Métodos Numéricos

Trabajo Práctico 2

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

Análisis de componentes principales y reconocimiento de imágenes

Integrante	LU	Correo electrónico
Montaron, Josemaría	328/22	pepemontaron@gmail.com
Plotek, Magalí	1535/21	magaliplotek+qva@gmail.com
Pivato, Santiago Ezequiel	426/22	santiagopivato@gmail.com

El objetivo de este trabajo es realizar un reconocedor de imágenes. En concreto, vamos a estar reconociendo imágenes de prendas de ropa, obtenidas del conjunto de datos Fashion Mnist. Para realizarlo, implementamos dos métodos muy importantes en el área de aprendizaje automático: el análisis de componentes principales y la clasificación por los k-vecinos más cercanos. Mediante una exploración de hiper parámetros con cross-validation se ajustaron ambos métodos para obtener cerca de un 90 % de precisión en las predicciones del modelo.

Palabras clave: *aprendizaje automático, KNN, PCA, reconocimiento de imágenes*

1. Introducción

Para empezar, nos presentan el conjunto de datos Fashion Mnist. El mismo está compuesto por 5500 imágenes en escala de grises de 28x28 píxeles, que representan prendas de ropa de 10 clases distintas. A continuación se puede obsevar una de las imagenes del dataset clasificada como T-shirt/top.

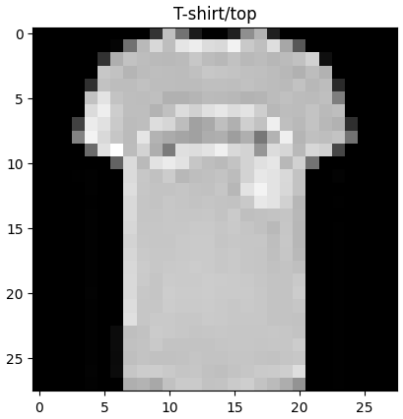

Figura 1: Imagen clase "T-shirt/top"

Para trabajar matricialmente vamos a 'aplanar' las imágenes de forma tal que cada una quede representada como un vector en $\mathbb{R}^{784}$. Para entrenar y evaluar el modelo vamos a particionar los datos en dos conjuntos, uno de entrenamiento y otro de prueba. Utilizaremos 5000 imagenes de entrenamiento, por lo que definimos $X_{train} \in \mathbb{R}^{5000\times784}$ como la matirz de datos de entrenamiento, donde cada fila representa una imagen. De la misma forma definimos $X_{test} \in \mathbb{R}^{500\times784}$ la matriz con los datos con los que evaluaremos nuestro modelo en la última instancia.

La idea del trabajo es desarrollar un algoritmo de *machine learning* que reconozca y clasifique los elementos de X_{test} a partir de los k elementos más "parecidos" de X_{train}. A este método se lo conoce como clasificación KNN (de sus siglas en inglés, *K-Nearest Neighbours*).

El método de KNN consiste en tomar los datos de entrenamiento, que en nuestro caso son imágenes cuya clase conocemos, aplanarlos en un vector y pensarlos como puntos en el espacio. Luego, cuando se presenta un dato de prueba que se quiere clasificar se le toma la *distancia coseno centrada* con cada uno de los datos de entrenamiento. Se arma un vector con las distancias, para ordenarlo de menor a mayor. Una vez hecho esto, tomamos los k primeros datos de este vector y vemos cuál clase es la mas frecuente entre estos k elementos. Finalmente, esa clase le es asignada al dato de prueba y queda clasificado. La distancia coseno entre dos vectores se obtiene de la siguiente manera:

$$D_{coseno}(x,y) = 1 - \frac{(x^t - \mu_x)(y - \mu_y)}{||x^t - \mu_x||_2||y - \mu_y||_2} \tag{1}$$

Debido a que tenemos un conjunto de datos con 784 dimensiones cada uno, ejecutar KNN puede resultar costoso computacionalmente. Por lo tanto nos interesea obtener una aproximación a la exactitud del algoritmo KNN que sea menos costosa. Para ello utlizamos otro método que se utiliza mucho en el área de aprendizaje automático, el análisis de componentes principales, también conocido como PCA por sus siglas en inglés. La idea es reducir la dimensionalidad

de los datos con los que se trabaja. Esto se logra aplicando un cambio de base ligado a la covarianza entre los atributos, de manera tal que nos quedamos con las p componentes más representativas de la matirz, sin una pérdida muy grande de precisión de los datos. De esta forma nos quedamos con datos de dimensión $p << 784$ y logramos reducir el costo de ejecutar KNN sobre nuestras matrices. Para obtener la matriz de cambio de base primero definimos la matriz de covarianza C como:

$$X = \begin{bmatrix} x_1 - \mu_{x_1} & x_2 - \mu_{x_2} & \dots & x_m - \mu_{x_m} \end{bmatrix} \qquad C = \frac{X^t X}{n-1} \tag{2}$$

