DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 2

Reconocimiento de Dígitos

Métodos Numéricos

Integrante	LU	Correo electrónico
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

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 = 28x28, 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 un algoritmo de *reducción*: análisis de componentes principales (o PCA, por las siglas en inglés de *principal component analysis*).

Este identifica la información más relevante de nuestros datos usando propiedades de autovalores y autovectores, y nos va 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 aplicarlo, 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.

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 pixeles cada una, por esto es que los vectores son de 785 coordenadas cada uno.

También creamos una segunda matriz con la misma estructura que la anterior para las imágenes a ser probadas. 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 probada.

2. Métodos numéricos

En esta sección explicaremos los distintos métodos que utilizamos para encontrar a que dígito corresponde cada imagen nueva a ser probada. Incluimos aquí métodos de comparación de imágenes (kNN), métodos de pre-procesamiento (PCA) 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 identificar 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 una clase de dígito para devolver como respuesta.

Para elegir las k imágenes más parecidas nuestro procedimiento constó en hallar la distancia vectorial (recordemos que a cada imagen la tenemos representada como un vector) entre la imagen a identificar 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 prueba. Estas distancias se fueron agregando una por una en un vector de tuplas de dos coordenadas cada una, donde la primera contenía la etiqueta (número del 0 al 9) de la imagen en la base de entrenamiento con la cual se calculó 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 más "parecidas", lo siguiente fue seleccionar aquella cuya etiqueta se repetía más veces. Para ello se observó la primera coordenada de cada una de las k tuplas y se eligió como respuesta aquella que aparecía más veces. Por ejemplo, si k fuese igual a 5 y de esas cinco imágenes, tres fuesen 0s, y las dos restantes 8s, la respuesta seria 0. Por otro lado, en el caso de que halla dos o más 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 autovalor de mayor módulo 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.

Es decir, un autovalor *dominante*.

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 candidato inmediatamente anterior a 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 el resto de los 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, luego nuevamente deflación y así α veces, donde α 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 se utiliza para pre-procesar las imágenes de la base de entrenamiento con el objetivo de conseguir mejoras en tiempos de cómputo y a veces también en la calidad de los resultados.

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

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

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 está

definida 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 α autovalores y sus respectivos autovectores, donde α 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, dividirlas 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 categoria pertenece la imagen a testear. En nuestro caso, este último paso lo hacemos mediante kNN.

2.4. Validación cruzada K-plegada (K-fold CV)

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 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 imagenes 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 parcicionamiento.

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.ª la imagen testeada.

Finalmente se llama a la funcion 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 el parámetro α de PCA y, a partir de ello, 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 sobreescribir dichas matrices con los nuevos valores, pero sin cambiar sus dimensiones. Con lo cual los primeros α valores de estas matrices seran 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 dividirlas 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 utilizo varias veces en el método PCA. 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 28000 de prueba, vamos a partir nuestro conjunto de imágenes de entrenamiento (cuya clasificación conocemos para toda imagen) una parte en entrenamiento y otra en prueba.

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* (que el éxito del análisis realizado por el programa dependa demasiado del conjunto de datos particular utilizado para entrenarlo, que sea poco generalizable).

4.1.2. Métricas de evaluación de clasificadores

Para evaluar la precisión de nuestros resultados utilizaremos tres métricas distintas.
La primera es el porcentaje de aciertos, la cual simplemente evalúa la cantidad de identificaciones positivas sobre la cantidad total de imágenes probadas, multiplicado por cien. El resultado de dicha métrica sera un porcentaje.

La segunda métrica 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: la cantidad de imágenes que se clasificaron como no-pertenecientes a cierta clase, cuando en realidad sí pertenecían.

Por ultimo utilizaremos *precision*, la cual es una métrica que informa la cantidad de imágenes correctamente clasificadas, y aquellas que se identificaron como pertenecientes a cierta clase y en realidad no lo eran (falsos positivos).

4.1.3. Mediciones de tiempo

Para realizar las mediciones de tiempo utilizamos la biblioteca *ctime*. Para que nuestras mediciones sean lo mas precisas posibles, iniciamos la medición de tiempo justo antes de realizar el método y lo paramos apenas terminada el mismo, para no tomar en cuenta tiempos que no sean exclusivamente del método.

También realizamos cada experimento de medición tres veces y tomamos el promedio de los valores obtenidos.

Los experimentos fueron realizados en computadoras contiguas de los laboratorios de computación de FCEN UBA.

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, el porcentaje de aciertos, *precision* y *recall*.

4.1.5. Recopilación de datos y generación de gráficos

En este trabajo práctico realizamos experimentos para los cuales tomamos gran cantidad de mediciones. Volcamos los mismos en un programa de planillas de cálculo y confeccionamos los gráficos con las herramientas incluidas en el mismo.

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 guiones (*scripts*) *shell*.

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.

4.2. PCA + kNN vs kNN solo, variando k y α

En este experimento vamos a analizar la calidad de los resultados obtenidos combinando PCA con kNN vs utilizando únicamente 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 α de dimensiones a considerar en PCA.

