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

Métodos Numéricos

Primer cuatrimestre de 2024

Trabajo práctico Integrador

Análisis de componentes principales y procesamiento de lenguaje

Integrantes

Integrante	LU	Correo electrónico
Ranieri, Martina	1118/22	martubranieri@gmail.com

Resumen

Para este trabajo se nos propuso realizar un reconocedor de textos para identidicar a qué género pertenece la película que están describiendo. Para esto vamos a utilizar k-vecinos más próximos (KNN). También se implementa el método de la potencia con deflación para realizar el análisis de componentes principales la cual es utilizada para describir un conjunto de datos en términos de nuevas variables no correlacionadas. El dataset utilizado para este trabajo es Wikipedia Movie Plots [1].

Palabras Claves

PCA; KNN; Método de la potencia con deflación; Validación Cruzada;

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

[1] Wikipedia Movie Plots

Índice

1. Introducción **3**
1.1. Autovectores y autovalores . . . 3
1.2. Método de la potencia . . . 4
1.3. Covarianza y correlación . . . 4
1.4. Análisis de componentes principales (PCA) . . . 4
1.5. K-vecinos más cercanos (KNN) . . . 5
1.6. Validación cruzada . . . 6

2. Desarrollo **6**
2.1. Funciones generales de los algoritmos . . . 6
2.2. Método de la potencia con deflación . . . 7
2.3. KNN . . . 8
2.4. PCA . . . 8
2.5. Reconocedor y optimización . . . 9

3. Resultados **10**
3.1. Método de la potencia con deflación . . . 10
3.2. KNN . . . 11
3.3. PCA . . . 13
3.4. Reconocedor y optimización . . . 14

4. Puntos opcionales **14**
4.1. TF-IDF . . . 14
4.2. KNN distancia euclídea vs distancia coseno . . . 15
4.3. Variante validación cruzada . . . 17

5. Conclusiones **18**

1. Introducción

Los textos de cada película del dataset ya vienen procesados para poder generar un vector x_i el cual indica la cantidad de veces que aparece un token para la película i. En total tenemos 9581 tokens, entonces podemos pensarlo como una matriz de $\mathcal{R}^{Nx9581}$ siendo N la cantidad de películas. Como las dimensiones de la matriz son muy grandes y tardaríamos mucho en procesarlo, nos vamos a quedar con las 5000/1000/500 tokens más frecuentes (el valor exacto va a depender de lo que nos diga el enunciado). Las películas vienen separadas en 4 categorías; 'crime', 'romance', 'science fiction' y 'western'. Nuestro objetivo es poder armar un clasificador de películas, es decir que dada una nueva, poderle asignar correctamente a qué categoría pertenece.

Los métodos de KNN y PCA serán explicados detalladamente a lo largo del documento pero vamos a utilizar k-vecinos más próximos para clasificación. El $\boldsymbol{k}$ representa el número de vecinos más cercanos que se consideran a la hora de hacer la predicción. Por otro lado, utilizamos PCA para acotar las dimensiones del espacio, transformando los datos originales con las componentes ordenadas en función de su varianza La $\boldsymbol{p}$ nos indica el número de componentes principales que juntan mayor varianza. Al mantener estas componentes, podemos conservar la mayor parte de la información importante, reduciendo la complejidad del modelo, pues las componentes con mayor varianza explican la mayor parte de la estructura y relaciones de los datos originales.

1.1. Autovectores y autovalores

Sea A una matriz $\in C^{nxn}$, entonces x $\in C^n$ no nulo es un autovector de A $\leftrightarrow \exists\lambda$ escalar tal que:

$$Ax = \lambda x$$

El escalar λ se denomina autovalor de A y se dice que x es un autovector asociado a λ de A. Es importante destacar a la hora de hablar sobre autovalores y autovectores es que si se tiene un autovalor λ de una matriz A, entonces la matriz $A - \lambda I$ es singular. Esto se debe a que:

$$\begin{aligned} Ax &= \lambda x \\ Ax - \lambda x &= 0 \\ (A - \lambda I)x &= 0 \end{aligned} \tag{1}$$

1: *Como x es no nulo por definición, entonces es singular. Su determinante es cero.*

Al desarrollar el determinante, el resultado es un polinomio $P(\lambda) = det(A - \lambda I)$ llamado Polinomio Característico de A. Podemos decir que λ es autovalor $\leftrightarrow$ es raíz de ese polinomio, por lo tanto, cada matriz de $n \times n$ tiene n autovalores contados con su multiplicidad por ser las raíces de un polinomio de grado n.

Vamos a decir que las matrices A y B $\in \mathcal{R}^{nxn}$ son semejantes si existe una matriz P de $n \times n$ inversible tal que:

$$A = P^{-1}BP$$

Podemos construir la matriz P a partir de los autovectores de A, colocándolos como columnas. Este concepto de matrices semejantes es importante ya que si A y B lo son, entonces comparten autovalores. Por lo tanto tenemos que; dado P de la forma $A = P^{-1}BP$ y v_i el autovector asociado a λ_i de A:

$$\begin{aligned} Av_i &= \lambda_i v_i \\ \rightarrow B * P_{v_i} &= \lambda_i * P_{v_i} \end{aligned}$$

Además, hay matrices que son semejantes a una matriz diagonal, es decir que dada $A \in \mathcal{R}^{nxn}$, $\exists D$ diagonal que comparte autovalores con A. En este caso se dice que la matriz A es diagonalizable por semejanza y tiene la propiedad de tener base de autovectores.

$$A = P^{-1}DP \ \wedge \ \textit{Los autovectores de A forman una Base}$$

Otra forma de verificar si una matriz tiene base de autovectores es utilizar la propiedad que nos dice que si A tiene todos los autovalores reales, entonces existe $Q \in R^{n \times n}$ ortogonal tal que $Q^t AQ = T$ con T triangular superior. Entonces tenemos que A es semejante a una triangular superior. Si además A es simétrica, entonces T es diagonal con los autovalores en su diagonal y las columnas de Q son los autovectores de A. Por lo tanto, esta propiedad nos dice que si A es simétrica, entonces $\exists$ base ortonormal de autovectores y, por lo tanto, es diagonalizable por semejanza vía una matriz ortogonal.

$$Q^t AQ = D \rightarrow A = Q^t DQ \tag{2}$$

1.2. Método de la potencia

Es un método iterativo que calcula sucesivas aproximaciones a los autovectores y autovalores de una matriz. Se usa principalmente para calcular el autovalor de mayor valor de una matriz.

Sea $A \in \mathcal{R}^{nxn}$, λ_i i = 1...n, los autovalores con $v_1...v_n$ autovectores asociados que conforman una base, tenemos un autovalor dominante, es decir $|\lambda_1| > |\lambda_2| \geq ... \geq |\lambda_n|$. Vamos a querer obtener el autovalor principal y su autovector asociado, por lo que partimos de un vector x_0. En cada paso k, se calcula $x_{k+1} = \frac{Ax_k}{||Ax_k||}$ entonces x_k converge normalmente al mayor autovalor.

Si trabajamos con una matriz A que tiene sus autovalores tal que $|\lambda_1| > |\lambda_2| > |\lambda_3| > ... > |\lambda_n|$ y tiene base ortonormal de autovectores, podemos conseguir todos los autovalores y autovectores aplicando el **método de deflación**.

Notemos que si partimos de una matriz con las características mencionadas, Podemos aplicar n veces el método de la potencia y si lo hacemos en combinación con el método de deflación de manera iterativa hallaremos todos los autovectores y autovalores de A.

Podemos definirlo de la siguiente manera (Ecuación 3):

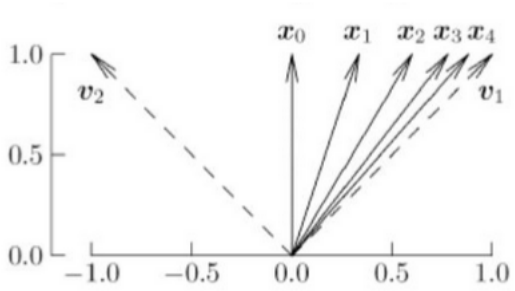

Figura 1: Objetivo del método de la potencia, por cada iteración aproximar más el x_0 al v_1

$$A' = A - \lambda_1 . u_1 . u_1^t \tag{3}$$

Con u_1 autovector asociado a λ_1

Entonces para poder aplicar este algoritmo vamos a tener como hipótesis que los autovalores son todos distintos, que nuestra matriz de datos conforma una matriz ortogonal y además el vector inicial no tiene que tener coordenada nula en dirección del autovector dominante. Es decir:

$$\langle v_0, v_1 \rangle \neq 0$$

En nuestro caso no es seguro que todos los autovalores sean distintos, pero por el método de la potencia que utilizamos en el laboratorio, con asegurar que todos los autovalores sean positivos es suficiente para que el algortitmo funciones. Esto lo podemos garantizar porque trabajamos con todos datos positivos y además la matriz es semi-definida positiva.

1.3. Covarianza y correlación

Dados dos vectores x e y definimos la covarianza como:

$$Cov(x, y) = \frac{(x - \mu_x).(y - \mu_y)}{n - 1} \tag{4}$$

μ representa el valor medio del vector. Podemos considerar la fórmula como producto interno entre los vectores "centrados" y normalizada por la dimensión menos uno.

