DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 2

Analisis de componentes principales y reconocimiento de imágenes

Mayo-Junio 2024 Métodos Numéricos

Integrante	LU	Correo electrónico
Begalli, Juan Martin	139/22	juanbegalli@gmail.com
Carrillo, Mariano	358/22	carr.mariano@gmail.com
Serapio, Noelia	871/03	noeliaserapio@gmail.com

Resumen

En el presente trabajo buscamos implementar un reconocedor de imágenes, utilizando el método de la potencia con deflación, validación cruzada, knn y pca.
Desarrollamos los 4 algoritmos mencionados anteriormente y los relacionamos a todos en el calculo del pipeline final el cual se encarga de la exploración de los hiperparámetros de k vecinos para knn y p componentes para pca.
Utilizamos validación cruzada para el calculo de dichos hiperparámetros con los datos de entrenamiento. Luego, al tener los mejores parámetros pudimos calcular la exactitud (performance) en los datos de test. Para el método de la potencia con deflación estudiamos su convergencia y la cantidad de iteraciones óptimas.

Palabras Clave: *KNN, metodo de la potencia, k-cross-validation, PCA*

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

Índice

1. Introduccion

En el presente trabajo desarrollaremos un modelo de reconocimiento de imágenes, utilizando el conjunto de datos Fashion Mnist.

Para ello, implementaremos el algoritmo de *k-nearnest neighbors (KNN)* y *análisis de componentes principales*(PCA, por sus siglas en inglés).

KNN se basa en el hecho de que los datos pueden ser subdivididos en distintas clases distinguibles entre sí (en este caso prendas). De esta forma se busca, dada una imagen de una prenda desconocida, encontrar a qué clase pertenece. Para ello, se debe decidir cómo se medirá la cercanía de una imagen a otra. Como en este caso se trabaja con imágenes, ellas pueden ser descriptas como un vector de pixeles en donde cada pixel tendrá cierto valor indicando su color (si trabajamos con imágenes en blanco y negro, será la escala entre 0 y 1).

Luego se utiliza, un subset de los datos principales, $T = \{x_i : i = 1, ..., n\}$ que llamaremos de *entrenamiento*. Para realizar predicciones sobre otras imágenes $x \notin T$ calculamos "que tanto se parece"la imagen x a las imágenes de T y luego nos quedamos con las k más cercanas. De este conjunto de imágenes debemos buscar cuál es la clase que más se repite, y de esa forma asignarle a x dicha clase.

Algo a considerar es que k es lo que se conoce como *hiperparámetro* pues es un parámetro que elige quien implementa el algoritmo. Y es posible que la exactitud y los resultados del modelo varíen dependiendo del k elegido.

Por otro lado, se buscará optimizar *KNN* implementando el algoritmo de PCA. Este algoritmo usa fuertemente el concepto de *Covarianza*. La covarianza entre dos vectores se define como:

$$Cov(\mathbf{x}, \mathbf{y}) = \frac{(\mathbf{x} - \mu_x) \cdot (\mathbf{y} - \mu_y)}{n - 1} \tag{1}$$

Donde μ es el promedio del valor de los vectores (valor medio). La covarianza nos indica que tanto cambia un vector con respecto al otro. Es decir, si normalizamos esta formula (*correlación*) obtendremos un valor entre -1 y 1. Donde cuanto más cercano a 1 esté la correlación, más paralelos serán los vectores entre sí (en la misma dirección). Si es negativa, irán en direcciones contrarias. Si es 0 es porque son paralelos.

$$Corr(\mathbf{x}, \mathbf{y}) = \frac{(\mathbf{x} - \mu_x) \cdot (\mathbf{y} - \mu_y)}{\sqrt{(\mathbf{x} - \mu_x) \cdot (\mathbf{x} - \mu_x) \cdot (\mathbf{y} - \mu_y) \cdot (\mathbf{y} - \mu_y)}} \tag{2}$$

Usando esta fórmula, podemos construir una matriz $\mathbf{C}$ que llamaremos martiz de covarianza. En donde sus columnas serán los vectores columna $\mathbf{x} - \mu_{x_i}$.

$$\mathbf{C} = \frac{X^t X}{n - 1} \tag{3}$$

Ahora bien, si representamos a los datos en una matriz $\mathbf{X} \in \mathbb{R}^{m \times n}$ el algoritmo de PCA busca reordenar los datos de manera tal que el orden final nos permita operar con los datos más relevantes en las primeras iteraciones. Es decir, ordena las dimensiones segun su varianza, de mayor a menor. El ordenamiento se realiza mediante una matriz de cambio de base. Esta matriz se encuentra utilizando la matriz de covarianza 3, pues se busca realizar una descomposicion en autovectores y autovalores.

$$\mathbf{C} = \mathbf{V}\mathbf{D}\mathbf{V}^t \tag{4}$$

Donde $\mathbf{V}$ contiene los autovectores de $\mathbf{C}$ en sus columnas y $\mathbf{D}$ los autovalores. Los autovalores y autovectores se encontrarán implementando el **método de la potencia** [1]. Entonces, realizando la operación $\mathbf{XV}$ se encuentra la transformación de los datos deseada. Una parte importante de este algoritmo, es que no siempre es necesario utilizar todas las componentes principales (columnas de $\mathbf{V}$) y de está manera podemos reducir la dimensionalidad de los datos logrando obtener buenos resultados y haciendo que el computo sea más eficiente. Por eso, se explora el hiperparámetro p buscando cuántas p componentes son necesarias para realizar una buena aproximación sin perdida de información relevante.