4.2.1. Hipótesis

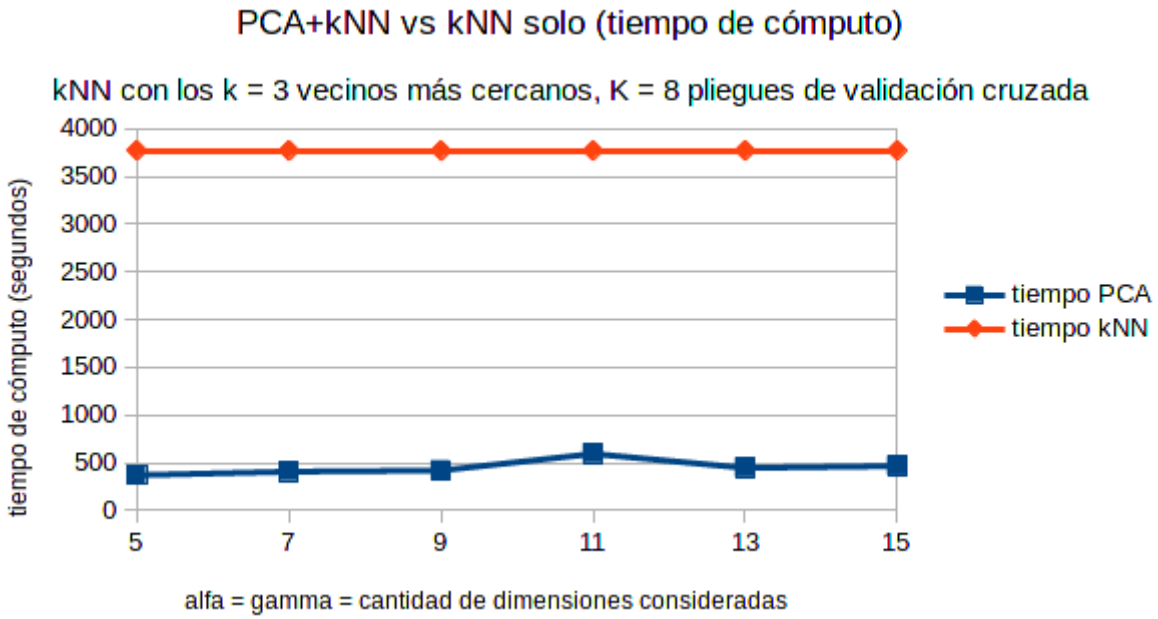
Esperamos que considerar una mayor cantidad de dimensiones al momento de identificar un dígito manuscrito, mejore nuestro porcentaje de aciertos, y a partir de cierto parámetro deje de hacerlo (cuando se empiecen a considerar dimensiones con muy poca varianza). Es decir, que el crecimiento se estanque.

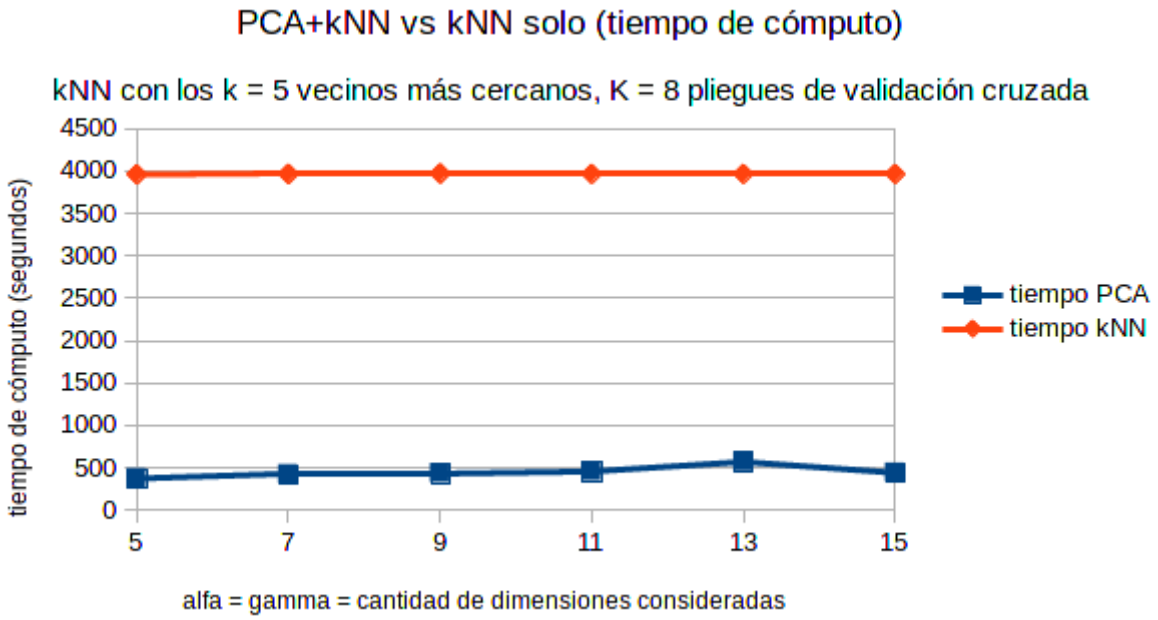
Por otro lado, considerar una mayor cantidad de dimensiones creemos que nos va a costar más tiempo de cómputo.

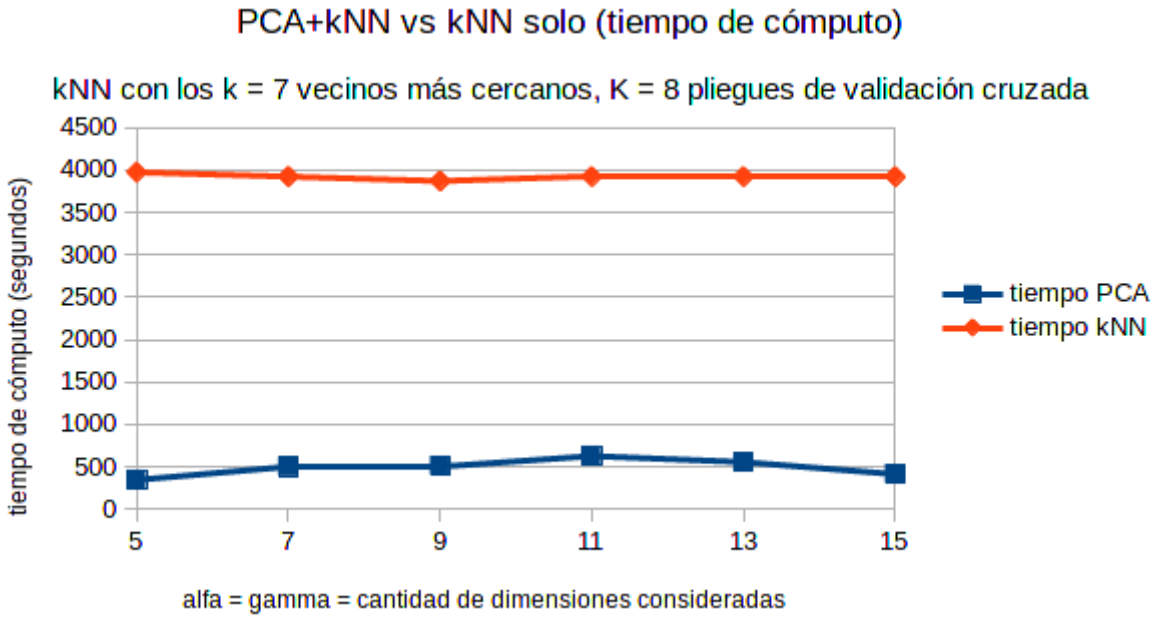
Pensamos que aumentar la cantidad k de vecinos de kNN, hará que aumente la cantidad de aciertos. Pero si la llegáramos a aumentar demasiado, podría llegar a perjudicar la clasificación (a diferencia de aumentar la cantidad de dimensiones, que suponemos que sólo pagaríamos con mayor tiempo de cómputo).