Donde X es una matriz con sus columnas 'centradas', es decir vectores columnas a los que se les resta su media, y n es la dimensión de las columnas. En nuestro caso vamos a tomar como x_i a las columnas de las matrices de entrenamiento (más adelante se verá porqué hay más de una). Una vez calculada la matriz de covarianza, obtenemos una diagonalización de la misma. Como es una matriz simétrica, la misma será de la forma:

$$C = VDV^t \tag{3}$$

Con $V \in \mathbb{R}^{784\times784}$ ortogonal y $D \in \mathbb{R}^{784\times784}$ diagonal positiva. D es una matriz que en su diagonal posee los autovalores de la matriz de covarianza C ordenados descendentemente, mientras que V es la que posee como columnas los autovectores asociados a cada autovalor respectivamente. Los autovalores se pueden interpretar como la varianza explicada por cada autovector. Por lo tanto si los primeros p autovalores en D acumulan un gran porcentaje de la varianza total, entonces con una combinación lineal de los primeros p autovectores podemos obtener una aproximación cercana a la matriz de covarianza original. En otras palabras, las primeras p componentes son más importantes que el resto, pues son las que más peso tienen la estructura de la matriz. Por todo esto, definimos a nuestra matriz de cambio de base como:

$$\hat{V} = \begin{bmatrix} v_1 & v_2 & \dots & v_p \end{bmatrix} \qquad \hat{X} = X\hat{V} \tag{4}$$

Con v_i las primeras p columnas de la matriz V. Así logramos computar $\hat{X}$ una matriz de p columnas que contiene la información más relevante de cada dato.

Como obtener autovalores y autovectores es una tarea costosa, sería interesante poder implementar algún método que lo haga de manera eficaz. La función que utilizaremos para esto último, entonces, será el método de la potencia.

Con el objetivo de verificar nuestra implementación del método de la potencia con deflación, vamos a querer correr el algoritmo sobre matrices cuyos autovalores conozcamos de antemano. Las matrices diagonales tienen la propiedad de que sus autovalores son los elementos que se encuentran en la diagonal. Como son demasiado triviales, introduciremos el truco de la matriz de Householder para a partir de una matriz diagonal, obtener una matriz semejante que no sea tan trivial. El truco consiste en, dada $D \in \mathbb{R}^{n\times n}$ diagonal, elegir un vector $u \in \mathbb{R}^n$ arbitrario para generar $H \in \mathbb{R}^{n\times n}$ matriz de Householder, y definir la nueva matriz M como:

$$M = HDH^t \qquad con \qquad H = \mathbb{I} - 2uu^t \tag{5}$$

Esta nueva matriz M, tendrá los mismos autovalores que D por ser semejantes.

El método de la potencia junto con el de deflación nos permiten de manera iterativa obtener en cada paso una aproximación, bastante precisa, del autovalor de mayor módulo de una matriz junto a su autovector correspondiente. Si lo implementamos de manera eficiente, entonces la descompocisión necesaria para PCA se puede lograr en un tiempo razonable.

Para evitar sesgos al momento de medir precisión, no trabajaremos con X_{test} hasta el final del trabajo. ¿Por qué? Porque si al realizar la exploración del hiperparámetro k y p se le presenta de entrada los datos de X_{test}, la exactitud que devuelva con esos p y k será en base a estos datos; lo que puede ser perjudicial para experimentos futuros con datos no vistos. Lo que haremos entonces, será trabajar solo con los datos de X_{train} con la metodología 5-folding de validación cruzada.

La validación cruzada con 5-folding consiste en hacer 5 particiones de X_{train}: 4 que sigan siendo de entrenamiento a la cual llamaremos $X_{newtrain}$ y 1 que sea de desarrollo que será llamada X_{dev}. Con estas particiones hechas, se realiza

una busqueda de la mejor combinación de p componentes y k vecinos que maximicen la exactitud del reconocimiento. Estas particiones se van rotando, para que cada una de ellas tome el papel de X_{dev} exactamente una vez. Finalmente, se obtiene la mejor combinación de p y k que maximice la exactitud promedio de todas las iteraciones, para que en última instancia, se pruebe la precisión del modelo con los datos de X_{test}. En resumen, siempre que se esté buscando un parámetro maximizando algún valor, se hará con la validación cruzada. X_{test} será utilizado únicamente para la verificación del modelo.

