Laboratorio de Métodos Numéricos

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

Trabajo Práctico 2

Digit recognizer

Integrante	LU	Correo electrónico
Hosen, Federico	825/12	federico.hosen@gmail.com
Palladino, Julián Alberto	336/13	julianpalladino@hotmail.com
Rajngewerc, Guido	379/12	guido.raj@gmail.com
Silvani, Damián Emiliano	658/06	dsilvani@gmail.com

En este trabajo práctico se desarrolló y eva luó una herramienta de reconocimiento óptico de caracteres (OCR) para el capturar dígitos manuscritos en imágenes. En particular, se estudió la base de datos de dígitos MNIST proporcionada por Kaggle. Se implementó el algoritmo de aprendizaje automático k-NN, y se empleó Análisis de Componentes Principales para reducir la dimensionalidad. En la evaluación se utilizó validación cruzada con k-fold para obtener una medida del accuracy y F1-score con diferentes valores de los parámetros.

OCR **MNIST** **k-NN** **PCA** **K-fold**

Índice

1. Introducción

El objetivo del trabajo que aquí se presenta consiste, en primer instancia, en el desarrollo de una herramienta de Reconocimiento óptico de caracteres (OCR, por sus siglas en inglés *Optical Character Recognition*) que permita reconocer dígitos escritos a mano en imágenes. Además el grueso del trabajo está en la experimentación de los distintos métodos y sus parámetros, que forman parte de la herramienta desarrollada, con la motivación de poder hacer una evaluación de la misma.

El conjunto de datos analizado es un subconjunto de la base de datos MNIST, que está compuesta por 42000 imágenes etiquetadas de 28×28 píxeles y que representan cada una a un dígito del 0 al 9. Las imágenes que fueron provistas fueron preprocesadas para que los dígitos estén centrados y con fondo negro. La herramienta que se desarrolló es un clasificador multiclase k-NN (*k-nearest neighbours*), en el que se utilizó Análisis de Componentes Principales (PCA, *Principal Component Analysis*) para reducir las dimensiones. El modelo se evaluó utilizando validación cruzada con *k-fold*, y *accuracy* y *F1-score* como métricas principales.

2. Desarrollo

El desarrollo del trabajo se dividió en dos partes: clasificación con k-NN y reducción de dimensionalidad con PCA. En la implementación se reutilizaron estructuras de datos, clases y funciones empleadas en el trabajo práctico anterior, tales como la clase `Matrix`, extensiones sobre la clase `vector`, implementaciones eficientes de multiplicación por matriz transpuesta, transposición, norma vectorial, etc.

2.1. Clasificación con k-NN

La clasificación se hace por medio de la función `knn`, que:

- Recibe un conjunto de entrenamiento, un vector de etiquetas correspondiente al conjunto de entrenamiento, un elemento de test a etiquetar, y un parámetro k que representa cuántos vecinos cercanos debe considerar.
- Calcula la distancia euclidiana entre el elemento de test y todas las muestras del modelo. Estas distancias se ordenan, y toma los k elementos más cercanos.
- Calcula la moda entre los k elementos más cercanos, y según cual sea, devuelve esa etiqueta.

2.2. Reducción de dimensionalidad con PCA

Dado que el algoritmo k-NN es susceptible a vectores de alta dimensionalidad y que puede ser costoso también en términos computacionales trabajar con vectores de 784 elementos, se implementa *Principal Component Analysis* para reducir la dimensión de las muestras.

Principalmente se definieron dos funciones: `pca` para obtener las componentes principales, y tc para aplicar el cambio de base.

2.2.1. Función `pca`

- Calcula la matriz de covarianza como es descripta en la presentación del TP.
- Realiza α veces:
  - Usa método de la potencia para calcular el autovalor más grande en módulo, y su autovector asociado.
  - Almacena el autovalor y autovector obtenidos.
  - Usa deflación en la matriz, para que ahora el segundo autovalor más grande pase a ser el primero.
- Finalmente devuelve los primeros α autovectores de la matriz de covarianza, y sus autovalores asociados[1].

[1] Con *primeros* nos referimos a los autovectores de autovalor más grande en módulo

2.2.2. Función `tc`

Ya teniendo los primeros α autovectores obtenidos de `pca`, la función realiza el cambio de base de la muestra dada multiplicando los autovectores con la muestra.

2.3. Método de la potencia

2.3.1. Convergencia

El primer criterio de parada que definimos fue el de iterar una cantidad finita de veces. Después de varias pruebas con el conjunto de tests de PCA provistos por la cátedra, consideramos que con iterar 300 veces lográbamos obtener en la mayoría de los casos autovalores muy parecidos a los esperados (con cierto margen de error). Luego consideramos agregar un chequeo de convergencia en cada iteración: si el autovalor no cambió demasiado con respecto al valor en la anterior iteración, termina. Esto mejoró considerablemente los tiempos puesto que vimos que la mayoría de las veces los valores convergían en menos de 100 iteraciones. Finalmente establecimos como valor máximo 1000 para mejorar la aproximación.

2.3.2. Errores en el método de la potencia

El método de la potencia, al calcular autovalores de manera aproximada, tiene una muy pequeña chance de fallar en dos casos, los cuales son:

Caso de vector inicial malicioso: Como fue visto en la materia, existen algunos casos en que, partiendo de un vector malicioso, el procedimiento no converja a una solución. En nuestra implementación del método de potencia, se genera un vector aleatorio con distribución uniforme. Luego de consultarlo concluimos que las chances de que esto ocurra son **demasiado** pequeñas y despreciables, en principio porque la probabilidad de que sea uno que cause que el método no converja es muy baja. Además los errores de precisión que se van acumulando en cada paso a su vez decrementan esa probabilidad.

Caso de α muy elevado: Al realizar PCA con un α muy cercano a la dimensión, es posible que el método de la potencia falle. Esto ocurre porque, tal como está cubierto en la bibliografía [1], cuando se divide por la norma del vector que se usa para iterar, es posible que la norma de cero, resultando en un error en tiempo de ejecución de división por cero. Esto ocurre únicamente cuando el autovalor que se esté calculando sea cero. O bien, por problemas de precisión, el autovalor no es exactamente cero, aunque extremadamente cercano a cero.

Luego de consultarlo, concluimos que es un caso que pierde interés en nuestro TP, ya que un autovector de autovalor 0 (o casi 0) no tiene incidencia en las componentes principales, y ocurre únicamente cuando se llama a la función con un α absurdamente alto. En consecuencia, nos conformamos con hacer levantar una excepción cuando se esté dividiendo por cero, y en la experimentación evitamos aplicar PCA con valores de α tan extremos.

2.4. SVD para encontrar las componentes principales

Correspondiendo al respectivo punto opcional, mostraremos la relación entre la factorización SVD y PCA.

Sea M_x nuestra matriz de covarianza. Sea $M_x = U\Sigma V^t$ su factorización SVD. Vamos a demostrar que V es la misma matriz que obtenemos al diagonalizar M_x.

Recordemos que el proceso de diagonalización tiene como motivación buscar los autovectores de M_x, de los cuales tomaremos los primeros[2] α, que serán las componentes principales que estamos buscando.

Para ello, definimos un cambio de base para nuestra matriz de datos X, dada por una matriz P. Nuestra matriz con la base cambiada resulta en $\hat{X}^t = P.X^t$. Y, justamente, la P que tomamos es una $\hat{V}$, la cual tiene como columnas los autovectores de M_x, ordenados según el módulo de su autovector asociado, y normalizados.

[2]Con *primeros* nos referimos a los de mayor autovalor asociado en módulo

2.4.1. Demostración

Lo que queremos demostrar es que $\hat{V} = V$. O sea, que V:

1. Tiene las columnas normalizadas
2. Sus columnas son los autovectores de M_x, ordenados por el módulo de su autovalor asociado de mayor a menor.

El primer item vale por definición de SVD: Si la factorización resulta $M_x = U\Sigma V^t$, tanto U como V deben ser matrices ortogonales. Por lo tanto, V tiene las columnas normalizadas.

Pasamos a demostrar entonces el segundo ítem. Por definición de SVD, las columnas de V son los autovectores de $M_x^t M_x$, ordenados descendentemente según el módulo de sus autovalores asociados. Entonces, nos queda ver que:

1. Los autovectores de $M_x^t M_x$ son los mismos que los autovectores de M_x.
2. Sus respectivos autovalores están en el mismo orden, según sus módulos.

Demostramos el primer ítem:
Sea λ un autovalor de M_x, v su autovector asociado. Veamos que de hecho, v es autovector de $M_x^t M_x$, y encontremos su autovalor asociado:

$$M_x^t M_x \ . \ v = M_x M_x \ . \ v = M_x \ \lambda \ v = \lambda^2 \ v$$

Donde el primer paso vale porque M_x es simétrica, y los otros valen porque λ es autovalor de M_x con v autovector asociado.

Hemos demostrado que si v es autovector de M_x con λ asociado, entonces v es autovector de $M_x^t M_x$ con λ^2 asociado. Demostramos la vuelta:

Supongamos que no vale: Sea v autovector de $M_x^t M_x$ pero no de M_x.

Observemos que, como ambas M_x y $M_x^t M_x$ son simétricas, ambas tienen una base de autovectores. Además, como ambas matrices tienen la misma dimensión, tienen la misma cantidad de autovectores, y por lo tanto $E(M_x) = E(M_x^t M_x)$.[3]

Ya demostramos que todos los autovectores de M_x también son autovectores de $M_x^t M_x$, entonces:

$$E(M_x) \supseteq E(M_x^t M_x)$$

Por lo tanto:

$$\#E(M_x) \leq \#E(M_x^t M_x)$$

Recordamos que habíamos asumido que v es autovector de $M_x^t M_x$ pero no de M_x. Como sabemos que ambos conjuntos de autovectores son base, quitarle un elemento a una base implica quitarle una dimensión. Lo cual significa que:

$$\#E(M_x) < \#E(M_x^t M_x)$$

Lo cual es absurdo! Ya que habíamos dicho que $E(M_x) = E(M_x^t M_x)$. Vino de suponer que $\#E(M_x) \not\geq \#E(M_x^t M_x)$.

Resta ver el segundo ítem: Que el orden queda igual según el módulo de sus autovalores.

Sean $|\lambda_1| > \ldots > |\lambda_n|$ los autovalores de M_x. Si elevamos todos términos al cuadrado, se preserva la desigualdad. Queda:

$$(\lambda_1)^2 > \ldots (\lambda_n)^2$$

Ya que x^2 es una función creciente para números igual o mayores a cero. Por lo tanto, se preserva la desigualdad de los autovalores, y entonces el orden de las columnas es igual.

En consecuencia, V y $\hat{V}$ tienen las mismas columnas, y en el mismo orden.

Por lo tanto, $V = \hat{V}$

[3]Notamos $E(A)$ como el subespacio generado por los autovectores de A.

3. Experimentación

3.1. Métricas elegidas para experimentar

- Accuracy
- F1-Score

Luego de una investigación sobre métricas en el caso de clasificadores multiclase, se eligió *F1-score* como complemento de Accuracy. F1-score es la "media armónica" entre *precision* y *recall* y por lo tanto combina las dos métricas en una sola. Se define como:

$$2\,\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Que el valor sea cercano a 1,0 quiere decir que tanto precision como recall son altos.

