DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Métodos Numéricos

Trabajo Práctico 2

Reconocimiento de dígitos.

Resumen

En este trabajo pondremos en práctica distintos algoritmos para reconocer una cierta cantidad de dígitos manuscritos. Se trabajará con una base train, a la cual le realizaremos particiones para entrenar los algoritmos. Se implementará el metodo de kNN(k-Nearest Neighbors). Debido a que este es sensible a la dimensión de los objetos a considerar, además se implementará el método de Análisis de Componentes Principales para reducir el tamaño de dichos objetos. Se llevarán a cabo experimentaciones para poder determinar los parámetros óptimos para cada método. Al final del trabajo, se llegarán a conclusiones sobre lo descubierto.

Integrante	LU	Correo electrónico
Armagno, Julián	377/12	julian.armagno@gmail.com
More, Ángel	931/12	angel_21_fer@hotmail.com
Pinzón, Germán	475/13	pinzon.german.94@gmail.com
Porto, Jorge	376/11	cuanto.p.p@gmail.com

Palabras claves:

Learning Machine. kNN. Análisis de Componentes Principales. Auto-Valores. Auto-Vectores. Método de las Potencias. K-fold cross validation

Índice

1. Introducción Teórica

1.1. k Nearest Neighbor y Principal Component Analysis

En el presente trabajo utilizaremos los métodos de kNN y PCA para llevar a cabo el reconocimiento de dígitos manuscritos. Comenzaremos explicando resumidamente los métodos utilizados, de una manera más general. Dado un vector v y un conjunto de vectores U, queremos saber a que clase pertenece v. Nosotros sabemos a que clase pertenece cada vector $u_i \in U$, entonces una manera sencilla de "estimar" a que clase pertenece v en base a la información que poseemos, es comparándolo con cada vector de U. Una manera de efectuar esta comparación es tomando la distancia (norma 2) entre v y cada vector $u_i \in U$ y quedarnos con la clase del u_m que minimice esa distancia sobre todos los demás. Lo que hace el algoritmo kNN es generalizar un poco esta idea. En lugar de quedarnos con la clase del u_m que minimice la distancia a v, kNN toma las clases de los k vectores $u_1, u_2, ..., u_k$ cuyas distancias sean mínimas y elige entre ellas la clase que más aparezca. Para conocer el parámetro k es necesario realizar experimentos probando distintos valores y tomar una decisión en base a la calidad de los resultados.

El algoritmo kNN es conceptualmente fácil de entender e implementar, lo cual constituye una ventaja, pero su desventaja principal es el tiempo de cómputo que requiere. Esto se debe a que, como los vectores representan imágenes, la dimensión de estos vectores suele ser grande. Dado que la idea detras del reconocimiento de imágenes en este trabajo radica en utilizar una base de datos relativamente grande (de ahora en más dbTrain) para determinar que tipo de imagen es la que estamos tratando de reconocer, el costo de computar estas distancias se multiplica por la cantidad de imágenes que tenemos en nuestra base de datos. Vemos que este método, así como está planteado, no escala.

Como ya dijimos, por cuestiones de performance no es conveniente utilizar kNN de manera directa con los datos que tenemos. Lo que vamos a intentar hacer entonces, es realizar una transformación de nuestros datos de manera tal que luego, cuando queramos aplicar kNN no nos resulte tan costoso. Recordemos que nuestros datos son vectores en un espacio $\mathbb{R}^n$, entonces lo que vamos a querer hacer es transformarlos a vectores en otro espacio $\mathbb{R}^\alpha$ tal que $\alpha < n$. Pero también vamos a querer que esta transformación no "cambie por completo" a nuestros vectores, en el sentido de que sigan representando las imágenes que representaban o al menos conserven las "partes relevantes" de ellas. Para poder llevar a cabo esta transformación realizaremos los siguientes pasos:

1. Tomamos los vectores de dbTrain que usaremos para comparar la imagen que queremos reconocer en forma matricial (cada vector una fila de la matriz) y hallar la matriz de covarianza $M \in \mathbb{R}^{nxn}$.

2. Hallar una matriz $P \in \mathbb{R}^{nxn}$ que nos permita disminuir la covarianza de los datos de dbTrain. Esto equivale a buscar las variables que tengan la mayor varianza entre sí y covarianza 0. El objetivo de esto es disminuir la redundancia en nuestros datos.

3. Sea P' la matriz P con las primeras α columnas (este parámetro se fija mediante experimentación), es decir $P' \in \mathbb{R}^{nx\alpha}$, y x_i la i-ésima imagen de dbTrain, realizamos el producto $x'_i = P'^t \widehat{x}_i$, ahora x'_i es nuestra nueva i-ésima imagen de train y está en el espacio $\mathbb{R}^{nx\alpha}$.

4. Sea x una imagen vectorizada cuya clase queremos reconocer, realizamos el producto $x' = P'^t \widehat{x}$ donde $\widehat{x} = (x - \mu)/(\sqrt{n-1})$, μ es la media de las imágenes de dbTrain y n la cantidad de imágenes de dbTrain. Como ahora $\widehat{x} \in \mathbb{R}^\alpha$ y los vectores de dbTrain están en el mismo espacio, podemos aplicar kNN con x' y los x'_i.

Para obtener P lo que hacemos es hallar la base ortonormal de autovectores de M. Sabemos que dicha base existe por ser M simétrica. Entonces la columna i de P va a ser el vector v_i, el i-ésimo autovector de M. Esta base lo que nos permite hacer es "observar" a nuestros datos desde otro lugar, o sea ahora nuestros ejes de coordenadas van a estar en las direcciones donde más varianza existe entre los datos.

Una vez completados estos tres pasos, cuando queramos reconocer a que clase pertenece un vector v, trabajamos con $v' = P'^t v$ y, dado que ahora v' es un vector de $\in \mathbb{R}^\alpha$ al igual que los vectores de dbTrain podemos aplicar kNN.

1.2. Método de la potencia y deflación

En la sección anterior explicamos los métodos que nos permiten reducir la dimensión de las imágenes con las cuales vamos a trabajar y, en base a cierta informacion que tenemos, efectuar comparaciones para predecir a que clase pertenece una imagen. Vimos que, uno de los pasos de PCA es obtener una matriz P cuyas columnas son los autovectores de otra matriz M. El método de la potencia, junto con el método de deflación, nos permite encontrar los autovectores y autovalores asociados a la matriz M.

