DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico #2, Clasificador de películas

31 de Octubre de 2024 — Métodos Numéricos

Integrante	LU	Correo electrónico
Daniela Leilany del Carmen Melendez Rangel	102/20	byleilany@gmail.com
Valentino Murga	1375/21	valenmurga23@gmail.com
Thiago Tiracchia	1502/21	thiago.tiracchia@gmail.com
Grupo 11		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Índice

1. Introducción

El objetivo de este trabajo va a ser desarrollar un clasificador de películas a partir de su resumen sacado de Wikipedia. Las vamos a estar clasificando en 4 categorías: crimen, romance, ciencia ficción y western. Para esto vamos a usar distintos métodos de manipulación de datos y clasificación matriciales. Nuestro conjunto de datos va a estar partido en dos secciones, una de entrenamiento y otra de prueba, ambas con la misma proporción de películas de cada género. Durante la gran mayoría del trabajo nos vamos a manejar solo con el subconjunto de entrenamiento y solo vamos a usar el de prueba al final para verificar la implementación y obtener resultados reales.

Para empezar, vamos a necesitar encontrar una manera de convertir el texto en lenguaje humano en algo que se pueda representar computacionalmente. Para eso vamos a usar algunas técnicas simples que nos permiten reducir el texto a sus "tokens" básicos. Esto significa eliminar las palabras que no agregan contexto y normalizar las distintas versiones de la misma palabra (conjugaciones, género, pluralidad, etc.). Luego para convertir a cada película en un vector, vamos a asignarles una posición a cada palabra y su valor será la cantidad de repeticiones. Es por esto que a lo largo del trabajo nos vamos a estar refiriendo a la cantidad de palabras que hay en un vector como "dimensiones". Juntando todas las películas (muestras) de nuestro conjunto de entrenamiento, nos queda una matriz que tiene a cada película como filas y las apariciones de las palabras como columnas.

Una vez tenemos a las muestras representadas de forma vectorial / matricial, elegimos un criterio para compararlas. Nosotros vamos a usar la distancia coseno, un valor que aumenta en módulo cuanto más parecida es la dirección de los vectores y se acerca a 0 si son más perpendiculares. Pudiendo comparar, vamos a usar el algoritmo KNN para clasificar una muestra nueva a partir de un conjunto de muestras para las cuales ya conocemos sus clases. La idea básicamente va a ser tirarla en el medio del campo de "vecinos" y fijarnos a quienes tiene en sus proximidades. Se termina clasificando según la clase que sea mayoría en su cercanía.

El problema de KNN es que no escala, si tenemos matrices enormes deja de ser viable. Para esto vamos a recurrir a PCA. Este método nos va a permitir reducir el tamaño de nuestros datos eliminando redundancia. Para poder implementarlo, vamos a necesitar tener algún algoritmo que calcule los autovectores de una matriz. Vamos a usar el método de la potencia con deflación que, a partir de un vector aleatorio, lo multiplica con la matriz que nos interese múltiples veces haciendo que tienda al autovector asociado al autovalor dominante. Para poder hacer esto para cada autovalor, tenemos que desinflar la matriz de forma que el autovalor dominante pase a ser el siguiente.

Una vez tenemos nuestro clasificador de películas con KNN y PCA, falta encontrar los mejores valores para los parámetros que toma, que serían la cantidad de vecinos a mirar para KNN y la cantidad de componentes principales para PCA. Estos son conocidos como "hiperparámetros ya que no son generales para cualquier situación, sino que dependen de como estén estructurados los datos de un caso concreto. Para encontrar los valores ideales de estos hiperparámetros vamos a analizar ciertos puntos claves del proceso que nos permitan reducir las posibilidades y luego vamos a testear todas las que nos queden. Finalmente nos quedamos con las que den los mejores resultados sobre nuestro conjunto de entrenamiento y probamos su rendimiento sobre el conjunto de prueba.

2. Desarrollo teórico

2.1. Procesamiento de lenguaje

Previo a poder empezar a desarrollar la herramienta que nos dejará predecir el genero de una película dado su sinopsis, tendremos que, de alguna forma, entrenar el modelo con muchas sinopsis de otras películas para poder llevar a cabo la tarea.

Para eso vamos a modelar las descripciones de las películas usando matrices pero para eso necesitamos realizar el pasaje de lenguaje natural a matriz. Dado un texto, lo primero que vamos a hacer es normalizarlo. Esto implica, quitar puntuaciones y palabras vacías, aquellas que no agregan contexto y por lo tanto no sirven para entender mejor un resumen. Luego habrá que lematizar el texto, lo cual quiere decir que, dado una palabra, se hallará el lema correspondiente de dicha palabra pasando de plural a singular para los sustantivos, masculino singular para adjetivos e infinitivo para verbos, y luego se la remplazará por dicho lema. Un ejemplo pueden ser las palabras "gatos - gata - gatas", para las cuales el lema será "gato". Resumidamente, se estandarizaran todas las palabras del texto dado y con esto se generaran "tokens" de las

palabras. Luego, se contarán las palabras que hay en un texto con su cantidad de apariciones. En el ejemplo anterior, si "gato" o sus formas no estandarizadas aparecen 4 veces, el token de "gato" tendrá como valor 4 para la película dada. A cada token distinto le asignamos una dimensión.

Una vez procesadas todas las muestras, cada película será representada como un vector con tantas dimensiones como tokens generamos en nuestro sistema. Eso significa que cada vector tiene muchísimas dimensiones de las cuales la gran mayoría van a ser nulas porque sus tokens asociados no formaban parte de su resumen. Como las palabras que sean demasiado únicas no nos van a servir para distinguir entre clases de películas, podemos descartarlas y quedarnos con una cantidad de dimensiones mucho menor para los tokens más usados. Juntando todos los vectores obtenemos una matriz que tiene tantas filas como muestras (películas) y tantas columnas como tokens únicos a considerar. También en nuestra implementación tendremos una estructura auxiliar para guardarnos el género de la cada película de nuestro conjunto de datos, pero eso lo vamos a obviar a lo largo del trabajo ya que se vuelve muy implementativo y no agrega valor a las explicaciones. Simplemente consideramos que cada muestra de entrenamiento está atada a su género correspondiente y eso es algo que tenemos forma de acceder.

Por ejemplo, si el token 3 refiere a la palabra "gato" y la posición $(2, 3)$ de nuestra matriz contiene un 2, eso querrá decir que en la descripción de la película número 2 la palabra "gato" (o sus formas no estandarizadas) aparece 2 veces. Una vez hecho esto todo este proceso, podemos aplicar métodos y algoritmos matriciales para construir nuestro clasificador.

2.2. K vecinos más cercanos

Vamos a usar un método de clasificación simple, KNN (K Nearest Neighbors - K Vecinos Más Cercanos). Es un algoritmo fácil de implementar y que funciona bien y rápido para conjuntos de datos con pocas dimensiones. El mayor problema que tiene es que sufre mucho a medida que aumenta esa cantidad de dimensiones, volviéndose particularmente lento. Este es un problema del que nos vamos a ocupar más adelante con PCA.

