DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP2 de Métodos Numéricos

3 de mayo de 2024 Métodos Numéricos

Grupo: 21

Estudiante	LU	Correo electrónico
Lucas Mas Roca	122/20	lmasroca@gmail.com
Juan Ignacio Ponce	420/21	juaniponce0@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Introducción

En este trabajo práctico nos proponemos resolver un problema de reconocimiento de dígitos, utilizando algunas técnicas de Machine Learning con los conocimientos vistos en la materia. Usaremos kNN (K-Nearest Neighbors) para reconocer una imagen de un dígito desconocido (base de *test*), clasificándolo como algún dígito posible en base su semejanza a dígitos ya conocidos (base de *train*). Complementaremos este proceso con el método de PCA (Primary Component Analysis) y compararemos los resultados obtenidos con los resultados de utilizar solamente kNN.

El objetivo de este informe es comprender, desde un punto de vista computacional, cómo funcionan los modelos estadísticos predictivos, conseguir un mejor entendimiento del funcionamiento de los métodos utilizados y qué tanto puede mejorar realmente la performance del procesamiento de imágenes al usar otras herramientas como PCA, que en la teoría son más eficientes y rápidos.

Vamos a medir el tiempo de ejecución y distintas métricas vistas en la materia: accuracy, precision, recall, F1-score y kappa de Cohen, este último nos servirá para poder comparar ambos métodos. Experimentaremos con distintos parámetros dentro de cada método para intentar de mejorar las métricas y buscar mejores parámetros. Finalmente, usaremos lo aprendido en la etapa de experimentación para participar en la competencia abierta de Kaggle de digit recognition.

2. Desarrollo

El desarrollo de este trabajo práctico fue realizado en C++ utilizando la librería Eigen recomendada por la materia para operaciones matriciales y vectoriales, esto ayuda mucho a mejorar el tiempo de ejecución, ya que las operaciones están muy optimizadas. Por otro lado, para la experimentación de nuestros algoritmos, usaremos una base de datos de la página web Kaggle, del cual obtendremos una muestra de 42k imágenes previamente etiquetadas con el dígito que representan. Esto representará la base de entrenamiento (*train*) y otras 28k imágenes sin sus etiquetas que representan la base de prueba (*test*). Como nuestro objetivo es experimentar con un modelo de predicción de aprendizaje supervisado, necesitamos que nuestra base de train como de test ya se encuentren etiquetados, por lo que usaremos solamente las 42k imágenes de la base de datos de entrenamiento e iremos experimentando con distintas particiones para nuestra base de train y test intentando no caer en overfitting ni underfitting. Para esto último implementamos en Python el método de K-Fold Cross Validation que nos permite obtener las métricas de nuestro clasificador a partir de varias particiones distintas (*folds*) de una misma base.

2.1. kNN

Como la idea del trabajo práctico es implementar un clasificador de dígitos, debemos tener una forma de decidir qué dígito es una imagen basándonos en datos de la base de entrenamiento. Para esto usaremos kNN, el método visto en la materia, el cual consta de comparar la imagen actual a clasificar (llamémosla z) contra todas las imágenes de la base de train (llamémoslas x_i), luego nos quedaremos con las k imágenes de la base de train que más se parecen a z. Para medir cuanto se parecen 2 imágenes, veremos ambas imágenes como una matriz de píxeles de misma dimensión. En este caso, todas las imágenes estarán en una dimensión de 28×28 píxeles, las cuales serán convertidas a vectores de 28 * 28 = 784 elementos, que estarán ordenados como las filas de la matriz original que representan. Tendrán como valor la intensidad de cada píxel (el cual se encuentra entre 0 y 255). Calcularemos la norma 2 de la resta entre cada uno de los valores de los píxeles de nuestra base de train contra los píxeles de nuestra imagen actual. Luego, como buscamos los k vecinos más cercanos,

nos quedaremos con las k imágenes de la base de train que minimicen esa norma 2:

$$min_i \|x_i - z\|_2$$

Podemos quedarnos con los k índices de imágenes que minimizan esa norma 2 o podemos quedarnos con sus labels (nosotros optamos por la segunda opción, en este caso también hay que guardarse el resultado de esa norma 2 para poder compararlas con el resto para iterar sobre la base de train y conseguir los k más cercanos).

Utilizando esos k elementos debemos clasificar a este elemento de la base de test, para hacer esto tenemos varias opciones.

Para esto usamos 2 métodos distintos, el primero es una moda o votación no pesada (se clasifica el elemento como perteneciente a la clase que tenga mayor representación dentro de esos k vecinos más cercanos). El segundo método es una votación pesada, se clasifica el elemento como la clase que tenga menor norma 2 en promedio dentro de los k vecinos más cercanos (la clase que más se acerca en promedio al elemento).

2.2. PCA

Este método visto en la materia consiste en intentar de reducir las dimensiones y reescribir toda la base de test y train en una base nueva. La idea es usar una base que nos ayude a clasificar mejor los dígitos y más rápido. Esto ayuda a reducir las dimensiones de las bases, ya que cada imagen tendrá α *"píxeles"* en vez de 784. Además de hacer el proceso de kNN más rápido (ya que las imágenes son más chicas) ayuda a que la norma 2 de la resta sea más representativa del parecido entre 2 imágenes debido a que baja la dimensión (vimos en clase la "maldición de la dimensionalidad").

La base que usamos para hacer esto es la base de autovectores asociados a los α autovalores de mayor módulo de la matriz de covarianza de las imágenes de entrenamiento ($M_X = \frac{X^t X}{n-1}$ con X siendo la base de train luego de restarle la media a cada columna, donde la columna i-ésima de X representa los valores del píxel i de cada imagen de entrenamiento). En particular, como sabemos que M_X es simétrica podemos decir, por lo visto en la teórica, que existe una base ortonormal de autovectores, a partir de estos autovectores vamos a armar nuestra matriz ortogonal V de cambio de base.

Vamos a usar el método de la potencia para obtener el autovalor de mayor módulo y su autovector asociado, luego usaremos deflación para *"sacar"* ese autovalor de la matriz y buscaremos el siguiente. Repetimos este proceso hasta tener los α autovectores asociados a los α autovalores de mayor módulo. Finalmente, ponemos los autovectores calculados como columnas de la matriz V, la cual usaremos para los cambios de base.

