DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP 2 - Reconocimiento de Números

Keywords: Machine Learning - PCA - KNN

01/11/2020 Métodos Numéricos

Grupo TP: 4

Integrante	LU	Correo electrónico
Miodownik, Federico	726/18	fede@miodo.com.ar
Puerta, Ezequiel	812/09	armando.ezequiel.puerta@gmail.com
Sujovolsky, Tomás	113/19	tsujovolsky@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción **3**
1.1. Motivación . . . 3
1.2. Presentación . . . 3
1.3. Términos usados . . . 3

2. Desarrollo **4**
2.1. kNN . . . 4
2.1.1. Algoritmo . . . 4
2.1.2. Posibles mejoras . . . 4
2.2. PCA . . . 5
2.2.1. Algoritmo . . . 5
2.3. Métricas de efectividad . . . 5

3. Experimentación **7**
3.1. kNN . . . 7
3.2. PCA . . . 9
3.3. Ajustes finales de experimentación . . . 13

4. Conclusiones **15**

5. Apéndice **16**

1. Introducción

1.1. Motivación

El reconocimiento de imágenes es un problema muy abarcado y analizado dentro del área de la computación. Desde reconocimiento de patentes hasta vehículos autónomos, son muy diversos los usos que pueden tener, pero todos comparten un mismo objetivo: interpretar la realidad sin la necesidad de una etiqueta.

1.2. Presentación

En este trabajo práctico abordaremos el reconocimiento de números (dígitos del cero al nueve) mediante dos métodos : los K vecinos más cercanos y el análisis de componentes principales (o kNN y PCA por sus siglas en inglés). Estudiaremos sus ventajas y limitaciones y exploraremos algunos de sus usos. Para medir su efectividad en la clasificación, utilizaremos métricas populares dentro de Machine Learning, tal como *accuracy*, *recall* y *precision* así como el tiempo de ejecución necesario para obtener los resultados. La base de datos que usaremos es MNIST de Kaggle.[1]

1.3. Términos usados

1. kNN: K vecinos más cercanos. Método numérico.
2. PCA: Análisis de componentes principales. Método numérico.
3. K-fold Cross Validation: Método de validación cruzada de datos en K bloques.
4. Accuracy: Métrica de efectividad de un método.
5. Precision: Métrica de efectividad de un método.
6. Recall: Métrica de efectividad de un método.
7. Confusion matrix: Gráfico para la efectividad de un método que reúne varias métricas.
8. Train: la componente de entrenamiento de la base de datos, utilizada para optimizar los modelos.
9. Test: la componente de validación de la base de datos, utilizada para poner a prueba los modelos.

[1] disponible en https://www.kaggle.com/c/digit-recognizer

2. Desarrollo

2.1. kNN

Desde la teoría, al buscar clasificar, kNN se basa en dos supuestos:

1. los vecinos cercanos a un elemento son similares al elemento en cuestión,
2. existe una manera relevante de medir esta "cercanía", en nuestro caso será la norma dos vectorial.

Si se cumplen los supuestos, entonces se puede asumir que clasificar de acuerdo a la clase de los vecinos más cercanos tiene poder predictivo, si no se cumple alguna entonces no tiene por qué ser así. Ahora bien, cuando estamos trabajando con vectores, el concepto de distancia depende de cada una de las componentes, y al aumentar la cantidad de dimensiones, el espacio disponible aumenta exponencialmente mientras que el ocupado crece en menor proporción con la cantidad de muestras disponible. [Grus 2019]

Esto se convierte en un problema de consecuencias prácticas para la utilidad de este algoritmo en dimensiones altas, usualmente conocido por el nombre de "maldición de la dimensionalidad". Para atravesar esta dificultad, una opción es continuar incrementando la cantidad de muestras, tal de seguir llenando el espacio disponible. Esto es válido y en la práctica es útil, hasta un punto. Eventualmente la potencia de cálculo requerida para realizar las comparaciones contra miles o millones de muestras superará la disponible o el tiempo deseado para obtener las respuestas. Otra posible solución es proceder utilizando otros métodos que puedan reducir la dimensionalidad del problema encontrando patrones comunes en los datos. Uno de estos algoritmos es el llamado PCA, el cual detallaremos más adelante.

Con respecto a kNN, nos surge la pregunta de si 42000 muestras es suficiente para mantener el poder predictivo del clasificador, a sabiendas de que contamos con 784 dimensiones para la activación de píxeles de cada imagen.

2.1.1. Algoritmo

Definimos kNN como una clase con tres funciones: la inicialización (donde se asigna el K), el entrenamiento y la predicción. Tanto la inicialización como el entrenamiento simplemente guardan los datos pasados por parámetros como variables internas. En el caso de kNN sin PCA, en el entrenamiento se toma el conjunto de n imágenes (de 28x28 píxeles) en forma de una matriz $M \in \mathbf{R}^{n \times 784}$, donde cada fila representa a cada imagen vectorizada. De esta forma, la celda $M(i, j)$ representa el pixel $(div(j, 28), mod(j, 28))$ de la i-ésima imagen.

El verdadero algoritmo está en la predicción. Éste toma como parámetro una matriz de imágenes y devuelve un vector con las predicciones. Por cada fila de la matriz parámetro (es decir, cada imagen), se utiliza la función para predecir un solo vector o imagen. Esta función toma para todas las imágenes guardadas como variable interna en el entrenamiento la norma de la resta con el vector parámetro y la guarda junto a su *label* en una listas de tuplas. Una vez comparado el vector parámetro con todos las imágenes guardadas, se ordena la lista de manera creciente y se toman las primeras K (también guardada en una variable interna) posiciones de la lista. Acá guardamos los K vecinos mas cercanos, ahora resta realizar la votación para ver cuál tomamos. Para esto, nos aprovechamos un poco de la problemática que estamos resolviendo. Como las clases que analizamos van del 0 al 9, creamos un nuevo vector de 10 posiciones inicializadas en cero e iteramos la lista, sumando de a 1 en la posición dictada por el *label* de cada elemento de la lista. Para finalizar, tomamos el argumento máximo de este vector y eso determina la predicción.

Un dato curioso que descubrimos en la experimentación es que los resultados entre nuestra implementación y la de la biblioteca *sklearn* (que quisimos utilizar como comparatoria) son idénticos.

2.1.2. Posibles mejoras

El algoritmo implementado cuenta las apariciones de cada clase en la cantidad de vecinos elegida, pero...¿cómo decide en caso de empate? A priori, no cuenta con ningún criterio lógico para seguir en estos casos. ¿Y qué tan posibles son estos casos realmente en la práctica? Claramente con $K = 1$ esto no es un problema, pero al aumentar K comienza a ser una posibilidad. Con $K = 2$, la probabilidad de empate es máxima, porque son todos los casos donde ambos vecinos son distintos. Este factor de error es observable en la sección 3.1 (Figura 1), donde nuestro algoritmo de kNN pierde aproximadamente un 2 % de *accuracy*

en todos los casos cuando $K = 2$, que recupera a medida crece K y se vuelven más improbables los casos fronterizos.

