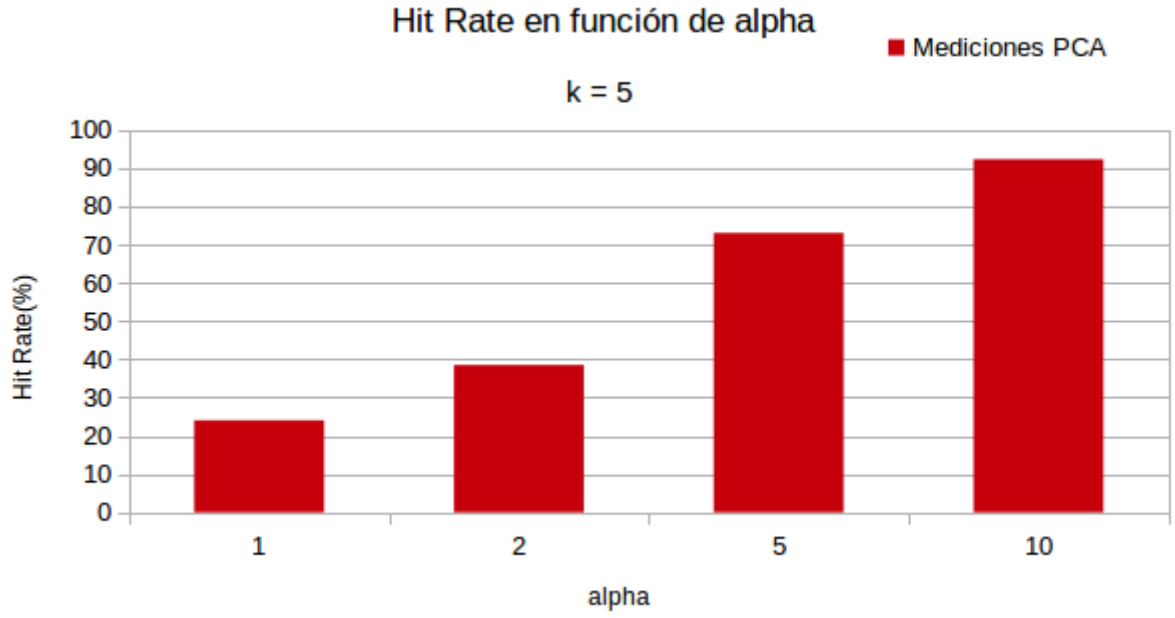


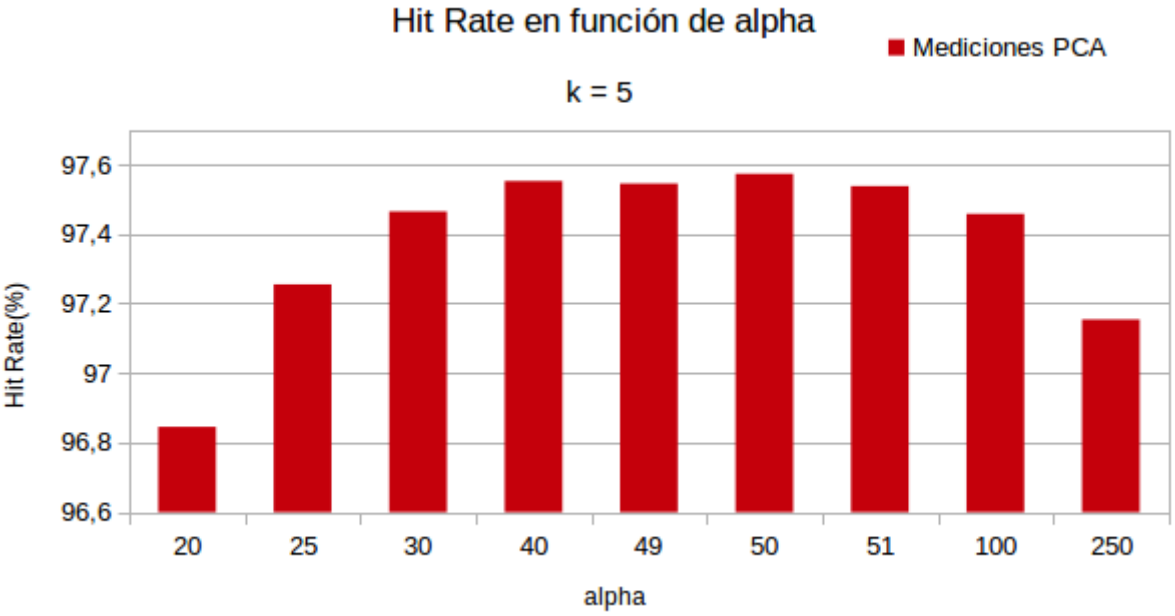
k óptimo entre corrida 1 y 2: 5

Instancia 2: Se busca el α óptimo mediante el método mencionado en la descripción. Se utiliza la siguiente sucesión para valores de α y se fija el k en 5, óptimo obtenido en la instancia anterior.