Dada una matriz $A \in \mathbb{R}^{nxn}$ cuyos autovalores $\lambda_1, ..., \lambda_n$ cumplen $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq ... \geq |\lambda_n|$ y $v_1, ..., v_n$ los autovectores asociados, el Método de la potencia estimará v_1 y λ_1. Decimos estimará porque para tener el valor exacto deberíamos calcular un límite, entonces vamos a "simular" este límite computacionalmente con lo cual el resultado puede tener cierto error. Este método necesita de un vector inicial x_0 y un valor de k relativamente grande y lo que va a hacer es calcular $x_{i+1} = \frac{Ax_i}{||Ax_i||_2}$ y $\widehat{\lambda}_1 = \frac{\Phi(Ax_{i+1})}{\Phi(Ax_i)}$ desde $i = 1$ hasta k. El resultado será $x_{k+1} = \widehat{v_1} \approx v_1$ y $\widehat{\lambda}_1 \approx \lambda 1$. Para que esto se cumpla es importante aclarar que la función $\Phi : \mathbb{R}^n \to \mathbb{R}$ que usamos debe cumplir las siguientes propiedades:

- Debe ser continua
- Si $v \in \mathbb{R}^n$ y $v \neq 0$ entonces $\Phi(v) \neq 0$
- Si $v \in \mathbb{R}^n$ y $\alpha \in \mathbb{R}$ entonces $\Phi(\alpha v) = \alpha \Phi(v)$

El Método de la potencia también pide que el vector inicial x_0 no sea perpendicular a v_1, sin embargo a la hora de implementarlo en una computadora esto puede no ser tenido en cuenta y no traerá grandes consecuencias. Esto se debe a que nuestro resultado va a estar arrastrando cierto error a medida que el método itere y este error va a estar cambiando la dirección de x_i (aunque sea mínimamente) y en ese caso dejaría de ser perpendicular a v_1.

Vimos como funciona el Método de la potencia y que nos devuelve el autovalor de mayor módulo junto con su autovector asociado, sin embargo nosotros queremos obtener todos los autovalores y autovectores de A. Esto se soluciona definiendo una nueva matriz $\widehat{A}$ y aplicando nuevamente el Método de la potencia pero ahora sobre $\widehat{A}$. Supongamos que ya tenemos los valores $\widehat{v_1} \approx v_1$ y $\widehat{\lambda}_1 \approx \lambda_1$ obtenidos al aplicar el Método de la potencia sobre A, entonces para obtener $\widehat{v_2}$ y $\widehat{\lambda}_2$ aplicamos nuevamente el método pero ahora sobre $\widehat{A} = A - \widehat{\lambda}_1 \widehat{v_1} \widehat{v_1}^t$. Para el caso general, si queremos obtener $\widehat{v_{i+1}}$ y $\widehat{\lambda}_{i+1}$, tenemos $\widehat{v_i}$ y $\widehat{\lambda}_i$ y nuestra matriz es $\widehat{A}$, definimos $\widehat{\widehat{A}} = \widehat{A} - \widehat{\lambda}_i \widehat{v_i} \widehat{v_i}^t$ y aplicamos el método sobre $\widehat{\widehat{A}}$. A esto se le llama *deflación*. Los autovalores y autovectores de la nueva matriz que definimos van a ser los mismos que la matriz original (la primera o la que definimos en el paso anterior) excepto por el autovalor de mayor módulo y su autovector asociado.

Si bien vimos que para aplicar el método de la potencia en una matriz A se tiene que cumplir que A tenga un autovalor mayor estricto (de multiplicidad 1) en módulo a todos los demás, para poder hacer deflación iterativamente y obtener todos los autovalores y autovectores de A es necesario que valgan las desigualdades de manera estricta: $|\lambda_1| > |\lambda_2| > |\lambda_3| > ... > |\lambda_n|$. Esto debe ser así para que cada nueva matriz que definimos cumpla las hipótesis que requiere el Método de la potencia.

2. Desarrollo

2.1. k-Nearest Neighbors (kNN):

El primer método utilizado es kNN. Como se dijo en la introducción teórica, este método es muy intuitivo, se basa en tomar a cada imagen como un punto en el espacio, y se hace una votación de la moda de los k vecinos cercanos. Todas las imagenes de dbTrain (28x28 pixeles), se encuentran guardadas vectorizadas en una matriz de $\mathbb{R}^{42000x785}$. La primera columna establece el label de cada imagen, las columnas restantes son los pixeles.
Mediante el K que determine la partición, utilizando la técnica de K-Fold Cross Validation, vamos a dividir esa matriz, en dos, una de train, con $(42000/K) * (K-1)$ imágenes y otra de test, con $42000/K$ imágenes. A su vez, vamos a eliminar la primera columna de cada matriz, y nos las vamos a guardar en 2 vectores diferentes. Asi podremos comparar cada imagen de la matriz de test, con todas las imágenes de la matriz de train y al mismo tiempo tener guardados los labels correspondientes a cada imagen. Finalmente la idea algoritmica se detalla en el siguiente pseudocódigo:

Algoritmo 1 kNN(int k, matriz Test, matriz Train, vector digTest, vector digTrain)

```
1: reconocidos = 0
2: for z = 0; z <Test.filas() ; z++ do
3:     vector<pair<int, int> > normas2AlCuadrado
4:     \\ La primera componente de cada tupla es el label del digito de la base de train(de ahora
   en más d), y la segunda componente es la distancia entre el digito de la base de test y d
5:     for m = 0; m < Train.filas(); m++ do
6:         \\ calculamos las distancias
7:         distanciaAlCuadrado = 0
8:         for i = 0; i <Test.columnas() ; i++ do
9:             distanciaAlCuadrado += (Test[z][i] - Train[m][i]) *(Test[z][i] - Train[m][i])
10:        end for
11:        if normas2AlCuadrado.size() < k then
12:            \\ colocamos las primeras k normas
13:            pair<unsigned int, int> a
14:            a.first = digTrain[m]
15:            a.second = distanciaAlCuadrado
16:            normas2AlCuadrado.pushback(a)
17:        else
18:            if haymayor(normas2AlCuadrado, distanciaAlCuadrado then
19:                \\ si ya tengo k voy sacando las mayores distancias
20:                int posmayor = dondemayor(normas2AlCuadrado)
21:                normas2AlCuadrado[posmayor].first = digTrain[m]
22:                normas2AlCuadrado[posmayor].second = distanciaAlCuadrado
23:            end if
24:        end if
25:    end for
26: end for
27: \\ Ahora hago la votación entre los k vecinos
28: digitos[10] =  0, 0, 0, 0, 0, 0, 0, 0, 0, 0
29: for int t = 0; t < k; t++ do
30:    digitos[normas2AlCuadrado[t].first]++
31: end for
32: ganador = 0
33: for int x = 0; x < 10; x++ do
34:    if digitos[x] > digitos[ganador] then
35:        ganador = x
36:    end if
37: end for
38: \\ Si en la votación gano el verdadero label de la imagen de test, sumamos reconocidos
39: if ganador == digTest[z] then
40:    reconocidos++
41: end if
42: tasaDeReconocimiento = reconocidos/Test.filas()
```

2.2. Principal Component Analysis (PCA):

El segundo método utilizado es PCA. Dado que las imagenes estan representadas por vectores y las dimensiones de los mismos pueden ser muy grandes, a la hora de analizar las imagenes se requiere una gran cantidad de tiempo de computo así como de memoria. PCA tiene como objetivo redimensionar dichos vectores, siempre que se obtenga un determinado compromiso entre las nueva cantidad de variables y la calidad de las imagenes que ahora se representan. Para esto se construye un nuevo sistema de ecuaciones donde en el eje i se representa la i-esima varianza de mayor valor para el conjunto original de imagenes dadas. De esta manera cada variable esta correlacionada,

siendo las primeras las de mayor importancia, por lo que se pueden obviar las últimas componentes (ya que serían las que menos importancia tienen) reduciendo el número de variables. Para poder llevar a cabo esta transformacion lineal es necesario construir la matriz de covarianza para el conjunto de valores que representan a las imagenes. Para esto, sean n observaciones (imagenes) y dadas dos coordenadas x_i, x_k su covarianza se obtiene como:

$$\sigma_{x_j x_k} = (1/(n-1)) \sum_{i=1}^{n} (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k) = (1/(n-1))(x_k^{(i)} - \mu_k v)^t (x_j^{(i)} - \mu_j v)$$

con μ_j, μ_k la media de las coordenadas j y k respectivamente; $x_j^{(i)}$, $x_k^{(i)}$ las coordenadas j y k de la i-esima imagen vectorizada y v^t = (1,.....,1).