Entonces, nuestro plan de acción a seguir para este trabajo es el siguiente: Primero, implementar el método de KNN con Python, utilizando funciones de NumPy para optimizarlo. Luego, implementar el método de la potencia para obtener autovalores y autovectores de una matriz y analizar cuán preciso es. Esto lo realizaremos en C++ puesto que Python resulta muy poco eficiente para este caso. Finalmente, implementar el método de PCA y de validación cruzada y realizar una exploración de los hiperparámetros k y p para verificar con los datos de prueba.

2. Desarrollo

Como primera parte del experimento, tuvimos que desarrollar el algoritmo KNN en Pyhton. Lo primero que hicimos fue tratar de implementarlo con ciclos y luego optimizarlo con funciones de Numpy. En ambos casos, usamos la distancia coseno para hallar a los k vecinos más cercanos. La implementación en Python consistía en calcular para cada dato de desarrollo, las distancias coseno a cada dato de entrenamiento, para luego ordenarlas y así obtener las menores k distancias. Estas k distancias tienen asociadas la clase de prenda del dato que representan. Por lo tanto lo siguiente que hicimos fue calcular entre estos k datos cuál es la clase de prenda más frecuente. Si la prenda más frecuente coincide con la clase del dato de desarrollo, decimos que KNN acertó su predicción. Finalmente obtenemos la exactitud como:

$$exactitud = \frac{\#\text{aciertos}}{\#\text{datos de desarrollo}} \tag{6}$$

Si bien este procedimiento es intuitivo y sencillo de implementar, en la práctica resulta extremadamente ineficiente debido al overhead que tienen los *for loops* de Python. Sabiendo que la biblioteca Numpy está optimizada para el cómputo de cálculos matriciales, planteamos el problema en forma matricial:

$$A = \begin{bmatrix} \frac{x^t_{dev_1} - \mu_{x_{dev_1}}}{||x^t_{dev_1} - \mu_{x_{dev_1}}||_2} \\ \frac{x^t_{dev_2} - \mu_{x_{dev_2}}}{||x^t_{dev_2} - \mu_{x_{dev_2}}||_2} \\ \vdots \\ \frac{x^t_{dev_{1000}} - \mu_{x_{dev_{1000}}}}{||x^t_{dev_{1000}} - \mu_{x_{dev_{1000}}}||_2} \end{bmatrix} \text{B} = \begin{bmatrix} \frac{x_{train_1} - \mu_{x_{train_1}}}{||x_{train_1} - \mu_{x_{train_1}}||_2} & \frac{x_{train_2} - \mu_{x_{train_2}}}{||x_{train_2} - \mu_{x_{train_2}}||_2} & \cdots & \frac{x_{train_{4000}} - \mu_{x_{train_{4000}}}}{||x_{train_{4000}} - \mu_{x_{train_{4000}}}||_2} \end{bmatrix}$$

$$distancias = \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} - AB \tag{7}$$

$$(distancias)_{ij} = 1 - (A)^t_i (B)_j = 1 - \frac{(x^t_{dev_i} - \mu_{x_{dev_i}})(x_{train_j} - \mu_{x_{train_j}})}{||x^t_{dev_i} - \mu_{x_{dev_i}}||_2 ||x_{train_j} - \mu_{x_{train_j}}||_2} = D_{coseno}(x_{dev_i}, x_{train_j})$$

Una vez que tenemos la matriz de distancias, la ordenamos ascendentemente por filas, y nos quedamos con los índices de los primeros k elementos de las filas ordenadas. Con estos índices podemos obtener las clases de prenda de los k elementos más cercanos a cada dato de desarrollo, y calculando la moda de cada fila de clases, obtenemos la clase más frecuente para cada dato de desarrollo. Si la moda de las clases coincide con la clase del dato, entonces decimos que KNN acertó la predicción. Por último usamos este criterio para contar aciertos y calcular la exactitud de la misma forma que en la implementación anterior.

