DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# TP 2: Reconocimiento de dígitos

7 de abril de 2022 — Métodos numéricos


| Integrante | LU | Correo electrónico |
|---|---|---|
| Chami, Uriel Alberto | 157/17 | uriel.chami@gmail.com |
| Oliveira Gariboglio, Matias Nahuel | 392/19 | matiasnoliveira@gmail.com |
| Ciruzzi, Ramiro Augusto | 228/17 | ramiro.ciruzzi@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

# 1. Introducción

El objetivo de este trabajo práctico es desarrollar y evaluar un clasificador para el reconocimiento de dígitos manuscritos en imágenes. Nos limitaremos a reconocer los dígitos entre el 0 y el 9.

La idea de este algoritmo de clasificación es utilizar la información de una base de caracteres **ya etiquetada** para '*entrenarse*' y así después, dada una nueva instancia de esos caracteres no presente en la base, poder determinar que etiqueta le corresponde. Nos referiremos a esa base de datos etiquetada como al conjunto de entrenamiento (train) y a aquellos datos no presentes en la misma como al conjunto de prueba (test).

La base sobre la que implementaremos nuestras pruebas es MNIST, en la versión utilizada en *Kaggle* [1]. La misma consta un conjunto de entrenamiento de 42000 dígitos (entre 0 y 9) y uno de prueba de 18000. Donde cada dígito es una imagen en escala de grises de 28x28.

Para clasificar a las imágenes de dígitos no catalogadas usaremos el método de los k vecinos más cercanos, conocido como kNN por su sigla en inglés (k-nearest neighbors).

# 2. Método kNN

La idea general del algoritmo de k vecinos más cercanos es considerar a cada imagen como un vector $x_i \in R^m$ $\forall i \in \{1, .., n\}$ , donde $m = 28*28 = 784$ en nuestro caso. En particular para las imágenes que pertenecen a la base de entrenamiento se conoce a qué clase corresponden. De esta forma frente a la llegada de una nueva imagen $z$, también expresada como un punto en el espacio euclídeo m-dimensional, se recorre el conjunto de entrenamiento buscando aquellas k imágenes que minimicen

$$\arg\min_{i=1,\dots,n} \|z - x_i\|_2.$$

En otras palabras, como lo infiere el nombre del método, se buscan las k imágenes más cercanas a $z$. Luego con ese subconjunto de los k vecinos más cercanos se realiza una votación y se elige como etiqueta de $z$ a la clase o label que posea el mayor número de repeticiones dentro del mismo.

En el algoritmo 1 se puede ver como kNN realiza el 'entrenamiento' que básicamente consiste en pasarle una base de muestras con labels ya definidos. Por otro lado la predicción de kNN como se observa en el algoritmo 2 y como dijimos anteriormente, se fija en las k muestras más cercanas y entre las mismas realiza una votación quedándose con el label que posea más votos.

Podemos observar como puntos fuertes de kNN que es un algoritmo simple y que pareciera funcionar eficientemente para imágenes de dimensiones bajas. El problema reside en que aumentar las dimensiones de estas muestras, como veremos más adelante en la experimentación, significa un incremento considerable del tiempo de ejecución. Es por esto que a la hora de utilizar este método se suele *preprocesar* el conjunto de imágenes de la base de datos para tener una mejor eficacia en términos de tiempo. Teniendo esto en cuenta, a lo largo de este informe evaluaremos las ventajas de *preprocesar* el conjunto de muestras con un método reducción de dimensionalidad llamado *Análisis de componentes principales*, conocido como PCA por su sigla en inglés (Principal component analysis).

**Algorithm 1** Algoritmo para entrenar al clasificador kNN

```
1: function kNN_fit(Matrix X,Matrix y)
2:     matrix_train ← copia de X
3:     matrix_train_labels ← copia de y
```

# 3. Método PCA

Además de reducir la cantidad de dimensiones de las muestras en pos de tener un menor número de variables en juego, el método PCA busca disminuir la redundancia de dichas variables, es decir, la covarianza que tienen

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

**Algorithm 2** Algoritmo de predicción con el clasificador kNN

```
1:  function kNN_predict(Matrix X)
2:      ret ← Vector(tamaño : #filasDeX)
3:      for k ← 0 to #filasDeX do
4:          train ← matrix_train
5:          distances ← Vector <tupla <enteros , reales>>
6:          for fila en filasDetrain do
7:              fila ← fila − k-ésimaFilaDeX
8:          for i ← 0 to #filasDetrain do
9:              agregarAtrás(distances, tupla <i , ‖i-ésimaFilaDetrain‖>)
10:         distances ← ordenarAscPor2daCompTupla(distances)
11:         votos ← diccionario<entero, entero>
12:         for i ← 0 to k vecinos pedidos do
13:             label ← clase del vecino i
14:             if label ∈ votos then
15:                 significado(votos, label)++
16:             else
17:                 significado(votos, label) = 1
18:         maxVoto ← (0, 0)
19:         for voto en votos do
20:             if voto_2 >maxVoto_2 then
21:                 maxVoto ← voto
22:         ret[k] ← maxVoto_1
        return ret
```

entre sí. A estas nuevas muestras con variables con información más representativa las denomina **componentes principales**. Para lograr esto realiza una transformación de los datos a otra base:

Sea entonces $X' \in R^{nxm}$ (con $m$=784 en nuestro caso) la matriz que representa nuestra base de datos, con sus filas las $n$ muestras de la base. Antes de hacer nada, centraremos los datos en $X'$, es decir en cada columna tomaremos el promedio (es decir el valor promedio de ese píxel para estos datos) y se lo restaremos a toda esa columna, llamaremos a esta matriz con todos sus pixels centrados alrededor del valor promedio $X$ y trabajaremos de ahora en adelante con ella.
Lo que queremos es encontrar una matriz $P$ que al multiplicarla por $X$ nos lleve a otro espacio con mejores propiedades que el original $\hat{X}$ en donde las variables de las muestras no estén correlacionadas:

$$\hat{X} = XP$$

Para esto calculamos la matriz de covarianza de $X$ que nos va a dar algo de la pinta:

$$M_X = \begin{bmatrix} \sigma^2_{x_1} & \sigma_{x_1x_2} & \sigma_{x_1x_3} & . & . & \sigma_{x_1x_n} \\ \sigma_{x_1x_2} & \sigma^2_{x_2} & \sigma_{x_2x_3} & . & . & \sigma_{x_2x_n} \\ \sigma_{x_1x_3} & \sigma_{x_2x_3} & \sigma^2_{x_3} & . & . & \sigma_{x_3x_n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ \sigma_{x_1x_n} & \sigma_{x_2x_n} & \sigma_{x_3x_n} & . & . & \sigma^2_{x_n} \end{bmatrix}$$

Donde $\sigma_{x_ix_j}$ representa la covarianza entre las variables $x_i$ y $x_j$ y $\sigma^2_{x_i}$ representa la varianza de la variable $x_i$. Entonces lo que nosotros queremos es transformar esta matriz en una matriz diagonal ya que eso implicaría que todas las covarianzas sean 0 y por ende que no haya correlación entre variables distintas. Para conseguir esto debemos encontrar los autovalores y autovectores respectivos de $M_X$. Un dato no menor es que sabemos que tiene una base de autovectores por ser $M_X$ simétrica.

