DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP2: Reconocimiento de dígitos

Métodos Numéricos

Integrante	LU	Correo electrónico
Camjalli, Roni Ezequiel	12/18	rcamjalli@gmail.com
Rodriguez, Miguel	57/19	mmiguerodriguez@gmail.com
Itzcovitz, Ryan	169/19	ryanitzcovitz@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

https://exactas.uba.ar

Índice

1. Introducción

1.1. Resumen

A lo largo del informe vamos a analizar distintas técnicas para resolver el problema de clasificación aplicado en reconocimiento de dígitos manuscritos en imágenes. Vamos a utilizar la técnica de kNN de clasificación por vecinos mas cercanos y también vamos a intentar mejorarla combinándola con análisis por componentes principales (PCA). También vamos a ver cross-validation para obtener resultados sin sesgos que podrían surgir por como esta formado nuestro conjunto de pruebas. En los experimentos tomaremos métricas como por ejemplo la de accuracy, para poder medir la efectividad de nuestro clasificador. Para cerrar realizaremos unas pruebas del mundo real y compararemos nuestro clasificador con otros a través del sitio Kaggle [1].

1.2. Introducción teórica

En este trabajo vamos a resolver el problema de reconocimiento de dígitos manuscritos en imágenes. En la actualidad el reconocimiento de caracteres es un método utilizado para traducir imágenes digitales a texto de forma rápida y hasta en algunos casos en tiempo real. Muchas de las implementaciones para resolver este tipo de problemas se basan en redes neuronales que son entrenadas con el fin de reconocer patrones y poder deducir que es lo que dice un texto o que número es el que estamos viendo. En nuestro caso, vamos a *entrenar* y programar nuestro sistema de reconocimiento con un método mas simplificado, basado en clasificación por k-vecinos más cercanos (kNN) junto con analisis por componentes principales (PCA).

El objetivo de este trabajo se limita a reconocer dígitos manuscritos entre el 0 y el 9. Estos dígitos serán representados por una lista de 784 números que a su vez representan una imagen de 28x28 píxeles en escala de grises, donde en valores numéricos el 0 representa un píxel apagado y el 255 representa un píxel completamente encendido.

Para poder predecir el valor de un dígito, el sistema requerirá de una *base de entrenamiento* o *training set* la cual va a utilizar para realizar las predicciones sobre el *conjunto de prueba* o *test set*. El formato que usaremos como archivos de entrada y salida son los mismos usados en Kaggle para la competencia *Digit Recognizer* [1].

En nuestro caso, usaremos nuestra base de entrenamiento original y la dividiremos en dos partes, una será usada para entrenamiento y otra que funcionara como prueba para ver como esta funcionando el sistema. Para decidir que parámetros o métodos son mas eficaces usaremos distintas métricas como por ejemplo:

- Accuracy
- Precision
- Recall
- Kappa de Cohen
- F1-Score

Estas métricas, junto con el tiempo de ejecución, permiten observar desde distintos puntos de vista que tan bien esta funcionando el sistema, y mas específicamente, que mezcla de parámetros y métodos son los mas adecuados para tener una mayor eficacia a la hora de reconocer dígitos. Vale la pena recalcar que la tasa de efectividad o *accuracy* es una de las métricas mas importantes ya que nos dice exactamente el porcentaje de efectividad que obtuvimos.

2. Desarrollo

En esta sección se hará un análisis de los métodos implementados para resolver el problema y sus implementaciones. Los dos métodos implementados fueron kNN y PCA, que además, pueden ser utilizados en conjunto.

2.1. kNN (k-Nearest Neighbors)

El método kNN para realizar una predicción sobre una imagen se basa en comparar la imagen (en forma de matriz) con todas las imágenes de entrenamiento y utilizar las k imágenes mas cercanas para decidir que numero es. Para realizar esta comparación obtenemos a cada imagen de entrenamiento como una fila de 784 números, y agrupamos todas las imágenes en una matriz en donde cada fila será una imagen.

Para predecir el valor de una imagen del conjunto de validación vamos a restarla contra cada una de las de entrenamiento. Al obtener esta nueva matriz calculamos la norma de cada fila y nos quedamos con los k valores mas cercanos a 0 (siendo k el valor de vecinos mas cercanos configurado al inicio de la ejecución). Para finalizar con el algoritmo se hace una votación entre las k imágenes con norma mas cercana al 0 y el numero que mayor cantidad tenga será la predicción final. Este comportamiento esta definido por el siguiente algoritmo:

Algorithm 1 Algoritmo del método kNN

$T \in \mathbb{R}^{m \times n}$ matriz con imágenes de entrenamiento
$k \in \mathbb{N}$ cantidad de vecinos a comparar
$X \in \mathbb{R}^n$ vector de la imagen
function $ClasificadorKNN(X)$
    $normas \leftarrow []$
    para i **desde** 1 **hasta** $T.rows()$ **hacer**
        $normas[i].norma \leftarrow \|T[i] - X\|_2$
        $normas[i].label \leftarrow T[i][0]$ ▷ Obtiene el verdadero numero
    end
    $normas.sort()$
    $votos \leftarrow [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
    para i **desde** 0 **hasta** k **hacer**
        $votos[normas[i].label] = votos[normas[i].label] + 1$
    end
    $prediccion \leftarrow 0$
    $cantidad \leftarrow votos[0]$
    para i **desde** 1 **hasta** 9 **hacer**
        if $cantidad < votos[i]$ **then**
            $cantidad = votos[i]$
            $prediccion \leftarrow i$
        end
    end
    return $prediccion$

2.2. PCA (Principal Component Analysis)

Para hacer este método, lo primero que necesitamos hacer es calcular la matiz de covarianza. Para calcularla, primero debemos restar a cada valor la media de su columna, obteniendo así una nueva matriz y luego hacer la cuenta para tener la matriz de covarianza como lo describe el siguiente algoritmo:

Algorithm 2 Cálculo de la matriz de covarianza (Mx)

$X \in \mathbb{R}^{n \times m}$ matriz que tiene las columnas restadas por su media
$n \in \mathbb{R}$ cantidad de imágenes en el entrenamiento

function $MatrizDeCovarianza(X, n)$
    $Mx \leftarrow \frac{1}{n-1} * X^t * X$
    return Mx

Para continuar realizamos una diagonalizacion de la matriz de covarianza Mx para luego calcular sus autovalores y autovectores con el método de la potencia y deflación. Esto se puede realizar tantas veces como columnas tenga la matriz Mx ya que tiene base ortonormal de autovectores por ser simétrica.