Algorithm 1 KNN

KNN(**in** $X_{dev} : \mathbb{R}^{m \times n}$,**in** $X_{train} : \mathbb{R}^{h \times n}$, **in** $y_{dev} : \mathbb{R}^m$, **in** $y_{train} : \mathbb{R}^h$, **in** $k : \mathbb{N}$) $\rightarrow res : \mathbb{R}$

$A = X_{dev}$ - rowMeans$(X_{dev})*(1, ..., 1)^t$ ▷ $(1, ..., 1) \in \mathbb{R}^n$
$A =$ normalizeRows(A)
$B = X_{train}$ - rowMeans$(X_{train})*(1, ..., 1)^t$ ▷ $(1, ..., 1) \in \mathbb{R}^n$
$B =$ normalizeRows(B)
$distancias = (1, ..., 1) * (1, ..., 1)^t$ - $A * B^t$ ▷ $(1, ..., 1) \in \mathbb{R}^m, (1, ..., 1)^t \in \mathbb{R}^{1 \times h}$
$indiceOrdenados =$ argSortRows$(distancias)$
$indicesMasCercanos = indicesOrdenados$[:,0:$k$] ▷ Indexación tipo Python
$clasesMasCercanas = y_{train}[indicesMasCercanos]$ ▷ Indexación tipo Numpy 'Avanzado'
$modas =$ rowModes$(clasesMasCercanas)$
$aciertos =$ logicalAnd$(modas, y_{dev}\ )$
$res =$ countTrue$(aciertos)$ / count(y_{dev})

Como segunda parte, implementamos el método de la potencia con deflación en C + +. Para lograrlo hicimos uso de la biblioteca Eigen. El método de la potencia es un método iterativo para calcular el autovalor de mayor módulo de la matriz. El algoritmo comienza tomando un vector random v_0, el cual será nuestra aproximación inicial. Luego, se entra a un ciclo de n iteraciones, ese n es fijo y se pasa por parámetro. Dentro del ciclo, se calcula una nueva aproximación al autovector. Si la norma infinito de la diferencia entre el nuevo autovector y el su predecesor es menor a la tolerancia dada, podemos considerar que ya convergió a una solución lo suficientemente buena. Para testear nuestro código, realizamos múltiples casos de tests generando matrices a partir del truco de Householder, para así conocer sus autovalores de antemano y poder comparar con lo que devuelve nuestra implementación.

Algorithm 2 Método de la potencia

Método de la potencia(**in** $X : \mathbb{R}^{n \times n}$, **in** $CantIteraciones : \mathbb{N}$, **in** $Tolerancia : \mathbb{R}$) $\rightarrow autovalor : \mathbb{R}, autovector : \mathbb{R}^n$

$m = X[0].size()$
$v_1 = random(m)$
v_0
for $i = 0, \ldots, CantIteraciones$ **do**
  $v_0 = v_1$
  $v_1 = X * v_1$
  $v_1 = v_1 / v_1.norm_2$
  if $(v_1 - v_0).norm_\infty < Tolerancia$ **then**
    print("Salió por tolerancia")
    break
  end if
end for
autovalor $= v_1.transpose() * X * v_1 / (v_1.norm_2)^2$
print("Salió por iteraciones")
return autovalor, v_1

Una vez realizada la experimentación con el método de la potencia, pasamos a implementar el método de la potencia con deflación. Esta variante se utiliza para conseguir todos los autovalores de la matriz, no solo el autovalor dominante. Veamos que si tenemos una matriz $B \in \mathbb{R}^{n \times n}$ cuyos autovalores sean tales que $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \cdots \geq |\lambda_n|$ y cuyos autovectores formen una base ortonormal. Entonces la matriz $B - \lambda_1 v_1 v_1^t$ tiene como autovalores asociados los mismos que B, salvo λ_1 que es remplazado por 0.

$$(B - \lambda_1 v_1 v_1^t) v_1 = B v_1 - \lambda_1 v_1 v_1^t v_1 = \lambda_1 v_1 - \lambda_1 v_1 = 0 \tag{8}$$

$$(B - \lambda_1 v_1 v_1^t) v_i = B v_i - \lambda_1 v_1 v_1^t v_i = \lambda_i v_i - 0 = \lambda_i v_i \tag{9}$$

Por lo tanto, si aplicamos el método de la potencia sobre $B - \lambda_1 v_1 v_1^t$ el siguiente par autovalor y autovector que se calculará será el asociado a λ_2. Si aplicamos está estrategia sucesivamente, podemos calcular los n autovalores de B.