Utilizaremos el método de la potencia para el cálculo de esos autovalores y autovectores, después veremos porque específicamente este. Este método obtiene el autovalor más grande en módulo y su respectivo autovector; lo

repetiremos usando un método de deflación que descarte los ya obtenidos para encontrar de esa manera todos los autovalores y autovectores. Suponiendo entonces que la matriz $V$ tiene en sus filas todos los autovectores de $M_X$ tenemos una matriz diagonal $D$:

$$D = VM_XV^t$$

Pero sabiendo que la matriz de covarianza de una matriz se calcula:

$$M_X = \frac{1}{n-1}X^TX$$

Tenemos entonces que:

$$\begin{aligned} VM_XV^t &= V\frac{1}{n-1}X^TXV^t \\ &= \frac{1}{n-1}(VX^t)(XV^t) \end{aligned}$$

Pero si elegimos ese $V^T$ como nuestra matriz $P$ inicial que cambiaba de base a $X$ a un espacio con mejores propiedades que el original obtenemos:

$$\begin{aligned} VM_XV^t &= \frac{1}{n-1}\hat{X}^T\hat{X} \\ VM_XV^t &= M_{\hat{X}} \end{aligned}$$

Entonces nos queda la matriz de covarianza de $\hat{X}$ como una matriz diagonal, es decir, variables distintas no se correlacionan entre si. Esto era lo que buscábamos por ende la matriz $P$ que me cambia de base a $X$ a ese espacio ideal es aquella que tiene los autovectores de $M_X$ como columnas.

Ahora bien, si recordamos como obteníamos la matriz $P$ podemos observar que los autovectores de $M_X$ que se encuentran en las columnas de $P$ están ordenados de tal forma que el primero de ellos está asociado al autovalor de módulo más grande de $M_X$ el segundo al segundo más grande y así siguiendo. Esto se debe justamente a que calculamos los mismos con el método de la potencia.

De que nos sirve esto? Si nos fijamos como estaba compuesta la matriz de covarianza del nuevo espacio ideal $M_{\hat{X}}$ podemos advertir que es una matriz diagonal que tiene en la misma los autovalores de $M_X$ ordenados de mayor a menor (en módulo). Pero las matrices de covarianza en su diagonal tienen la varianza de la componente $x_i$, entonces siendo el primer autovalor el más grande en módulo sabemos que la varianza de $x_1$ es la mayor de todas la de $x_2$ la segunda mayor y así.

De esta forma al multiplicar nuestra matriz de muestras iniciales $X$ por $P$, el producto entre las filas de $X$ y la primer columna de $P$ nos va a dar variables lo más distintas posibles pues la primer columna de $P$ es el autovector asociado al autovalor más grande en módulo, es decir, la máxima varianza posible. Con la segunda columna de $P$ vamos a obtener variables bastante distintas también pero menos que con la primera, y lo mismo con las subsiguientes columnas.

Esto nos lleva a intuir que a partir de cierto número de columnas de $P$ las variables resultantes de ese producto van a empezar a aportar cada vez información menos representativa porque los autovectores van a estar asociados a varianzas más chicas, esto es el algo que analizaremos en la experimentación.

En resumen, como vemos en los algoritmos 3 y 4 para hacer la transformación de PCA agarramos la matriz de muestras calculamos su matriz de covarianza, luego los autovectores de la misma y con esos autovectores en columnas multiplicamos a la matriz inicial para obtener la nueva matriz de muestras con variables más representativas.

Luego tenemos los algoritmos 5 y 6 que usamos para calcular los autovalores y autovectores de la matriz $M_X$.

**Algorithm 3** Algoritmo para la obtención de los autovectores de la matrix de covarianza

```
1: function eigenvecs_Mx(Matrix X)
2:     x_train ← X
3:     divisor ← √(#filasDex_train − 1)
4:     v ← Vector <reales>
5:     for columna en columnasDex_train do
6:         agregarAtrás(v, promedio(columna))
7:     for fila en filasDex_train do
8:         fila ← fila − v
9:     x_train ← x_train / divisor
10:    M_x ← x_train^T * x_train
11:    res ← get_first_eigenvalues(M_x, components_a_reducir, 10000, 10e^−6)
12:    transformation ← res_2
```

**Algorithm 4** Algoritmo de transformación característica de PCA

```
1: function tc(Matrix X)
       return X * transformation
```

**Algorithm 5** Algoritmo del método de la potencia

```
1: function power_iteration(Matrix X, int num_iter, double eps)
2:     b ← VectorRandom(tamaño : #columnasDeX)
3:     for i ← 0 to num_iter do
4:         producto ← X * b
5:         nuevoB ← (producto / ‖producto‖)
6:         if ‖nuevoB − b‖ < eps^3 then
7:             b ← nuevoB
8:             break
9:         b ← nuevoB
10:    eigenvalue ← b^T * X * b / b^T * b
       return (eigenvalue , b / ‖b‖)
```

**Algorithm 6** Algoritmo para la obtención de los autovalores más significativos de una matriz y sus respectivos autovectores

```
1: function get_first_eigenvalues(Matrix X,int num, int num_iter, double epsilon)
2:     A ← X
3:     eigvalues ← Vector(tamaño : num)
4:     eigvectors Matrix(tamaño : #filasDeA x num)
5:     for i ← 0 to num do
6:         res ← power_iteration(A, num_iter, epsilon)
7:         lamda_i ← res_1
8:         v_i ← res_2
9:         eigvalues[i] ← lamda_i
10:        eigvectors.columna(i) ← v_i
11:        A ← A - (lamda_i * v_i * v_i^T)
       return (eigvalues , eigvectors)
```

# 4. Experimentación

## 4.1. Aclaraciones

Todos los experimentos que se detallan a continuación fueron ejecutados en una computadora con Ubuntu 18 con 11.6 GB de memoria RAM, un procesador Intel® Core™ i7-7500U CPU @ 2.70GHz × 4 y un disco SSD de 235,2 GB. Se ejecutaron con la base de datos MNIST completa, dividiendo (cuando no se aclare lo contrario) 80 % para *train* y 20 % para *test*.
Durante todo el desarrollo nos referiremos a:

- $k$: La cantidad de vecinos utilizados en el método $kNN$
- $\alpha$: La cantidad de componentes principales del método $PCA$
- $K$: La cantidad de folds del método de test *K-fold cross validation*.
- $n$: la cantidad de elementos en el conjunto de elementos taggeados (para MNIST $n = 42k$ )
- $\#train$: la cantidad de elementos en la subdivision del conjunto de elementos taggeados que usaremos para entrenar ( $\#train < n$ )
- $\#test$: la cantidad de elementos en la subdivision del conjunto de elementos taggeados que usaremos para testear ( $\#test < n$ )

## 4.2. kNN + PCA con $\alpha$ fijo y $k$ variable

Buscamos la cantidad de vecinos del método de $kNN$ que obtenga el mejor resultado en términos de accuracy. Habiendo ejecutado algunos casos del experimento opuesto (kNN + PCA con $\alpha$ variable y $k$ fijo) sabemos que $\alpha$ obtiene un accuracy cercano a óptimo para valores de entre 30 y 40. Por eso usaremos $\alpha = 30$ en este experimento (en experimentos posteriores afinaremos aún más el criterio), de esta manera trataremos de eliminar de la ecuación ésta variable y ver cuál es el $k$ que mejor performa.

Intuitivamente sabemos que $k$ no puede ser muy grande (proporcionalmente a la muestra). Para ejemplificar el por qué de esta afirmación observemos el caso de $k$ máximo. El máximo de vecinos posibles a un elemento es $\#train$, todo el conjunto de elementos que conocemos.
¿Cuáles son los $\#train$ vecinos más cercanos a una imagen? Son todos pues el conjunto tiene $\#train$ elementos. Eso significa que al tomar la moda **dará el mismo resultado para cualquier imagen**, esto no solo es pésimo en términos de accuracy, es mucho peor en términos de recall y precision para todas las clases, sobre todo las que no son iguales a la moda del conjunto de train.

En el caso más general, lo que queremos priorizar son aquellos elementos **cercanos** a la imagen, y nuestro método de selección le da el mismo peso al elemento mas cercano que al $k$-ésimo más cercano ya que toma la *moda* del conjunto. En otras palabras, **cuanto más grande es el $k$, menos importan los elementos más cercanos**.

Dicho esto, nuestros $k$'s del experimento serán: $[1, 2, 3, ..., 30, 40, 50, ..., 100]$ poniendo la granularidad en los valores pequeños de $k$ ya que sabemos que allí es donde habrá una diferencia más notoria entre uno y el otro.

### 4.2.1. Hipótesis

Tomando $k = 1$ creemos que se trata de una forma no muy inteligente de elegir el número asociado a la imagen, de la misma forma tomando $k = n$, sabemos empíricamente que no será bueno. Lo que creemos sucederá es que se obtendrá un resultado decente con $k = 1$ que mejorará al hacer crecer $k$ hasta cierto punto y luego decrecerá el accuracy a medida que nos acercamos a valores 'grandes'. Recordemos que de todos modos 100 es bastante poco respecto de $n$.

### 4.2.2. Análisis de resultados

Podemos ver que la figura 3 nos confirma nuestras hipótesis, para k=6 obtuvimos un sorprendente 97,4 % de accuracy. Además dicho acccuracy fue bastante sostenido para los distintos $k$'s cercanos a 6, obteniendo 97 % de accuracy promedio para $k$ entre 2 y 16. Esto nos permite visualizar **cuán pequeña** debe ser la vecindad para este

problema. En la cola del grafico, para valores más grandes efectivamente notamos una caída de la performance, esto está también alineado con lo que predijimos.

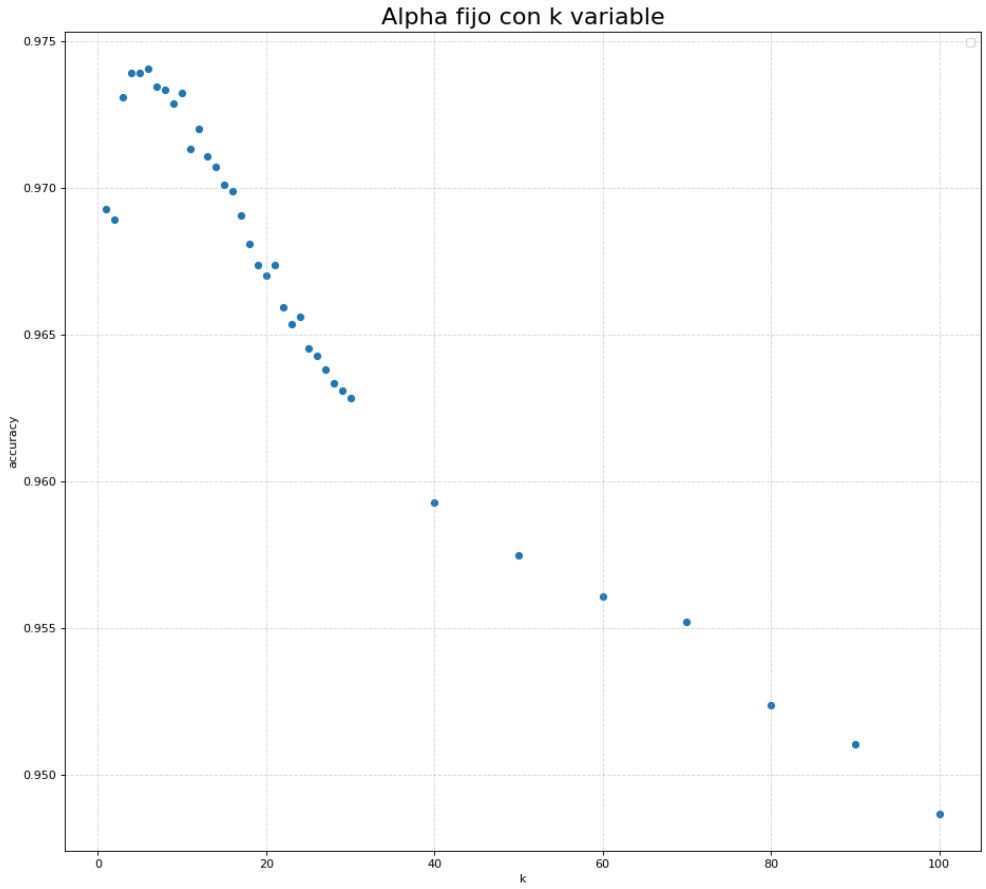


Figura 1: $\alpha$ fijo y k variable

### 4.2.3. Complejidad

El algoritmo para la búsqueda de la vecindad de tamaño $k$ depende (obviamente) de $k$, pero también depende fuertemente del tamaño de la muestra. Como ya vimos, no tiene mucho sentido utilizar vecindades muy grandes con lo cual el $k$ será un factor chico de la complejidad, en otras palabras, sin importar si $k = 1$ o $k = 100$, el algoritmo requerirá un orden de magnitud similar de tiempo. Eso, precisamente, es lo que podemos observar en la figura 2.