Dado que estamos tratando con muchas clases, se decidió usar F1-score con *macro average*, donde se calculan las métricas para cada clase, y finalmente se toma el promedio de los resultados. A diferencia del *micro average*, el macro average es susceptible a clases desbalanceadas, pero como no sucede en este caso (el conjunto de datos de entrenamiento está balanceado), nos fue indistinto.

También se calcularon matrices de confusión, para analizar cualitativamente los resultados y hacer observaciones sobre como se comporta nuestro clasificador en casos problemáticos.

3.2. Diseño de los tests

3.2.1. Estructuras utilizadas

Para realizar las experimentaciones utilizamos **Python 3**, sobre todo para aprovechar las herramientas de evaluación de modelos y métricas provistas por la biblioteca de aprendizaje automático **Scikit Learn**. Para esto creamos una clase `KnnClassifier`, que implementa las siguientes funciones:

- `fit`, que toma una lista de datos y una lista de etiquetas, correspondientes al training del programa. Almacena ambas listas, para cuando se ejecute la predicción.
- `predict`, que toma una lista de datos, correspondientes al conjunto de test del programa. Ejecuta el binario en $C++$, utilizando los datos almacenados previamente en `fit` para el training, y los datos recibidos como parámetro como test. Al terminar el proceso, lee los resultados del archivo de predicciones, los carga y los devuelve.

3.2.2. Reducción del training set

Para iniciarnos en la experimentación del trabajo práctico, comenzamos utilizando valores bajos para k (de k-NN) y α, y usando como training la base de datos entera. A pesar de que los valores de los parámetros eran extremadamente bajos (lo cual implica mucho menos nivel de cómputo), cada test tardaba demasiado en correr (aproximadamente media hora).

Por lo tanto, decidimos realizar los experimentos utilizando solamente 10.000 elementos del conjunto de training. Para ello, utilizamos la función de Scikit-Learn, `StratifiedShuffleSplit`, la cual nos permite quedarnos con 10.000 elementos del conjunto de training, con una cantidad balanceada de cada etiqueta.

Observemos que los resultados que obtengamos de los tests con 10 mil elementos no necesariamente serán los mismos que se obtengan del conjunto de training entero. De todas maneras, consideramos que los resultados obtenidos serán proporcionales y representativos a los obtenidos con el conjunto entero. Esto será confirmado en el segundo experimento obligatorio.

3.2.3. Semilla para K-Fold

Para realizar los K-Fold, se utilizó siempre la misma semilla para realizar los cortes aleatorios sobre el training set. Esto es para poder comparar los resultados de los experimentos que se vayan realizando, sin correr riesgo de que algún resultado se vea alterado por una mejor/peor partición que haya tocado hacer en el K-Fold.

3.3. Primer experimento obligatorio

3.3.1. Sin PCA, variando k

Como primer experimento oficial, realizamos el más simple y prioritario: Correr el programa sin utilizar PCA, variando el parámetro k. Utilizamos las herramientas ya descriptas, y realizamos K-Fold con la función de Scikit-Learn `cross_validate`, con su valor de K igual a 5.

Para acelerar las pruebas, tomamos una muestra de 10.000 observaciones de todo el conjunto de entrenamiento. En el experimento 2 se muestra que dicho número es suficiente para que los siguientes resultados sean representativos de todo el dataset.

Queremos ver como afecta el score la cantidad de vecinos. Es por eso que evaluamos el algoritmo sin PCA, con k variando entre [5.,1000]. Se tomó 1.000 como máximo dado que es la cantidad (aproximada) de cada clase en el training set.

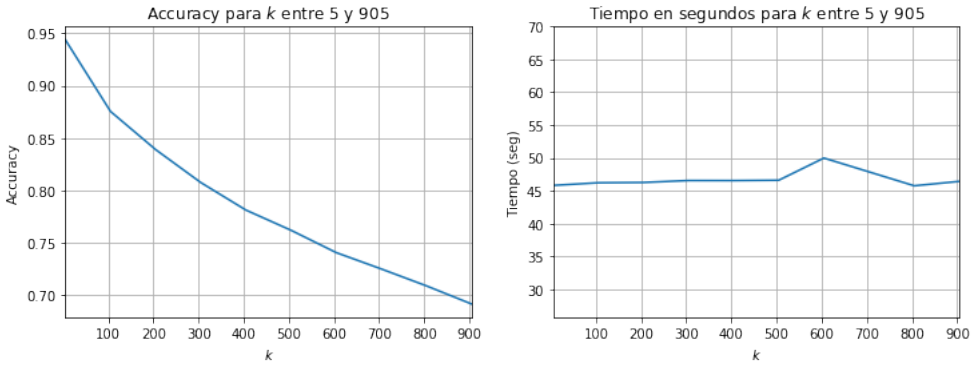


Figura 1: Variando k entre 5 y 905

Ya teniendo una pinta global del score con respecto al k, notamos (figura 1) que en los primeros valores están los mejores valores se encuentras en los k más chicos.

En cuanto a los tiempos, salvo un pequeño *outlier* en $k = 600$, observamos que k no modifica el tiempo de cómputo del programa (se hará un mayor análisis en la segunda parte del experimento).

Refinamos nuestra búsqueda del k óptimo, y hacemos hincapié en los primeros valores:

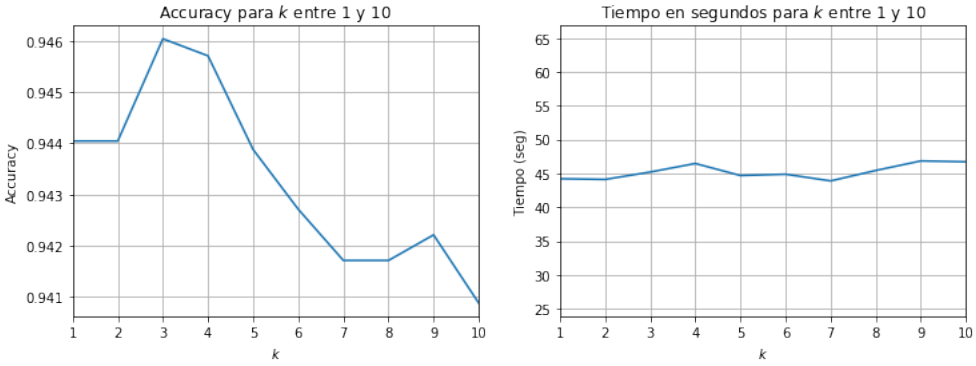


Figura 2: Variando k entre 1 y 10

Queda claro de la figura 2 que con $k = 1$ y $k = 2$ es demasiado pequeño, y que nuestro pico de optimalidad se encuentra en $k = 3$ con un Accuracy de 0,946, seguido con cercanía por $k = 4$, que difieren apenas en el cuarto decimal. Ya con $k = 5$ los valores comienzan a bajar abruptamente, y no parecen opciones viables.

3.3.2. Con PCA, variando k y α

Luego de probar los resultados sin utilizar PCA, analizamos qué ocurre si lo utilizamos. Para ello, debemos ejecutar el algoritmo variando tanto k como α, y chequeando para cada combinación su calidad resultante, evaluada según las métricas que elegimos: Accuracy y F1-score.

En el anterior experimento, encontramos que el rango óptimo del k está entre los primeros valores. Realizamos ahora algunos experimentos preliminares, para ubicar los rangos óptimos de k y α.

Corremos primero con $k = 3$, $\alpha = 30$.

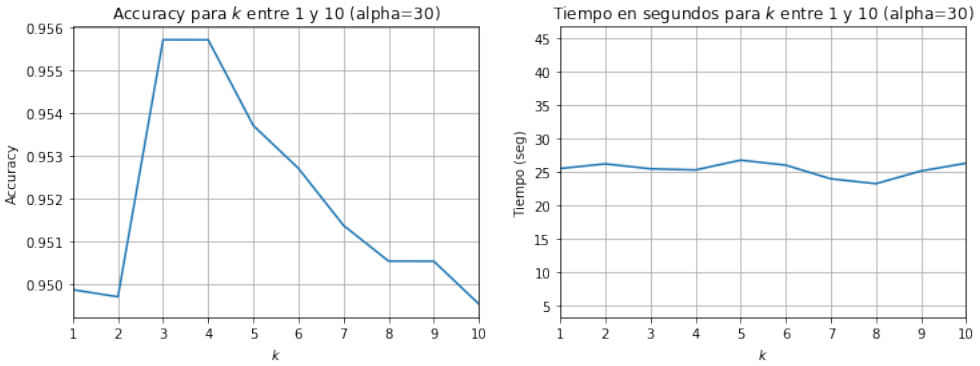


Figura 3: $\alpha = 30$, variando k entre 1 y 10

En la figura 3, de manera similar que antes, encontramos: Primero, que el k no afecta el tiempo de cómputo. Segundo, que el pico de optimalidad de k parece estar entre 3 y 4. El accuracy máximo nos resulta de 0,956.

Repetimos el experimento con un α más grande.

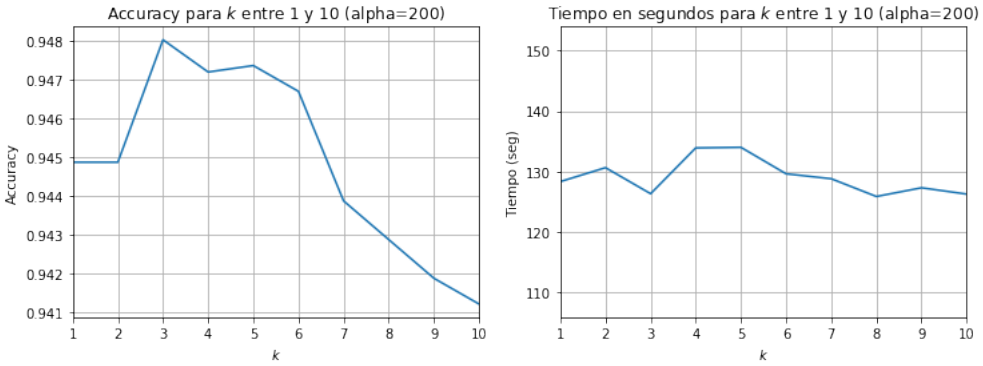


Figura 4: $\alpha = 200$, variando k entre 1 y 10

En la figura 4, nuevamente el comportamiento es similar: Para $k = 3$, $k = 4$ se tienen los mejores valores y a partir de ahí decrece el accuracy. En comparación con el anterior, el accuracy decreció (el pico es 0,948, antes era de 0,956). Con lo cual nos dice que quizás un α tan alto nos hace perder precisión. Entonces, realizamos el mismo experimento pero con α muy chico, en particular, igual a 2.

En la figura 5, los resultados con un α tan pequeño nos resultan muy pobres (un pico de 0,44). Como el máximo está en el borde del gráfico, probamos ampliar el rango de k para ver si mejora con valores mayores.