Luego definimos la correlación como una covarianza normalizada para que su rango sea entre -1 y 1.

$$Corr(x, y) = \frac{(x - \mu_x).(y - \mu_y)}{\sqrt{(x - \mu_x).(x - \mu_x).(y - \mu_y).(y - \mu_y)}} \tag{5}$$

Esta expresión es equivalente al coseno del ángulo entre los vectores cos θ_{xy}

1.4. Análisis de componentes principales (PCA)

El **análisis de componentes principales** (PCA) busca encontrar un cambio de base de los vectores, tal que las dimensiones se ordenen en componentes que explican los datos en orden de mayor a menor relevancia. De forma análoga, el cambio de base busca ordenar las dimensiones según su varianza, de mayor a menor. Para lograr esto se realiza una descomposición en autovectores y autovalores de la matriz de covarianza de los datos:

$$C = VDV^t \tag{6}$$

Siendo $\mathcal{V}$ la matriz con los autovectores en las columnas, $\mathcal{D}$ una matriz diagonal con los autovalores y $\mathcal{C}$ matriz diagonalizable 1.1 de covarianza

Los autovalores en la matriz diagonal $\mathcal{D}$ representan la cantidad de varianza por cada componente principal. Los autovalores más grandes corresponden aquellos que tienen mayor varianza en los datos. Estos además están ordenados de manera tal que la primera componente principal tiene la mayor varianza posible, la segunda componente principal tiene la segunda mayor varianza, y así sucesivamente. La varianza por componente principal indica cuánta información del conjunto de datos original es capturada por cada componente principal.

La varianza explicada acumulada es la suma acumulativa de la varianza explicada por cada componente principal. Esto sirve para determinar cuántos componentes principales $\boldsymbol{p}$ son necesarios para explicar un porcentaje significativo de la varianza total de los datos. Se calcula de la forma que podemos ver en Ecuación 7 y nos da el porcentaje de la varianza total que es explicada por los primeros i componentes principales.

$$\text{Varianza explicada acumulada}_i = \frac{\sum_{j=1}^{i} \lambda_j}{\sum_{j=1}^{k} \lambda_j} \times 100\,\% \tag{7}$$

La matriz $\mathcal{V}$ permite transformar los datos $\mathcal{X}$ en la nueva base mediante la operación $\mathcal{X} \times \mathcal{V}$, en nuestro caso $\mathcal{X}$ es la matriz de conteo. De esta manera, es posible reducir la dimensionalidad de los datos utilizando solo las $\boldsymbol{p}$ componentes principales que se deseen, es decir, los primeros $\boldsymbol{p}$ vectores columna de la matriz $\mathcal{V}$. Así podemos quedarnos solo con las componentes principales más significativas.

En conclusión, el PCA transforma un conjunto de datos original en un nuevo conjunto de datos en el que las nuevas características (componentes principales) son combinaciones lineales de las originales. Este proceso facilita la reducción de la dimensionalidad.

1.5. K-vecinos más cercanos (KNN)

K-NN se clasifica como un algoritmo de aprendizaje automático supervisado. En este se considera a cada objeto del conjunto de entrenamiento como un punto del espacio m-dimensional. Para este, se conoce de qué categoría es cada objeto, para luego dada una nueva película nueva, poder asociarle la clase del/los punto/s más cercano/s del conjunto de datos que teníamos. La dimensión m es la cantidad total de tokens.

Para realizar el procedimiento, se define un conjunto $\mathcal{D} = \{x_i : i = 1, \ldots, n\}$. Dada una nueva película $y \notin \mathcal{D}$, para clasificarla se busca un subconjunto de $\mathcal{D}$ con los $\boldsymbol{k}$ elementos más cercanos a la película nueva (sus tokens). Para calcular qué elementos son los más cercanos a y utilizamos la distancia coseno (Ecuación 8) la cual mide qué tan diferentes son los tokens de la nueva película con respecto a nuestro conjunto $\mathcal{D}$ (tomamos a los tokens como vector $\in \mathcal{R}^{9581}$). Finalmente se asigna la categoría que más se repita en el subconjunto.

$$d(A, B) = 1 - \frac{A \cdot B}{\|A\|\|B\|} \tag{8}$$

Distancia coseno

Notar que los resultados de la predicción pueden cambiar drásticamente dependiendo del valor $\boldsymbol{k}$ y la naturaleza de los datos. Por esto, las predicciones son sensibles a la dimensionalidad de las películas.

Por ejemplo cuando los tokens de la película y están lejos de los puntos del conjunto de datos $\mathcal{D}$, o cuando el conjunto $\mathcal{D}$ está muy desordenado (Figura 2).

Otras desventajas del algoritmo es que se necesita mucho tiempo para calcular la distancia entre cada muestra de prueba y todas las muestras de entrenamiento. Además ocupa mucha memoria, dado que no se guarda de antemano ningún dato, entonces cada vez que queremos a predecir un valor se repiten los mismos pasos. La complejidad de este algoritmo es de $\mathcal{O}(n * d)$, pues se tienen que calcular las distancias entre el punto de prueba y cada uno de los $\boldsymbol{n}$ puntos de entrenamiento, donde $\boldsymbol{d}$ es la dimensionalidad de los datos.

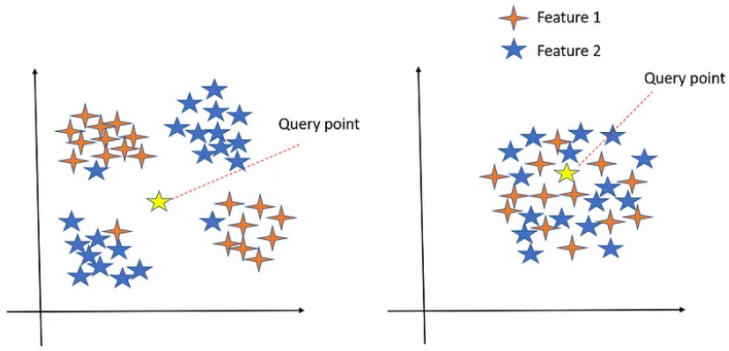

Figura 2: Ejemplos de KNN cuando el algortimo falla. Imagen de: Medium.com [1]

El valor $\boldsymbol{k}$ puede tomar números entre 1 y n, siendo n la cantidad de elementos que tenemos en $\mathcal{D}$. Dependiendo del valor de $\boldsymbol{k}$ vamos a terminar clasificando el texto y en una categoría u otra.

Para elegir el k más óptimo, lo que vamos a hacer es entrenar KNN. Para esto realizamos 4 particiones (Figura 3) en nuestra matriz de datos, x_{train} con los tokens. Una de las particiones como conjunto de prueba (test set) y las otras 3 como conjunto de entrenamiento (train set). Para cada película en el conjunto de prueba, calculamos la distancia coseno (Ecuación 8) con todas las películas en el conjunto de entrenamiento y las ordenamos de menor a mayor distancia. Luego nos quedamos con los k tokens más cercanas y medimos en porcentaje su performance a la hora de predecir la categoría a la que correspondía cada película de testeo. Realizamos ese procedimiento en cada partición, es decir 4 veces y se calcula el promedio de las 4 particiones. Todo esto para un k, entonces vamos a probar diferentes valores de k y capturar las métricas de rendimiento para ese valor en k, quedándonos con el resultado de mejor rendimiento a la hora de predecir.

1.6. Validación cruzada

En este proyecto, abordamos el desafío de optimizar los hiperparámetros de los algoritmos KNN y PCA para maximizar la precisión de clasificación en un conjunto de datos de películas. El algoritmo KNN es sensible a la elección del parámetro k, que define el número de vecinos más cercanos utilizados para clasificar cada instancia. Un valor de k demasiado pequeño puede hacer que el modelo sea muy específico (overfitting), mientras que una k muy grande puede resultar en un modelo demasiado general (underfitting), lo cual podría reducir la precisión en datos no vistos. De manera similar, en el caso de PCA, el parámetro p define el número de componentes principales seleccionadas para la reducción de dimensionalidad. Este parámetro impacta directamente en la capacidad del modelo de mantener la varianza de los datos originales mientras simplifica la estructura de entrada, mejorando potencialmente la eficiencia y la capacidad de generalización del modelo KNN.

Dado que no conocemos a priori los valores óptimos de k y p para este conjunto de datos, es esencial realizar una experimentación sistemática. En particular, la validación cruzada es un método robusto para evaluar el rendimiento general del modelo y evitar el ajuste excesivo al conjunto de entrenamiento. La validación cruzada (Figura 3) implica dividir los datos en k subconjuntos (folds) aproximadamente iguales. En cada iteración, se entrena el modelo en k-1 subconjuntos y se evalúa en el fold restante, permitiendo observar el rendimiento en múltiples divisiones del conjunto de datos. En este proyecto, utilizamos k = 4, repitiendo el proceso cuatro veces para promediar los resultados obtenidos y así minimizar el riesgo de sobreajuste. Esto ayuda a construir una métrica de rendimiento más estable y confiable, al considerar distintas particiones de los datos.

Para hallar los valores óptimos de k y p, exploraremos un rango amplio de ambos parámetros en nuestro conjunto de entrenamiento, x_{train}. Esta exploración se realiza utilizando una grilla de búsqueda, probando combinaciones de k y p, con el fin de identificar el par de valores que maximiza la precisión en los datos de validación. Los detalles específicos del procedimiento y la implementación de esta búsqueda de hiperparámetros se presentan más adelante en la Subsección 3.4, donde se discuten los algoritmos y los resultados obtenidos en profundidad.
Este enfoque no solo permite identificar la configuración óptima para los datos actuales, sino también construir un modelo con mayor capacidad de generalización para datos no vistos, lo cual es esencial en contextos de clasificación donde los patrones de los datos pueden variar ampliamente.