Para buscar los parámetros k y p óptimos utilizaremos el proceso llamado *k-fold cross-validation*, con $k = 5$ basado en la validación cruzada [2]. La idea consiste en, dados los conjuntos de datos de entrenamiento y pruebas, generar un nuevo conjunto de datos llamado de *desarrollo* a partir del de entrenamiento. Este proceso se va a repetir *5* veces a partir de una partición de *5* partes del conjunto de entrenamiento donde se espera que la distribución de las prendas de cada clase sea igual en cada partición. Usando validación cruzada se busca encontrar valores óptimos para k la cantidad de vecinos en knn y p la cantidad de componentes principales en pca que permita generar modelos con precisión lo más alta posible y generalizable a datos nuevos.

2. Desarrollo

2.1. KNN

Para utilizar en el modelo de reconocimiento de imágenes se implementó el algoritmo de *k-nearest neighbours* (KNN) utilizando *Python* y *NumPy*. Para ello utilizamos dos conjuntos de datos que llamaremos de *desarrollo* y de *entrenamiento*. Donde el segundo cuenta con mayor cantidad de datos que el primero.

El esquema general a seguir fue el siguiente: Para cada dato en el conjunto de desarrollo se le calculó la *distancia* con respecto a cada dato del conjunto de entrenamiento. Luego se ordenaron estas distancias de menor a mayor y nos quedamos con las k imágenes más cercanas. Por último, identificamos a qué tipo de prenda pertenecen cada una de las k imágenes y nos quedamos con el tipo predominante (la moda). La medida de distancia utilizada fue la *distancia coseno*, que dada una imagen A y otra B (representadas como un vector de 784 pixeles) se define así:

$$d(A, B) = 1 - \frac{A \cdot B}{\|A\|\|B\|} \tag{5}$$

En un primer momento la implementación consistió en recorrer los conjuntos de entrenamiento y de desarrollo con dos *for* de python, calculando en cada iteración la distancia del dato de desarrollo con respecto a todos los de entrenamiento. Ordenar esas distancias, ver los k primeros y calcularle la moda. Este enfoque, si bien funcionaba, era altamente ineficiente pues los *for* de python son lentos en comparación con otra operaciones.

Es por esto, que la versión final utilizó fuertemente las operaciones de *NumPy* para matrices. Ya que tanto los datos de desarrollo como los de entrenamiento eran matrices donde cada fila representaba una imagen.

Para la distancia coseno, primero se calcula la norma de todos los datos de entrenamiento y los de desarrollo utilizando *numpy.linalg.norm* con lo cual con solo dos llamados a esta función ya estaban calculadas todas las normas. Luego, para calcular el producto escalar entre cada imagen de desarrollo con cada imagen de entrenamiento bastaba con realizar una multiplicación de matrices entre dichas matrices. Así, en la posición *(i,j* de esa matriz resultado estaba el producto escalar entre la imagen de desarrollo i y la de entrenamiento j. Podemos dividir por las normas previamente calculadas (que como se guardan en el mismo orden de cada imagen, se realiza la división en el lugar correcto). Finalmente, a una matriz con 1s con el tamaño adecuado se le restó la matriz con el producto escalar normalizado. Y de esa forma realizamos la distancia coseno (5) de todas las imágenes deseadas.

Una vez que tuvimos la matriz con distancias, se ordenó cada fila utilizando *numpy.argsort*. Eliminamos las columnas a partir de la k+1 (pues solo nos interesan las primeras k) y obtenemos a qué tipo de prenda corresponde cada una de las k prendas utilizando *advanced indexing* [1]. Haciendo esto, obtengo una matriz donde los elementos de la fila i se corresponden con las imágenes de la fila i en la matriz con las k imágenes más cercanas. Finalmente, a cada fila de la matriz calculada en el paso anterior se le calcula la moda, la posición i del array se corresponde con la predicción sobre la imagen de desarrollo i. Como último paso realizamos el computo de la performance total que es el promedio de los aciertos que se efectuaron en los datos de prueba y se devuelve dicho valor que es la exactitud.

Pseudocodigo:

in: $X_t \in R^{n \times q}$, $X_d \in R^{m \times q}$, $Y_t \in R^n$, $Y_d \in R^m$, $k \in N$
$\Rightarrow$ **out:** $m \in R^n$

KNN

1: $normas\_X\_d \leftarrow \|X_d\|$
2: $normas\_X\_t \leftarrow \|X_t\|$
3: $prod\_escalar \leftarrow X_d \cdot X_t$
4: $tmp \leftarrow \frac{prod\_escalar}{normas\_X\_t \cdot normas\_X\_d}$
5: $dists\_coseno \leftarrow matriz\_unos - tmp$
6: $ordenar(dists_coseno)$
7: $primeros\_k \leftarrow mantener\_k\_cols(dists\_coseno)$
8: $prendas \leftarrow indexar(Y_t, primeros\_k)$
9: $m \leftarrow moda(prendas)$
10: $e \leftarrow promedio(m == Y_d)$
11: **return** e

[1] https://numpy.org/doc/stable/user/basics.indexing

2.2. Método de la Potencia con deflación

Este algoritmo fue hecho en C++ debido a que la velocidad de ejecución es mejor que en Python, y como se trata de un algoritmo en el que se itera muchas veces, es bastante relevante esta diferencia de tiempos de computo. Por consiguiente, implementamos las funciones *metodoPotencia* y *eigen* en C++.