El método de la potencia calcula el primer autovector y autovalor de la siguiente forma:

Algorithm 3 Método de la potencia

1: $A \in \mathbb{R}^{n \times n}$ matriz de covarianza Mx
2: $x_0 \in \mathbb{R}^m$ vector inicial que vamos a transformar hasta obtener el autovector
3: **function** $MetodoDeLaPotencia(A, x_0, niter)$
4: $v \leftarrow x_0$
5: **para** i **desde** 1 **hasta** $niter$ **hacer**
6: $v \leftarrow \frac{A * v}{\|A * v\|}$
end
7: $\lambda \leftarrow \frac{v^t * B * v}{v^t * v}$
8: **return** λ, v

Luego de obtener el primer autovector λ y autovalor v queremos seguir el procedimiento para los α autovectores y autovalores, para poder seguir aplicando el método de la potencia primero tenemos que hacer deflación de la siguiente manera:

Algorithm 4 Método de la deflación

1: $A \in \mathbb{R}^{n \times n}$ matriz a la cual le calculamos su primer autovalor y autovector
2: $\lambda \in \mathbb{R}$ primer autovalor de A
3: $v \in \mathbb{R}^n$ autovector asociado al autovalor λ de A
4: **function** $Deflacion(A, v, \lambda)$
5: $A \leftarrow A - \lambda * v * v^t$
6: **return** A

Aquí obtenemos una nueva matriz A que si le aplicamos el método de la potencia conseguiremos el segundo autovector y autovalor de la matriz original. Este procedimiento de hacer método de la potencia sumado con el de deflación lo repetiremos α veces según el parámetro de entrada.

Para finalizar agrupamos los α autovectores obtenidos por columnas en una matriz V que serán las componentes principales de los datos. Aplicamos la reducción de la dimensión con la transformación característica multiplicando $V^t * X$ aplicando el cambio de base a cada muestra de X consiguiendo una matriz con la misma cantidad de filas, pero con α columnas.

Para poder obtener la predicción de una imagen podemos, aplicar el método kNN teniendo como matriz de entrenamiento a $V^t * X$ y la imagen P que queremos predecir le debemos aplicar la transformación característica correspondiente como con la matriz de entrenamiento, por lo que $P = V^t * P$ así ambas tienen la misma dimensión.

2.3. Cross-Validation

Para el entrenamiento de nuestro clasificador utilizaremos la técnica de Cross-Validation para evitar cualquier tipo de sesgo una vez separado nuestro conjunto de pruebas en uno de entrenamiento y uno de validación. Esto va a implicar dividir nuestro conjunto de pruebas

en K folds (subconjuntos) de mismo tamaño y realizar K pruebas rotando, tomando en cada una un distinto fold como el conjunto de validación. De esta forma vamos a calcular las distintas métricas K veces y luego tomar la media entre ellas. En la figura 1 se puede ver para cada iteración del método cross-validation como este va seleccionando un conjunto de validación con distintas imágenes dentro del conjunto de pruebas.

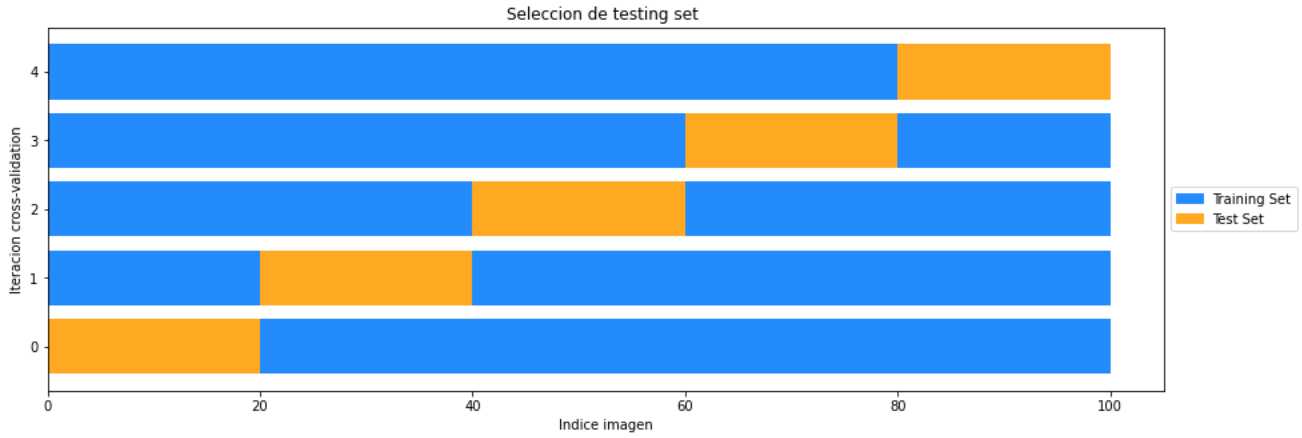


Figura 1: Selección de testing set para cada iteración de Cross-Validation

Un punto importante a tener en cuenta en la separación de folds es la diversidad de clases en cada fold. Si cuando separemos nuestros folds no nos aseguramos de alguna manera que tengamos la mayor cantidad de clases en todos los folds, cuando seleccionemos los folds como conjunto de validación no vamos a estar validando todas las clases. Por ejemplo en la figura 2 como para un dataset ordenado se dividen nuestros folds en conjuntos de la misma clase y luego podemos observar como en la figura 3 se divide el mismo dataset en folds luego de hacerle un shuffle antes de dividirlo.