De esta forma la covarianza entre dos muestras puede ser calculado como el producto entre dos vectores. Por lo que si A ϵ $\Re^{nx784}$ representa n imagenes de train vectorizadas de 784 variables cada una, la covarianza pueden ser calculadas como:

$$M_x = A'^t * A'; \qquad con\ A' = A/\sqrt{n-1}\ y\ M_x\ \epsilon\ \Re^{784x784}$$

obteniendo en la diagonal la varianza de cada coordenada. Con el fin de disminuir las redundancias (covarianza), se plantea un cambio de base. Para esto se diagonaliza la matriz M_x, de forma tal que:

$$M_x = P * D * P^t; \qquad D\ diagonal\ y\ P\ matriz\ ortogonal\ con\ los\ autovectores\ de\ A\ como\ columnas^1$$

Para obtener los autovectores de M_x se empleó el método de la potencia[2] . Dado que la misma es una técnica iterativa que converge al autovector asociado, se estableció que la misma se detenga luego de alcanzar un maximo de 3000 iteraciones o cuando la diferencia entre cada coordenada del vector obtenido en la iteración k respecto de la k+1 sea muy poca (en este caso optamos por 0.0000009). Una vez obtenido el primer autovector procedimos a calcular el segundo, aplicando el método de deflación[3] .Ambas técnicas serán aplicadas un número α de veces. En la sección experimentación se evaluará como se comportan los resultados y el tiempo al variar este valor. Una vez obtenidos los α autovectores se procedió a llevar a cabo el cambio de base, tanto para las imagenes utilizadas como train como para las de test, por lo que se multiplicó cada autovector obtenido por las imagenes vectorizadas en A':

$$v_i^t * a^{(i)}\ ;\ v_i\ el\ i\text{-}esimo\ autovector\ y\ a^{(i)}\ la\ i\text{-}esima\ imagen\ vectorizada$$

lo que matricialmente se puede obtener al hacer: A'*P (llamaremos a la matriz resultante $TcTrain$). De esta forma la matriz TcTrain ϵ $\Re^{n*\alpha}$. Para poder comparar las imagenes de test (matriz B) con TcTrain, es necesario llevar a las imagenes de test a las mismas dimensiones, por lo que a cada una se le restara la media que corresponda y dividirá por $\sqrt{n-1}$, obteniendo B'. Una vez realizado esto procedemos a cambiar la base tal como se hizo para las de train:
TcTest = $B' * P$. En las matrices TcTrain y TcTest se encuentran las nuevas coordenadas de las imágenes(con reducción de la dimensión) almacenadas como filas.

Finalizado los cambios de bases obtenemos imagenes en un mismo espacio vectorial cuyas dimesiones son menores a las originales. Pudiendo aplicar distintas técnicas para llevar a cabo el reconocimiento de los dígitos. Optaremos por aplicar KNN para así poder comparar las diferencias entre tiempo de computo y análisis cuando se busca reconocer dígitos utilizando PCA + KNN y solo KNN.

A continuación se muestra un pseudocódigo de los procedimientos descriptos anteriormente:

Algoritmo 2 PCA(matriz Train, matriz Test, int α)

```
1: n = cantidad de imagenes de train
2: \\Se procede a calcular el promedio de las imagenes de train
3: vectorPromedio ← crear(784)
4: for i = 0; i < 784; i++ do
5:     sum = 0
6:     for j = 0; i < n; j++ do
7:         sum = sum + Train[j][i]
8:     end for
9:     promedio[i] = sum/n
10: end for
11: \\ calculamos la matriz de covarianza:
12: for i = 0; i < n; i++ do
13:     for j = 0; j < 784; j++ do
14:         Train[i][j] = Train[i][j] - promedio[j]
15:     end for
16:     Train[i][j] = Train[i][j]/√(n − 1)
17: end for
18: matriz covarianza ← Train^t * Train
19: \\ obtenemos la matriz con los α autovectores de la matriz de covarianza como columna (P):
20: P ← metodo de la potencia y de deflacion (covarianza, α)
21: for i = 0; i < cantidad de imagenes de test; i++ do
22:     for j = 0; j < 784; j++ do
23:         test[i][j] = test[i][j] - promedio[j]
24:     end for
25:     test[i][j] = test[i][j]/√(n − 1)
26: end for
27: \\ realizamos la transformación caracteristica para train (TcTrain) y para test (TcTest)):
28: TcTrain ← Train * P
29: TcTest ← Test * P
30: \\ finalmente procedemos a reconocer los dígitos:
31: Knn(tcTriain, TcTest)
```

2.3. K-Fold Cross Validation

Para poder concentrarnos en la evalución de los métodos y la óptima eleccion de los parámetros, necesitamos probar los mismos sobre la base de train, ya que sobre sus elementos conocemos el digito al que pertenece cada uno. Ante esta situación, particionaremos dicha base en dos, utilizando una parte de ella en forma completa para el training y la restante como test, pudiendo asi verificar la predicción realizada y mostrar las tasas de efectividad (La cantidad de dígitos correctamente clasificados respecto a la cantidad total de casos analizados).
Sin embargo, surge un problema no menor, realizar la experimentación sobre una única partición podría traer como consecuencia predicciones no deseadas, y una mala estimación de los parámetros, por ejemplo podría ocacionar *overfittin*, que el algoritmo del método bajo análisis quede ajustado a una serie de características muy específicas de la base de train que no tienen relación con el obejeto de estudio. Para soluconar el problema en cuestión, se usará la técnica de Cross Validation, en particular el K-Fold Cross Validation, para que los resultados de la experimentación resulte estadísticamente más robustos.
La técnica de K-Fold Cross Validation consiste en particionar de forma aleatoria la base de train en K conjuntos de igual tamaño (Siempre asumiendo que el cardinal del conjunto de la base de train es múltiplo de K). Luego se realizan K ejecuciones del programa, cada una de ellas eligiendo un conjunto para test, y utilizando los K-1 conjuntos restantes para train. Para las K ejecuciones se usarán los mismos parámetros en cada una de ellas. Además, en el método tradicional se suelen realizar varias corridas para un mismo valor de K.