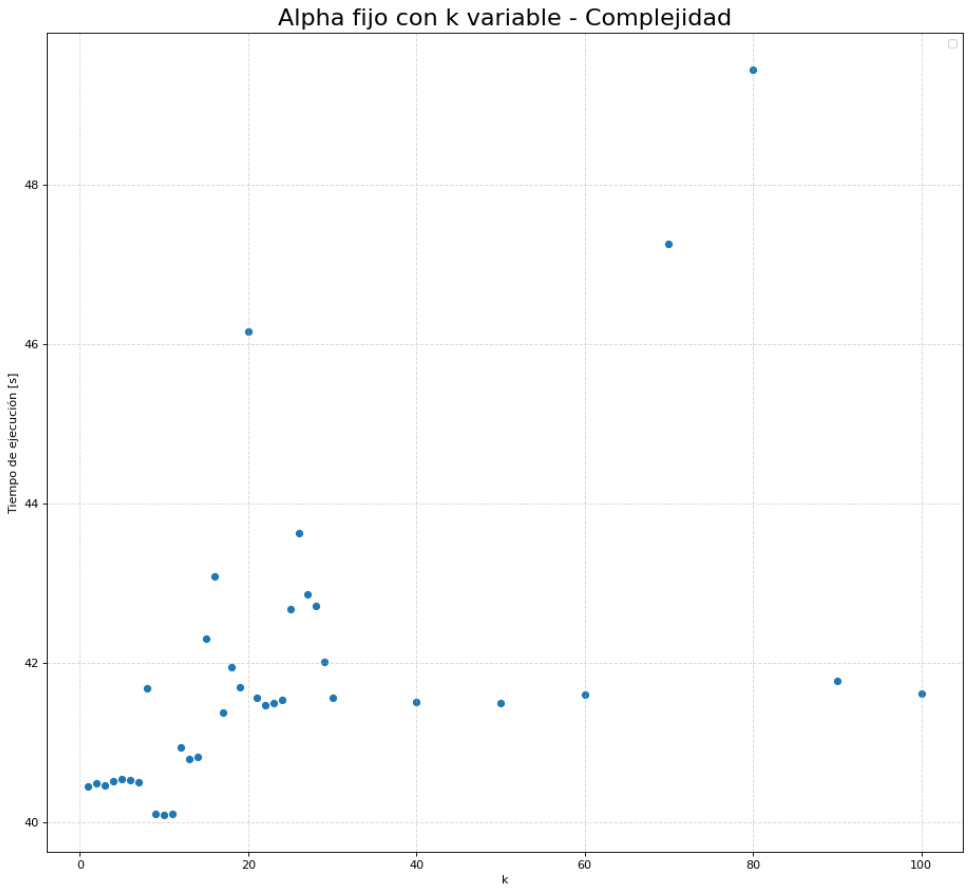


Figura 2: $\alpha = 30$ y k variable

### 4.3. kNN + PCA con $\alpha$ variable y $k$ fijo

Similar al experimento anterior, buscamos el $\alpha$ que maximice el accuracy fijando el $k$ en un valor cercano a óptimo, en este caso elegimos $k = 8$.
En este caso el tiempo que toma el método de la potencia que nos provee de los autovectores de la matriz de covarianzas está relacionado linealmente a la cantidad de autovectores que deseamos obtener, ya que por cada autovector se debe ejecutar el mismo algoritmo con la matriz "desinflada", es decir, cambiandole un autovalor (el recien obtenido) por 0. Nuestro método de deflación en particular no reduce la dimensión de la matriz al desinflarla, todos los autovectores se calculan contra una matriz de $784 \times 784$.

Es por esto que querremos ahorrar ejecuciones del método de la potencia lo más posible. Incentivados por la mejora en eficiencia que implica, este experimento hace uso del hecho de que los autovectores son acumulativos. Es decir, que si $\alpha = 10$ calcularemos los primeros 10 autovectores, pero luego si quisiesemos calcular la transformación para $\alpha = 7$ basta reutilizar la transformación y recortarla. Más aún, basta con recortar el resultado transformado. Simbólicamente: sea $X \in \mathbb{R}^{n \times 784}$ la matriz de imágenes y $P^T \in \mathbb{R}^{784 \times 784}$ la transformación al espacio latente de $X$.

$$XP^T = X \begin{pmatrix} | & | & | & | & | \\ p_1 & p_2 & . & . & p_{784} \\ | & | & | & | & | \end{pmatrix} = \begin{pmatrix} | & | & | & | & | \\ Xp_1 & Xp_2 & . & . & Xp_{784} \\ | & | & | & | & | \end{pmatrix}$$

Luego si en vez de querer 784 componentes se quieren $l$, basta extraer de la matriz $XP^T$ las primeras $l$ columnas.

De esta manera trabajaremos con $\alpha = [1, 2, ..., 40, 50, 60, ..., 150]$ y sin embargo solo calcularemos una vez la transformación de PCA para $max(\alpha)$ que en este caso es 150.

#### 4.3.1. Hipótesis

Por un lado los $\alpha$'s más chicos obtendrán un peor accuracy ya que cuantas menos componentes se incluyan, más información se está ignorando. Por el otro, para $\alpha$'s muy grandes la información es demasiada y se perderá eficiencia. Analicemos en más profundidad esto último, recordemos que luego de transformar con PCA las imagenes, se busca la 8-vecindad más cercana de cada imagen donde la distancia utilizada para decidir esa cercanía es la norma 2.
Dicha norma le da la misma importancia a todas las coordenadas, no obstante, nosotros sabemos que las primeras componentes son las que más varianza explican y en consecuencia las más importantes para diferenciar a dos imágenes. Es entonces que se puede dar el caso de obtener vecinos muy cercanos que se parecen mucho en las últimas componentes pero distan mucho en las primeras, empeorando el accuracy final.
Lo interesante de este experimento será por un lado confirmar cuán real es el efecto comentado y por el otro dónde se encuentra el punto intermedio en el que se explica la mayoría de la varianza con la menor cantidad de autovectores.