Creemos que implementar un criterio de decisión para estos casos puede mejorar la precisión del método de manera significativa para los casos borde. Si bien no probamos su efectividad en la práctica, proponemos los siguientes criterios:

1. en caso de empate, eliminar el vecino más lejano y volver a encuestar. De repetirse el resultado, continuar realizando el mismo proceso hasta lograr una decisión única.

2. en caso de empate, para las clases que hayan empatado, encontrar el promedio de la distancias de los dígitos de la clase contra el *test* y seleccionar la clase que tenga el mínimo.

2.2. PCA

Como se explica anteriormente, kNN es sensible a la dimensionalidad. Es de interés práctico el encontrar un algoritmo que pueda, perdiendo la mínima información, reducir la cantidad de dimensiones de las muestras. El que implementamos es PCA, el cual encuentra las correlaciones más fuertes entre los distintos parámetros de las muestras, y las reúne como combinación lineal en una sola dimensión para cada patrón encontrado. Como si fuera poco, las componentes transformadas se ordenan por varianza descendiente, lo que significa que los datos son más diferenciables entre sí en esa dimensión, por lo que son más fáciles de clasificar, y consecuentemente tienen mayor porcentaje de información total de la muestra que las siguientes. En la sección 3 veremos gráficamente la cantidad de dimensiones que teóricamente podemos determinar para asegurar estar perdiendo la menor información posible aprovechando la reducción de dimensionalidad. Compararemos este valor con el óptimo encontrado vía experimentación.

Por cómo funciona kNN, sabemos que, al aplicar PCA, cada dimensión tendrá el mismo peso que las demás para el cálculo de la distancia. Es por esto que teorizamos que existirá un valor de cantidad de dimensiones elegidas donde se maximiza el *accuracy* al realizar kNN, donde elegir menos resulta en perder información importante, y elegir más resulta en agregar ruido innecesario a la clasificación.

2.2.1. Algoritmo

Definimos PCA como una clase con tres funciones: la asignación de la cantidad de componentes, el entrenamiento y la transformación. La predicción luego se realizará con kNN.

La asignación es simple, guarda como variable interna la cantidad de componentes que se pasan como parámetro.

El entrenamiento esta en conseguir estas componentes principales. Para ello, utilizamos el método provisto por la cátedra en la presentación del trabajo práctico. Primero, calculamos la matriz de covarianzas. Luego le tomamos los n primeros autovectores, siendo n la variable interna guardada en la asignación. Estos autovectores los conseguimos utilizando el método de la potencia con deflación. Para ver si converge o diverge, la tolerancia de corte del método de la potencia la *seteamos* en 10000 iteraciones y en la práctica no nos encontramos con ningún problema. El épsilon que usamos para indicar que efectivamente encontramos el autovalor convergente es e^{-16}. Una vez obtenidos los autovectores, los guardamos como matriz en otra variable interna para usar en las subsecuentes transformaciones.

Cabe destacar que no tenemos ningún método para prevenir dar con un vector que haga que el método de la potencia divergiera, ya que las probabilidades de que eso sucediera son muy bajas y las ramificaciones serian notables. Si sucediera, simplemente volveríamos a correr el algoritmo.

La transformación es entonces simplemente la multiplicación de una matriz de imágenes, por las componentes principales guardadas como variable interna después del entrenamiento.

2.3. Métricas de efectividad

Para definir las métricas que utilizaremos en este trabajo, primero definimos cuáles son las opciones que pueden ocurrir cuando un clasificador predice un dígito. Como en el caso estudiado contamos con 10 clases (dígitos distintos), podemos *binarizar* las respuestas, es decir, tomando un dígito como el relevante, y al resto como los no-relevantes, puede ocurrir un:

1. True positive (tp): el clasificador predice correctamente el número relevante.

2. False positive (fp): el clasificador considera relevante un número que no lo era.
3. True negative (tp): el clasificador rechaza correctamente un número no relevante.
4. False negative (fn): el clasificador rechaza incorrectamente un número relevante.

Esta clasificación se puede extender por repetición considerando cada dígito distinto como el relevante, y resulta cómodo definirlos de esta manera para poder entender **qué mide exactamente cada métrica.**

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn}$$
$$Recall = \frac{tp}{tp + fn}$$
$$Precision = \frac{tp}{tp + fp}$$

Durante la experimentación, usaremos una combinación de las tres métricas definidas para reportar y analizar los resultados. *Accuracy* será la métrica estándar usada, la cual sabemos que será efectiva como tal porque las clases se encuentran bien balanceadas y tenemos una cantidad satisfactoria de dígitos totales. El *recall* (sensibilidad) y la *precision* (precisión) serán utilizadas en análisis más profundos de algunos resultados, para poder diferenciar mejor la efectividad de los métodos respecto a cada dígito, o la tendencia a responder algún dígito en particular con mayor frecuencia que otro, entre otros factores.

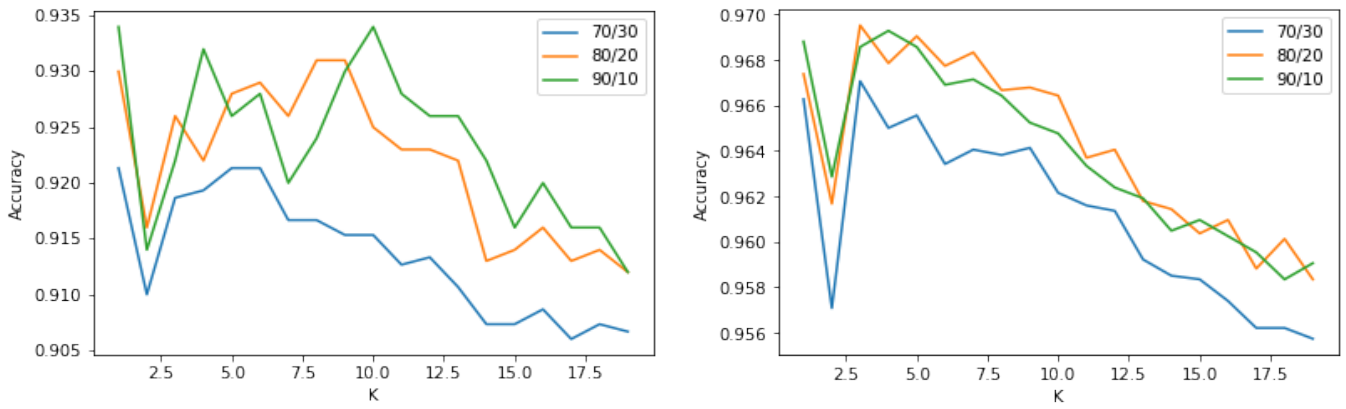
3. Experimentación

Dada la magnitud del problema de clasificación y la cantidad de variables involucradas, creemos que la mejor manera de armar un clasificador efectivo es con extensa experimentación, comenzando desde los casos más simples, y con pocos retoques de variables, para luego acarrear los aprendizajes a métodos de mayor complejidad.