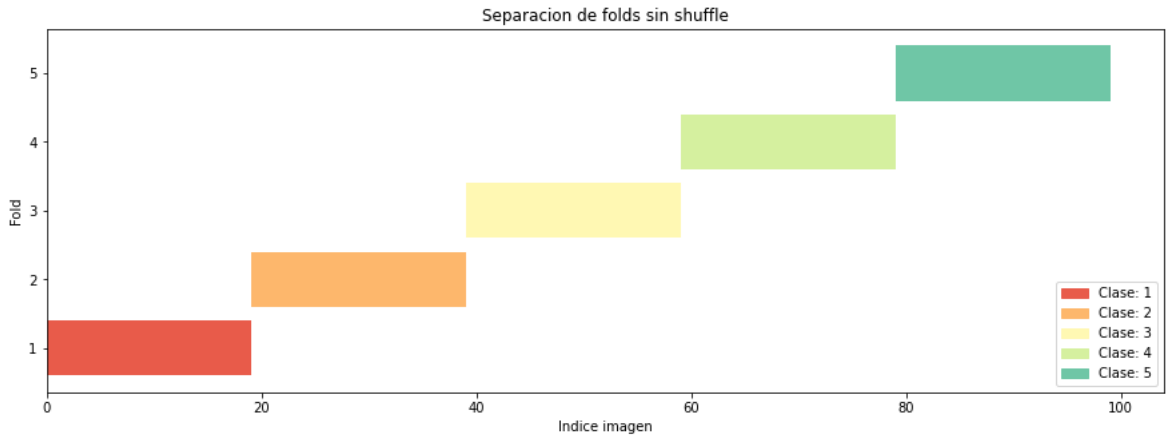


Figura 2: Separación en folds para conjunto de entrada ordenado.

Figura 3: Separación en folds para conjunto de entrada ordenado luego de realizar un shuffle.

3. Experimentación

3.1. kNN con instancias de tamaño variable

Vamos a intentar entender cuando la cantidad de instancias de entrenamiento y de prueba dejan de representar un cambio significativo sobre las métricas de nuestro clasificador. Tomamos como hipótesis que a menor cantidad de imágenes peor va a ser el resultado del clasificador y a partir de cierta cantidad de instancias dejara de haber un cambio notorio. En este experimento vamos a analizar los resultados obtenidos en el método kNN para diferentes tamaños de instancias (500, 1000, 5000, 10000 y 20000 imágenes), con un 100 % para entrenamiento y un 20 % extra para validación. A continuación observamos la tabla con los resultados obtenidos:

Instancias	Accuracy
500	0.72
1000	0.845
5000	0.92
10000	0.9355
20000	0.947

Visualizando los resultados podemos ver que efectivamente a menor cantidad de instancias hay una diferencia notoria en el accuracy, y a medida que se agranda la cantidad de instancias este también crece. Sin embargo, vemos que a partir de 5000 instancias ya cuesta mas aumentar el accuracy. Viendo los resultados concluimos que para pruebas de muchas comparaciones podemos usar 5000 instancias y de esa forma generar mayor cantidad de pruebas en menor tiempo sin sacrificar demasiado la precisión de las predicciones.

3.2. PCA + kNN con instancias de tamaño variable

Para este experimento, lo que queremos hacer es minimizar la cantidad de imágenes sin perder accuracy y poder optimizar el tiempo de ejecución. La hipótesis que tenemos es que a medida que aumentemos el tamaño de la instancia el accuracy va a crecer mas lento hasta que en un punto no se vuelva notorio el aumento.

Al igual que en el experimento anterior, en este vamos a analizar los resultados obtenidos utilizando el método PCA + kNN para diferentes tamaños de instancias (500, 1000, 5000, 10000 y 20000 imágenes), con un 100 % para entrenamiento y un 20 % extra para validación. A continuación observamos los resultados obtenidos:

Instancias	Accuracy
500	0.78
1000	0.855
5000	0.938
10000	0.9495
20000	0.957

Al igual que con kNN, a medida que agrandamos el tamaño de las instancias obtenemos mejores resultados, a pesar de que es una diferencia mínima, no será los mejores en relación al tiempo que tengamos que invertir.

En conclusión si queremos lograr resultados mas precisos utilizaremos la mayor cantidad de instancias que permita el sistema, pero si lo que queremos es una mejoría en el tiempo de ejecución, con 5000 instancias obtenemos resultados suficientemente precisos como para probar distintas hipótesis.

3.3. kNN vs PCA + kNN tiempos de ejecución

La motivación sobre este experimento es para poder tener una idea de cuanto mas eficiente en relación velocidad y efectividad hay entre la ejecución de los métodos de kNN y PCA + kNN para la mayor instancia que podemos conseguir y una k y α prefijados. Nuestra hipótesis es que el método PCA + kNN va a ser ampliamente superior ya que podemos realizar una reducción considerable de dimensiones, lo cual disminuirá la cantidad de comparaciones que hace el sistema en comparación con el método kNN que siempre va a estar comparando todos los píxeles de la imagen con los de la base de entrenamiento. Ambos experimentos se ejecutarán con $k = 25$ y $\alpha = 28$ para PCA.

