Metodos Numericos

Trabajo Práctico 2

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

Trabajo Práctico 2

Analisis de componentes principales y procesamiento de lenguaje

Integrante	LU	Correo electrónico
Sanchez, Facundo	97/22	facundosanchez636@gmail.com.ar
Esposito, Camilo	725/22	espositocamilo@gmail.com
Sessarego, Santiago	693/22	ssessarego72@gmail.com

Índice

1. Resumen

En este informe se implementaron algoritmos de KNN (k - nearest neighbors), Método de la Potencia con Deflación, y Análisis de Componentes Principales (PCA). Con las mismas, se desarrolló un clasificador de películas, el cual se buscó mejorar mediante la exploración de los hiper-parámetros k de KNN y p de PCA. Esto nos lleva a una versión del clasificador que transforma los datos para trabajar con matrices de menor tamaño. Gracias a esto, podemos tener un clasificador de películas que a costo de tener que preprocesar los datos, puede realizar predicciones suficientemente buenas con datos más pequeños.

2. Introducción Teórica

El objetivo del siguiente informe será resolver un problema de clasificación en el que partiremos de un conjunto de datos de películas las cuales podrán pertenecer a 4 géneros posibles: *western*, *ciencia ficción*, *romance*, o *crimen*, y mediante su descripción trataremos de diseñar un algoritmo de *machine learning* el cual pueda deducir a qué género pertenecen un conjunto de películas, a partir de otro conjunto más grande de *entrenamiento* del cual "aprenderá" a distinguir géneros de películas.

Los datos sobre los que trabajaremos será una matriz $X \in R^{m \times 67462}$, donde cada fila representará a cada una de las m películas junto con la cantidad de apariciones de cada uno de los 67462 *tokens* que estas comparten entre sí. De esta forma, podemos representar a cada película como un vector. Para armar el clasificador de géneros usaremos la técnica de aprendizaje KNN (K - Nearest Neighbors), y exploraremos nuestro conjunto de datos de entrenamiento buscando el mejor valor de k para KNN mediante la técnica de 4-*fold cross-validation*.

Luego, nos interesará reducir la dimensionalidad de los datos mediante el *Análisis de Componentes Principales* (PCA). Lo que haremos será buscar el mejor valor de p con el cual se mantenga a lo sumo el 95 % de la varianza de los datos y que los explique mejor. De esta forma, podremos utilizar la base formada por las componentes principales aprendidas para proyectar otro conjunto de datos. Para implementar nuestro PCA, necesitaremos los autovalores y autovectores de la matriz de covarianza de X, lo cual obtendremos mediante la implementación de un algoritmo de *Método de la Potencia con Deflación*.

Por último, luego de hallar los mejores valores de k para KNN y p para PCA, haremos una medición de performance final.

3. Desarrollo

Lo primero que queremos hacer para implementar nuestro clasificador de películas será tener algun método con el cual dado un conjunto de películas cuyos géneros son conocidos, podamos inferir a qué género pertenece otra que el algoritmo desconoce, para lo cual usaremos el método de KNN (k - nearest neighbors). KNN buscará, dado un punto en el espacio R^n, saber a qué clase pertenece viendo cual es la "moda" entre sus k puntos más cercanos, por lo cual necesitaremos representar a nuestro conjunto de películas como puntos en el espacio R^n.

Luego, usaremos el *método de la potencia con deflación* para obtener los autovalores y autovectores de una matriz, así posteriormente podemos hacer nuestras experimentaciones con PCA.

3.1. Implementación KNN (k - nearest neighbors)

Para la implementación de *KNN (k - nearest neighbors)* en lenguaje Python [1] hicimos uso de la librera NumPy [2] para representar vectores y matrices. Luego para la realización de gráficos usaremos la librería MatPlotLib[3].

3.1.1. Implementación

Nuestra implementación tomará como parámetros un valor k entero, una matriz de datos de entrenamiento $E \in R^{a \times n}$ (siendo a la cantidad de datos de entrenamiento), una matriz de datos de prueba $P \in R^{b \times n}$ (siendo b la cantidad

[1] https://www.python.org/
[2] https://numpy.org/
[3] https://matplotlib.org/

de datos de prueba), un vector T_E (donde $T_E[i]$ es la clase del i-esimo dato de entrenamiento), y un vector T_P (donde $T_P[i]$ es la clase del i-esimo dato de prueba). Para clasificar cada dato de prueba $p = P[i]$ / $1 \leq i \leq b$, mediremos la distancia de p a todos los datos $e = E[i]$ / $1 \leq i \leq a$ de entrenamiento y haremos una predicción de la clase de p como la mayoritaria entre los k datos de entrenamiento más cercanos. Finalmente, veremos para cada p si nuestra predicción fue acertada viendo el vector T_P y devolveremos un vector $R \in B^b$, donde en cada posición i / $1 \leq i \leq b$ se indica si nuestra predicción fue acertada para el i-esimo dato de prueba.