α = 1, 2, 5, 10, 20, 25, 30, 40, 50, 100, 250.

Resultados de la Instancia 2: El alpha óptimo resultó ser 50. Se realizó una segunda corrida para ver si α= 49 ó 51 derivaban en un mejor hit rate. Pero no fue así. Luego concluimos que la configuración óptima para PCA, apoyandonos tambíen en los resultados del experimento 3, es k = 5 y α= 50.





Conclusión: Pensamos que tomar una gran cantidad de particiones aleatorias conduciría a nuevos parámetros óptimos, distintos de los del experimento 3, sin embargo, parece que la técnica de K-crossvalidation utilizada anteriormente es lo suficientemente efectiva como para equiparar a las 100 particiones aleatorias. Por otro lado fue interesante considerar el tiempo de cómputo que requiere un experimento de este carácter y saber que de no ser por el experimento 5 y la utilización de una óptimización a nivel tiempo de cómputo, no se podría haber llevado a cabo.

Resultados en Kaggle

Los resultados obtenidos al enviar nuestra predicción a Kaggle en la competencia, fueron de 0.97443 para PCA con $\alpha = 50$ y $k = 5$. También mandamos una predicción utilizando PLS, con $\gamma = 100$ y $k = 5$, obteniendo 0.97300 de puntuación.

Conclusiones finales

Para concluir el informe, podemos englobar las conclusiones de los experimentos mencionando los siguientes puntos.

- Aumentar desmedidamente el k en el método de vecino más cercano es contraproducente. Conviene asignar peso a cada distancia para la votación y utilizar k pequeños como 5, 3 o incluso 1.

- PCA parece ser un método de preprocesamiento un poco más eficaz que PLS-DA en cuanto a calidad de resultados.

- PLS-DA es más eficiente en cuanto a tiempo de cómputo que PCA si se tiene un gran conjunto de train y/o se considera el preprocesamiento , es decir, si se calcula la estructura de transformación de manera online.

- Resulta muy interesante el hecho descubierto en el experimento 5, de poder aproximar de forma tan eficiente la matriz de covarianza con muchas menos imágenes. Lo que reduce significativamente el tiempo de cómputo sin afectar prácticamente la calidad.

- La técnica de K-fold Cross-Validation, es más eficaz que un gran número de particiones aleatorias.
- Ante la posibilidad, siempre será preferible y más fiable, realizar K-fold Cross-Validation, para disminuir el error relativo de las mediciones y no obtener conjuntos de train y/o test desbalanceados.

- Existe un óptimo de cómponentes principales que caracterizan a un objeto de estudio. No es conveniente trabajar con todos los factores ya que cada objeto tiene sus particularidades, que pueden no responder a las de una especie.

Optimizaciones y puntos opcionales:

En cuanto a lo concerniente al punto opcional, implementamos una versión de k-NN lo más eficiente que pudimos en el archivo knn.cpp. Pensamos en que, en vez de minimizar las distancias entre vectores podríamos minimizar las distancias al cuadrado, ya que la función raiz cuadrada es monótona creciente. De esta forma nos ahorrariamos calcular esta última dentro del algoritmo, para cada vector. Por otro lado, consideramos las imágenes como vectores de unsigned char con el fin de ahorrar memoria(ya que valores de 0 a 255 entran en un byte provisto por este tipo). Para calcular la distancia al cuadrado de un vector de unsigned char(una imagen) decidimos guardar la sumatoria en una variable de tipo unsigned int. Está tiene un rango de representación suficientemente alto para lidiar con la máxima sumatoria posible, que sería igual a $((255-0)^2) * m$ (sabiendo que m es menor o igual a 42000 en nuestro contexto). Con esto los componentes del vector debian ser solo casteados a int y no a double, y de este modo, el procesador utilizaría aritmética de punto fijo y a la vez ahorrariamos memoria. A su vez, buscamos optimizar la eficiencia de la votación de los k vecinos poniendoles peso a cada uno, leimos y concluimos que sería una buena opción asignarle a cada vecino un peso de $\frac{1}{d}$ donde d es la distancia al cuadrado antes calculada. De esta manera, si el vecino era más cercano, d sería menor, y por consiguiente, tendría mayor peso.