Para calcular estas particiones usaremos el comando CVPARTITION de MATLAB de la siguiente manera:

Algoritmo 3 calcularParticion(int K, int n)

```
1: C = cvpartition(n,'KFold',K)
2: fileID = fopen('K.txt','w');
3: for int i = 1:K do
4:     fprintf(fileID,'d ',C.training(i)')
5:     fprintf('n')
6: end for
7: fclose(fileID)
```

3. Experimentaciones

3.1. Elección de K, hipótesis 1

Para comprobar la eficacia del método KNN y KNN + PCA procederemos a desarrollar distintas experimentaciones. En primer lugar para KNN variaremos la cantidad de vecinos (k) que vamos a utilizar para reconocer a que clase pertenece cada imagen de test. Luego, para PCA + KNN también vamos a variar la cantidad de vecinos a tomar, fijando un α (cantidad de componentes principales), y para el mejor de ellos (evaluando calidad de la solucion y tiempo de computo) procederemos a variar el valor de α utilizando el mejor k obtenido anteriormente.
Además para dichas experimentaciones aplicaremos la técnica de $K-Fold\ Cross\ Validation$, para la misma es necesario la elección de un valor K que representa la cantidad de conjuntos en la que dividiremos la base de train. Dado que para la formación de las particiones, que serán utilizadas tanto para test como para train, utilizamos el comando CVPARTITION la misma las calcula como:

cantidad de imágenes de test = n/K (1)
cantidad de imágenes de train = n - n/K (2)
con n la cantidad de imágenes total, en nuestro caso las 42.000 de la base de train.

A partir de esta distribución elaboramos una primer hipótesis. **Hipótesis 1**: Para los K de mayor valor, la tasa de reconocimiento promedio es mayor que para los de menor valor.

Como se menciono, lo que nos llevo a formular esta hipótesis es la distribución en la cantidad de imágenes de test y de train al utilizar distintos K. Notemos por (1) que la cantidad de imágenes utilizadas como test aumenta a medida que K disminuye y como consecuencia las de train disminuyen (2).

Si al momento de reconocer un dígito i, se tienen pocas imágenes con las que comparar (train) entonces tenemos menos representaciones de ese dígito, por lo que al momento de elegir los k vecinos estamos mas propensos a que solo una fracción de los mismos pertenezcan a la misma clase que i. Como en la elección de la clase, a la que supuestamente pertenecería i, también intervienen los otros vecinos (los que tienen clase distinta a la de i) podría ser que la clase prioritaria entre los k vecinos no sea la de i, dado a que hay pocas representaciones de este dígito, terminando en un reconocimiento erróneo. Es por esto que creemos que, en contraste, si aumentamos las imágenes de train habria mayores probabilidades de que la mayoria de los vecinos sean de la misma clase que i, obteniendo una mayor tasa de reconocimiento.

Para poder corroborar esta hipótesis a las experimentaciones mencionadas anteriormente (variación de k, y alpha) también las haremos variando el valor de K. Se eligieron tres valores para experimentar, los mismos son K = 2, 10, y 20, dicha elección esta ligada a como quedan distribuidas la cantidad de imágenes utilizadas tanto para test como para train. Cuando K = 2 solo un 50 % del total de las imágenes son de entrenamiento (además es el mínimo valor permitido por matlab para el comando cvpartition, usando Kfold); para K = 10 un 90 % y para 20 un 95 %, de esta manera abarcamos un amplio y variado rango de valores para la cantidad de imágenes a reconocer y las utilizadas como entrenamiento. Como consecuencia de esta elección surge otro problema a analizar y es que, si al aumentar el K las tasas de reconocimiento promedio aumentan ¿que porcentaje de mejora obtenemos (si es que lo hacen)? y ¿como influye el tiempo de computo en estas variaciones?. Por lo que en las experimentaciones además buscaremos responder estas preguntas para poder determinar la mejor relación entre calidad de soluciones y tiempo.

3.1.1. Experimento 1: Variando el k en kNN

Se llevará a cabo una cierta cantidad de experimentaciones para poder contrastar la siguiente hipótesis:

- Hipótesis N° 2: En el método de kNN, al incrementar k vamos obteniendo una mayor tasa de efectividad. En otras palabras, se resumiría que a mayor valor de k, mayor porcentaje de

la tasa.

Para verificar la hipótesis N° 2, se llevará a cabo una experimentación corriendo el programa con la base de train de Kaggle, utilizando la técnica de K-Fold cross validation, con 2, 10 y 20 particiones. En este caso, como se ejecutará solo el método de kNN sin ningún complemento, la elección del parametro α no incide en la misma.
Luego se procederá a realizar un análisis de las tasas de efectividad resultantes.

3.2. PCA + KNN

3.2.1. Variando k

Dados los valores de K elegidos previamente, realizaremos una serie de mediciones de tiempos y tasas para $k \in \{2, 30, 100, 500\}$, fijando un valor de $\alpha = 50$. El objetivo de estas mediciones es conseguir el valor de k que mejor se comporte de manera que estén equilibrados el tiempo de cómputo y la tasa de reconocimiento. Luego, tomaremos el valor de k conseguido y procederemos a variar α. Los valores de k elegidos representan un subconjunto de los que se utilizaron en el experimento sobre KNN con el objetivo de que luego podamos comparar las tasas de reconocimiento y el tiempo de cómputo que producen ambos métodos. El valor de α fue fijado en 50 porque, si bien es un valor relativamente chico para el rango en el que se encuentra (de 1 a 784), observamos que las tasas de reconocimiento con este valor son sorprendentemente altas. Esto significa que no tendría sentido aumentar α para obtener (o no) una mínima diferencia en la tasa de reconocimiento y terminar pagando un costo de cómputo y memoria (a mayor α mayor son las dimensiones de las matrices que utilizamos) mucho mayor.

3.2.2. Variando α

Una vez ya conseguido el valor de k que buscamos, la idea ahora es probar distintos valores de α y ver si al tomar valores grandes obtenemos tasas de reconocimiento mayores (esta será nuestra tercera hipótesis del trabajo). En principio parece muy razonable creer que esto va a suceder ya que al aumentar α estamos agrandando la nueva base en la cuál vamos a expresar nuestros datos. O sea cuanto más grande es α, menor es la cantidad de información que "recortamos" de cada muestra. Los valores de α que vamos a elegir van a ser los del siguiente conjunto: $\{20, 100, 300\}$, no contamos el $\alpha = 50$ porque ya lo usamos cuando variamos k pero también tenemos ese resultado. El máximo que usamos es $\alpha = 300$ debido a que, como estamos utilizando PCA con el objetivo de reducir el tiempo de cómputo de KNN, en la práctica vamos a tratar de no utilizar valores muy grandes y 300 es un valor que está cerca de la mitad en el rango de valores que puede llegar a tomar α. El valor 20 lo elegimos para ver si al reducir α la tasa de reconocimiento empeora así como el 100 es elegido para ver si aumentar al doble el valor de α nos da una tasa de reconocimiento mucho mayor y, si esto no sucede entonces vemos que pasa con un valor mucho más grande como puede ser 300.