Nos guiaremos del siguiente pseudocódigo **(1)**:

(1) K-Nearest Neighbors (KNN)

```
1:  procedure KNN(k, train_data, test_data, train_types, test_types)
2:      asserts ← zeros(test_data.size())
3:      test_normalizado ← test_data / ||test_data||
4:      train_normalizado ← train_data / ||train_data||
5:      distancias ← 1 − (test_normalizado · train_normalizado^t)
6:      k_cercanos ← train_types[argsort(distancias)[:, :k]]
7:      predicciones ← [ ]
8:      para vecinos en k_cercanos hacer
9:          unicos, cantidad ← unique(vecinos)
10:         moda ← unicos[argmax(cantidad)]
11:         predicciones.append(moda)
12:     asserts ← predicciones == test_types
13:     devolver asserts
```

Para medir la distancia entre datos usaremos la *distancia coseno*:

$$\text{dist\_cos}(\mathbf{A}, \mathbf{B}) = 1 - \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

3.1.2. Verificación de la Implementación

Para realizar este clasificador de películas, vamos a contar con un conjunto de datos de 400 películas distribuidas de la siguientes manera: 100 del género *western*, 100 de *science fiction*, 100 de *romance*, y 100 de *crimen*. De estos datos, el 80 % será utilizado como datos de *entrenamiento*, y el restante 20 % como datos de *prueba*, con los cuales mediremos la performance.

Primero procesamos estos datos para extraer de las descripciones de todas las películas los *tokens* (palabras claves mas repetidas), lo cual nos dio en total 67462 tokens. Como trabajar con esta cantidad de datos puede ser muy costoso, vamos usar de este conjunto solo los Q tokens mas repetidos. Entonces, crearemos una matriz $X \in R^{N \times Q}$ con N cantidad de peliculas (que en nuestro caso serán 400) y Q la cantidad de tokens en la descripción de las mismas, donde $X[i][j]$ $(1 \leq i \leq N$, $1 \leq j \leq Q)$ representa la cantidad de repeticiones del token j en la descripción de la película i.

Mediremos la performance del algoritmo KNN tomando solo los Q tokens más frecuentes, para $Q \in \{500, 1000, 5000\}$, para finalmente mostrar los resultados mediante un gráfico de barras. Nuestras expectativas son que a medida que tengamos un Q más grande, el algortimo tenga mayor proporción de aciertos, pero en consecuencia un mayor tiempo de cómputo.

3.2. Implementación Método de la Potencia con Deflación

Para esta implementación en lenguaje C++ [4] hicimos uso de la librería Eigen [5] para representar vectores y matrices. Esta elección de lenguaje es por cuestiones de performance, ya que C++ es más rápido que Python para este tipo de algoritmos. Para poder usar esta implementación desde Python usaremos la interfaz PyBind [6], la cual permite compilar módulos de Python a partir de código de C++.

[4] https://isocpp.org/
[5] https://eigen.tuxfamily.org/
[6] https://github.com/pybind

3.2.1. Implementación

Para la implementación de este algoritmo, tomaremos como parámetros una matriz $A \in R^{n \times n}$, una cantidad it de iteraciones máximas, y una tolerancia t, bajo la precondición de que todos los autovalores λ_i de A son positivos ($|\lambda_1| \geq |\lambda_2| \geq ... \geq |\lambda_n|$) y que los autovectores de A forman una base ortonormal. Esta precondición la vamos a poder asegurar debido a que aplicaremos el método a la matriz de covarianza de PCA (ver 3.3.2), la cual es una matriz simétrica semi-definida positiva, que implica que sus autovalores son todos positivos, y además sus autovectores son ortogonales entre sí, y sabemos que en el caso de PCA no es necesario exigir que los autovalores tengan magnitudes distintas (veremos en los resultados del estudio de la convergencia de este método en la sección 4.3 que incluso si hay dos autovalores muy cercanos o iguales, mediante una cantidad de iteraciones lo suficientemente grande se logra la convergencia, aunque con un error posiblemente mayor al resto).

Finalmente, retornaremos una matriz $A \in R^{n \times n+2}$ donde las primeras n columnas serán las aproximaciones de los n autovectores de A, la $(n+1)$-esima columna será las aproximaciones de los autovalores de A, y la última columna mostrará la cantidad de iteraciones que tomó para cada autovalor conseguir una aproximación hasta que la diferencia entre dos aproximaciones seguidas fue menor a t ($A[i][n+2]$ = cantidad de pasos que costó calcular el i-esimo autovector y autovalor de A).