4.2.2. Resultados y discusión

En los siguientes tres gráficos se muestra la variación en el tiempo de computo para los métodos PCA+kNN y kNN solo, comparándolos entre si, y variando la cantidad de vecinos más cercanos.

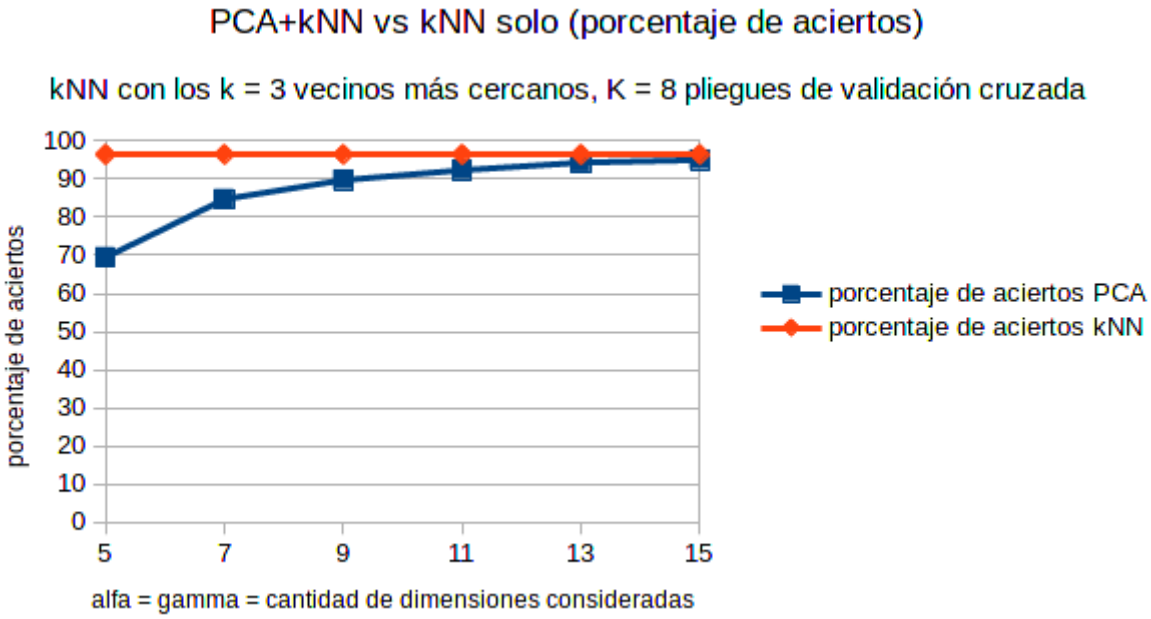


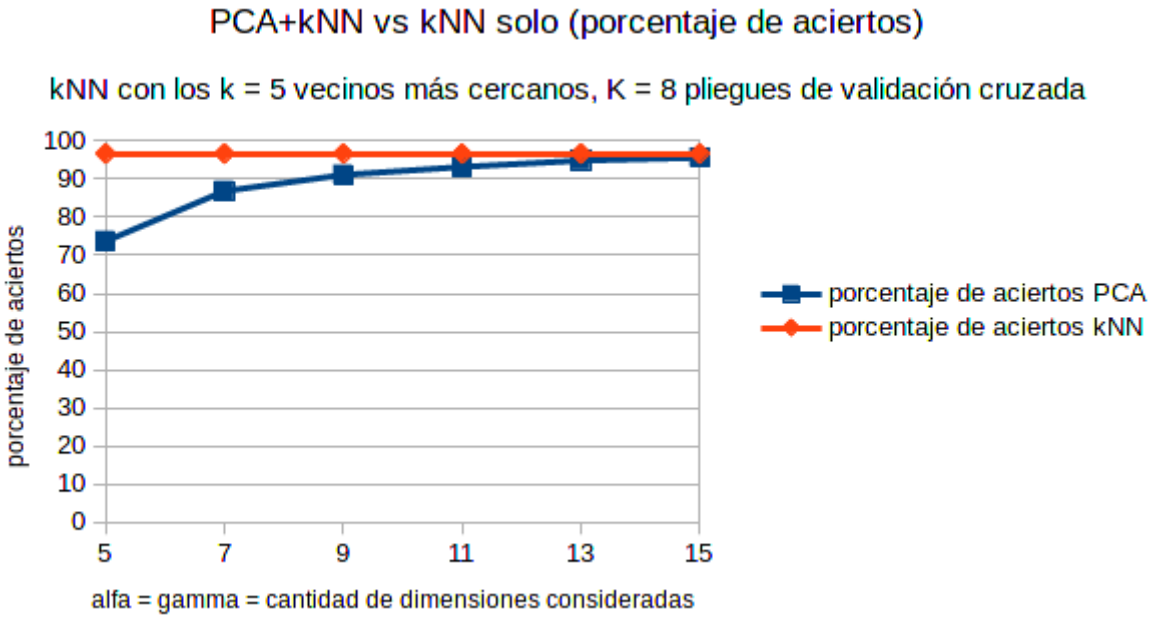


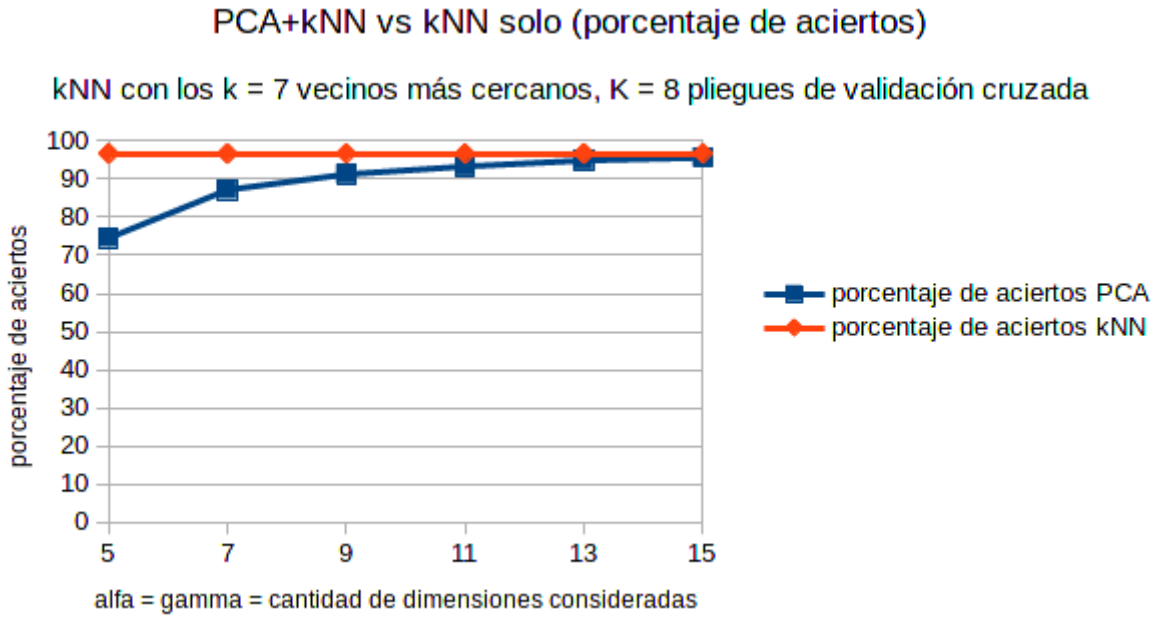


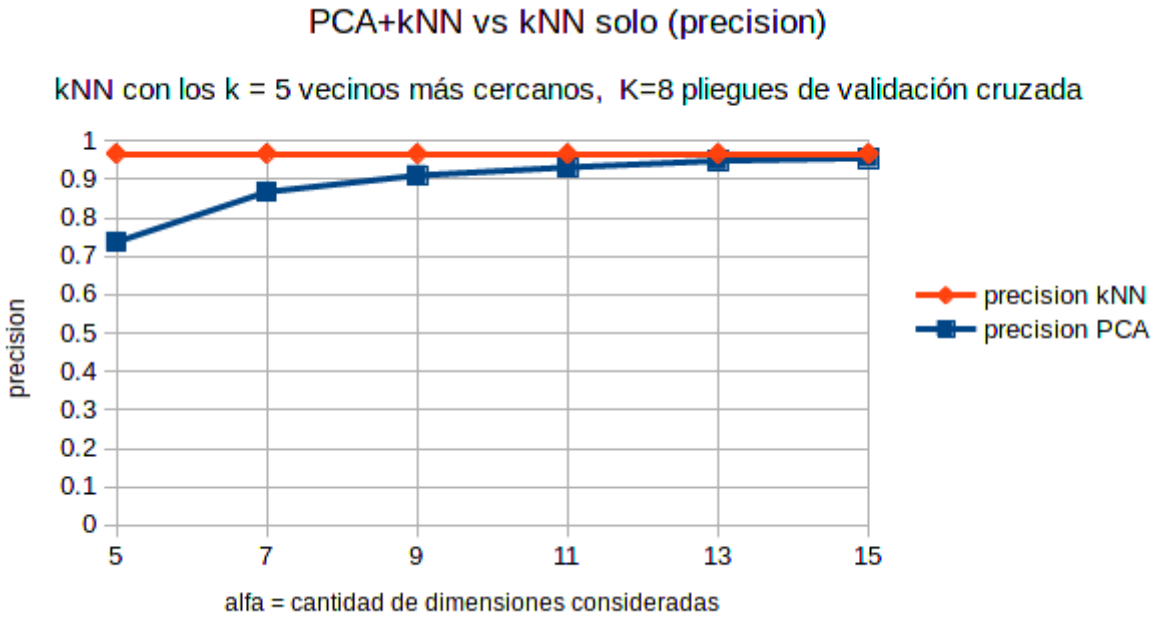
Observamos que el tiempo de computo de PCA + kNN es siempre casi un orden de magnitud menor al de kNN solo.