La idea del algoritmo consiste en representar nuestro modelo, un conjunto de muestras para las que conocemos sus clases, como puntos en un espacio. Luego, para un nuevo dato que queramos clasificar, lo metemos en ese espacio y nos fijamos a quienes tiene en su cercanía. La clase que predomine en sus alrededores, la moda, es la que tiene mayor probabilidad de ser la correcta. Por ejemplo: Si tenemos una película para la cual no sabemos su género, la colocamos en el espacio y observamos a sus 7 vecinos más cercanos. Supongamos que son 3 películas de clase "romance" y 4 películas de clase "western". Lo que sucederá es que a la nueva película se la clasificará como western porque, de sus 7 vecinos mas cercanos, la clase que mas "peso" tiene es la de western. La cantidad de vecinos que observamos para tomar esta decisión (7 en el ejemplo) es nuestro hiperparámetro k, y decidir cuál es su valor óptimo es un problema que vamos a tener que analizar en profundidad a partir de estudiar nuestro conjunto de datos con su distribución y probar manualmente para obtener resultados.

Para que todo esto funcione tenemos que definir algo muy importante, lo que significa que dos muestras estén "cerca". Para nosotros, dos muestras están cerca cuando sus resúmenes se parecen y, por lo tanto, tienen más probabilidades de ser del mismo género. Digamos que tenemos una película A de género western que contiene 5 apariciones del token "sombrero", 3 del token "revolver" y 4 de "caballo". Nos llega película B cuyo género es desconocido pero tiene un valor parecido en esos mismos tokens. Se podría concluir que esta película B también sería de genero western ya que su descripción se parece a la de una que sabemos que lo es. Sin embargo, esto no es infalible y requiere tener en cuenta a todas las dimensiones con cuidado. Por ejemplo, si existiera una película de género romance ambientada en el Lejano Oeste, tendría sentido que sus descripciones repitan palabras del genero western y el algoritmo podría clasificar mal la película.

Hay dos formas de medir esta distancia matemáticamente: la distancia euclidiana y la distancia coseno. La distancia euclidiana puede parecer intuitiva porque es literalmente la distancia entre dos puntos en el espacio. Sin embargo, tiene un problema y es que dos vectores que se encuentren en la misma dirección pero con magnitudes distintas van a quedar lejos. Pensándolo con nuestro caso, esto no tiene mucho sentido porque si dos vectores usan las mismas palabras con la misma o similar distribución, teniendo como única diferencia el volumen de las mismas, probablemente pertenezcan al mismo género pero tengan resúmenes estructurados de forma distinta. Con esa intuición, podemos llegar a la conclusión de que lo que en realidad nos interesa es comparar las direcciones.

La forma matemáticamente formal de llegar a esta conclusión es analizar la covarianza entre dos vectores, es decir, que tan similar se comportan. La fórmula para la covarianza de dos muestras es

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

Donde n es la cantidad de muestras y μ_x y μ_y son los valores medios de los vectores, que al ser restados los terminan "centrando" para que no se queden restringidos a un solo cuadrante. Sin embargo, por la forma que tienen nuestros datos, en realidad nos conviene no centrarlos. Así que, si bien esa es la definición formal, en nuestra implementación vamos a evitar esa parte. De todas formas nosotros no vamos a usar esa cuenta de forma directa, sino que la vamos a derivar para llegar a la correlación, que es lo mismo pero normalizada para que quede entre -1 y 1.

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

Pero al hacer esa normalización, el resultado es exactamente lo mismo que el coseno del ángulo entre x e y. Eso significa que la covarianza depende de la dirección, que es lo que intuímos antes. Obtenemos valores altos (hasta 1) cuando los vectores tienen direcciones similares, negativos cuando van en sentidos opuestos (caso en el que "no se parecen") y valores cercanos a 0 cuando tienen direcciones más ortogonales. Finalmente, podemos correr los resultados de $[-1, 1]$ a $[0, 2]$ restándoselos a 1 y así llegamos a la fórmula de la distancia coseno que si es la que vamos a usar para comparar vectores en nuestro algoritmo.

$$D_{\text{coseno}}(x, y) = 1 - \text{Corr}(x, y) = 1 - \frac{x \cdot y}{\sqrt{x \cdot x}\sqrt{y \cdot y}}$$

Con esto ya tenemos todas las herramientas necesarias para desarrollar nuestro algoritmo clasificador usando KNN.

Algorithm 1 KNN

```
Entrada: vector, modelo, k
vecinos ← Lista vacía
for all muestra in modelo do
  vecinos.agregar(distanciaCoseno(muestra, vector))
end for
vecinos.ordenar() {De menor a mayor}
kvecinos ← vecinos[:k]
return kvecinos.moda().clase()
```

El algoritmo toma 3 parámetros:

- El vector que queremos clasificar. Mientras entrenemos va a ser un vector para el que igual conozcamos la clase para poder comparar con el resultado, pero en producción realmente sería desconocida.
- Nuestro modelo, que es un conjunto de muestras variadas para los que conocemos las clases y vamos a usar como direcciones de referencia en el espacio para tratar de aproximar la clase del vector a clasificar.
- El hiperparámetro k, la cantidad de vecinos a observar en la cercanía de nuestro vector para terminar de definir a que clase pertenece.

La forma de chequear cuáles son las muestras del modelos más cercanas a nuestro vector, simplemente calculamos las distancias entre el mismo y todas las muestras y luego las ordenamos de menor a mayor para quedarnos solo con las k primeras. Es importante volver a mencionar que a la hora de implementar esto es necesario que tengamos guardada en algún lugar la información de a que clase pertenecen las muestras de nuestro modelo. En el pseudocódigo simplemente usamos ".clase()" por claridad pero es toda una parte de los datos que siempre hay que mantener atada a su muestra correspondiente y no desordenarla accidentalmente.

Con eso concluye la explicación de KNN, ahora tenemos que tomar ciertas medidas para poder aplicarlo a conjuntos de datos muy grandes sin sufrir tanto los golpes al rendimiento por la dimensionalidad.

2.3. Método de la potencia con deflación

Para reducir las dimensiones de nuestro conjunto de datos vamos a querer usar PCA, pero para eso vamos a tener que diagonalizar una matriz y para eso tenemos que poder calcular los autovalores y autovectores de esa matriz. Ahí es donde llega el método de la potencia con deflación.

