MapReduce and Document clustering

27-11-2022

He Chen

Daniel Muñoz

Grupo 12

Implementación

A la hora de hacer la implementación, nos costó hacernos a la idea de que eran los prototipos y cómo se representaba. Una vez entendimos cómo se representaban, entendimos el funcionamiento general y ya nos fue más sencillo implementar las funciones faltantes de los códigos.

También cabe destacar, que el Kmeans, mirando por internet, normalmente se suele utilizar con distancias. En nuestro caso no es directamente así, pero es parecido. Nos piden utilizar una función de similitud de Jaccard. Nosotros lo hemos querido hacer con distancias mínimas, ya que nos era más fácil de pensar que el caso contrario, con lo cual, mientras más similitud haya, queremos que haya menos distancia y viceversa, así que, si hacemos el complementario de jaccard, podemos buscar las distancias mínimas, ya que por ejemplo si tenemos dos palabras muy similares, la similitud de jaccard será cercana a 1 y al hacer el complementario para tratarlo con distancias mínimas, la distancia será cercana a 0, lo cual quiere decir que está muy cerca. En el caso de que haya poca similitud, Jaccard nos dará valores cercanos a 0 y si lo miramos como distancia, haciendo el complementario, obtendremos valores cercanos a 1, cosa que quiere decir que están muy lejos.

Experimento 1: impacto de frecuencia

Primero de todo, creamos el índice a los archivos archiv_abs con IndexFiles.py. Una vez hecho esto, empezaremos con el primer experimento. Este consiste en mirar la influencia de la inicialización de las palabras en los documentos, mirando cómo convergen los 10 clusters iniciales a medida que avanzan las iteraciones. Para ello utilizaremos como mucho 200 palabras que estén en porcentajes de frecuencias bajas (1-10 y 20-30) altas (80-100) y medias (50 - 60)

Número de cluster después de cada iteración en función del rango de frecuencias					
Iteraciones / Rango de frecuencias (%)	[1 - 10]	[20 - 30]	[50 - 60]	[70 - 80]	[80 - 100]
0 (initial prototype)	10	10	10	10	10
1	10	10	7	1	1
2	10	10	Algorithm converged	Algorithm converged	Algorithm converged
3	10	10	Algorithm converged	Algorithm converged	Algorithm converged
4	10	10	Algorithm converged	Algorithm converged	Algorithm converged
5	10	10	Algorithm converged	Algorithm converged	Algorithm converged

Como se puede observar, con porcentajes de frecuencias muy altas ([80-100]), de 10 clusters que había inicialmente, en la primera iteración pasa a 1 y en las demás iteraciones no hace ninguna modificación. Esto puede deberse a que las palabras de mucha frecuencia serán comunes para todos los documentos y por lo tanto todos los documentos tendrán prácticamente las mismas palabras. Debido a esto, la similaridad de jaccard será 1 o un valor muy parecido, cosa que quiere decir que serán muy similares y entonces estarán muy cerca. Por lo tanto se clasificarán todos los documentos en el mismo cluster.
También se puede ver que con frecuencias bajas([1-10] o [20-30]), después de 5 iteraciones, sigue teniendo el número de clusters inicial. Esto puede ser debido a que, al tratar con documentos con palabras de frecuencias muy bajas, cada palabra aparecerá en muy pocos documentos. Al hacer la similaridad de jaccard, obtendremos un valor de 0 o muy cercano a 0, así que quiere decir que no serán similares y por lo tanto que están muy separados. Pero como hay más documentos que clusters y hay que hacer una agrupación, se agrupan en clusters de manera aleatoria en el prototipo inicial ya que ninguno de los documentos tienen palabras en común, y en las siguientes asignaciones seguiría siendo más o menos aleatorio.
Para las probabilidades de frecuencias de entre 50-60, se puede ver que acaba convergiendo con 7 clusters, esto es debido a que ya no son casos tan extremos como los anteriores, ya que hay documentos que comparten palabras pero no son prácticamente todas en todos los documentos, pero sí en algunos, ya que ha acabado teniendo 7 clusters en vez de 10.
También podemos ver que el número de palabras para cada documento varía mucho, dependiendo del rango de frecuencias que pongamos. Para frecuencias altas, las palabras que se obtendrán serán muy pocas, ya que lo que debería de pasar es que salgan las mismas para todos los documentos. En nuestro caso, solo sale 1 en el rango de porcentajes de frecuencia de 80-100. Para frecuencias muy bajas, es fácil que salgan muchas palabras, lo que en nuestro caso solo obtenemos 200, ya que las limitamos. Así que inicialmente, para frecuencias muy bajas, obtendremos muchas palabras, pero a medida que aumentamos los rangos de frecuencias, irán disminuyendo el número de palabras.