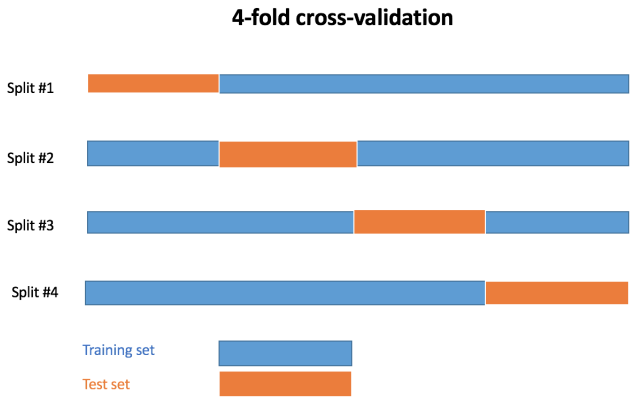

Figura 3: Validación cruzada 4-fold. Imagen de: ieeexplore.ieee.org [2]

2. Desarrollo

2.1. Funciones generales de los algoritmos

- $traspuesta(x)$ calcula la traspuesta de x.
- $longitud(x)$ es la longitud de x [2].
- $matriz(n, m)$ Devuelve una matriz con la longitud (n, m).
- $promedio(x)$ calcula el promedio de x. En Python es mean [3].
- $norma(x)$ calcula la norma de x [4].

[2] len
[3] mean
[4] numpy.lianalg.norm

- *arrayDeMatrices*() Crea una matriz vacía con las dimensiones que se le pasen por parámetro. En Python es empty [5].

2.2. Método de la potencia con deflación

Para implementar el método de la potencia con deflación primero realizamos el algoritmo del método de la potencia sin deflación y luego con. Se utilizó la librería eigen de C++ [6].

Debemos recordar que para aplicar el método de la potencia con deflación, nuestra matriz debe cumplir ciertos requieres;

1. Tener base ortonormal de autovectores.
2. Todos los autovalores sean positivos; $|\lambda_1|...|\lambda_n| > 0$.
3. El vector inicial no tiene que tener coordenada nula en dirección del autovector dominante.

Es por eso que nos armamos las matrices a partir de una matriz diagonal, ya que utilizando $Q = I - 2vv^T, \|v\|_2 = 1$ la matriz de Householder, obtenemos las matrices ortogonales y podemos armarnos la matriz A, a la cual vamos a calcularle los autovalores y autovectores. Como la armamos de esta manera, sabemos que los autovalores serán los elementos que pongamos en nuestra matriz diagonal por ser matriz diagonalizable simétrica (1.1).

$$A = Q^T \begin{pmatrix} d_1 & 0 & 0 & 0 \\ 0 & d_2 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & d_n \end{pmatrix} Q \tag{9}$$

Con todos los d_i distintos

Algorithm 1 Método de la potencia

Parámetros: Matriz **A**, número máximo de iteraciones **niter** = **10,000**, tolerancia $\varepsilon = 10^{-7}$
v = vectorAleatorio(dimensión(A)[0])
for $i = 0...niter$ **do**
    $v\_viejo = v$
    $v = \frac{A \cdot v}{\|A \cdot v\|}$ ▷ Normalizamos el vector v
    $a = \frac{(v^T \cdot A \cdot v)}{(v^T \cdot v)}$ ▷ Calculamos el autovalor aproximado
    if $\|v - v\_viejo\|_\infty < \varepsilon$ **then** ▷ Verificamos si el cambio es menor que la tolerancia
        break
    end if
end for
return a, v

Algorithm 2 Método de la Potencia con Deflación

Parámetros: Matriz **A**, número máximo de iteraciones **niter** = **10,000**, tolerancia $\varepsilon = 10^{-8}$
autovectores = []
autovalores = []
for i = 1 ... longitud(A) **do**
    a, v = metodoPotencia(A, $niter$, ε)
    autovalores[i] = a
    autovectores[:, i] = v ▷ Guardamos el autovector como columna
    $A = A - a \cdot (v \otimes v)$ ▷ Actualizamos A aplicando deflación. Producto externo
end for
return autovalores, autovectores ▷ Devolvemos autovalores y autovectores

Para verificar la implementación realizamos distintas diagonales para obtener las Q ortogonales con $I - 2vv^T, \|v\|_2 = 1$. Para comprobar que esos sean los autovalores y autovectores de A usamos un for para recorrer cada autovector y autovalor. Usamos $np.allclose(A \ @ \ autovector_i, \ autovalor_i \ @ \ autovector_i)$[7] ya que por definición deben ser iguales pero por el error numérico que tenemos a la hora de usar la computadora, no podemos usar "==". Además debemos recordar que estamos trabajando con un algoritmo iterativo que buscar aproximaciones, no valores exactos. (Ver detallado en **pruebas-potencia-verificacion.ipynb** para ver la verificación)

[5] empty
[6] eigen
[7] allclose

- D1 = [1, 2]
- D2 = [6, 1, 2]
- D3 = [1, 10, 6, 2, 3]
- D4 = random.sample(range(500), 100), el propósito era probar una matriz grande con números random [8].

Algo interesante para estudiar con este algoritmo es su convergencia cuando las columnas de nuestra matriz son "casi" linealmente dependiente y cómo varía el error cuando esto sucede. Esto lo analizamos en detalle en la Subsección 3.1.

2.3. KNN

Para el algoritmo de **KNN** con un $\boldsymbol{k}$ fijo (Algortimo 4), por cada película de test calculamos su distancia con todas las del conjunto train, ordenamos la lista por la distancia, y nos quedamos con las primeras $\boldsymbol{k}$ películas más cercanas a la nueva. Luego, vemos qué categoría es la más repetida de esa lista y esa será nuestra predicción. Verificamos si la predicción es correcta o no guardando 1 o 0 según el caso. Finalmente devolvemos el porcentaje de aciertos.
Para verificar la implementación, utilizamos KNeighborsClassifier [9]. Detallado mejor en Subsección 3.2

- $ordenarDistancias(x)$ ordena de mayor a menor con segunda componente de x. En python usamos argsort [10].
- $mayorRepetido(x)$ Nos devuelve la categoría que más se repite en x. En python usamos stats.mode [11].
- $concatenandoParticion(i)$ Devuelve las particiones del array concatenadas, excepto la de la posición i.

Algorithm 3 Distancia coseno

1: **Parámetros: A, X**
2: producto = A $\times$ X^T
3: A_{norma} = norma(A)
4: X_{norma} = norma(X)
5: $prod_{norma}$ = A_{norma} $\times$ X_{norma}
6: res = 1 - ($\frac{producto}{prod_{norma}}$)
7: **return** res

Algorithm 4 knn

1: **Parámetros:** x_{train}, x_{test}, y_{train}, y_{test}, k
2: acertados = []
3: distancias = distCoseno(x_{train}, x_{test})
4: indices = ordenarDistancias(distancias)[:k] ▷ Nos quedamos con los k indices para ordenar distancias
5: categoriasOrdenadas = $y_{train}[indices]$
6: mayorRepetido= mayorRepetido(categoriasOrdenadas)
7: acertados = (mayorRepetido== y_{test}) ▷ Acertados es array con 1 si acertó, 0 sino
8: **return** promedio(acertados) ▷ Devolvemos porcentaje de los acertados de la predicción

2.4. PCA

Para realizar el algoritmo de PCA primero centralizamos los datos. Para cada columna i del conjunto de datos, calculamos su media. Luego, restamos esta media a cada elemento de la columna i. Este proceso asegura que cada columna tenga una media cero. Con los datos centralizados podemos calcular la matriz de covarianza.

Utilizamos la función $metodoPotenciaConDeflacion(A)$ (algoritmo 2) para calcular los autovalores y autovectores de la matriz de covarianza. Los autovalores obtenidos representan la varianza de cada componente principal mientras que los autovectores indican la direcciones de estas componentes principales.

Para verificar que nuestra implementación sea correcta, usamos la librería sklearn.decomposition [12]. la cual cuenta con una función de PCA.

- $mediaPorColumna(A)$ calcula la media por columna. En python usamos mean [13].

[8] random-sample
[9] KNeighborsClassifier
[10] argsort
[11] stats.mode
[12] sklearn.decomposition
[13] mean

Algorithm 5 PCA

Parámetros: A
X = centralizar(A)
matrizCovarianza = $\frac{1}{\text{longitud}(X)-1} \cdot (X^T \cdot X)$ ▷ Calculamos la matriz de covarianza
autovalores, autovectores = metodoPotenciaConDeflacion(matrizCovarianza)
return autovalores, autovectores

Algorithm 6 Centralizar

Parámetros: A
u = mediaPorColumna(A) ▷ Calculamos el vector de medias de cada columna
B = matriz(longitud(A)) ▷ Matriz de igual tamaño que A
for i ∈ 0..longitud(A) **do**
    B[i] = A[i] - u
end for
return B

La varianza acumulada por cada componente principal indica cuánta varianza del conjunto de datos x_{train} tiene cada componente. Para determinar el número de componentes principales $\boldsymbol{p}$ necesarios para explicar al menos 95 % de la varianza total calculamos la varianza explicada acumulada y obtenemos el número mínimo de componentes principales cuya varianza acumulada sea mayor o igual al 95 %. Nos quedamos con los primeros $\boldsymbol{p}$ componentes principales. Para tener el nuevo conjunto de datos reducido, proyectamos los datos originales x_{train} en el espacio de los $\boldsymbol{p}$ componentes principales seleccionados, haciendo el producto punto entre x_{train} y los $\boldsymbol{p}$ autovectores correspondientes.

- $sumaAcumulada(A)$ Devuelve la suma acumulada de A. En Python es cumsum [14].
- $suma(x)$ Devuelve la suma de x.
- $maximaVarianza(A)$ me devuelve los índices que tienen varianza mayor igual a 95. En Python es argmax [15].
- $achicar(A, x)$ Devuelve una matriz únicamente con las columnas cuyo índice están en el vector x.