Pasamos a la instancia de experimentar y analizar los resultados. Para ambos métodos utilizamos todos los datos del archivo `data/train.csv`, dividiéndolo en 33600 imágenes de entrenamiento y 8400 de validación y con kNN obtuvimos el siguiente resultado:

```
Accuracy: 0.9536904761904762
CPU times: user 5min 33s, sys: 2.07 s, total: 5min 35s
Wall time: 5min 36s
```

Figura 4: Accuracy + tiempo (kNN)

Con PCA + kNN tenemos que analizar 2 tiempos distintos. Uno es la reducción a la matriz de transformación con α columnas y luego el tiempo que toma ejecutar kNN con las imágenes transformadas (matrices de menor dimensión). Los resultados obtenidos fueron los siguientes:

```
CPU times: user 7.68 s, sys: 499 ms, total: 8.18 s
Wall time: 8.3 s
```

Figura 5: Calculo de la matriz de transformación (PCA)

```
Accuracy: 0.9608333333333333
CPU times: user 39.4 s, sys: 397 ms, total: 39.8 s
Wall time: 39.6 s
```

Figura 6: Accuracy + tiempo (PCA + kNN)

En conclusión kNN tardó 5 minutos y 36 segundos y obtuvo un accuracy de 0.9536904761904762 mientras que PCA + kNN generó mejores resultados con un tiempo de ejecución total de 47.9 segundos y obtuvo un accuracy de 0.9608333333333333, entonces podemos afirmar que a pesar de tardar 7 veces menos y utilizar una matriz de que reduce su tamaño considerablemente logramos mejores resultados con PCA + kNN que utilizando únicamente kNN.

3.4. Variación del k en kNN

En este experimento vamos a ejecutar el método kNN con 5 k Folds variando el k entre 1 y 30 y la imágenes de entrenamiento y validación para luego analizar los siguientes datos: Accuracy, Recall y Precision, F1 score y Cohen's kappa y tiempos de ejecución.

En el análisis de accuracy observamos como a medida que agregamos mas k vecinos vamos perdiendo efectividad y que el rango de números donde tenemos los mejores resultados son entre $k = 3$ y $k = 7$ con accuracy mayor a 0.9650. El mejor resultado fue un accuracy de 0.967452 para $k = 3$.

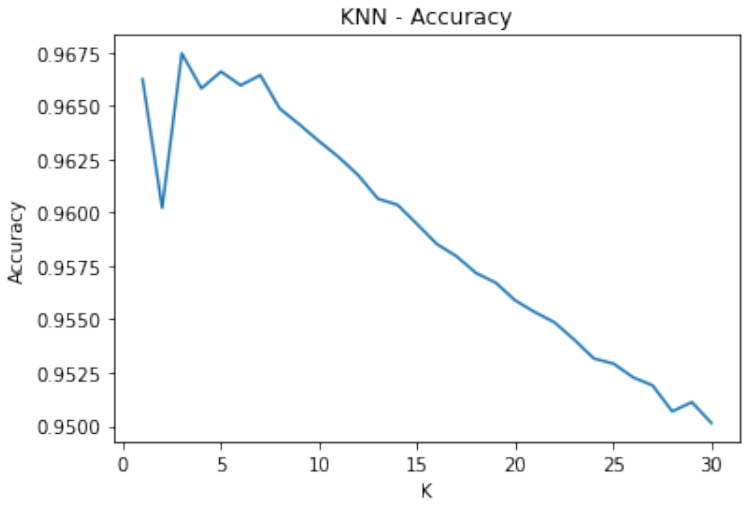
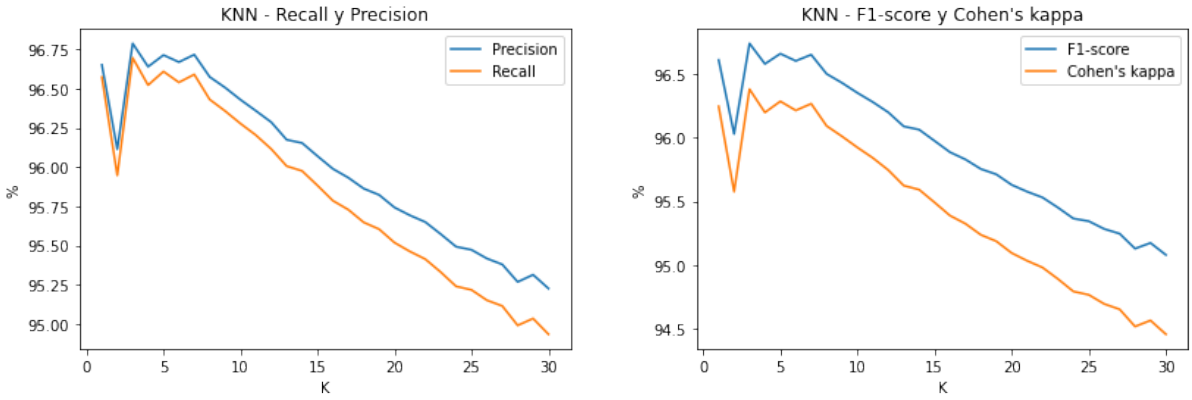


Figura 7: Accuracy (kNN)

Podemos observar que las figuras 8a y 8b son bastante similares entre si. Esto se debe a que todas estas métricas se desprenden del cálculo de accuracy (7), pero tienen características diferentes, aunque no sean suficientemente significativas como para darle relevancia en este experimento.



(a) Recall y precision (kNN)

(b) F1 score y Cohen's kappa (kNN)

Figura 8

En conclusión podemos observar a medida que agregamos mas k vecinos se va perdiendo relevancia en los datos y las predicciones comienzan a equivocarse con mas frecuencia, esto impacta en todas las áreas que analizamos ya que se relacionan entre si. En todos los gráficos analizados llegamos a que el mejor rango de k esta entre 3 y 7 ya que luego desciende sin tener ningún otro pico similar al del rango mencionado.