#### 4.3.2. Análisis de resultados

Como podemos ver en la figura 3 alrededor de $\alpha = 30, .., 40$ obtuvimos los mejores accuracies, alrededor del 97 %, luego, aunque poco notoria en el grafico, se aprecia una caída de la eficacia como predijimos. Sin embargo el efecto es mucho más marcado al ignorar información que al incorporar información de sobra. Por limitantes de tiempo no calculamos los 784 autovectores pero creemos que se podría apreciar una baja más marcada si lo hiciéramos.

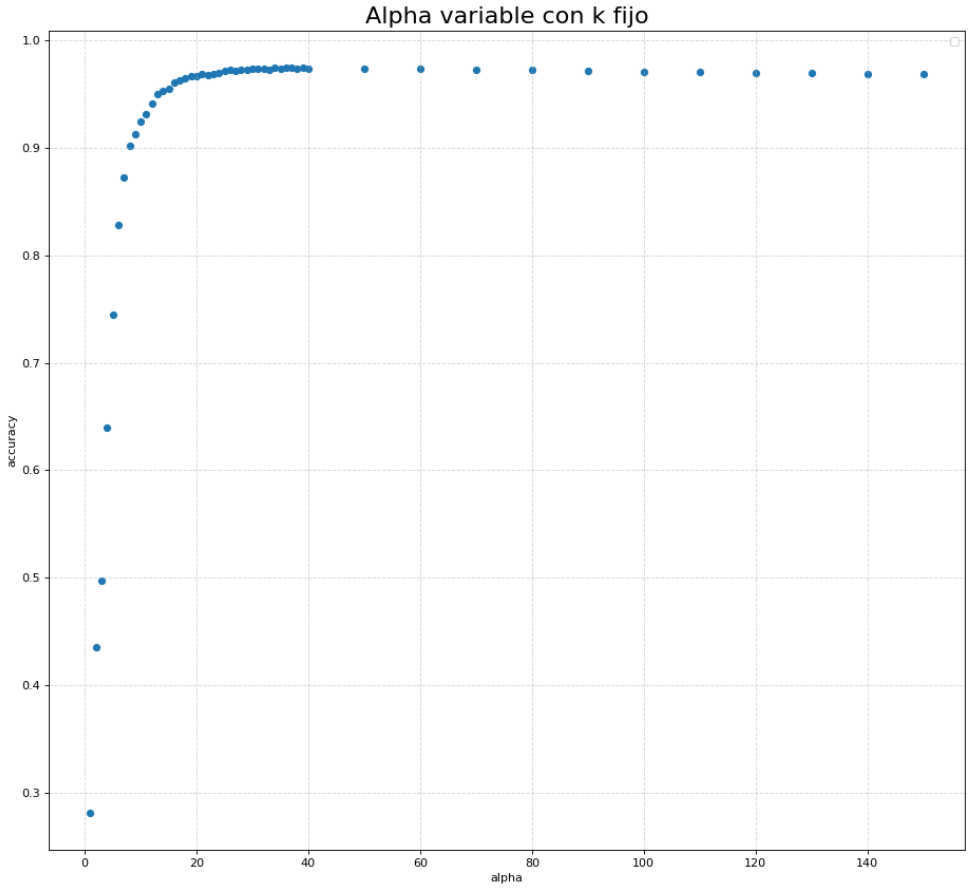


Figura 3: $\alpha$ variable y k = 8

## 4.4. $k$ y $\alpha$ variables

Ahora que tenemos un rango más acotado de posibilidades nos interesa probar todas las combinaciones posibles de parámetros dentro de los rangos que ya vimos óptimos en los análisis anteriores. Estos rangos son $k = [7, 8, .., 15]$ y $\alpha = [30, 31, .., 40]$, sabemos que dejando fijo uno y moviendo el otro quizás estamos dejando afuera alguna combinación óptima, por eso decidimos variar ambos.

### 4.4.1. Análisis de resultados

Podemos ver en la figura 4 que en todos los casos obtuvimos muy buenos accuracies que no bajan del 95 %. El valor más alto obtenido fue $k = 7$ y $\alpha = 35$, estos son los valores que usaremos de aquí en adelante.

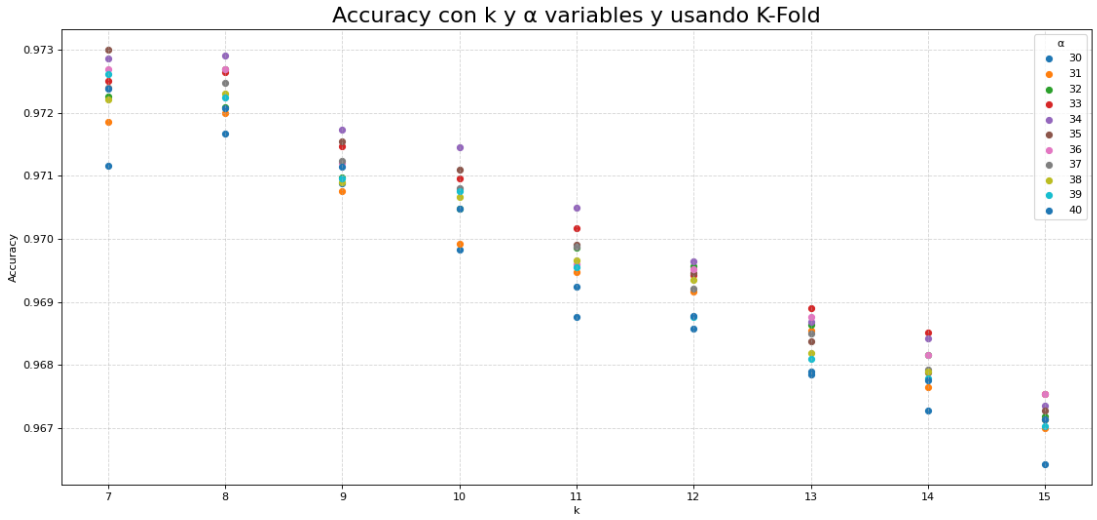


Figura 4: $\alpha$ variable y k = 8

#### 4.4.2. ¿Es realmente bueno? - Precision y recall

En términos de accuracy el método kNN + PCA con $k = 7$ y $\alpha = 35$ mostró ser óptimo pero: ¿Es realmente el caso? ¿Es acaso el accuracy la única medida de interés? Claramente la respuesta es que no. Para nuestra base de datos particular (MNIST) las preocupaciones sobre cuán buena es la medida de accuracy no son tantas ya que la muestra está sumamente balanceada, es decir que hay una cantidad parecida de elementos de cada clase y el tamaño del conjunto de train es muy apropiado respecto de la cantidad de clases.

