Reconocimiento de dígitos manuscritos a partir de la factorización SVD

Métodos Numéricos, Departamento de Computación, Universidad de Buenos Aires

Guillermo Gallardo Diez, Damian Eliel Aleman, y Luis Scoccola

gagdiez.c@gmail.com
damianealeman@gmail.com
luis.scoccola@gmail.com

Reconocimiento de dígitos manuscritos a partir de la factorización SVD

Métodos Numéricos, Departamento de Computación, Universidad de Buenos Aires

Índice

Resumen

En el presente trabajo estudiaremos el uso de SVD en el reconocimiento automatizado de dígitos manuscritos, utilizando las autoimágenes de los digitos extraidos de una serie de entrenamiento para compararlas con imágenes de dígitos manuscritos. Analizaremos performance y calidad de resultados de dos métodos distintos para hallar autovectores de matrices y veremos como varían los resultados en base a la precisión con la que se calculan los autovectores. También analizaremos la calidad de los resultados utilizando varios métodos de comparación de autoimágenes.

Index Terms

autoimágenes, reconocimiento automatizado, SVD, OCR, reconocimiento óptico, dígitos manuscritos

I. Introduccíon teórica

SE denomina factorización SVD (*singular value decomposition*) de una matriz A (de dimensiones arbitrarias) a la descomposición de dicha matriz en el producto: $A = U\Sigma T^t$. Donde U y T son ortogonales y Σ tiene coeficientes nulos en todo elemento $\sigma_{i,j}, i \neq j$ (usualmente llamada diagonal, si bien sus dimensiones son iguales a las de A). Si los coeficientes de la diagonal de Σ tienen la particularidad de estar ordenados de la forma $\sigma_{i,i} \geq \sigma_{j,j}, i < j$, entonces esta factorización es única. Una propiedad importante de esta factorización es que las matrices U y V tienen como columnas los autovectores de AA^t y A^tA respectivamente. Los elementos de la diagonal de Σ se denominan valores singulares de X.

La factorización puede ser interpretada de varias formas y suele brindar mucha información acerca de la matriz. En este caso queremos calcular las componentes principales de un cierto vector que representa una imagen de un dígito manuscrito. Esto lo logramos calculando la factorización SVD de la matriz de covarianza entre píxeles X^tX. Para conseguir la matriz X se parte de una base de datos $x_1, \ldots, x_n$ con imágenes de dígitos manuscritos de dimensiones iguales. Luego se interpreta cada imagen x_i como un vector fila y se calcula, índice a índice, el vector promedio μ de todas las imágenes. La matriz X se obtiene al poner como filas los vectores $x_i - \mu$ y dividiendo por $\sqrt{n-1}$.

Las k componentes principales de una imagen dada se obtienen ralizando el producto interno entre el vector que representa la imagen y los k autovectores que se corresponden con los k autovalores de mayor módulo de la matriz de covarianza entre píxeles. Es decir, las k componentes principales de una imagen x son los primeros k coeficientes del vector que resulta de multiplicar V^tx, suponiendo que se realizó la factorización SVD de X de manera tal que los valores singulares de X se encuentran ordenados de mayor a menor.

Una vez obtenidas las componentes principales de la imagen que quiere reconocerse, pueden compararse con las componentes principales de los dígitos de la base de datos de diversas maneras. Típicamente se toma la distancia correspondiente a la norma euclídea entre vectores para buscar la imagen cuyas componentes principales se encuentren más cercanas a las de la imagen que quiere reconocerse.

II. Desarrollo

A. Descomposición SVD

De la introducción se deduce que no es necesario calcular de forma completa toda la descomposición en valores singulares. Basta tener las primeras k columnas de V. Es decir necesitamos un método para obtener los autovectores que se corresponden con los autovalores de mayor módulo de una matriz. Es importante notar que la matriz con la que trabajamos es simétrica con coeficientes reales y por lo tanto podremos utilizar metodos optimizados. Para obtener los autovectores implementamos dos estrategias:

- **Método de la Potencia y deflación:** Suponiendo que tenemos autovalores $|\lambda_1| \geq \ldots \geq |\lambda_n|$ y los primeros k son distintos dos a dos, obtenemos λ_1 (y su atovector asociado), mediante el método de la potencia. Luego construimos una matriz que tenga como autovalores $|\lambda_2| \geq \ldots \geq |\lambda_n| \geq 0$ usando el método de deflación. De esta forma inductiva calculamos los primeros k autovalores con sus correspondientes autovectores.
- **Método QR para matrices simétricas tridiagonales y método de la potencia inversa:** Asumiendo que partimos de una matriz simétrica, obtenemos una matriz de hessemberg (y en este caso tridiagonal, por ser simétrica) semejante, mediante reflexiones de Householder. Luego, mediante el algoritmo QR calculamos sus autovalores. Finalemente usamos esta aproximación de los autovalores para calcular los primeros k autovectores con el método de la potencia inversa.

A continuación se explican las implementaciones con más detalle.

A.1 Método de la potencia y deflación

El método de la potencia es un método iterativo. Se parte con un vector inicial v_0 y se genera una sucesión de vectores v_i realizando, en cada iteración,

$$v_{i+1} = \frac{Av_i}{\|Av_i\|_\infty}$$

Asumiendo que la matriz tiene un autovalor $\bar{\lambda}$ de módulo estrictamente mayor al resto de los autovalores y que v_0 no es ortogonal al autovector $\bar{v}$ asociado a dicho autovalor, la sucesión converge a $\bar{v}$.

El método no depende fuertemente de la norma que se usa para renormalizar en cada iteración. La única diferencia es que el autovector obtenido estará normalizado con la norma usada. En nuestro caso usamos la norma infinito ya que se calcula de manera rápida computacionalmente.

Una vez que tenemos el autovalor $\bar{\lambda}$ obtenemos una matriz A' que conserva los mismos autovalores y autovectores exceptuando $\bar{\lambda}$ y $\bar{v}$, realizando:

$$A' = A - \bar{\lambda}\bar{v}\bar{v}^t$$

A.2 Método QR para matrices simétricas tridiagonales y método de la potencia inversa

Como la matriz de covarianza es simétrica, utilizamos el algoritmo 9,5 de [2] para obtener una matriz tridiagonal semejante a ésta. El método se basa en transformaciones de Householder. Una vez obtenida usamos el algoritmo 9,6 para calcular sus autovalores. Dado que trabajamos con matrices tridiagonales los autovalores pueden calcularse rápidamente y con mucha precisión.

Luego utilizamos el método inverso de la potencia. Este es bastante similar al método de la potencia pero permite calcular cualquier par autovalor-autovector asumiendo las mísmas hipótesis que el método de la potencia con el agregado de tener una aproximación razonable de los autovalores correspondientes a los autovectores que quieren calcularse.

Partiendo de la matriz A, un autovector v_0 y una aproximación a un autovalor λ obtenemos una sucesión v_i de autovectores que converge al autovector asociado al autovalor λ. Para esto planteamos la iteración:

$$v_{i+1} = (A - \lambda Id)^{-1} v_i$$

Notemos que este es el concepto matemático. En nuestra implementación no invertimos la matriz, sino que resolvemos un sistema de ecuaciones en cada iteración. Para alivianar el tiempo de cómputo realizamos una descomposición LU de la matriz del sistema que nos sirve para todas las iteraciones que se realicen con un mismo autovalor λ.

Dado que se usan transformaciones ortogonales, el método QR suele brindar resultados de mucha calidad.

B. Reconociendo dígitos usando las componentes principales

PARA reconocer dígitos a partir de sus componentes principales implementamos tres métodos. Cada uno de los metodos puede ser usado con la cantidad de componentes principales que se desee y por lo tanto no se hará referencia a dicha cantidad en la explicación. En la parte tests experimentaremos con diversas cantidades y buscaremos la más adecuada para cada método, ya que, veremos, que los mismos suelen requerir diversas cantidades para comportarse de forma óptima.

B.1 kVecinos

Se calculan las componentes principales c de la imagen a evaluar y las componentes c_i de todas las imágenes x_i de la base de datos. Se toman las distancias euclídeas $\|c - c_i\|$ y se cuentan las apariciones de cada dígito $(0, \ldots, 9)$ en el conjunto de los k c_i más cercanos a c. El dígito que registre mayor frecuencia se tomará como el el dígito escrito en la imagen a evaluar.

B.2 kVecinos ponderados

En un principio pensamos calcular el promedio de las distancias $\|c - cj_i\|$. Donde cj representa el conjunto de componentes principales de las imágenes del dígito j. De esta manera se está tomando la distancia promedio a todos los dígitos 0, a los dígitos 1, etc.

Vimos que este método no daba resultados razonables y parametrizamos el mismo, para dar más granularidad a la hora de utilizarlo. En lugar de promediar la distancia a todos los dígitos de la base de datos, lo hacemos únicamente con los k dígitos que se encuentren más cerca.

B.3 Distancia al promedio de las componentes

Para cada dígito $0, \ldots, 9$ se iteran todos los c_i y se generan diez $\bar{c}_j$ correspondientes con el promedio componente a componente de los c_i de los dígitos j. Luego se calcula la distancia de c a cada uno de los $\bar{c}_j$ y se toma como dígito escrito al j tal que $\|c - \bar{c}_j\|$ sea mínimo.

C. Tests

Instancias utilizadas:
Para realizar los distintos test se crearon dos instancias en base a los datos de entrenamiento, una de 6000 imagenes y otra de 60000, obtenidas a partir de los archivos `train-images-idx3-ubyte.gz` y `train-labels-idx1-ubyte.gz`. Por otra parte, la instancia de prueba que denominados `primeras5000.dat` corresponde a los primeros 5000 dígitos obtenidos del archivo `t10k-labels-idx1-ubyte.gz`[1].

C.1 Error de los autovectores calculados

Para los dos métodos utilizados para calcular autovectores diseñamos un experimento que busca cuantificar el error máximo que presentan los resultados. Se calculan los primeros k autovectores v_i y se normalizan con la norma euclídea. Luego se realiza Av_i para cada i y se normaliza el resultado, obteniendo $\hat{v}_i$. Finalmente se toman las diferencias $\|v_i - \hat{v}_i\|$. Mientras más precisos sean los resultados del algoritmo, más pequeñas serán estas distancias. De esta manera podemos comparar el comportamiento de los dos algoritmos con distintas tolerancias de entrada. La definición anterior está sujeta a un problema: un autovector se encuentra contenido en una recta y esa recta contiene dos vectores de igual norma. Ambos vectores son autovectores y la única diferencia entre ellos es el sentido. Teniendo esto en consideración podría ocurrir que la distancia calculada esté por arriba de $\sqrt{2}$ e incluso que sea igual a 2[2]. Por este motivo, cuando se obtiene una distancia mayor a $\sqrt{2}$ se multiplica a uno de los vectores por -1 y se la recalcula.

Esperamos obtener resultados más precisos al utilizar la estrategia *QR-potencia Inversa*. Basamos esta hipótesis en que se usan matrices ortogonales para calcular los autovalores. Estas matrices tienen el número de condición más bajo posible, 1. Una vez que tenemos aproximaciones de los autovalores de buena calidad el método de la potencia inversa brinda buenas aproximaciones de los autovectores.