Nos guiaremos del siguiente pseudocódigo **(2)**:

(2) Método de la Potencia Con Deflacion

```
1:  procedure METODO_DE_LA_POTENCIA_DEF(A, iteraciones, tolerancia)
2:      res ← matrix(rows(A), cols(A) + 2)
3:      para i ← 0 hasta cols(A) − 1 hacer
4:          v ← vector_aleatorio_normalizado(A.rows)
5:          pasos ← 0
6:          para j ← 0 hasta iteraciones − 1 hacer
7:              nuevo_v ← (A · v).normalizado()
8:              pasos ← pasos + 1
9:              si normaInf(v - nuevo_v) < tolerancia entonces
10:                 v ← nuevo_v
11:                 break
12:             v ← nuevo_v
13:         res.col(i) ← v
14:         δ ← (v^T · A · v) / (v^T · v)
15:         A ← A − δ · v · v^T
16:         res(i, cols(A)) ← δ
17:         res(i, cols(A) + 1) ← pasos
        devolver res
```

3.2.2. Verificacion de la Implementacion

Para comprobar la correctitud de nuestra implementación realizaremos pruebas sobre matrices para las cuales conocemos sus autovalores y autovectores. Para generar estas matrices, tomaremos una matriz diagonal D con los autovalores λ_i como elementos de la diagonal y construiremos matrices A_k semejantes a D de la forma $A_k = H_k D H_k^t$ siendo H_k una matriz de Householder. De esta forma, las matrices A_k tendrán los mismos autovalores conocidos λ_i de la matriz D y podremos compararlos con los obtenidos al aplicar nuestra implementación del método de la potencia a cada matriz A_k, esperando que coincidan.

3.2.3. Estudio de la convergencia (Grado de error y cantidad de iteraciones)

Vamos a analizar dos aspectos de la implementación del método de la potencia: la cantidad de pasos que se necesitan para obtener una aproximación adecuada de los autovalores y el error generado por la misma. Para ello tomaremos un parámetro ϵ y generaremos matrices de Householder aleatorias, análogamente al ítem anterior, que tengan los autovalores {10, 10 - ϵ, 5, 2, 1}. Evaluaremos ϵ en un rango de valores desde 10^{-4} hasta 10^0 espaciado logarítmicamente y observaremos cómo se comportan los diferentes autovalores respecto a las medidas planteadas.

Para cada matriz A_ϵ, y cada autovector y autovalor aproximado v_i y λ_i que calculemos, mediremos el error del método como $||Av_i - \lambda_i v_i||_2$. Haremos una cantidad r de repeticiones del cálculo para cada A_ϵ y tomaremos un promedio de cada una de esas repeticiones para calcular el grado de error y las iteraciones requeridas.

Nuestras expectativas son principalmente, que a medida que ϵ sea mas perqueño, y por lo tanto los autovalores 10 y 10 - ϵ se parezcan mas, entonces la velocidad de convergencia del algortimo se reduzca para el calculo del autovalor 10, ya que este autovalor dominante y el segundo mayor son muy parecidos.

3.3. Clasificador de género de películas

Ya con nuestros algoritmos KNN y *método de la potencia con deflación* implementados y testeados, procederemos a experimentar con ellos explorando los valores k de KNN y p de PCA para a lo ultimo realizar un *pipeline final* con los resultados obtenidos.

3.3.1. Exploración del hiperparámetro k (4-fold cross validation)

Volviendo a los explicado en páginas anteriores, vamos a tener un conjunto de datos de películas, los cuales tradujimos a un matriz $X \in R^{N \times Q}$ (siendo N la cantidad de películas y Q la cantidad de tokens que utilizaremos).

Para hallar el mejor valor de k y hacer KNN para clasificar un conjunto de películas, vamos a realizar una experimentación del parámetro k empleado en el algoritmo KNN que indica la cantidad de elementos que se tienen en cuenta a la hora de medir la cercanía y semejanza entre la película que se desea clasificar y las pertenecientes a los datos de entrenamiento. Para explorar el valor de k, emplearemos el procedimiento de *4-fold cross-validation* sobre el conjunto de entrenamiento. Dividiremos este conjunto en 4 partes *(folds)* que tengan la misma proporción de películas de los diferentes géneros existentes, y a continuación las iteraremos cíclicamente donde en cada vuelta tomaremos tres folds que utilizaremos como datos de entrenamiento para entrenar el fold restante y obtener la performance de las predicciones. Realizaremos un promedio de las perfomances de las cuatro iteraciones como estimación final para el rendimiento del modelo con el k específico. De esta manera, compararemos los diferentes rendimientos para diferentes valores de k y así hallar el más adecuado para nuestro modelo.