En este proceso no se toman decisiones, solamente se reescriben las imágenes en otra base, la cual intenta de sacar redundancia al quedarse con los autovectores que capturan mayor varianza. Para decidir en este caso usaremos kNN sobre la base de test y train reescritas en la nueva base.

2.3. Método de la Potencia

Teniendo en cuenta que el método de la potencia es un método iterativo, debemos establecer condiciones de corte para poder utilizarlo. Por ese motivo, usamos dos criterios de corte: niter y épsilon, ambos vistos en la materia.

Usamos niter para acotar la cantidad máxima de iteraciones posibles para un autovector, este criterio es un valor hardcodeado en la condición de salida del ciclo. Mientras tanto, usamos épsilon para comparar (norma 2 de la resta de estos vectores, $\|v_k - v_{k+1}\|_2 \leq \epsilon$) el autovector actual con el autovector de la iteración anterior en todas las iteraciones, en caso de que la norma 2 calculada

sea menor a épsilon salimos del ciclo (en caso de que se llegue a niter iteraciones dentro del ciclo se imprime un mensaje de advertencia por consola "no conv" para indicar que algún autovector no convergió en niter iteraciones para su valor inicial).

Otra opción puede ser calcular el autovalor en cada iteración y medir su convergencia en cada iteración ($|\lambda_k - \lambda_{k+1}| \leq \epsilon$), pero esto parece que puede ser más lento, ya que debemos realizar 3 productos matriciales para calcular λ en cada iteración, mientras que para $\|v_k - v_{k+1}\|_2 \leq \epsilon$ solo debemos guardar el vector anterior y realizar una resta de vectores y luego calcular su norma 2. Otra opción sería revisar en cada iteración si la aproximación del autovalor que tenemos ya puede ser considerada suficientemente buena ($Av_k \approx \lambda_k v_k$) pero para esto también debemos calcular el autovalor y hacer algunas operaciones extra ($\|Av_k - \lambda_k v_k\|_2 \leq \epsilon$). Esta última opción probablemente sea la mejor del punto de vista teórico para medir la convergencia del autovector y autovalor, pero también debería ser la más lenta en la práctica.

2.4. Dígitos del grupo

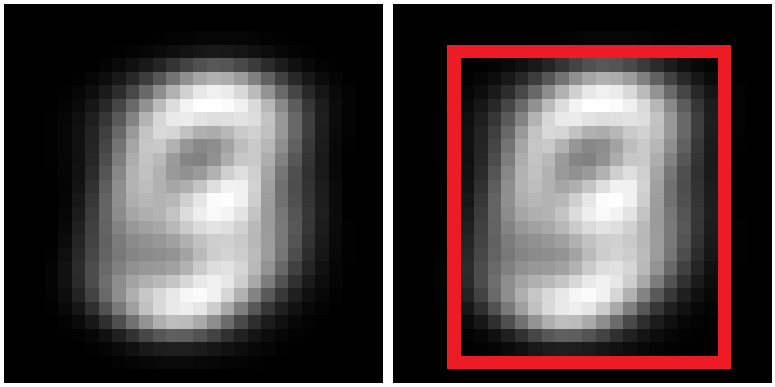
Para este punto dibujamos varios dígitos de distintas formas, algunos fueron dibujados en computadora, mientras que otros fueron dibujados en papel y luego escaneados. Para los que fueron dibujados en papel debemos primero pasarlas por un filtro para sacar los colores, luego invertir las imágenes antes de pasarlas a escala de grises.

Este proceso (invertir y pasar a grises) fue realizado en Python, mientras que para el filtro usamos Cam Scanner para pasar las fotos a blanco y negro. Se incluye el notebook de Python imagenes.ipynb el cual usamos para transformar las imágenes, luego las pasamos a vectores y las escribimos a .csv para usar como test.

Algo para mencionar de este proceso es que algunas de las imágenes son RGB y otras son RGBA, afortunadamente como pasamos por el filtro blanco y negro, las matrices R, G y B tienen los mismos valores, mientras que las matrices A tienen todos sus elementos en 1, con lo cual podemos quedarnos solo con la matriz R. Luego debemos transformarla para que pase de valores entre 0 y 1 a valores entre 0 y 255 enteros, hacer un reshape y finalmente pasarla a vector para luego escribir el vector en el csv.

2.5. Mejoras a kNN

Una posible mejora al método de kNN es recortar las imágenes de la base de test y train de forma de incluir menos borde, quedándonos con menos píxeles que en la mayoría de las imágenes no serán relevantes (ya que tendrán un valor de 0 en la gran mayoría de las imágenes). Para ver cuanto recortar de los bordes, podemos ver la media de la base de train y recortar de forma que saquemos la mayor cantidad de píxeles con valor 0 posibles, por ejemplo:



Si bien este recorte hace que las imágenes no sean cuadradas, esto no genera problema, ya que el código está pensado para imágenes de resolución genérica (incluso imágenes no cuadradas). El objetivo de este recorte es reducir las dimensiones de las imágenes, esto lo hacemos por dos motivos, tiempo y relevancia. Al reducir las dimensiones de las imágenes, se reduce el tiempo de cómputo de kNN (con lo cual se acelera la parte de decisión luego de aplicar PCA, tengamos en cuenta que para valores razonables de épsilon, esta parte es la que más tiempo lleva dentro de PCA)y a su vez lo que estamos sacando no resulta relevante y al reducir las dimensiones además reducimos el efecto de la maldición de la dimensionalidad.

Para implementar esto no hace falta modificar el código de kNN, podríamos simplemente recortar la base de test y train en Python y luego escribirlas en .csv nuevos antes de correr el código.

3. Experimentación

Para todos los experimentos que hagamos a continuación usaremos como base de train la base de datos de Kaggle de la cual solo usaremos las 42k imágenes ya etiquetadas para poder calcular las métricas. De estas 42k imágenes haremos k-fold cross validation con 5 folds para evitar overfitting y underfitting. Luego, para el valor de k usaremos 5 vecinos, con los cuales podremos tomar una mejor decisión sin tener que tampoco usar demasiados. Además, utilizaremos votación pesada como criterio de decisión entre estos k vecinos. Para el valor de alpha, optamos por usar 50 componentes principales que nos permitan tener una gran precisión sin necesidad de tener demasiados componentes que no lleguen a ser relevantes. Usaremos 1e-6 como valor de épsilon para acotar el tiempo de convergencia y disminuir costo de cómputo. Con respecto a PCA, este lo usaremos siempre que el objetivo de nuestro experimento no sea analizar el rendimiento de esta herramienta. Por último, usaremos como niter el valor 10000 para poder obtener un mejor criterio al acotar las iteraciones y a la vez no tener un costo muy alto del rendimiento del algoritmo.