C.2 Diferencias en el tiempo de ejecución para obtener autovectores

Dado que ambos métodos son esencialmente distintos cabe preguntarse si sus complejidades son distintas. Teóricamente notamos una diferencia importante: el método de la potencia inversa resuelve un sistema de ecuaciones por cada autovector y por ende tiene una complejidad de $\Omega(n^3)$, donde n representa la dimensión de la matriz. Por otro lado el método de la potencia sólo realiza una multiplicación entre matriz y vector y por lo tanto tiene una complejidad de $\Omega(n^2)$.

Si bien estas son solamente cotas inferiores esperamos ver una diferencia significativa en el tiempo

[1] Para generar esta instancia ver la sección *Código relevante y modo de uso, conversión de datos usando Matlab*
[2] En el caso en que obtengamos el mismo vector pero cambiado de signo.

de ejecución de ambas estrategias.

C.3 Cantidad de dígitos reconocidos en función del método, la cantidad de componentes utilizada y la precisión de los autovectores

Comenzamos buscando buenos parámetros para los métodos de reconocimiento de dígitos. Para esto calculamos autovectores con una buena precisión y experimentamos con la cantidad de componentes entre 20 y 100. Una vez que obtenemos los parámetros que se comportan mejor para cada método analizamos la diferencia de comporamiento de los métodos al variar la cantidad de componentes principales y la precisión con la cual se calculan los autovectores.

C.4 Influencia del tamaño de la base de datos

Basándonos en el test anterior comparamos el *hit rate* logrado por los métodos de reconocimiento de dígitos al variar al cantidad de imágenes en la base de datos.

C.5 Dígitos mejor reconocidos en función del método

Dado que implementamos más de un método para el reconocimiento de dígitos, nos preguntamos si los mismos funcionan mejor en distintos dígitos. Para responder al interrogante calculamos el *hit rate* de cada dígito por separado y comparamos los resultados de los distintos métodos.

III. Resultados

A. Error de los autovectores calculados

Calculando el error de los autovectores obtenidos mediante el método de la potencia[3] bajo distintos niveles de tolerancia obtenemos el siguiente resultado:

precisión	máximo	promedio
10	1.339164	0.11594685
1	1.339164	0.11594685
0.1	1.339164	0.11594685
0.01	1.113330	0.10259884
0.001	0.995734	0.10986505
0.0001	1.353526	0.08291460
0.00001	1.414020	0.19694885

Tolerancia Vs error máximo y error promedio de los autovectores
Método de la potencia en instancia `primeros5000.dat`

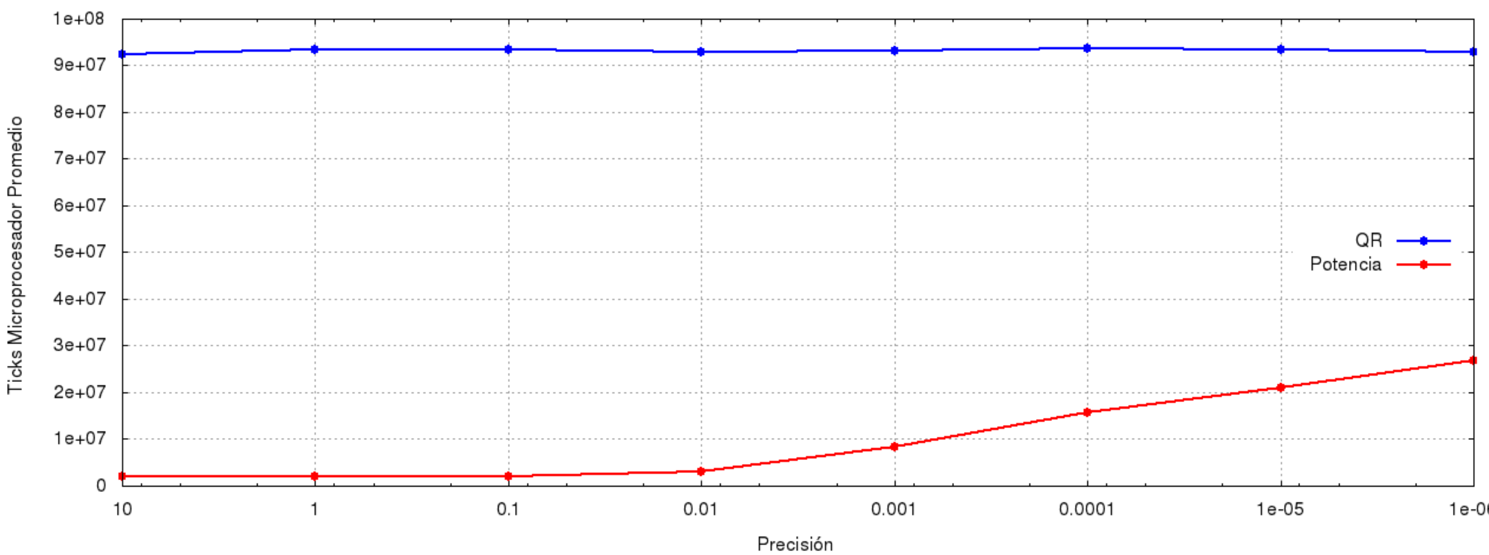
Vemos que los resultados mejoran hasta la tolerancia de 0,001, luego se desestabilizan.

[3] De la manera que se explica en el **Desarrollo**.

Cuando realizamos este mismo test al método *QR-potencia Inversa* no obtuvimos resultados significativos. Es decir, para toda tolerancia el error es de a lo sumo 10^{-5} y por lo tanto no nos resultó significativo.

B. Diferencias en el tiempo de ejecución para obtener autovectores

Comparamos el tiempo de ejecución de ambas estrategias con distintas tolerancias entre 10 y 10^{-6}, para esto calculamos el promediando de la cantidad de ticks que tardaba cada una en finalizar durante diez corridas[4].

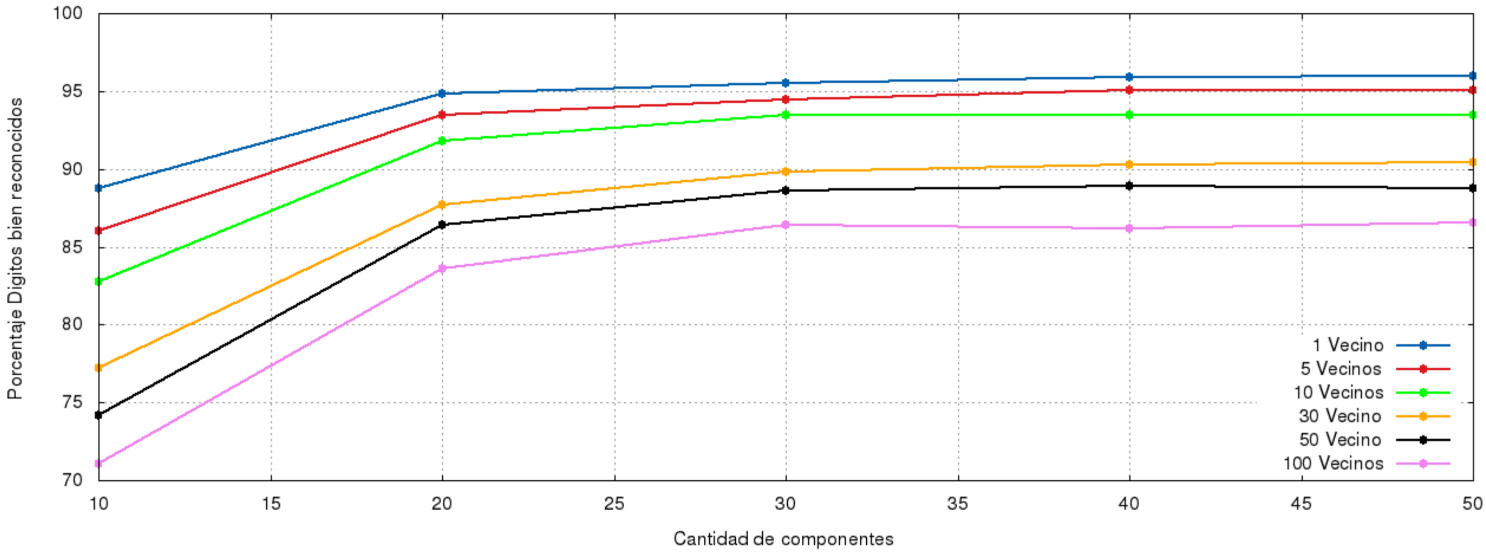


TOLERANCIA VS CANTIDAD DE CICLOS DE PROCESADOR
Método de la potencia y método *QR-potencia Inversa* en instancia `primeros5000.dat`

C. Cantidad de dígitos reconocidos en función del método, cantidad de componentes utilizada y la precisión de los autovectores

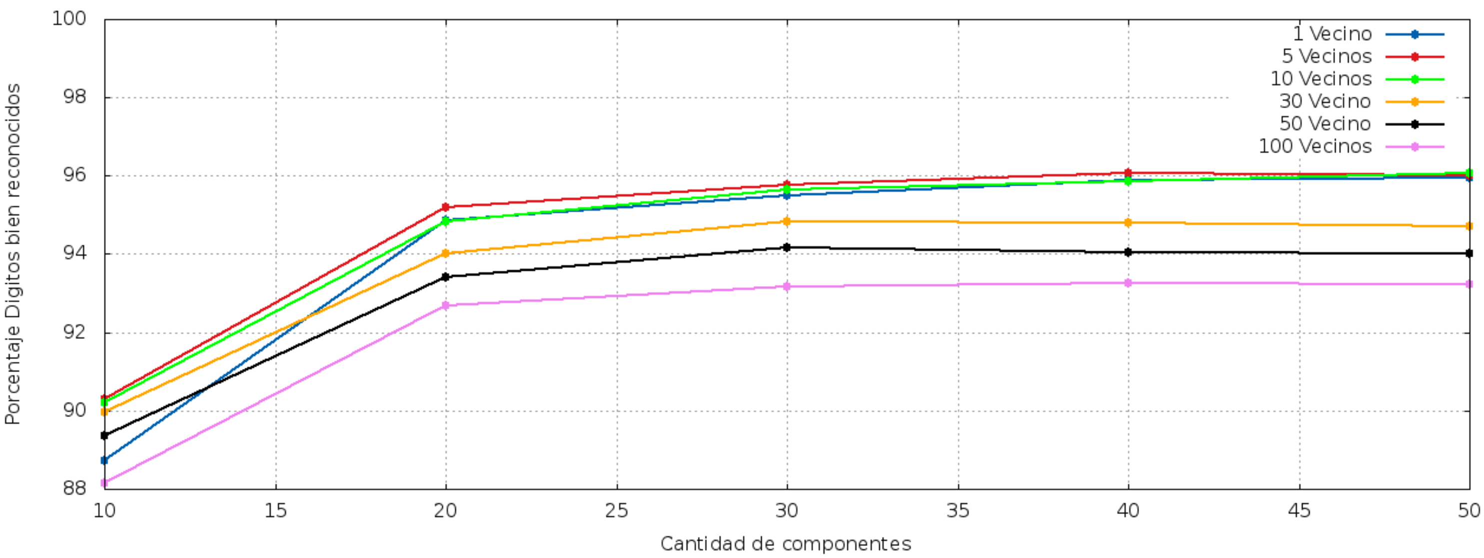
Tomando una precisión de 0,001 buscamos la cantidad de vecinos que optimice el reconocimiento de dígitos. Para el método de los vecinos ponderados obtenemos[5]:

[4] Ver Tabla A
[5] Ver Tabla B



Cantidad de componentes Vs porcentaje de dígitos bien reconocidos variando cantidad de vecinos
Método de la potencia/k-vecinos-ponderados con tolerancia 0,001 en instancia `primeros5000.dat`

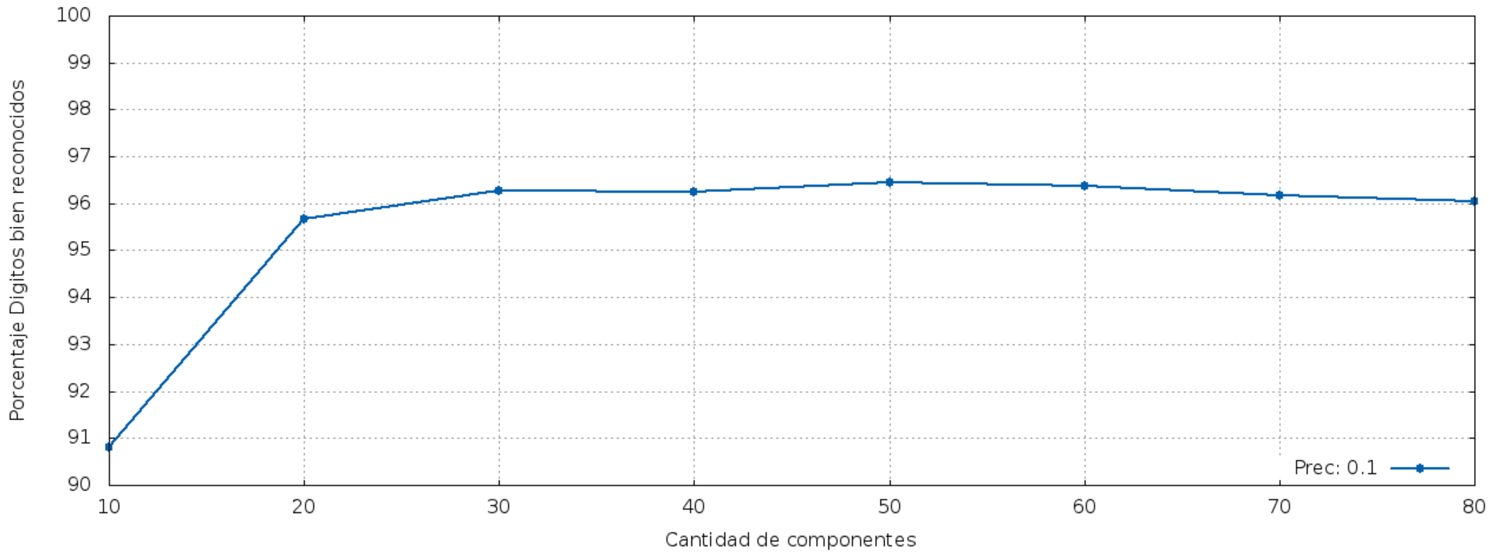
Para el método estándar de los k-vecinos[6]:



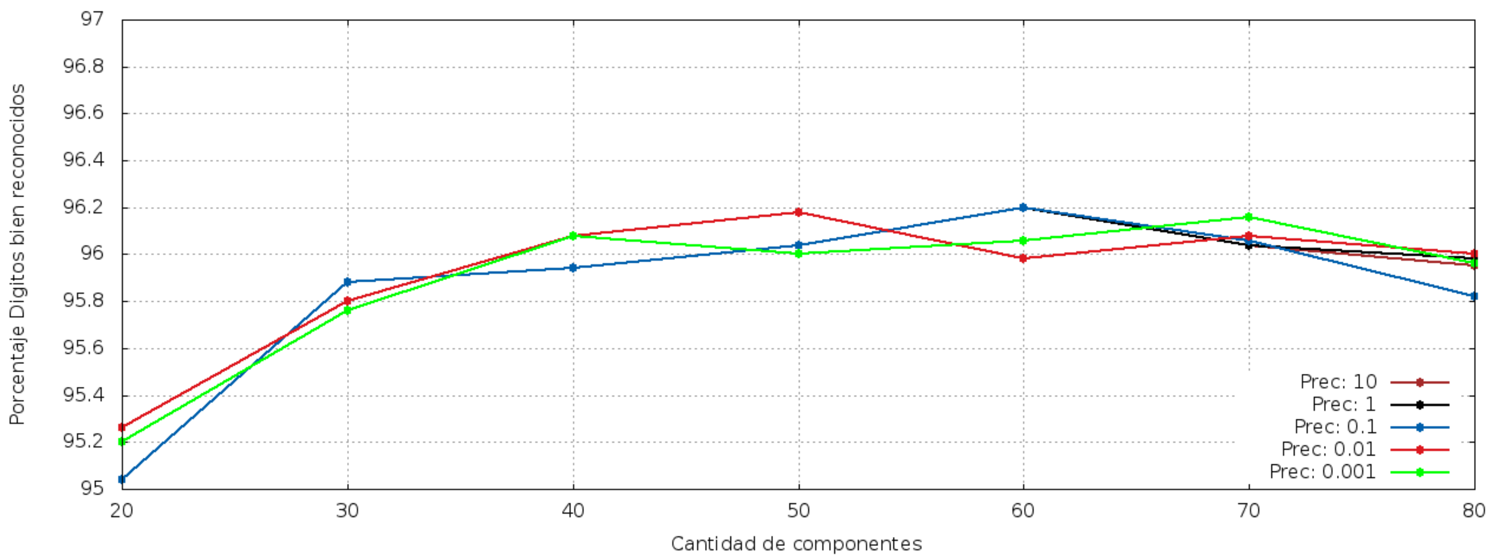
Cantidad de componentes Vs porcentaje de dígitos bien reconocidos variando cantidad de vecinos
Método de la potencia/k-vecinos con tolerancia 0,001 en instancia `primeros5000.dat`

Una vez que encontramos una cantidad de vecinos adecuada comparamos los métodos de cálculo de autovectores, variando la precisión del método de la potencia[7]:

[6] Ver Tabla C
[7] Ver Tabla D



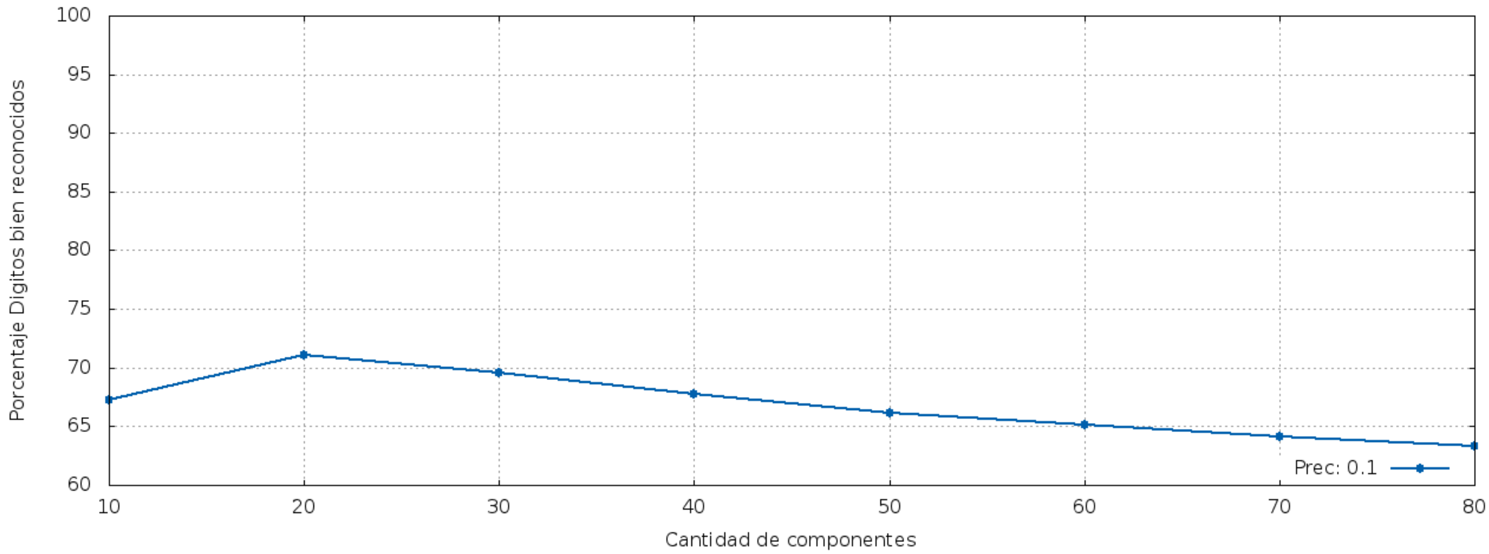
Cantidad de componentes Vs porcentaje de dígitos bien reconocidos
Método *QR-potencia Inversa*/k-vecinos con tolerancia 0,1 en instancia `primeros5000.dat`



Cantidad de componentes Vs porcentaje de dígitos bien reconocidos variando la tolerancia
Método de la potencia/k-vecinos en instancia `primeros5000.dat`

Finalmente comparamos con el comportamiento del método de la distancia al promedio utilizando *QR-potencia Inversa* para calcular autovectores[8]:

[8] Ver Tabla E



Cantidad de componentes Vs porcentaje de dígitos bien reconocidos
Método *QR-potencia Inversa*/distancia al promedio en instancia `primeros5000.dat`

D. Influencia del tamaño de la base de datos

Utilizando 5 vecinos para el método de los k-vecinos y una tolerancia de 0,001 para los métodos de reconocimiento de autovectores comparamos la cantidad de dígitos reconocidos usando un *training set* de 6000 imágenes y otro de 60000 imágenes. La cantidad de componenes utilizada fue 50 en el caso de los vecinos y 20 en el caso de la distancia al promedio de las componentes.

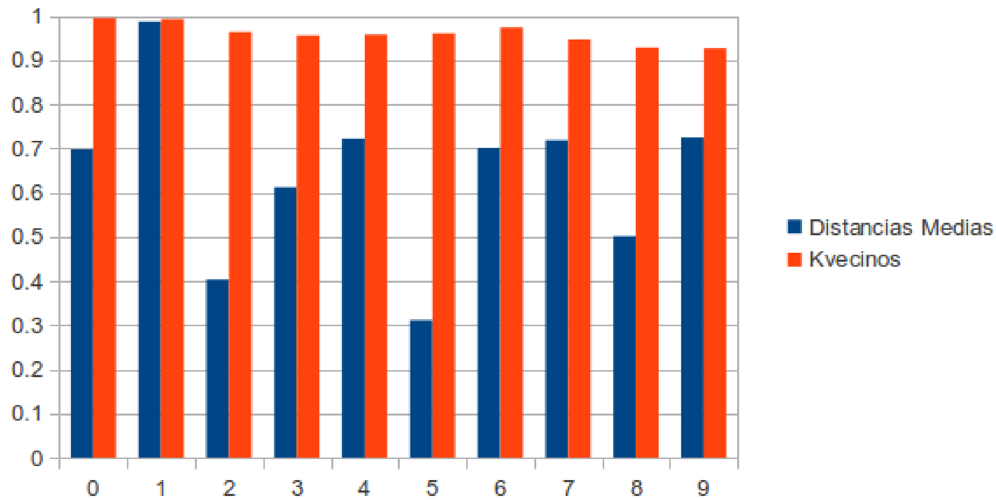
	6000	6000	60000	60000
	QR	POTENCIA	QR	POTENCIA
kvecinos	92.50%	92.20%	96.30%	96.00%
promedio	69.70%	65.50%	71.10%	66.90%

Porcentaje de dígitos bien reconocidos
variando métodos y tamaño de *training set*
Tolerancia 0,001 en instancia `primeros5000.dat`

E. Dígitos mejor reconocidos en función del método

Comparamos *hit rate* entre dígitos usando ambos metodos de reconocimiento de dígitos. Los autovectores se consiguen mediante el método *QR-potencia Inversa*[9]:

[9] Ver Tabla F



hit rate DE CADA DÍGITO
VARIANDO MÉTODO DE RECONOCIMIENTO DE DÍGITOS
Método *QR-potencia Inversa* con tolerancia 0,1 en instancia `primeros5000.dat`

IV. Discusión

A continuación discutimos cada uno de los resultados obtenidos en los tests realizados.