3.1. kNN

Decidimos empezar la experimentación de manera de conocer a fondo el método de kNN, y cómo modifican sus variables principales a las métricas que finalmente obtiene. Primero, fijamos una cantidad de muestras totales a considerar, de las cuales se obtendrán los dígitos de *train*, los utilizados como "vecinos", y los de *test*, los cuales se medirán contra éstos para obtener su clase. El valor de K solamente tiene sentido en el rango $[1, \#(train)]$, y la interpretación corresponde con: desde comparar frente al más cercano solamente, hasta comparar contra todos. A medida que K se acerca a $\#(train)$ creemos que el método va a perder precisión, pues su implementación otorga igual peso al más cercano de todos que al k-ésimo. En el caso límite, el método compara con todos, por lo tanto devolverá la moda de la base de datos utilizada para comparar para todos los dígitos a clasificar.

Para comenzar, decidimos reducir el dataset a 5000 dígitos, manteniendo el balance de clases, para modificar la proporción *train*/*test* y cómo ésta puede tener un efecto en el K óptimo de kNN. Luego repetimos el experimento con el dataset completo (42000 dígitos). La métrica utilizada para determinar la efectividad de cada combinación fue *accuracy*, la cual consideramos suficiente al existir diez clases y un balance saludable de las mismas.



(a) 5000 digitos

(b) 42000 digitos

Figura 1: K óptimo frente a distinto tamaño de muestra

A simple vista pudimos ver la sensibilidad que tiene kNN frente a la cantidad de muestras. En efecto se aprecia que el K óptimo es distinto en ambos casos. También lo es para cada composición *train*/*test*, aunque al incrementar considerablemente las muestras en *train* este efecto pierde fuerza y las curvas se asemejan más.

Un detalle interesante (y una falencia del método) es que al tener $K = 2$ el mismo se encuentra más veces con un empate técnico entre dos clases que en otros valores de K, y no tiene implementado cómo elegir en estos casos[2]. Es por esto que vemos que, independientemente de la cantidad de muestras, $K = 2$ presenta menor *accuracy* que otros valores cercanos.

También podemos llegar a apreciar la pérdida de efectividad del método al incrementar K más allá de los máximos locales, esto se debe a que los nuevos vecinos cercanos tienen un voto igual de válido que el más cercano e incorporan ruido a la decisión. Donde quizá los 4 dígitos más cercanos eran los correctos, se sumaron 10 dígitos incorrectos y la decisión resulta errada.

A partir de las conclusiones que obtuvimos en este experimento, decidimos:

[2]Se sugieren posibles soluciones en la sección 2.1.1

1. seguir variando la cantidad de muestras utilizadas en cada método, para poder utilizar como comparatoria su sensibilidad a la misma.

2. fijar la composición *train/test* en 80/20, para simplificar la cantidad de variables e información a procesar manteniendo un *accuracy* alto y a la vez estable.

3. fijar el K de kNN en **3**, entendiendo el mismo como un valor de alta efectividad en la composición 80/20 para distintas cantidades de muestras, aprovechando también su imparidad con tal de minimizar el efecto de empate.

Para profundizar los conocimientos, decidimos mirar más en detalle la efectividad de kNN en sus valores óptimos encontrados. Utilizando la base de datos completa, $K = 3$ y la proporción 80/20, graficamos una matriz de confusión para observar no sólo **cuántas** veces se equivoca el método, si no con **cuáles** números lo hace.

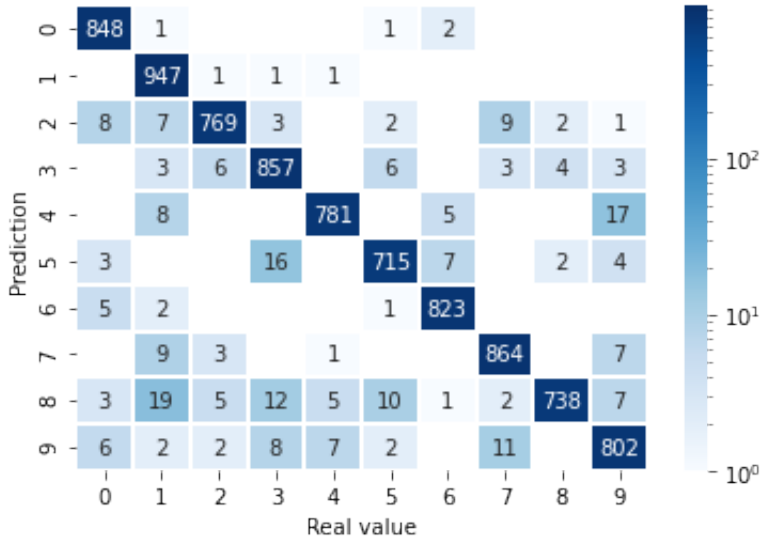


Figura 2: Matriz de confusión de kNN

Observamos bastante información relevante:

1. El algoritmo confunde algunos dígitos con otros específicos, pero no viceversa. Por ejemplo, confunde 19 veces el 1 con el 8, pero el 8 real nunca lo confunde con un 1. Dado que kNN solamente mide distancias en términos de la activación de píxeles, es lógico pensar que deberíamos encontrar relaciones de equivalencia (el 1 está cerca del 8) y que consecuentemente, los confunda entre sí. La evidencia indica que las relaciones tienen un fuerte carácter de orden. No estamos seguros de qué es lo que genera este comportamiento.

2. En líneas similares, se observa una tendencia general a preferir algunos números contra otros en la clasificación. Cuando el dígito es un 8, no suele equivocarse (alto recall), pero asimismo en 64 ocasiones donde el dígito **no** era 8, kNN lo asignó así (baja precisión). Podemos ver el efecto contrario cuando se trata del 0 ó 1. Como tiende a evitar esos dígitos, finalmente sólo decide predecirlos cuando son obvios (o no tiene otras opciones), lo que explica que cuando predice esos dígitos, entonces seguro que lo son (alta precisión pero bajo recall).

Por último, para tener mayor seguridad de que el K óptimo encontrado efectivamente lo sea, debemos incorporar una medida de generalización del parámetro, de la base de datos *train* utilizada. Esto es importante porque puede haber ocurrido que los dígitos que conformaron el *train*, el cual nunca se modificó hasta ahora, pueden haber sido buenos para el *test* actual, pero no necesariamente buenos para nuevas muestras futuras, lo que se conoce como $overfitting$. Para corroborar la clasificación usaremos entonces K-fold Cross Validation manteniendo la proporción 80/20, con la base de datos completa, y $K = 3$. Esperamos que al graficar el *accuracy* en cada iteración (donde se modifica la conformación del *test* y *train*) obtengamos resultados similares. En caso que no, deberíamos poner en tela de juicio que los parámetros encontrados sean los más óptimos.