El metodo justamente se divide en las dos partes del nombre, empezando por la potencia. La idea es que si multiplicamos un vector cualquiera por alguna matriz cuadrada, el resultado se "tuerce" mayoritariamente en la dirección del autovector asociado al autovalor dominante λ_1 (el de mayor módulo) de esa matriz. Si repetimos el proceso muchas veces, retroalimentando el vector a multiplicar con el resultado de la multiplicación anterior, en algún punto el resultado va a ser lo suficientemente parecido al autovector. Formalmente, tenemos una sucesión $\{v^j\}$ definida como

$$v^j = \frac{Av^{j-1}}{||Av^{j-1}||}$$

Donde $j = 1...$, y v^0 es algún vector aleatorio con norma 1. Usamos vectores normalizados a lo largo del proceso entero para que no nos queden números muy grandes o muy chicos que empiecen a generar problemas de error numérico, la norma que usemos no importa salvo para cuestiones de optimización. Cuando j tiende a infinito, v^j converge al autovector asociado al autovalor dominante, que es lo que buscábamos. Una vez obtenemos ese autovector, se puede conseguir el autovalor asociado con una cuenta simple.

$$\lambda = \frac{v^T B v}{v^T v}$$

Hay un detalle no menor que es que la convergencia es teórica y ocurre cuando j tiende a infinito. Pero nosotros no podemos hacer infinitos pasos en un algoritmo, por lo que tenemos que decidir manualmente cuando frenar. Hay dos momentos en los que tiene sentido frenar: cuando tenemos un resultado que se parece lo suficiente al autovalor y autovector reales, o cuando hicimos demasiados pasos y tenemos que forzar el corte. Para evaluar si nuestros resultados del paso actual son lo suficientemente buenos, basta probar que la diferencia entre el vector actual y el vector del paso anterior es tan chica que muestre que estamos cerca de la convergencia. Eso es lo mismo que hacer la cuenta $v^j - v^{j-1}$ y esperar que el resultado sea lo más cercano a 0 posible. Como la igualdad seguramente no se cumpla porque eso solo ocurre en la convergencia, vamos a poner cierto nivel de tolerancia y tratar de que esa diferencia sea menor a ese nivel. Para asegurarnos de que todos los valores del vector están lo suficientemente cerca de 0, un buen criterio a usar es la norma infinito de la diferencia, que nos asegura que si el máximo valor es menor a la tolerancia, entonces todos los demás también lo son. Entonces cuando logramos pasar ese umbral frenamos, pero se podría tardar demasiado en llegar a eso, especialmente en casos en los que hay dos autovalores que se parecen mucho en módulo. Es por eso que ponemos un límite duro de pasos para evitar frenar todo lo demás cuando tenemos un autovector que tarda demasiado en converger.

Algorithm 2 Calcular autovalor dominante y autovector asociado

Entrada: A, tolerancia, límite
$v \leftarrow$ vector aleatorio con norma 1
$w \leftarrow$ nada {O un vector enorme para que la resta no de algo chico}
for i de 0 a límite mientras $||v - w||_\infty >$ tolerancia **do**
  $w \leftarrow v$
  $v \leftarrow \frac{Av}{||Av||}$
end for
$\lambda \leftarrow \frac{v^T B v}{v^T v}$
return λ, v

Solo queda una salvedad, que es que el vector inicial aleatorio no puede ser ortogonal al autovector, porque si fuera el caso entonces la sucesión no convergiría y en nuestro algoritmo terminaríamos devolviendo un vector que no tiene nada que ver con lo que buscamos y que seguramente no sea un autovector. El motivo por el que apenas mencionamos esto y por el que ni siquiera lo tenemos en cuenta en nuestra implementación, es que la probabilidad de que al elegir un vector aleatorio que justo sea ortogonal al autovector es prácticamente nula, así que suponemos que eso nunca va a ocurrir.

Todo esto sirve para conseguir el primer autovector y autovalor de la matriz, pero una matriz A cuadrada de tamaño $n \times n$ tiene n de esos. Además, no podemos simplemente volver a aplicar el mismo proceso otra vez porque estaríamos en las mismas condiciones iniciales y llegaríamos al mismo resultado (con un poquito de variación por la aleatoriedad pero mismo autovector y autovalor). Para poder pasar a los siguientes autovalores y autovectores, vamos a tener que hacer lo que se conoce como deflación. En particular vamos a usar la deflación de Hotelling, que con una sola cuenta nos permite conseguir una matriz que cumple lo que queremos.

La matriz $B = A - \lambda_1 v_1 v_1^T$ tiene los mismos autovalores y autovectores que la original menos λ_1 y v_1, los que acabamos de conseguir con el método de la potencia. Eso significa que ahora la matriz nueva tiene un nuevo autovalor dominante, en particular, el que le seguía a λ_1 en módulo. En esas condiciones, podemos hacer el método de la potencia sobre B y, como ni siquiera tuvimos que cambiar las dimensiones de nuestra matriz original, vamos a obtener el próximo autovalor y autovector de A. Después solo queda repetir el proceso n veces y terminamos consiguiendo todos.

Algorithm 3 Desinflar

Entrada: A, λ, v
return $A - \lambda_1 v_1 v_1^T$

Parecería que, salvo porque no podemos hacer infinitos pasos para converger del todo, el algoritmo funciona perfectamente. Sin embargo, para poder hacer la deflación tranquilamente hay que cumplir ciertos requisitos. Por un lado, la matriz debería tener todos sus autovalores distintos y, por lo tanto, poder ser ordenados estrictamente a partir de sus módulos. Por el otro, necesitamos que tenga una base ortonormal de vectores para que, a la hora de eliminar un vector cuando desinflamos, esa eliminación solo afecte a la dirección en la que apuntaba ese vector y no modifique ninguna de las demás de forma que sigamos teniendo la misma información que tenía la matriz original para seguir consiguiendo el resto de autovalores y autovectores. El segundo requisito es absolutamente necesario, pero el primero puede tener algunos casos específicos en los que no hace falta. De hecho, para nuestro caso de uso, no lo cumplimos pero vamos a ver que igual podemos usarlo. De todas formas si se quisiera conseguir los autovalores y autovectores de una matriz a la que no se le puede aplicar este método particular, hay otros más que pueden lograr lo mismo pero que posiblemente no sean tan sencillos.

Algorithm 4 Método de la potencia con deflación

Entrada: A, tolerancia, límite
res $\leftarrow$ lista vacía de tuplas {Pares de autovalor y autovector}
$n \leftarrow$ #Filas de A {= #Columnas porque es cuadrada}
for i de 1 a n **do**
  $\lambda, v \leftarrow$ calcularDominantes(A, tolerancia, límite)
  res.agregarAtrás($\{\lambda, v\}$)
  $A \leftarrow$ desinflar(A, λ, v)
end for

Finalmente, deberíamos verificar que nuestra implementación sea correcta. Para eso vamos a usar un truco en el construimos una matriz que ya sabemos que es diagonalizable y para la cual conocemos sus autovalores y autovectores. La idea es pensarlo al revés de lo normal, en vez de llegar de una matriz a su diagonalización, vamos a armar una diagonalización arbitraria y vamos a hacer las multiplicaciones para llegar a la matriz correspondiente a esa diagonalización. En la diagonal ponemos los autovectores que queremos conseguir elegidos arbitrariamente de forma que sean todos distintos. Para las matrices que van a contener los autovectores, no solo vamos a necesitar que sean inversibles, sino que también formen una base ortonormal para poder usar nuestro método. Una matriz que es fácil de construir que cumpla estas propiedades es una de Householder. Como es ortogonal, sus columnas (y filas) forman una base ortonormal y entonces nuestros autovectores van a cumplir nuestros requisitos. También es muy fácil encontrar la inversa de esta matriz porque la inversa de una matriz ortogonal es la traspuesta de la misma. Luego la matriz se construye con la fórmula

$$H = I - 2uu^T$$

Donde u es un vector aleatorio con norma 1. Finalmente usamos nuestra diagonal y matriz ortogonal para construir matrices aleatorias M para verificar nuestra implementación.

$$M = HDH^T$$

Para probar que los resultados son (suficientemente) correctos, probamos ver que en efecto sean los que generamos nosotros manualmente. Eso significa que los autovalores que obtuvimos sean los elementos que guardamos en D y que los autovectores sean los que guardamos en H, en particular sus columnas. Los resultados no van a dar exactos porque el método de la potencia se acerca todo lo que puede, así que en realidad nos vamos a fijar que estén lo suficientemente cerca. Podemos crear tantas matrices aleatorias como queramos y, con una tolerancia lo suficientemente baja, el algoritmo debería dar resultados que cumplan con lo requerido si está bien implementado.