Ahora sí, en la figura 6 encontramos nuestro pico de optimalidad para k con α fijado en 2: Resulta $k = 200$. Aún así, con un Accuracy de 0,460, pierde con gran diferencia contra la combinación de $\alpha = 30$. Ya tenemos una idea del rango de nuestros parámetros: α tan chico como 2 ofrece resultados muy pobres. $\alpha = 200$ es levemente peor que valiendo 30. Probamos entonces con distintas combinaciones de α entre 10 y 100, buscando achicar nuestro rango. A su vez, siendo que los k pequeños funcionaban mejor con $\alpha = 30$, definimos el rango de k del 1 al 10.

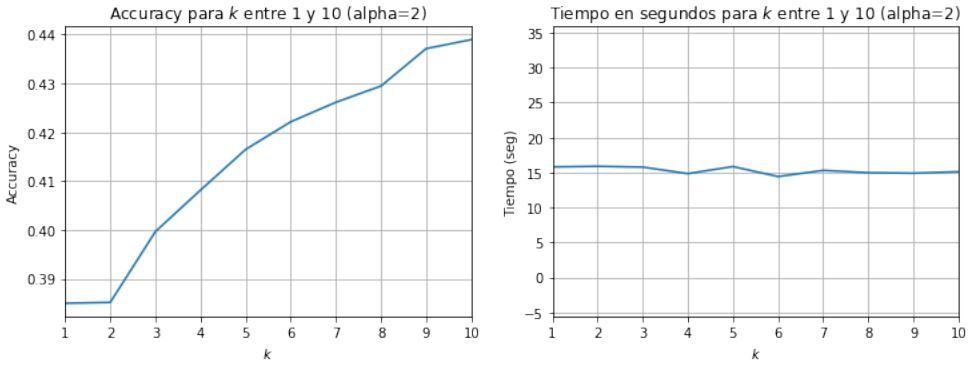


Figura 5: $\alpha = 2$, variando k entre 1 y 10

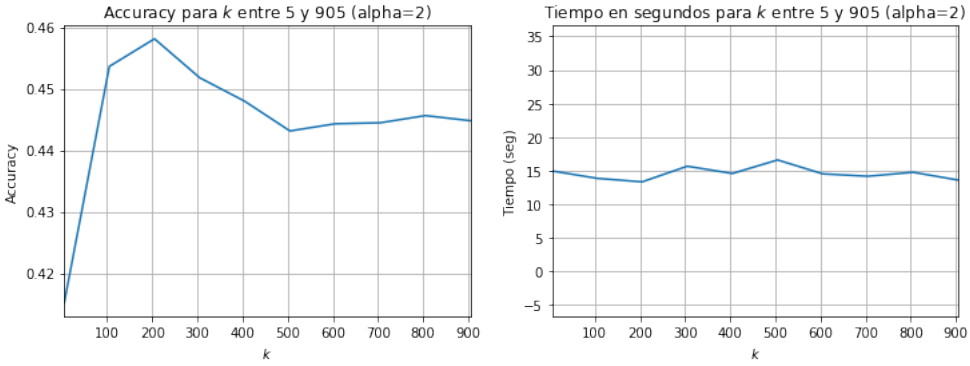


Figura 6: $\alpha = 2$, variando k entre 5 y 905

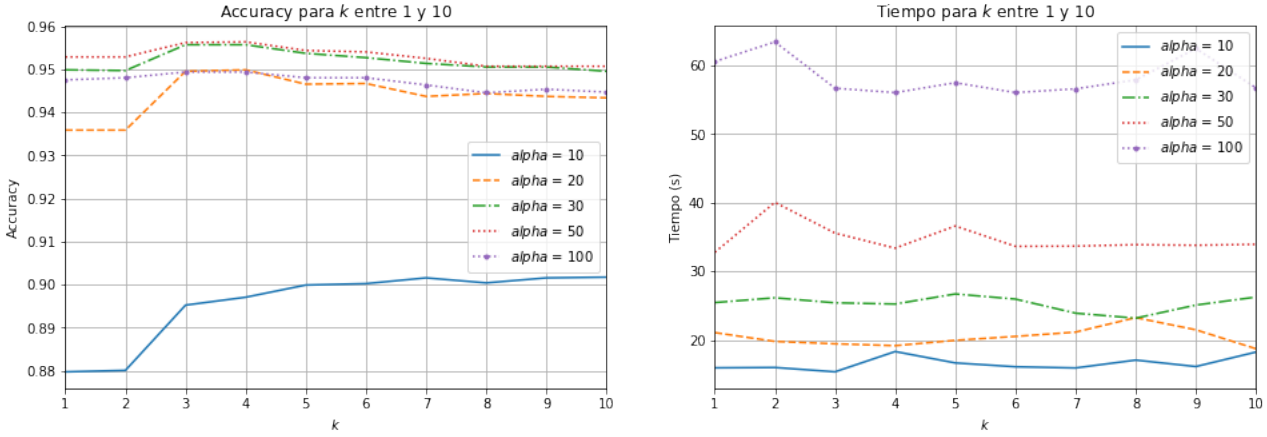


Figura 7: $\alpha = (10, 20, 30, 50, 100)$, k del 1 al 10

En la figura 7 podemos observar, en primer lugar, que $\alpha = 10$ queda descartado como un parámetro viable. Además, notamos que para el resto de los parámetros, $k = 3$ y $k = 4$ son ofrecen el pico de optimalidad, sin importar el α. Finalmente, apreciamos que $\alpha = 30$ y 50 parecerían empatar en dichos valores óptimos de k, dejando atrás a $\alpha = 20$ y $\alpha = 100$ por 0,06 puntos de score. Esto último nos sugiere que 20 y 100 son valores muy extremos para α, por debajo y por arriba, respectivamente.

Por otro lado, la medición de tiempos nos indica que no importa el k que tomemos, el tiempo de cómputo que tarde el programa va a depender más bien de α.

Análisis más detallado con *heatmaps*

Ya teniendo una idea de los rangos en lo que nos estamos manejando, buscamos hacer un análisis más detallado, con un rango más acotado para los valores.

Como nos encontramos con un gráfico de 3 dimensiones (k, α y resultado), realizamos un *heatmap*, que nos permite observar como se combinan las tres variables a la vez, y con más exactitud que los gráficos ya utilizados.

Se deben elegir los rangos de k y α de forma tal que se vean claramente los valores óptimos y notando que si los alteramos, tanto aumentando o reduciendo, estamos empeorando el resultado.

Para encontrar un rango adecuado nos basamos en el experimento anterior, y lo definimos de la siguiente manera:

- Variamos k desde 1 a 10, de a un valor.
- Variamos α desde 10 a 100, de a 10 valores.

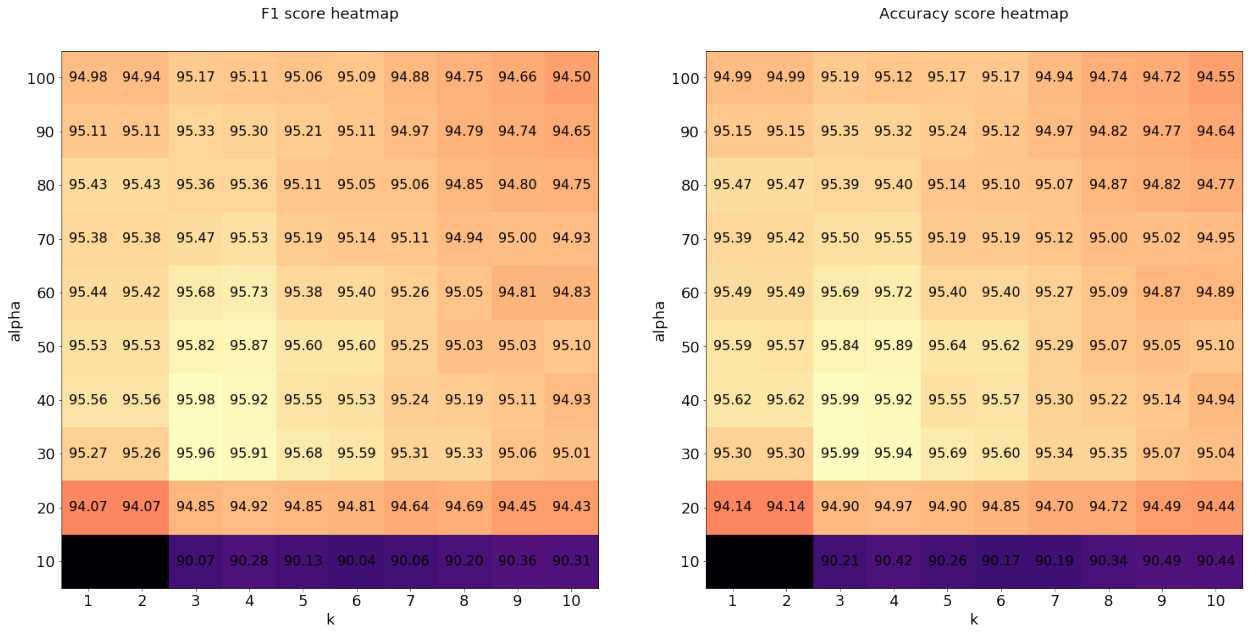


Figura 8: Heatmaps del F1 score y del accuracy. El eje horizontal es el k de k-NN, el eje vertical el α de PCA. Para que se noten más las diferencias, los resultados fueron multiplicados por 100 en ambos casos. El training set fue de 10.000, con K de K-Fold = 5.

Lo primero que debemos apreciar en los resultados es que efectivamente elegimos bien el rango de testeo. Esto se hace observando que, en ambos gráficos, las casillas de mejor resultado se encuentran en el centro del gráfico para ambas coordenadas, y no en un extremo de ellas (lo cual indicaría que quizás el verdadero óptimo no está en el rango).

Lo siguiente que se observa con claridad son los peores casos: En cuanto a α, para ambas métricas tomar 10 resulta muy malo, y ya desde $\alpha = 20$ se observa una diferencia de aproximadamente 4 puntos de score en ambos gráficos. Por otro lado, los peores k parecen empatar entre 1 y 10, los cuales no tienen una diferencia tan marcada con sus respectivos vecinos.

Pasamos ahora a buscar la mejor combinación de (k, α) para cada métrica: En cuanto a F1-Score, los mejores resultados parecen vivir con α entre 30 y 50, y k entre 3 y 4. En este pequeño rango, la mayor diferencia es de 0.14 puntos de score, y la combinación óptima resulta ser (40, 3) con un resultado de 0,9598.

En cuanto al Accuracy, también ocurre que los mejores resultados viven con α entre 30 y 50, y k entre 3 y 4. En dicha familia, la mayor diferencia es de 0.15 puntos de score, y la mejor combinación también resulta ser (40,3) con un resultado de 0,9599.

Sorprende un poco que F1-Score y Accuracy coincidan en la mejor combinación, pero es posible, considerando que el accuracy influye en el F1-Score. De haber analizado según Recall y Accuracy quizás nos habríamos topado con un *trade-off* entre ambas.

Según estos resultados, podríamos concluir que nuestra combinación óptima es (40,3). Sin embargo, considerando la cercanía con los resultados de su alrededor, decidimos refinar nuestra búsqueda repitiendo el experimento, pero restringiendo el rango de los parámetros a la familia de combinaciones óptima ya descripta. Los rangos quedarían:

- K entre 3 y 4
- α entre 30 y 50, tomando de a 2 valores.