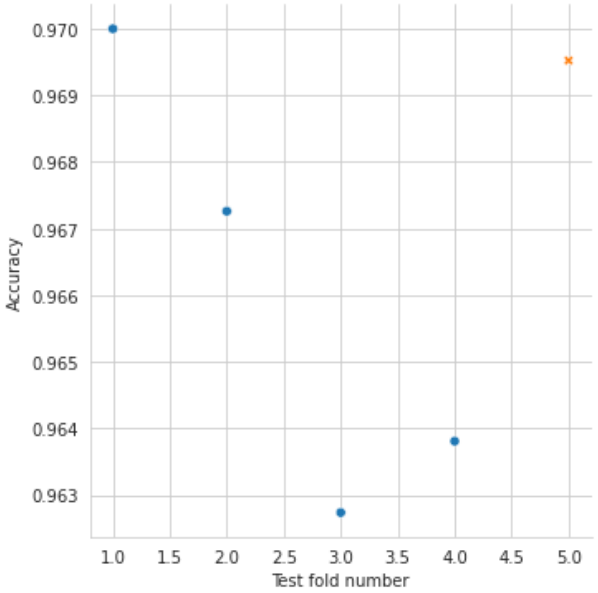


Figura 3: Revisión de los parámetros con K-fold

Al realizar la validación cruzada, nos encontramos con que las diferencias de *accuracies* cambiando el conjunto de *test* son en todos los casos menores al 1 %. Esto nos permite asegurar con mayor confianza que $K = 3$ es un parámetro óptimo robusto frente al problema del $overfitting$.

Una observación que queremos mencionar, es que, si bien podríamos evaluar con mayor correctitud los parámetros de kNN corriendo siempre el algoritmo con validación cruzada, ya que se pueden encontrar puntos donde un K es ligeramente mejor (o peor) que el encontrado, esto es muy pesado en términos de tiempo de cómputo, un factor que consideramos importante porque la experimentación fue hecha localmente. No descartamos que utilizando herramientas de mayor procesamiento y con más tiempo de cómputo se puedan encontrar parámetros más óptimos que los que se aquí se presentan.

3.2. PCA

Con los aprendizajes que extrajimos del experimento anterior, decidimos implementar una mejora al método de kNN utilizando PCA. Este método transforma el espacio original de los vectores imagen en otro de dimensiones menores, optimizado para cancelar las covarianzas entre píxeles manteniendo la varianza máxima. Por lo tanto, al realizar kNN a los vectores transformados, se compara en menores dimensiones, lo que nos permitirá inferir sobre la sensibilidad de kNN frente a la dimensionalidad, e incluso diferenciar mejor los dígitos entre sí. Para encontrar la cantidad de dimensiones óptima para transformar el espacio, fijaremos la composición *train/test* y el K obtenido en el experimento previo, modificando solamente p en intervalos regulares. Mediremos la efectividad de cada p por el *accuracy* que se obtenga, y finalmente veremos más a fondo los mejores resultados que obtengamos.

Algo a tener en cuenta en este experimento es que kNN asume de igual importancia a cada dimensión de los datos, pero entendemos que PCA reúne los datos en *componentes principales* donde cada una contiene menos información que la anterior. Esta idea está atada a que el algoritmo del método de la potencia (el cual obtiene los autovalores de la matriz de covarianza) los obtiene justamente en orden descendente. Esto nos lleva a teorizar que, a medida aumenta p, iremos incrementando el *accuracy* hasta llegar a un máximo, donde desde allí en adelante las nuevas dimensiones incrementen el ruido aportando cada vez menos información, lo que reducirá la métrica cada vez más.

Para comprobarlo, analizaremos el dataset reducido a 5000 números en proporción 80/20, con K vecinos fijos en 6[3] y haremos variar las componentes principales desde 10 hasta 600, en incrementos de 10 (para ahorrar tiempo de computo). Luego, tomaremos el subconjunto de componentes principales con mayores *accuracies* y los veremos en incrementos de 5 para tener un análisis más detallado de su comportamient, y finalmente veremos los mejores candidatos de 1 en 1.

[3]el cual entendemos como un compromiso entre el óptimo en la muestra completa (3) y el óptimo en 5000 números (8), el cual resulta subóptimo al escalar la cantidad de muestras,