Por otro lado, para los tres k distintos, los tiempos de cómputo parecen ser muy similares. No observamos un cambio importante cambiando la cantidad de vecinos mas cercanos entre 3, 5 y 7, pues los 3 gráficos son muy similares.

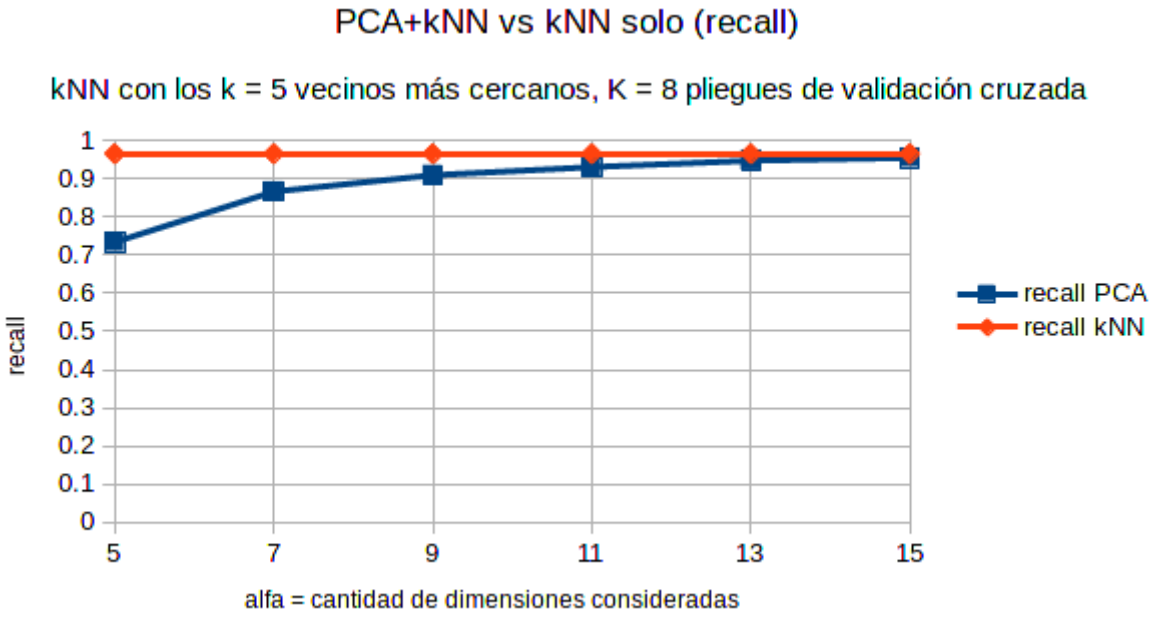
En los gráficos que siguen analizaremos la eficiencia de ambos métodos a la hora de clasificar las imágenes. Comparamos a PCA+kNN y a kNN solo nuevamente, variando la cantidad de vecinos más cercanos, y utilizaremos las métricas de porcentaje de aciertos, precision y recall.