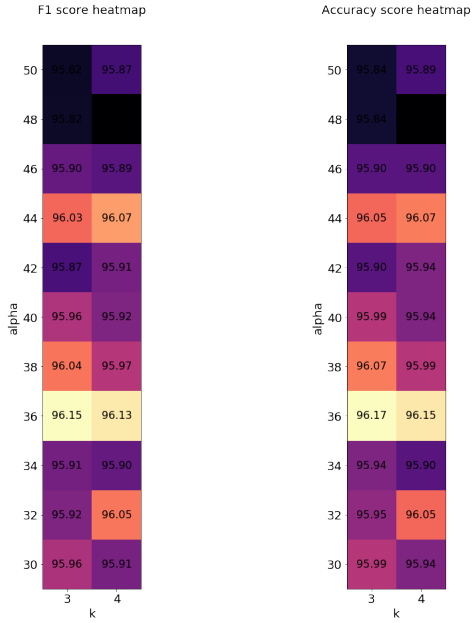


Figura 9: Heatmaps del F1 score y del accuracy, con rango reducido. El eje horizontal es el k de KNN, el eje vertical el α de PCA. Para que se noten más las diferencias, los resultados fueron multiplicados por 100 en ambos casos. El training set fue de 10.000, con K de K-Fold = 5.

Encontramos entonces una combinación óptima más refinada: $\alpha = 36$, $k = 3$, resultando en F1-Score = 0,9615 y Accuracy = 0,9617.

Notamos que aún podríamos seguir probando con valores más refinados. ¿Qué ocurriría con $\alpha = 37$ y $k = 3$?

Para probar esto, decidimos correrlo directamente en *Kaggle*. Por tener mucho mayor cardinal de training set, es esperable que los resultados sean mejores en general. Por ello, probamos con α 36 y 37 y comparamos cuál da mejor en Kaggle. Los resultados fueron:

- Para $\alpha = 36$, $k = 3$ dio un Accuracy de 0,97528.
- Para $\alpha = 37$, $k = 3$ dio Accuracy de 0,97585.

Concluimos que los mejores valores son $\alpha = 37$, $k = 3$.

Evaluando tiempos

Ya habiendo analizado la calidad del algoritmo para distintos valores de k y α, ahora analizamos sus tiempos de ejecución. La pregunta que buscamos responder es: ¿Qué influye más en el costo temporal del programa? ¿k o α?

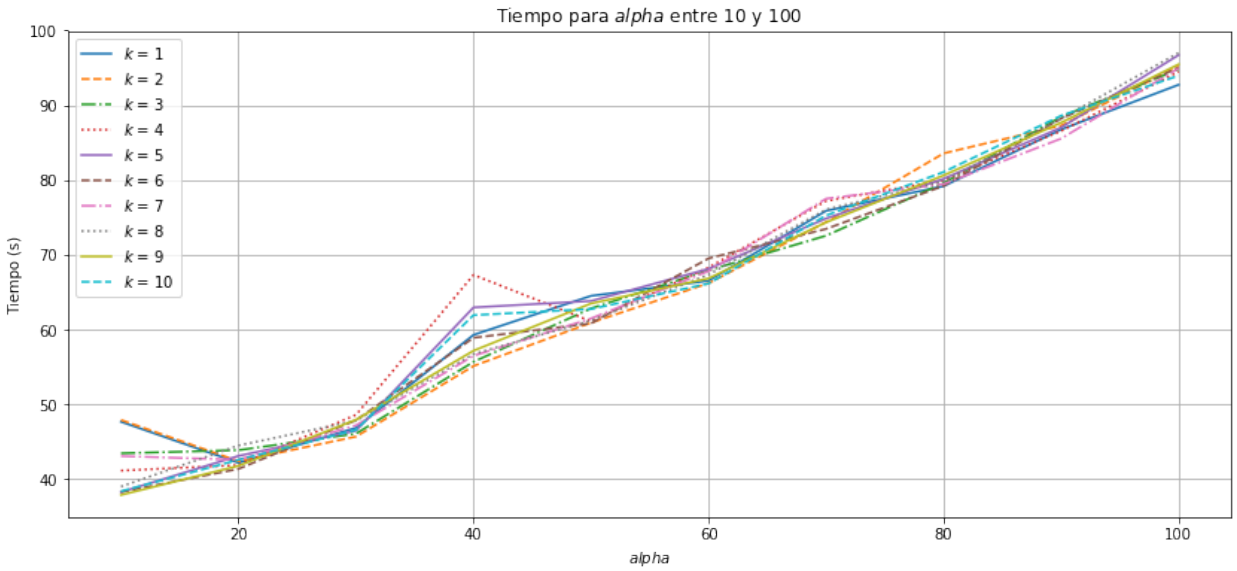


Figura 10: Evaluación de tiempo para distintos valores de k y α. Ejecutado con k de K-Fold = 5, el resultado es el promedio de los 5 tiempos de ejecución de cada corrida

Para responder esto, corremos el programa de la misma manera que antes, utilizando el mismo rango para k y α que el test anterior (k entre 1 y 10, α entre 10 y 100).

A pesar de ciertos *outliers*, se puede apreciar que la variación del k casi no tiene influencia en el tiempo de ejecución, mientras que α sí, de manera aparentemente lineal.

En cuanto al k, pensemos por qué no influye en el tiempo. En nuestro algoritmo, para obtener los k vecinos más cercanos, primero calculamos la distancia euclidiana desde el elemento de test a todos los elementos del training. Luego ordenamos de menor a mayor y nos quedamos con los primeros k elementos, que serán los k más cercanos. Variando este k desde 1 a 10 no va a tener influencia en la complejidad temporal. Incluso no tendría influencia si hiciéramos la implementación más óptima: Recorrer k veces el vector de las distancias y quedarnos con los k más grandes.

Por otro lado, reflexionemos sobre la influencia lineal de α sobre el tiempo de cómputo. El α determina cuántas componentes principales se utilizan. En consecuencia, el α incide en dos partes del programa:

- Al hacer PCA, se calculan α autovectores de la matriz de covarianza. Cada autovector se consigue en aproximadamente la misma cantidad de tiempo (utilizando método de la potencia), con lo cual se tiene una relación lineal entre α y el tiempo de ejecución de este ítem.
- Al hacer k-NN, se calculan las distancias euclidianas de cada elemento, los cuales tienen α componentes. El cálculo de la distancia euclidiana es calcular la norma 2, lo cual es lineal en la dimensión del vector. Por lo tanto, se tiene nuevamente una relación lineal entre α y el tiempo de ejecución de este otro ítem.

Entonces concluimos que sí, es sensato que el tiempo de cómputo del algoritmo dependa linealmente de α, mientras que no cambie según el valor de k.

3.3.3. Matriz de confusión

Ya analizamos la calidad del modelo en términos de accuracy y F1-Score, ahora nos interesa hilar más fino y ver precisamente en dónde el modelo funciona mejor. Con éste propósito presentamos 3 matrices de confusión.

En la figura 11 podemos observar en las diagonales los aciertos del modelo, es decir, cuando se ha predicho un cierto dígito para una imagen su etiqueta coincidió con la predicción realizada. Los elementos fuera de la diagonal nos marcan los errores, i.e., el modelo predijo un cierto dígito X pero en realidad era Y.

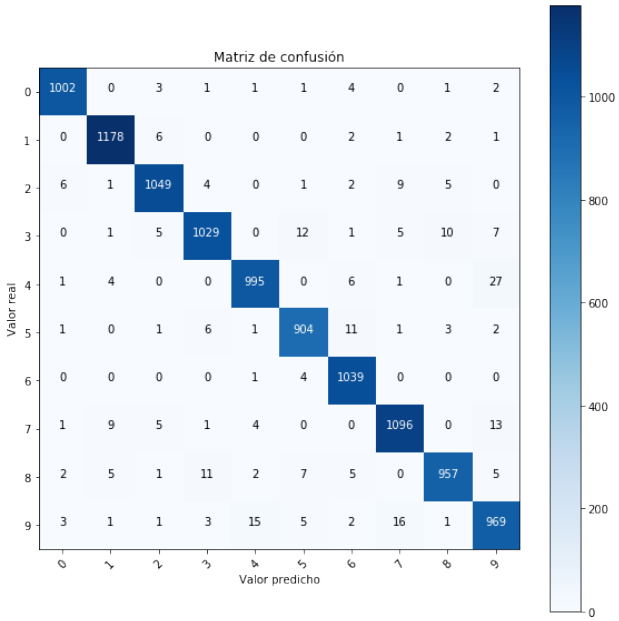
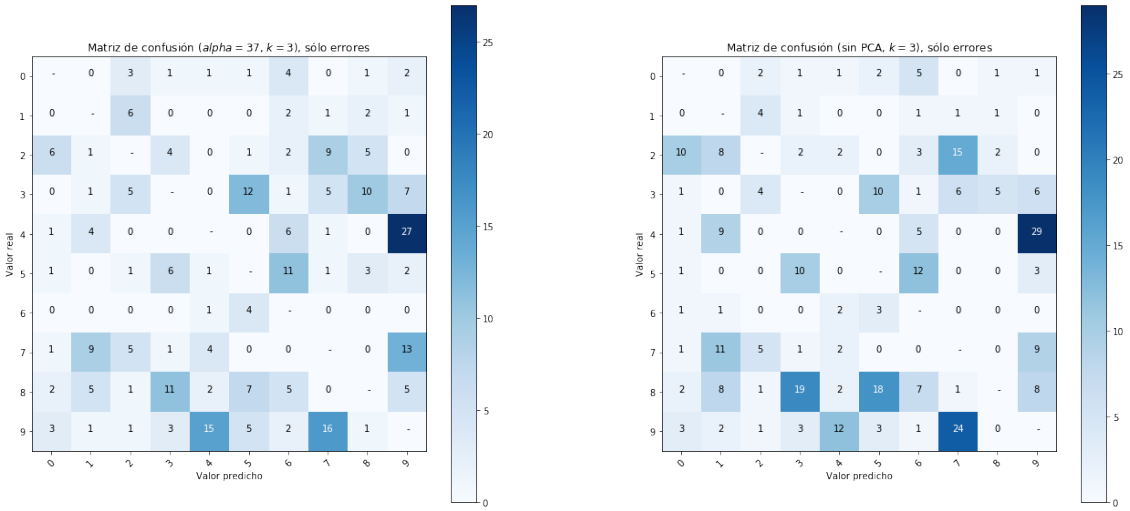


Figura 11: Matriz de confusión para k-NN + PCA con parámetros $\alpha = 37$ y $k = 3$



(a) k-NN + PCA con parámetros $\alpha = 37$ y $k = 3$

(b) k-NN sin PCA con parámetro $k = 3$

Figura 12: Matrices de confusión sin las diagonales (los aciertos)

Ésta primer imagen no nos aporta demasiado, pues ya sabíamos que el modelo era bastante preciso, entonces la diagonal le roba el protagonismo a los errores. Por este motivo presentamos dos matrices en donde la diagonal ha sido anulada, con el propósito de observar los errores con más atención.

En la figura 12 se pueden apreciar dos matrices de confusión, la primera corresponde a los errores de nuestro modelo, utilizando los mejores parámetros que encontramos, la segunda corresponde a utilizar sólo $k - NN$ (sin PCA) con el propósito de observar cualitativamente en qué se diferencian.