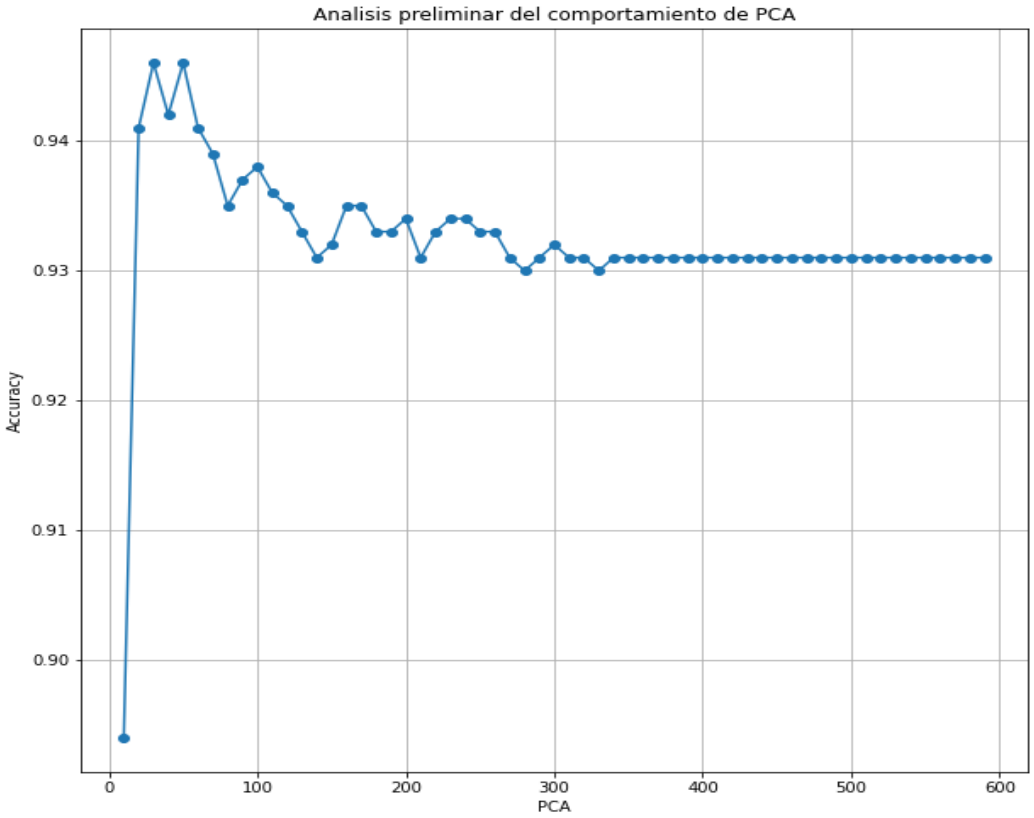


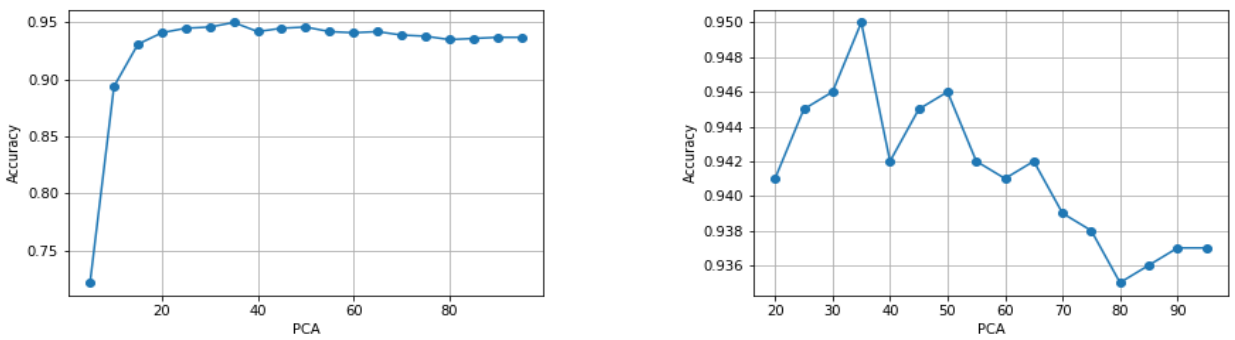
Figura 4: Accuracy por valor de PCA, en incrementos de 10

Hay tres elementos notables de esta observación. El primero es el gran salto desde 10 componentes principales hasta 20 componentes principales. Esto se debe a que la cantidad de información en una componente dada es directamente proporcional con su orden, como se ve en [University s.f.]. Esto se vera nuevamente en la próxima observación.

El segundo elemento notable es que para cierto punto, al rededor de las 40 componentes, el accuracy empieza a bajar. Intuimos que esto significa que la información que estas otorgan ya no es tan relevante y comienza a formarse una especie de ruido que hace que baje el accuracy.

El tercer elemento notable es el punto en el cual la función se vuelve constante, a partir de las 350 componentes. Inferimos que esto se debe a que estas componentes dejan de otorgar información significativa.

Realizamos entonces una segunda observación, mas detallada, del rango de 5 a 100 componentes principales en incrementos de 5.



(a) Desde 5 hasta 100 en incrementos de 5

(b) Desde 20 hasta 100 en incrementos de 5

Figura 5: Accuracy por valor de PCA, en incrementos de 5

Nuevamente, notamos el gran salto de 5 componentes principales a 10. Esto es otro ejemplo de lo mencionado en la observación anterior. Sin embargo, si analizamos nuevamente el gráfico dejando de lado los primeros valores de pca y vemos desde las 20 componentes principales en adelante, observamos que hay un pico entre 30 y 40. Para mayor comprensión decidimos realizar una ultima observación desde 30 a 40 componentes principales en incrementos de 1, para así analizar este pico.

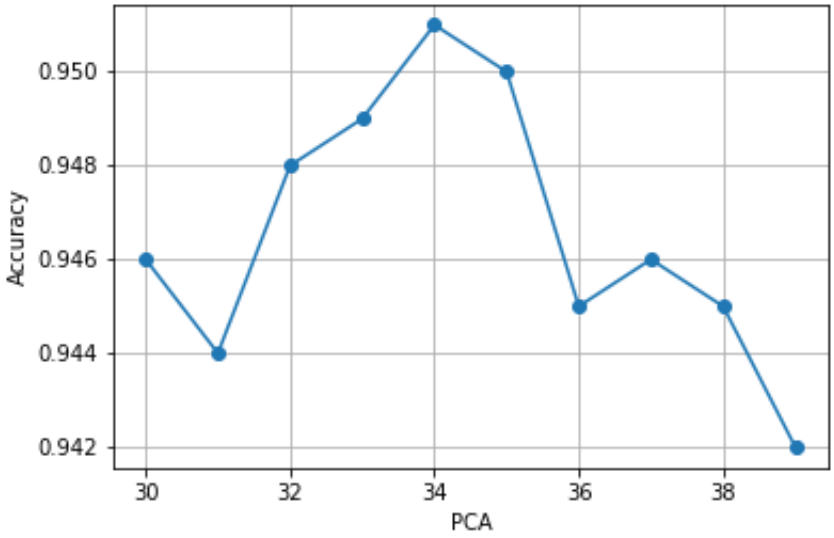
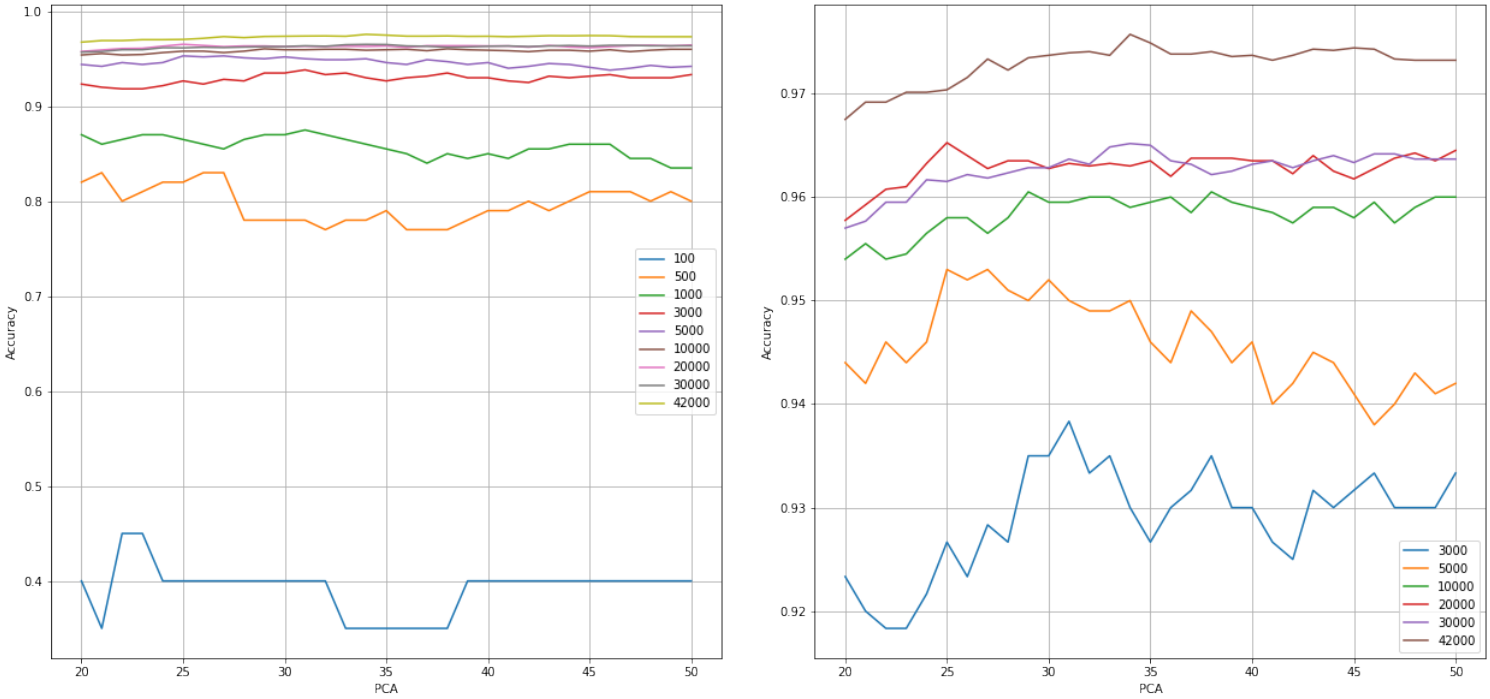