Estos valores para los parámetros servirán como default para correr los distintos experimentos y solo iremos variando los parámetros que se especifiquen en cada subsección.

3.1. Épsilon

Para este primer experimento veremos cómo distintos valores de épsilon afectan la velocidad de convergencia (medimos la cantidad de milisegundos del loop que hace método de la potencia α veces) y la calidad de los resultados, para esto último mediremos accuracy. La idea sería variar solamente el valor de épsilon y en caso de que sea necesario cambiar el valor de niter (en caso de que para algún valor de épsilon aparezca el mensaje "no conv" en consola).

La hipótesis es que mientras más bajo es el valor épsilon, nuestros autovectores son más precisos (su aproximación se debería acercar más al valor real), con lo cual esperaríamos que el accuracy promedio aumente. Por otro lado, el método de la potencia tomará más tiempo en converger, con lo cual tendrá más tiempo de ejecución. Experimentaremos con 6 valores de épsilon entre 1e-1 a 1e-6 y compararemos los resultados obtenidos.

3.2. Alpha y k

Este experimento servirá para hacer una mejor submission en kaggle y también servirá para el próximo experimento. Nos servirá también para comprender cuanto conviene aumentar los parámetros para lograr balancear el trade-off entre tiempo de ejecución y accuracy. Lo haremos tanto para kNN como para PCA. En kNN tenemos un solo parámetro para cambiar: k, mientras que en PCA tendremos otro parámetro más para probar: α, pero además compararemos para ambos votación

pesada contra moda.

Las hipótesis que tenemos son:

- Votación pesada en kNN tendrá más accuracy en general que votación no pesada, independientemente de que se use PCA o no, ya que en este caso tenemos en cuenta el *"parecido"* entre las k imágenes y nos quedaremos con la clase que más se *"parezca"*.
- Alphas más altos en general darán más accuracy al estar agregando más componentes principales para analizar, pero mientras más grande es el valor de alpha, tendremos que calcular más autovectores, con lo cual debemos aplicar más veces método de la potencia y deflaciones, entonces llevará más tiempo. Intuimos que esto va a tener un límite, ya que mientras más grande es alpha, más componentes estaremos agregando que son cada vez menos relevantes (autovectores asociados a autovalores de módulos cada vez más chicos) con lo cual agregaremos más ruido y tiempo de cómputo mientras que reducimos la accuracy.
- En cuanto a los valores de k la hipótesis es que valores de k más altos serán mejores, ya que estamos comparando más números y nos quedaremos con la clase que tenga números más parecidos dentro de esos k vecinos más cercanos. Esto también parece que va a tener un límite, ya que al agregar más números cada vez tendremos más probabilidad de agregar números que no coincidan con lo esperado (ya sea porque están dibujados de forma parecida o están mal etiquetados).

3.3. kNN vs. PCA

En este experimento compararemos los resultados obtenidos entre kNN y kNN + PCA para distintos parámetros, concentrándonos en los mejores parámetros para cada uno que conseguimos en el experimento anterior, compararemos la accuracy de cada uno y usaremos otras métricas como Kappa de Cohen para comparar estos clasificadores. Esto se hará con k-fold cross validation para obtener una comparación de clasificadores haciendo Kappa de Cohen con cada una de las particiones realizadas. Adicionalmente, compararemos el tiempo de cómputo de cada uno de estos clasificadores para cada uno de los folds y lo tendremos en cuenta a la hora de la experimentación.

La hipótesis es que PCA tardara menos tiempo (si tenemos un valor de alpha no muy alto, lo cual intuimos que va a ser así) y a su vez tendrá más accuracy (al menos para alguno de los mejores valores de alpha) que kNN. Lo primero se debe a la reducción de la dimensión, si tenemos un buen valor de alpha se va a reducir la dimensión sobre la cual hacemos kNN y esto lleve a que sea más rápido; lo segundo se debe a que al mirar los α componentes principales, sacaremos redundancia de la base de train y test, ayudando a clasificar mejor las imágenes.

3.4. Imágenes del grupo

En este último experimento usaremos una base de 67 dígitos dibujados por el grupo, los cuales están labeleados, algunos de ellos fueron dibujados en computadora, mientras que otros fueron manuscritos y luego escaneados (ver más en sección de desarrollo, Dígitos del grupo). En particular usaremos PCA y kNN con los mejores parámetros obtenidos del experimento 2, luego aprovecharemos las labels (y que la base de test es chica) para intentar de ver cuanta es la accuracy máxima que podemos obtener.

En caso de que sea necesario, separaremos los dígitos dibujados en distintas categorías para ver si hay alguna característica que haga que el clasificador sea más propenso a equivocarse (por ejemplo, los dígitos en computadora solo tienen 0 y 255 como valores en la matriz y los trazos son de 1 píxel

de ancho).

La hipótesis en este experimento será que es más probable que el clasificador identifique un dígito correctamente si este se encuentra bien centrado, tiene un trazo no muy ancho y no muy fino (al rededor de 2-3 píxeles de ancho) y no está muy rotado. Intuimos que el clasificador identificara mejor a las imágenes manuscritas que las imágenes de computadora por la hipótesis mencionada y que los manuscritos suelen tener valores intermedios entre 0 y 255, mientras que los de computadora solo tienen valores extremos.

4. Resultados

Antes de mostrar los resultados hay que aclarar que a pesar de haber hecho los 5-folds en todos los experimentos, solo visualizaremos uno de estos para cada experimento, ya que todos son bastante parecidos debido a la homogeneidad de la base de MNIST, con lo cual ninguna de las métricas cambia en gran medida entre los distintos folds, por lo tanto, decidimos mostrar un resumen de los folds explicando uno solo por experimento. El objetivo de los folds era asegurarnos de que nuestros experimentos no estén segados por un problema de overfitting, y como en todos los folds de cada experimento obtuvimos resultados semejantes, notamos que no tuvieron ninguno de estos problemas.