Pseudocodigo:

in: $A \in R^{n \times n}$, $niter \in \mathbb{Z}$, $eps \in$ double $\Rightarrow$ **out:** metodoPot_res

metodoPotencia

1: $v \leftarrow random(tam(n))$
2: $i \leftarrow 0$
3: **while** $(i < niter)$ **do**
4: $v\_viejo \leftarrow v$
5: $v \leftarrow \frac{Av}{\|Av\|}$
6: $\lambda \leftarrow \frac{v^t Av}{v^t v}$
7: **if** $\|v\_viejo - v\|_\infty \leq eps$ **then**
8: **break**
9: **end if**
10: $i \leftarrow i + 1$
11: **end while**
12: $metodoPot\_res \leftarrow (\lambda, v, i)$
13: **return** $metodoPot\_res$

in: $A \in R^{n \times n}$, $num \in \mathbb{Z}$, $niter \in \mathbb{Z}$, $eps \in$ double $\Rightarrow$ **out:** eigenRes

eigen

1: $A\_copia \leftarrow A$
2: $autoValores$
3: $autoVectores$
4: $iteraciones$
5: **for** $i \leftarrow 0,..., num - 1$ **do**
6: $p \leftarrow metodoPotencia(A\_copia, niter, eps)$
7: $autoValor \leftarrow p.autovalor$
8: $autoVector \leftarrow p.autovector$
9: $iteracion \leftarrow p.iteracion$
10: $iteraciones(i) \leftarrow iteracion$
11: $autoValores(i) \leftarrow autoValor$
12: $autoVectores.fila(i) \leftarrow autoVector^t$
13: $A\_copia \leftarrow A\_copia - autoValor(autoVector \,.\, autoVector^t)$
14: **end for**
15: $eigenRes \leftarrow (autoValores, autoVectores, iteraciones)$
16: **return** $eigenRes$

El primer algoritmo recibe una matriz cuadrada A, un entero niter (determina la cantidad máxima de iteraciones que se harán) y un entero eps (usado como valor para el criterio de parada). En este algoritmo obtenemos el autovector y autovalor dominante a partir de un vector aleatorio haciendo que el nuevo vector sea igual a la matriz A multiplicada por el vector viejo dividido la norma de esta multiplicación. De esta manera en cada iteración, encontramos un vector que se acerca cada vez más al autovector dominante. A partir de este obtenemos el autovalor (λ) asociado haciendo "$autovector^t \cdot A \cdot autovector$´´ para luego dividirlo por el producto interno del autovector consigo mismo, así también terminará convergiendo en el autovalor dominante.

Para este algoritmo usamos como método de corte la norma infinito de la diferencia entre el vector viejo con el vector nuevo, esto es necesario para que algoritmo no itere de más en el caso de ya haber encontrado una buena aproximación del autovalor.

Finalmente devolvemos en estructura de tupla el autovector, autovalor y las iteraciones (útil para estudiar la convergencia posteriormente) necesarias para llegar al valor deseado.

Luego el algoritmo *eigen* representa al método de potencia con deflación. Este recibe lo mismo que el algoritmo anterior más un entero *num* que determina la cantidad de autovalores que debemos encontrar. Como precondición debemos asegurarnos que los autovalores de la matriz sean tales que $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$.

Para hallar todos los autovalores y autovectores de la matriz lo que hacemos es usar el método de la potencia para obtener el primer autovalor y autovector, luego con estos podemos restarle a la matriz A este producto $autovalor(autovector.autovector^t)$,

con la cual obtenemos una matriz con autovalores $0, \lambda_2, \cdots, \lambda_n$. Con esta matriz nueva podemos volver a usar el método de la potencia y así sucesivamente hasta hallar todos.

Una vez encontrados todos los autovectores y autovalores usando la deflación, los devolvemos en una tupla con un vector con los autovalores, una matriz con los autovectores(cada uno es una columna de la matriz) y un vector de enteros con las iteraciones correspondientes.

Por último para poder usar esta función con Python para realizar gráficos e experimentos, tuvimos que implementar un main que lee de un archivo de texto la cantidad de filas/columnas, la cantidad de autovalores y la matriz de la cual se quieren obtener estos autovalores; y también recibe el niter, el eps y 3 archivos de salida. Seguido a esto llama a la función eigen pasando los parámetros necesarios, al terminar se escriben los autovalores, autovectores y las iteraciones realizadas en 3 diferentes archivos de texto. Estos tres archivos son los que usaremos para leer desde Python los resultados.

Para poder corroborar nuestra implementación, escribimos en Python una función que genera matrices(generate_matrixTest) en las cuales ya conocemos los autovalores usando el truco de HouseHolder.

$$M = Q^t \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ 0 & 0 & \lambda_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 0 & \lambda_n \end{bmatrix} \text{Q}$$

en donde $Q = I - 2vv^t$ con $\|v\|_2 = 1$.

Además tenemos una función (*guardarMatriz_file*) que escribe estas matrices generadas junto con la cantidad de autovalores al archivo de texto que le pasamos por parámetro del cual el programa en C++ leerá para realizar el método de potencia con deflación.

Para poder llamar a la función *eigen* de C++ tenemos la función *ejecutar* que recibe la ruta en donde se ubican los archivos donde deberá leer y escribir y utiliza el compilador g++, para ejecutar desde Python con los parámetros correspondientes (usamos el comando de compilación -O3 como optimización al compilar).