3.5. Variación del α en PCA + kNN

En este experimento vamos a correr el método PCA + kNN con 5 K-folds variando el α entre 1 y 31 y la imágenes de entrenamiento y validación con $k = 5$ para luego analizar los siguientes datos: Accuracy, Recall y Precicison, F1 score y Cohen's kappa y tiempos de ejecución.

En el análisis de accuracy observamos como a medida que aumentamos el α obtenemos mayor efectividad y desde del $\alpha = 14$, donde el resultado es de 0.953714 empieza a crecer de a 0.003 o menos por cada α que agregamos, lo cual vuelve irreconocible el aumento en el gráfico ya que es mínimo. El mejor resultado fue un accuracy de 0.973143 para $\alpha = 30$ ya que es una función creciente.

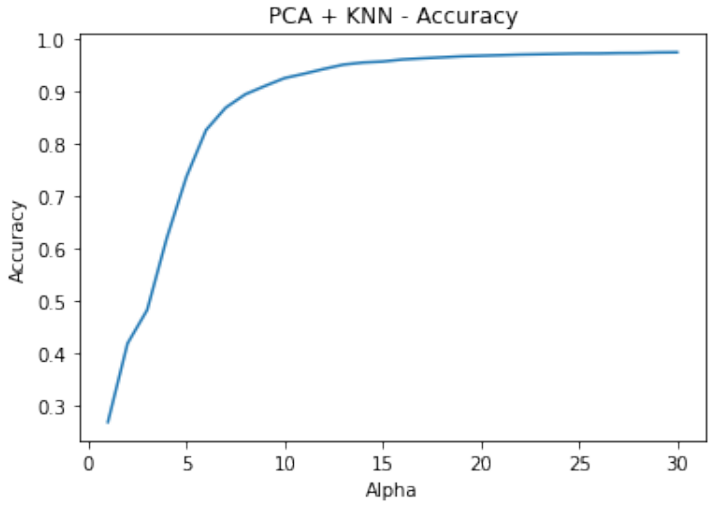
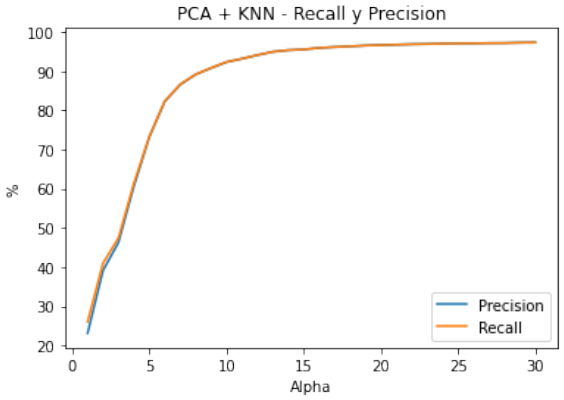
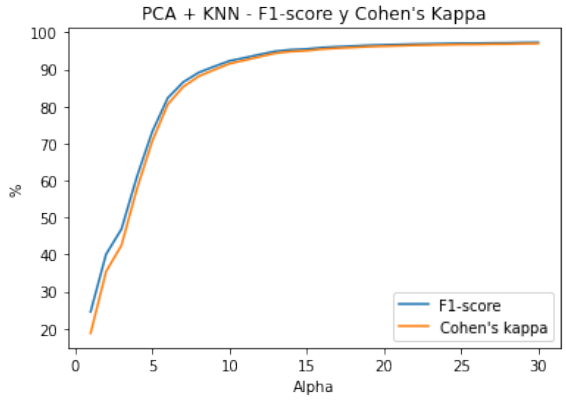


Figura 9: Accuracy (PCA + kNN)

Las diferentes métricas relacionadas con el experimento fueron las siguientes:



(a) Recall y precision (PCA + kNN)



(b) F1 score y Cohen's kappa (PCA + kNN)

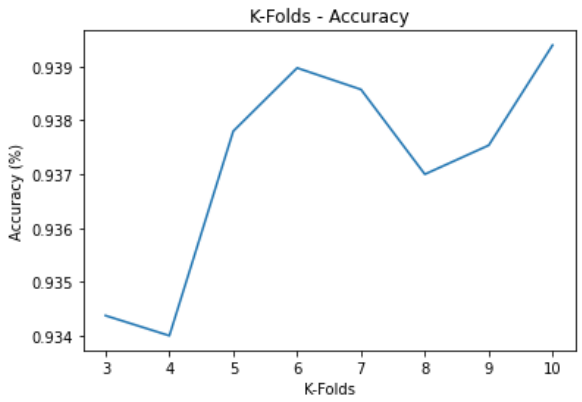
3.6. K-fold variable en PCA + kNN

En este experimento nos va a interesar poder analizar como al modificar la cantidad de folds en el cross-validation afecta a la calidad de nuestro clasificador PCA + kNN a la hora del entrenamiento.

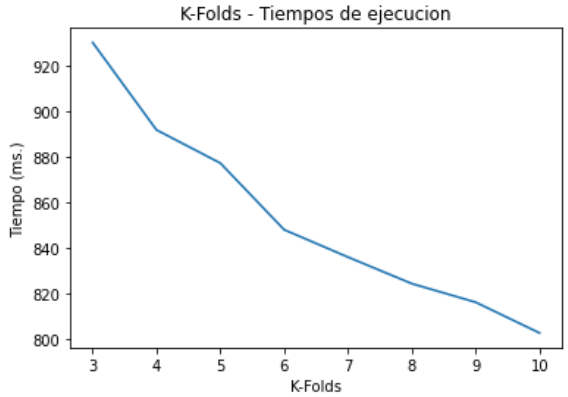
Vamos a realizar el experimento con pruebas entre 3 y 10 folds, con un conjunto de entrenamiento de 5000 imagenes, y medir el accuracy, tiempo de ejecución, recall, precision, F1-Score y Kappa de Cohen. Nuestra hipótesis es que el cambio de accuracy no va a ser relevante, aunque a mayor cantidad de folds vamos a obtener mejor resultados por disminuir la cantidad de test y aumentar la cantidad de training.