4.1. Épsilon

Para este primer experimento veremos una tabla con el accuracy promedio y el desvío estándar entre todos los 5 folds hechos para cada valor distinto de épsilon. Cabe aclarar que cada valor de épsilon está expresado como $1e - i$ donde i es el valor representado en la columna 'épsilon'.

epsilon	accuracy	standard deviation
1	0.963976	0.001633
2	0.963929	0.001147
3	0.964071	0.001170
4	0.964024	0.001272
5	0.964048	0.001308
6	0.964048	0.001308

Figura 1: Tabla de promedios de 5 folds de Accuracy y Standard Deviation para 6 épsilons distintos

Como podemos ver, el accuracy aumenta levemente a medida que el épsilon disminuye, al contrario de su desvío estándar que tiende a incrementar, exceptuando el épsilon 1e-3, el cual presenta más accuracy que los demás, y el 1e-1, que tiene el mayor desvío estándar. A pesar de ello, podemos notar que los accuracy de los diferentes folds no varía mucho, ya que la varianza en cada épsilon es muy baja (todo por abajo de 0.002), y entre los accuracy promedio de los diferentes épsilon tampoco varían demasiado.

Además, veremos el tiempo de convergencia en milisegundos que obtuvo cada épsilon:

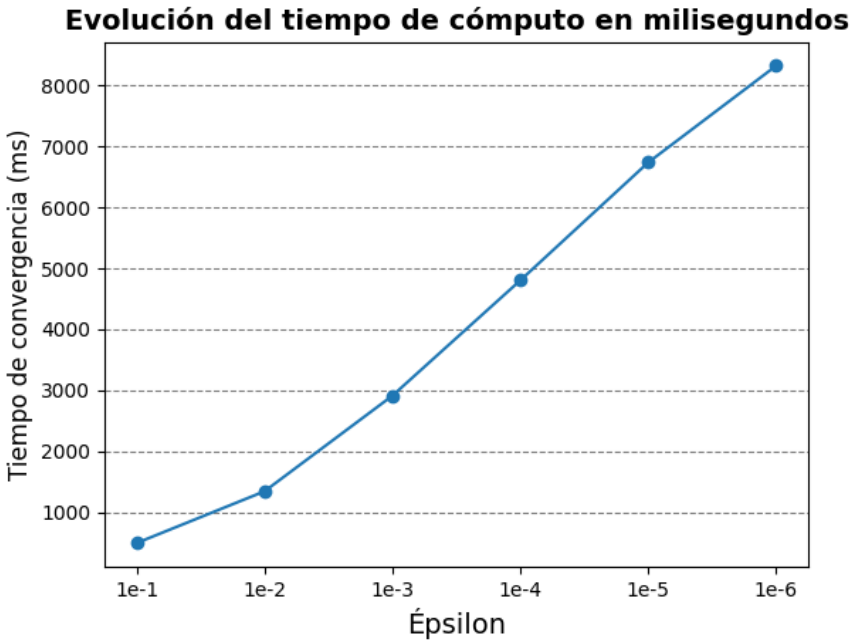


Figura 2: Gráfico del tiempo de cómputo que tarda cada épsilon en converger

En este gráfico se ve claramente el aumento de tiempo que le lleva converger a medida que la cota del épsilon se hace más baja, y este incremento parece tener una forma bastante lineal.

Durante la experimentación inicial, partimos la base de una forma (sin K-fold cross validation) y probamos la accuracy con distintos valores de épsilon y volvimos a partir la base, repitiendo este proceso 5 veces. Al ver los resultados nos encontramos con que mientras más alto épsilon (por ejemplo 1e-1), nos daba mejor accuracy para 3 de los 5 experimentos. Luego de ver esos resultados inesperados, decidimos repetir el experimento con las mismas bases varias veces para ver si esos resultados estaban relacionados con el hecho de que el vector inicial del método de la potencia empieza con un valor aleatorio. Pero para todas las bases los resultados (accuracy) fueron siempre exactamente los mismos para todas las veces que repetimos el experimento. Al ver esto, nos pareció que estos resultados inesperados sean causa de un overfitting.

En cuanto al valor de niter, notamos durante un experimento de prueba que para épsilon entre 1e-6 y 1e-15 tenemos exactamente los mismos resultados de accuracy (y los .csv de salida son iguales) para distintas bases. A pesar de que no cambian nuestras decisiones, cambia nuestro tiempo de cómputo. A partir de épsilon 1e-16 en adelante empieza a aparecer el mensaje *"no conv"* en consola para niter=10000 y empieza a subir aún más el tiempo de cómputo, por este motivo, no consideramos que aumentar el niter mejore nuestro accuracy (pero si incrementara fuertemente el tiempo de convergencia).

Como dato extra de este experimento, mostramos las métricas de accuracy, precision, recall y F1-score para cada clase, es decir, para cada dígito. Esto lo haremos para el fold número 5 con épsilon 1e-6. Esta decisión fue tomada a partir de haber visto que todos los épsilon tenían un margen de accuracy muy parecido (figura 1), por lo que con un épsilon fijo tenemos un buen resumen.

numero	accuracy	precision	recall	f1-score
0	0.996190	0.974537	0.988263	0.981352
1	0.995476	0.969136	0.991579	0.980229
2	0.992857	0.961395	0.963795	0.962594
3	0.990714	0.962069	0.948980	0.955479
4	0.992143	0.963885	0.954377	0.959108
5	0.992381	0.964626	0.949130	0.956815
6	0.996310	0.973934	0.989170	0.981493
7	0.992619	0.965986	0.963801	0.964892
8	0.990357	0.962773	0.935162	0.948767
9	0.987143	0.931604	0.940476	0.936019

Tanto en el número 9 como en el 8 podemos apreciar un claro declive en las métricas a comparación de los otros dígitos. Con lo cual podemos concluir que sin importar el fold o el épsilon, estos números presentan más dificultades para ser reconocidos.

4.2. Alpha y k

Primero presentaremos una visualización de kNN con diferentes valores de k tanto para votación pesada como no pesada (o moda).