A. *Error de los autovectores calculados*

No nos resultó sencillo interpretar los resultados. Esperabamos ver una variación más significativa en las distancias. Deducimos que las instancias que manejamos tienen una influencia en estos resultados: se trata de matrices de covarianza con coeficientes realmente grandes (del orden de 10^4 y más). Además las matrices son considerablemente esparsas ya que las imágenes de los dígitos suelen contener muchos ceros, pues se trata de lineas sobre fondo blanco. Considerando lo anterior nos resultó razonable usar una tolerancia de 0,001 para obtener resultados óptimos en el caso del método de la potencia.

Con respecto al método *QR-potencia Inversa* llegamos a la conclusión de que tolerancias de estos órdenes no tiene influencia práctica en los resultados. Esto se debe a que el método comienza buscando una matriz semejante que sea tridiagonal. Una vez obtenida esta matriz verificamos que los elementos de la diagonal suelen ser algún orden de magnitud más grandes que los de la sub-diagonal. Tengamos en cuenta que hasta este punto no se utilizó la tolerancia ya que obtener la matriz de hessemberg es un método determinista, en el sentido de que lleva una cantidad de pasos bien definida (que depende del tamaño de la matriz). No se trata de un método iterativo.

Teniendo una diferencia realmente grande entre diagonal y sub-diagonal, al método QR, no le toma más que unas pocas iteraciones converger. En este momento la tolerancia entra en juego. Pero la convergencia del método es muy rápida y por ende habrá muy pocas iteraciones de diferencia para dos tolerancias significativamente distintas.

Al tener autovalores calculados con una muy buena precisión, el método de la potencia inversa converge casi instantaneamente[10]. Esto se debe a que resuelve un sistema de ecuaciones utilizamndo una aproximación buena de los autovalores. Al utilizar la descomposición LU para resolver el sistema buscamos mantener el error lo más acotado posible. Suponemos que esto es efectivo al menos en estas instancias, pues, como afirmamos en la sección **Resultados**, el error de los autovectores es realmente insigificante para toda tolerancia razonable.

B. *Diferencias en el tiempo de ejecución para obtener autovectores*

Notamos que, como da a entender el experimento anterior, la velocidad de convergencia del método *QR-potencia Inversa* no depende fuertemente de la precisión utilizada. Por el contrario el método de la potencia parece mostrar un comportamiento lineal en la precisión. Esto es avalado por la teoría, ya que el método de la potencia tiene, teóricamente, una convergencia lineal.

Estos resultados cumplen con nuestras expectativas, ya que el método *QR-potencia Inversa* toma un tiempo de circa un orden de magnitud más que el método de la potencia.

C. *Cantidad de dígitos reconocidos en función del método, cantidad de componentes utilizada y la precisión de los autovectores*

Para el método de los vecinos ponderados encontramos que la catitdad óptima de vecinos es 1. Esto muestra que el método no es realmente efectivo, ya que, para un solo vecino es equivalente al método de los vecinos estándar. Por este motivo concluimos que el metodo no es útil.

El método de los vecinos estándar mostró un mejor comportamiento al usar 5 vecinos. Esto concuerda con resultados discutidos con nuestros compañero de clase. Para el resto de los test utilizamos esta cantidad de vecinos.

Para el método de los vecinos encontramos que una buena cantidad de componentes principales a utilizar es 50. Como deducimos del primer test no notamos una diferencia importante en el *hit rate* cuando se varía la tolerancia usando el método de la potencia. Notamos una variación de $\sim 0{,}2\,\%$ al cambiar la precisión. Con lo cual nos convencemos de que, utilizando el método de la potencia, una tolerancia del órden de 0,001 es suficiente para conseguir resultados óptimos. Por otro lado, es importante notar que la calidad de los resultados del método de los vecinos depende más de la cantidad de componentes usada que de la cantidad de vecinos.

Finalmente analizamos el método de la distancia al promedio de las componentes principales. Si bien el método es computacionalmente más simple, ya que se compara cada dígito a reconocer únicamente con diez promedios, no muestra resultados satisfactorios, apenas logra superar el 70 % de *hits*. Notamos que para un comportamiento óptimo necesita del orden de la mitad de componentes principales que el método de los vecinos, con lo cual el método funciona mucho más rápido.

[10] Comprobamos que no suele tomar más de dos iteraciones. En la discusión del próximo experimento se hace evidente que la cantidad de iteraciones que le lleva converger no depende significativamente de la tolerancia utilizada.

D. *Cantidad de dígitos reconocidos en función del método, cantidad de componentes utilizada y la precisión de los autovectores*

Vemos que el tamaño del *training set* determina el porcentaje de dígitos bien reconocidos en todos los casos. Pero la diferencia no es abismal. Utilizando una instancia diez veces más grande logramos una mejora de circa 5 %.

Aquí también se pone en evidencia que la diferencia de precisión entre el método *QR-potencia Inversa* y el *pontencia-deflación* no implica resultados demasiado distintos.

E. *Dígitos mejor reconocidos en función del método*

Aquí se nota bien la diferencia entre los dos métodos de reconocimiento de dígitos. El método de los vecinos tiene un comportamiento casi perfecto en los dígitos 0 y 1, y para cualquier otro dígito no baja del 90 % de *hits*. Por el contrario, el método de la distancia al promedio de las componentes es mucho más inestable. Por ejemplo vemos que con el 4 y el 5 tiene un *hitrate* por debajo del 50. Este tipo de analisis puede ayudar a desarrollar software más inteligente: sabiendo en cuales dígitos es efectivo cada método, pueden usarse varios métodos simultaneamente, y tomar una decisión en funcion del *guess* de los distintos métodos.

V. Conclusiones

ANALIZAMOS dos métodos de cálculo de autovectores y dos (efectivos) de reconocimiento de dígitos.

Para los autovectores el método más preciso resultó el *QR-potencia Inversa*. Comprobamos que los autovectores suelen ser muy precisos, ya que su dirección no se ve prácticamente alterada por multiplicarlos por la matriz. El método se comporta muy bien para tolerancias relativamente grandes (en el tipo de matrices que analizamos) y no suele depender fuertemente de la tolerancia elegida. Por otro lado no es un método particularmente rápido, y, si se admite un poco de error, pueden obtenerse resultados razonables bastante más rapido con el método *potencia-deflación*. Para este caso particular notamos una diferencia en los resultados obtenidos al comparar los dos métodos de obtención de autovectores. Pero las diferencias pueden considerarse tolerables ya son del órden de $\sim 0{,}5\,\%$ de *hit rate* para ambos métodos de reconocimiento de dígitos.

Con respecto al reconocimiento de dígitos creemos que el método del promedio (al menos nuestra implementación) no es suficientemente buena para ninguna aplicación práctica. En el mejor caso obtuvimos un *hit rate* del $\sim 71\,\%$. A lo sumo puede combinarse con un método tamás poderoso (como el de los vecinos) para hacer un *double check* en el caso en el que el de los vecinos no llegue a un resultado concluyente [11]. El método de los vecinos resultó ser muy satisfactorio. Su mayor deficiencia está en los dígitos 8 y 9 como mostramos en el último test.

De los resultados concluimos que el parámetro más sensible para tener un bien *hit rate* es la cantidad de componentes principales utilizadas pues, tanto la precisión de los autovectores, como la cantidad de vecinos del método de los vecinos, influyen de manera mucho menos notable en la cantidad de dígitos acertados.

[11] Por ejemplo, si los vecinos contienen cantidades similares de dígitos a y dígitos b, $a \neq b$.

Referencias

[1] Cátedra de Métodos numéricos,
Tercer Trabajo Práctico,
Primer cuatrimestre 2013

[2] Richard L. Burden,
Numerical Analysis,
9th ed.

VI. Apéndices

A. Enunciado

Laboratorio de Métodos Numéricos - Primer Cuatrimestre 2013
Trabajo Práctico Número 3: OCR+SVD

Introducción

El reconocimiento óptico de caracteres (OCR, por sus siglas en inglés) es el proceso por el cual se traducen o convierten imágenes de dígitos o caracteres (sean éstos manuscritos o de alguna tipografía especial) a un formato representable en nuestra computadora (por ejemplo, ASCII). Esta tarea puede ser más sencilla (por ejemplo, cuando tratamos de determinar el texto escrito en una versión escaneada a buena resolución de un libro) o tornarse casi imposible (recetas indescifrables de médicos, algunos parciales manuscritos de alumnos de métodos numéricos, etc).

El objetivo del trabajo práctico es implementar un método de reconocimiento de dígitos manuscritos basado en la descomposición en valores singulares, y analizar empíricamente los parámetros principales del método.

Como instancias de entrenamiento, se tiene un conjunto de n imágenes de dígitos manuscritos en escala de grises del mismo tamaño y resolución (varias imágenes de cada dígito). Cada una de estas imágenes sabemos a qué dígito se corresponde. En este trabajo consideraremos la popular base de datos MNIST, utilizada como referencia en esta área de investigación[12].

Para $i = 1, \ldots, n$, sea $x_i \in \mathbb{R}^m$ 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 $X \in \mathbb{R}^{n\times m}$ como la matriz que contiene en la i-ésima fila al vector $(x_i - \mu)^t/\sqrt{n-1}$, y

$$X = U\Sigma V^t$$

a su descomposición en valores singulares, con $U \in \mathbb{R}^{n\times n}$ y $V \in \mathbb{R}^{m\times m}$ matrices ortogonales, y $\Sigma \in \mathbb{R}^{n\times m}$ la matriz diagonal conteniendo en la posición (i,i) al i-ésimo valor singular σ_i. Siendo v_i la columna i de V, definimos para $i = 1, \ldots, n$ la *transformación característica* del dígito x_i como el vector $\mathbf{tc}(x_i) = (v_1^t\, x_i, v_2^t\, x_i, \ldots, v_k^t\, x_i) \in \mathbb{R}^k$, donde $k \in \{1, \ldots, m\}$ es un parámetro de la implementación. Este proceso corresponde a extraer las k primeras *componentes principales* de cada imagen. La intención es que $\mathbf{tc}(x_i)$ resuma la información más relevante de la imagen, descartando los detalles o las zonas que no aportan rasgos distintivos.