Algorithm 7 Selección componentes principales

Parámetros: autovalores, autovectores
varianzaAcumulada = $\frac{sumaAcumulada(autovectores)}{suma(autovalores)}$ ▷ Calculamos la proporción acumulada de varianza
indicesMayor95 = maximaVarianza(varianzaAcumulada) ▷ Índices de componentes con $\geq$ 95 % de varianza
mayor95 = achicar(autovectores, indicesMayor95)
res = x_{train} × mayor95 ▷ Transformamos los datos con los autovectores seleccionados
return res

2.5. Reconocedor y optimización

Todo lo realizado en las anteriores secciones era para poder realizar el reconocedor de películas y para eso vamos a utilizar los algoritmos de KNN (4) y PCA (5). Para optimizar los parámetros $\boldsymbol{p}$ y $\boldsymbol{k}$ realizaremos validación cruzada. Primero para nuestro conjunto de datos x_{train}, vamos a realizarle las 4 particiones y por cada una de las particiones vamos a calcularle el PCA para reducir la dimensionalidad de los datos, la redundancia y mejorar el rendimiento.

Luego de esto, tenemos que obtener el mejor par $\boldsymbol{p}$ y $\boldsymbol{k}$. Para esto, por cada partición que tenemos del x_{train} vamos a probar siguiente conjunto de $\boldsymbol{k}$: 1, 5, 10, 15, 20, 30, 35, 40, 45, 50, 60, 70, 80, 90, 100, 150, 200. Por cada uno de estos, vamos a medir su rendimiento con cada $\boldsymbol{p}$. En este caso nos vamos a quedar con el conjunto: 1, 5, 10, 20, 50, 70, 90, 100, 120, 130, 140, 145, 150, 180, 200, 230, 250. La elección de estos $\boldsymbol{p}$ la vamos a detallar en la Subsección 3.3 pero principalmente nos permite una evaluación amplia y completa del impacto de la reducción de dimensionalidad en el rendimiento del modelo. Con los valores muy chicos del $\boldsymbol{p}$ puede disminuir el rendimiento del modelo por estar eliminando información importante al reducir demasiado las dimensiones. Si, en cambio, el $\boldsymbol{p}$ es muy grande no va a haber una mejora significativa del rendimiento del modelo y puede que aumente innecesariamente el tiempo de ejecución.

Haciendo esto, vamos a obtener una matriz de matrices, una matriz por cada partición. En cada una de ellas, vamos a tener filas con, el $\boldsymbol{p}$ y $\boldsymbol{k}$ con el cual se realizó el knn, además del porcentaje de efectividad que tuvo. Luego, con esa matriz, vamos a calcular el promedio que tuvo cada par de $\boldsymbol{p}$ y $\boldsymbol{k}$ con las 4 particiones y nos vamos a quedar con el par que además

[14] cumsum
[15] argmax

de tener el mejor promedio de % de rendimiento, que el $\boldsymbol{p}$ sea el más cercano al punto medio de todos los $\boldsymbol{p}$ ingresados y en caso de empate el k más chico.

Esta elección del par $\boldsymbol{p}$ y $\boldsymbol{k}$ se debe a que:

- ✓ Con el promedio de los rendimientos por cada partición nos aseguramos que el resultado no esté sesgado por un conjunto en particular.
- ✓ Al elegir a los que tengan el mejor promedio, nos aseguramos que estamos eligiendo un subconjunto del total que tienen buena efectividad en el conjunto de datos.
- ✓ Elegir el $\boldsymbol{p}$ más cercano al punto medio nos mejora la estabilidad ya que los valores extremos de $\boldsymbol{p}$ pueden no funcionar muy bien para todos los conjuntos de datos, mientras que los puntos medios suelen ser más equilibrados.

- *mejorPar*() Selecciona el par $\boldsymbol{p}$ $\boldsymbol{k}$ con el valor de p más cercano al punto medio y, en caso de empate, el menor k.
- *promedioParticiones*() Calcula el promedio de las 4 particiones

Algorithm 8 Par P y K más óptimos

Parámetros: x_{train}, y_{train}
valoresP = [1, 5, 10, 20, 50, 70, 90, 100, 120, 130, 140, 145, 150, 180, 200, 230, 250]
valoresK = [1, 5, 10, 15, 20, 30, 35, 40, 45, 50, 60, 70, 80, 90, 100, 150, 200]
rendimientos = arrayDeMatrices((longitud(valoresP), longitud(valoresK))
= longitud(x_{train})
alto = n // 4
indices = array(n)
for i ∈ 0..4 **do**
    $testIndices$ = indices[i * alto: (i + 1) * alto]
    $trainIndices$ = concatenar((indices[:i * alto], indices[(i + 1) * alto:]))
    $autovalores_{train}$, $autovectores_{train}$ = pca(x_{train}[$trainIndices$])
    for p ∈ valoresP **do**
        $x_{autovectoresActual}$ = $autovectores_{train}$[:, :p]
        $x_{trainCentralizado}$ = x_{train}[$trainIndices$]
        $x_{testCentralizado}$ = centralizar(x_{train}[$testIndices$])
        $x_{trainReducido}$ = $x_{trainCentralizado}$ × $x_{autovectoresActual}$
        $x_{testReducido}$ = $x_{testCentralizado}$ × $x_{autovectoresActual}$
        for k ∈ valoresK **do**
            rendimiento = knn($x_{trainReducido}$, $x_{testReducido}$, y_{train}[$trainIndices$], y_{train}[$testIndices$], k)
            matrizRendimientos[p, k] += rendimiento
        end for
    end for
end for
rendimientos = promedioParticiones(matrizRendimientos)
mejorP, mejorK = mejorPar(valoresP, valoresK, rendimientos)
return mejorP, mejorK ▷ Devolvemos mejor par p y k

Para poder medir el rendimiento del modelo con los valores de $\boldsymbol{p}$ y $\boldsymbol{k}$ tenemos que utilizar estos valores para entrenar y evaluar el modelo final con el x_{train}. Para ello debemos utilizar PCA y el $\boldsymbol{p}$ para transformar los datos de entrenamiento y de prueba para luego utilizar el KNN y $\boldsymbol{k}$ con los datos transformados por PCA (Algoritmo 9).

Algorithm 9 Evaluar mejor modelo

Parámetros: x_{train}, y_{train}, x_{test}, y_{test}, p, k
autovalores, $autovectores_{train}$ = pca(x_{train})
$x_{trainPCA}$ = x_{train}* $autovectores_{train}$[:, p]
$x_{testPCA}$ = x_{test} * $autovectores_{train}$[:, : p] ▷ Nos quedamos con p componentes principales
rendimiento = knn($x_{trainPCA}$, $x_{testPCA}$, y_{train}, y_{test}, k)
return rendimiento

3. Resultados

3.1. Método de la potencia con deflación

Para el método de la potencia con deflación se nos planteó estudiar su convergencia. Para esto medimos la cantidad de iteraciones que tarda en converger y medir el error del método. Para poder calcular la cantidad de iteraciones, necesitamos

un método de corte para devolver una vez convergemos. En el algoritmo que presentamos utilizamos:

$$\|autovector^{k+1} - autovector^{k}\|_\infty < \epsilon$$

De esta forma medimos el cambio entre las iteraciones y cuando sea menor al ϵ, dejamos de iterar y devolvemos. v_k es el vector propio aproximado de la iteración k-ésima. Para medir el error de método usamos:

$$\|A * v_i - \lambda_i * v_i\|_2$$

Siendo v_i , λ_i la aproximación al autovector y autovalor i.

Luego, a partir de la matriz diagonal dada, que son nuestros autovectores; $D = \begin{bmatrix}10, & 10-\epsilon, & 5, & 2, & 1\end{bmatrix}$ con 20 ϵ entre 0 y 1 espaciados logarítmicamente. Con esa matriz diagonal, calculamos 20 Householders distintos (digámosle M) para luego calcular los autovectores y autovalores de la matriz M con el método de la potencia con deflación. Después, a ese resultado le calculamos el error. Finalmente calculamos el promedio de los errores por cada componente de las 20 matrices calculadas y este mismo procedimiento lo hacemos para las 20 diagonales distintas.

Nuestra primera hipótesis antes de realizar el experimento era que mientras más chico sea el épsilon, más iba a tardar en converger el método, mientras que si el épsilon es más grande, como los autovalores son distintos por un rango mayor, el método debería converger en menos cantidad de iteraciones. Luego de realizar el experimento, corroboramos que esto es lo que sucede (Figura 4). Como tenemos dos métodos de corte, el método converge con la norma infinito como explicamos antes, o por cantidad de iteraciones, en nuestro caso 10000.

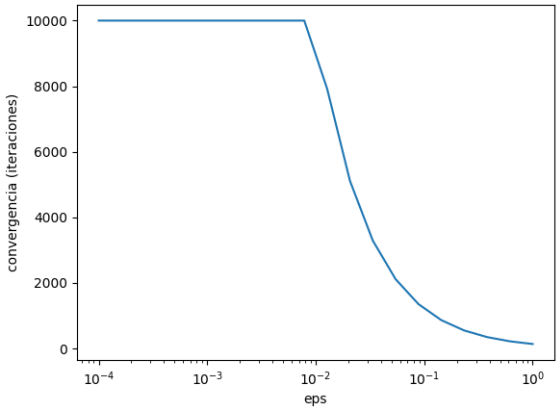

Figura 4: *Resultado con épsilon entre 0 y 1 espaciados logarítmicamente. Primera componente*

En el caso de la Figura 5 como el método converge por cantidad de iteraciones, las componentes 1 y 2 de los autovectores no tienen un error significativamente mayor a las demás pero esto se debe a que iteramos el método tantas veces que aproximadamente da similar (Figura 6). Además, a medida que el epsilon es mayor, el error va disminuyendo hasta ser prácticamente igual a la de las demás coordenadas. Eso ocurre aproximadamente en $\epsilon = 10^{-2}$, que si volvemos a mirar la Figura 4 notamos que es cuando el método empieza a converger por el método de corte.