En primer lugar, observamos que el error más grande se encuentra al predecir un 9 cuando en realidad correspondía a un 4 (y viceversa). Esta mala predicción sucede en ambos modelos, de forma casi exacta, intuitivamente podemos explicarlo por la similitud entre ambos dígitos.

En general notamos que en la mayoría de los casos, introducir PCA reduce en un bajo porcentaje algunos de los errores. Los cambios en los errores más importantes al introducir PCA son:

- (9,4): de 41 a 42
- (7,9): de 33 a 29
- (3,8): de 24 a 21
- (3,5): de 20 a 18
- (5,8): de 18 a 10

Para cada caso se suman los errores de los dígitos (x, y) e (y, x), el primer número corresponde a los errores sin usar PCA, y el segundo utilizando PCA.

Ésto muestra lo que observamos en el primer experimento, que usar PCA mejora los resultados en comparación a sólo usar $k - NN$. Aunque a priori no entendemos por qué mejoran los resultados, o mejor dicho, por qué hay menos dígitos confundidos, quizás tenga que ver con el poder de abstracción que provee utilizar una técnica que reduzca la dimensionalidad, relacionándolo a la maldición de la dimensionalidad.

3.3.4. Cantidad de componentes principales

Dado que los autovalores de la matriz de covarianza corresponden a la varianza de cada uno de los píxeles de las imágenes, se decidió hacer la siguiente prueba: se calculó la suma acumulada de los autovalores de mayor a menor valor en módulo, y se buscaron algunos cuantiles. En la figura 13 podemos observar que tomando 86 autovalores tendríamos el 90 % de la varianza acumulada. Esto nos da una idea de qué α tomar al momento de correr el clasificador.

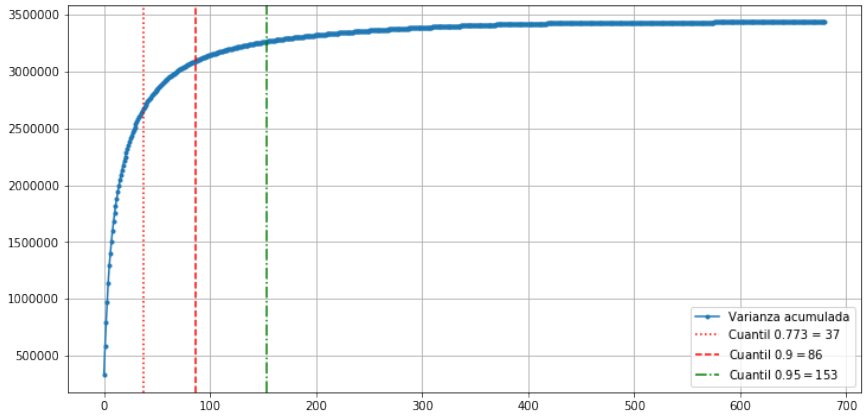


Figura 13: Varianza acumulada (autovalores acumulados)

3.4. Segundo experimento obligatorio: Variando la cantidad de imágenes

Como segundo experimento, se nos plantea la idea de introducir una nueva variable: La cantidad de imágenes del training set. Esta nueva variable muy probablemente influya en tiempos de ejecución, pero la pregunta más importante es cómo influye en la calidad del clasificador.

El sentido común nos indica que, cuanto más training set tenga el programa, mejor podrá predecir cada etiqueta. Incluso, aunque se agregue cierto ruido (porque más imágenes implican mayor cantidad de outliers, dígitos que se parecen demasiado a otros), es claro que un mayor entrenamiento implicará mejores predicciones.

Probamos, entonces, correr nuestro programa con los parámetros óptimos obtenidos en el anterior experimento: $\alpha = 40$, $k = 3$. Si bien en el anterior experimento concluimos que era aún mejor $\alpha = 37$, $k = 3$,

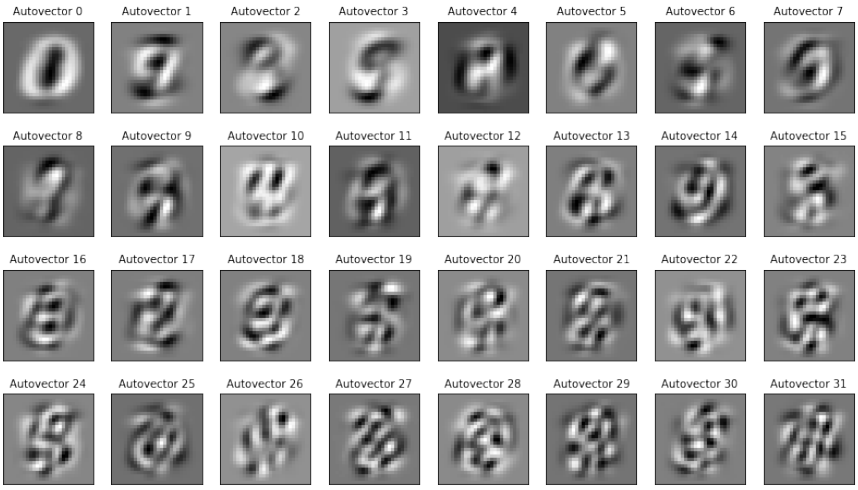


Figura 14: Componentes principales interpretados como imágenes (autovectores)

este segundo experimento fue realizado antes del refinamiento de los parámetros, y como la diferencia entre ambos resultados es despreciable, no amerita que ejecutemos el experimento una segunda vez: No modificará significativamente las conclusiones que saquemos.

Por otro lado, el rango que le demos a la cantidad de imágenes será clave para poder encontrar una relación con las métricas que elegimos. Por ello, definimos el rango del cardinal del training set con los valores:

- Del 100 al 1000, variando de a 100 valores.
- Del 1000 al 10000, variando de a 1000 valores.
- Del 10000 al 30000, variando de a 2000 valores.
- De 30000 al total (42000), variando de a 3000 valores.

Ya que nos parece que la diferencia en los primeros 1000 valores va a ser más significativa que a partir de los 1000.

Los resultados obtenidos se pueden apreciar en la figura 15.

Ambos gráficos muestran una función parecida entre el cardinal del entrenamiento y cada métrica. Notamos que, efectivamente, los saltos entre 100 y 1000 imágenes son muchísimo más abruptos, y alrededor de las 5.000 imágenes comienzan a reducirse los aumentos. Ya en 20.000 imágenes casi no hay diferencias en cada salto, para ambos gráficos.

Concluimos entonces que, al principio, cada incremento en la cantidad de datos ofrece mucha información, por lo que crece rápidamente la calidad del etiquetado. Sin embargo, luego de cierto punto, cada nuevo dato empieza a ofrecer menos información, entonces decrece el crecimiento. En consecuencia, podemos afirmar que nuestro score converge a un valor muy bueno a partir de 5.000 elementos del training set (alrededor de 0.95 para ambas métricas).

Observación: Ya con 10.000 imágenes no hay cambios tan significativos en los scores con respecto al total del entrenamiento, lo que confirma nuestras sospechas al hablar de **reducción del training set (sección 3.2.2)**. O sea, si bien aumenta, estamos hablando del mismo orden de magnitud cuando evaluamos con un training set de entre 10.000 y 42.000 valores.

Evaluando tiempos

Ahora quisiéramos tener los costos temporales de nuestro algoritmo, para establecer nuestro $tradeoff$ de calidad/tiempo de cómputo en cuanto a cuánto entrenamiento utilizar.

Corremos entonces, el programa con los mismos valores de α, k, y el mismo rango de cantidad de imágenes para entrenamiento. Resulta dando:

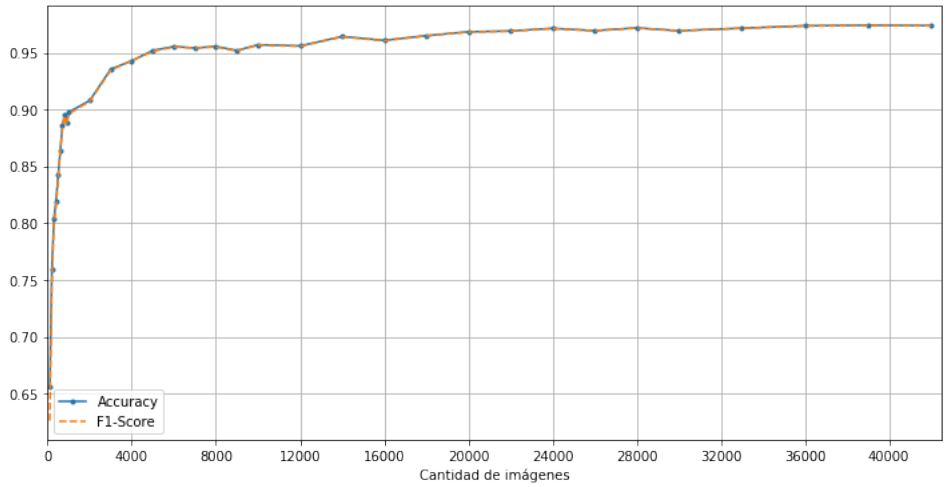


Figura 15: Accuracy y F1-score con $k = 3$ y $\alpha = 40$ variando la cantidad de imágenes

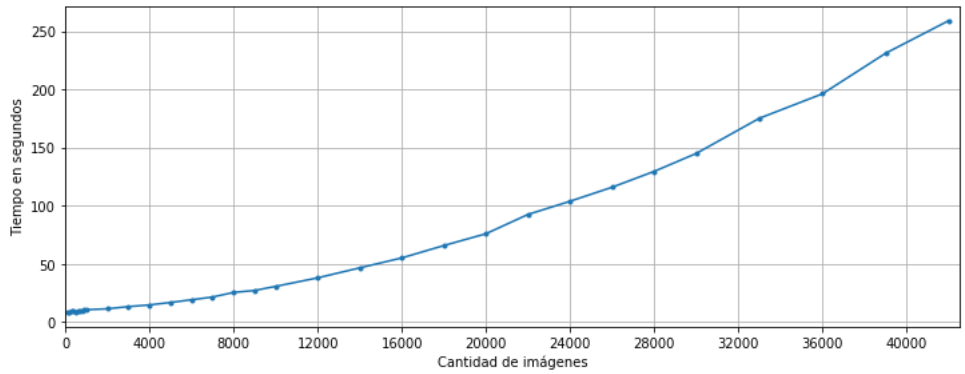


Figura 16: Costo temporal del algoritmo con diferentes tamaños de entrenamiento. Parámetros $\alpha = 40$ y $k = 3$

Podemos apreciar que el costo temporal es mayor que lineal, lo cual tiene sentido ya que realizamos un ordenamiento lineal-logarítmico de las imágenes luego del PCA, sumado a las operaciones matriciales.

Planteamos entonces la pregunta: Analizando el *tradeoff* de tiempos vs. calidad, ¿Con qué cardinal del training set nos quedamos?

Como planteamos antes, con 5.000 imágenes ya tenemos un puntaje de 0.95 en ambas métricas, lo cual sube de a pasos muy pequeños hasta llegar a 95.98 en el cardinal máximo. Por el otro lado, previo a 5.000, las diferencias siguen siendo muy grandes.