Dada una nueva imagen x de un dígito manuscrito, que no se encuentra en el conjunto inicial de imágenes de entrenamiento, el problema de reconocimiento consiste en determinar a qué dígito corresponde. Para esto, se calcula $\mathbf{tc}(x)$ y se compara con $\mathbf{tc}(x_i)$, para $i = 1, \ldots, n$.

Enunciado

Se pide implementar un programa que lea desde archivos las imágenes de entrenamiento de distintos dígitos manuscritos y que, utilizando la descomposición en valores singulares, se calcule la transformación característica de acuerdo con la descripción anterior. Para ello se deberá implementar

[12] `http://yann.lecun.com/exdb/mnist/`

algún método de estimación de autovalores/autovectores. Dada una nueva imagen de un dígito manuscrito, el programa deberá determinar a qué dígito corresponde. El formato de los archivos de entrada y salida queda a elección del grupo. Si no usan un entorno de desarrollo que incluya bibliotecas para la lectura de archivos de imágenes, sugerimos que utilicen imágenes en formato Raw.

Se deberán realizar experimentos para medir la efectividad del reconocimiento, analizando tanto la influencia de la cantidad k de componentes principales seleccionadas como la influencia de la precisión en el cálculo de los autovalores.

Fecha de entrega

- *Formato electrónico:* viernes 21 de junio de 2013, hasta las 23:59 hs., enviando el trabajo (informe+código) a `metnum.lab@gmail.com`. El subject del email debe comenzar con el texto `[TP3]` seguido de la lista de apellidos de los integrantes del grupo.
- *Formato físico:* lunes 24 de junio de 2013, de 18 a 20hs (en la clase de la práctica).

B. Código relevante y modo de uso

B.1 Conversión de datos usando Matlab

Las instancias utilizadas fueron bajadas de `http://yann.lecun.com/exdb/mnist/`. Debe utilizarse el *script* `generarPuntoDat.sh` que se encuentra en `imagenes/MNIST_Matlab` para convertir las instancias bajadas a archivos legibles por nuestra implementación. Al ejecutar el *script* se verá el modo de uso.

B.2 Modo de uso del programa

Al ejecutar el programa `tp3.exe` que se encuentra en `ejecutables/` se imprimen las instrucciones de uso.

B.3 Generar matrices de covarianza

Puede usarse el programa `guardarMatrizCovarianza.exe` para guardar una matriz de covarianza ya calculada y usarla posteriormente.

B.4 Clase *Reconocedor*

Esta clase permite utilizar cualquiera de los métodos para el cálculo de autovalores y combinarlo con cualquiera de los método de reconocimiento de dígitos.

```
#include "reconocedor.h"

#define TAMANO_IMAGEN 784
#define MIN_SIGNIFICATIVO 0.01f

using namespace std;

// PARA COMPARAR TUPLAS POR LA SEGUNDA COMPONENTE
bool compararTupla ( pair<int,double> i, pair<int,double> j ) { return ( i.second < j.second
    ); }


Reconocedor::Reconocedor( char *puntoDat )
{
      instanciaAbierta = false;
      tcsCalculados = false;

      int scan;
      FILE *data = fopen( puntoDat, "r" );

      scan = fscanf( data, "%d\n", &cantidad );

      imagenes = new Matriz<double>( cantidad, TAMANO_IMAGEN );
      labels = new int[cantidad];
      int numerito;

      for ( int i=0 ; i<cantidad ; ++i ) {
            scan = fscanf( data, "%d ", &numerito );
            labels[i] = numerito;
      }

      for ( int i=0 ; i<cantidad ; ++i ) {
```

```
            for ( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
                scan = fscanf( data, "%d ", &numerito );
                (*imagenes)[i][j] = (double) numerito;
            }
        }

        fclose( data );

        Matriz<double> X (*imagenes);


        // tomamos promedio
        double promedio[TAMANO_IMAGEN];
        // arreglo limpio
        memset( promedio, 0, sizeof(double)*TAMANO_IMAGEN );

        for ( int i=0 ; i<cantidad ; ++i )
            for ( int j=0 ; j<TAMANO_IMAGEN ; ++j )
                promedio[j] += (*imagenes)[i][j];

        double cant = (double) TAMANO_IMAGEN;

        for ( int i=0 ; i<TAMANO_IMAGEN ; ++i )
            promedio[i] /= cant;

        double denominador = sqrt( cant - 1.f );

        // calculo matriz X

        for ( int i=0 ; i<cantidad ; ++i )
            for ( int j=0 ; j<TAMANO_IMAGEN ; ++j )
                (*imagenes)[i][j] = ((*imagenes)[i][j] - promedio[j]) / denominador;

        covarianza = new Matriz<double> (TAMANO_IMAGEN,TAMANO_IMAGEN);
        covarianza->cargarTranspuestaPorMat(X);


        promediosCalculados = false;

        cantAutovectores = 0;
}

Reconocedor::Reconocedor( char *puntoDat, char *matCovarianza )
{
        instanciaAbierta = false;
        tcsCalculados = false;

        int scan;
        FILE *data = fopen( puntoDat, "r" );

        scan = fscanf( data, "%d\n", &cantidad );

        imagenes = new Matriz<double>( cantidad, TAMANO_IMAGEN );
        labels = new int[cantidad];

        int numerito;

        for ( int i=0 ; i<cantidad ; ++i ) {
            scan = fscanf( data, "%d ", &numerito );
            labels[i] = numerito;
        }

        for ( int i=0 ; i<cantidad ; ++i ) {
            for ( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
                scan = fscanf( data, "%d ", &numerito );
                (*imagenes)[i][j] = (double) numerito;
            }
```

```
	}

	fclose( data );


	//matriz de covarianza

	FILE *dataCov = fopen( matCovarianza, "r" );

	int dim1, dim2;
	scan = fscanf( dataCov, "%d %d", &dim1, &dim2 );

	assert( dim1==TAMANO_IMAGEN && dim2==TAMANO_IMAGEN );

	covarianza = new Matriz<double> (dim1, dim2);

	double numeritoDoubel;
	for ( int i=0 ; i<TAMANO_IMAGEN; ++i ) {
		for ( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
			scan = fscanf( dataCov, "%lf ", &numeritoDoubel );
			(*covarianza)[i][j] = numeritoDoubel;
		}
	}

	fclose( dataCov );

	promediosCalculados = false;

	cantAutovectores = 0;
}

Reconocedor::~Reconocedor()
{
	delete imagenes;
	delete[] labels;

	delete covarianza;

	if (cantAutovectores>0) delete autovectores;

	if (tcsCalculados) delete tcs;

	if (instanciaAbierta) {
		delete aEvaluar;
		delete[] labels_aEvaluar;
	}

	if (promediosCalculados)
		delete promediosTcs;
}

void Reconocedor::guardarCovarianza( char *nombre )
{
	FILE *guardar = fopen(nombre, "w");

	//tamano de la matriz
	fprintf(guardar, "%d %d\n", covarianza->cantFil(), covarianza->cantCol());

	//coeficientes de la matriz
	for ( uint i=0 ; i<covarianza->cantFil() ; ++i ) {
		for ( uint j=0 ; j<covarianza->cantCol() ; ++j )
			fprintf(guardar, "%d ", (int) (*covarianza)[i][j]);

		fprintf(guardar, "\n");
	}

	fclose(guardar);
```

```
}

void Reconocedor::abrir_instancia_a_evaluar( char *archivo, int primero, int ultimo )
{
	FILE *data = fopen( archivo, "r" );
	int scan;
	scan = fscanf( data, "%d\n", &cantidad );

	if ( ultimo > cantidad ) {
		printf("pediste mas indices de los que hay, esto va a explotar\n");
		return;
	}

	int cuantosPidio = ultimo - primero + 1;

	aEvaluar = new Matriz<double>( cuantosPidio, TAMANO_IMAGEN );
	labels_aEvaluar = new int[cuantosPidio];

	int numerito;

	int indice = 0;
	for ( int i=0 ; i<cantidad ; ++i ) {
		scan = fscanf( data, "%d ", &numerito );

		if ( i+1 >= primero && i+1 <= ultimo ) {
			labels_aEvaluar[indice] = numerito;
			indice++;
		}
	}
	indice = 0;
	for ( int i=0 ; i<cantidad ; ++i ) {
		for ( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
			scan = fscanf( data, "%d ", &numerito );

			if ( i+1 >= primero && i+1 <= ultimo ) {
				(*aEvaluar)[indice][j] = (double) numerito;
			}
		}
		if ( i+1 >= primero && i+1 <= ultimo )
			indice++;
	}

	fclose( data );

	instanciaAbierta = true;

	calcular_tcs();
}

void Reconocedor::calcularAutovectores_QR( int maxIterQR, int maxIterInvPotencia, double
    tolerancia, int cuantosAutovec )
{

//	printf("Me pidieron esta tolerancia: %f\n", tolerancia);

	if ( (uint) cuantosAutovec > covarianza->cantFil() ) {
		printf("pediste mas autovalores que la cantidad de dimensiones de la matriz, esto
		     va a explotar\n");
		return;
	}

	cantAutovectores = cuantosAutovec;

	autovectores = new Matriz<double> (TAMANO_IMAGEN, cantAutovectores);

	Matriz<double> temp (*covarianza);
	//temp.contieneNaN();
```

```
	temp.Householder( tolerancia );
	//temp.contieneNaN();

	vector<double> autoval;
	int iter = maxIterQR;
	temp.QR( tolerancia, iter, autoval );

	//ya tengo los autovalores
	sort(autoval.begin(),autoval.end() );

//	for ( int i=0 ; i<autoval.size() ; ++i ) {
//		printf("%f\n", autoval[i]);
//	}

	Matriz<double> x(TAMANO_IMAGEN, 1);

	for (int i=0 ; i<cuantosAutovec ; ++i ) {
		// agarro los autovalores de menor a mayor!
		double autovalActual = autoval[autoval.size()-i-1];

		if ( fabs(autovalActual) < MIN_SIGNIFICATIVO ) break;

//		printf("%f %f\n", autovalActual, autoval[autoval.size()-i-2]);
//		__BITACORA


		// le paso un autovector que este lejos de 0
		for( int j=0 ; j<TAMANO_IMAGEN; ++j ) {
			x[j][0] = 1.f;
		}

		covarianza->potenciaInversa(autovalActual, x, tolerancia, maxIterInvPotencia);
x.normalizar();

		// lo guardo en mi matriz de autovectores
		for( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
			(*autovectores)[j][i] = x[j][0];
		}
	}
}

void Reconocedor::calcularAutovectores_potencia( int maxIterPotSim, double tolerancia, int cuantosAutovec )
{
	if ( (uint) cuantosAutovec > covarianza->cantFil() ) {
		printf("pediste mas autovalores que la cantidad de dimensiones de la matriz, esto va a explotar\n");
		return;
	}

	double autovalActual;

	autovectores = new Matriz<double> (TAMANO_IMAGEN, cuantosAutovec);

	Matriz<double> autovectorActual(TAMANO_IMAGEN, 1);

	for( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
		autovectorActual[j][0] = 1.f;
	}


	for (int i=0 ; i<cuantosAutovec ; ++i ) {

		covarianza->potenciaSimple(autovalActual,autovectorActual,tolerancia, maxIterPotSim);

		if ( fabs(autovalActual) < MIN_SIGNIFICATIVO )
```

```
                break;
            else
                cantAutovectores++;     //calculo un autovector


            // lo guardo en mi matriz de autovectores
            for( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
                (*autovectores)[j][i] = autovectorActual[j][0];
            }

            covarianza->deflacion(autovalActual,autovectorActual);
        }
}

int Reconocedor::reconocer_kVecinos( int cantComponentes, int k, int indice_imagen )
{
        if (!instanciaAbierta) {
            printf("No se elegio instancia a evaluar\n");
            return -1;
        }

        if ( (uint) indice_imagen > aEvaluar->cantFil() ) {
            printf("pediste una imagen fuera de rango\n");
            return -1;
        }

        if ( cantComponentes > cantAutovectores ) {
            printf("pediste mas componenes principales que la cantidad de autovectores que
                calculaste\n");
            return -1;
        }

        //me lo dan partiendo de 1
        //pero yo lo uso desde 0
        indice_imagen--;

        int cantIm = imagenes->cantFil();

        vector< pair<int,double> > distancias;

        Matriz<double> resta(cantComponentes,1);

        for ( int i=0 ; i<cantIm ; ++i ) {

            for( int j=0 ; j<cantComponentes ; ++j )
                resta[j][0] = (*tcs_aEvaluar)[indice_imagen][j] - (*tcs)[i][j];

            double distancia = resta.norma();

            pair<int,double> labelDistancia;
            labelDistancia.first = labels[i];
            labelDistancia.second = distancia;

//          printf("label %d, a distancia %f\n", labelDistancia.first, labelDistancia.second)
    ;

            distancias.push_back(labelDistancia);
        }


        sort( distancias.begin(), distancias.end(), compararTupla );

        vector<int> frecuencias(10, 0);

        for (int i=0 ; i<k ; ++i ) {
            frecuencias[ distancias[i].first ]++;
        }
```

```

      int mejor = 0;
      int digito = -1;

      for (int i=0 ; i<10 ; ++i ) {
//          printf("%d ", frecuencias[i]);
            if ( frecuencias[i] > mejor ) {
                  mejor = frecuencias[i];
                  digito = i;
            }
      }

      return digito;
}

int Reconocedor::reconocer_distanciaMedia( int cantComponentes, int indice_imagen )
{
      if (!instanciaAbierta) {
            printf("No se elegio instancia a evaluar\n");
            return -1;
      }

      if ( (uint) indice_imagen > aEvaluar->cantFil() ) {
            printf("pediste una imagen fuera de rango\n");
            return -1;
      }

      if ( cantComponentes > cantAutovectores ) {
            printf("pediste mas componenes principales que la cantidad de autovectores que
                calculaste\n");
            return -1;
      }

      indice_imagen--;

      int cantIm = imagenes->cantFil();

      vector<double> distanciasMedias(10, 0.f);
      vector<int> cantApariciones(10, 0);

      Matriz<double> resta(cantComponentes,1);

      for ( int i=0 ; i<cantIm ; ++i ) {

            for( int j=0 ; j<cantComponentes ; ++j )
                  resta[j][0] = (*tcs_aEvaluar)[indice_imagen][j] - (*tcs)[i][j];

            double distancia = resta.norma();

            //aparecion un a vez mas
            cantApariciones[ labels[i] ]++;
            //con esta distancIA
            distanciasMedias[ labels[i] ] += distancia;
      }


      double mejor = INFINITY;
      int digito = -1;

      for (int i=0 ; i<10 ; ++i ) {
//          printf("%d ", frecuencias[i]);

            double mediaActual = distanciasMedias[i]/(double)cantApariciones[i];
            if ( mediaActual < mejor ) {
                  mejor = mediaActual;
                  digito = i;
            }
```

```
	}

	return digito;
}

int Reconocedor::reconocer_kVecinosPonderados( int cantComponentes, int k, int indice_imagen )
{
	if (!instanciaAbierta) {
		printf("No se elegio instancia a evaluar\n");
		return -1;
	}

	if ( (uint) indice_imagen > aEvaluar->cantFil() ) {
		printf("pediste una imagen fuera de rango\n");
		return -1;
	}

	if ( cantComponentes > cantAutovectores ) {
		printf("pediste mas componenes principales que la cantidad de autovectores que calculaste\n");
		return -1;
	}

	//me lo dan partiendo de 1
	//pero yo lo uso desde 0
	indice_imagen--;

	int cantIm = imagenes->cantFil();

	//first LABEL
	//second DISTANCIA
	vector< pair<int,double> > distancias;

	Matriz<double> resta(cantComponentes,1);
	for ( int i=0 ; i<cantIm ; ++i ) {

		for( int j=0 ; j<cantComponentes ; ++j )
			resta[j][0] = (*tcs_aEvaluar)[indice_imagen][j] - (*tcs)[i][j];

		double distancia = resta.norma();

		pair<int,double> labelDistancia;
		labelDistancia.first = labels[i];
		labelDistancia.second = distancia;

		distancias.push_back(labelDistancia);
	}


	sort( distancias.begin(), distancias.end(), compararTupla );

	vector<int> frecuencias(10, 0);
	vector<double> distanciasMedias(10, 0.f);

	for (int i=0 ; i<k ; ++i ) {
		int labelActual = distancias[i].first;

		frecuencias[ labelActual ]++;
		distanciasMedias[ labelActual ] += distancias[i].second;
	}

	double mejor = INFINITY;
	int digito = -1;

	for (int i=0 ; i<10 ; ++i ) {