Para implementar el algoritmo en si, la idea principal fue utilizar una cola de prioridad de la std implementada sobre max heap, de esta manera no tendríamos que ordenar todas las distancias al cuadrado. Lo que hacemos es simplemente meter las primeras k distancias calculadas en un max heap en forma de duplas<dist, label>. La relación de orden se da por la primer componente. Luego se pregunta si la siguiente distancia calculada es menor que el primer elemento de la cola. Si esto sucede se extrae al primero en $\theta(1)$ y se ingresa la nueva distancia en $\theta(\log(k))$. Este ciclo itera n-k veces. Sin embargo, muchas de estas iteraciónes quizas ni si quiera tengan que ingresar la distancia a la cola.
Finalizado el ciclo, se extraen los k resultados en un vector de 10 posiciones que representan los dígitos, sumando el peso del vecino correspondiente a la posición i por cada resultado que contenga el label i.
Se retorna la posición del vector tal que v[i] es el máximo del v.

Ahora, veamos que si se tiene la SVD M = U ΣV^t, V es la misma matriz que se obtiene al diagonalizar M. Sea X la matriz X definida a lo largo del informe. M la matriz de covarianza asociada. Como $M = X^t XyX \in \mathbb{R}^{nxm}$, M $\in \mathbb{R}^{mxm}$. Además, es simétrica, por lo cual es diagonalizable por semejanza, y entonces existe una base de autovectores de esta matriz. Por otro lado, también usando la simetría de M, resultan ser ortonormales(todo esto fue probado en la teórica). Luego, sea $\{v_1, v_2, ..., v_m\}$ con $v_i \in \mathbb{R}^m \ \forall$ 1 $\leq$ i $\leq$ m , el conjunto ortonormal de autovectores de M.

Ahora, como v_i es autovector de M $\Longrightarrow$ M $v_i = \lambda_i v_i$ (con $\lambda_i \in \mathbb{R}$ por ser M simétrica). Por otro lado $M^t M$ también es simétrica y por lo tanto tiene una base de autovectores ortonormales. Se puede ver facilmente que esta base es igual a $\{v_1, v_2, ..., v_m\}$. Ya que como M es simétrica $M^t M v_i = MMv_i$= $M\lambda_i v_i = \lambda_i M v_i = \lambda_i^2 \ v_i$, lo que indica que los v_i son autovectores de $M^t M$ con autovalor asociado λ_i^2 = σ_i^2. Siendo los σ_i la diagonal de la matriz Σ. Además como M es simétrica, es semidefinida positiva y por consiguiente sus autovalores son mayores o iguales a 0. Luego como σ_i >0 y $\lambda_i \geq 0$, se deduce que σ_i =$\lambda_i \ \forall$ 1 $\leq$ i $\leq$ m.

Tomemos V a la matriz que tiene al conjunto $\{v_1, v_2, ..., v_m\}$ ortonormal de autovectores de $M^t M$ como columnas. V $\in \mathbb{R}^{mxm}$. Además elegimos los u_i tales que $\frac{Av_i}{\sigma_i} = u_i$. Luego $\frac{\lambda_i^2 v_i}{\sigma_i} = u_i$, y entonces se tiene $v_i = u_i$.

Entonces vemos que tomando U $\in \mathbb{R}^{mxm}$ como la matriz que tiene a los v_i como columnas, Σ $\in \mathbb{R}^{mxm}$ como la matriz que tiene a los λ_i en la diagonal, y a V también $\in \mathbb{R}^{mxm}$ como la matriz que tiene a los v_i como columnas. Tenemos una descomposición SVD de M / M = U ΣV^t . Considerando que U = V y que es ortonormal(V es la inversa de V^t), podemos decir que M = V ΣV^t .

Finalmente como M es diagonalizable por semejanza $\exists$ P $\in \mathbb{R}^{mxm}$ / M = P^{-1} D P . Tomando P^{-1}= V, siendo D la matriz que tiene los autovalores de M en la diagonal, se puede ver facilmente que se cumple la ecuación. Ya que

$$\text{M } [v_1, v_2, ..., v_m] = [v_1, v_2, ..., v_m] \text{ D}$$
$$\text{M V} = \text{V D}$$
$$\text{M} = \text{V D } V^-1 = \text{V D } V^t = \text{V } \Sigma \ V^t$$
notar que V es inversible por ser ortonormal.

Dejando en evidencia lo que queríamos ver. La V, es la matriz que tiene a los autovectores de M como columnas.

Bibliografía

Métodos numéricos para ingenieros, quinta edición Steven C.Chapra Raymond P.Canale
Apuntes teóricos de Isabel, ama y señora de los métodos numéricos.