Finalmente generamos los tests mediante el algoritmo *tests* que recibe como parámetro la cantidad de autovalores, un array con los niters que vamos a usar, la ruta donde deberá guardarse y el número de test que estoy ejecutando. Usa las implementaciones de *generate_matrixTest* y *guardarMatriz_file* para generar y guardar las matrices aleatorias del tamaño de cantidad de valores y, luego ejecutamos con la función mencionada anteriormente. Primero hacemos un test con 6 cantidad de autovalores diferentes(10, 25, 50, 100, 300, 784) y usamos un máximo de 5000 iteraciones. Una vez terminado de ejecutarse la función *eigen* para cada cantidad de autovalores leemos los resultados de los tres archivos de texto y los guardamos en tres arrays. Para corroborar que los resultados sean correctos los comparamos con los autovalores que ya conocíamos usando *np.allclose*.

Luego hacemos un test con una matriz con 100 autovalores y con diferentes números de niter(10, 100, 1000, 10000) para ver como varía dependiendo la cantidad de iteraciones y con un eps 10^{-8} para que tampoco sea necesario que llegue siempre al máximo de iteraciones. Este test lo representamos usando cuatro gráficos, uno por cada valor de niter para poder poder comparar más fácil luego. En este test creemos que con 10 y 100 iteraciones no alcanza para dar un resultado preciso sin embargo a partir de 1000 creemos que si.

Por último tenemos un test en el cual tenemos una matriz de 500 autovalores, con un eps de 10^{-15}, así llega a usar todas las iteraciones dadas por los siguiente niters : 10, 500, 1000, 10000, 20000. Esto lo hacemos ya que queremos graficar el error que hay en cada autovalor dependiendo de cuantas iteraciones realice el algoritmo. Para medir el error usamos $|\lambda_{metPot} - \lambda_{real}|$. En este caso creemos que podemos asegurar que a medida que mayor sea la cantidad de iteraciones más precisos serán todos los autovalores obtenidos, es decir habrá menos error.

2.2.1. Convergencia

Para estudiar la convergencia de este método, medimos la cantidad de iteraciones que tarda en converger con una tolerancia de 10^{-7} y una cantidad máxima de 4000 iteraciones, además medimos el error usando $\|Av_i - \lambda_i\ v_i\|$ donde λ_i y v_i son la aproximación al autovector y autovalor i. En este caso generamos matrices de Householder aleatorias con los valores de la diagonal $\{10, 10 - \epsilon, 5, 2, 1\}$, el epsilon nos sirve para ver como cambian las iteraciones a medida que dos autovalores se acercan o se alejan más entre sí.

Para este test creamos un array de 80 epsilons de menor a mayor con valores entre 10^{-4} y 10^0, luego por cada epsilon generamos 30 matrices aleatorias de Householder usando la función *generate_matrizError* que recibe por parámetro el *epsilon* y buscamos autovalores y autovectores para cada una de ellas. De esas 30 matrices sacamos el promedio de cuantas iteraciones tardó cada autovalor en converger y el error promedio de cada uno de estos. También calculamos el desvío estándar para mejorar la calidad de los valores del gráfico.

Creemos que la cantidad de iteraciones promedio y el error promedio disminuirá a medida que haya más diferencia entre el autovalor dominante y el segundo, es decir a medida que crezca el epsilon.

2.3. Validación cruzada

Si disponemos de un conjunto de datos de entrenamiento y otro de prueba, validación cruzada permite medir la performance de un sistema de forma más generalizada obteniendo los mejores hiperparámetros, en comparación a si se realiza una exploración que indica que cierta configuración de los hiperparámetros tiene la mejor performance en el conjunto de prueba, estaríamos obteniendo un resultado poco generalizable, sesgado y demasiado optimista. Es decir, no podemos saber si esa configuración de hiperparámetros será la mejor para conjuntos de datos no vistos.

Por lo tanto, implementaremos primero validación cruzada solamente para KNN para el dataset Fashion MNist, partiendo los datos de entrenamiento en 5 partes iguales, con la misma cantidad de prendas de cada tipo en cada partición. Para hacer esto fue necesario solamente partir los datos en orden, pues las prendas en el conjunto de datos venían ordenados con la siguiente secuencia: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 y luego se vuelve a repetir dicho patrón en todo el dataset de y_train e y_test.

De las 5 particiones que hicimos tomaremos 4 como datos de entrenamiento y la restante como de desarrollo, esto lo iteraremos de forma cíclica 5 veces, y de esta manera todos los datos se usaron como datos de desarrollo en algún momento, como se puede ver en la siguiente figura:

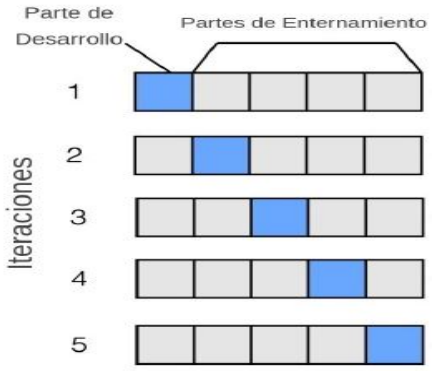

Figura 1: Validación cruzada usando 5-fold

Pseudocodigo:

in: $X\_train \in R^{n \times m}, Y\_train \in R^n$ **out:** $exactitud$

validacionCruzadaKNN

1: $x1, x2, x3, x4, x5 \leftarrow particionar(X\_train)$
2: $y1, y2, y3, y4, y5 \leftarrow particionar(Y\_train)$
3: **for** i = 1,...,5 **do**
4: $\quad x_{junta} \leftarrow concatenar(x1, x2, x3, x4, x5) - xi$
5: $\quad y_{junta} \leftarrow concatenar(y1, y2, y3, y4, y5) - yi$
6: $\quad performance[i] = knn(x_{junta}, xi, y_{junta}, yi, k)$
7: **end for**
8: $exactitud \leftarrow promedio(performance)$
9: **return** exactitud

Al ejecutar **validacionCruzadaKNN** en los datos de entrenamiento explorando con $k \in 1, .., 9$ se obtiene que el mejor k, es decir, la mejor cantidad de vecinos que maximiza la exactitud del programa es **k=4** y su exactitud es **0.8058000000000002**.