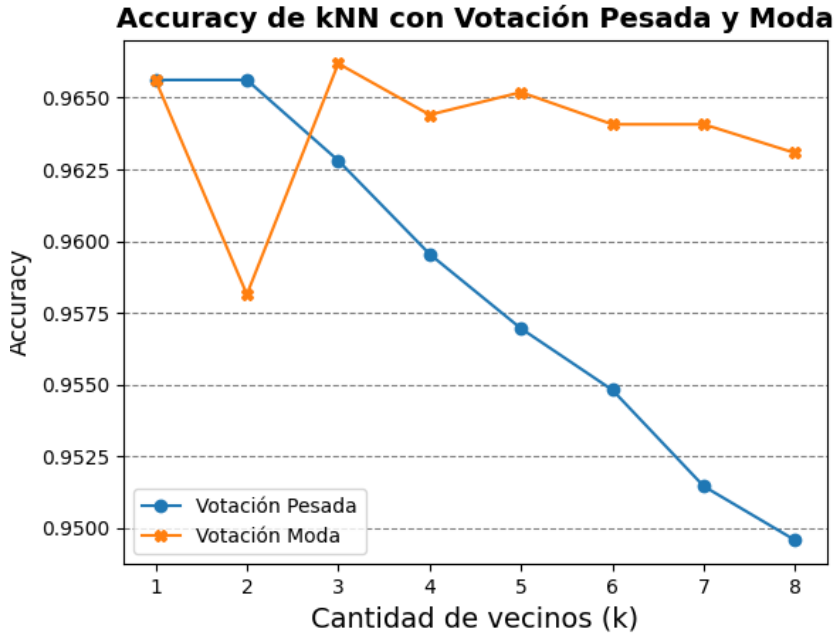


Figura 3: Gráfico del accuracy correspondiente a cada k para los dos métodos de decisión

Acá podemos ver que los mejores valores de k para kNN (sin PCA) son k=1 (con o sin votación pesada da los mismos resultados) y k=3 (con votación moda).

También podemos ver en el gráfico que pasa algo raro para k=2 con votación no pesada, podemos ver que la accuracy baja bastante y luego vuelve a subir. Esto se debe al criterio de desempate usado, al mirar solamente 2 vecinos, van a haber muchos casos donde ocurra un empate en la moda no pesada, y en este caso se usa el criterio de desempate, el cual es quedarnos con el dígito de más chico (prioridad el dígito 0, luego el 1, etc.). Para mejorar esto podríamos cambiar el criterio de desempate a medir los pesos para desempatar por peso.

En cuanto a las hipótesis:

- Votación pesada es mejor que moda: Parece ser que pasa lo contrario si medimos las accuracy promedio, ya que esta baja si usamos votación pesada, pero también baja la varianza si usamos votación pesada.
- k más alto mejor: Esto no se cumple, de hecho pasa lo contrario, los mejores k son números entre el 1 y el 5, como podemos ver en el gráfico anterior (Figura 3).
- Alphas más altos mejor: Esto parece cumplirse hasta cierto punto. Para visualizar esto usamos los k más óptimos sacados de la figura 3 e iremos variando el valor de alpha. Para esto usaremos el k=1 con votación pesada y k=3 con votación no pesada. A pesar de ser los mejores valores para kNN, haremos el siguiente con PCA porque estos valores siguen siendo también los mejores para este otro modelo.

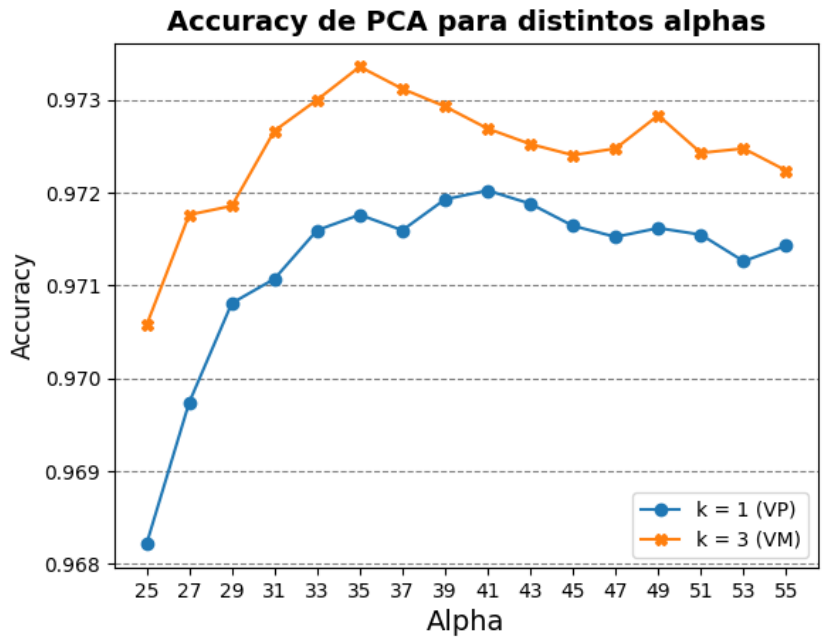


Figura 4: Gráfico del accuracy de distintos alphas para los dos k más óptimos

Al principio la accuracy aumenta a medida que el alpha también lo hace, pero alrededor de 33 a 35 empezamos a ver como baja cada vez más la accuracy para k=3 con votación no pesada, mientras que para k=1 esto sucede a partir del 41.

Además, visualizaremos el tiempo de cómputo en segundos para cada alpha con el objetivo de mostrar que a medida que aumenta el alpha, también aumenta el tiempo de ejecución. El gráfico a continuación fue realizado con los parámetros de mejor accuracy en PCA, los cuales son $k = 3$ y votación no pesada.