Figura 6: Accuracy por valor de PCA, en incrementos de 1

Podemos ver entonces que son alrededor de 34 las componentes principales que acarrean la información mas relevante sin generar ruido entre si.

Pero, ahora, con este nuevo numero 34 en mente, salta una nueva incógnita ¿Son las componentes principales mas relevantes de un sistema, propias del dataset o de las clases en cuestión? Para ello, planteamos un nuevo experimento. Cambiando la cantidad de datos tomados del dataset, y fijando los k vecinos en 6, decidimos ver si el comportamiento de la variación de componentes principales era similar. Nosotros hipotetizamos que para subconjuntos con poca cantidad de datos, el entrenamiento sera débil y no podremos sacar información relevante, pero a partir de los de mayor cantidad de datos, veremos un comportamiento similar entre los distintos subconjuntos donde a mayor cantidad de datos mejor accuracy tendrán y esperamos ver los picos de los gráficos alrededor de las 34 componentes principales.



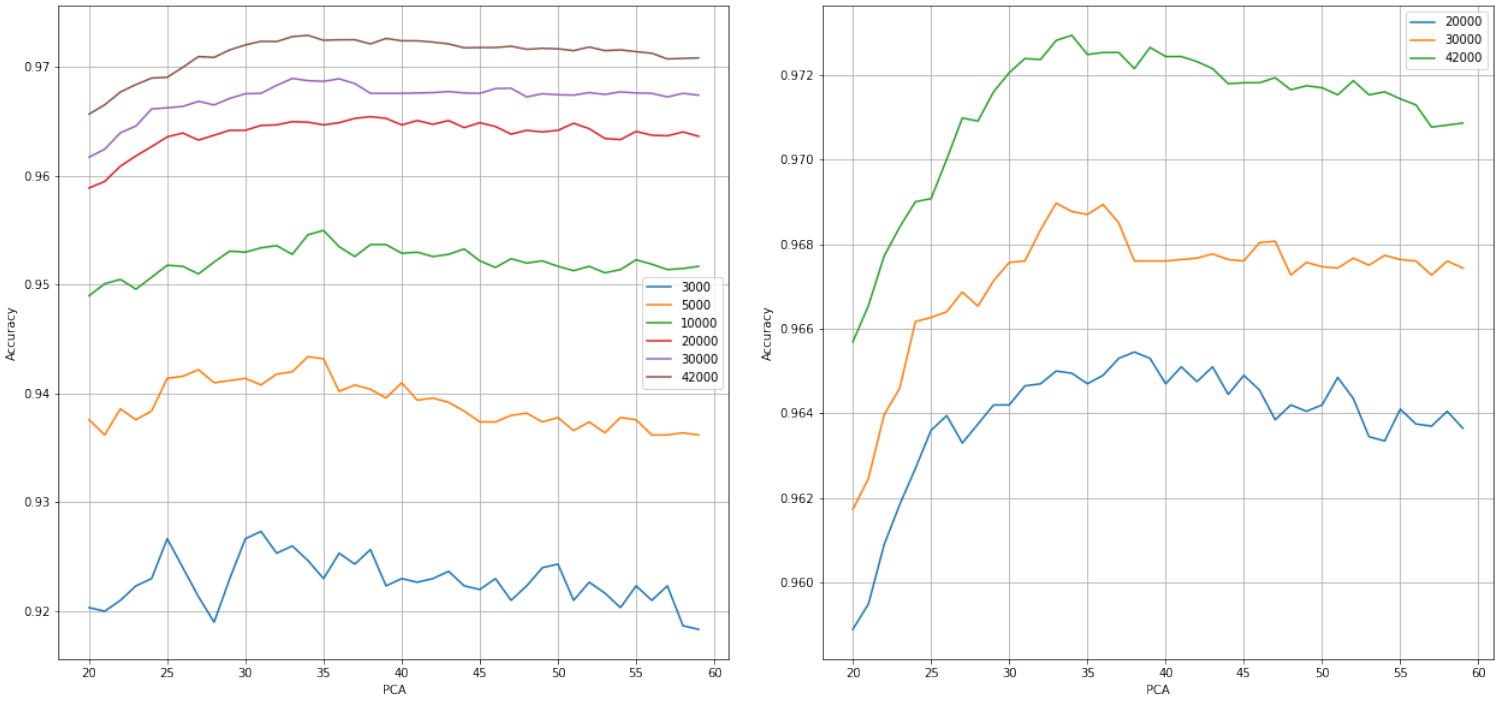
(a) Accuracy de PCA por tamaño de muestra

(b) Top 6 accuracy de PCA por tamaño de muestra

Figura 7: Análisis del comportamiento PCA

Como creíamos, para subconjuntos de pocos datos, el entrenamiento es débil y su información es poco relevante, por ende, en la fig 7b excluimos esos subconjuntos y vemos el comportamiento con datasets de suficientes muestras.

Para la primera evaluación analizamos desde 20 componenetes principales hasta 50 en incrementos de 1. Parecen tener un comportamiento similar pero al ver los valores de PCA que dan el máximo accuracy de cada subconjunto del dataset vemos que no solo no son 34 si no que varían mucho mas de lo que esperábamos. Esto significa que con estos datos no podríamos todavía afirmar nuestra hipótesis. Por ello, decidimos utilizar K-fold cross validation, con $K = 5$, y ver desde 20 componentes princiaples hsta 60, para tener una mejor visión mas detallada ya que cada punto sintetiza 5 evaluaciones del algoritmo.



(a) los 6 datasets mas grandes

(b) los 3 datasets mas grandes

Figura 8

Este gráfico todavía no muestra una tendencia hacia un número fijo de componentes principales, por lo cual no podemos afirmar que las componentes principales dependen en mayor medida de las clases que del dataset. Como las componentes principales se consiguen del dataset, si bien describen las clases, las variaciones del dataset todavía influyen fuertemente en ellas. Sin embargo, con la información que tenemos, sí podemos afirmar que existe un rango que contienen las componentes principales mas relevantes para estas clases y ese rango es de 30 a 40.