4. Resultados

4.1. Experimento 1

Se procedió a ejecutar el método de kNN variando k, para K = 2, 10 y 20 con k ∈ {2, 30, 100, 500} .

Se observo que la memoria utilizada por el programa era de 127,6 Mb en sus faces iniciales, hasta que en un punto se queda en 380,5 Mb.

Los resultados obtenidos fueron los siguientes:

- Con K = 2

k	Tiempo	Tasa
2	3297,38	0,952476
30	3307,48	0,942452
100	3344,49	0,91681
500	3760,26	0,844905

Tabla 1: Tasas para Knn variando k para dos particiones

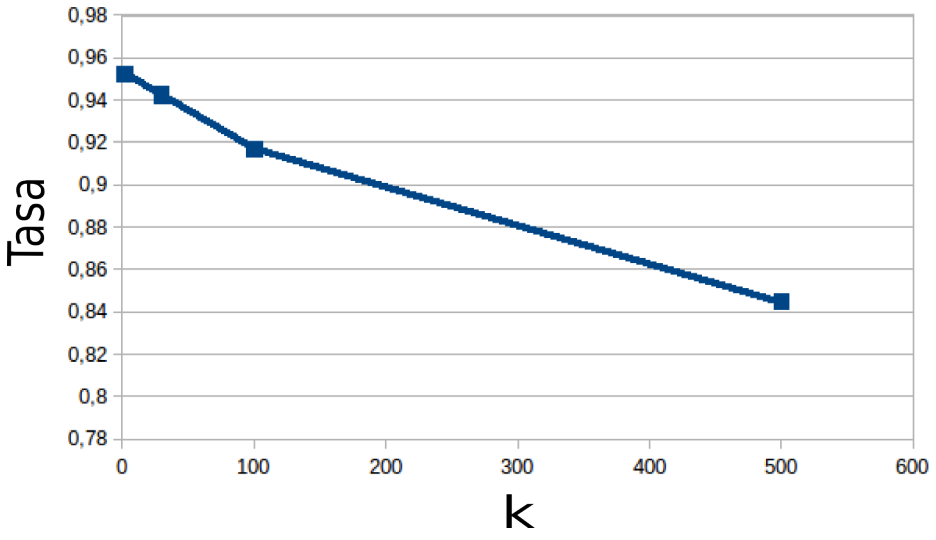


Figura 1: Tasas para Knn variando k para dos particiones

- Con K=10

k	Tiempo	Tasa
2	5853,64	0,96169
30	5881,12	0,952262
100	6026,68	0,93169
500	6845,26	0,878833

Tabla 2: Resultados para Knn variando k para diez particiones

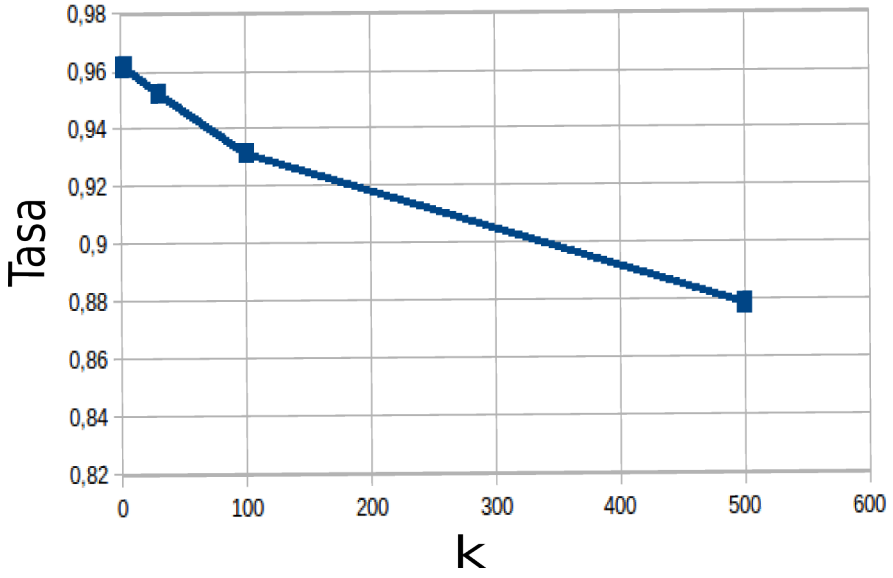


Figura 2: Tasas para Knn variando k para diez particiones

- Con K = 20

k	Tiempo	Tasa
2	44812,6	0,962381
30	44730,7	0,953167
100	26247	0,933429
500	5752,72	0,881381

Tabla 3: Tasas para Knn variando k para veinte particiones

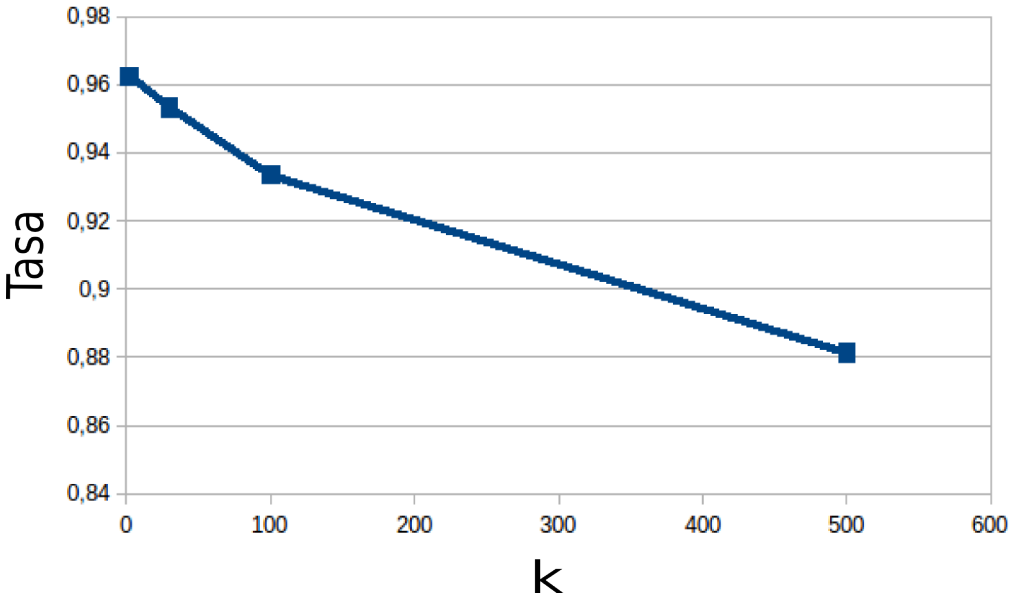


Figura 3: Tasas para Knn variando k para veinte particiones

Podemos ver que en ningún caso se obtiene una tasa del 100 %. Con un k chico se corre el riesgo de que los vecinos mas cercanos no sean los correctos por una situación especial que pueda llegar a ocurrir, y entonces el dígito no sea reconocido. Sin embargo se comprueba empíricamente que con un k grande las tasas disminuyen apreciablemente. Esto se lo podemos atribuir a que hay una probabilidad no despreciable de que un dígito a reconocer este mas cercano a un dígito distinto que a uno igual. No es una cuestión de falta de dígitos iguales al que se esta reconociendo. Por ejemplo al tomar 500 vecinos mas cercanos, no es que no halla sido posible tomar a estos 500 como el dígito a reconocer por falta de este ultimo en la matriz de train, sino que en lugar de tomarse un dígito igual, se toma uno distinto, pero mas cercano.