Experimento 2: análisis de clusters

En este experimento miramos cómo cambia el contenido de los clusters fijando un rango de frecuencias muy bajo.
Hemos partido de 20 clusters, un rango de frecuencia entre [0.01, 0.05] y un tamaño de vocabulario de 200. Con esta configuración aseguramos que las palabras del vocabulario sean de tema muy específicos.
Después de 20 iteraciones ejecutando el algoritmo, el número de clusters resultando no ha cambiado. Analizando los clusters vemos que hay palabras tipo [accept, adjust, admit] que parecen en muchos clusters ya que se pueden usar en muchas ocasiones y no son palabras específicas para un tópico.
Fijando en palabras más raras, encontramos asociaciones como:

- [game, github, parameter, overlap] aparecen juntos en la mayoría de los clusters, sobre todo la palabra ‘game’ y ‘github’.
- [norm, pipelin, pixel, orthogon] que pertenecen a la familia de graficos.
- [medic, conform, message] que pertenecen al trópico de salut

Aunque sí hemos encontrado alguna relación, hemos visto que los clusters resultantes están compuestos de temas muy variados, por eso, hicimos otra prueba incrementando el número de clusters a 100, para ver si podemos obtener clusters referentes a tópicos más específicos. Esta vez hay 99 cluster resultantes y sí que encontramos algunos cluster interesantes, aquí están los dos más cortos:

- CLASS0: [away, choos, complic, game, hierarch, horizon, miss, monoton, notabl, onto, outsid,reinforc, sign, throughout, trigger]. Referente al tópico de videojuegos.

- ['CLASS43: [accept', 'adjust', 'aid', 'annot', 'au', 'away', 'baryon', 'belong', 'choos', 'collabor', 'column', 'complic', 'contact', 'coverag', 'denot', 'ell', 'emit', 'epoch', 'explan', 'game', 'gtrsim', 'hierarch', 'horizon', 'hubbl', 'hydrogen', 'ioniz', 'jet', 'manual', 'miss', 'mm', 'monoton', 'net', 'norm', 'notabl', 'nu', 'onto', 'orthogon', 'outsid', 'overlap', 'parameter', 'pc', 'phi', 'put', 'rare', 'reinforc', 'rho', 'ring', 'setup', 'sign', 'stationari', 'store', 'strict', 'tau', 'throughout', 'trigger', 'until', 'white', 'wise']. Aquí las palabras se pueden dividir en tópicos como, matemática, química, informática… Lo podemos clasificar como ciencia.

Experimento 3: análisis del tiempo de ejecución

En este experimento miraremos la influencia que tiene ejecutar las iteraciones con diferentes cores y cómo esto afecta al tiempo de ejecución. Para ello, escogeremos un rango de frecuencias intermedio entre 0.01 y 0.1, un número de clusters iniciales de 10 y iremos ejecutando el Kmeans para diferentes cores. Los resultados obtenidos para diferente tamaño de vocabulario están en la siguientes tablas:

Tiempo de ejecución para los diferentes número de cores (100 palabras)				
Iteración/ número de cores	1 (cores)	2	4	8
1 (iteracion)	3.91 s	3.15 s	2.87 s	2.99 s
2	16.73 s	9.78 s	9.23 s	8.02 s
3	16.03 s	9.42 s	8.51 s	8.21 s
4	16.90 s	8.72 s	8.51 s	8.52 s
5	16.32 s	9.77 s	7.15 s	8.31 s

Tiempo de ejecución para los diferentes número de cores (250 palabras)				
Iteración/ número de cores	1 (cores)	2	4	8
1 (iteracion)	6.55s	5.12 s	4.09 s	3.96 s
2	37.99s	26.26 s	19.38 s	16.70 s
3	36.95s	21.86 s	15.03 s	15.30 s
4	35.46s	22.90 s	14.39 s	15.86 s
5	38.69s	23.05 s	16.41 s	13.99 s

Como podemos observar en las dos tablas, las primeras iteraciones son más rápidas que el resto. Esto es debido a que al crear el prototipo inicial, solo se usan las palabras de un documento. Una vez acabada la primera iteración, ya se computan las palabras de todos los documentos distribuidas en los diferentes clusters. Como los cálculos dependen de cuántas palabras haya, ya que, entre otras cosas, la similitud de Jaccard depende del número de palabras, será más costoso.

De la misma manera, como en la primera tabla hay 100 palabras por documento y en la segunda 250 palabras por documento y el código depende del número de palabras, habrá muchas más palabras al hacer los cálculos de la segunda tabla que en la primera, ya que por cada documento hay 150 palabras más en la segunda tabla que en la primera. Así que es normal que el tiempo de cómputo con 100 palabras por documento sea bastante menor que en el de 250 palabras por documento.

Finalmente, podemos ver que el tiempo a medida que aumentamos los números de cores linealmente, no aumenta el tiempo linealmente, sino que hay más del esperado. Por ejemplo, con 1 core para la iteración 2 con 250 palabras se obtiene un tiempo de 37.99s, con dos cores, el resultado esperado sería de 18.995 s, en cambio obtenemos un tiempo mayor (26.26 s), esto es un 144,67 % peor de lo esperado. Con 4 cores se esperaría un valor de 9,4975 s, pero volvemos a tener un valor mayor del esperado (19.38 s), lo que supone un 204,05% peor de lo esperado. Con 8 cores se esperaría un tiempo de 4,74875 s, pero de nuevo obtenemos un valor mucho peor que los anteriores (16.70 s) con un 351,67 % peor del resultado esperado. Lo mismo pasa con el resto de iteraciones y con la otra tabla. Así que mientras más cores, obtenemos un resultado cada vez peor del esperado y hay veces que llega a tardar incluso más tiempo con más cores, como podemos ver en la iteración 4 de ambas tablas, si miramos las ejecuciones con 4 y 8 cores. Así que es mejor ejecutar con 4 cores que con 8, ya que no se encuentra una mejora significativa y a veces es incluso tarda más con 8 que con 4. Esto podría ser por los overheads.