PCA+kNN vs kNN solo (porcentaje de aciertos)
kNN con los k = 7 vecinos más cercanos, K = 8 pliegues de validación cruzada
porcentaje de aciertos
100
90
80
70
60
50
40
30
20
10
0
5
7
9
11
13
15
alfa = gamma = cantidad de dimensiones consideradas
porcentaje de aciertos PCA
porcentaje de aciertos kNN


PCA+kNN vs kNN solo (precision)
kNN con los k = 5 vecinos más cercanos, K=8 pliegues de validación cruzada
precision
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
5
7
9
11
13
15
alfa = cantidad de dimensiones consideradas
precision kNN
precision PCA



Notamos que kNN cuenta con una ventaja en cantidad de aciertos sobre PCA, que oscila entre un 20 % y un 25 % para los 3 valores de k considerados, cuando PCA tiene en cuenta únicamente las 5 dimensiones más significativas.

Pero a medida que PCA considera más dimensiones, esta diferencia se va achicando, hasta que practicamente el porcentaje de aciertos converge, cuando las α-dimensiones están entre 13 y 15.

Precision y *recall* reflejan este mismo comportamiento y tal vez no aportan tanta información, para este experimento en particular.

También notamos que la mejora al aumentar α no parece depender en absoluto de la variación de los k vecinos más cercanos utilizados para la realización de kNN.

Una posible explicación, pensamos, 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 qué clase pertenece.

4.3. PCA+kNN vs kNN solo, variando K-pliegues de validación cruzada

En este experimento quremos observar qué sucede con PCA+kNN vs kNN solo, cuando varía la cantidad K de pliegues de validación cruzada que utilizamos para calcular cada uno de ellos.

4.3.1. Hipótesis

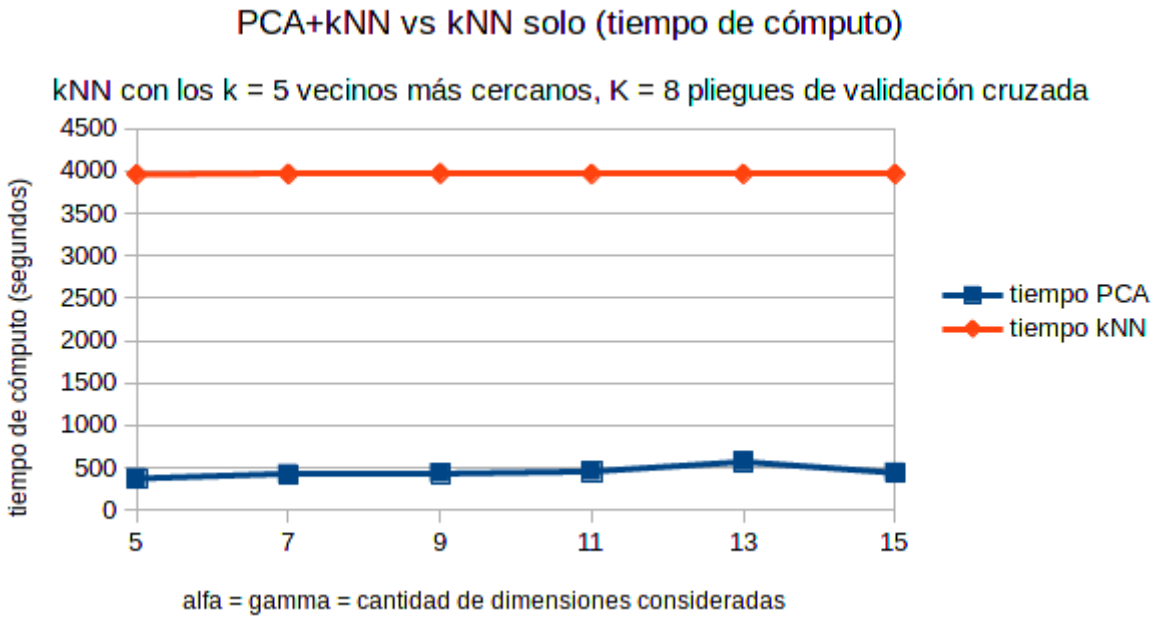
El valor de K es proporcional a la cantidad de imagenes de entrenamiento a utilizar. Suponemos que mientras más imágenes usemos para entrenar, mejores van a ser nuestros resultados en cuanto a porcentaje de aciertos.