```

```
//          printf("%d ", frecuencias[i]);
            double mediaActual = (frecuencias[i]==0)? INFINITY : distanciasMedias[i]/(double)frecuencias[i];

            if ( mediaActual < mejor ) {
                  mejor = mediaActual;
                  digito = i;
            }
      }

      return digito;
}

int Reconocedor::reconocer_digitoMedio( int cantComponentes, int indice_imagen )
{
      if (!instanciaAbierta) {
            printf("No se elegio instancia a evaluar\n");
            return -1;
      }

      if (!promediosCalculados) {
            printf("No promediaste antes de llamar a la funcion, esto va a explotar\n");
            return -1;
      }

      if ( (uint) indice_imagen > aEvaluar->cantFil() ) {
            printf("pediste una imagen fuera de rango\n");
            return -1;
      }

      if ( cantComponentes > cantAutovectores ) {
            printf("pediste mas componenes principales que la cantidad de autovectores que calculaste\n");
            return -1;
      }

      indice_imagen--;

      vector<double> distanciasMedias(10, 0.f);

      Matriz<double> resta(cantComponentes,1);

      for ( int i=0 ; i<10 ; ++i ) {
            for( int j=0 ; j<cantComponentes ; ++j ) {
                  resta[j][0] = (*tcs_aEvaluar)[indice_imagen][j] - (*promediosTcs)[i][j];
            }

            double distancia = resta.norma();

            //con esta distancIA
            distanciasMedias[i] = distancia;
      }


      double mejor = INFINITY;
      int digito = -1;
      for (int i=0 ; i<10 ; ++i ) {

            double mediaActual = distanciasMedias[i];

            //printf("%f ", distanciasMedias[i]);
            if ( mediaActual < mejor ) {
                  mejor = mediaActual;
                  digito = i;
            }
      }
      //printf("\n");
```

```

	return digito;
}

void Reconocedor::promediarTcs( int cantComponentes )
{
	promediosTcs = new Matriz<double> (10, cantComponentes);

	//inicializo
	for ( uint i=0 ; i<10 ; ++i )
		for ( int j=0 ; j<cantComponentes ; ++j )
			(*promediosTcs)[i][j] = 0.f;


	vector<int> apariciones(10, 0);

	int cuantasImagenes = tcs->cantFil();

	for ( int i=0 ; i<cuantasImagenes ; ++i ) {
		for ( int j=0 ; j<cantComponentes ; ++j ) {
			(*promediosTcs)[ labels[i] ][j] += (*tcs)[i][j];
			apariciones[ labels[i] ]++;
		}
	}


	for ( int i=0 ; i<10 ; ++i ) {
		for ( int j=0 ; j<cantComponentes ; ++j ) {
			if (apariciones[i] == 0) {
				(*promediosTcs)[i][j] = INFINITY;
			} else {
				(*promediosTcs)[i][j] /= apariciones[i];
			}
		}
	}

	promediosCalculados = true;
}

void Reconocedor::calcular_tcs()
{
	int cantIm = imagenes->cantFil();
	int componentes = cantAutovectores;

	tcs = new Matriz<double>( cantIm, componentes );

	int cantImEval = aEvaluar->cantFil();

	tcs_aEvaluar = new Matriz<double>( cantImEval, componentes );

	tcs->cargarMultiplicacion( *imagenes, *autovectores );
	tcs_aEvaluar->cargarMultiplicacion( *aEvaluar, *autovectores );

	tcsCalculados = true;
}
```