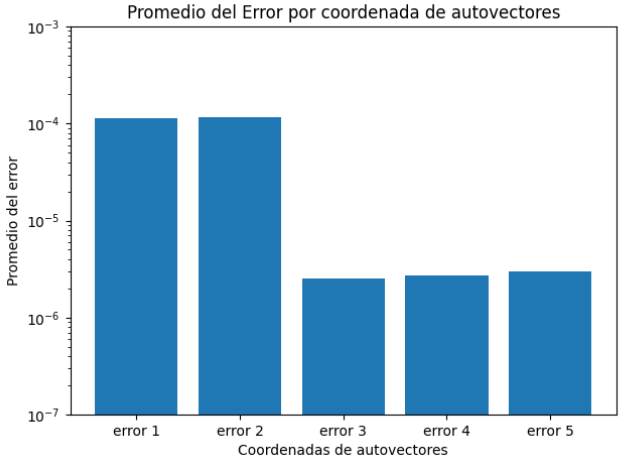

Figura 5: Promedio del Error por coordenada de autovectores

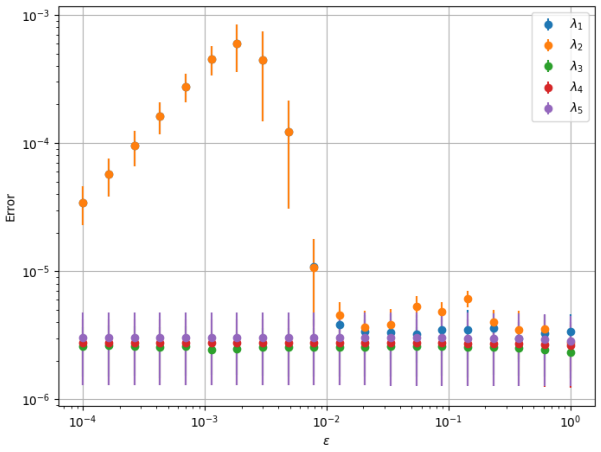

Figura 6: *Error con épsilon entre 0 y 1 espaciados logarítmicamente. Todas las componentes*

3.2. KNN

Para el algoritmo de KNN se pidió realizar un clasificador de géneros de películas para un $\boldsymbol{k} = 5$ y medir la performance. Como utilizar las 9581 dimensiones es muy costoso, nos quedamos con los $\mathcal{Q} \in \{500, 1000, 5000\}$ tokens más frecuentes con

con la medida de exactitud en el 20 % restante. Para ver que además nuestro algoritmo de knn es correcto, se comparó los resultados con los de la librería *KNeighborsClassifier*[16] de *sklearn.neighbors* [17] y efectivamente lo es. Los resultados fueron:

- Con Q = 500: efectividad del 65.625 %
- Con Q = 1000: efectividad del 75 %
- Con Q = 5000: efectividad del 68.75 %

Con estos resultados podemos ver que no necesariamente al utilizar más cantidad de tokens, estamos mejorando la efectividad del algoritmo. Por eso para cuando hagamos la validación cruzada para la exporación de los hiper-parámetros de $\boldsymbol{k}$ y $\boldsymbol{p}$, no es necesario utilizar todas las dimensiones y se nos pide realizarlo con Q = 1000.

Luego, se nos pidió explorar el hiperparámetro $\boldsymbol{k}$ usando 4-fold cross-validation para cada valor de $\mathcal{Q} \in \{500, 1000, 5000\}$ y medir la performance. Se probaron valores de $\boldsymbol{k}$ del 1 al 320 y los valores óptimos fueron:

- Con Q = 500: k = 30 y efectividad: 75.937 %
- Con Q = 1000: k = 28 y efectividad: 79.688 %
- Con Q = 5000: k = 31 y efectividad: 81.562 %

Si miramos las Figura 7 Figura 8 y Figura 9 podemos notar que a medida que el valor de $\boldsymbol{k}$ aumenta, tiende a descender el porcentaje de efectividad en los tres casos. También podemos notar que no mejora significativamente tomar Q = 5000 comparado a tomar Q = 1000, además de ser mucho más costoso el tiempos de cómputo. Esto refuerza el argumento de que más tokens no siempre garantizan mejor rendimiento.

Además, al obtenerse valores de $\boldsymbol{k}$ similares, se sospecha que su valor óptimo debe estar por ese rango.

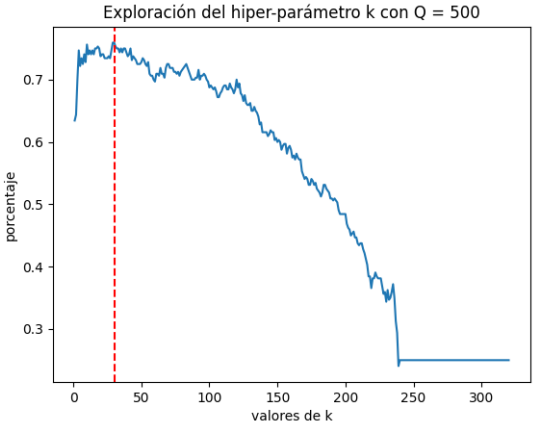

Figura 7

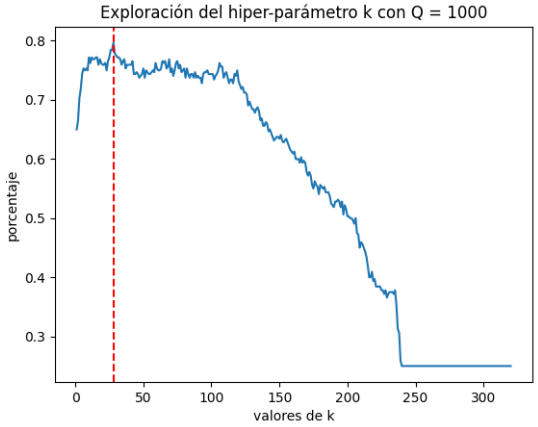

Figura 8

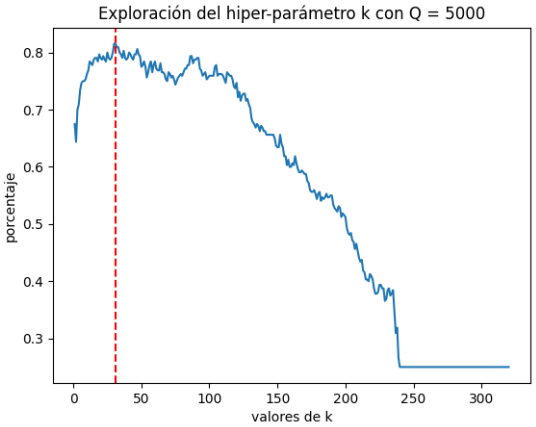

Figura 9

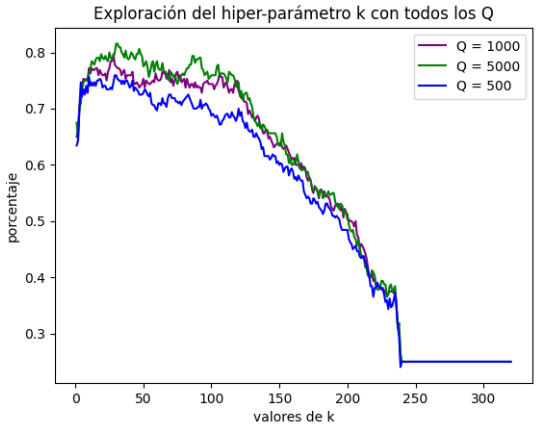

Figura 10

[16] KNeighborsClassifier
[17] sklearn.neighbors

3.3. PCA

Para el algoritmo de PCA se nos pidió visualizar la cantidad de vaianza explicada en función de la cantidad de componentes $\boldsymbol{p}$

En el análisis de componentes principales, la Figura 11 muestra un gráfico que representa la proporción de la varianza total de los datos explicada por los componentes principales seleccionados. A medida que aumenta el número de componentes principales también lo hace la varianza explicada acumulada. Se puede identificar el punto de *codo* de la curva, donde empieza a aplanarse, lo que indica que agregar más componentes principales no lleva a un aumento significativo en la varianza explicada acumulada.

En Figura 12 podemos ver como la varianza está distribuida entre los componentes principales individualmente y cómo los primeros son quienes capturan la mayor varianza y los últimos menos. Esto implica que los primeros componentes tienen un impacto mayor en la captura de información de la estructura de los datos.

En nuestro conjunto de datos, luego de aplicar PCA, se obtuvo 147 componentes principales. Podemos observar cómo luego de la curva, el gráfico empieza a aplanarse significando que al añadir mas componentes principales no incrementa la varianza explicada. Con esto, podemos reducir las dimensiones de nuestra matriz principal facilitando la visualización y el análisis de los datos. También mejora el rendimiento de los algoritmos de aprendizaje automático y con menos componentes, se evitan problemas como el sobreajuste, donde un modelo se adapta demasiado a los datos de entrenamiento y pierde capacidad de generalización.

Es por esto que para los $\boldsymbol{p}$ que probamos para encontrar la mejor pareja de $\boldsymbol{p}$ y $\boldsymbol{k}$, se decidió tomar esos valores. Se consideró que a partir de 320 aproximadamente ya no vale la pena porque a partir de 147 notamos una decrementación de la varianza. Además, con este resultado se sospecha que el valor óptimo de $\boldsymbol{p}$ está aproximadamente por ese rango.