2.4. Análisis de componentes principales

Ahora que tenemos una forma de conseguir los autovalores y autovectores para ciertas matrices, podemos armar un módulo de PCA (Principal Component Analysis - Análisis de Componentes Principales). El objetivo de PCA es reducir la cantidad de dimensiones de un conjunto de datos quedándonos solo con los "componentes" que retengan la mayor cantidad de información que represente a los datos originales. Esto sirve para tratar de reducir la redundancia de los datos todo lo posible y así disminuir los tiempos de cómputo, los requisitos de memoria e incluso el ruido que puede generar esa redundancia.

Pero tenemos que definir lo que significa que haya redundancia. La idea es bastante intuitiva, si alguna dimensión se parece mucho para la mayoría de las muestras, entonces esa dimensión no nos está aportando información útil. Justamente queremos quedarnos principalmente con las dimensiones que tengan más variación a lo largo de las muestras, ya que estas nos van a permitir encontrar patrones y distinguir las direcciones en las que podrían encontrarse las distintas clases. Análogamente, si dos dimensiones varían bastante pero de la misma forma para todas las muestras, entonces con una sola alcanzaba y la otra no agrega nada.

Estos conceptos en realidad ya los definimos antes. Que una dimensión cambie mucho a lo largo de las muestras significa que la variable de esa dimensión tenga una varianza alta. Por otro lado, que dos dimensiones se comporten de forma distinta significa que esas dos variables tienen covarianza baja. Entonces, si queremos reducir la redundancia, vamos a querer quedarnos con las dimensiones que tengan mayor varianza y menor covarianza con el resto. Así surge la matriz de covarianza $C \in R^{m \times m}$:

$$C = \frac{X^T X}{n-1}$$

Donde $X \in R^{n \times m}$ es nuestra matriz que tiene n muestras de m dimensiones. Si se desarrolla esa cuenta, se puede ver que la matriz C tiene las varianzas de las columnas de X en la diagonal y las covarianzas entre cada columna en el resto de posiciones. Como la varianza entre dos columnas es la misma en cualquier orden, se puede ver que C es simétrica. Con un poco más de trabajo también se puede demostrar que va a ser semidefinida positiva. Como es simétrica, tiene base de autovectores y por lo tanto puede ser diagonalizada. Esto es muy útil porque la base de autovectores puede formar una matriz de cambio de base que nos va a servir para transformar a nuestras matrices y vectores a partir de sus varianzas y covarianzas. Además, los autovalores contienen la información de cuanta varianza aportan sus autovectores asociados cuando se usan para generar una nueva dirección. Eso significa que, si ordenamos los autovectores de mayor a menor de acuerdo a la cantidad de varianza que aportan, vamos a poder recortar la matriz de cambio de base tanto como queramos para quedarnos solo con algunos de los vectores con más peso y hacer que nuestras transformaciones devuelvan matrices y vectores con menos dimensiones pero que representen bien a las muestras.

De ahí sale que las dimensiones transformadas serían los "componentes" y, en particular, los principales son aquellos que tienen la mayor cantidad de varianza y por lo tanto son más útiles para distinguir entre clases y separarlos en el espacio. Esto deriva en que PCA es un un proceso compuesto por 3 partes.

En primer lugar hay que construir la matriz de covarianza y luego diagonalizarla. Construirla es sencillo, la cuenta está más arriba. La parte más compleja y computacionalmente intensa es la diagonalización. Por suerte, es una matriz que cumple los requisitos para que se pueda aplicar el método de la potencia con deflación. Lo malo es que, dependiendo del tamaño de nuestros datos y de la tolerancia que tengamos, este proceso va a ser largo y no hay mucho que hacer al respecto. Este proceso se conoce como Fit. Lo bueno

es que, salvo a la hora de experimentar, esto solo hay que hacerlo una vez sobre el conjunto de datos que vayamos a usar como modelo, así que, aunque tarde mucho tiempo, a la hora de clasificar datos nuevos esto no va a afectar.

Algorithm 5 PCA Fit

Entrada: X {Matriz de muestras}
$C \leftarrow \frac{X^T X}{n-1}$ {n es la cantidad de filas de X}
$V \leftarrow$ potenciaYDeflación(C).autovectores {V variable de clase}

El siguiente paso es transformar la matriz haciéndole el cambio de base, lo cual es simplemente una multiplicación de matrices. Esto también se hace solo una vez para que quede como modelo y en este momento es donde hay que definir la cantidad de componentes principales que se van a querer usar, ya que la dimensión del resultado depende de eso y en el futuro vamos a tener que comparar con cosas de la misma dimensión.

Algorithm 6 PCA Transformar Matriz

Entrada: X, p {Cantidad de componentes principales}
return $X(V[:, :p])$

Finalmente, a la hora de usar nuestro algoritmo para efectivamente clasificar datos, vamos a tener que aplicar la misma transformación con la misma cantidad de componentes a esos datos. Eso también es solo una multiplicación de un vector por una matriz. Así logramos tener tanto un modelo como unos datos que viven en el mismo espacio pero con mucha menos redundancia que al inicio, teniendo un alto costo computacional solo a la hora de crear el modelo.

Algorithm 7 PCA Transformar Vector

Entrada: w, p
return $w^T(V[:, :p])$

Hasta ahí llega la parte teórica, pero a la hora de experimentar hay que tener varias cosas en cuenta, como que forma tienen los datos y que cantidad de componentes principales es la óptima para reducir la redundancia todo lo posible sin perder información importante. Esto último es otro hiperparámetro, como lo fue la k de KNN. De hecho, para nuestro caso nosotros vamos a combinar las dos técnicas. Por un lado, vamos a querer transformar nuestros datos usando PCA para achicarlos disminuyendo la redundancia, y luego vamos a meter esos datos transformados en KNN para terminar clasificándolos. Tiene sentido esperar que los resultados de KNN mejoren con PCA ya que cuando calculemos las distancias entre vecinos no vamos a tener en cuenta direcciones en las que se encuentren todos al mismo tiempo, las cuales hacen ruido y pueden llevar a que un vecino parezca estar más cerca de lo que debería.