En conclusión, basándonos en este gráfico, si tuviéramos que elegir un cardinal para ejecutar nuestro algoritmo y el tiempo de cómputo fuera prioritario, una decisión sensata podría ser la de 5.000 imágenes.

Variando los parámetros

Nos resulta de interés ver que el comportamiento de la calidad (medida en accuracy) con respecto a la cantidad de imágenes sea compartido para otras combinaciones de parámetros. Para analizar esto, tomamos las siguientes combinaciones de k, α y también K (de KFold):

- $k = 3$ $\alpha = 40$ $K = 5$
- $k = 3$ $\alpha = 36$ $K = 5$
- $k = 10$ $\alpha = 10$ $K = 5$

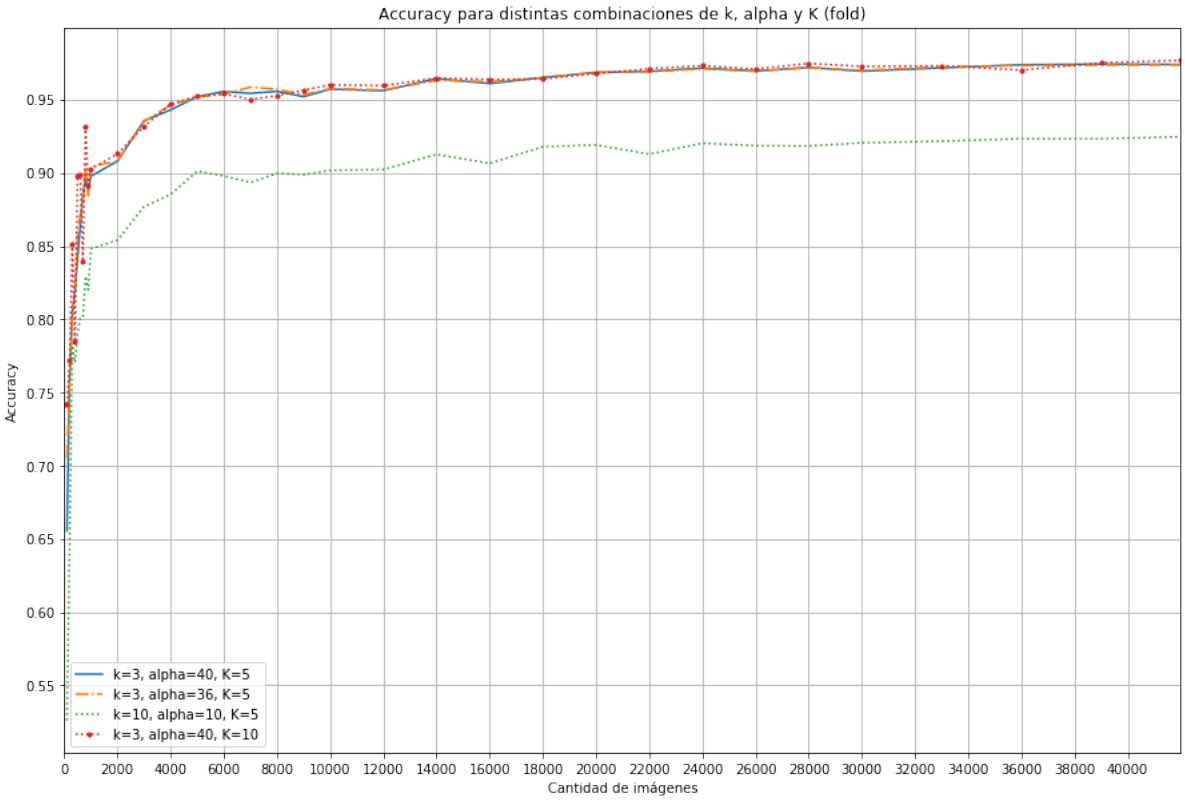


Figura 17: Accuracy para distintas combinaciones, variando la cantidad de imágenes

- $k = 3$ $\alpha = 40$ $K = 10$

La idea fue tomar dos buenas combinaciones de parámetros (las dos primeras) y una mala (la tercera), según los resultados obtenidos en el primer experimento. Adicionalmente, agregamos una combinación extra (la última), en la cual K es 10, con el fin de analizar impacto de variar K en la accuracy.

Observando los resultados (ver figura 17), primeramente se desprende que el análisis realizado previamente se condice al tomar distintas combinaciones de parámetros. Es decir, tener más imágenes para entrenar al algoritmo provoca mejores resultados, sin importar que la combinación de parámetros no sea la mejor. La elección de parámetros $k = 10$ y $\alpha = 10$ está bastante por debajo de la mejor, pero sin embargo su curva de accuracy tiene el mismo movimiento que las demás.

Con respecto a cómo influye el K en la calidad del algoritmo podemos ver que al aumentar su valor de 5 a 10 (primera y última combinación) los resultados no varían sustancialmente, si bien por momentos los resultados son ligeramente mejores. Ésto último puede deberse a que tomar un K más grande implica que la cantidad de imágenes para entrenarse es mayor, con lo cual el algoritmo debería funcionar mejor (esto es lo que estamos analizando en el experimento).

3.5. Tercer experimento obligatorio: Relación entre k y cantidad de imágenes

En los anteriores experimentos ya encontramos los valores óptimos de k y α para una cantidad fija de imágenes, pero ¿Qué ocurre cuando variamos el cardinal del conjunto de training?

Comenzamos por analizar los casos extremos

3.5.1. Menor k posible

¿Cuál es el menor k posible que podríamos tomar? ¿Tiene sentido tomar $k = 1$?

Si tomáramos $k = 1$, la imagen de test será etiquetada según la imagen de training más similar. Esto podría funcionar si nuestro cardinal de imágenes es extremadamente pequeño, pero en el caso general hace el algoritmo **muy susceptible a outliers**. Por ejemplo, si en nuestro training set tuviéramos un 5 que se parece demasiado a un 8, nos puede llegar a arruinar la clasificación de todas las imágenes correspondientes a 8, malinterpretando que son 5.

3.5.2. Mayor k posible

¿Cuál es el mayor k posible que podríamos tomar?

Supongamos que tenemos un training set balanceado de 10.000 imágenes. Por ser balanceado, podemos asumir que tendremos 1.000 imágenes de cada etiqueta. ¿Tendría sentido tomar k mayor a 1.000?

Si se tomara k = 1.200, al buscar la moda entre los 1.200 vecinos más cercanos, sabríamos con certeza que **al menos 200 vecinos van a aportar ruido** a la clasificación. Pues en el mejor de los casos estaríamos tomando los 1000 elementos de la etiqueta correcta y el resto corresponderían a otro dígito.

3.5.3. Variación de k y cardinal del training set

Evaluamos empíricamente la relación entre el valor de k óptimo y el cardinal del training set, chequeando si los resultados realmente respetan las cotas que propusimos. Realizamos entonces los gráficos (ver figuras 18, 19), donde:

- El eje horizontal representa el cardinal del conjunto de training.
- El eje vertical representa el score alcanzado.
- Cada curva representa el score obtenido para cierto tamaño de training set y un k fijo.
- Se fija $\alpha = 37$ por ser aquel de mejor resultado en los experimentos anteriores

Los k utilizados fueron:

- $k = 1$: Es el menor k posible
- $k = 3$: Basandonos en los experimentos anteriores este es el mejor valor que podríamos tomar para k
- $k = 100$: Representa un valor de k fijo, pero mucho más grande que el óptimo encontrado anteriormente
- $k = 12\,\%$ del training set: Representa un valor que necesariamente, por lo justificado anteriormente, debe introducir caracteres de otra clase al querer encontrar los k-vecinos más cercanos de un dígito de test

En estas imágenes (figuras 18 y 19) se puede apreciar lo que habíamos analizado teóricamente antes. Como era de esperar por lo detallado en el experimento 1 el mejor resultado se obtiene utilizando el $k = 3$, la diferencia no parece mucha con $k = 1$, y esto condice con lo obtenido en la figura 8. Por otro lado también se ve reflejada la tendencia establecida en el experimento 1 (ver figura 1) en cuanto a que mayores k (en este caso k=100) arrojan peores resultados.

Más aún se puede ver la prueba empírica de lo dicho anteriormente vinculado al desempeño general esperado si se utiliza un k que sabemos es mayor a la cantidad de elementos de cada una de las clases.

3.6. Variando el K de K-Fold

Habiendo utilizado K-Fold en los anteriores experimentos, buscamos responder ciertas preguntas que nos surgen.

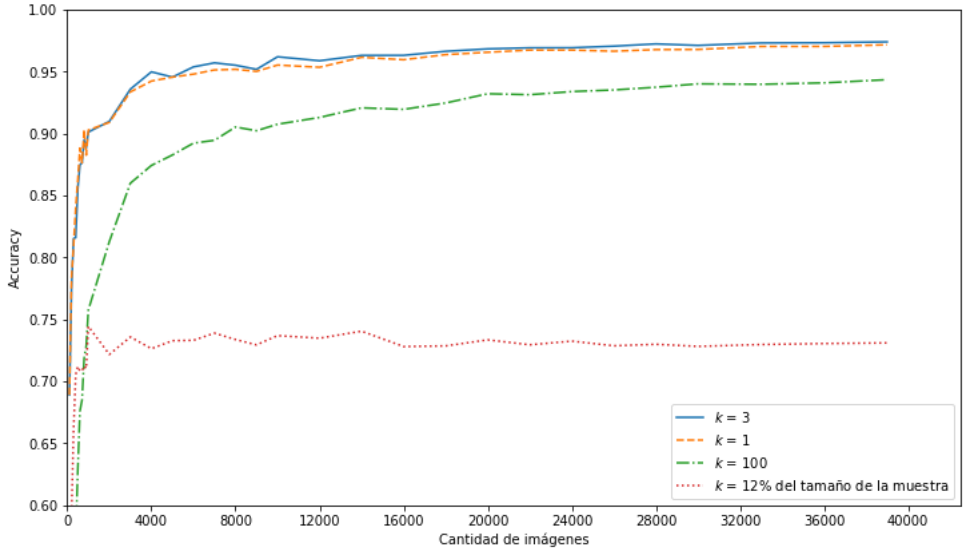


Figura 18: Comparativa de scores variando el k y el tamaño del training set utilizado

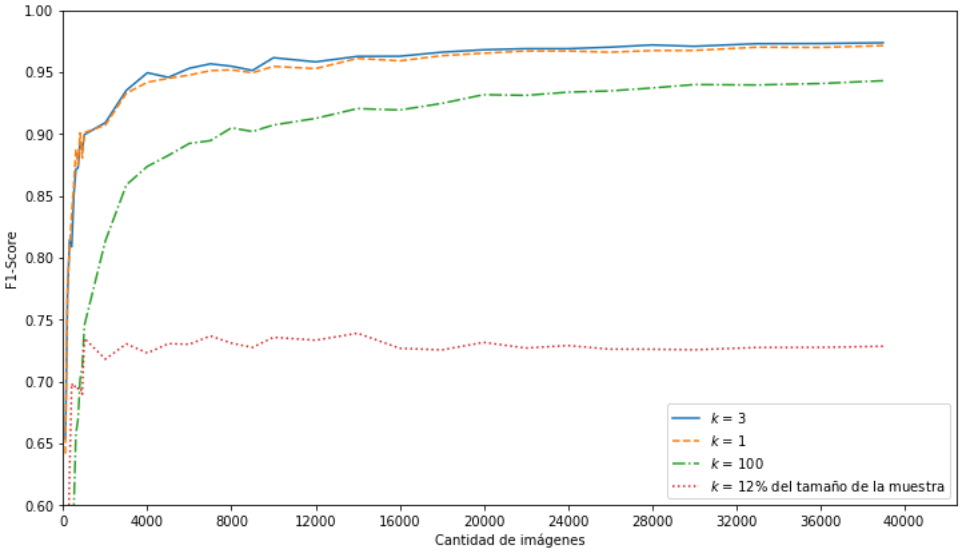


Figura 19: Comparativa de scores variando el k y el tamaño del training set utilizado

3.6.1. ¿Cuál es el valor máximo de K que podemos tomar?