B.5 Clase *Matriz*

```
#ifndef __MATRIZ_H
#define __MATRIZ_H

#include "master_header.h"

#define COMPARAR_DOUBLE(a, b) (fabs( (a) - (b) ) < 1e-9)
#define SIGNO_DOUBLE( a ) ((fabs( (a) ) > 1e-9) ? (((a) > 0) ? 1.f : -1.f) : 1.f)
```

```

template< typename T >
class Matriz
{
	private:
		uint n,m;
		uint triangulada;

	public:
		T ** matriz;

		Matriz(int _n, int _m) : n(_n),m(_m)
		{
			matriz = new T*[n];
			for( uint i = 0; i < n; ++i)
				matriz[i] = new T[m];
			triangulada = 0;
		}

		Matriz( Matriz<T> &B )
		{
			n = B.n;
			m = B.m;
			triangulada = 0;
			uint i,j;

			matriz = new T*[n];

			for( i=0 ; i<n ; ++i )
				matriz[i] = new T[m];

			for ( i=0 ; i<n ; ++i )
				for ( j=0 ; j<m ; ++j )
					matriz[i][j] = B[i][j];
		}


		~Matriz()
		{
			for(uint i = 0; i < n; ++i)
				delete [] matriz[i];

			delete [] matriz;
		}

		uint cantFil() { return n; }
		uint cantCol() { return m; }

		T* operator[](uint i)
		{
			if(i >= n) return NULL;
			return matriz[i];
		}

		void cargarSuma( Matriz<T> &A, Matriz<T> &B)
		{
			assert( n==A.n && m==A.m && A.n==B.n && A.m==B.m );

			uint i,j;
			for ( i=0 ; i<n ; ++i )
				for ( j=0 ; j<m ; ++j )
					matriz[i][j] = A[i][j] + B[i][j];
		}

		void cargarResta( Matriz<T> &A, Matriz<T> &B)
		{
			assert( n==A.n && m==A.m && A.n==B.n && A.m==B.m );
```

```

		uint i,j;
		for ( i=0 ; i<n ; ++i )
			for ( j=0 ; j<m ; ++j )
				matriz[i][j] = A[i][j] - B[i][j];
	}

	void restarle( Matriz<T> &A )
	{
		assert( n==A.n && m==A.m );

		uint i,j;
		for ( i=0 ; i<n ; ++i )
			for ( j=0 ; j<m ; ++j )
				matriz[i][j] -= A[i][j];
	}


	void cargarMultiplicacion( Matriz<T> &A, Matriz<T> &B )
	{
		assert( n==A.n && A.m==B.n && B.m==m );

		uint i,j,k;

		for ( i=0 ; i<n ; ++i ) {
			for ( j=0 ; j<m ; ++j ) {
				double sum = 0;
				for ( k=0 ; k<A.m ; ++k )
					sum += (double) A[i][k] * (double) B[k][j];

				matriz[i][j] = (T) sum;
			}
		}
	}

	void cargarTranspuestaPorMat(Matriz<T> &A )
	{
		uint i,j,k;
		for ( i=0 ; i<A.m ; ++i ) {
			for ( j=0 ; j<A.m ; ++j ) {
				double sum = 0.f;
				for ( k=0 ; k<A.n ; ++k )
					sum += (double) A[k][i] * (double) A[k][j];

				matriz[i][j] = (T) sum;
			}
		}
	}

	void transponer( Matriz<T> &B )
	{
		assert( n==B.m && m==B.n );

		uint i,j;

		for ( i=0 ; i<n ; ++i )
			for ( j=0 ; j<m ; ++j )
				matriz[i][j] = B[j][i];
	}

	void imprimirMatriz()
	{
		uint i,j;

		for ( i=0 ; i<n ; ++i ) {
			for ( j=0 ; j<m ; ++j )
				printf("%f\t", (double) matriz[i][j]);
```

```

                        printf("\n");
                }

                printf("\n");
        }

        void contieneNaN()
        {
                uint i,j;

                for ( i=0 ; i<n ; ++i )
                        for ( j=0 ; j<m ; ++j )
                                assert( !(matriz[i][j]!=matriz[i][j]));
        }

        void guardarArchivo(char * archivo)
        {       //DEBUG - NO ME SAQUES

                FILE *nuevo = fopen( archivo , "w" );

                int i;

                fprintf( nuevo, "%d\n", n );

                for ( i=0 ; i<n ; ++i )
                        fprintf( nuevo, "%f ", matriz[i][0] );

                fprintf( nuevo, "\n" );

                fclose( nuevo );
        }

        void Householder ( double tol )
        {       // la matriz de entrada es la que llama a la funcion, se llama matriz
                std::vector<double> v(n, 0.f);
                std::vector<double> u(n, 0.f);
                std::vector<double> z(n, 0.f);
                std::vector<double> y(n, 0.f);

                for (uint k = 0; k < n-2 ; k++)      // reveer los indices hasta donde van
                {
                        double q = 0.f;
                        for (uint j=k+1; j<n ; j++) {
                                q += ( (matriz[j][k])*(matriz[j][k]) );
                        }

                        // si la columna son todos ceros,
                        // no hay nada que hacer
                        // paso a la proxima
                        if( COMPARAR_DOUBLE(q, 0.f) ) continue;

                        // ahora q es la sumatoria

                        double alfa = - sqrt(q) * SIGNO_DOUBLE(matriz[k+1][k]);

                        double rsq = alfa*alfa - alfa * matriz[k+1][k];

                        v[k] = 0.f;
                        v[k+1] = matriz[k+1][k] - alfa;

                        for (uint j = k+2; j<n; j++)
                                v[j] = matriz[j][k];

                        double suma;
                        for (uint j=k; j<n ; j++)
                        {
```

```
                suma=0.f;
                for (uint i = k+1; i<n ; i++)
                    suma += matriz[j][i] * v[i];

                u[j] = (1.f/rsq) * suma;
            }

            double prod = 0.f;
            for (uint i=k+1; i<n ; i++)
                prod += v[i] * u[i];

            for (uint j=k; j<n ; j++)
                z[j] = u[j] - (prod/(2.f*rsq)) * v[j];

            for (uint l=k+1; l<n-1 ; l++)
            {
                for (uint j=l+1; j<n ; j++)
                {
                    matriz[j][l] = matriz[j][l] - v[l] * z[j] - v[j] * z[l];
                    matriz[l][j] = matriz[j][l];
                }

                matriz[l][l] = matriz[l][l] - 2.f * v[l] * z[l];
            }

            matriz[n-1][n-1] = matriz[n-1][n-1] - 2.f * v[n-1] * z[n-1];

            for (uint j=k+2; j<n ; j++)
            {
                matriz[k][j] = 0.f;
                matriz[j][k] = 0.f;
            }

            matriz[k+1][k] = matriz[k+1][k] - v[k+1] * z[k];
            matriz[k][k+1] = matriz[k+1][k];
        }
    }

    void submatriz(int desde1, int hasta1, int desde2, int hasta2, Matriz<T> &salida)
    {
        for (int i = desde1-1; i < hasta1 ; i++)        //hago desde-1 para que se corresponda con los indices de la bibliografia
            for (int j = desde2-1; j < hasta2; j++)
                salida[i-(desde1-1)][j-(desde2-1)] = matriz [i][j];
    }

    void dameDiagonales( std::vector<double> &As, std::vector<double> &Bs )
    {
        As.push_back( matriz[0][0] );
        Bs.push_back( 0.f );

//      printf("%f\n",matriz[0][0] );

        for ( uint i=1 ; i<n ; ++i ) {
            As.push_back(matriz[i][i]);
            Bs.push_back(matriz[i][i-1]);
//          printf("%f\t%f\n",matriz[i][i], matriz[i][i-1]);
        }
    }


    void QR( double tol, int maxIter, std::vector<double> &autoval )
    {
        std::vector<double> As;
        std::vector<double> Bs;

        dameDiagonales(As,Bs);
```

```
        double shift = 0.f;

        QR_rec( As, Bs, tol, maxIter, autoval, shift );
    }

    void QR_rec( std::vector<double> &As, std::vector<double> &Bs, double tol, int &
        maxIter, std::vector<double> &autoval, double shift )
    {
        int tam = As.size();

        //#if DEBUG
        //printf("llamado con diag de tamano: %d\n", n);
        //#endif

        std::vector<double> Cs(tam, 0.f);   // cosetamos
        std::vector<double> Ds(tam, 0.f);
        std::vector<double> Qs(tam, 0.f);
        std::vector<double> Rs(tam, 0.f);
        std::vector<double> Ss(tam, 0.f);   // senos
        std::vector<double> Xs(tam, 0.f);
        std::vector<double> Ys(tam, 0.f);
        std::vector<double> Zs(tam, 0.f);

        while ( maxIter > 0 ) {

            if (tam==0) return;


            // busco la seguidilla mas larga de ceros
            // en la subdiagonal empezando desde el final

            while ( fabs(Bs[tam-1])<tol && tam>0 ) {
                // el ultimo b es suficientemente chico, tengo un autoval
                autoval.push_back( As[tam-1]+shift );
                tam--;
            }

            if (tam==0) return;

            // busco la seguidilla mas larga de ceros
            // en la subdiagonal empezando desde el principio

            int inicio = 1;
            while ( fabs(Bs[inicio]) < tol && inicio<tam) {
                // el primer b es suficientemetne chico, tengo un autoval
                autoval.push_back( As[inicio-1]+shift );
                inicio++;
            }
            if (inicio==tam) return;

            //////// reacomodo
            inicio--;
            if ( inicio>0 ) {
                for ( int j=0; j<tam-inicio ; ++j ) {
                    As[j] = As[j+inicio];
                    Bs[j] = Bs[j+inicio];
                }
            }
            tam -= inicio;


            ////////// casos Base
            if ( tam==0 ) return;

            if ( tam==1 ) {
                autoval.push_back( As[0] + shift );
```

```
        return;
    }

    //////// posible llamado recursivo
    for ( int j=2 ; j<tam-1 ; ++j ) {
        if ( fabs(Bs[j]) < tol ) {

            //si tengo un numero pequeno parto la matrz
            //para seguir teniendo una convergencia rapida

            //construimos los As y Bs para el llamado recursivo
            std::vector<double> As1;
            std::vector<double> Bs1;
            std::vector<double> As2;
            std::vector<double> Bs2;

            for ( int k=0 ; k<tam ; ++k ) {

                if ( k<j-1 ) {
                    As1.push_back(As[k]);
                    Bs1.push_back(Bs[k]);
                } else {
                    As2.push_back(As[k]);
                    Bs2.push_back(Bs[k]);
                }
            }

            QR_rec( As1, Bs1, tol, maxIter, autoval, shift );
            QR_rec( As2, Bs2, tol, maxIter, autoval, shift );

            return;
        }
    }


    ///////// iteracion QR
    double b = -(As[tam-2] + As[tam-1]);
    double c = As[tam-1]*As[tam-2] - Bs[tam-1]*Bs[tam-1];
    double d = sqrt(b*b - 4.f*c);

    double mu1, mu2;
    if ( !COMPARAR_DOUBLE(b,0.f) && b>0.f ) {
        mu1 = -2.f*c / (b+d);
        mu2 = -(b+d)/2.f;
    } else {
        mu1 = (d-b)/2.f;
        mu2 = 2.f*c / (d-b);
    }

    //caso Base n=2
    if ( tam==2 ) {
        autoval.push_back( mu1 + shift );
        autoval.push_back( mu2 + shift );
        return;
    }

    double sigma;
    sigma = (fabs(mu1-As[tam-1]) < fabs(mu2-As[tam-1])) ? mu1 : mu2;

    shift += sigma;

    for ( int j=0 ; j<tam ; ++j )
        Ds[j] = As[j] - sigma;

    Xs[0] = Ds[0];
    Ys[0] = Bs[1];

```

```
                        for ( int j=1 ; j<tam ; ++j ) {
                              Zs[j-1] = sqrt( Xs[j-1]*Xs[j-1] + Bs[j]*Bs[j]);

                              Cs[j] = Xs[j-1] / Zs[j-1];

                              Ss[j] = Bs[j] / Zs[j-1];

                              Qs[j-1] = Cs[j] * Ys[j-1] + Ss[j] * Ds[j];
                              Xs[j] = -Ss[j] * Ys[j-1] + Cs[j] * Ds[j];

                              if ( j != tam ) {
                                    Rs[j-1] = Ss[j] * Bs[j+1];
                                    Ys[j] = Cs[j] * Bs[j+1];
                              }
                        }

                        Zs[tam-1] = Xs[tam-1];

                        As[0] = Ss[1] * Qs[0] + Cs[1] * Zs[0];

                        Bs[1] = Ss[1] * Zs[1];

                        for ( int j=1 ; j<tam-1 ; ++j ) {
                              As[j] = Ss[j+1] * Qs[j] + Cs[j] * Cs[j+1] * Zs[j];
                              Bs[j+1] = Ss[j+1] * Zs[j+1];
                        }

                        As[tam-1] = Cs[tam-1] * Zs[tam-1];

                        maxIter--;
                  }

//                printf("Paso el max de iteraciones y QR no converge\n");
            }

            void multiplicarEscalar ( double beta)
            {
                  for (uint i = 0; i < n ; i++)
                        for (uint j = 0; j < m ; j++)
                              matriz[i][j] = matriz[i][j] * beta;
            }

            void copiar (Matriz<T> &entrada)
            {
                  for (uint i = 0; i < n ; i++)
                        for (uint j = 0; j < m ; j++)
                              matriz[i][j] = entrada[i][j];
            }

            void identidad( double factor )
            {
                  for (uint i = 0; i < n ; i++)
                        for (uint j = 0; j < m ; j++)
                              matriz[i][j] = (i==j) ? factor : 0.f;
            }

            double distanciaAutovector( Matriz<T> & autovector ){

                  assert( autovector.cantFil() == n && autovector.cantCol() == 1 );

                  Matriz<double> autov(autovector);
                  Matriz<double> multi(autovector);
                  Matriz<double> autovN(autovector);

                  autov.normalizar();
                  autovN.normalizar();