A su vez, realizaremos esta exploración para diferentes dimensiones Q de tókens usados en la clasificación, para obtener una generalización más amplia de las configuraciones apropiadas para el clasificador.

3.3.2. Exploración del hiperparámetro p (Preprocesamiento con PCA)

Hasta ahora, hemos procesado los datos de entrenamiento mediante KNN y 4-fold cross validation usando los datos de manera normal, solo cambiando la proporción de información que en este caso es el valor Q. Lo que nos interesa hacer ahora será preprocesar los datos de entrenamiento de la matriz X mediante PCA, ya que este es un modelo que aprende a partir de un conjunto de datos de entrenamiento para luego aplicarse a nuevos datos.

PCA buscará cambiar la base de los N vectores de la matriz X, con el objetivo de ordenar las dimensiones en componentes y así explicar los datos de mayor a menor. Mediante esta transformación de los datos, conseguimos disminuir la redundancia, es decir, la covarianza. Lo logramos realizando una descomposición en autovectores y autovalores de C_x, la matriz de covarianza de datos. Esta es simétrica y semidefinida positiva, entonces sus autovalores no son negativos. Al diagonalizarla obtenemos la descomposición $C_x = VDV^t$, donde V será una matriz que contenga los autovectores en las columnas (los componentes principales), y D una matriz diagonal que contiene los autovalores, que representan la varianza de cada nueva dirección dada por los autovectores. Queremos seleccionar los p componentes principales que tengan mayor varianza, ya que esto nos permitirá reducir la dimensionalidad sin perder información significativa. Para hallar el valor p más adecuado, lo que hacemos es ir considerando la varianza acumulada de las primeras componentes, estableciendo un criterio donde buscamos el valor p que alcanza el 95 % de varianza.

Entonces, obtendremos mediante nuestro algoritmo de *método de la potencia con deflación* los autovalores que aparecen en la diagonal de la matriz D y graficaremos cómo va aumentando la varianza acumulada a medida que crece p. Tenemos expectativas de que en algún punto la varianza acumulada dejará de crecer de forma considerable, demostrando que existe un valor p_0 que indica la cantidad de componentes principales que contienen la mayor parte de la varianza.

Durante la experimentación de este punto, en su primera iteración no tuvimos en cuenta la necesidad de centrar los datos, lo cual produjo resultados anómalos. Por eso, posteriormente decidimos centrar los datos de la matriz X (con los datos de entrenamiento) restando a cada columna de la matriz la media de la misma.

Lo que esperamos de todo esto, es que mediante la reducción de la dimensionalidad y de la redundancia, se logre mejorar la expresividad de los datos, facilitando el análisis de estos, ya que concentraremos la información en las direcciones principales.

Nos guiaremos del pseudocódigo (3) para la implementación de PCA.

(3) Análisis de Componentes Principales (PCA)

```
1: procedure PCA(X)
2:     X ← X − X.mean()
3:     X_covarianza ← X^T·X / (X.size()−1)
4:     autovalores ← metodo_de_la_potencia_def(X_covarianza).sorted()
```

3.3.3. Pipeline final

Para esta iteración final, teniendo en consideración los resultados del experimento anterior del preprocesamiento con PCA, lo que haremos será una exploración conjunta de los hiperparámetros k de KNN y p de PCA. Partiremos de los datos de las películas en la matriz X_{train}, trabajando solo sobre los datos de entrenamiento y con $Q = 1000$, y haremos *4-fold cross-validation* para cada combinación de k y p ($1 \leq k \leq 50$, $1 \leq k \leq cant\_filas\_de\_datos\_train$), buscando así la mejor combinación de k y p. Aclaramos que acotamos el rango de evaluación de k por la cantidad de filas del conjunto de entrenamiento ya que si la cantidad de columnas C es menor a la de filas F, entonces a partir de las $C + 1$ son redundantes.

Luego de hallar los mejores k y p posibles, tomaremos ahora sí toda X (ahora incluyendo los datos de test) y entrenaremos con PCA los datos de entrenamiento. Después, preprocesaremos la parte de entrenamiento y de prueba con las p primeras componentes principales obtenidas. Ahora, con estos nuevos datos de entrenamiento y prueba ya preprocesados haremos KNN con k.

Nuestras expectativas serán que la performance obtenida con los mejores valores de k y p sea cercana a la que se obtendrá cuando analicemos toda X con KNN y PCA con estos parámetros obtenidos.