3. Resultados

3.1. Velocidad de convergencia y error del método de la potencia

Una vez implementado el método de la potencia nos va a interesar explorar la cantidad de pasos que el algoritmo va a tardar en llegar los autovalores y autovectores buscados para poder tenerlo en cuenta a la hora de ajustar los parámetros del algoritmo para que no tarde demasiado. También es importante la medida de error que tienen los resultados, ya que si nuestros autovalores y autovectores no son lo suficientemente buenos podrían ser inútiles.

Para hacer estas mediciones vamos a usar una matriz con los autovalores $\{10, 10-\epsilon, 5, 2, 1\}$ y usaremos el mismo truco de las matrices de Householder que usamos para verificar la correctitud de nuestra implementación con la intención de, usando matrices aleatorias, los resultados deberían ser bastante generales, especialmente si hacemos múltiples mediciones. Para calcular el error, veremos el resultado de aplicarle la norma 2 a $||Av_i - \lambda v_i||$ para $i = 1 \dots 5$, que debería ser lo más cercano 0 posible porque, si los resultados son buenos, se supone que $Av_i = \lambda v_i$ porque son autovalores y autovectores asociados. Por último iremos variando el ϵ, espaciándolo logarítmicamente entre 10^-4 y 10^0 para analizar como afecta la cercanía entre

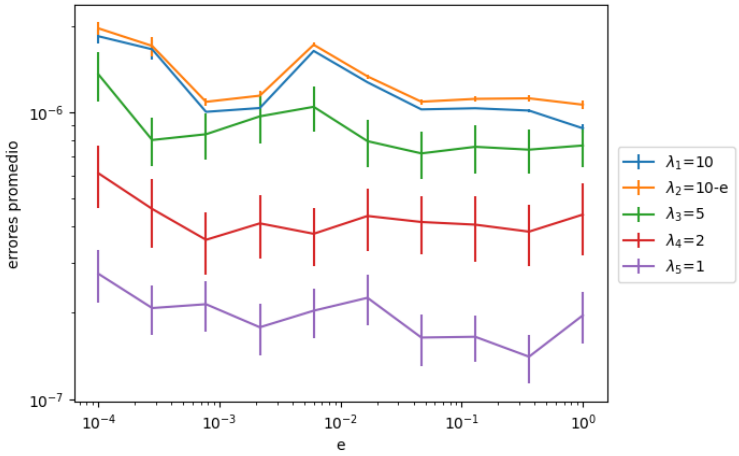

Figura 1: Errores promedio con desvío estándar para cada autovalor

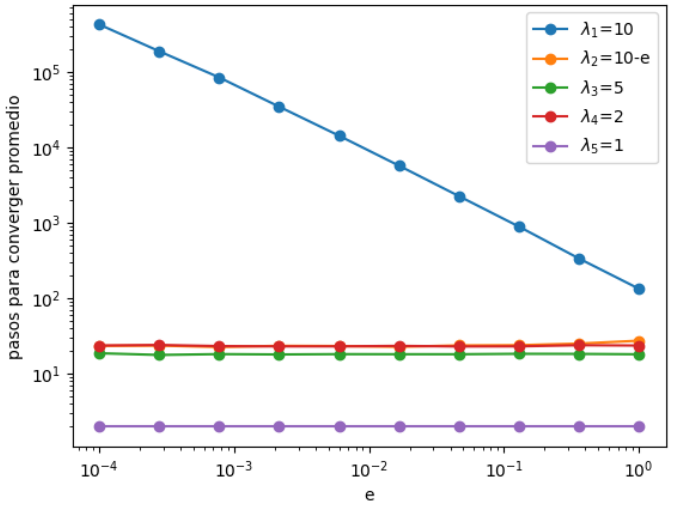

Figura 2: Pasos para converger para cada autovalor

dos autovalores los resultados. Realizaremos muchas mediciones con muchas matrices aleatorias distintas y tomaremos el promedio de estas mediciones con su desvío estándar.

Podemos ver los resultados de estos experimentos en las figuras 1 y 2.

Podemos observar que, para valores muy pequeños de ϵ, el primer autovalor tarda muchísimo más que los demás en converger. Esto tiene sentido, ya que en esos casos los primeros dos autovalores se encuentran muy cerca el uno del otro en módulo, osea que tienen "fuerzas" muy similares y van a hacer que sus autovectores asociados "tiren" en direcciones distintas y realenticen la convergencia. A medida que se agranda ϵ, empieza a haber más separación y el primer autovalor gana fuerza sobre el que le sigue, haciendo que se converja más rapido a su autovector asociado. El resto de autovalores tardan todos aproximadamente lo mismo porque tienen suficiente separación entre ellos así que no llegan a afectarse entre si. También es mínimamente curioso ver como el último autovalor siempre se consigue casi instantáneamente. Eso es porque, en el último paso, nos queda una matriz con un solo autovalor que puede transformar vectores hacia una sola dirección, por lo que con solo uno o dos pasos se consigue un vector en la dirección requerida.

En la otra figura, vemos que los errores son realmente pequeños. En realidad eso iba a estar asegurado porque a nuestro algoritmo le dimos un nivel de tolerancia específico, por lo que, siempre y cuando converja antes de llegar al límite de pasos, el error va a ser acorde a esa tolerancia. La única diferencia que podemos observar es que cuanto más grande es el módulo del autovalor, mayor es el error en promedio, cosa que seguramente se deba a cuestiones de error numérico por magnitudes más grandes y no algo específico de nuestro algoritmo. El desvío estándar también nos muestra que es un algoritmo muy consistente, ya que apenas hay desvío en todos los casos, recordemos que el gráfico está en escala logarítmica y las cosas se encuentran en el orden de 10^{-7}, por lo que las barras largas en realidad son minúsculas. Incluso si miramos los casos más altos que ya pasan al orden de 10^{-6}, se puede ver mejor que esas barras, que se veían más

grandes en un orden más bajo, son realmente muy chicas. Podemos concluir que tenemos un algoritmo que devuelve resultados consistentemente buenos y acordes a la tolerancia pedida.

3.2. Clasificador de películas usando KNN

Ahora toca probar nuestros clasificadores, que eran el objetivo del trabajo. Vamos a empezar por lo simple, un clasificador que use exclusivamente KNN con 5 como valor arbitrario para k. Refrescando, tenemos un conjunto de muestras de entrenamiento para las que conocemos sus clases y se encuentran representadas como vectores. De ese conjunto de datos, vamos a usar un 80 % como modelo y el otro 20 % como muestras a clasificar para luego contar la cantidad de aciertos. Es importante que ambas particiones tengan la misma proporción de muestras de cada clase para que el algoritmo no se desbalancee y tienda a ninguna clase en particular. También vamos a querer ir variando de forma aleatoria los datos que le tocan a cada partición para tratar de generalizar los resultados.

Vamos a realizar muchas mediciones cambiando los datos que se encuentran en cada partición y después sacamos el promedio para conseguir una aproximación a lo que podríamos esperar si usáramos este como nuestro clasificador final. También vamos a repetir el proceso para distintos tamaños, osea, cambiando la cantidad de dimensiones que tienen nuestras muestras. Si refrescamos el principio del trabajo, estas son los tokens con más apariciones en nuestro conjunto de datos. En particular vamos a probar con 500, 1000 y 5000 dimensiones de un total de 9581 que vienen con nuestro conjunto original post procesamiento de texto.