Por último, queremos ver qué tan cercana resulta nuestra cantidad de componentes principales obtenida experimentalmente con una medida teórica de esta cantidad.

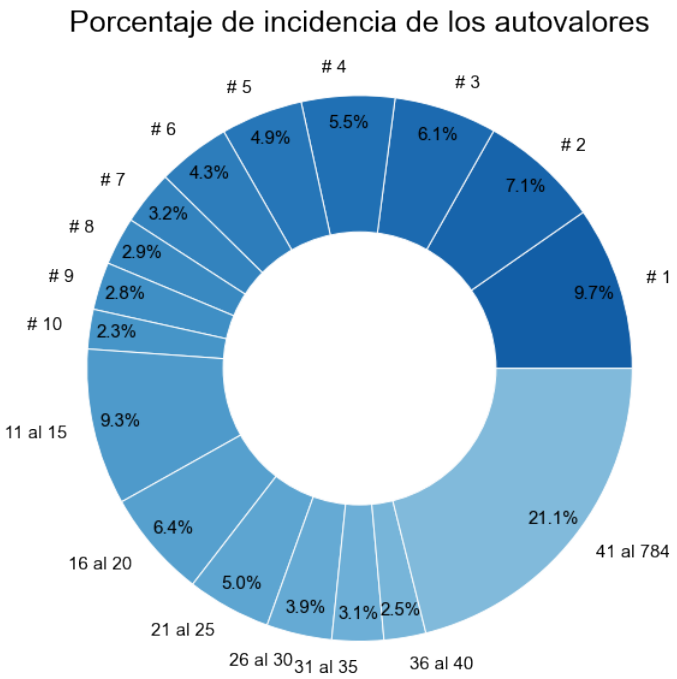


Figura 9: Porcentaje de información captada por componente principal

Al ver el gráfico de porcentaje de la información contenida en cada componente principal, nos encontramos con una revelación sorprendente. Creímos originalmente que nuestro punto óptimo de dimensiones iba a contener un porcentaje más alto de la información total (mayor al 90 %) antes de que el factor de ruido comenzara a tener un efecto mayor que la incorporación de nueva información, pero esto no fue así.

Es más, con sólo un 80 % de la información original, usando PCA obtenemos un *accuracy* ligeramente mejor a la clasificación sin usar esta herramienta, lo cual indica que la dimensionalidad afecta muy fuertemente al poder de predicción de kNN. Al mismo tiempo, ganamos significativamente en reducción de tiempo de cómputo al comparar en hasta 40 dimeniones, en vez de 784.

3.3. Ajustes finales de experimentación

A esta altura ya contamos con un clasificador satisfactorio, pero como en pasos anteriores, nos interesa verificar que los parámetros encontrados escalan bien a muestras generales y no al conjunto específico de datos actuales. Si bien contamos con una base de datos suficientemente grande como para reducir las posibilidades de que este factor sea considerable, lo verificaremos utilizando K-Fold Cross Validation.

Para ello, tomaremos PCA con un valor hallado dentro del rango de optimización, por ejemplo 34. Y utilizaremos kNN 3 para después comparar con las cuentas realizadas en la sección 3.1. Realizaremos 5 folds y comparemos sus resultados con kNN sin PCA.

Métrica	Fold1	Fold2	Fold3	Fold4	Fold5	Average
accuracy	0.974	0.977	0.968	0.971	0.974	0.973
precision	0.974	0.977	0.968	0.971	0.974	0.973
recall	0.974	0.977	0.968	0.971	0.973	0.972

(a) kNN con PCA

Métrica	Fold1	Fold2	Fold3	Fold4	Fold5	Average
accuracy	0.970	0.967	0.962	0.963	0.969	0.966
precision	0.970	0.967	0.963	0.964	0.970	0.967
recall	0.969	0.966	0.962	0.963	0.968	0.966

(a) kNN sin PCA

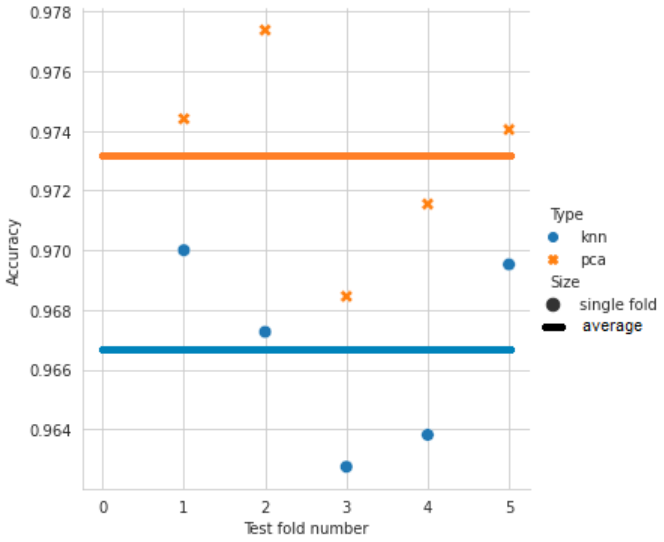
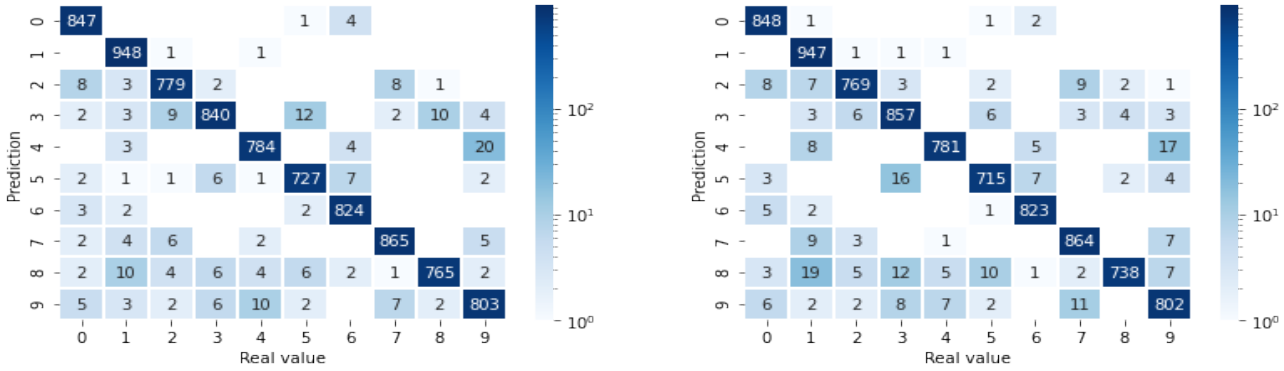


Figura 10: Accuracies de kNN y kNN+PCA con kfold

La primer comparación que se nos hizo notable son los tiempos de ejecución. Sin PCA, el K-fold con kNN 3 tardo 16m 52s en finalizar el entrenamiento y la predicción. Como el tiempo de entrenamiento para kNN sin PCA es despreciable (del orden de los 300ms), podemos suponer que por iteración tardo alrededor de 3m 20s. Mientras que al aplicar PCA el K-fold tardo 4m 21s. Como el entrenamiento tarda 10s aproximadamente para 34 componentes, podemos suponer que por iteración tardo 42s en predecir. Esta diferencia en tiempo de cómputo cobra mayor relevancia al incrementar los folds, la cantidad de muestras y la cantidad de dimensiones de los datos.