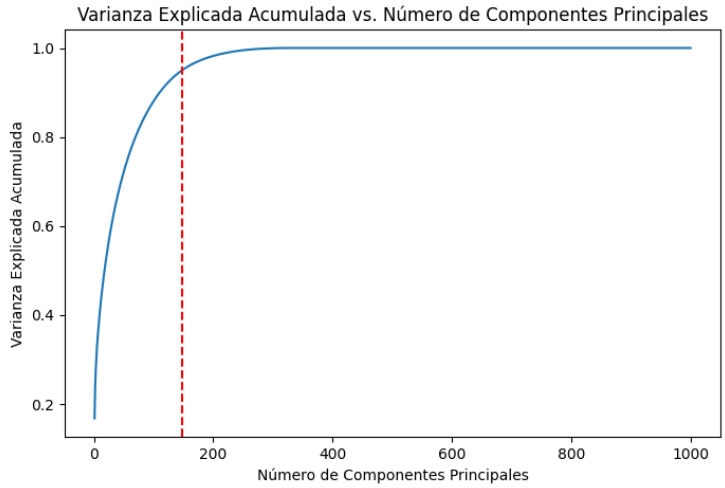

Figura 11: Gráfico de la varianza acumulada en función del número de componentes

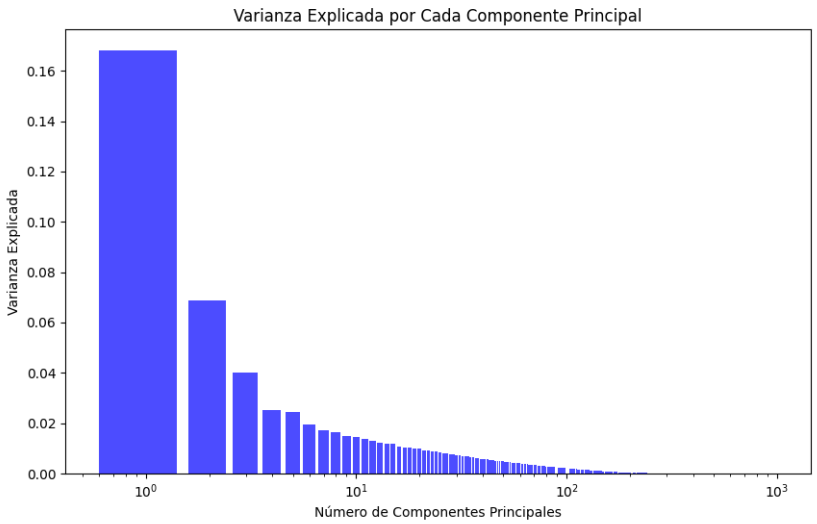

Figura 12: Gráfico de la varianza acumulada por cada componente principal

3.4. Reconocedor y optimización

Después de ejecutar el algoritmo 8 el cual detallamos toda su implementación en la Subsección 3.4, encontramos el par más óptimo para los parámetros $\boldsymbol{k}$ y $\boldsymbol{p}$. El valor óptimo de $\boldsymbol{p}$ es 20, y el de $\boldsymbol{k}$ es 5. Esto significa que reducimos nuestro conjunto de datos original (x_{train}) a 20 componentes principales y consideramos los 5 vecinos más cercanos para determinar a qué grupo pertenece la película que queremos reconocer. Este par de parámetros los utilizamos para el algoritmo 9. En este algoritmo realizamos PCA sobre todo nuestro conjunto de datos x_{train}. Luego, utilizamos un nuevo conjunto de datos x_{test}, que nunca habíamos usado para entrenar, con el fin de evaluar nuestro reconocedor de lenguaje. Finalmente, obtuvimos un porcentaje de rendimiento del 80 %, lo que indica que nuestro modelo tiene buenas probabilidades de reconocer correctamente una película.

En Figura 13 tenemos en los ejes del heatmap representan los diferentes valores de $\boldsymbol{p}$ (componentes principales) y $\boldsymbol{k}$ (vecinos). Los valores en las celdas del heatmap representan el rendimiento promedio del modelo KNN para cada combinación de $\boldsymbol{p}$ y $\boldsymbol{k}$. Este rendimiento es una medida de precisión, que varía entre 0 y 1, donde 1 indica un rendimiento perfecto. Estos valores los vemos representados con los colores y al costado tenemos las magnitudes.

Por otro lado, al aumentar $\boldsymbol{k}$ hacia valores más altos (aproximadamente mayores a 30) o al reducir $\boldsymbol{p}$ por debajo de 10, el rendimiento disminuye, como se observa en las áreas de color verde y morado. Esto podría indicar que un número excesivo de vecinos genera un efecto de "promediado"que reduce la efectividad del modelo, posiblemente añadiendo ruido en lugar de información útil para la clasificación.

La combinación de un bajo número de vecinos ($\boldsymbol{k} = 5$) y una cantidad moderada de componentes principales ($\boldsymbol{p} = 20$) permite que el modelo sea lo suficientemente preciso sin aumentar la complejidad innecesariamente. Esto sugiere que el modelo evita problemas de sobreajuste y generaliza bien a datos no vistos.

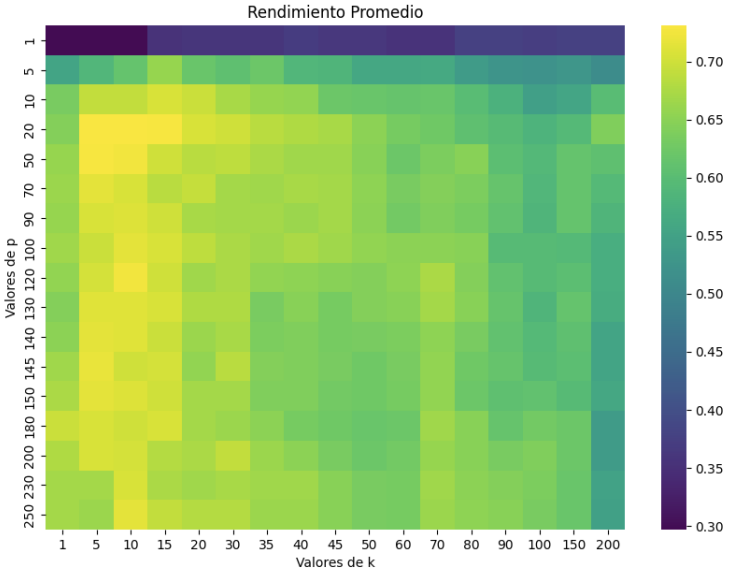

Figura 13

4. Puntos opcionales

4.1. TF-IDF

TF-IDF son las siglas en inglés de **Term Frequency-Inverse Document Frequency**, un método que ayuda a resaltar las palabras más significativas en un documento dentro de un conjunto de documentos (corpus). En nuestro contexto, el corpus son todos los tokens de las n películas.

Para calcularlo, primero observamos TF (frecuencia de término), que mide cuántas veces aparece una palabra en un documento en comparación con el total de términos en ese documento. Se calcula como:

$$TF(t,d) = \frac{\text{Número de veces que aparece el token en el documento } d}{\text{Número total de tokens en el documento } d}$$

A continuación, calculamos IDF (Inversa de la Frecuencia de Documento) para evaluar la importancia de un término en el corpus. Este valor penaliza los términos que aparecen en muchos documentos y se calcula de la siguiente manera:

$$IDF(t) = \log\left(\frac{N}{df(t)}\right)$$

donde: - N es el número total de documentos. - $df(t)$ es el número de documentos que contienen el término t.

Finalmente, una vez que tenemos TF e IDF, podemos calcular TF-IDF:

$$TF - IDF(t,d) = TF(t,d) \cdot IDF(t)$$

Dado que ya contamos con una matriz de conteo de tokens de nuestras películas, podemos aprovechar esa estructura para calcular **TF-IDF**. Para calcular IDF, primero contamos el número de documentos que contienen cada token y luego aplicamos la fórmula mencionada.

Algorithm 10 Cálculo de TF-IDF

```
function TF_IDF(A)
    longitudes_documentos =SumaPorFilas(A)        ▷ Suma por filas para obtener la longitud
    TF = A / longitudes_documentos
    N = longitud(A[0])                            ▷ Total de documentos
    df = suma(A >0)                               ▷ Número de documentos que contienen cada término
    IDF = log(N/(df+1))                           ▷ +1 para evitar log(0)
    TF_IDF = TF * IDF
    return TF_IDF
end function
```

Antes de aplicar validación cruzada del algoritmo 8, se le aplicó TF-IDF a la matriz de x_{train}. En Figura 15 podemos ver que con la aplicación del algoritmo resultó que la matriz de promedios tiene valores más altos que los de Subsección 3.4. A la hora de encontrar el mejor par de valores ***(p, k)***, obtuvimos resultados diferentes a la sección anterior.

	sin TF-IDF	con TF-IDF
valor p	20	180
valor k	5	50

Aunque la aplicación de TF-IDF inicialmente resultó en una matriz de promedios más alta en términos de rendimiento, la evaluación del modelo en el conjunto de prueba mostró un rendimiento inferior en comparación con el modelo que utilizó la matriz de conteo de palabras sin esta transformación.