4. Resultados y Discusión

A continuación, mostremos los resultados obtenidos en las experimentaciones planteadas durante el desarrollo del informe, mediante gráficos y conclusiones:

4.1. Verificación de la implementación de KNN

Ejecutamos el algoritmo de KNN implementado para cada X_Q, siendo $Q \in \{500, 1000, 5000\}$ la cantidad de columnas que utilizaremos de la matriz original X que creamos. Los porcentajes de predicciones exitosas dado un $k = 5$ fueron los siguientes:

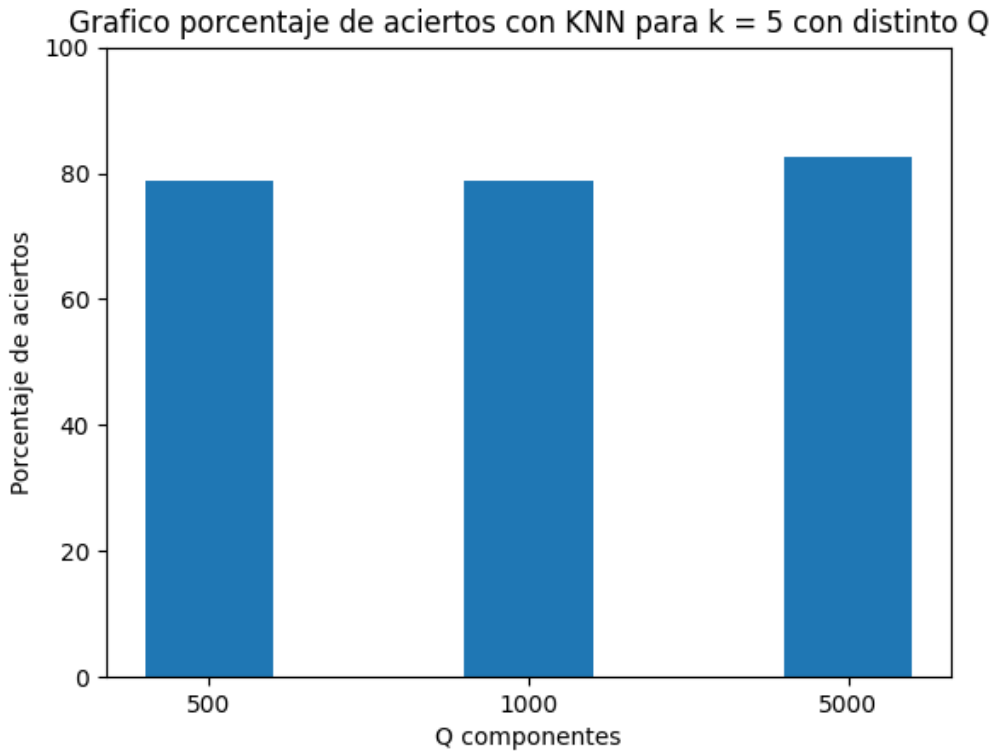

Figura 1: Porcentaje de predicciones existosas para $Q \in \{500, 1000, 5000\}$

Se obtuvieron resultados bastante similares entre cada valor de Q. Para $Q = 500$ se obtuvo un porcentaje de aciertos de %78,75, para $Q = 1000$ también fue de %78,75, y para $Q = 5000$ se obtuvo un %82,5. Lo que podemos concluir es que a medida que tengamos un mayor valor de Q, es decir, teniendo en cuenta una mayor cantidad de tokens para definir las películas, se obtienen mejores resultados.

4.2. Verificación de la implementación de Método de la Potencia

Para este experimento creamos distintas matrices aleatorias usando el truco de la matriz de Householder (ver 3.2.2) $A \in R^{n \times n}$ $(1 \leq n \leq 100)$, todas con autovalores distintos menores entre sí. En cada una se ejecutó el método de la potencia con deflacion (con límite de iteraciones de 10000 y $tolerancia \in \{10^{-2}, 10^{-7}, 10^{-14}\}$), y se midió el error en los autovectores y autovalores obtenidos como el promedio de $||Av_i - \lambda_i v_i||_2$. Estos fueron los resultados obtenidos:

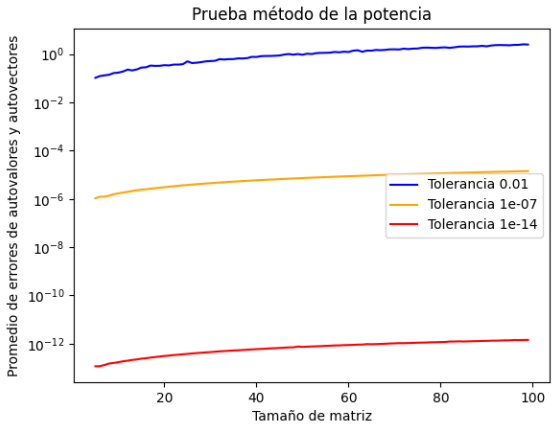

Figura 2: Promedio de errores en los autovectores y autovalores para cada matriz $A \in R^{n \times n}$

Se puede ver cómo nuestra implementación mejora sus resultados mientras se le pase una tolerancia menor, pues se ve que con la tolerancia de 10^{-14} se obtiene menos grado de error entre los resultados esperados y los obtenidos.