También podemos ver que al incrementar k se obtiene un tiempo mayor. Esto se debe a que en el algoritmo de knn se debe en cada paso, recorrer un vector de tamaño k donde se encuentran los mas cercanos, ir actualizandolo.

4.2. PCA + KNN

4.2.1. Variando k

A continuación, presentamos las tablas correspondientes a las tasas de reconocimiento y el tiempo de cómputo para los valores de k elegidos en el desarrollo y para los distintos valores de K:

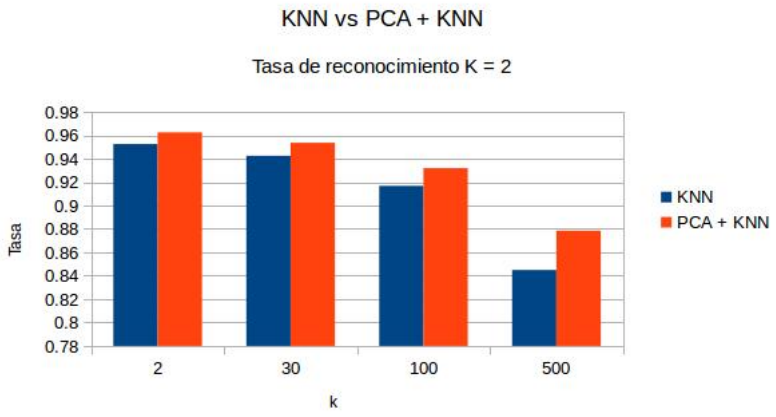
Tasas:

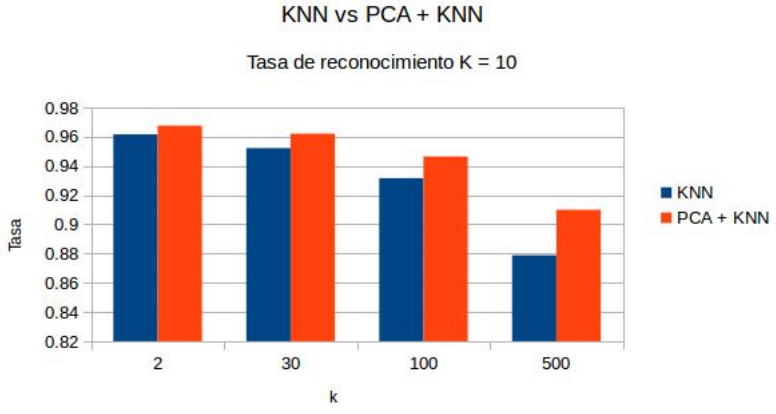
k	K = 2	K = 10	K = 20
2	0.9625	0.96769	0.968214
30	0.953595	0.962143	0.962357
100	0.931976	0.946493	0.945905
500	0.878476	0.910008	0.906643

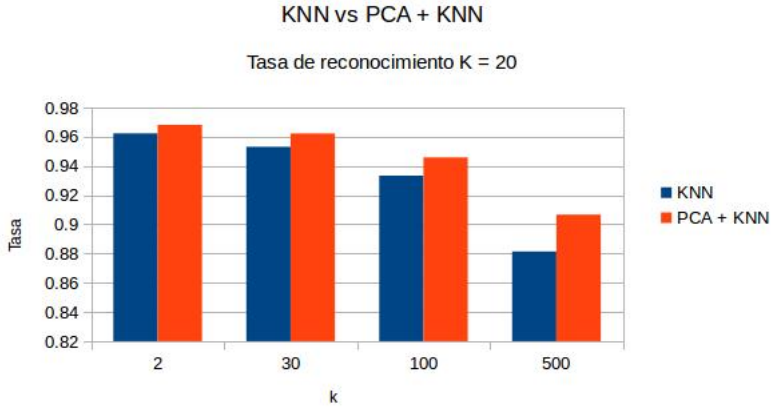
Tiempos:

k	K = 2	K = 10	K = 20
2	6717.76	12842.6	27005.4
30	7167.14	12968.6	26608
100	8099.62	13892.1	28122.2
500	13445.4	16788.3	30181.4

Ahora vamos a comparar las tasas de PCA + KNN con las de KNN para los distintos K:

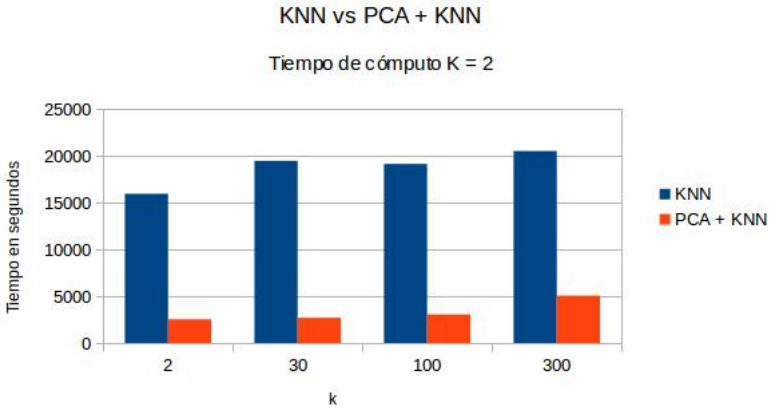


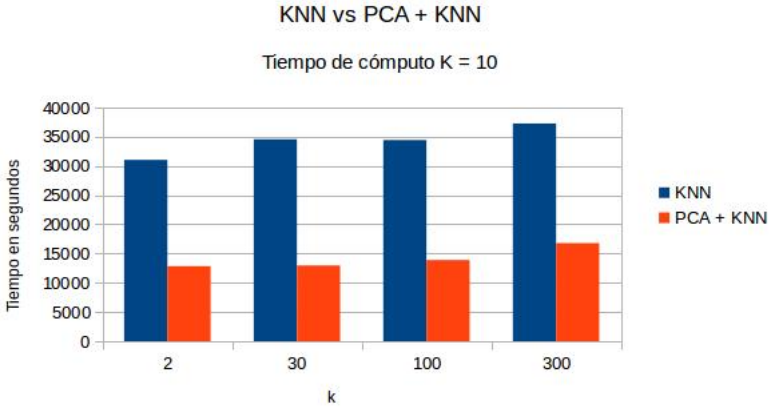


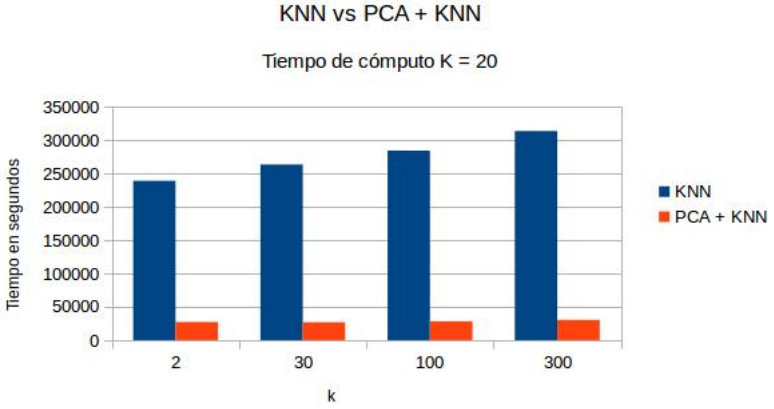


Podemos observar como sorprendentemente PCA siempre supera a KNN, teóricamente esto no debería suceder.