2.4. PCA

El análisis de componentes principales (PCA) busca encontrar un cambio de base de los datos (que los podemos ver como vectores), de tal manera que las dimensiones se ordenen en componentes que explican los datos de manera decreciente. El cambio de base busca ordenar las dimensiones según su varianza de forma decreciente. Para hacer esto se realiza una descomposición en autovectores y autovalores de la matriz de covarianza de los datos. Para encontrar los autovectores y autovalores de PCA utilizaremos el método de la potencia con deflación (eigen) hecho en C++, lo invocamos con un niter de 10000.

Pseudocodigo:

in: $X \in R^{n \times n}$ **out:** $(autoval, autovect)$

PCA

1: $matrizCov \leftarrow generarMatrizCovarianza(X)$
2: $(autovalores\ , autovectores) \leftarrow eigen(matrizCov)$
3: **return** $(autovalores\ , autovectores)$

2.5. Pipeline Final

Por último, se implementó la exploración final de los hiperparámetros k y p de KNN y PCA, respectivamente. Para ello, utilizamos *5-fold cross-validation* particionando al conjunto de datos de entrenamiento en 5 partes (folds) con la misma proporción de tipos de prendas en cada fold.

Luego, entrenamos PCA con cada una de esas partes, probando con distintos valores de k y de p en cada iteración. Es decir, para la primera iteración, luego de entrenar PCA con ese fold, exploramos distintos valores de k y de p y nos fuimos guardando la performance en **ese** fold para cada par (k,p).

Una vez terminado ese proceso, la etapa final consistió, en primero, tomar el promedio entre las performance de cada uno de los folds para cada par (k,p) explorado. En cada iteración de esta étapa se buscaba mejorar el promedio actual. Si se encontraba un mejor promedio, también se actualizaba el valor de k y de p, pues este par (k,p) era el que mejor performance tenía.

Luego, una vez encontrados los hiperparamétros k y p que mejor performance obtenían, ejecutamos PCA con todos los datos de entrenamiento. Esto, para luego aplicarle el cambio de base obtenido a los datos de entrenamiento y los de testeo. Y por último, obtener la performance de KNN con los datos de testeo (aun no explorados) con el k óptimo que se obtuvo antes.

Es de esperarse que la performance con los datos de testeo fuera similar a la obtenida durante el entrenamiento (por lo menos a la de mejor performance). Si esto ocurriese, quiere decir que el modelo fue entrenado correctamente y que los resultados que devuelve son generalizables y no tan sesgados. Por lo tanto, el modelo satisfaría lo que se planteó al principio del trabajo.

Pseudocodigo:

in: $X\_train \in R^{n \times q}$, $X\_test \in R^{m \times q}$, $Y\_train \in R^{n}$, $Y\_test \in R^{m}$ **out:** *exactitud*

PipeLineFinal

1: $x1, x2, x3, x4, x5 \leftarrow particionar(X\_train)$
2: $y1, y2, y3, y4, y5 \leftarrow particionar(Y\_train)$
3: $PCA_i \leftarrow 5\_fold - cross\_validation(\{x_i\}_{i=1,..,5}, \{y_i\}_{i=1,..,5})$
4: $\{res_i\}_{i=1,..,5} \leftarrow crear\_5\_listas()$
5: **for** i = 1,...,5 **do**
6: **for** k = 1,..,9 **do**
7: **for** p in cant_comp_pcpales **do**
8: $\hat{x}_{train} \leftarrow cambioBase(\{x_j\}_{j=1,...,5 \neq i}, PCA_i)_p$
9: $\hat{x}_{dev} \leftarrow cambioBase(x_i, PCA_i)_p$
10: $res_i \leftarrow agregarAtras((p, k, KNN(\hat{x}_{train}, \hat{x}_{dev}, \{y_j\}_{j=1,...,5 \neq i}, y_i, k)))$
11: **end for**
12: **end for**
13: **end for**
14: $mejor\_promedio \leftarrow Obtener\_mejor\_promedio(\{res_i\}_{i=1,...,5})$
15: $PCA_{final} \leftarrow PCA(X_{train})$
16: $\hat{x}_{train\_final} \leftarrow cambioBase(X\_train, PCA_{final})_{mejorP}$
17: $\hat{x}_{test} \leftarrow cambioBase(X\_test, PCA_{final})_{mejorP}$
18: $exactitud \leftarrow KNN(\hat{x}_{train\_final}, \hat{x}_{test}, Y\_train, Y\_test, mejorK)$
19: **return** exactitud

3. Resultados y discusión

3.1. Exploración con KNN

Calculamos utilizando KNN la exactitud para $k = 5$ con los datos de Fashion MNist y nos dio **0.834**.
Al analizar los resultados explorados en el desarrollo del testeo de nuestra implementación de validación cruzada para KNN se obtuvo que el k que daba la mejor performance entre 1 y 9 era **k=4**. Este resultado puede compararse con el que se obtuvo posteriormente luego de procesar las imágenes mediante PCA. Pues este procesamiento ayudó a reducir las dimensiones de las imágenes y facilitó su análisis.