En la figura 11a podemos ver los resultados de los accuracy en función de la cantidad de K-Folds. Vemos que la variación de accuracy es menor al 1 %, aunque se nota que a mayor cantidad de folds mejora levemente el accuracy. En este caso nos ponemos a analizar en detalle que esta ocurriendo por detrás, y vemos que a medida que aumentan los folds disminuye el tamaño de las instancias de pruebas y aumentan las instancias de entrenamiento. Esto puede ser bueno porque al aumentar el training set obtenemos mayor variedad de imágenes; pero a la vez es malo porque a menor testing set, disminuye la confianza sobre nuestro accuracy, ya que esta medido en base a menor cantidad de pruebas.

En cuanto al tiempo de ejecución podemos observar en la figura 11b, la disminución del tiempo de ejecución a medida que aumentan los folds, creemos que esto ocurre por que el testing set disminuye y debe realizar menor cantidad de pruebas.

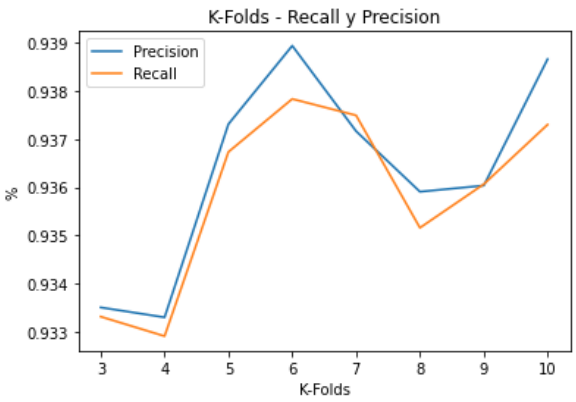


(a) Accuracy en función de la cantidad de folds

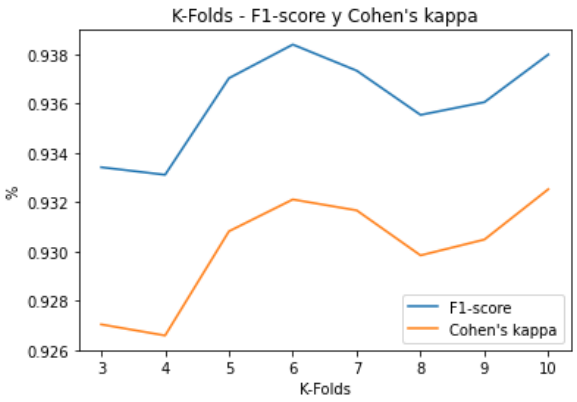


(b) Tiempos de ejecución en función de la cantidad de folds

Siguiendo con la figura 12a podemos observar que el comportamiento para el Recall y Precision es parecido que con el accuracy y lo mismo podemos observar en la figura 12b para el F1-Score y la Kappa de Cohen.



(a) Recall y Precision en función de la cantidad de folds



(b) F1-Score y Kappa de Cohen en función de la cantidad de folds

En conclusión podemos hacer el siguiente análisis general; aumentar la cantidad de folds es algo positivo para el entrenamiento, aunque hay que evitar reducir la cantidad de instancias de pruebas. Para conjuntos grandes esta bien aumentar la cantidad folds ya que el testing set sigue siendo numeroso. En nuestro caso creemos que usando 5 folds es un balance correcto que nos permite reservar el 20 % de nuestro conjunto para pruebas.

3.7. Números similares y errores de predicción

La motivación de este experimento es poder analizar los errores que puede llegar a tener el sistema a la hora de predecir un número, si tiene errores al azar o sus errores van a ser con imágenes similares entre si.

Tomando como base un experimento ejecutado con el método PCA + kNN con una instancia de 33600 imágenes de entrenamiento y 8400 de validación queremos analizar mas en profundidad los errores de cada numero en particular, ver con cuales se confundió mas y comparar sus similitudes a la hora de escribirlos.

La hipótesis que manejamos es que probablemente se confunda mas los números mas parecidos para el ojo humano como el 2 con 5, 4 con 9, 1 con 7, entre otros. A continuación se puede observar la matriz de confusión obtenida en el experimento.

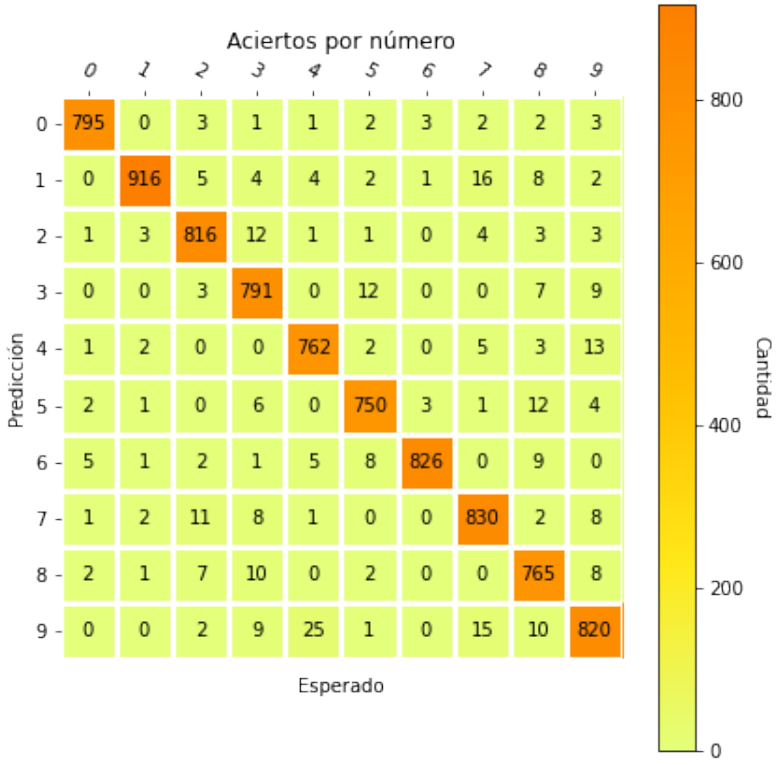
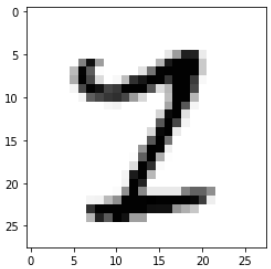