Habiendo realizado las mediciones, obtuvimos

- 66.69 % de aciertos para 500 dimensiones.
- 66.06 % de aciertos para 1000 dimensiones.
- 66.16 % de aciertos para 5000 dimensiones.

Parecería que para $k = 5$ la cantidad de dimensiones no termina de afectar a los resultados y siempre acertamos aproximadamente dos tercios de los intentos, lo cual no es particularmente malo para un algoritmo tan simple y un valor de k elegido arbitrariamente.

3.3. Validación cruzada y búsqueda de mejor k sin PCA

Probar el rendimiento para una cantidad de vecinos (el hiperparámetro k) arbitraria no es muy útil. Los resultados pueden variar bastante para distintos valores y tratar de adivinar cual es el mejor no es lo más óptimo. Como el algoritmo es bastante rápido para estos tamaños, vamos a probar todas las posibilidades y quedarnos con la mejor.

Para hacer esto, también queremos asegurarnos de que los resultados dependan lo menos posible de los datos de entrada y que sean generalizables, por lo que vamos a querer usar distintos conjuntos de datos. El problema es que solo tenemos un conjunto de datos, así que vamos a tener que particionarlo de distintas maneras para usarlo como si fueran varios. Este procedimiento se conoce como validación cruzada y en nuestro caso vamos a usar una versión conocida como *k-fold*. La notación puede volverse confusa (k de *k-fold* y k de KNN), por lo que vamos a usar números concretos a lo largo del informe, cosa que concuerda con nuestro trabajo ya que siempre usamos *4-fold* para experimentar.

4-fold consiste en partir los datos de entrenamiento en 4 partes del mismo tamaño, teniendo la misma distribución de los datos en cada partición. De esas particiones, 3 las vamos a usar como el nuevo conjunto de entrenamiento y la otra como el conjunto de desarrollo o validación. Hasta acá parecería que estamos en la misma situación, un solo conjunto de entrenamiento que además ahora es más chico, es hasta peor. Los beneficios de *4-fold* surgen cuando empezamos a cambiar las particiones a las que les toca ser entrenamiento y desarrollo como se ve en la figura 3.

A cada partición le va a tocar el rol de desarrollo una vez teniendo al resto como datos de entrenamiento. Así, a todos los datos de nuestro conjunto de entrenamiento original terminan siendo usados con ambos roles y en distintas situaciones, permitiéndonos obtener un resultado mucho más general que si solo hubiéramos usado un único conjunto que podría haber dado resultados muy sesgados. También conviene repartir los datos que le toca a cada partición de forma aleatoria (manteniendo las proporciones) para que el orden original tampoco afecte a los resultados.

Ahora si, queremos encontrar el mejor valor de k para nuestro algoritmo de clasificación. Como nuestro conjunto de entrenamiento completo es de 320 muestras, cada una de las 4 particiones va a tener 80 y en

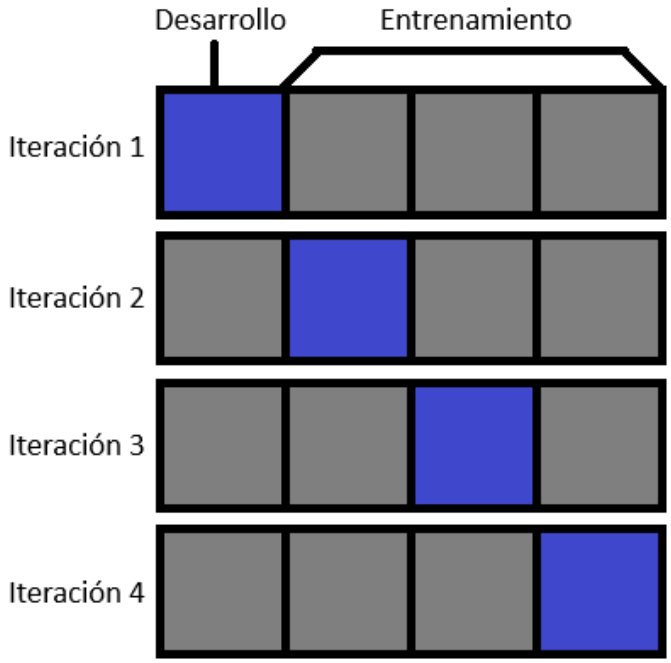

Figura 3: Validación cruzada 4-fold

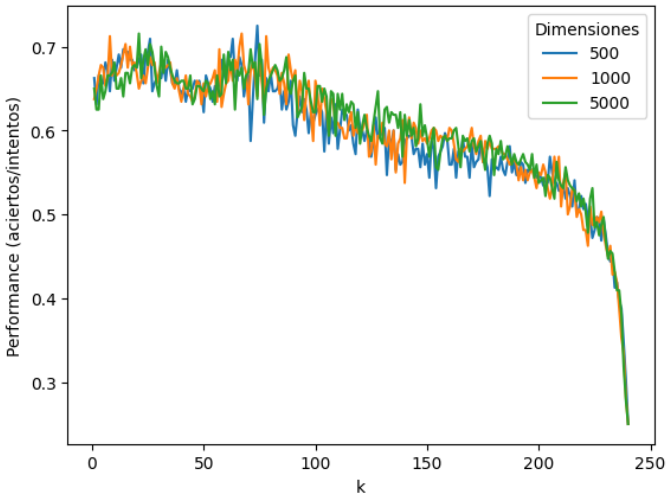

Figura 4: Exploración de mejor k para clasificador con KNN

total vamos a tener 240 muestras de entrenamiento y 80 de desarrollo por iteración. En cada iteración vamos a probar clasificar cada película de la partición de desarrollo usando las particiones de entrenamiento como los posibles vecinos con todos los valores posibles, de 1 a 240. Para cada valor, nos guardamos la cantidad de aciertos que después vamos a sumar con la del resto de iteraciones. Finalmente, podemos obtener el porcentaje de aciertos dividiendo por la cantidad total de pruebas y fijarnos que valor dio el mejor resultado. Experimentando con distintas cantidades de dimensiones, podemos visualizar los resultados de la exploración en la figura 4.

Si observamos los máximos para cada dimensión, obtenemos que los mejores valores son:

- $k = 74$ para 500 dimensiones, con un porcentaje de aciertos de 75.5 %
- $k = 67$ para 1000 dimensiones, con un porcentaje de aciertos de 71.56 %
- $k = 21$ para 5000 dimensiones, con un porcentaje de aciertos de 71.56 %

Estos resultados son bastante buenos, especialmente para un algoritmo tan sencillo. También conseguimos un poco de información (aunque bastante limitada) sobre la cantidad de dimensiones óptima. Si queremos armar un clasificador de películas exclusivamente con KNN, una buena opción sería usar un k de 74 y usar solo 500 dimensiones para comparar.