	sin TF-IDF	con TF-IDF
Rendimiento	80 %	66.25 %

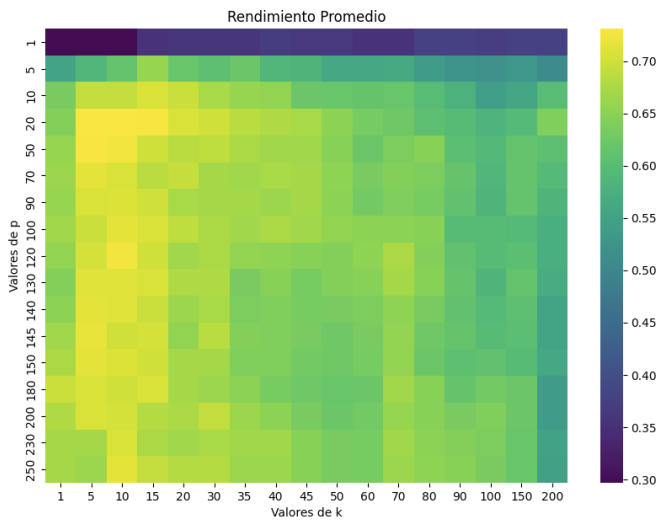

Figura 14: Rendimiento validación cruzada sin haber aplicado TF-IDF

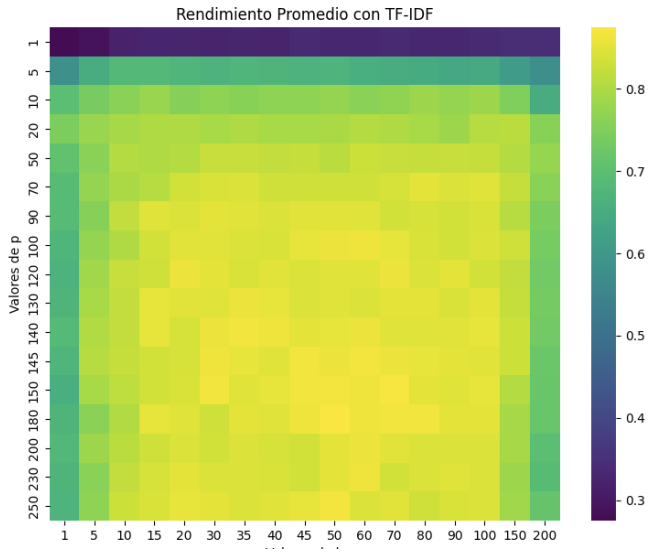

Figura 15: Rendimiento validación cruzada habiendo aplicado TF-IDF

Estos resultados pueden ocurrir ya que TF-IDF normaliza las frecuencias de términos, lo que puede resultar en la pérdida de información clave. Palabras que son comunes en el conjunto de datos, pero que podrían ser relevantes para la tarea de clasificación, reciben un peso menor. Esto puede afectar negativamente la capacidad del modelo para identificar patrones importantes en los datos. Es posible que la transformación con TF-IDF haya llevado a un sobreajuste en el conjunto de entrenamiento, donde el modelo aprende a ajustarse demasiado a los datos específicos de entrenamiento sin generalizar bien a los datos no vistos. Esto se traduce en un mejor rendimiento en el conjunto de entrenamiento pero en un menor rendimiento en el conjunto de prueba. Esta observación subraya la importancia de considerar no solo la técnica de preprocesamiento elegida, sino también su interacción con el modelo y su capacidad para aprender y generalizar a partir de las representaciones de datos.

4.2. KNN distancia euclídea vs distancia coseno

Se pidió comparar el algoritmo de KNN comparando la distancia coseno con la distancia euclídea. Para poder hacer esto primero recordemos cuál es la distancia euclídea:

$$d_{\text{euclidiana}}(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Para poder optimizar este cálculo usando operaciones matriciales de Numpy, podemos aprovechar la álgebra lineal y evitar bucles como hicimos con la distancia coseno. Una forma es utilizar la expansión de la fórmula de la distancia euclidiana:

$$d(x, y) = (x - y)^2 = x^2 + y^2 - 2xy$$

Algorithm 11 Distancia euclídea

1: **Parámetros: A, X**
2: A_{norma} = norma(A)
3: X_{norma} = norma(X)
4: producto = productoMatriz(A, traspuesta(X))
5: res = $\sqrt{A^2_{norma} + X^2_{norma} - 2 * producto}$
6: **return** res

Para verificar que la implementación sea correcta, al igual que en Subsección 3.2, se probó con un $\boldsymbol{k} = 5$ fijo, medir la performance y comparar los resultados con los de la librería *KNeighborsClassifier*[18] de *sklearn.neighbors* [19] y efectivamente lo es. Los resultados fueron:

Valor de Q	Coseno	Euclídea
500	65.625 %	50 %
1000	75 %	45.312 %
5000	68.75 %	40.625 %

La primera observación que se puede hacer es que la efectividad de predicción es mucho menor que a la distancia coseno.

Luego, también se pidió explorar el hiperparámetro $\boldsymbol{k}$ y al igual que en Subsección 3.2, se utilizó 4-fold cross-validation para cada valor de $\mathcal{Q} \in \{500, 1000, 5000\}$ y se midió la performance. Se probaron valores de $\boldsymbol{k}$ del 1 al 320 y los valores óptimos fueron:

Valor de Q	Coseno	Euclídea
500	75.937 %	54.687 %
1000	79.688 %	50.937 %
5000	81.562 %	44.99 %

En ambas pruebas que realizamos podemos ver que a medida que aumentamos la dimensión de nuestra matriz de conteo, la efectividad disminuye. Esto se debe a que en espacios de alta dimensión, los puntos de datos tienden a estar más dispersos lo que significa en términos de distancia euclidiana, los puntos que antes estaban cerca pueden parecer más lejanos y esa es la causa de la baja eficacia de la distancia euclídea en nuestro contexto. Es por esto que para poder clasificar las películas por género a partir de los tokens, es más efectivo utilizar la distancia coseno.

En Figura 16 podemos notar la diferencia de la performance entre una distancia o la otra.

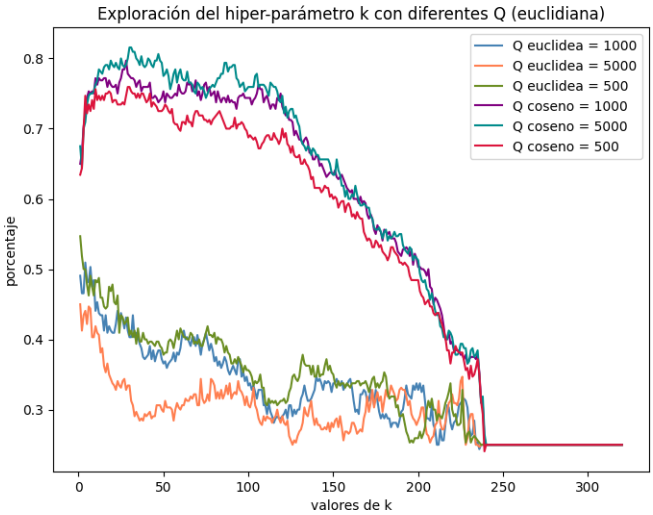

Figura 16: Gráfico con las performance de distancia coseno y euclidiana

[18] KNeighborsClassifier
[19] sklearn.neighbors

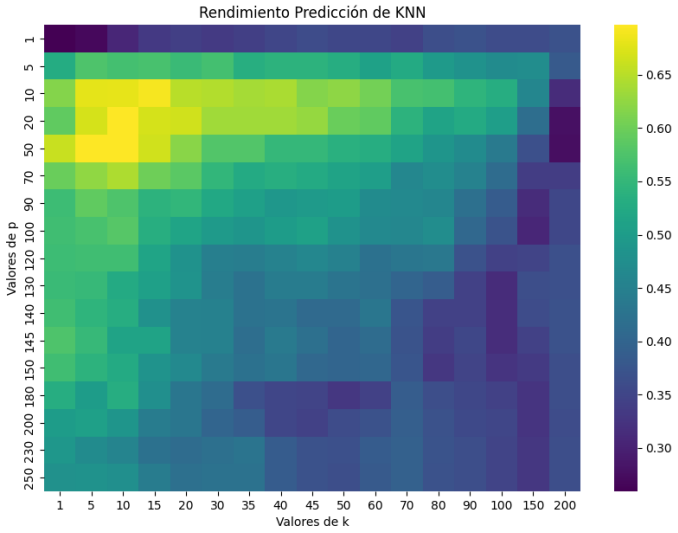

Figura 17: Rendimiento de Predicción de KNN con Distancia Euclídea

Se realizó validación cruzada al igual que en Subsección 3.4, pero ahora realizando KNN con la distancia euclídea de 11. En este caso, el mejor par de valores ***(p, k)*** encontrados fueron $\boldsymbol{p} = 50$ y $\boldsymbol{k} = 5$. También se evaluó el rendimiento de predicción y se obtuvo un rendimiento de 73.75 %.

	Coseno	Euclídea
valor p	20	50
valor k	5	5
rendimiento	80 %	73.75 %

Estos resultados sugieren que, para este conjunto de datos, la distancia coseno podría ser más adecuada debido a su capacidad para captar mejor las relaciones angulares en espacios de alta dimensión, donde las magnitudes pueden no ser tan relevantes como las direcciones en el espacio de características.

Observando la Figura 17, se nota que el rendimiento de predicción varía significativamente con los valores de $\boldsymbol{p}$ y $\boldsymbol{k}$. Valores de $\boldsymbol{p}$ muy bajos pueden llevar a que el modelo no capture suficiente información de los datos, resultando en un subajuste. Por otro lado, valores de $\boldsymbol{p}$ demasiado altos pueden incluir características irrelevantes, aumentando el ruido y potencialmente llevando a un sobreajuste. En cuanto a $\boldsymbol{k}$, un valor muy bajo puede hacer que el modelo sea demasiado sensible al ruido en los datos, mientras que valores altos pueden suavizar excesivamente la clasificación, afectando negativamente la precisión.

En conclusión, el análisis comparativo entre ambas distancias y el ajuste de parámetros demuestra la importancia de seleccionar cuidadosamente tanto la métrica de distancia como los hiperparámetros en KNN.

4.3. Variante validación cruzada

En esta variante, seguimos dividiendo el conjunto de datos en k particiones, pero en lugar de calcular y promediar los rendimientos de cada partición de prueba de forma independiente, se hace lo siguiente:

- **Predicciones y Agregación de Particiones de Prueba:** Cada vez que se entrena el modelo en k-1 particiones, se genera una predicción que es específico para esas particiones de datos. Se evalúan todas las predicciones en un solo conjunto combinado de las particiones de prueba.
- **Cálculo de la Métrica Final en el Conjunto Combinado:** En lugar de obtener una métrica de rendimiento separada para cada fold y luego promediar esos valores, evaluamos el rendimiento total en este conjunto de prueba combinado y calculamos una métrica de rendimiento única. Esto lo podemos hacer porque como estamos tratando de predecir todo el x_{train}, ya sabemos a qué categoría pertenece cada película.

El código para implementarlo es muy similar a algoritmo 8 pero con unas modificaciones en el for de *valoresK* y un nuevo for al final para medir las performance.