A continuación presentamos un gráfico que compara los tiempos de KNN y PCA + KNN para $K = 2$ y $K = 10$, fijando $\alpha = 50$ en PCA y variando k:







En los gráficos muestran lo esperable que es que los tiempos de KNN son más grandes que los de PCA + KNN. Para los gráficos sobre $K = 20$, el tiempo de cómputo fue calculado de manera aproximada (multiplicando por 20 el tiempo que se tarda en procesar una partición de las 20) debido a que es demasiado grande. La tasa de reconocimiento en este caso tampoco es el promedio de las tasas porque no las tenemos, si no que es la tasa de reconocimiento de la partición particular que procesamos.

4.2.2. Variando α

A continuación, presentaremos la tablas de los distintos K para los cuales corrimos PCA + KNN variando α. En estos experimentos siempre vamos a estar utilizando el mismo valor de k que va a ser 2 ya que fue elegido por ser el que maximiza la tasa de reconocimiento y minimiza el tiempo de cómputo.

Alfa	20	50	100	300
K = 2	0.95269	0.962524	0.95831	0.956952
K = 10	0.957619	0.96769	0.965524	0.965214
K = 20	0.958595	0.968214	0.966524	0.965929

Vemos como no se cumple la hipótesis que planteamos de que si aumentamos el valor de α entonces va a aumentar la tasa de reconocimiento ya que en los tres casos la tasa más alta está asociada a $\alpha = 50$ y es estrictamente mayor a las tasas con $\alpha = 100$ y $\alpha = 300$.
Veamos los tiempos de cómputo asociados a estos valores de α:

	Alfa = 20	Alfa = 50	Alfa = 100	Alfa = 300
K = 2	2524	2693	3043	5052
K = 10	11234	12842.6	16167.8	34122.9
K = 20	27604.4	27005.4	33377	61360.8

Se puede ver en el gráfico como al hacer un aumento significativo en el α el tiempo de cómputo puede llegar a aumentar bastante. En los tres casos vemos que trabajar con $\alpha = 300$ requiere un tiempo de cómputo cercano al doble de lo que requiere con $\alpha = 20$. Esto significa que para valores cercanos de α el tiempo de cómputo no se va a ver afectado de manera significativa pero si realizamos aumentos grandes entonces es probable que el tiempo aumente de manera lineal.

Dado que nosotros queremos encontrar la mejor configuración tanto a nivel calidad de los resultados como tiempo de cómputo y vimos que las tasas variando α en 20, 50 100 y 300 son muy parecidas podríamos elegir $\alpha = 20$. De esta manera estaríamos minimizando el tiempo de cómputo y obteniendo una tasa de reconocimiento bastante alta. Sin embargo, en un contexto donde el tiempo de cómputo puede llegar a ser mucho más grande, por ejemplo porque estamos trabajando con un K mucho mayor, porque contamos con bases de datos de mayor tamaño o por cualquier otro motivo, sería interesante poder reducir lo máximo que podamos al valor de α. Es decir, nosotros sabemos que las tasas con $\alpha = 20, 50, 100$ y 300 son muy similares, pero ¿cuanto más podemos disminuir α sin que la tasa de reconocimiento baje demasiado?. Para contestar esta pregunta decidimos disminuir el valor de α de una manera más drástica y ver que sucede. A continuación, se presentan las tasas de reconocimiento asociadas a los distintos K, para $\alpha = 5$:

K	Tasa de rec.
2	0.667595
10	0.651357
20	0.650857

Los resultados demuestran que la diferencia de las tasas con respecto a los valores anteriores de α es claramente notable, lo que significa que deberíamos elegir un α entre 5 y 20. Es importante observar como al realizar una variación en el valor de α, las tasas para los distintos valores de K se mantienen muy parecidas. Esto significa que para reducir el tiempo de cómputo de la experimentación, podemos a tratar de encontrar este α utilizando $K = 2$ y lo que obtengamos también nos va a servir para $K = 10$ y $K = 20$. Probemos con $\alpha = 7$, 13 y 16.

Alfa	Tasa de rec.
7	0.821143
13	0,926857
16	0.944714

Bien, la tabla muestra que para $\alpha = 13$ la tasa es ≈ 0.92. Dado que con un valor de 7 estamos pasando el 0.8 y con 13 estamos pasando 0.92, tomando un valor de $\alpha = 10$ nuestra tasa va a ser ≈ 0.9, lo cual es más que aceptable. Partimos de un valor de $\alpha = 50$ y experimentando, llegamos a que si bajamos ese valor a 10 entonces vamos a tener una tasa un poco peor pero igual muy buena y vamos a estar reduciendo mucho más la dimensión de nuestras muestras.

5. Discusión

5.1. Hipótesis 1:

Con respecto a la hipótesis 1, que establecía que corriendo cualquiera de los 2 métodos implementados(kNN y kNN+PCA) fijando k y α, a mayor valor de K mayor porcentaje de tasas, procederemos primero a observar los resultados de kNN+PCA.
Comparando las tasas de efectivad resultantes de usar K=2 contra K=10 y K=20, se nota claramente que las de K=2 son menores que las de los otros dos valores de K. Esto es consecuencia principal del uso de la técnica de K-Fold Cross Validation, ya que con un valor K chico nuestra partición de la base de entrenamiento se divide en partes mas grandes, en consecuencia la longitud de la base de test tiende a ser igual a la base de entrenamiento. Dicho en otros términos, tenemos menos elementos en la base de entrenamiento para entrenar nuestro algoritmo y menos permutaciones sobre cuales elementos los consideramos parte del test. Esto es la principal causa de que con K=2 obtengamos los menores valores de la tasa de efectivida. Para que la tasa de efectividad tienda a 1, es necesario contar con una base lo más extensa posible. Y además al aumentar K, tenemos mas combinacion de particiones posibles y un entrenamiento más profundo. En este caso, la disminucion de la misma no se ve afectada por el valor que tome k y alpha.
Hasta aqui la hipótesis se confirmaría, pero al analizar y comparar las tasas de K=10 contra K=20, observamos que aca si incide el valor que tome k. Para k con valores chicos, se ve una diferencia de la tasa cuando saltamos de K=10 a K=20, pero a medida que k aumenta, las tasas empiezan a comportarse de manera similar, teniendo vaivenes para algunos k, en el que la tasa de K=10 le gana a K=20 o viceversa. Ante esta situación, decidimos observar más detalladamente los datos de la experimentación y logramos ver que para k grandes y K=20, la muestra de las tasas de cada partición posee una mayor varianza, es decir que los valores tienen mayores fluctuaciones en comparación a la media de la muestra. Realizando un análisis más intensivo, deducimos que esto se debe a que a mayor valor de K, tengo mas elementos de entrenamiento, y a su vez a mayor valor de k, tengo más elemento que entrarán en la definición de vecinos más cercanos, pudiendo aqui encontrar elementos que no son del mismo label que el elemento en estudio y así alternar notablemente la tasa de efectividad en cada una de las particiones.
Como último dato, el valor de α no incide en las tasas de efectividad entre diferentes K, es decir, para un α fijo, a medida que se aumenta el valor de K, aumenta la tasa.
Con respecto al método de kNN, la elección de los K, se comporta de la misma manera y tiene las mismas consecuencias que en el método de kNN+PCA. Finalmente se pordría concluir que la hipótesis queda parcialmente verificada, ya que la única parte que se refuto de la misma fue la parte de mantener k constante.