4.3. Estudio de la convergencia del Metodo de la Potencia

Para este experimento se vio la convergencia de nuestra implementación del *método de la potencia con deflación* tomando el promedio de los errores y cantidad de iteraciones para cada uno de los 5 autovalores y autovectores de la matriz A_ϵ con autovalores {10, 10 - ϵ, 5, 2, 1}. Como cada matriz A_ϵ se genera mediante autovectores aleatorios, tomaremos el promedio de los errores y las repeticiones 100 veces para cada matriz, y muestrearemos 100 valores de ϵ entre 10^{-4} y 10^0.

Se obtuvieron los siguientes gráficos:

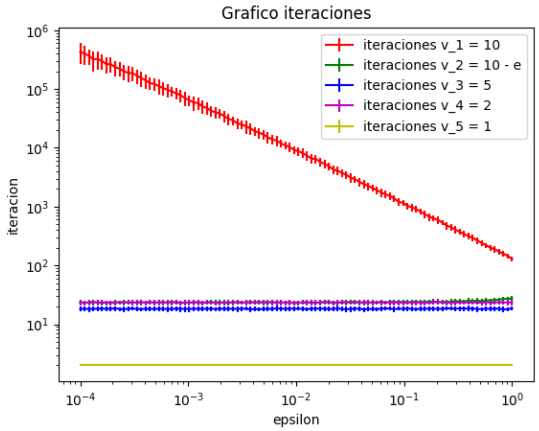

Figura 3: Evolución iteraciones de A_ϵ para cada autovalor y autovector

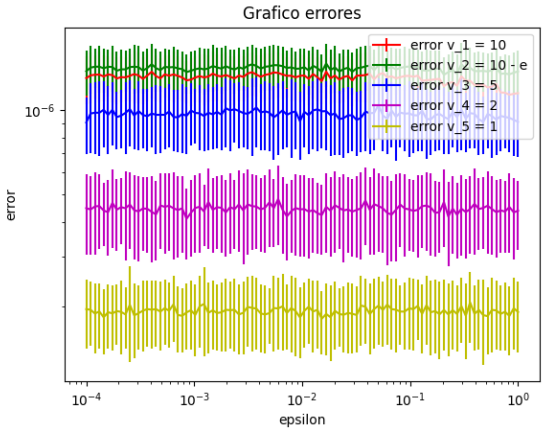

Figura 4: Evolución errores de A_ϵ para cada autovalor y autovector

El gráfico de iteraciones fue acorde a nuestras predicciones. Sabemos que cuando se aplica el *método de la potencia* en matrices cuyos autovalores no son estrictamente distintos, el algoritmo converge a una combinación lineal del autoespacio dominante. A medida que ϵ se hace más pequeño, los autovalores 10 y 10 - ϵ se vuelven más similares. Como la convergencia es lineal y depende del factor $|\frac{\lambda_{i+1}}{\lambda_i}|$ entonces este pequeño ϵ causa una disminución en la velocidad de convergencia del algoritmo, requiriendo más iteraciones para calcular el autovalor dominante debido a la proximidad de los autovalores, comparado con situaciones donde los autovalores son más dispares, como los que se ven mas abajo.

En el gráfico de errores, se ve como el error de la aproximación de los autovalores es mayor con el autovalor $10 - \epsilon$, y luego va bajando para cada autovalor de mayor a menor, siendo el autovalor 1 el que da menor error. Podemos concluir que el ϵ tiene influencia sobre el error de la aproximacion, aunque en este caso no es tan visible debido a la alta cantidad de iteraciones. Si se hicieran menos iteraciones, entonces podriamos ver el error en general crecer, debido a que el algoritmo no llega a converger.