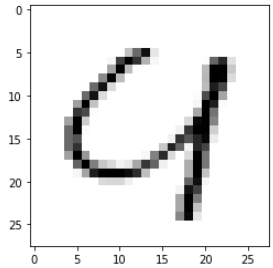
Figura 13: Matriz de confusión (PCA + kNN)

Luego de observar la matriz de confusión de dicho experimento podemos observar que el sistema predice de forma incorrecta números con características similares, como por ejemplo el 1 y el 7 (18 errores) o el 4 y el 9 (38 errores). A pesar de esto, creemos que nuestro sistema esta bastante ajustado a la realidad ya que se confunde en números cuando son muy parecidos como haría cualquier persona.

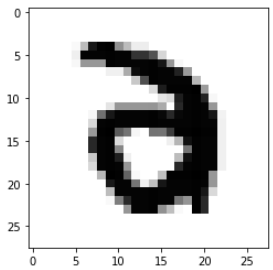
Para ver mas gráficamente estos errores, buscamos ejemplos en los que el sistema predice valores incorrectos de números y dibujamos las imágenes. Algunos ejemplos de esto son los siguientes:



(a) Numero: 1
Predicción: 7



(b) Numero: 9
Predicción: 4



(c) Numero: 2
Predicción: 3

Al observar estos números y sus predicciones ocurre que en algunos casos, ni siquiera nuestro grupo de trabajo sabia exactamente que numero era a la hora de verlo. Llegamos a

la conclusión de que en estos casos los errores son esperables y un sistema simple como el implementado no sera perfecto, mucho menos cuando hasta a una persona le cuesta distinguir que numero es el que queremos predecir.

3.8. Competencia en Kaggle

Aparte de los experimentos explicados anteriormente, nos resulto interesante probar nuestro sistema con el conjunto de entrenamiento y de prueba de la competencia de Kaggle de reconocimiento de dígitos. Los resultados obtenidos fueron los siguientes:

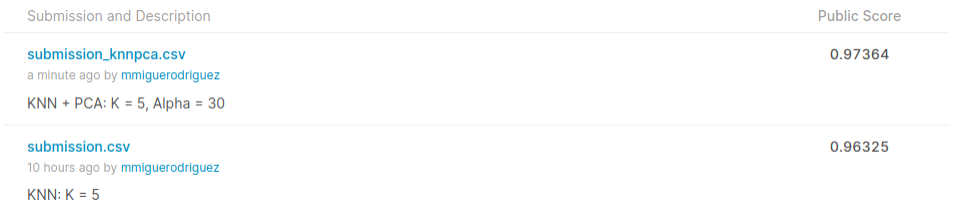


Figura 15: Resultados Kaggle

Como vimos en los experimentos anteriores, ocurre que kNN + PCA supera al método de kNN. La posición que obtuvimos en la competencia es el puesto 1502/2279, lo que nos coloca en el top 66 % algo que, para ser un algoritmo simple en comparación con las mejores implementaciones que se encuentran en el sitio, ya que, observando las mismas, estas utilizan redes neuronales que son mas escalables y eficientes que utilizar el método kNN o PCA + kNN.

Esto ocurre además, por el hecho de que si queremos por ejemplo, generar mayor cantidad de imágenes de entrenamiento a partir de las que ya tenemos haciéndoles transformaciones (data augmentation), utilizando el método kNN, la predicción de una imagen tomara mucho mas tiempo por el hecho de tener que comparar con todas las de entrenamiento.

3.9. Dígitos dibujados a mano

Aparte de todos los experimentos ya hechos, le agregamos una validación extra al trabajo, esto consistía en que cada uno de los integrantes del grupo dibuje los dígitos del 0 al 9, para así tener 30 imágenes de validación. Para este experimento utilizamos las 42000 imágenes de la base de entrenamiento para entrenar nuestro sistema. Algunos ejemplos de imágenes de entrenamiento dibujadas por cada uno de los integrantes son las siguientes:

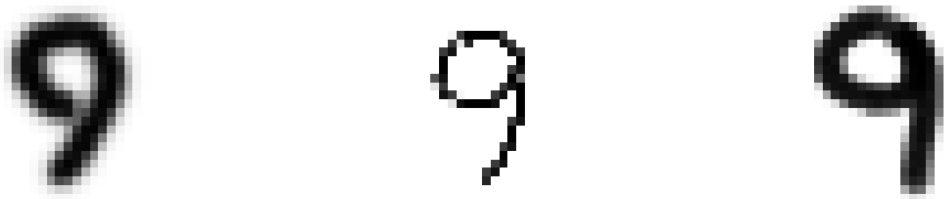


Figura 16: Número 9 dibujados por nosotros

Los resultados obtenidos con los distintos métodos fueron muy interesantes. Al ejecutar el sistema con $k = 5$ y utilizando únicamente el método kNN con las 30 imágenes de validación, el accuracy obtenido fue 0.53, mientras que con kNN + PCA con $\alpha = 28$ el accuracy obtenido fue de 0.56.