Análisis de tiempo de computo al variar K: En el método de kNN+PCA, claramente se incrementa el tiempo de cómputo a medida que aumenta el valor de K, esto se debe a que el cálculo de la matriz de covarianza del entrenamiento domina gran parte del tiempo del metodo. Dicha matriz se calcula de la siguiente forma: por ejemplo, si K=2, la matriz de entranamiento va a tener un tamaño de 21000x784, su transpuesta de 784x21000, entonces la matriz de covarianza se calcula multiplicando la transpuesta por la matriz de entrenamiento resultando siempre una matriz de 784x784. Para este caso de K, haremos 2 multiplicaciones de matrices de 784x21000 y 21000x784. Ahora bien, si suponemos que K=10, el procedimiento es el mismo, salvo que ahora, mi base de entrenamiento va a ser mas grande, por ende, su matriz también que este caso va a tener tamaño de (42.000-42.000/10)x784 = 37800x784. Para este caso de K, haremos 10 multiplicaciones de matrices de 784x37.800 y 37.800x784. Al aumentar el valor de K, además de hacer más multiplicaciones por tener más particiones, las matrices aumentan de tamaño por lo explicado anteriormente. Para K=20 es el mismo procedimiento.
En el método de kNN, por ejemplo para K=2, voy a recorrer 2 veces la base de entrenamiento de 21000 dígitos para un base de test con la misma cantidad de dígitos. Para k=10, voy a recorrer 10 veces la base de train de 37.800 dígitos para una base de test de 4.200 digitos. Realizando un análisis más exhaustivo, para el primer caso de K=2, estariamos recorriendo 21.000 veces la base de 21.000 digitos, todo eso 2 veces lo que resultaria en 2*21.000*21.000 = 822.000.000 iteraciones. Para el caso de K=10, estariamos recorriendo 4200 veces la base de 37800 digitos, todo eso 10 veces

lo que resultaria en 10*4.200*37.800 = 1.587.600.000 iteraciones, lo que implacarría un 93 % más de iteraciones. Para K=20 pasa lo mismo.

5.2. PCA + KNN

5.2.1. Variando k

Al realizar las variaciones de k propuestas en el desarrollo obtuvimos distintas tasas de reconocimiento que, como podemos observar en la primera tabla, van empeorando a medida que aumenta el k. Esto pasa porque, aunque estemos haciendo PCA primero, luego vamos a estar aplicando KNN y ya vimos que cuando aplicamos KNN y aumentamos el valor de K, las tasas empeoran, independientemente del preprocesamiento de los datos.

5.2.2. Variando α

En los experimentos vimos que nuestra hipótesis de que al aumentar el valor de α ibamos a estar aumentando la tasa de reconocimiento es falsa. Sin embargo, cuando bajamos de $\alpha = 50$ a $\alpha = 5$ pudimos notar una diferencia importante en la tasa de reconocimiento. Una posible explicación a lo que está sucediendo es que, si bien matemáticamente tiene sentido pensar que deberíamos tener tasas de reconocimiento más altas al aumentar α (porque agrandamos nuestra base en la cual expresamos nuestros datos), nosotros estamos implementando todo esto en una computadora. Esto significa que todo lo que hacemos tiene cierto error y que ese error puede ir incrementandose.

Los autovectores por ejemplo, los calculamos mediante el método de la potencia, pero no debemos olvidarnos que el método de la potencia se basa en aproximar un límite. Esto quiere decir que los autovalores y autovectores que estamos calculando, en realidad no van a ser exactamente los que queremos obtener. En la experimentación vimos que a partir de un cierto valor, ya no tiene sentido seguir aumentando α porque las tasas dan muy similares (incluso peores) y el tiempo de cómputo aumenta. Tranquilamente lo que podría estar pasando es que, los primeros autovectores que estemos calculando tengan menos error (porque el método de la potencia para ellos converge más rapidamente) y a medida que se va aumentando la cantidad de autovectores, el error que se produce al calcular cada uno de ellos es cada vez más grande. Esto explicaría lo que pasa en nuestros resultados porque los primeros autovectores de la base son los que estarían aportando más información (porque son los que menos error tendrían) y a partir de un punto, el error es tan grande que ya no aporta información confiable y puede ocurrir entonces que la tasa de reconocimiento empeore.

5.3. Competencia Kaggle

Para realizar nuestro submission en la página de Kaggle, a la competencia de Digit Recognizer, modificamos levemente el codigo de nuestro programa. El mismo se encuentra en la carpeta Kaggle. Las modificaciones se deben a que en este caso, la base de test proporcionada por la página no tiene labels, no utilizamos el método de K-Fold Cross Validation, la funcion kNN ahora tiene que devolver las predicciones de los dígitos, entre otras cosas.
Luego de toda la experimentación concluímos que los mejores parámetros, en cuanto tiempo y eficacia, para correr el programa son algún k y α cercanos a:

- $\alpha = 50$
- $k = 2$

A continuación probaremos con valores cercanos a estos parámetros para estudiar la efectividad que nos proporciona el sistema de Kaggle.

- Con k=2 y α=50, tuvimos una efectividad de 0.96686

- Con k=3 y α=50, tuvimos una efectividad de 0.96857
- Con k=3 usando solo kNN, tuvimos una efectividad de 0.97200
- Con k=4 y α=50, tuvimos una efectividad de 0.97286

6. Conclusiones

7. Apéndices

Referencias

[1] Faires, J. D. and Burden, R. L. Análisis Numérico, 7rd ed. Teorema 9.10 y Corolario 9.11. Pág 555, año 2011.

[2] Faires, J. D. and Burden, R. L. Análisis Numérico, 7rd ed. Pág 560, año 2011.

[3] Faires, J. D. and Burden, R. L. Análisis Numérico, 7rd ed. Teorema 9.15. Pág. 570, año 2011.