Algorithm 3 Deflación

Deflación(**in** $X: \mathbb{R}^{n\times n}$, **in** $CantAutovalores: \mathbb{N}$, **in** $CantIteraciones: \mathbb{N}$, **in** $Tolerancia: \mathbb{R}$) $\rightarrow$ $autovalores : \mathbb{R}^n, autovectores : \mathbb{R}^{n\times n}$

$n = X.size()$
$m = X[0].size()$
$autovalores = zero(CantAutovalores)$
$autovectores = zero(m, CantAutovalores)$
for $i = 0, \ldots, CantAutovalores$ **do**
  $autovalor_i, autovector_i$ = Método de la Potencia(X, CantIteraciones, Tolerancia)
  $autovalores[i] = autovalor_i$
  $autovectores[i] = autovector_i$
  $X = X - autovalor_i * autovector_i * autovector_i.transpose()$
end for
return autovalores, autovectores

Lo siguiente que hicimos una vez que nos convencimos de la correctitud de nuestra implementación fue estudiar la convergencia del método de la potencia. Para eso, corrimos el método de la potencia con deflación con 50 matrices aleatorias para cada ϵ generadas a partir de estos cinco autovalores 10, 10 - ϵ, 5, 2, 1 y el truco de Householder. Tomamos como tolerancia 10^{-7} y varios ϵ en un rango de 10^{-4} a 1 y estudiamos cuantas iteraciones necesitaba para converger en cada caso y cual era el error al final de la convergencia. A raíz de esos datos obtuvimos los siguientes gráficos:

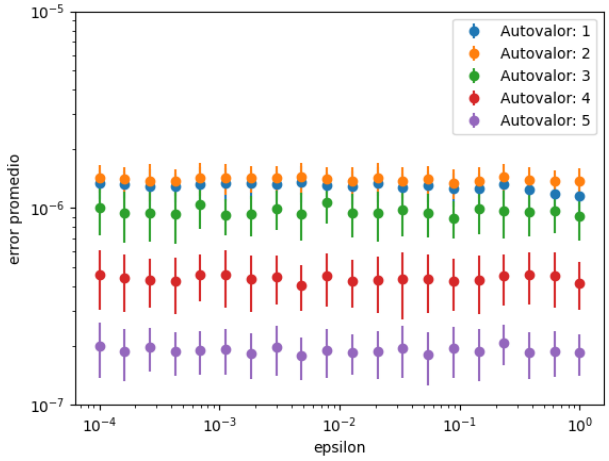

Figura 2: Media de errores por ϵ

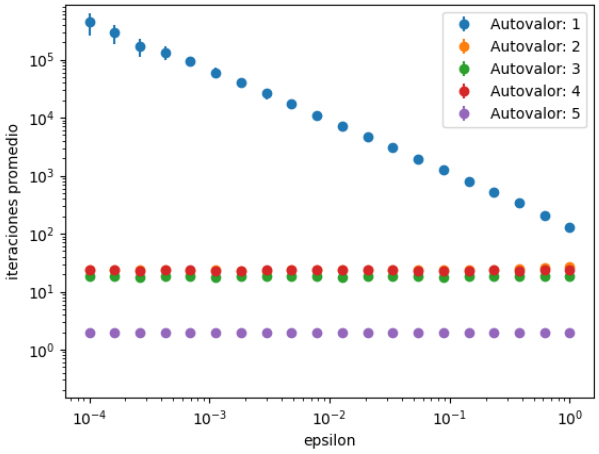

Figura 3: Media de iteraciones por ϵ

Para la figura 2, vemos que el error es similar para todos los autovalores y ronda entre 10^{-6} y 10^{-7}. Esto es coherente con la tolerancia que utilizamos, de haber usado una tolerancia más estricta habríamos tenido un error menor probablemente. Parece ser independiente del ϵ el error.

Para la figura 3, vemos que la cantidad de iteraciones para todos los autovalores, salvo el primero, es constante. Sabemos que la velocidad de convergencia en el método de la potencia está dada por $|\frac{\lambda_2}{\lambda_1}|$, lo que significa que cuando algún otro autovalor tiene módulo aproximado al autovalor dominante el método converge despacio, mientras que si están alejados el método converge más rápido. Este fenómeno se ve claramente reflejado en el autovalor 1. La cantidad de iteraciones necesarias para converger decrece a medida que el epsilon crece, es decir la velocidad de convergencia auementa a medida que la distancia entre 10 y $10 - \epsilon$ aumenta. Para los demás autovalores, $|\frac{\lambda_2}{\lambda_1}|$ es constante y es independiente del ϵ.