3.2. Método Potencia con deflación

En las figura 2 y 3 podemos observar, como predijimos, que con 10 y 100 iteraciones no era suficiente para asegurar que el autovalor obtenido sea el correcto. Con 10 iteraciones se ve que hay una gran cantidad de autovalores que difieren del real y con 100 hay menos error pero sigue habiendo un error suficiente como para decir que no convergió correctamente.

Luego en la figura 2 parecería que con 1000 y 10000 iteraciones es igual o casi igual a los autovalores reales pero luego en la figura 3 podemos ver como con 1000 iteraciones, si bien tienen una media de error bajo comparado con 100 o 500, todavía hay una cantidad importante de valores que se alejan mucho de esta por lo tanto no es del todo preciso.

Pero luego, con 10000 iteraciones vemos como la media de error baja bastante y se concentra mucho más que con 1000, aunque todavía ocurre que hay valores q se alejan de su media.

Finalmente con 20000 si bien la media con el anterior no varía tanto podemos observar como se reduce la cantidad de valores que quedan por fuera de este promedio, por lo tanto asumimos que a medida que aumenta la cantidad de iteraciones menos valores quedaran fuera de la media y más precisos serán.

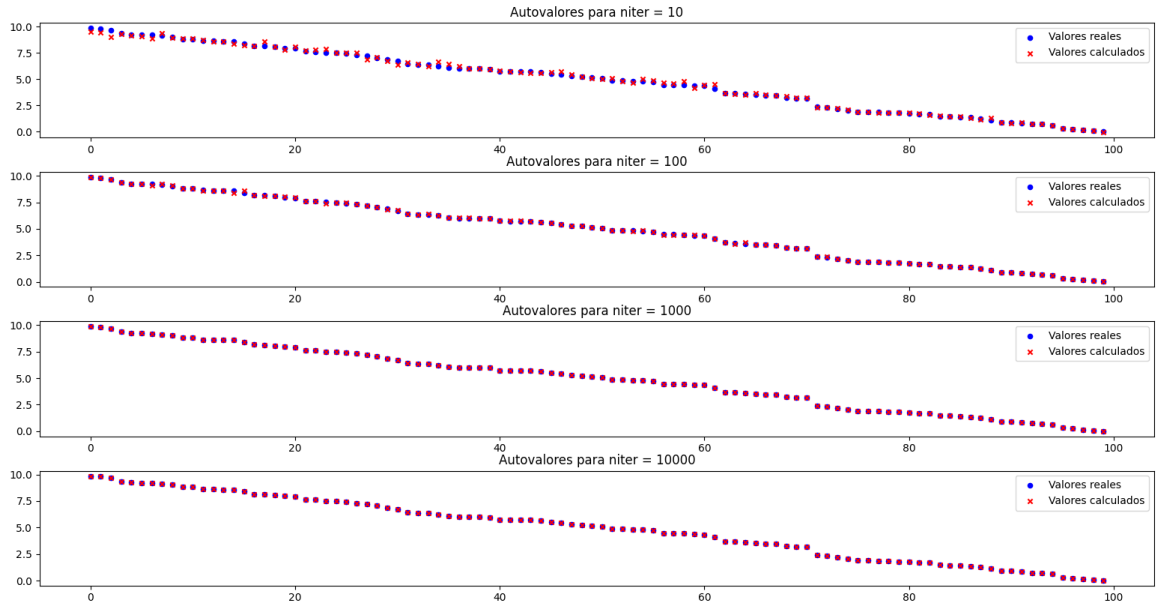

Figura 2: comparación de autovalores, obtenidos por nuestra implementación y autovalores reales, con 4 valores diferentes de niter

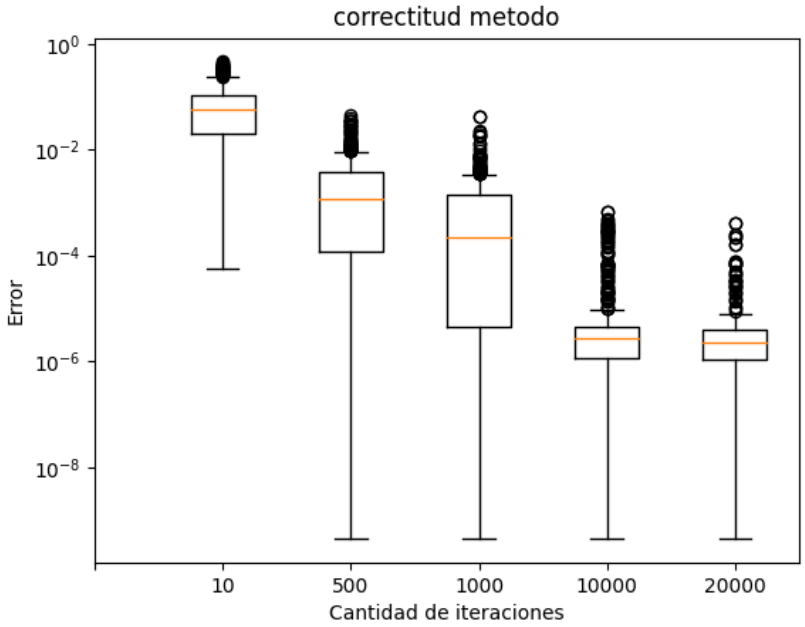

Figura 3: medida de error haciendo $|\lambda_m - \lambda_r|$, es decir autovalores obtenidos por nuestra implementación y autovalores reales, con 5 valores diferentes de niter

3.2.1. Convergencia

En la figura 4 se puede ver como el error promedio de los dos primeros autovalores disminuye cada vez más a medida que el epsilon (ϵ) aumenta y también parecería que se estabiliza, pues hay más distancia entre el autovalor dominante y el segundo. El resto de los autovalores, que no dependen del ϵ y no son cercanos entre sí, no dependen de como varía epsilon y poseen un error promedio bastante menor al de los otros dos, aunque con un desvío estándar mayor.