Algunas medidas que podrían resultar de interés en vez de la unificación de todo en un único valor de accuracy son el precision y el recall.

**Precision** Se define Precision como los aciertos relativos dentro de una clase. Dada una clase $i$:

$$precision_i = \frac{verdaderosPositivos_i}{verdaderosPositivos_i + falsosPositivos_i}.$$

Intuitivamente el precision nos expresa cuán correcta es nuestra clasificación de esa clase, es decir, de todas las imágenes que dijimos que representaban un 8 cuántas realmente lo eran.

**Recall** Se define Recall como una métrica para medir los reconocimientos dentro de una clase. Dada una clase $i$:

$$recall_i = \frac{verdaderosPositivos_i}{verdaderosPositivos_i + falsosNegativos_i}.$$

Esta métrica nos permite comprender de todos los elementos que pertenecían a una clase, cuántos realmente los clasificamos en esa clase.

**Heatmap** El siguiente gráfico nos permite observar (implícitamente) tanto precision como recall para un test en particular. El gráfico consta de dos ejes (x e y) ambos con puntos del 0 al 9, el eje y representa el valor real y el eje x la predicción. Por ejemplo en el lugar (3,5) de esta matriz tenemos el porcentaje de elementos que son 3's pero que fueron taggeados por nuestro predictor como 5's en este caso un 1 % (**Aclaración:** para que los números entren en sus respectivos cuadrados fueron redondeados a 2 decimales).
Observando la figura 5 podemos deducir el $precision_i$ si tomamos la **fila** $i$ del heatmap y hacemos $\frac{elemento_{ii}}{elemento_{ij} \forall j}$, también podemos deducir el $recall_i$ mirando la **columna** $j$ del heatmap y calculando $\frac{elemento_{jj}}{elemento_{ij} \forall i}$.
Sin importar con qué métrica se lo mire, el método kNN + PCA con $k = 7$ y $\alpha = 35$ mostró ser muy efectivo.

Estas métricas nos dan otro tipo de seguridad respecto a nuestra respuesta, pero en el fondo eran resultados que no sorprenden, no por el alto accuracy si no por la muestra con la que se trabaja. En MNIST un alto accuracy implica necesariamente buen precision y recall.

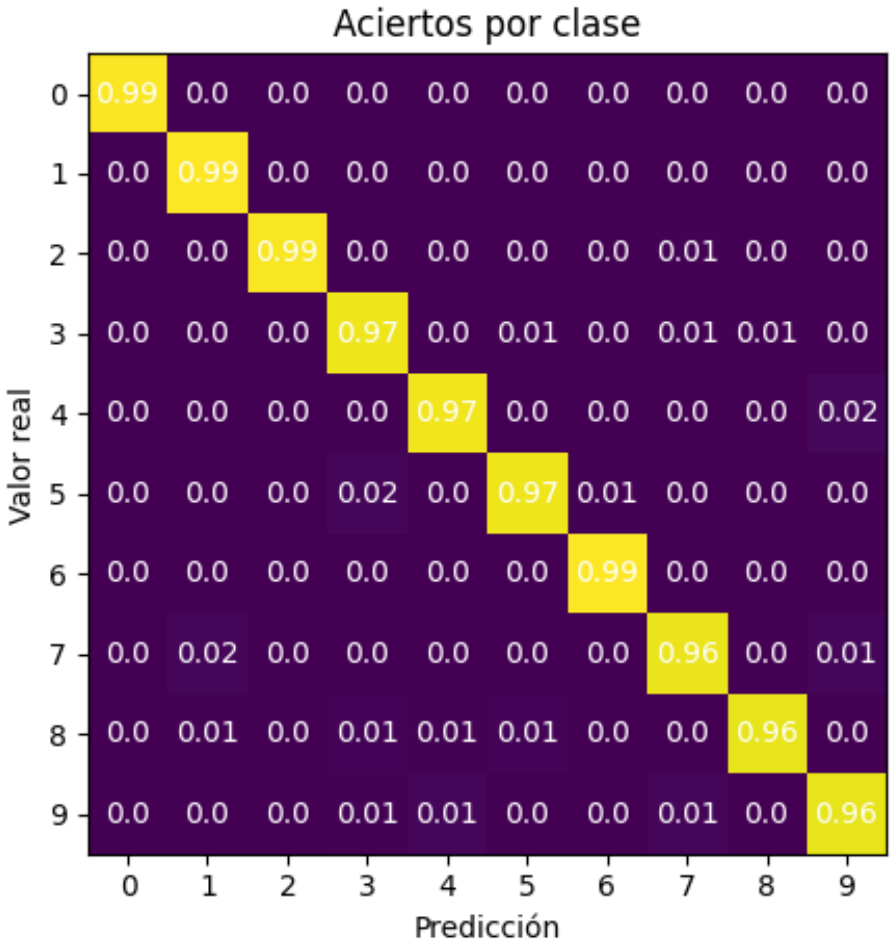


Figura 5: Heatmap para $k = 7$ y $\alpha = 35$

## 4.5. K-fold

Para tener una mejor medicion de los resultados se implementará una validación cruzada, en particular el método de K-fold que consiste en subdividir el conjunto de train en K partes iguales, de las cuales se usará una parte como test y el resto como train, realizando las K combinaciones posibles y obteniendo el promedio de los accuracy de cada uno se tiene el accuracy final.

### 4.5.1. ¿Cómo elegir K?

Si bien vemos que a medida que aumenta el K el accuracy también, esto no tiene relación real con la calidad del clasificador ya que en todas nuestras pruebas no se efectuó ningún cambio en los parámetros de ejecución de las mismas, sino que solo cambió el *#train* y *#test*, pudiéndose deducir que el aumento de accuracy se debe al aumento del *#train* y la disminución del *#test*. Con esto en mente podríamos decir que el maximo valor de K sería el tamaño de la muestra de entrenamiento, pero que esto causaría un accuracy sin gran valor real ya que sería el resultado de una sola muestra de test en cada combinación y además resultaría en un costo muy alto en ejecución por lo comentado a continuación. Otra de las cosas que concluimos es que si bien en este caso K-fold

aporta un gran método de validación, esto es gracias a características del conjunto de entrenamiento como que esta bien balanceado, este método no sería muy efectivo para casos donde no pase eso, dando por ejemplo casos donde en el conjunto de test queden elementos de dígitos de los cuales no existen elementos en el conjunto de train.