La segunda comparación que se nos hizo notable es el hecho de que mas allá del tiempo de ejecución, la mejora de rendimiento no fue substancial. El *accuracy* promedio con PCA es solo un 0.7 % mejor que sin PCA. Pero mirando la diferencia en términos de los dígitos errados, aplicando PCA obtenemos 20 % menos de errores con respecto a no usarlo.



(a) kNN 3 con PCA 34. Accuracy promedio = 0.973

(b) kNN 3 sin PCA. Accuracy promedio = 0.966

Figura 11: Matrices de confusión, Fold 5 como test

Comparando entre ambas matrices de confusión, podemos ver que con PCA, si bien los errores son menores en módulo, no logramos revertir los patrones que llevaban a estos errores. En otras palabras, los errores sistémicos que cometía kNN, (explicitados en la sección 3.1) los sigue cometiendo para la gran mayoría de los casos. Creemos que se podría haber logrado una diferencia más significativa de *accuracy* implementando alguna de las mejoras propuestas a kNN para resolver mejor los empates.

4. Conclusiones

Concluyendo el trabajo, realizamos una predicción sobre los datos de Kaggle utilizando nuestro mejores parámetros obtenidos en la experimentación:

1. kNN de 3
2. PCA de 34

y nuestros resultados fueron los siguientes:

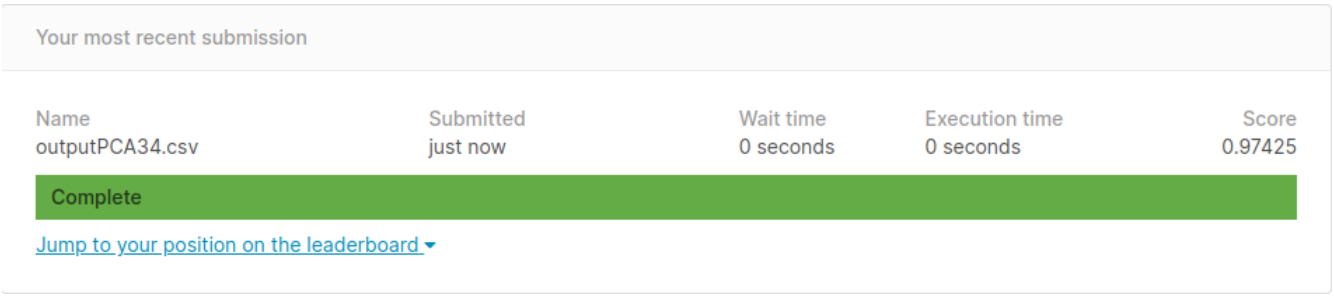


Figura 12: Kaggle: kNN 3 con PCA 34. Accuracy : 0.97425

Si bien quedamos en la posición 1473 de la competencia de Kaggle:

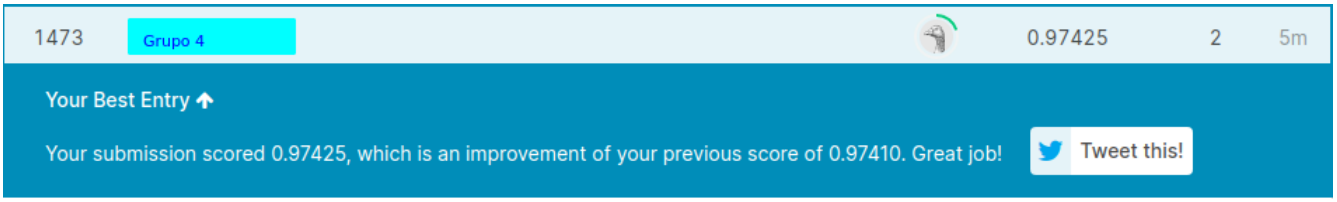


Figura 13: La posición del grupo 4

Estamos muy satisfechos por haber superado a Fede Pousa.

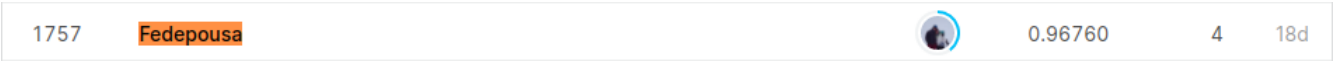


Figura 14: La posición de Fede Pousa

¿Qué nos llevamos de este trabajo? En primer lugar vimos que el Machine Learning, por más susto que genere el nombre en sí mismo, no sólo es mas sencillo de lo que nos esperábamos, si no que también fue un claro y divertido ejemplo del mundo real donde se aplican los métodos numéricos que estudiamos en clase.

Lo visto en este trabajo no sólo es relevante para el reconocimiento de números, utilizando el dataset de letras de EMNIST[4] y realizando un único cambio en el código de kNN para que pueda tomar mayor cantidad de clases, pudimos utilizar el mismo algoritmo para la predicción de letras. Simplemente reutilizando kNN con $K = 3$, obtuvimos un 0.847 de *accuracy*, un número sorprendentemente alto, que por estar las clases balanceadas sabemos que es confiable (ver en el Jupyter Notebook `letter experimentation`).

Esto nos da confianza en que la herramienta que construimos en este trabajo la podemos definir sin tantos cambios como un predictor generalizado de imágenes en blanco y negro. Y repitiendo los procedimientos de experimentación, se puede llegar a sus parámetros óptimos.

[4]disponible en https://www.kaggle.com/crawford/emnist

5. Apéndice

```
pair<Vector, Matrix> PCA::fit(Matrix images_to_fit) {
  Matrix X(images_to_fit);
  for (int i = 0; i < images_to_fit.cols(); i++) {
    double mean = images_to_fit.col(i).mean();
    Vector means(images_to_fit.rows());
    for (int j = 0; j < images_to_fit.rows(); j++) {
      means(j) = mean;
    }
    X.col(i) = X.col(i) - means;
  }

  Matrix covariances = pow(images_to_fit.cols() - 1, -1) * X.transpose() * X;
  return get_first_eigenvalues(covariances, selected_components, 10000, 1e-16);
}
```

Listing 1: PCA modificado

Este fragmento de código es una modificación de `pca.cpp` para obtener la Figura 9. Lo dejamos escrito para implementar en caso de desear reproducir el mismo.

Referencias

[1] Joel Grus. “Data Science from scratch”. En: O’Reilly, 2019. Cap. 12 - K Nearby Neighbours.

[2] Penn State University. Applied Multivariate Statistical Analysis. URL: https://online.stat.psu.edu/stat505 (accessed: 01.11.2020).