Como tercera parte, realizamos un reconocedor de imagenes utilizando 5-fold cross-validation. Elegimos dividir nuestro conjunto de entrenamiento en cinco partes de mil. Como los datos se encuentran ordenados e intercalados sabemos que vamos a tener la misma cantidad de prendas de cada tipo en cada partición. Procedemos a iterar sobre cada una de esas partes, tomando una parte distinta para desarrollo en cada iteración y las cuatro partes restantes

para entrenar. Medimos el desempeño de cada k en cada una de las partes, y calculamos la exactitud promedio para cada k. Obtuvimos los siguientes resultados:

k	1	2	3	4	5	6	7	8	9	10
exactitud	0.8745	0.8617	0.8670	0.8672	0.8694	0.8678	0.8710	0.8684	0.8702	0.8680

Cuadro 1: Exploración del parámetro k

La mejor exactitud promedio es 0.8745 y se da con $k = 1$, sin embargo con los otros k no obtuvimos valores muy distintos. Por lo tanto observamos que para nuestro conjunto de datos el algoritmo KNN se comporta de manera similar para distintos k.

Luego de eso, implementamos el PCA. Este algoritmo devuelve una matriz con los autovectores y un vector con los autovalores de la matriz de covarianza de los datos a los cuales les queremos aplicar el cambio de base. Sabemos que podemos aplicar el método de la potencia con deflación sobre la matriz de covarianza ya que es una matriz simétrica y en reales.[1].

Algorithm 4 PCA

PCA(**in** $X : \mathbb{R}^{m \times n}$, **in** $p : \mathbb{N}$) $\rightarrow autovalores : \mathbb{R}^n, autovectores : \mathbb{R}^{n \times n}$

$n = X.size()$
//centro en función de las columnas
$X_{centrada} = X - X.mean(cols)$
$C = X_{centrada}.transpose()$ @ $X_{centrada}/(n-1)$
$autovalores, autovectores = Deflacion(C, p, 20000, 1e-9)$
return autovalores, autovectores

Las primeras p columnas de V son las primeras p componentes principales, y estas se encuentran ordenadas según su varianza. Una vez tenemos V y un p, con cuantas componentes principales queremos trabajar, al aplicar $\hat{X} = XV$ obtenemos el conjunto de datos pero con dimensionalidad reducida. Para saber cuantas p columnas hay que usar, tuvimos que realizar varios experimentos, en los cuales fuimos probando con cuántos p obtenemos un resultado lo suficientemente bueno. El primer paso fue correr PCA sobre X_{train} para obtener los 784 autovalores que representan la varianza de las componentes. Una vez computados, los comparamos mediante los siguientes gráficos:

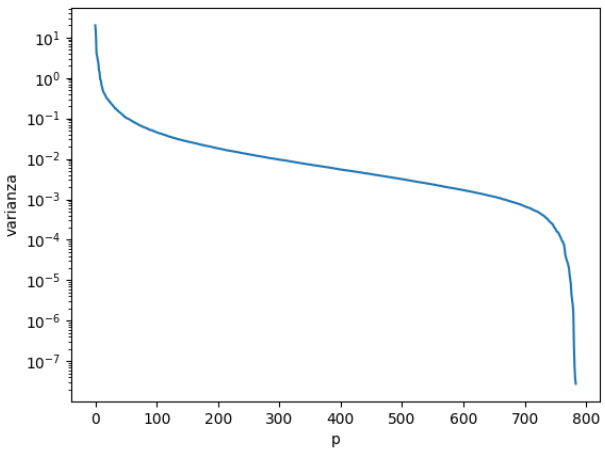

Figura 4: Varianza por componente principal

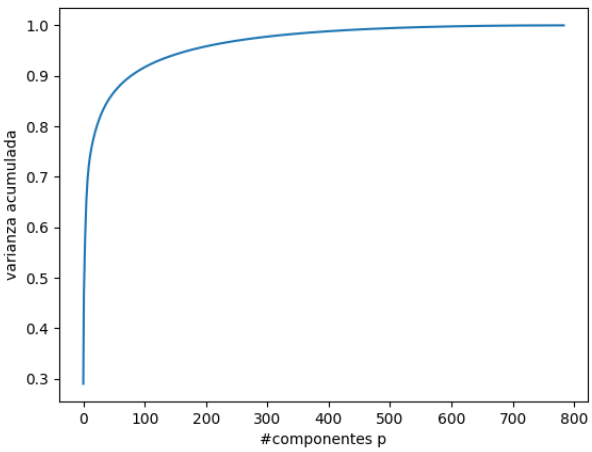

Figura 5: Varianza acumulada

Lo primero que observamos en la figura 4 es que los autovalores están ordenados, como era de esperar pues se obtuvieron mediante deflación. El autovalor dominante representa una varianza de 19,87, valor que decrece rápidamente hasta llegar a 0,89 en el décimo autovalor. Se observa la gran diferencia que hay entre la varianza de los autovalores más