Es entonces que volvemos a la pregunta disparadora de todo el análisis y la experimentación. A la hora de utilizar K-fold cross validation, **¿Qué K debemos elegir?**. Claramente el K elegido no puede ser muy grande (respecto de $n$) ya que si así fuese se subdividiría en demasiadas partes el conjunto de train arruinando la calidad del entrenamiento. Por el contrario, si K es 1, entonces no estaremos haciendo $K$-fold realmente. Entonces la respuesta recae mucho en el tamaño de $n$, de la homogeneidad de la muestra y de cuán rigurosa se desea la posibilidad de error. La realidad es que no hay una respuesta que se ajuste a todos los contextos. En nuestro caso, dado que MNIST está sumamente balanceada (es decir, que todas las clases constan con una cantidad parecida de imágenes) y dado que la muestra es de 42k imágenes, creemos que $K = 5$ es una elección razonable, ya que de esta manera el conjunto de train se mantiene de 33.5k mientras que el de test aún es razonable con 8.6k siempre que trabajemos con toda la muestra a la vez.

### 4.5.2. Complejidad

Aunque claramente es un método muy útil para evitar problemas como el overfitting (sobreentrenar el algoritmo con un mismo conjunto de train), también sufre de ser muy caro en cuanto a ejecución, ya que se requiere efectuar K repeticiones (todas las combinaciones posibles) para obtener un accuracy, por lo que emplear K muy altos puede verse reflejado en grandes tiempos de ejecución. Dado que K es parámetro de la función, el análisis teórico nos dice que para un algoritmo de costo $O(f(n))$ probar su K-fold cross validation costará $O(K.f(n))$. Por ejemplo si $f(n) = n^2$ y $K \simeq n$ entonces el estudio tendrá un costo cúbico en vez de cuadrático, una diferencia sustancial.

## 4.6. Tamaño de muestra variable

Luego de haber encontrado un k = 7 y $\alpha = 35$ que consideramos optimo, queremos ver como el ir cambiando el tamaño de la muestra total afecta el accuracy final obtenido, para eso vamos a probar usando desde un 10 % de la muestra hasta el total de la misma, aumentando de a 10 % en cada paso.

### 4.6.1. Hipótesis

Creemos que en el gráfico 6 podremos ver como al ir aumentando el tamaño de la muestra, esto resulte en un mejor accuracy.

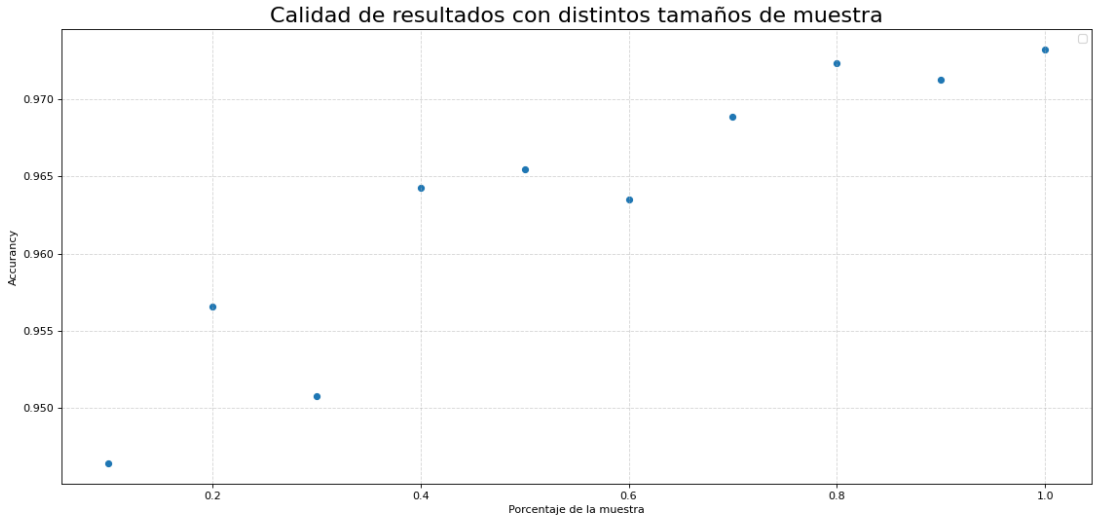


Figura 6: KNN con PCA variando el tamaño de la muestra

#### 4.6.2. Análisis de resultados

Si bien vemos que tiene unos leves descensos en algunos puntos, se puede observar como en su mayoría aumenta el accuracy a medida que aumenta el porcentaje de muestra, esto podríamos atribuírselo a que al tener un conjunto mas grande de train, posee mas valores contra los que comparar y a su vez al mantener el k fijo, es mas probable que los k vecinos mas cercanos sean el dígito correcto del elemento que se quiere reconocer. Ya que diciéndolo a la inversa, si achicamos la cantidad de muestras pero mantenemos el k, es mas probable que tengamos que buscar puntos mas lejanos para obtener lo k vecinos que necesitamos.

#### 4.6.3. Complejidad

Una de las grandes falencias del método es que no posee una memoria que simplifique la predicción, cada predicción requiere analizar las $n$ imágenes de train por lo que a mayor tamaño de muestra veremos un gran aumento en cuanto a tiempo de ejecución.

### 4.7. kNN con y sin PCA

Después de convencernos teóricamente con las aparentes ventajas de preprocesar los datos para el algoritmo de kNN, lo primero que queríamos ver era si la teoría se demostraba en la práctica. Para esto comparamos el tiempo de ejecución y la accuracy de kNN con una base de datos sin *preprocesar* contra kNN con una base de datos *preprocesada* con PCA. En un principio nos planteamos realizar las pruebas con la base de 42000 imágenes de MNIST pero visto y considerando que eso podría demandar mucho tiempo de ejecución decidimos hacer las pruebas con 8000 muestras seleccionadas aleatoriamente.

#### 4.7.1. Hipótesis

Respecto de kNN, debido a la implementación del algoritmo, sabemos que no es para nada veloz, además a pesar de que el costo de calcular 35 autovectores para el método PCA tiene un costo, estimamos que la ganancia que ese costo produce, al ejecutar kNN + PCA (debido a que ya no son 784 componentes si no 35 y esto disminuye el costo de los calculos de distancia) es muy superior. Respecto del accuracy pretendido, hemos probado empíricamente que al aumentar la cantidad de vecinos (en kNN + PCA) el accuracy disminuye notoriamente, estimamos que para el método kNN sin preprocesamiento obtendremos resultados similares.