Al ver la figura 5 podemos ver las iteraciones promedio que tarda en converger cada autovalor, usando el criterio de corte planteado en el método de la potencia. Aquí ocurre que el valor dominante de la matriz primero tarda 4000 iteraciones(es el máximo que le pusimos) pero a medida que vamos aumentando poco a poco el ϵ, es decir aumentamos la distancia entre el autovalor dominante y el segundo, las iteraciones empiezan a bajar, hasta menos de un cuarto de cuando el epsilon es chico. El segundo autovalor parecería aumentar pero en muy lentamente la cantidad de iteraciones a medida que el epsilon crece. No podemos observar ningún tipo de relación directa del valor de espilon con el resto de los autovalores pues no parecen tener un comportamiento anticipable a medida que este varía. Lo que si observamos es como los cuatro autovalores que no son el dominante poseen mayor desvío estándar que los del primer autovalor, parecería ser que sus resultados son menos consistentes.

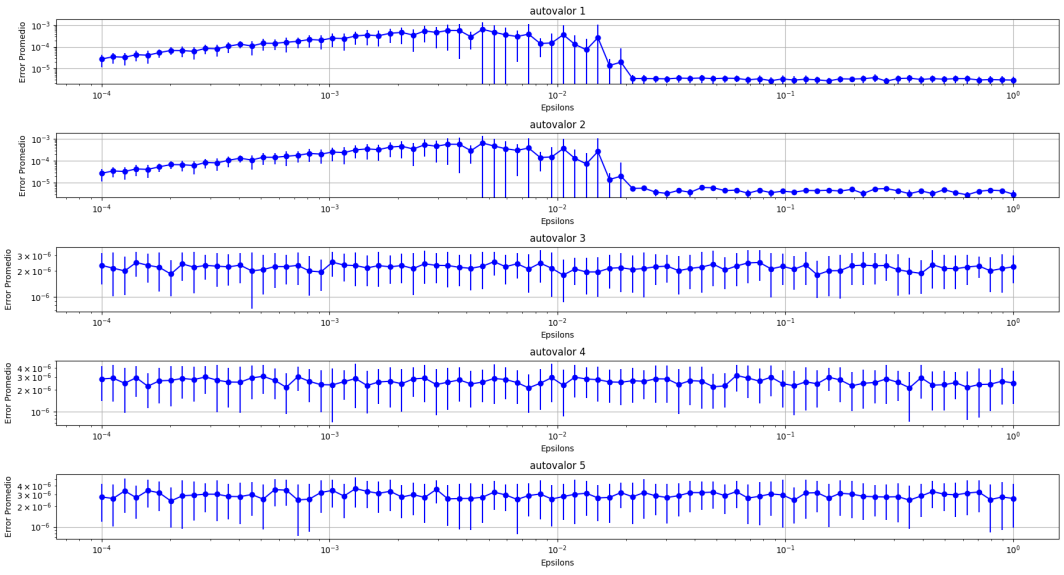

Figura 4: Error promedio de cada autovalor con respecto al ϵ. Medida de error dada por $\|Av_i - \lambda_i\ v_i\|$.

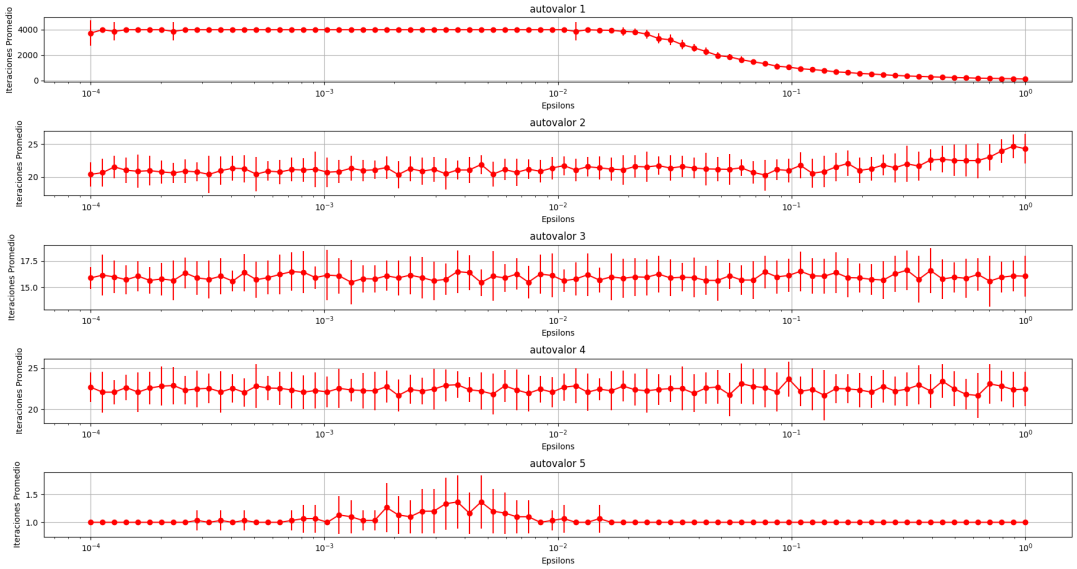

Figura 5: Cantidad iteraciones promedio para cada autovalor vs el ϵ

3.3. PCA

Calculamos PCA con todos los datos de entrenamiento, sus autovalores son la varianza explicada de mayor a menor. Utilizamos dichos valores para obtener la varianza acumulada en función de la cantidad de componentes (p) como se puede ver en la siguiente figura:

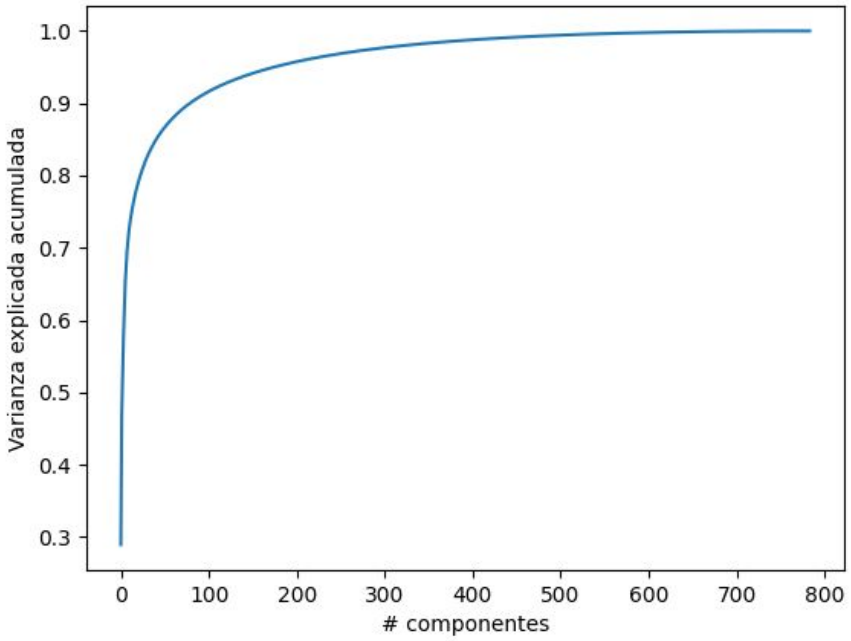

Figura 6: Varianza explicada en función de cantidad de componentes

Como se aprecia en el gráfico con 100 componentes principales (es decir, si el hiperparametro p es 100) se obtiene una varianza acumulada de aproximadamente el 90 %, que es un buen porcentaje para la varianza acumulada. Es decir, con estos porcentajes pudimos cumplir con la precisión que esperábamos y necesitábamos al implementar el algoritmo de PCA.

3.4. Pipeline final

Para hacer la exploración final de hiperparámetros y ver cual es el que nos da la máxima exactitud probamos con $k \in \{1, .., 9\}$ y $p \in \{2, 3, 4, 5, 25, 50, 75, 100, 200, 300, 400, 500, 600, 700, 784\}$ usando validación cruzada en los datos de entrenamiento obtuvimos que la exactitud del programa era de **0.8225999999999999** y está se alcanzada con $\mathbf{p = 200}$ y $\mathbf{k = 5}$. Es decir los mejores hiperparámetros encontrados fueron de 200 componentes principales para pca y de 5 vecinos para knn.

Con estos hiperparámetros se procesan todos los datos de entrenamiento con PCA y se utiliza los autovectores devueltos por PCA para transformar los datos de testing y obtener la performance (exactitud) en los datos de test, la cual es **0.838**.

Por lo tanto, corroboramos la hipótesis planteada durante el desarrollo del pipeline final (ver sección 2.5). Ya que se obtuvo una performance similar (incluso superior) a la obtenida durante la etapa de entrenamiento. Esto quiere decir que nuestro modelo, con los hiperparámetros encontrados, puede clasificar con precisión relativamente elevada y generalizable, datos desconocidos.

4. Conclusiones

A lo largo de este trabajo se construyó, mediante la implementación de diferentes algoritmos, un clasificador de imágenes utilizando un data set arbitrario (Fashion Mnist). Una de las principales características que se tuvo en cuenta para implementar el sistema de aprendizaje automático fue que la perfomance obtenida en la fase de entrenamiento se mantuviera similar al procesar datos no vistos.

Por esa razón fue importante el uso de la técnica de *k-fold cross-validation* detallada en la sección 2.3. A su vez, esta técnica nos permitió seleccionar los mejores hiperparámetros para las implementaciones de KNN y PCA (k y p respectivamente). Pues, era importante conocer y visualizar "que tan bueno era el hiperparámetro"para cada implementación, para poder utilizarlos en la etapa final donde se entrenó al modelo utilizando todos los datos de entrenamiento.

Por último, una sección importante en el trabajo fue la implementación de un algoritmo iterativo para encontrar autovalores y autovectores. En este caso, se implementó el método de la potencia. Fue significativo poder visualizar los errores y analizar la convergencia de nuestra implementación (basándonos en casos conocidos). Porque de esa forma pudimos estar seguros no solo de que la implementación cumplía con lo requerido, sino que lo hacía de manera óptima. Un ejemplo de esto fue cuando se analizó la cantidad de iteraciones necesarias para que el error entre el autovalor (o autovector) requerido y el obtenido fuera el menor. Ya que, por más de que con una cantidad alta de iteraciones la precisión aumenta, se podía ajustar dicha cota para no realizar cómputo de más. Esta implementación fue útil al realizar el análisis de componentes principales (PCA) de las imágenes, y como se debía hacer muchas veces, era necesario que fuese óptima.

Referencias

[1] J. Douglas Faires Richard L. Burden. *Análisis numérico*. International Thomson Editores, 2002.

[2] Peter E Hart Richard O Duda y David G Stork. *Pattern classification*. John Wiley Sons, 2012.