¿Cuál sería el caso extremo que se puede tomar para el K de K-Fold?

Sin balanceado Si n es el total de imágenes, el mayor valor posible es K $= n$. ¿Tiene sentido tomar n particiones, utilizar $n-1$ para training y la restante para testing?

Sí. De hecho, a esto se lo denomina *leave-one-out*[3]. Según averiguamos, suele ser una métrica muy efectiva, aunque claramente muy costosa. Hay que correr el algoritmo desde cero n veces, lo cual es extremadamente caro en este caso (ya con K $= 500$ los tiempos de cómputo eran extremos).

Balanceado Considerando una división estratificada (como tomamos en toda la experimentación), ya no podemos realizar n cortes tan tranquilamente. Observemos que surgen problemas tomando K $> n/c$, con n la cantidad de imágenes y c la cantidad de clases. Esto ocurre porque no se tienen suficiente cantidad de elementos de cada clase como para dejar todas las particiones balanceadas.

3.6.2. ¿En qué situaciones es más conveniente utilizar K-fold con respecto a no utilizarlo?

Como ya fue explicado previamente, K-Fold puede llegar a tardar mucho tiempo. El algoritmo de entrenamiento tiene que correr desde cero K veces. Lo cual significa que tarda K veces más calcular los resultados de K-fold que si usáramos el método de *holdout*[3] (el cual propone simplemente realizar una sola división entre training y testing), entrenar una vez y correr una vez.

3.6.3. ¿Cómo afecta el tamaño del conjunto de entrenamiento?

Como ya fue explicado, al hacer K-Fold utilizamos un dataset de entrenamiento de $\frac{(K-1)n}{K}$, ya que ese $\frac{n}{K}$ restante es utilizado para el testing. Esto implica que, a mayor K, mejores van a dar las métricas que utilicemos, ya que el conjunto de entrenamiento es mayor y, como fue explicado en el segundo experimento obligatorio, esto implica mejores resultados.

Repetimos el experimento 1 para sus parámetros óptimos (α=37 y k=4), variando el K de K-Fold de a saltos de a 100.

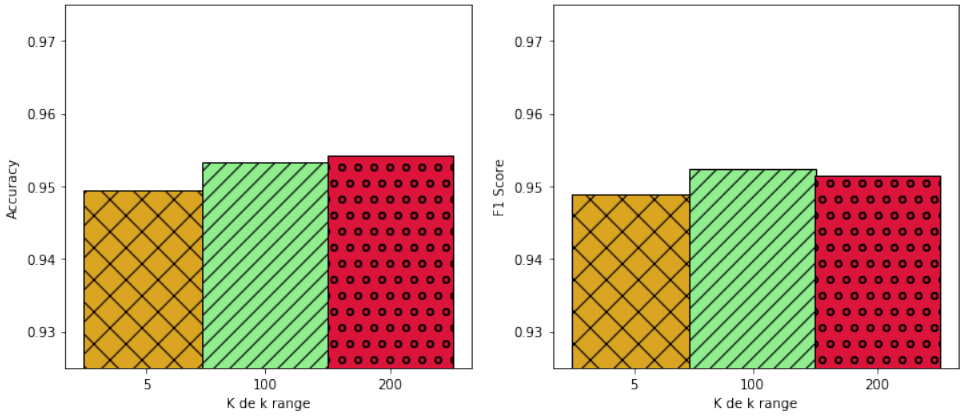


Figura 20: Puntajes de referencia variando el K de K-fold y usando k=3, α=37

Observamos que no hay claras mejorías en las métricas. Es más, el F1-Score aparenta dar peor con K = 200 que con K = 100.

Al mismo tiempo, con K = 200 tardó en correr aproximadamente una hora y media.

Teniendo en cuenta que los tiempos eran demasiado altos y las métricas no variaban mucho, nos quitó el incentivo a seguir probando con valores más grandes, y nos dejó la conclusión que con K tan grandes no vale la pena gastar tanto tiempo de cómputo, ya que no cambia mucho en cuanto a las métricas.

3.7. Dígitos manuscritos

Se hizo el experimento de escribir a mano sobre un papel una serie de dígitos, con el fin de poder extraerlos y reconocerlos con nuestro clasificador. Se utilizaron varias herramientas como Numpy, Scipy, Scikit-Image y OpenCV para preprocesamiento y búsqueda de contornos.

En la figura 21 se tiene la captura de la hoja de papel con los dígitos escritos con marcadores de diferente grosor, y distinta tipografía.

3.7.1. Preprocesamiento

Para poder alimentar la imagen al clasificador se debe primero preprocesar la imagen. Lo primero que se hizo fue pasar la imagen a escala de grises, tomar un umbral y binarizarla.

Dado las condiciones de luz de la imagen capturada, hay dígitos que no se pueden ver bien. Para mejorar esto se ecualizó el histograma. Como último paso, se aplicó un filtro de mediana de 3×3 para mejorar el grosor de esos dígitos borrosos y eliminar otros tipos de ruido. El resultado de este proceso se puede ver en la figura 22.

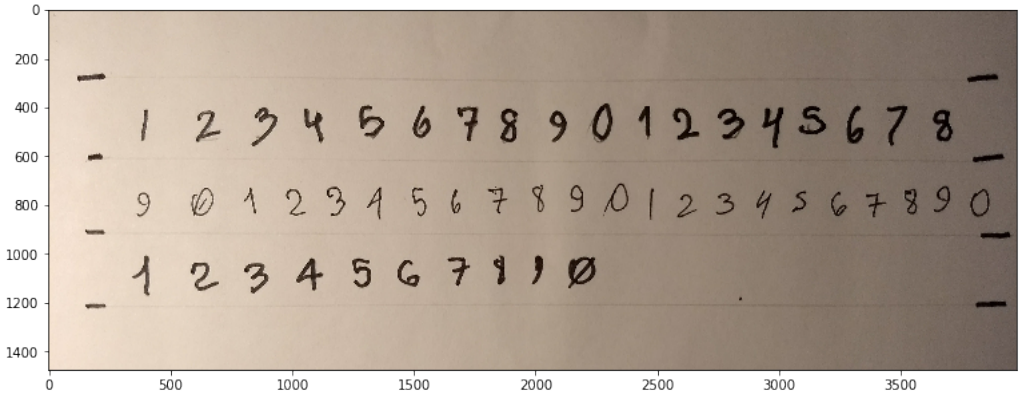


Figura 21: Fotografía del papel con los dígitos

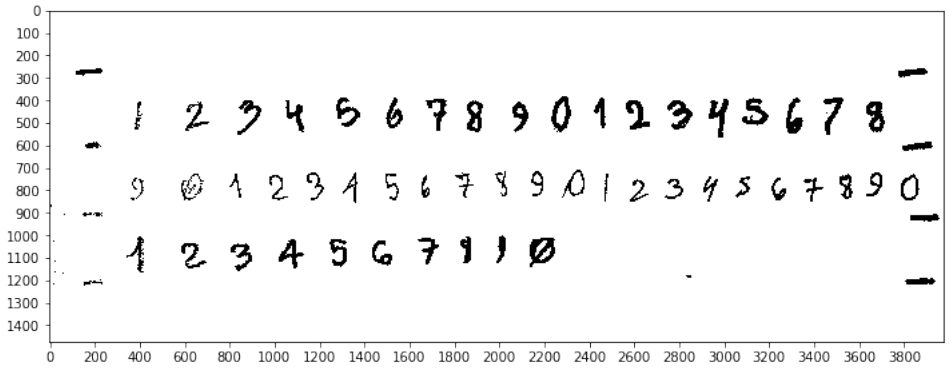


Figura 22: Imagen del papel preprocesada

3.7.2. Contornos

El siguiente paso fue hacer una búsqueda de contornos con OpenCV con la función `findContours`, que implementa el algoritmo propuesto por Suzuki[2] para encontrar contornos en una imagen binaria. Para que funcionara correctamente, se tuvo que invertir la imagen de modo que el fondo tenga valor 0 y los dígitos valores cercanos a 255.

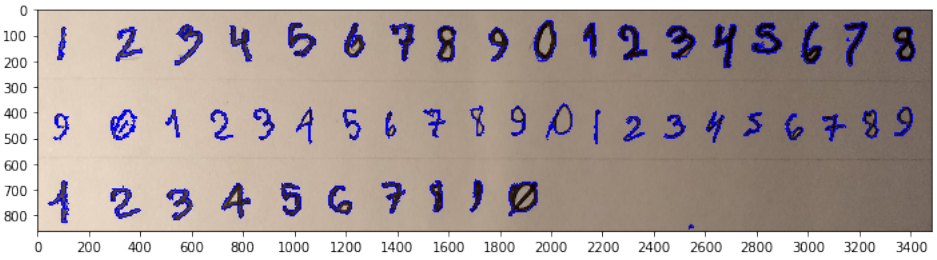


Figura 23: Contornos detectados en la imagen binaria

Teniendo los contornos (ver figura 23), se calcularon los *bounding boxes* de los contornos más grandes, obteniendo en la mayoría de los casos un recuadro de cada dígito.

Se puede observar en la figura 24 que hay algunos dígitos que tienen más de un contorno, como el 0 de la segunda fila. Se podrían haber filtrarlos viendo si la distancia al más cercano es menor a cierto número (relacionado al ancho esperado de los dígitos en la imagen), o buscando los rectángulos que intersecan y reemplazaros por uno que los contenga a todos, pero se decidió al final filtrarlos a mano dado que son pocos

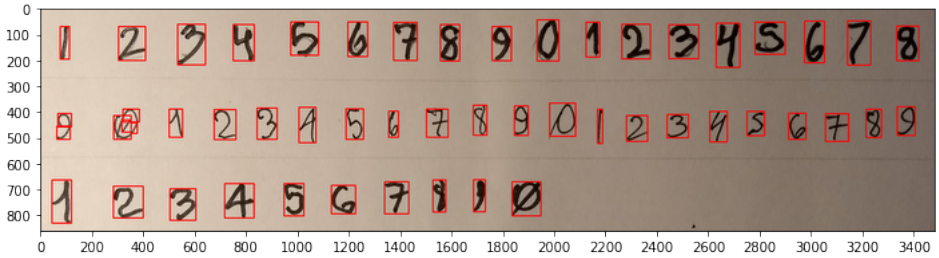


Figura 24: Recuadros de cada dígitos basados en los contornos detectados

dígitos.