```

```
                multi.cargarMultiplicacion(*this, autov);
                multi.normalizar();

                multi.cargarResta(autov,multi);

                double rtrn = multi.norma();

                if( rtrn > sqrt(2) ){
                      multi.cargarMultiplicacion(*this, autovN);
                      multi.normalizar();
                      multi.multiplicarEscalar(-1.);

                      multi.cargarResta(autovN,multi);

                      rtrn = multi.norma();
                }

                return rtrn;
          }

          double norma()
          {
                double res = 0;
                for (uint i = 0; i < n; i++)
                      res += matriz[i][0] * matriz[i][0];

                return sqrt(res);
          }

          void normalizar()
          {
                double norm = this->norma();
                this->multiplicarEscalar( 1.f/norm );
          }

          double normaInf()
          {
                double res = fabs(matriz[0][0]);
                for (uint i = 0; i < n; i++){
                      res = ( res  > fabs(matriz[i][0]) ) ? res : fabs(matriz[i][0]);
                }
                return res;
          }

          int menorP()
          {
                int p = 0;
                double res = matriz[0][0];
                for (uint i = 0; i<n; i++){
                      if ( fabs(res) < fabs(matriz[i][0]) )
                      {
                            res = matriz[i][0];
                            p = i;
                      }
                }
                return p;
          }

          void deflacion(double autovalor, Matriz<T> &autovector)
          {

                int n = autovector.cantFil();
                Matriz<double> temp(n,n);

                Matriz<double> autovectorN( autovector );
                Matriz<double> autovectorNT(1,n);

```

```
            autovectorN.normalizar();

            autovectorNT.transponer(autovectorN);

            temp.cargarMultiplicacion(autovectorN,autovectorNT);
            temp.multiplicarEscalar(autovalor);

            this->cargarResta(*this, temp);
        }

        void potenciaSimple(double &guess, Matriz <T> &x, double tol, int maxIter )
        {
            assert( x.cantFil() == n && x.cantCol()==1 );

            Matriz<double> resta(n,1);
            Matriz<double> mult(1,1);
            Matriz<double> y(n,1);

            int p = x.menorP();
            double xp = x[p][0];
            x.multiplicarEscalar(1.f/xp);

            double mu_0 = 0;
            double mu_1 = 0;
            double mu_s = 0;

            uint k = 0;
            while (k<n){

                y.cargarMultiplicacion(*this,x);

                p = y.menorP();

                double mu =  y[p][0];
                mu_s = mu_0 - (mu_1 - mu_0)*(mu_1 - mu_0)/( mu - 2*mu_1 + mu_0 );

//              printf("matrix Y");
                y.multiplicarEscalar(1.f/mu);

                resta.cargarResta(x,y);

                x.copiar( y );
                double err = resta.normaInf();

//              printf("%f\n", err);

                if ( fabs(err)<fabs(tol) && k >= 4 )
                {
                    guess = mu_s;
                    //printf("Termino Bien en %d iteraciones \n", k);
                    return;
                }

                k++;
                mu_0 = mu_1;
                mu_1 = mu;
            }

//          printf("se llego a la maxima cant de iteraciones");

            guess = mu_s;
        }

        void potenciaInversa(double &guess, Matriz <T> &x, double tol, int maxIter )
        {
            assert( x.cantFil() == n && x.cantCol()==1 );

```

```
            Matriz<double> resta(n,1);
            Matriz<double> mult(1,1);
            Matriz<double> y(n,1);
            Matriz<double> P(n,n);  // asumo que P, L y U son cuadradas
            Matriz<double> L(n,n);
            Matriz<double> U(n,n);
            Matriz<double> B(n,n);
            Matriz<double> qident(n,n);


            qident.identidad( guess );

//          x.imprimirMatriz();

            int p = x.menorP();
            double xp = x[p][0];
            x.multiplicarEscalar(1.f/xp);

            B.cargarResta(*this,qident);
            B.factorizacionLU(P,L,U);

            uint k = 0;
            while (k<n) {

                // resuelvo el sistema de ecuaciones
                y.resolverLU(x,P,L,U);

                p = y.menorP();
                double mu =  y[p][0];

                y.multiplicarEscalar(1.f/mu);
                resta.cargarResta(x,y);

                x.copiar( y );
                double err = resta.normaInf();

//              printf("%f\n", err);

                if ( fabs(err)<fabs(tol) )
                {
                //  printf("i: %d - e: %f\n",k,err);
                    mu = 1.f / mu + guess;
                    guess = mu;

                    //printf("Itere %d veces\n", k);

                    return;
                }

                k++;
            }
//          printf("se llego a la maxima cant de iteraciones\n");
        }

        void factorizacionLU(  Matriz<T> &P, Matriz<T> &L, Matriz<T> &U )
        {
            uint i,k,j;

            for(uint i=0;i<n;i++){
                for(k=0;k<n;k++){
                    U[i][k] = matriz[i][k];
                    P[i][k] = (i==k) ? 1.f : 0.f;
                }
            }

            for (i=0;i<n;i++){

```

```
                    U.pivoteoParcial(i,P,L);
                    L[i][i]=1.f;
                    for(j=i+1;j<n;j++){
                          assert( !COMPARAR_DOUBLE( U[i][i], 0.f ) );
                          double mji = U[j][i] / U[i][i];
                          for (k=i+1;k<n;k++){
                                U[j][k] = (double)U[j][k] - mji * (double)U[i][k];

                          }
                          U[j][i] = 0.f;
                          L[j][i] = (T) mji;
                    }
              }
        }

        void resolverLU(Matriz<T> &entrada, Matriz<T> &P, Matriz<T> &L, Matriz<T> &U)
        {
              // Tengo A x = b, como  P A = L U , tengo L U x = Pt b
              // sea Y = U x , entonces L Y = Pt b, lo resuelvo con SustitucionAtras
              // depues para obtener x tengo que U x = Y, lo resuelvo con forward
                  substitution

              Matriz<double> Y(n,1);
              Matriz<double> res(n,1);
              Matriz<double> b2(entrada.cantFil(),entrada.cantCol());
              /********* Transpongo P y lo multiplico por b *****/

              b2.cargarMultiplicacion(P,entrada);


              /*********Hago L Y = b2 ******************/
              forwardSubstituion(L,b2,Y);

              /*********Hago U x = Y ******************/
              backwardSubstitution(U,Y,res);

              for(uint i=0;i<n;i++){
                    matriz[i][0] = res[i][0];
              }
        }

        void forwardSubstituion( Matriz<T> &L, Matriz<T> &B ,Matriz<T> &res )
        {
              Matriz<double> resD(res.n,res.m);
              //resD[0][0]=B[0][0];

              for( uint f=0; f < n ; f++ ){
                    resD[f][0]= B[f][0];
                    for(uint j=0;j<f;j++){
                          assert( !COMPARAR_DOUBLE(L[f][f], 0.f) );
                          resD[f][0]-=L[f][j]*resD[j][0];
                    }
                    resD[f][0]/= L[f][f];
              }

              for(uint i=0;i<res.n;i++){
                          res[i][0] = (T) resD[i][0];
              }
        }

        void backwardSubstitution( Matriz<T> &U, Matriz<T> &B ,Matriz<T> &res )
        {
              Matriz<double> resD(res.n,res.m);
              resD[n-1][0]=B[n-1][0]/U[n-1][n-1];

              int f = n -2;

```

```
			for( f=n-2; f >= 0 ; f-- ){
				resD[f][0]= B[f][0];
				for(uint j=f+1;j<n;j++){
					assert( !COMPARAR_DOUBLE(U[f][f], 0.f) );
					resD[f][0]-=U[f][j]*resD[j][0];
				}
				resD[f][0]/= U[f][f];
			}

			for(uint i=0;i<res.n;i++){
					res[i][0] = (T) resD[i][0];
			}
		}

		void pivoteoParcial( int i,Matriz<T> &P,Matriz<T> &L)
		{
			uint k;
			int max=i;

			for (k=i;k<n;k++){
				 max = (fabs(matriz[k][i]) > fabs(matriz[max][i])) ? k :max;
			}

			if( max != i ){

				for (k=0;k<n;k++){
					//cambio de filas
					T temp=matriz[i][k];
					matriz[i][k]= matriz[max][k];
					matriz[max][k] = temp;

					temp=P[i][k];
					P[i][k]= P[max][k];
					P[max][k] = temp;

					temp=L[i][k];
					L[i][k]= L[max][k];
					L[max][k] = temp;
				}
			}
		}
};

#endif
```

C. Tablas

C.1 Tabla A.

Promedio tiempo corrida métodos:

Precisión	Promedio QR	Promedio PS
0,000001	9,28E+007	2,69E+007
0,00001	9,34E+007	2,09E+007
0,0001	9,36E+007	1,58E+007
0,001	9,32E+007	8,38E+006
0,01	9,28E+007	3,04E+006
0,1	9,33E+007	2,02E+006
1	9,35E+007	2,03E+006
10	9,24E+007	2,03E+006

C.2 Tabla B.

Porcentaje de digitos reconocidos Vecinos Ponderados:

	1 Vecino	5 Vecinos	10 Vecinos	30 Vecinos	50 Vecinos	100 Vecinos
10	88.740	86.060	82.78	77.200	74.160	71.04
20	94.860	93.440	91.8	87.700	86.420	83.6
30	95.500	94.440	93.44	89.820	88.620	86.38
40	95.900	95.060	93.44	90.280	88.900	86.16
50	95.940	95.080	93.46	90.420	88.780	86.52

C.3 Tabla C.

Porcentaje de digitos reconocidos Vecinos:

	1 Vecino	5 Vecinos	10 Vecinos	30 Vecinos	50 Vecinos	100 Vecinos
10	88.740	90.300	90.2	89.960	89.360	88.16
20	94.860	95.200	94.84	94.000	93.400	92.68
30	95.500	95.760	95.64	94.820	94.160	93.16
40	95.900	96.080	95.86	94.800	94.040	93.26
50	95.940	96.000	96.06	94.720	94.020	93.22

C.4 Tabla D.

Cantidad de componentes con Método QR Potencia Inversa Kvecinos

Componentes	%digitosRec
10	90.800
20	95.680
30	96.280
40	96.240
50	96.440
60	96.380
70	96.160
80	96.040

C.5 Tabla E.

Cantidad de componentes con Método QR Potencia Simple Kvecinos

Componentes	%digitosRec
10	67.240
20	71.080
30	69.540
40	67.780
50	66.120
60	65.160
70	64.160
80	63.340

C.6 Tabla F.

Hitrate Métodos

	Distancias Medias	Kvecinos
0	0.698	0.996
1	0.988	0.993
2	0.404	0.964
3	0.612	0.956
4	0.722	0.958
5	0.311	0.961
6	0.701	0.974
7	0.719	0.947
8	0.501	0.928
9	0.725	0.927