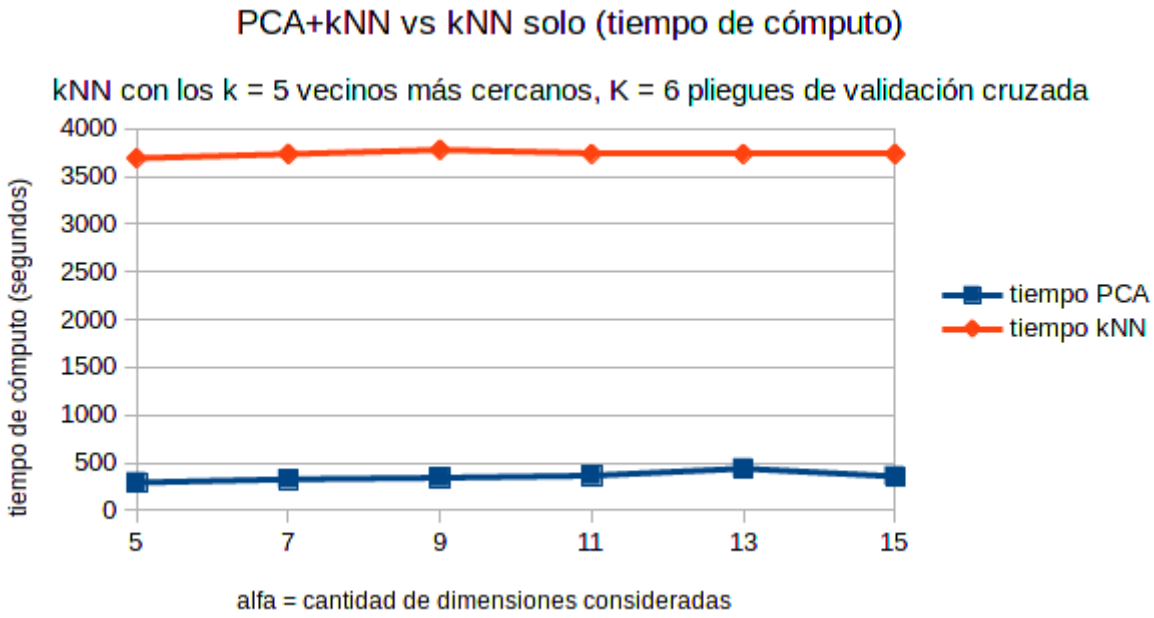
Para una mayor cantidad de pliegues, esperamos obtener mayores tiempos de cómputo en ambos experimentos.

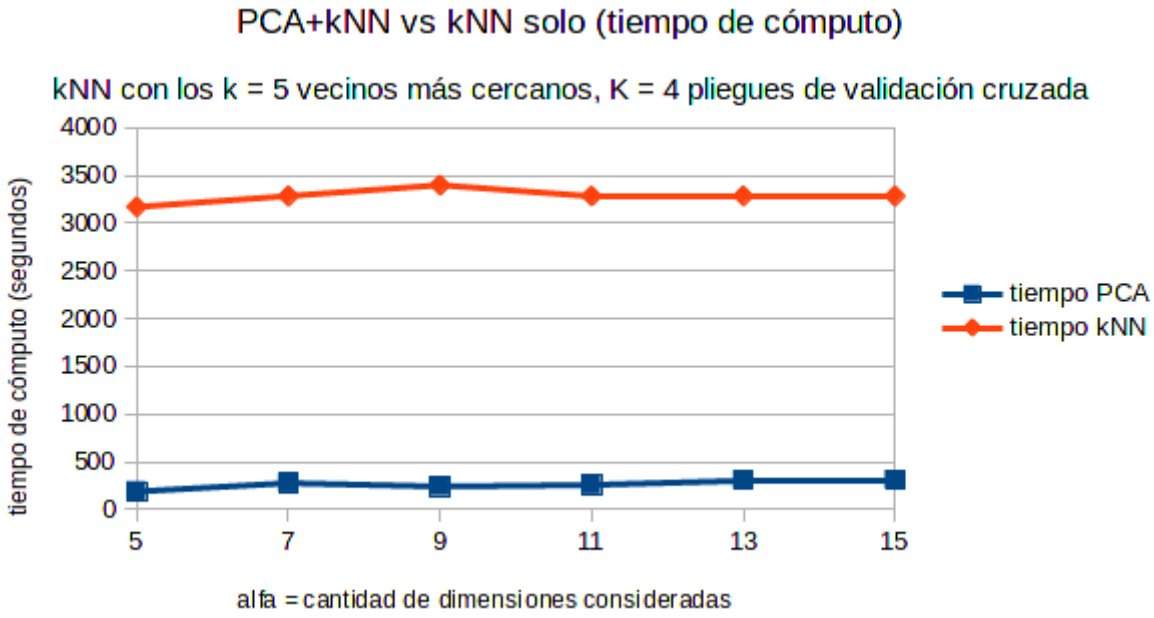
También pensamos que con mayor cantidad de pliegues nuestras métricas serán más precisas.

4.3.2. Resultados y discusión

En los siguientes gráficos se muestra la variación de tiempo de cómputo y la variación en porcentaje de aciertos comparando a los dos métodos entre si, fijando el parámetro k en 5:



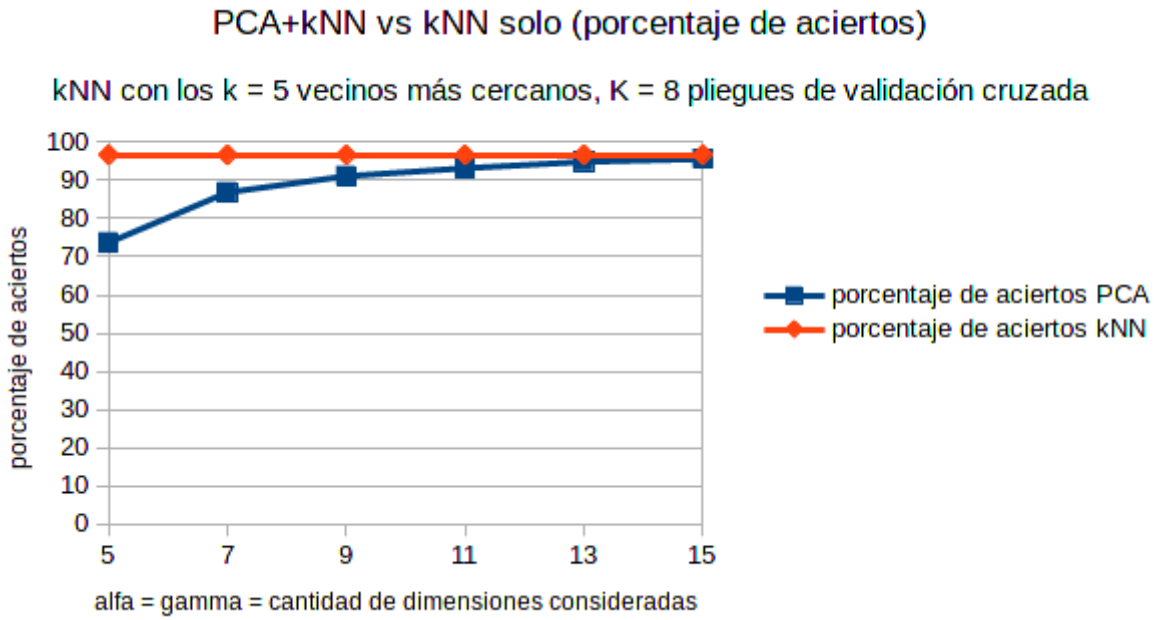


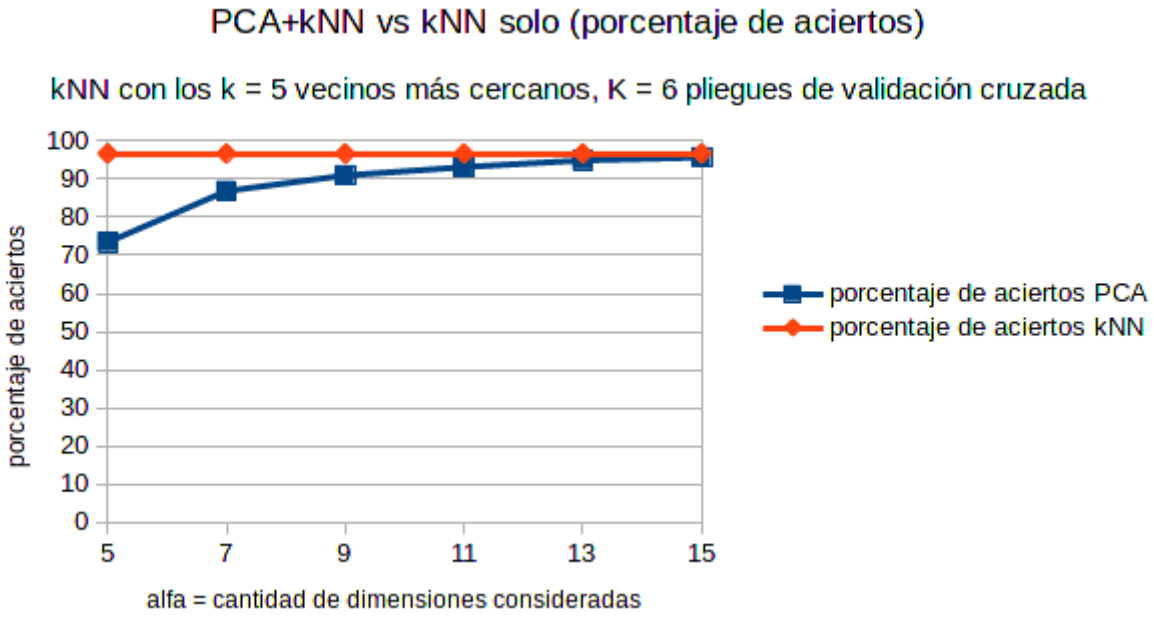


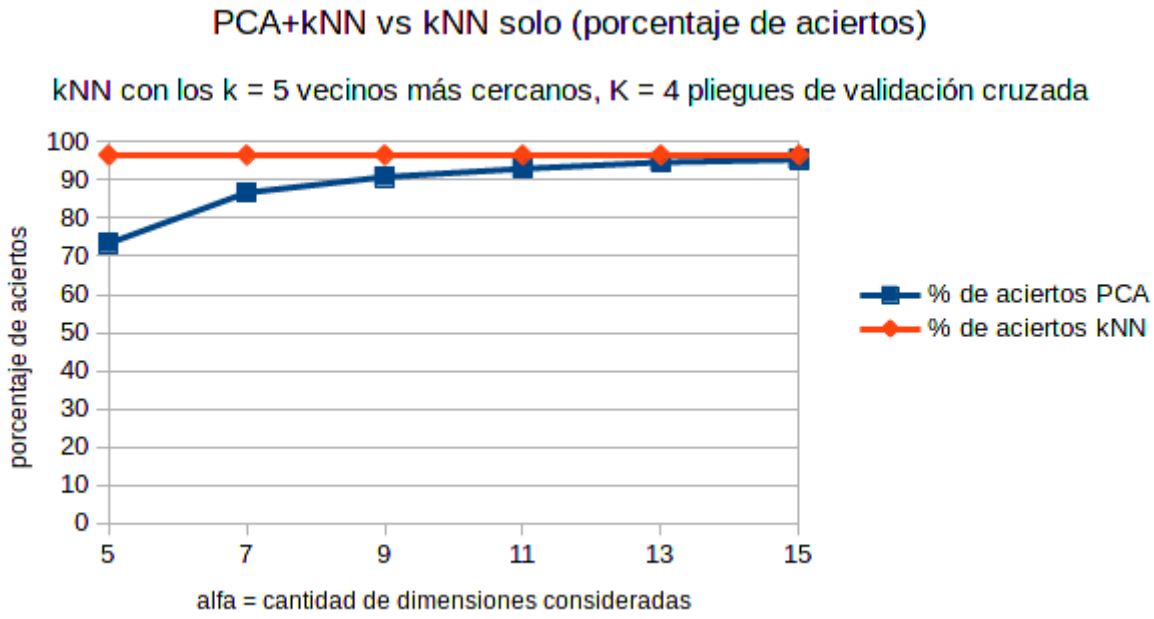
Predeciblemente, aumentar K aumenta el tiempo de cómputo. Con kNN tardando alrededor de 4000 segundos (aproximadamente 1 hora) para $K = 8$ y PCA+kNN alrededor de 500 segundos (aproximadamente 8 minutos), para bajar a alrededor de 3250 segundos y alrededor de 250, para $K = 4$, pasando por valores intermedios para $K = 6$.

Es lógico que kNN tarde mucho más que PCA+kNN para imágenes de casi 800 píxeles, ya que trabaja con las dimensiones más y menos significativas por igual.

Y es esperable también que más pliegues impliquen más tiempo, pues cada pliegue debe hacer de conjunto de prueba exactamente una vez, mientras el resto hace de conjunto de entrenamiento, lo cuál implica K experimentos en total.







Observando los porcentajes de aciertos al aumentar K, obtenemos 3 gráficos que son prácticamente iguales. Creemos que esto es una buena señal, ya que indica que el porcentaje de aciertos no depende del conjunto de datos que usemos. Es decir, no estamos sobre-entrenados para nuestro conjunto de datos: el análisis de nuestro programa se puede generalizar.

4.4. Observaciones

4.4.1. 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 el método de reducción PCA.

4.4.2. 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.