3.7.3. Recorte de dígitos, centrado y reescalado

A partir de los recuadros, se tomó un margen de 30 píxeles y se los recortó, como se puede observar en la figura 25.

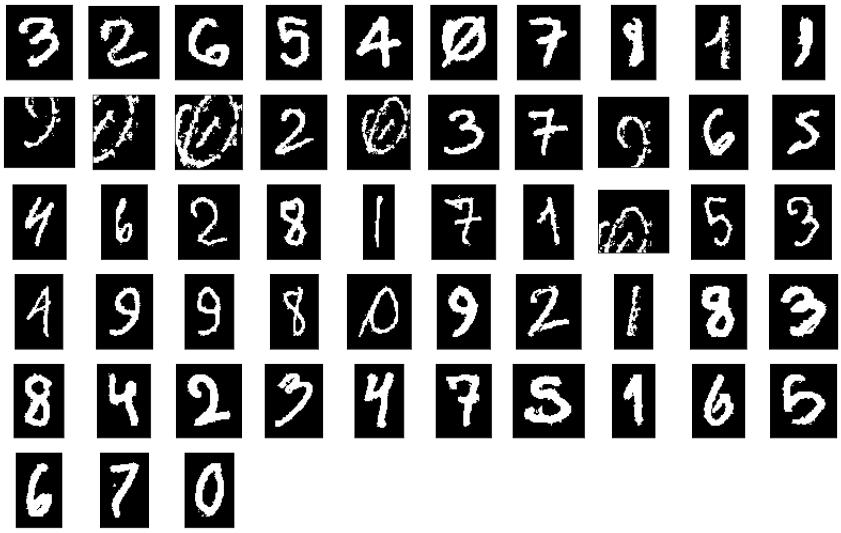


Figura 25: Dígitos recortados y con un margen de 30 píxeles

Finalmente, se ajustó el tamaño de cada imagen a 28×28, manteniendo el *aspect ratio* y reescalando con interpolación Lanczos, para evitar posibles distorsiones. Aprovechamos en este paso también de filtrar las imágenes repetidas o mal capturadas con el método de los contornos. En la figura 26 se puede observar el resultado, ya listo para ser procesado por el clasificador.

3.7.4. Evaluación

Luego de etiquetar las imágenes a mano, se armaron los arrays para alimentar al clasificador y se lo corrió con los parámetros óptimos primero. El accuracy obtenido al correr el clasificador con todo el conjunto de entrenamiento sobre las imágenes procesadas fue de 0,71, más bajo que el obtenido en los experimentos anteriores. Corriéndolo con un porcentaje más alto de componentes principales ($\alpha = 100$) se obtuvo una mejora: 0,73.

También se intentó aplicar un filtro de mediana *antes* de reescalar a 28×28, para reducir artefactos, pero al correr el clasificador se obtuvo 0,22 de accuracy.

Se concluyó de estas pruebas, y ajustando algunos pasos del preprocesamiento, que el clasificador k-NN es muy sensible a traslaciones y pequeñas distorsiones en la imagen, por lo que la etapa de preprocesamiento

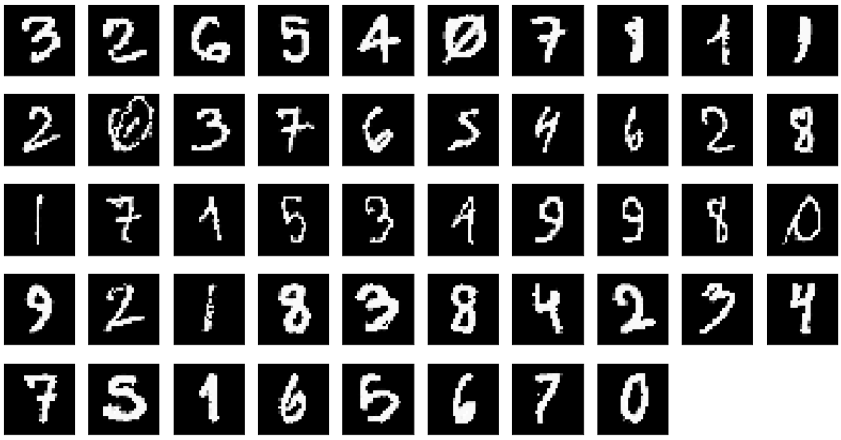


Figura 26: Imágenes de dígitos 28×28, resultado final

de las imágenes resulta ser de suma importancia a la hora de utilizar este clasificador para reconocer dígitos.

3.8. Base de datos MNIST original

Se hizo una breve evaluación del clasificador usando el conjunto de datos MNIST original[4], que contiene 60.000 imágenes de dígitos etiquetas, un 42 % más que el dataset provisto por la cátedra. Estas imágenes son un subconjunto del dataset estudiado y fueron preprocesadas de la misma manera.

Los resultados obtenidos fueron levemente mejores

Dataset	**Accuracy**	**F1-score**
MNIST Kaggle	0.97376	0.97367
MNIST Original	0.97590	0.97570

Cuadro 1: Comparación de puntaje entre datasets

El hecho de que los resultados no hayan cambiado mucho se condice con el resultado de las pruebas realizadas sobre el tamaño del dataset de entrenamiento (segundo experimento): aumentar el tamaño del dataset siempre parece produce mejoras, pero éstas son cada vez más pequeñas en comparación.

3.9. Participación de la competencia de Kaggle

Probamos nuestro clasificador en la competencia *Digit Recognizer* de Kaggle con los valores óptimos que nos arrojaron las pruebas realizadas: $k = 3$ y $\alpha = 37$. El puntaje de accuracy fue 0,97585, ubicándonos en la posición 837[4]. El nombre del equipo con el que se envió el resultado es `metnum172c-hprs`.

4. Conclusión

En este trabajo hemos desarrollado un modelo que permite reconocer dígitos manuscritos en imágenes, con cierto nivel de efectividad. Hemos desarrollado varios experimentos para poder analizar qué tan buenas son las técnicas utilizadas, y a su vez, cómo elegir los parámetros del modelo.

Al realizar el primer experimento concluimos que para utilizar $k - NN$ es conveniente tomar pocos vecinos, i.e. un k pequeño, más aún encontramos que $k = 3$ fue la mejor opción (con y sin PCA). Mientras más grande el k, peores métricas obteníamos.

[4] Al día 18 de octubre de 2017

Por otro lado, concluimos también que utilizar PCA mejoró los resultados, contra nuestra intuición que nos hacía pensar que al reducir la dimensión se iba a perder información. Quizás PCA logra generalizar lo suficiente utilizando la información importante, mientras que pierde la información no importante, y por eso funciona bien. Esto tiene que ver, como lo mencionamos previamente, la maldición de la dimensionalidad. O sea, no siempre perder información es malo, si es que uno se pierde de elementos que aportan más ruido que datos útiles y representativos.

Al explorar el rango donde el parámetro α se mueve, descubrimos que hay un intervalo pequeño, entre 30 y 50, en donde se obtienen muy buenos resultados, más aún encontramos el α óptimo en 37. Tomar valores grandes α empeora las métricas y también conlleva tiempos de ejecución más largos. Nos asombramos con la efectividad de PCA, ya que con 37 componentes bien elegidas, se pudieron representar las imágenes de una forma más barata y, además, más representativa.

En cuanto al training set, nos dimos cuenta desde un principio que realizar la experimentación sobre toda la base iba a ser problemático por cuestiones de tiempo de cómputo. Pudimos comprobar que efectivamente tomar un porcentaje del 20 % (o incluso menos) de la base es representativo del total. Creemos que, al trabajar con muchos datos, es una conclusión importante a tener en cuenta.

A la vez, al variar los tamaños de la base que tomamos, nos tomó por sorpresa que el k óptimo de KNN no fuera variando según el tamaño del training set. La intuición nos sugería que, a mayor training set, mayor debía ser el k. Puede ser que con un training set extremadamente más grande se deba aumentar el k, pero las tendencias de nuestros resultados nos sugirieron lo contrario.

Sobre la elección de las métricas, quizás no haya sido una buena idea tomar F1-Score como métrica alternativa a accuracy, pues siempre dan muy similares. Esto se debe a que la suma de falsos positivos y negativos es mucho más chica que los verdaderos positivos. Quizás hubiera sido más interesante utilizar una métrica que nos fuerce a analizar un *tradeoff* entre ella y el Accuracy. De todas maneras, F1-Score nos dio un resultado que mediaba entre Accuracy y Recall. En cierta forma, es como si él mismo nos solucionara el *tradeoff* que surge entre ambas.

Al agregarle dígitos manuscritos nuestros, nos dimos cuenta que el preprocesamiento es muy clave a la hora de ejecutar nuestro algoritmo. A pesar de haber obtenido buen resultado en nuestra experimentación y en la competencia de *Kaggle*, el programa demostró ser bastante ingenuo, y sensible a pequeñas traslaciones y distorsiones de las imágenes.

Éste trabajo nos pareció muy interesante, hay muchas aristas por donde desarrollar, y si bien agregamos algunos items no obligatorios, nos quedamos con las ganas de experimentar más. Algunas de las cosas que nos hubiera gustado ver son las siguientes:

- Elegir los α en relación a cuán grandes son los autovalores, cuál es el porcentaje que cubre sobre la suma de los valores absolutos del resto. Quizás pocos autovectores representan la transformación lo suficientemente bien.
- Procesamiento de imágenes: Ver como afecta nuestras métricas aplicar distintos tipos de procesos, como ruido o filtros, y transformaciones geométricas como traslaciones, rotaciones y *shear/skew*. También probar hacer *data augmentation* (incluir versiones modificadas de las imagenes en el dataset de entrenamiento) y si tiene algún efecto sobre cómo funciona el algoritmo k-NN.
- Base propia: armar un dataset propio más grande y etuiquetado para ver qué tan bien funciona acá. Replicar el estudio del NIST (en una escala más pequeña) con personas en el país y ver como funciona el modelo allí.
- Confianza: nos queda pendiente utilizar una medida de confianza sobre la predicción realizada, ¿qué tan lejos está el elemento a predecir de los vecinos mas cercanos? Esto nos podría decir cuánta confianza tenemos en la predicción.
- Sería interesante extender el programa para que reconozca caracteres de nuestro alfabeto, o incluso de otros alfabetos, como el chino. No necesariamente que diga cuál letra es, sino que pueda distinguir si la imagen corresponde a una letra, un número, o un ideograma chino.
- Probar alternativas a K-Fold y ver qué dan, como por ejemplo *holdout* o *leave-one-out*[3].

Referencias

[1] Richard L. Burden. *Numerical analysis*. 1978.

[2] S. Suzuki y K. Abe. *Topological Structural Analysis of Digitized Binary Images by Border Following*. 1985, págs. 32-46.

[3] Jerome Friedman Trevor Hastie Robert Tibshirani. *The Elements of Statistical Learning, Second Edition*. 2008, págs. 241-245.

[4] Christopher J.C. Burges Yann LeCun Corinna Cortes. *The MNIST Database of Handwritten Digits*. URL: `http://yann.lecun.com/exdb/mnist/`.