#### 4.7.2. Análisis de resultados

Podemos observar que kNN + PCA es ampliamente superior a kNN sin PCA en todos los aspectos analizados. No nos quedan dudas que para un problema como este y una muestra tan amplia, el mejor método de los dos es PCA.

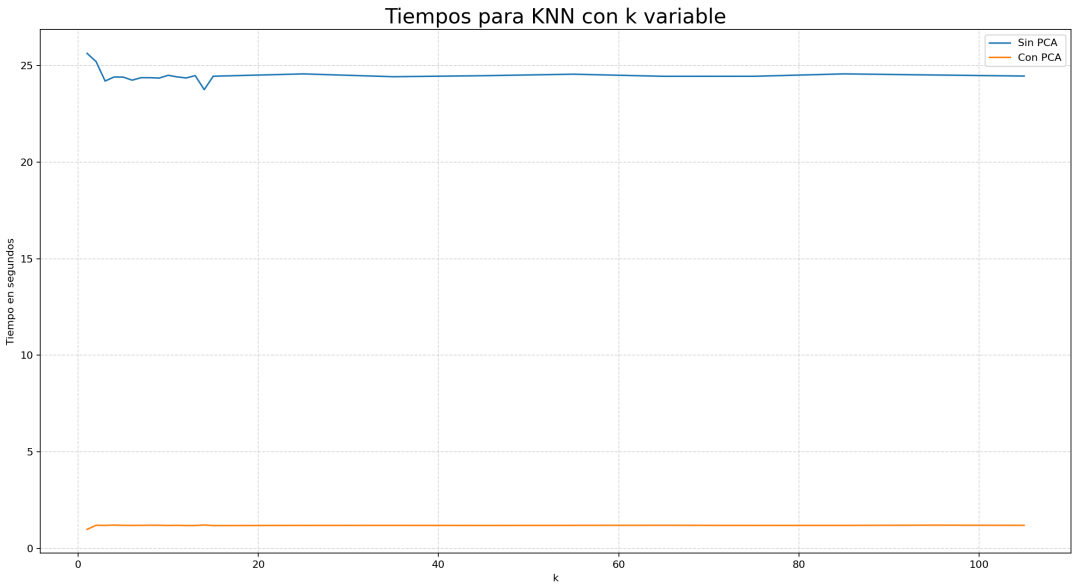


Figura 7: cálculo de tiempo de kNN con y sin PCA para k = [1,2,..15,25,35,..105]

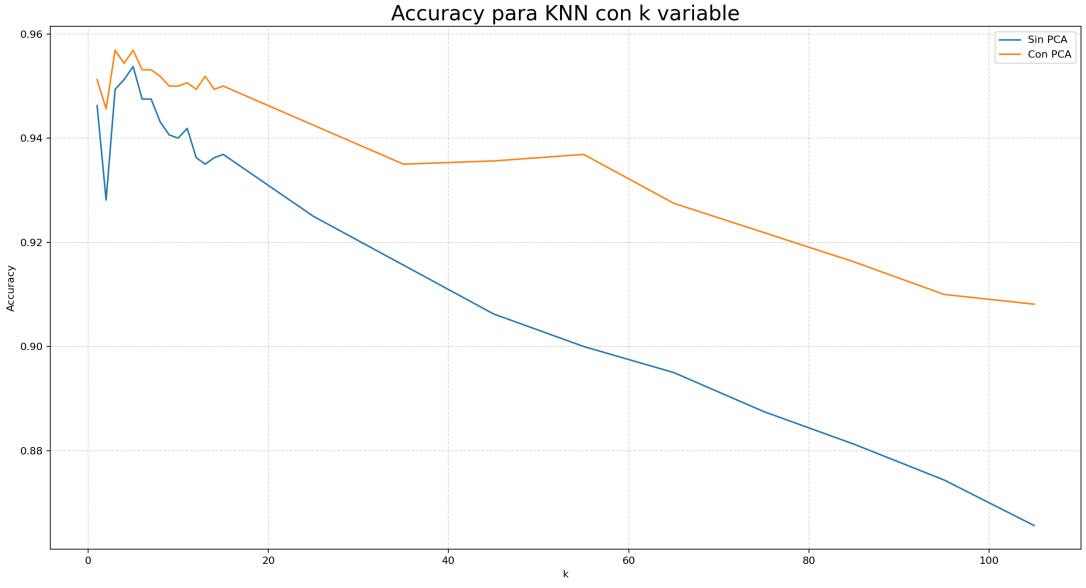


Figura 8: cálculo de accuracy de kNN con y sin PCA para k = [1,2,..15,25,35,..105]

# 5. Conclusiones

En el presente Trabajo Práctico estudiamos dos métodos clásicos de Machine Learning: $k$ vecinos más cercanos (kNN) y Análisis de Componentes Principales (PCA) para un problema aún más clásico: comprensión de dígitos

manuscritos del 0 al 9. Para ello utilizamos la reconocida base de datos MNIST que gracias a sus múltiples bondades nos permitió enfocarnos prácticamente de manera exclusiva en la métrica accuracy.
Lo primero que decidimos investigar fue la combinación de $k$ y $\alpha$ que obtengan el mejor predictor de tipo kNN + PCA. La búsqueda concluyó con $k = 7$ y $\alpha = 35$, parámetros que nos mostraron un 97,3 % de accuracy, resultados bastante competentes teniendo en cuenta que PCA es un método no supervisado, es decir que no conoce los labels del conjunto de train.
Luego introdujimos una herramienta de análisis llamada $K$-fold cross validation, que nos permitió validar que no estuviéramos haciendo overfitting, es decir, que no estuviéramos ajustando los parámetros para que el conjunto particular que estábamos testeando arroje mejores resultados. Luego de probar nuestros parámetros encontrados mediante el método $K$-fold sostuvimos nuestra seguridad de que no estábamos incurriendo en el overfitting y que efectivamente $k = 7$ y $\alpha = 35$ producen las mejores predicciones.
Realizamos múltiples pruebas respecto de variar el K de K-fold y sobre cómo el tamaño de la muestra afecta al accuracy obtenido.
En resumen, el estudio aquí presentado muestra nuestro proceso de elección de $\alpha$ y $k$ para obtener la mejor calidad de predicciones posibles con los métodos propuestos. Somos conscientes del amplio lugar a mejora que existe (nuestra eficacia medida con K-fold fue del 93 % un resultado no tan competente) no solo en la calidad de los resultados si no también en la complejidad (tanto temporal como espacial) de los algoritmos implementados, nos hubiera encantado poder ahondar aún más en estos temas y así llegar a análisis más ricos, lamentablemente el tiempo apremia y hasta aquí tuvimos la oportunidad de pulirlo.