3.4. Búsqueda de mejores p y k

Ahora vamos a agregarle otra capa de complejidad a nuestro algoritmo. Vamos a usar PCA para tratar de reducir el tamaño de las matrices y analizar solo los componentes más importantes. Eso significa que ahora tenemos un nuevo hiperparámetro, la cantidad de componentes principales p, para el que también tenemos que tratar de encontrar un valor óptimo. Usamos el mismo método que antes, *4-fold*, pero esta vez probando combinaciones de p y k. El problema de hacer eso es que hay demasiadas combinaciones y los algoritmos no son tan rápidos, por lo que probarlas todas tardaría mucho tiempo.

Un primer retoque que podemos hacer para que podamos probar todos los valores de k sin agregar mucho tiempo es partir el algoritmo de KNN. Para cada vector del conjunto de desarrollo vamos a querer probar todos los valores posibles, pero la distancia entre ese vector y los puntos del modelo no cambian, solo cambia la cantidad de vecinos a tener en cuenta. Entonces, no tiene sentido volver a calcular la distancia hacia cada punto por cada valor distinto de k, podemos hacerlo solo una vez y después calcular la moda para todos los valores mucho más rápido. Partiendo el algoritmo así nos ahorramos muchísimo tiempo porque el cálculo de las distancias y el ordenamiento de las mismas es la parte más pesada del algoritmo, así que pasar de hacerla $P \times K \times J$ veces a $P \times J$ veces es un cambio enorme (donde P es la cantidad de valores a probar para p, K la de k y J la cantidad de vectores de desarrollo en un fold). Así es como nos queda KNN dividido en las operaciones CalcularDistancias y Clasificar, donde la primera se ocupa de calcular todas las distancias de un vector hasta cada punto y ordenarlas y la segunda calcula la moda de los primeros elementos para un valor pasado por parámetro.

Algorithm 8 CalcularDistancias

```
Entrada: vector, modelo
distanciasAVecinos ← listaVacía {Variable de clase}
for all vecino in modelo do
    distancia ← distanciaCoseno(vector, vecino)
    distanciasAVecinos.agregar(distancia)
end for
sort(distanciasAVecinos)
```

Algorithm 9 Clasificar

```
Entrada: k
return moda(distanciasAVecinos[0, k])
```

Para PCA este procedimiento ya estaba hecho así que el fit se hace solo una vez por fold y después para cada valor de p y para cada vector se hacen las transformaciones necesarias. Con todo esto, lo único que nos queda para reducir el tiempo de cómputo es acotar la cantidad de valores a probar. Como dijimos recién, vamos a probar todos los k ya que se puede hacer lo suficientemente rápido. Tenemos que probar todos los vectores del conjunto de entrenamiento para que nuestros resultados sean lo más generales posibles. Solo podemos jugar con los p.

Por lo que vimos antes, los autovalores de la matriz de covarianza son la varianza que agregan las direcciones generadas con sus autovectores asociados. Como además están ordenados de mayor a menor, eso significa que los autovectores van agregando cada vez menos varianza si los vemos en el mismo orden.

En la figura 5 podemos ver que la cantidad de varianza que nos da cada componente decrece drásticamente al principio y alrededor del componente #200 directamente no aportan casi nada. Esto significa que a partir de ese punto esos componentes no son casi útiles para tratar de distinguir las clases de nuestros vectores porque varían demasiado poco. También podemos ver en la figura 6 que a partir de ese punto ya tenemos casi toda la varianza acumulada, así que el resto de componentes realmente va a aportar muy poco al proceso. De hecho, a partir de 114 tenemos más del 95 % de la varianza acumulada, así que si estamos muy cortos de recursos podríamos recortar incluso más y no perder tanta precisión.

Con esto podemos dar una cota para p en 300, que nos alcanza para observar el rendimiento de casi todos los valores de varianza acumulada distintos evitando probar casos donde la diferencia debería ser despreciable. Además, si bien igual tarda bastante en computar todos los resultados, es menos de un tercio que si probáramos las mil posibilidades y sabemos que el resto de las que no probamos podrían ser como mucho marginalmente mejores.