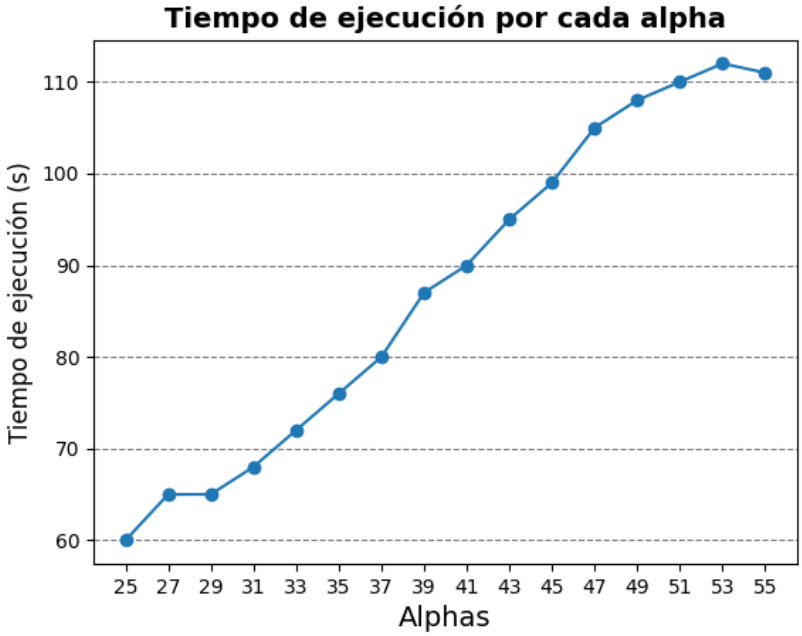


Figura 5: Gráfico del tiempo de ejecución de distintos alphas

4.3. kNN vs. PCA

Para este experimento tenemos que mostrar primero el índice de kappa de Cohen calculado para 5 folds y con los demás parámetros descritos en la sección 3 de experimentación, esto nos servirá para comparar qué tan distintos son estos modelos a la hora de predecir las diferentes clases. Haciendo 5 fold nos dieron 5 índices que no variaban mucho de la media. Su desviación estándar es de 0.04 y su índice promedio es el 0.66, con lo cual podemos decir que ambos modelos presentan una gran similitud a la hora de calcular y decidir sobre las diferentes clases. Esto también lo podemos observar con las siguientes matrices de confusión, las cuales muestran la cantidad de aciertos por cada clase.

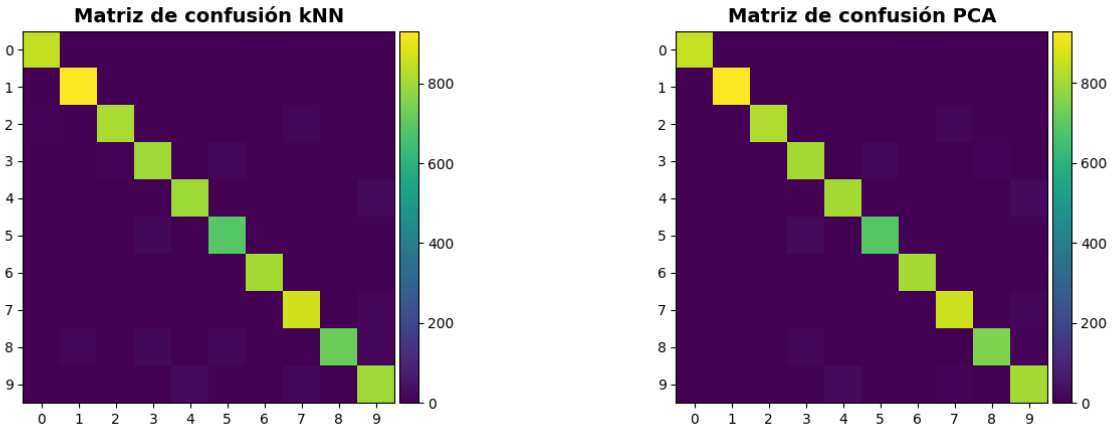


Figura 6: Matriz de confusión de ambos modelos

Acá podemos ver que los gráficos se parecen bastante, con los cual podemos decir que ambos predicen a las distintas clases de la misma manera, por lo que nuestro índice promedio de kappa de

Cohen es bastante coherente.

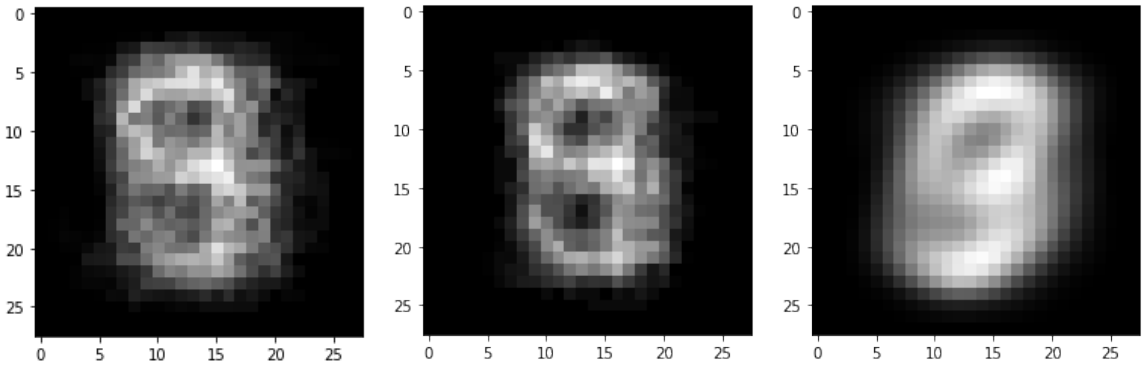
Por otro lado, podemos ver que el tiempo de cómputo sí cambia entre los diferentes modelos. Para ver esto, probamos hacer un bar plot de 5 folds para ver el tiempo de cómputo de cada fold para ambos modelos, pero notamos que todos los folds tardan lo mismo en ejecutarse. Para kNN, su tiempo de cómputo ronda por los 452 segundos, mientras que PCA mejora fuertemente con un tiempo promedio de 16 segundos por fold.

En cuanto a la accuracy, kNN obtuvo un promedio de 0.96667 accuracy con 0.00294 de varianza, mientras que PCA obtuvo un promedio de 0.97336 accuracy con 0.00286 varianza. Podemos observar entonces que PCA obtiene mejor promedio de accuracy, menor varianza y además, menor tiempo de ejecución. Con lo cual, podríamos decir que si usamos PCA con buenos parámetros, este es mejor que usar kNN (sin PCA) en todos los sentidos.

4.4. Imágenes del grupo

Para este experimento nuestra idea inicial era probar PCA sobre esta base de test usando la base de kaggle completa con los mejores parámetros obtenidos en los experimentos anteriores, pero durante la experimentación inicial sobre esta base vimos que tenía muy poca accuracy (comparado a partir la base de train y hacer folds). Por lo tanto, optamos por probar muchos parámetros de α y k, viendo si conseguimos mejorar así nuestra accuracy (aprovechando que la base de test es muy chica y tenemos los labels, podemos decidir rápido con PCA y luego medir la accuracy).