4.4. Exploración del hiper-parámetro k (4-fold cross validation)

Se realizaron mediciones de la performance de KNN para $1 \leq k \leq 100$ y se obtuvieron los siguientes promedios de performance para $Q \in \{500, 1000, 5000\}$:

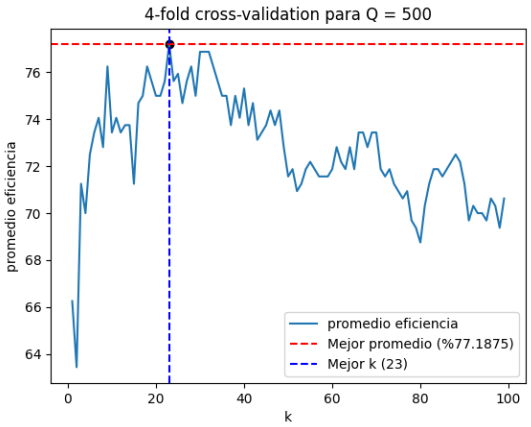

Figura 5: 4-fold cross-validation (Q = 500)

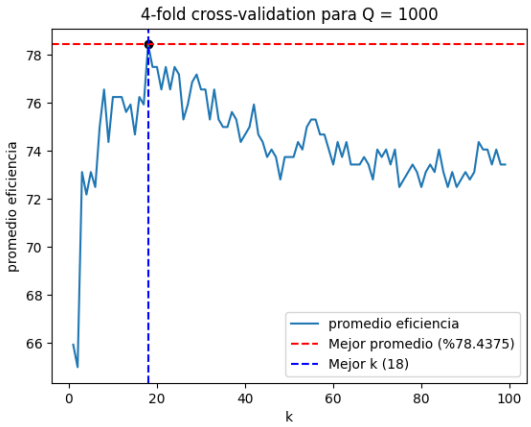

Figura 6: 4-fold cross-validation (Q = 1000)

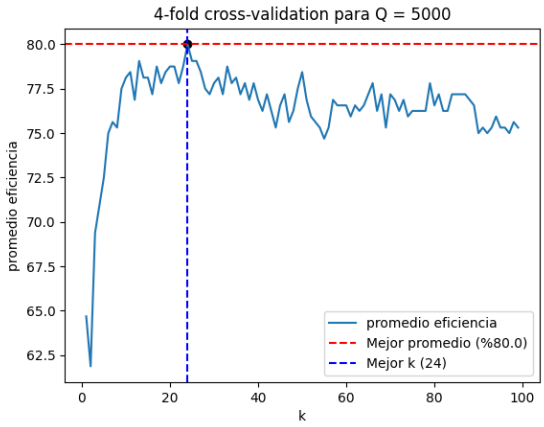

Figura 7: 4-fold cross-validation (Q = 5000)

Se hallaron distintos valores de k; para $Q = 500$ fue $k = 23$, para $Q = 1000$ fue $k = 18$, y para $Q = 5000$ fue $k = 24$.

4.5. Exploración del hiperparámetro p (Preprocesamiento con PCA)

Luego de calcular los autovectores (matriz V) y autovalores (matriz D) de la matriz de covarianza C_x de X, obtuvimos el siguiente gráfico de la varianza acumulada de C_x:

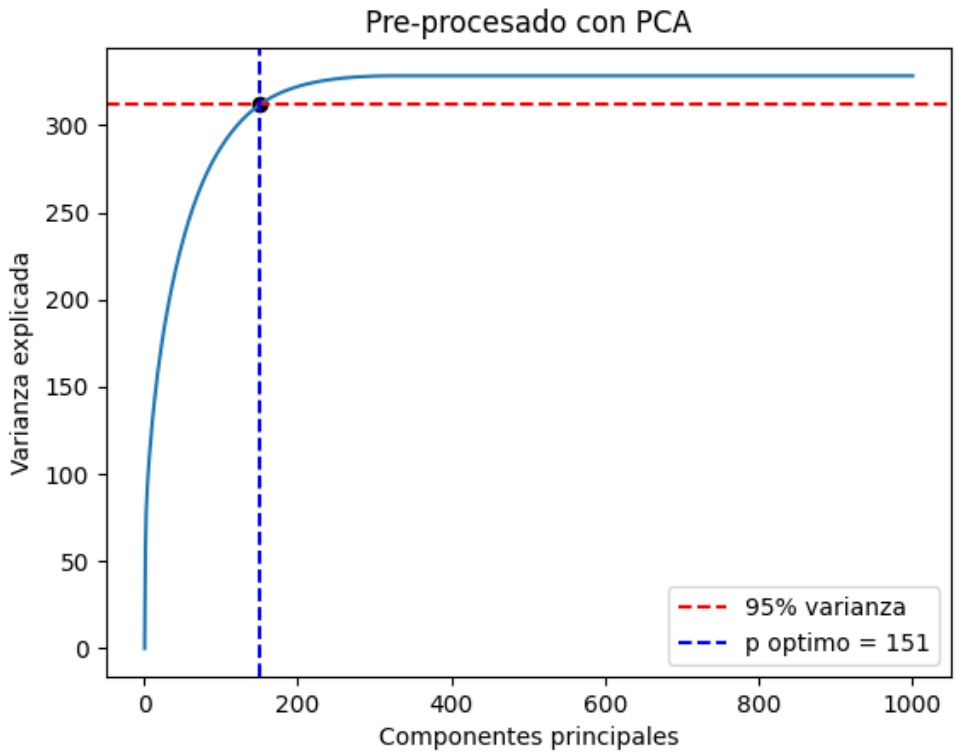

Figura 8: Evolución varianza acumulada ($Q = 1000$)