[1]Richard L. Burden, J. Douglas Faires. (2002). *Análisis numérico. 'Capítulo 9. Corolario 9.11'*. International Thomson Editores

dominantes y el resto, lo que nos indica que quizás con una matriz de cambio de base $\hat{V}$ de pocas dimensiones podemos lograr buenos resultados. Aún mejor se puede ver esto último cuándo observamos la figura 5. La curva del gráfico nos muestra que proporción de la varianza total se obtiene al usar las primeras p componentes. Con 25 componentes obtenemos un 80 % de la varianza total, con 80 componentes un 90 %, con 175 un 95 %, y con 450 un 99 %. De esta forma, podemos confirmar que reducir la dimensionalidad de la matriz $\hat{V}$, y por ende de las $\hat{X}$ de entrenamiento, desarrollo y prueba, nos permite bajar el costo de los cómputos sin perder demasiada precisión a cambio, pues tenemos el 80, 90, 95, etc. porciento de la información concentrada en menos columnas.

Como último paso del experimento, realizamos una exploración en conjunto del parámetro k y p. Tratamos de hallar la combinación de ambos que devuelva la mayor precisión. Para esto, mantenemos las mismas divisiones de nuestro espacio de prueba que utilizamos anteriormente. Luego, medimos la exactitud de cada una de las partes para cada par p y k. Finalmente, vemos qué par tuvo la mejor exactitud en promedio. Esos p y k son los mejores, por lo que procedemos a entrenar nuestro modelo con esos valores y todos los datos y procesar los datos de prueba.

Algorithm 5 Validación cruzada

Validación cruzada(**in** $X_{train} : \mathbb{R}^{m \times n}$, **in** $y_{train} : \mathbb{R}^m$, **in** $Valores_k : \mathbb{N}^k$, **in** $Valores_p : \mathbb{N}^p$) $\rightarrow k : \mathbb{N}, p : \mathbb{N}, exactitud : \mathbb{R}$

$cant_k = Valores_k.size()$
$cant_p = Valores_p.size()$
$exactitudes = zeros(cant_p, cant_k)$
for $i = 0, \ldots, 5$ **do**
  $X_{dev} = X_{train}[i * 1000 :: (i+1) * 1000]$
  $X_{newtrain} = \text{concatenate}(X_{train}[0 :: i * 1000], X_{train}[(i+1) * 1000 :: 5000])$
  $y_{dev} = y_{train}[i * 1000 :: (i+1) * 1000]$
  $y_{newtrain} = \text{concatenate}(y_{train}[0 :: i * 1000], y_{train}[(i+1) * 1000 :: 5000])$
  $X_{devCentrada} = X_{dev} - X_{dev}.mean(cols)$
  $X_{newtrainCentrada} = X_{newtrain} - X_{newtrain}.mean(cols)$
  for $indice_p = 0, \ldots, cant_p$ **do**
    $autovalores, V = pca(X_{train}, valores_p[indice_p])$
    $p = valores_p[indice_p]$
    $X_{devHat} = X_{devCentrada}$ @ $V[cols, 0 : p]$
    $X_{newtrainHat} = X_{newtrainCentrada}$ @ $V[cols, 0 : p]$
    for $indice_k = 0, \ldots, cant_k$ **do**
      $k = valores_k[indice_k]$
      $exactitudes[indice_p][indice_k] += \text{KNN}(X_{devHat}, X_{newtrainHat}, y_{dev}, y_{newtrain}, k)$
    end for
  end for
end for
$exactitudes /= 5$
$mejorIndice_p, mejorIndice_k = argsmax_{matriz}(exactitudes)$
$mejor_k = valores_k[mejorIndice_k]$
$mejor_p = valores_p[mejorIndice_p]$
$mejor_{exactitud} = exactitudes[mejorIndice_p][mejorIndice_k]$
return $mejor_k, mejor_p, mejor_{exactitud}$