A pesar de esto, no encontramos ninguna combinación de parámetros que supere el 55 % de accuracy, y usar kNN sin PCA no mejora nuestros resultados. Decidimos intentar de centrar mejor las imágenes de nuestra base, veamos la media de esta base de test:



En la izquierda vemos la media de nuestra base antes de intentar de centrarla mejor, en el medio tenemos la media de la base centrada y a la derecha tenemos la media de la base de train de kaggle. Buscamos que la media de la base se parezca más a la media de train de kaggle, podemos ver que al centrar mejor conseguimos mejorar un poco la media. A pesar de esto, los resultados no variaron, con lo cual parece que ese no es el problema.

De todas formas, tener 50 % de accuracy es mejor que decidir de forma aleatoria (si decidimos de forma aleatoria, tendríamos al rededor de 10 % de accuracy). Es posible que la *"baja"* accuracy esté dada por las diferencias entre los dígitos de la base de kaggle y los de nuestra base, probablemente la base de kaggle tenga algún cuidado extra (por ejemplo, rotar o alinear los números de cierta forma).

En cuanto a las hipótesis propuestas en la sección 3.4, el clasificador parecía tener accuracy similar sobre la base manuscrita que sobre la base de computadora, no encontramos un criterio razonable para separar los números que suba la accuracy. Adicionalmente, centrar mejor la base no tuvo mucho efecto en cuanto a mejorar la accuracy.

5. Conclusión

Con los resultados obtenidos durante la etapa de experimentación, podemos ver claramente las ventajas (tanto tiempo como accuracy) de utilizar PCA antes de aplicar kNN, realizando un análisis previo de los datos, logrando un mejor entendimiento sobre la base de datos. En este caso podemos ver también como usar una herramienta que parece tan *"simple"* como kNN obtenemos buenos resultados, obviamente este mecanismo no puede aplicarse para cualquier base (debido a su alto tiempo de ejecución y la maldición de la dimensionalidad). También podemos ver como tener una base de datos bien *"cuidada"* mejora nuestros resultados (ver sección 4.4). Esto último no solo sirve para obtener un mejor accuracy, sino también nos permite obtener otras ventajas de tiempo y costo de cómputo como pudimos observar en la sección 2.5.

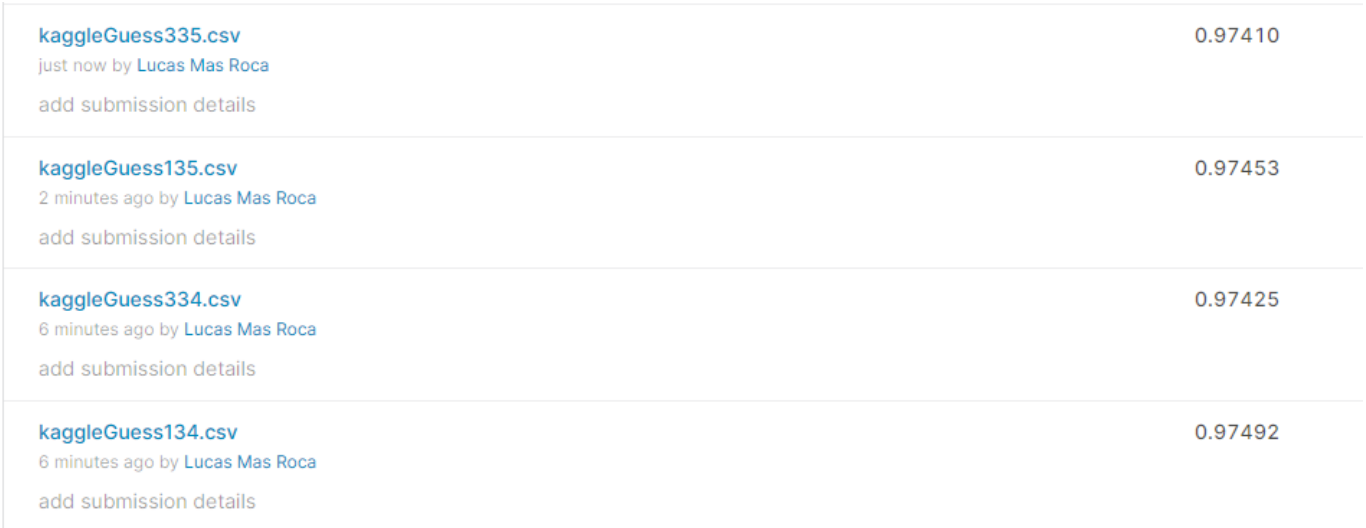
Por otro lado, pudimos observar contradicciones con la teoría de kNN que nos dice que a mayor cantidad de vecinos, mejor criterio de decisión, y mejor predicción. Esto parece ser que no siempre se cumple para toda base de datos, ya que a medida que la base está más balanceada, el mínimo cambio en la decisión puede resultar irrelevante y hasta perjudicial. Podemos ver un efecto similar sobre α de PCA, ya que al poner un valor mas bajo técnicamente estamos tirando mas información, pero pudimos ver en los experimentos que hay un punto en el cual tirar mas información resulta beneficioso para esta base de datos.

Además, vimos como distintos valores de épsilon afectan el tiempo de cómputo y su accuracy, mencionamos también que a partir de ciertos valores de épsilon nuestras decisiones son iguales, pero aumenta nuestro tiempo de cómputo, podríamos concluir que para el criterio de corte de método de la potencia utilizado no vale la pena usar esos valores de épsilon.

También vimos como a pesar de experimentar con distintos parámetros intentando de maximizar nuestra accuracy, no siempre conseguimos los resultados esperados (como vimos en la sección 4.4). Concluyendo también que además de experimentar con distintos parámetros, debemos tener cuidado de que nuestras bases de test y train deben ser similares (no solo cuidando el balanceo de las clases, sino que también cuidando la calidad de las imagenes).

Por último, fuimos capaces de entender un poco más a fondo la implementación de este modelo de predicción para reconocimiento de imágenes y entender las mecánicas y la estadística detrás de un modelo de aprendizaje supervisado.