Lo que podemos concluir es que la varianza de los datos se concentra principalmente en las primeras 151 componentes principales, por lo tanto, esto puede sugerir en principio un buen valor de p para PCA con el cual trabajar, o al menos una cota superior.

4.6. Pipeline final

En el siguiente gráfico se ven representados los niveles de performance para cada combinación de p y k:

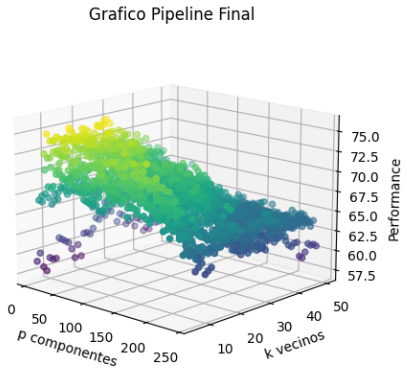

Figura 9

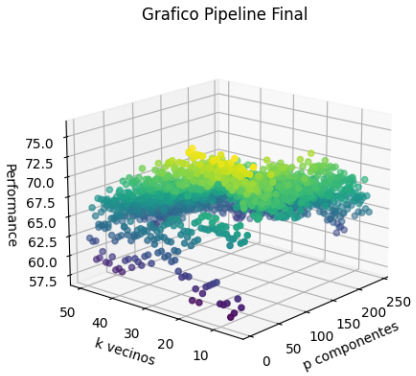

Figura 10

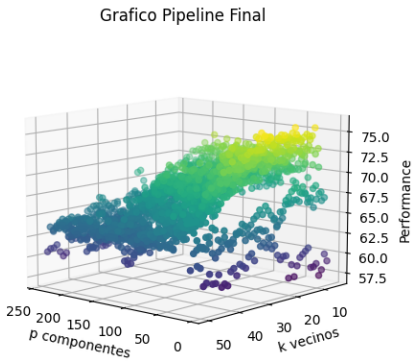

Figura 11

donde se obtuvo la mejor performance para $k = 11$ y $p = 30$, que fue de 75,625 %.

Posterioremente, tomamos estos valores para entrenar todos los datos de entrenamiento y medir la performance en los datos de prueba, donde se obtuvo una performance del 83,75 %. Esto indica que, en efecto, los valores $k = 11$ y $p = 30$ son buenos valores para transformar los datos y hacer KNN con una precisión similar a la que obtenemos con la matriz X normal. Notemos que usando $p = 30$ vamos a trabajar con una matriz mucho más pequeña, lo cual va a reducir significativamente los tiempos de cómputo que se tendrían con una matriz de tamaño $Q = 1000$.

Veamos que este performance obtenido es muy cercano a los conseguidos mediante KNN sin hacer PCA, por lo cual esta alternativa parece una buena opcion.

5. Conclusiones

Gracias al uso del álgebra de matrices, podemos crear algoritmos de machine learning asombrosos que sin saberlo están en nuestra vida cotidiana en varios tipos de software, como aquellos que nos recomiendan películas en nuestros servicios de streaming favoritos, o en sistemas de tiendas online, etc.

El algoritmo de KNN (k-nearest neighbours) es una forma de clasificar datos muy interesante y sencilla de entender. Su uso combinado con el método de la potencia para reducir la dimensión de los datos mediante un valor p adecuado nos muestra cómo la varianza se relaciona con nuestros datos. También, mediante *n-fold cross validation* podemos ver cómo la eficiencia de KNN varía según los datos de entrenamiento y el valor k que usemos. Esto, junto con la exploración del hiperparámetro p, nos ayudó para encontrar una mejor combinación con la cual entrenar a nuestro algoritmo de clasificación de películas.

Lo desarrollado durante este trabajo puede usarse en proyectos futuros para clasificación de distintos tipos de

datos además de texto, lo que propondrá nuevos desafíos en cómo procesaremos estos datos y qué hiper-parámetros utilizaremos si es que decidimos seguir trabajando con los mismos algoritmos.