Hicimos esta exploración para k entre 1 y 20, y para 10 valores de p distintos. No tuvimos en cuenta $p = 1$ ya que pierde sentido la noción de distancia coseno para matrices de una sola columna. Por lo tanto tomamos 5 valores de p menores iguales que 10, y los otros 5 como aquellos que representan el 80, 90, 95, 99 y 100 % de la varianza total resepctivamente. Obtuvimos los siguientes resultados:

k / p	2	3	4	5	10	25	80	175	450	784
1	0.7994	0.8415	0.8492	0.8577	0.8718	0.8760	0.8741	0.8747	0.8747	0.8747
2	0.6748	0.7954	0.8154	0.8331	0.8570	0.8618	0.8597	0.8608	0.8612	0.8617
3	0.6702	0.7884	0.8228	0.8419	0.8632	0.8677	0.8677	0.8672	0.8664	0.8667
4	0.6719	0.8032	0.8346	0.8474	0.8650	0.8676	0.8662	0.8672	0.8676	0.8672
5	0.7208	0.8116	0.8440	0.8512	0.8695	0.8695	0.8688	0.8702	0.8692	0.8694
6	0.6305	0.8211	0.8428	0.8554	0.8700	0.8662	0.8674	0.8680	0.8674	0.8678
7	0.6762	0.8206	0.8470	0.8568	0.8703	0.8702	0.8688	0.8695	0.8702	0.8710
8	0.6858	0.8240	0.8478	0.8562	0.8702	0.8698	0.8671	0.8665	0.8680	0.8680
9	0.7696	0.8256	0.8498	0.8559	0.8715	0.8686	0.8692	0.8686	0.8694	0.8701
10	0.7590	0.8286	0.8495	0.8572	0.8697	0.8686	0.8692	0.8674	0.8674	0.8677
11	0.7460	0.8301	0.8497	0.8565	0.8715	0.8698	0.8699	0.8702	0.8686	0.8692
12	0.8080	0.8298	0.8490	0.8588	0.8714	0.8699	0.8704	0.8688	0.8699	0.8694
13	0.7882	0.8318	0.8484	0.8591	0.8725	0.8709	0.8705	0.8698	0.8699	0.8708
14	0.8086	0.8306	0.8502	0.8603	0.8715	0.8702	0.8702	0.8694	0.8692	0.8694
15	0.7866	0.8308	0.8492	0.8600	0.8737	0.8702	0.8718	0.8702	0.8695	0.8695
16	0.7706	0.8304	0.8485	0.8596	0.8737	0.8698	0.8704	0.8699	0.8690	0.8698
17	0.7906	0.8303	0.8488	0.8606	0.8724	0.8702	0.8715	0.8710	0.8690	0.8699
18	0.7906	0.8295	0.8488	0.8596	0.8730	0.8698	0.8728	0.8706	0.8694	0.8695
19	0.7670	0.8303	0.8488	0.8600	0.8728	0.8695	0.8718	0.8704	0.8699	0.8698
20	0.7726	0.8298	0.8488	0.8597	0.8732	0.8698	0.8716	0.8698	0.8695	0.8702

Cuadro 2: Exploración de hiperparámetros p (columnas) y k (filas)

Nos quedamos con los valores $p = 25$ y $k = 1$, pues fueron los que mejor exactitud devolvieron durante la validación cruzada, con una exactitud $exa = 0{,}876$. Finalmente, pasamos a probar el modelo con nuestros parámetros óptimos ya encontrados, esta vez sí usando los datos de X_{test}. Luego de correr PCA y KNN con nuestros datos, obtuvimos una exactitud final $exa = 0{,}888$. Por lo tanto, decimos que nuestro modelo logró reconocer la clase de cerca de un 90 % de las imágenes de prueba en X_{test} a partir de un entrenamiento y exploración de hiperparámetros con la matriz X_{train}.

3. Conclusiones

- El análisis de componentes principales es costoso ya que requiere calcular autovectores y autovalores. Por eso mismo, es necesario implementar cada parte del método de manera eficiente, principalmente la parte de calcular autovalores y autovectores.
- El método de la potencia es realmente muy útil. Aproxima tanto autovalores como autovectores y permite tener control sobre la precisión de la aproximación.
- A su vez, Python resulta muy ineficiente por sí solo. En general, siempre resulta más útil utilizar NumPy, y para casos que requieran mucha más velocidad, considerar un lenguaje que sea compilado, como el caso del método de la potencia con C++.
- En los procesos de aprendizaje automático es de suma importancia conocer los datos con los que uno está trabajando y asignarles roles claros y definidos. Sobre todo, es esencial separar los datos de prueba de los de entrenamiento y no utilizarlos hasta el final para evitar condicionar el modelo.
- También conocer el conjunto de datos con el que uno trabaja permite suponer que no se requieren todos los atributos de los datos para trabajar con ellos. En estos casos, es conveniente realizar un análisis de componentes principales y en definitiva ver cómo se encuentra distribuida la varianza entre las componentes.

4. Referencias

1. Richard L. Burden, J. Douglas Faires. (2002). *Análisis numérico. 'Capítulo 9. Sección 1. Corolario 9.11 p.555'.* International Thomson Editores.