Algo que pudimos observar, es que los dígitos dibujados en 'lápiz' (menos bordes) por uno de los integrantes solían ser los mas difíciles a la hora de predecir por lo que, volvimos a ejecutar el experimento con 20 dígitos, y únicamente con los que estaban dibujados con un pincel (bordes mas anchos). Los resultados observados para este nuevo conjunto de prueba fue un accuracy de 0.75 con kNN y uno de 0.7 con kNN + PCA.

En conclusión, las diferencias acerca de como se dibujan los números resultan importantes a la hora de predecir nuevos valores que el sistema no vio, ya que, si la base de entrenamiento tiene en su mayoría números dibujados con un pincel, va a ser mas difícil predecir números nuevos que fueron dibujados con otras características.

3.10. Relación k vs cantidad de instancias

En el siguiente experimento nos interesa saber como varían los resultados para distintos valores de k y tamaños del conjunto de entrenamiento. Para variar el k utilizaremos los valores [1, 5, 10, 30, 50, 100] y los tamaños de entrenamiento serán [500, 1000, 5000, 10000, 20000]. Además para correr el experimento usamos el método PCA + kNN con $\alpha = 28$.

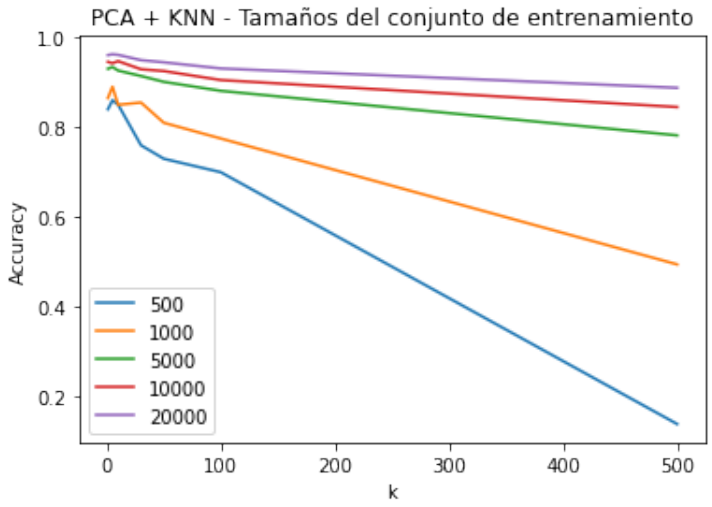


Figura 17: Tamaño del conjunto de entrenamiento vs k

En el experimento podemos ver la consistencia que tienen las instancias de mayor tamaño contra las de menor. En especial vemos como para los tamaños 5000, 10000 y 20000 no es tan profundo el descenso en los resultados cuando el k aumenta. Cuando analizamos para el conjunto de 1000 se ve que ya es notorio el descenso cuando llegamos a $k = 500$, y cuando tenemos el conjunto de 500 con $k = 500$ pierde todo el sentido ya que va a responder siempre el mismo número (el que mas veces esté en el conjunto de entrenamiento). En conclusión, a mayor tamaño de entrenamiento mayor consistencia de resultados con un k variable.

4. Conclusiones

Luego de lo visto anteriormente podemos sacar las siguientes conclusiones. En cuanto a la clasificación kNN vimos que el método no es lo mas efectivo de por si solo, ya que al agregarle la reducción de componentes de PCA mejoran sus resultados. También vimos que en kNN el parámetro k no mejora los resultados a medida que crece, sino que sus mejores resultados los da cuando toma valores entre 3 y 7. Para PCA vimos que es un método súper efectivo para mejorar la calidad de nuestro clasificador y además reducimos los tiempos de ejecución en kNN. En cuanto al parámetro α pudimos realizar el análisis que a partir del valor $\alpha = 30$ se aplana la curva de mejora en los resultados. Otro punto a tener en cuenta en kNN es el tamaño de entrenamiento a la hora de fijar nuestro k, notamos que puede tomar como valor mínimo 1, pero no estaríamos tomando en cuenta posibles errores de predicción, y como máximo el tamaño del conjunto de entrenamiento, aunque este valor no tiene sentido ya que el sistema no funciona correctamente. En cuanto al tamaño de entrenamiento notamos que para nuestro caso de 42000 instancias de pruebas, no necesariamente obtenemos los mejores resultados utilizando todas las instancias, ya que pudimos ver que a partir de 5000 instancias la varianza entre los resultados es cada vez menos notorio a medida que crece.

Luego realizamos un análisis de cross-validation, viendo cual era el mejor K para dividir en K-Folds nuestro conjunto de pruebas. Notamos que al aumentar la cantidad de folds mejorábamos los resultados del clasificador, aunque no hay que olvidarse que estamos reduciendo el tamaño del conjunto de validación, que resulta en una perdida de confianza sobre nuestros resultados. Observamos que mantener un punto medio era lo mejor por eso decidimos que una partición entre 5 folds era una buena opción para este trade-off. El método de cross-validation es útil para conjuntos de pruebas muy variados ya que nos permite poder ajustar nuestro clasificador con la mayor cantidad de conjuntos de validación. En conjuntos mas uniformes no tendría tanto sentido usarlo ya que todos los folds tendrían instancias de pruebas muy parecidas, aunque este no fue nuestro caso.

Por ultimo decidimos llevar nuestro clasificador a pruebas y comparación del mundo real. Ya con nuestros parámetros $k = 5$ y $\alpha = 30$ decidimos realizar unas pequeñas pruebas con números dibujados por nosotros para ver si efectivamente funciona nuestro clasificador. Para números dibujados de manera parecida al conjunto de entrenamiento obtuvimos buenos resultados, pero para números con otros estilos de dibujo no pudo clasificar de forma correcta.

Referencias

[1] Kaggle Digit Recognizer - `www.kaggle.com/c/digit-recognizer`