Parte III
Conclusiones

5. 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 computo. Siendo ultima esta la diferencia mas significativa entre ambos métodos, elegiríamos PLS-DA en el caso de tener que decidir entre una y 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 mas precisos. De todos modos esta decisión no sera definitiva, ya que dependiendo del caso, puede requerirse priorizar el tiempo de computo.
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, aun que 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 pliegues de validación cruzada. 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 (por lo explicado previamente sobre aumentar la cantidad de pliegues).

6. 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 observar mediante corridas básicas, 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.

También nos quedo 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 ultimo, 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

7. Apéndice A: código fuente relevante

7.1. 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[
                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;
}
```

7.2. 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;
}
```

7.2.1. 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;
}
```

7.3. 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(ImagenesEntrenamiento, media, ImagenesEntrenamiento.size());

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

centrar(ImagenesTest, media, ImagenesEntrenamiento.size());

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

resultados = Knn(ImagenesEntrenamiento, ImagenesTest, k, alfa, metodo);
```

7.4. Validación cruzada k-plegada (k-fold x-validation)

```
function particionarValidX(Kpliegues, rutaArchivoSalida)

pkg load statistics              % octave-cli

cantImgsEntrenamiento=42000;

fid=fopen(rutaArchivoSalida,'a');

c = cvpartition(cantImgsEntrenamiento,'kfold',Kpliegues);

for i = 1:Kpliegues
        a = training(c,i);
        b(i,:) = a';
end

dlmwrite(rutaArchivoSalida, b, '-append', 'delimiter', ' ');

fclose(fid);
```

7.5. Creador de casos de prueba

```
# Uso: ./crearTests l_kVecinos l_alfaDims l_Kpliegues PCA
# Por ejemplo: ./crearTests "4 5 6" "49 50 51" "9 10 11" 1

rutaDatos='../data/'

nomCarpetaNuevosTests='tests_recien_creados'

mkdir "$nomCarpetaNuevosTests"

for kVecinos in $1; do

  for alfaDims in $2; do

    for Kpliegues in $3; do

      for nroExperimento in 1 2 3; do

        if [[ $4 == 0 ]]; then
          nomArchivo="$nomCarpetaNuevosTests"'/test_k'$kVecinos'a'$alfaDims'K'$Kpliegues'-'$nroExperimento'.in'
        else
          nomArchivo="$nomCarpetaNuevosTests"'/test_k'$kVecinos'a'$alfaDims'K'$Kpliegues'PCA-'$nroExperimento'.in'
        fi

        echo "$rutaDatos" $kVecinos $alfaDims $Kpliegues > "$nomArchivo"

        #matlab -nojvm -nodesktop -r "particionarValidX($Kpliegues, '$nomArchivo'); quit;" && reset         # el "reset" es para des-bug

        octave-cli --eval "particionarValidX($Kpliegues, '$nomArchivo'); quit;"  # listos o no: migramos a octave. venció la licencia de
      done
    done
  done
done
```

7.6. Generador del script de corridas de máquina paralelas (fragmento)

```
indcArchivoPtoIn=0

while [ $indcArchivoPtoIn -lt $cantArchivosPtoIn ]; do

        archivoPtoInActual=`echo "${arrArchivosPtoIn[$indcArchivoPtoIn]}"`
        archivoPtoInActual=`echo "${archivoPtoInActual::-3}"`   # le saco el ".in"

        if [[ $archivoPtoInActual == *PCA* ]]; then
                numMetodoActual=1
        else
                numMetodoActual=0
        fi


        # esto es de obsesivo: es para que queden alineadas las tabulaciones

        numMaq=`echo -e "${maqsSSHables[$(($indcArchivoPtoIn % $cantMaqsSSHables))]}" | sed -r 's/^([^.]+).*$/\1/; s/^[^0-9]*([0-9]+).*$/\1/'`

        if [[ $numMaq -gt 9 ]]; then
                tabs=`echo -e "\t"`
        else
                tabs=`echo -e "\t\t"`
        fi


        echo -e "ssh\t-F ~/.ssh/ssh_config\t${maqsSSHables[$(($indcArchivoPtoIn % $cantMaqsSSHables))]}$tabs\"$rutatp2home/bajarRepoAlLibre.sh && "
        echo -e "\t\t\t\t\t\t\t\tcd $rutatp2libre/src && "
        echo -e "\t\t\t\t\t\t\t\t./tp $dire$archivoPtoInActual.in $dire$archivoPtoInActual.out $numMetodoActual && "
        echo -e "\t\t\t\t\t\t\t\t$rutatp2home/subirResultados.sh\" & \n\n"

        echo -e "sleep 10\t# agrego una pausa de 10 segundos para no saturar tanto el gitlab (que no baje el repo de todas a la vez tan al mismo tiempo)\n\n"

        indcArchivoPtoIn=$((indcArchivoPtoIn+1))

        if [[ $(($indcArchivoPtoIn % $cantMaqsSSHables)) == 0 ]]; then
                echo -e "sleep 1800\t# ya usamos todas las maquinas de los labos, bancar media hora antes de seguir\n\n"
        fi

done
```

8. Apéndice B

8.1. 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.

Esto es relevante porque la rutina de Octave que utilizamos para armar las particiones de validación cruzada, toma imágenes contiguas del conjunto de entrenamiento para la misma partición (en vez de tomarlas al azar).

Para el trabajo supusimos que *train.csv* estaba aleatoriamente desordenado y ninguna clase de dígito estaba sobre-representada en ninguna partición de los datos, pero esto podría no ser cierto y es algo que vale la pena estudiar de antemano.

Parte V
Referencias

9. Cursada de Métodos Numéricos de Computación en FCEN UBA

[MetNum] http://www-2.dc.uba.ar/materias/metnum/homepage.html

10. The C++ Programming Language

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

11. CPlusPlus.com - ctime reference

[CppTime] http://www.cplusplus.com/reference/ctime/clock/

12. Documentación de LibreOffice Calc

[LOCalc] https://help.libreoffice.org/Chart/Charts_in/es

13. GNU Octave cvpartition docs

[Octave] https://octave.sourceforge.io/statistics/function/@cvpartition/cvpartition.html

14. 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

15. Apuntes de Métodos Numéricos en CubaWiki.com.ar

[Apunte] http://www.cubawiki.com.ar/images/4/45/Metnum_apunte.pdf

16. Apunte de SSH de Marco Vanotti

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

17. Ayuda de Gitlab en la instancia del DC para configurar acceso con par de cripto-llaves

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