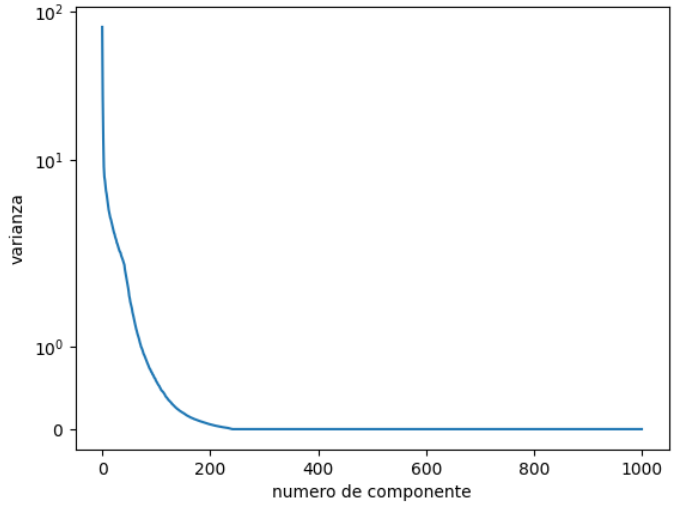

Figura 5: Varianza explicada por componente

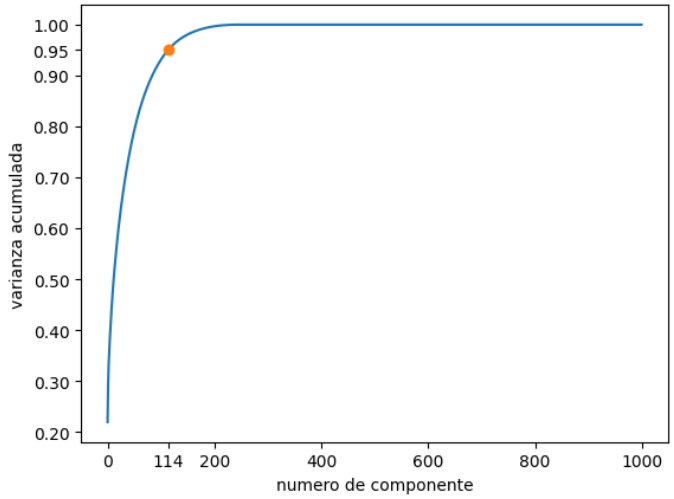

Figura 6: Porcentaje de varianza acumulada hasta cada componente

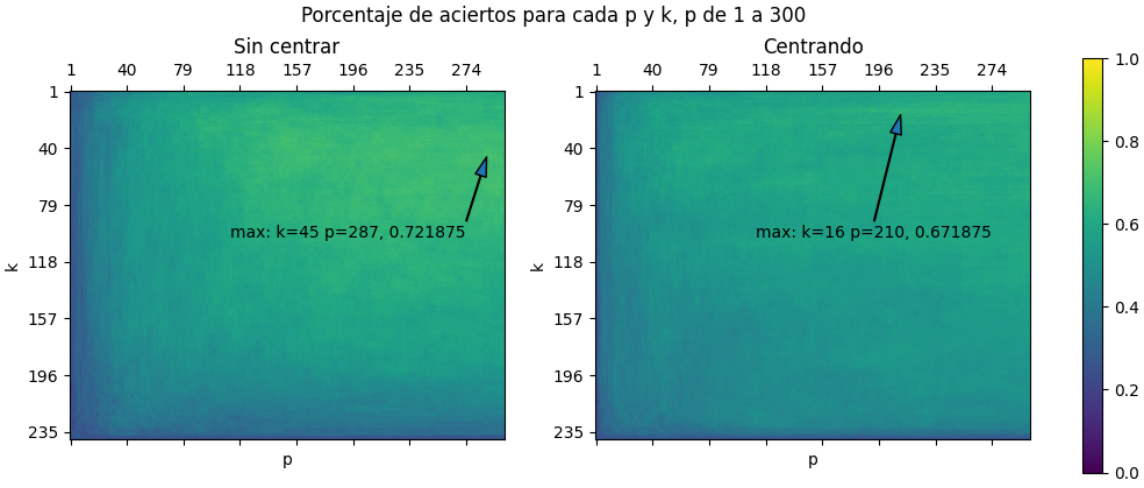

Figura 7: Rendimiento para valores de p y k con cota superior para p

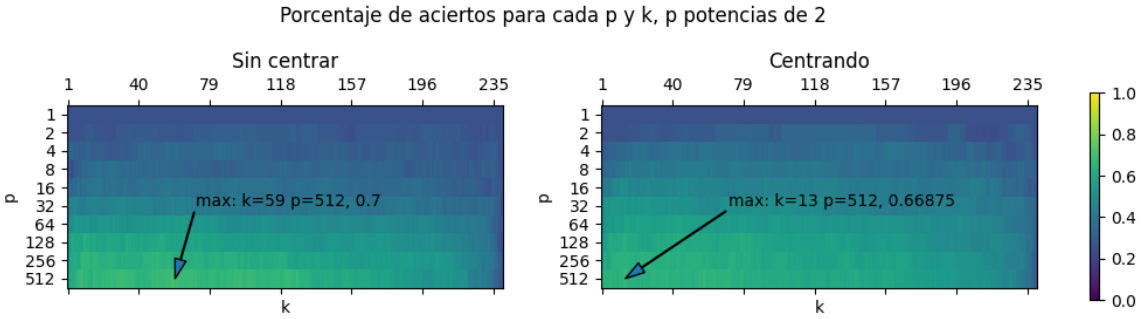

Figura 8: Rendimiento para valores de p y k probando potencias de 2 para p

Sin poder hacer mucho más, probamos todas esas combinaciones y vemos que resultados obtenemos. En la figura 7 podemos observar que hay una no tan sutil diferencia a la hora de centrar o no los datos. Por como es la distancia coseno, en un caso regular convendría centrarlos para que no queden restringidos a un solo cuadrante. Sin embargo, por la forma que tienen estos datos en particular, eso no aplica y en general obtenemos peores resultados cuando centramos. Más adelante vamos a comparar como de desempeña el clasificador con los datos de prueba reales, pero en principio parece que va a convenir no centrar los datos y usar un $k = 45$ y un $p = 287$, y esperaríamos tener un rendimiento de alrededor 70 %. Es interesante destacar como se refleja lo que dijimos antes de la varianza en nuestros gráficos. Se puede ver que a medida que aumentamos el valor de p, hasta aproximadamente 120 (y con un buen k), los resultados van mejorando de forma considerable. Sin embargo, a partir de ese punto se queda bastante estático con mejoras poco significativas.

Otra forma de hacer esta exploración es probando con ciertos valores para p estratégicos. Como ya vimos, el porcentaje de varianza que aporta cada cantidad de componentes crece de forma "logarítmica" (crece rápido al principio y después se aplana), por lo que tendría sentido ir probando valores de p de manera exponencial. Así, tratamos de balancear la cantidad de varianza que agregamos en cada iteración y obtenemos una gradiente similar la de probar todos los valores haciendo muchísimos menos cómputos.

En la figura 8 podemos ver los resultados de tomar potencias de 2 como valores de p y vemos que tenemos una situación muy similar al caso anterior, hasta 128 las mejoras son drásticas y luego se aplana. Llegamos a $k = 59$ y $p = 512$. Tiene sentido que el máximo esté en 512 ya que es donde más varianza se llegó a acumular, pero la diferencia con 256 es muy pequeña y podría valer la pena hacer ese recorte para cuestiones de optimización. Otra vez aparece el mismo problema cuando centramos así que queda bastante claro que con este conjunto de datos deberíamos evitar hacerlo.

Lo último que es interesante de observar es que los resultados para k coinciden con los que obtuvimos cuando la exploramos por separado. El valor de k óptimo no puede ser muy grande porque entonces se comparara con demasiados vecinos que ya dejan de estar realmente cerca y pasan a ser ruido.

3.5. Rendimiento del clasificador completo con mejores p y k

Llegamos al final, desarrollamos nuestros algoritmos, verificamos que fueran correctos y exploramos los hiperparámetros para encontrar los óptimos que mejoren nuestros resultados todo lo posible. Lo único que queda es probar nuestro clasificador completo con casos "reales" que no hayan formado parte del proceso en ningún momento y que no hayan podido sesgar las elecciones del clasificador.

Acá es donde entra ese último pedazo de nuestro conjunto de datos original, el que estaba clasificado como "test". No lo tocamos nunca a lo largo del proceso para poder usarlo en este paso. Vamos a probar todos estos vectores con nuestro clasificador para cada una de las configuraciones óptimas que encontramos y vamos a comparar cuál nos da los mejores resultados con datos completamente nuevos.

p	k	centrado	% de aciertos
287	45	no	0.7375
210	16	si	0.5875
512	59	no	0.7125
512	13	si	0.75

En la tabla podemos ver que en general los resultados fueron bastante buenos, manteniendo el nivel de calidad que nos brindaba KNN pero analizando muchísimas menos dimensiones y por lo consumiendo menos

recursos y de manera más rápida al momento de clasificar. Podemos observar otra vez que hay diferencia entre centrar los datos y no hacerlo, pero que para las distintas una vez fue mejor centrar y la otra no. Esto probablemente se relacione a la naturaleza de que los algoritmos tienen una buena cantidad de aleatoriedad y no siempre van a devolver los mismos resultados, siendo a veces peores o mejores. Lo mas importante que podemos sacar de los resultados es que, en efecto, usando una cantidad mucho menor de dimensiones (287 es casi la mitad que 512), obtenemos resultados muy similares, lo que concuerda con el análisis que hicimos previamente. Entonces logramos nuestro objetivo, hacer un clasificador de resúmenes de películas que funciona con KNN y puede ser lo suficientemente óptimo gracias a PCA sin perder mucha precisión.

4. Conclusiones

A lo largo de este trabajo construimos un clasificador de películas a partir de KNN y PCA, lo cual nos llevó a procesar texto en lenguaje humano de forma matricial, implementar esos de clasificación y reducción, y finalmente analizar los hiperparámetros necesarios para obtener los mejores resultados. Obtuvimos resultados bastante buenos con aproximadamente 80 % de aciertos sobre nuestro conjunto de pruebas. No es un clasificador infalible pero para su implementación simple y su complejidad tanto temporal como espacial está bastante bien.