En cuanto a la competencia de kaggle, decidimos subir archivos para α 34 y 35 con k 1 y 3 sin votación pesada, ya que esos parecían ser los mejores valores en nuestra experimentación. El que dio mejor resultado fue $\alpha = 34$ con $k = 1$ (0.97492 accuracy), nombramos a los archivos kaggleGuess$k\alpha$.csv:



6. Comentarios adicionales

6.1. Cosas raras

Primero vamos a mencionar brevemente algunas cosas *"raras"* o que nos llamaron la atención como para mencionar.

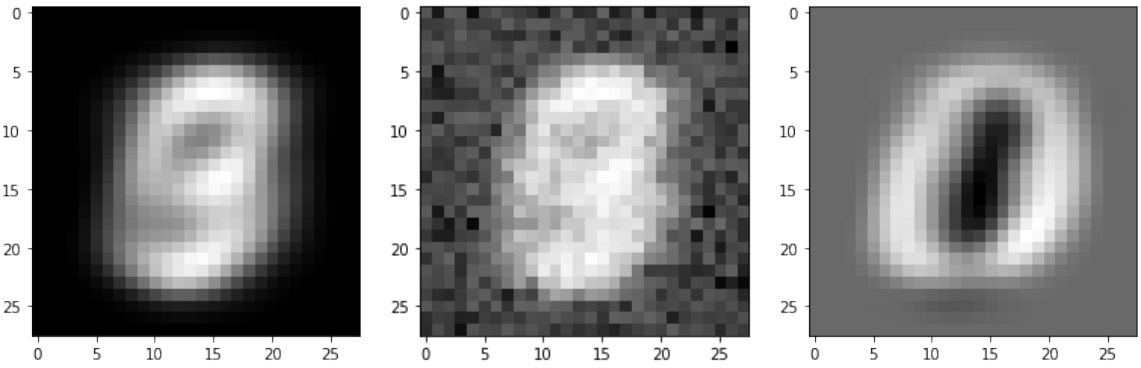
6.1.1. Media?

Durante la etapa de experimentación, nos dimos cuenta de que en el código de PCA al armar la matriz X (antes de armar la matriz de covarianza M_X) en vez de restar la media de cada columna a cada columna de la base de test, restábamos 1 (el vector columna que restábamos a cada columna lo inicializábamos en 1 en vez de μ_i). Sin embargo, los resultados (en cuanto a accuracy) eran parecidos y seguían siendo buenos (en general $\pm 1\,\%$ para varios parámetros, con accuracy al rededor de $97\,\%$ para parámetros "buenos").

Esto nos dio la intuición de que tal vez la media de cada columna era parecida a 1, pero cuando probamos imprimir la media de cada columna, la mayoría de las medias estaban entre 20 y 50, con lo cual la media no es parecida a 1.

6.1.2. Primer autovector

Otra cosa para mencionar, la cual está parcialmente relacionada con la sección anterior, es que durante los experimentos iniciales nos dimos cuenta de que no estábamos centrando correctamente la base de train (PCA) al calcular M_X. A pesar de esto, teníamos una muy buena accuracy ($+97\,\%$ para parámetros de α y k en un rango "bueno", ver sección 3.2). Sin embargo, notamos que pasaba algo raro con el primer autovector:



La imagen de la izquierda era el primer autovector antes de corregir el error de la sección anterior (6.1.2), esta imagen es idéntica a la media de la base de train de kaggle (no incluimos una imagen porque es idéntica). Luego de arreglar ese error conseguimos que el primer autovector era la imagen del medio (esto sigue sin ser lo esperado), la cual se parece mucho a la imagen de la izquierda (con ruido). Finalmente, nos dimos cuenta de que estábamos calculando de forma errónea el μ, con lo cual terminábamos restando ruido, luego de corregir esto último conseguimos que nuestro primer autovector sea el de la imagen de la derecha.

Durante todo este proceso, el segundo autovector del código viejo (ambos códigos viejos) era algo muy similar (más bien idéntico) al primer autovector del código nuevo. De hecho, parecería que todos los autovectores siguientes también eran los mismos, nada más que están desplazados por uno (ya que los códigos viejos tenían el primer autovector parecido a la media y los siguientes coincidían con

los del código nuevo). Podríamos pensar que al centrar los datos estamos realizando un proceso con un efecto similar al de una deflación y así "sacarnos" ese autovector, y este primer autovector nos sirve sacarlo, ya que para intentar de diferenciar números no aporta nada, ya que la mayoría de los dígitos serán muy parecidos a ese autovector (ya que es la media).

6.2. Cosas extra

El código de C++ está pensado para imágenes de resolución genérica (podrían incluso no ser cuadradas) con lo cual si usamos el ejecutable con un .csv de imágenes de entrenamiento labeleadas de la misma forma que en kaggle (ponemos en el primer renglón label,pixel0,pixel1,...,pixel$m$ para imágenes de tamaño $m = filas * columnas$, los siguientes renglones tendrán los valores correspondientes para cada imagen) y un csv de test de la misma forma que kaggle (primer renglón pixel0,pixel1,...,pixel$m$ para imágenes de tamaño $m = filas * columnas$, los siguientes renglones tendrán los valores correspondientes para cada imagen) podemos clasificar dígitos en distintas resoluciones sin modificar el código.

Además de esto, con algunas modificaciones del código podemos hacer que el clasificador sirva para identificar caracteres (letras y números manuscritos), para esto debemos pasar los labels de int a char y además cambiar parte del código de kNN (el vector count tenía 10 posiciones representando cada dígito posible, hay que modificar para poder poner todos los labels posibles).

Finalmente, el ejecutable compilado que incluimos está compilado con la versión optimizada del código, que tiene la posibilidad de guardar y leer tanto los autovectores como los kVecinosMasCercanos de archivos .csv. Vamos a entregar ambas versiones del código (tp2.cpp y tp2OPT.cpp) porque la versión no optimizada del código es mas fácil de leer. Este código optimizado toma argumentos opcionales, con lo cual se puede usar de forma idéntica al código no optimizado si usamos comandos sin estos argumentos opcionales. Notemos que el código no optimizado no funciona si intentamos poner los argumentos opcionales, con lo cual si intentamos usar ese código puede que alguna de las funciones implementadas en Python se rompan. El propósito de este código es reusar cosas que ya calculamos, pero debemos tener cuidado al usarlo y saber cuando podemos usarlo (debemos usarlo sobre una misma base de test y train, teniendo en cuenta que el tiempo de ejecución medido será muy bajo).