- *knn_predict* Es igual al algoritmo knn pero en lugar de devolver el promedio, devuelve el vector de las predicciones.

Algorithm 12 Variante validación cruzada pt1

```
Antes de entrar al for 0...4 creamos vector
todas_predicciones = arrayDeMatrices((longitud(valoresP), longitud(valoresK), n)
mismo código que algoritmo 8 hasta línea 18
for k ∈ valoresK do
    predictions = knn_predict(x_train_redudicod, x_testReducido, y_train[trainIndices], k, distancia)
    todas_predicciones[p, k, testIndices] = predictions
end for
```

Algorithm 13 Variante validación cruzada pt2

```
matrizRendimientos = arrayDeMatrices((len(valoresP), len(valoresK)))
for p ∈valoresP do:
    for k ∈ valoresK do:
        res = todas_predicciones[p, k, :] == y_train
        matrizRendimientos[p, k] = promedio(res)
    end for
end for
mejorP, mejorK = mejorPar(valoresP, valoresK, matrizRendimientos)
return mejorP, mejorK
```

A la hora de comparar resultados con ambas implementaciones de validación cruzada, si miramos los dos heatmap, notamos que los rendimientos máximos y la distribución de rendimiento son sorprendentemente similares en ambos métodos, a pesar de que la variante combinada utiliza un enfoque que debería, en teoría, aprovechar mejor los datos. En ambos heatmaps, el rendimiento máximo se encuentra alrededor de 0.72, logrando su mejor resultado con $\boldsymbol{p} = 20$ componentes y con valores de $\boldsymbol{k}$ bajos como 5 o 10. Esto sugiere que el modelo se beneficia de una dimensionalidad reducida sin perder información importante, y de un número de vecinos moderado, evitando el sobreajuste.

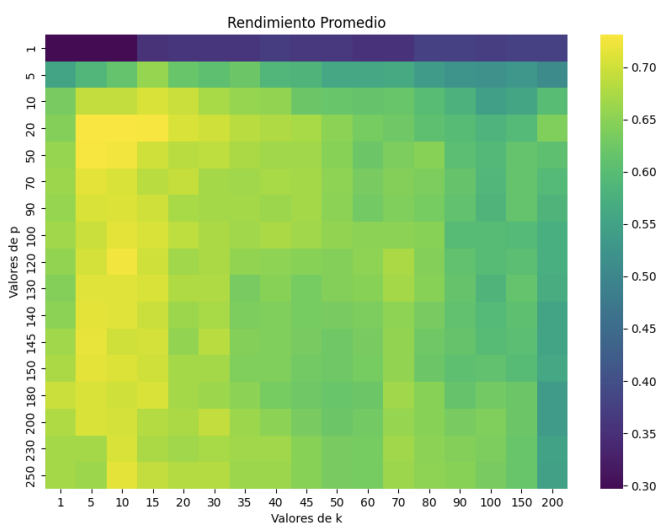

Figura 18: Rendimiento Promedio

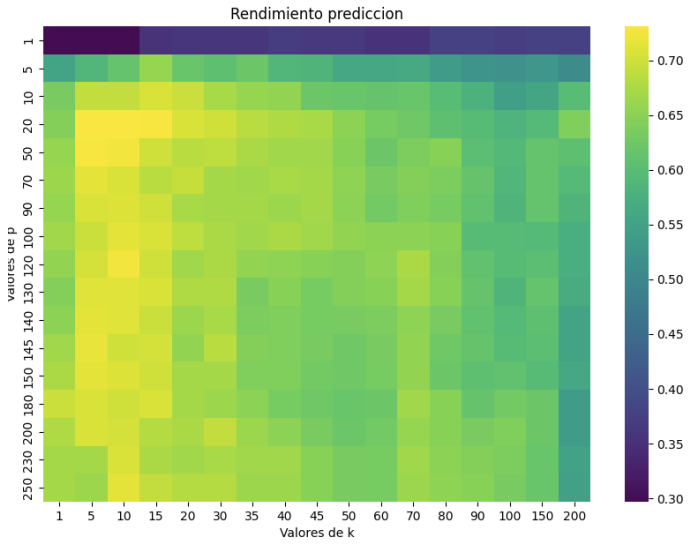

Figura 19: Rendimiento Predicción

La similitud en los resultados puede indicar que el conjunto de datos y el modelo son relativamente estables, es decir, que el modelo no sufre grandes variaciones en rendimiento cuando cambia la partición de entrenamiento y prueba. Esta estabilidad suele ser una buena señal, ya que implica que el modelo generaliza bien y no depende excesivamente de una única partición de datos para obtener buenos resultados.

El uso de PCA como paso previo al KNN puede estar contribuyendo a esta consistencia. La reducción de dimensionalidad a través de PCA selecciona los componentes que más varianza explican en el conjunto de datos, lo cual ayuda a eliminar el ruido y preservar los patrones relevantes. Esto estabiliza las características que se utilizan en el clasificador, haciendo que la efectividad del KNN dependa menos de las pequeñas variaciones entre las particiones.

En algunos conjuntos de datos, la naturaleza de las muestras hace que la validación cruzada tradicional tenga más sentido porque las diferentes particiones presentan variaciones significativas. Sin embargo, si el conjunto de datos es altamente representativo en todas sus partes, como parece ser nuestro caso, ambos métodos producen resultados similares.

5. Conclusiones

Luego de realizar todas las experimentaciones, podemos llegar a varias conclusiones. Una de ellas es que el algoritmo KNN es bastante ineficiente ya que aunque se trabaje en un mismo conjunto y sólo varíe la cantidad de vecinos que observamos, vuelve a calcular todas las distancias de cero. Lo óptimo sería calcular las distancias de cada elemento de nuestro conjunto una vez y luego calcular los porcentajes cambiando la cantidad de elementos cercanos que miramos. Además, puede llegar a presentar problemas en datasets que no tengan los datos balanceados, no es nuestro caso pero podría suceder. Utilizando el clasificador KNN, evaluamos la efectividad del modelo para clasificar géneros de películas, experimentando con tres diferentes tamaños de tokens: **Q = 500, Q = 1000, y Q = 5000**. Los resultados mostraron que, aunque el uso de 1000 tokens mejoraba el rendimiento en comparación con 500 tokens, no se observó una mejora significativa al aumentar a 5000 tokens, sugiriendo que un mayor número de características no garantiza una mayor precisión en la clasificación.

Por otro lado, con el método de la potencia con deflación hemos visto que se vuelve más lento al tener una matriz cuyos autovalores son muy cercanos ya que en este caso va a terminar por cantidad de iteraciones y no por el método de convergencia. Además, puede ser que la cantidad de iteraciones que le pasemos a la función no sean suficientes para aproximar los autovectores y autovalores, por ende el error podría ser mayor.

Luego, Con PCA vimos que es óptimo para reducir la dimensionalidad de los datos, así mejorando la eficiencia de los algoritmos. Esto puede jugar a favor o en contra ya que se corre el riesgo de eliminar información valiosa. Nuestro enfoque se centró en visualizar la proporción de varianza explicada en función de la cantidad de componentes seleccionados. Observamos el "punto de codo" en el gráfico de varianza acumulada, el cual indica el número de componentes óptimos para capturar la mayor parte de la información sin redundancia significativa.

Para obtener el mejor par de valores ***(p, k)*** combinando PCA y KNN, se utilizó la validación cruzada, lo cual nos permitió observar que, al aplicar PCA antes de KNN, se reducía la dimensionalidad eliminando componentes irrelevantes y mejorando así la precisión de KNN. Sin embargo, un desafío clave de esta combinación es que probar todos los valores posibles de $\boldsymbol{k}$ y $\boldsymbol{p}$ es muy costoso en términos de cómputo. Es por esto que nos enfocamos en un conjunto reducido de valores basándonos en resultados previos y en criterios de selección que consideramos razonables. Aun así, esto implica el riesgo de no encontrar el par $\boldsymbol{k}$ y $\boldsymbol{p}$ óptimo absoluto, ya que es posible que algún valor no explorado pudiera haber ofrecido un rendimiento superior.

Al incorporar TF-IDF en el preprocesamiento de la matriz de conteo de palabras, se logró inicialmente una mejora en la matriz de promedios de rendimiento. Sin embargo, los resultados finales en el conjunto de prueba mostraron una disminución en la precisión, bajando a un 66.25 %. Esto sugiere que la normalización que introduce TF-IDF puede penalizar términos clave para la clasificación en este contexto, afectando la capacidad del modelo para reconocer patrones importantes en el conjunto de prueba. Esto resalta la importancia de evaluar cuidadosamente el impacto de las técnicas de preprocesamiento en la generalización del modelo.

Además, se realizó una comparación entre las métricas de distancia coseno y euclidiana, mostrando que la distancia coseno ofrece un rendimiento significativamente mejor, especialmente al manejar datos de alta dimensionalidad y esparcidos, como los vectores de palabras en el modelo KNN. A pesar de una implementación optimizada de la distancia euclidiana, los resultados en términos de rendimiento no alcanzaron los niveles de precisión obtenidos con la distancia coseno. Esto respalda el uso de la distancia coseno como métrica preferida en tareas de clasificación de texto o cuando los datos están normalizados en magnitud.

En general, estos experimentos subrayan la importancia de la selección de hiperparámetros y métricas adecuadas, así como la relevancia de considerar cuidadosamente el preprocesamiento aplicado, ya que estas decisiones impactan directamente en el rendimiento y la capacidad de generalización del modelo de reconocimiento de películas en el conjunto de prueba.

Referencias

[1] URL: https://medium.com/analytics-vidhya/understanding-k-nearest-neighbour-algorithm-in-detail-fc9649c1d196 (visitado 02-09-2024).

[2] URL: https://ieeexplore.ieee.org/mediastore/IEEE/content/media/9339608/9339597/9339676/nurra3-2020_87-small.gif (visitado 02-09-2024).