CAIM LAB6

Yimin Pan

November 2019

1 Introduction

En esta práctica intentamos simular el funcionamiento del `Hadoop Map-Reduce` usando la libreria `MRJob` de Python para implementar el algoritmo `K-Means Clustering` para documentos.
Como ya sabemos, en `K-Means` los datos se proyectan como puntos en el espacio, y asignamos estos puntos al cluster más cercano, dependiendo de la distancia al centroide del cluster. Cada documento se representará como un vector booleano de estos tokens, y los centroides como frecuencia de tokens. De este modo, podemos calcular la similitud o distancia usando `Jaccard`:

$$Jaccard(doc_1, doc_2) = \frac{doc_1 \cdot doc_2}{\|doc_1\|_2^2 \|doc_2\|_2^2 - doc_1 \cdot doc_2}$$

En cuanto a la implementación, no se ha encontrado gran dificultad, era simplemente entender el funcionamiento de MapReduce y K-Means. Sin embargo, a la hora de debugar fue un poco costoso, ya que la salida standard estaba ocupada por `MRJob`.
Al principio creía que la fórmula de `Jaccard` que se daba en el pdf calculaba la distancia entre documentos, ya que en `K-Means` se suele usar funciones de distancia. Pero al analizar un poco la fórmula, me dí cuenta que lo que calcula es la similitud. Es importante saber esta diferencia, ya que en un caso asignamos el documento al cluster con menor distancia, y en el otro al cluster con mayor similitud; por un lado buscamos el mínimo y por el otro el máximo.

2 Experimentación

La experimentación lo haremos sobre los documentos de arxiv como pide la práctica, y la dividimos en dos fases, análisis de los clusters resultantes y el tiempo de ejecición.

2.1 Análisis del número de clusters

Nos centraremos principalmente en los efectos que tiene al modificar el rango de frecuencias de los tokens, con las que representaremos los documentos. `Numwords`

lo fijamos a 200, ya que en realidad solo limita el número de tokens seleccionados por el rago de frecuencia, por lo que no debería afectar mucho a los clusters resultantes. En cuanto a iteraciones, pondremos 5 por defecto y es suficiente para observar la evolución de los clusters

Número de Clusters resultantes				
	[0.01, 0.05]	**[0.1, 0.3]**	**[0.3, 0.7]**	**[0.7, 0.9]**
it0	10	10	10	10
it1	10	9	8	1
it2	10	9	8	1
it3	10	9	8	-
it4	10	9	8	-
it5	10	9	8	-

Como podemos observar, cada vez que escogemos frecuencias mayores, tenemos menos clusters. Esto se debe a que cuando escogemos tokens frecuentes para representar los documentos, todos los documentos son muy parecidos (tiene los mismos tokens), por lo tanto la distancia entre ellos es muy pequeña. Y por el funcionamiento de K-Means, estos se asignarán al mismo cluster. Dicho esto, el centroide del cluster se calculará apartir de los documentos asignados, con lo que en la siguiente iteración tendremos el centroide muy cercano a los documentos asignados en la iteración anterior, y el método convergerá rápidamente. Por este motivo en el experimento [0.7, 0.9] solo hemos tardado 2 iteraciones en converger.
Hay que darse cuenta qeu cuando la frecuencia es baja, juega un pequeño factor aleatorio, debido a que los centroides de los clusters son escogidos aleatoriamente entre los documentos. Pero por lo generar debido a que la similitud de documentos no es tan alta, los documentos se suelen clasificarse en clusters diferentes, por lo que el número de cluster final es cercano al limite que le ponemos. (`nclust = 10`).

Para experimentar más con los documentos de arxiv, fijaremos el número de clusters inicial a un valor evelevado, 30 por ejemplo. Generamos los tokens con frecuencia [0.05, 0.2], que es lo suficientemente bajo para que los documentos sean diferentes.
El proceso ha tardado 13 iteraciones en convergerse, y acaba con 28 clusters. Esto nos dá un indicio de que aunque los documentos de `arxiv` se puede separar en temas diferentes, podemos clasificarlos en más subtemas aún, ya que se trata de textos muy cortos y hablan de cosas específicas. Esto también se vé reflejado con otro experimento, escogiendo un valor muy exagerado para el número de cluster inicial, por ejemplo 1200 (en total solo tenemos 1128 documentos en arxiv), después de unas cuantas iteraciones, el resultado son 730 clusters.

Si analizamos las palabras de algunos de los 28 clusters resultantes. Podemos

observar que hablan ya de temas muy específicas, debido a que [0.05, 0.2] son frecuencia tan bajas, para que no salgan palabras muy comunes.

- ClASS0: ['network', 'dynam', 'power', 'increas', 'analysi']
- CLASS10 :['converg', 'asymptot', 'regular', 'determin', 'type']
- CLASS2:['rate', 'achiev', 'character', 'channel', 'probabl']
- CLASS23: ['extend', 'algebra', 'recent', 'correspond', 'so']

La clase 0 habla de redes, la 10 de matemáticas...etc. Dicho esto, podemos observar que por lo general los textos de `arxiv` hablan de temas científicas.

2.2 Análisis del tiempo de ejecución

Ahora analizaremos como influye los parámetros `ncores` y `numwords` al tiempo de ejecución. Daremos un rango de frecuencia lo suficientemente amplio ([0.05, 0.5]) para que haya suficientes numero de tokens, y así hacer que `numwords` entre en juego, y sea quien recorte el número de tokens finales. De este modo será el factor que principalmente influye al tiempo de ejecución. Otro factor puede ser `nclust`, cuando más cluster tengamos, más tiempo tardaremos en asignar un documento a un cluster, dado que tenemos que iterar sobre ellos para decidir cual es el más cercano. Dicho esto, fijaremos `nclust` = 10 para los experimentos.

Tiempo de Ejecución(s) para nwords = 100				
ncores	**1**	**2**	**4**	**8**
it1	1.17	1.09	1.08	1.18
it2	1.37	1.17	1.22	1.2
it3	1.37	1.2	1.17	1.34
it4	1.36	1.29	1.33	1.28
it5	1.39	1.28	1.32	1.45

Sin contar la primear iteración, con 1 core, en media las iteraciones tardan 1.37s. Con 2 cores, tardamos algo menos, pero cuando intentamos usar más cores, el tiempo de ejecución en lugar de disminuir, aumenta.

Tiempo de Ejecución(s) para nwords = 250				
ncores	**1**	**2**	**4**	**8**
it1	1.27	1.18	1.2	1.18
it2	1.87	1.47	1.29	1.3
it3	1.88	1.48	1.28	1.32
it4	1.91	1.49	1.31	1.54
it5	1.87	1.49	1.3	1.34

Aquí pasa algo similar que en `nwords = 100`, solo que en este caso el número de cores óptimo es 4.

En resumen, con estos experimentos, observamos que no siempre usando más cores obtenemos mejor rendimiento, sinó también depende de más factores. Como la cantidad de trabajo total, que en este caso lo determina el número de tokens que usamos. Por lo que el speedup sube hasta llegar a una punta máxima, y despues decae.
Cuando tenemos varios núcleos, podemos reducir el trabajo total, dividiendolo a estos núcleos. Lo lógico es pensar que, cuando más núcleos, menos workload se le asignará a cada uno, por lo tanto menos tiempo de ejecución. Pero no hemos de olvidar que existe un coste de overhead para sincronizar estos procesos, por lo que cuando el workload total no es muy alto, no recompensa usar tantos núcleos. Por este motivo, en `nwrods = 100` el óptimo es 2, y en `nwrods = 20` es 4.
Por otra parte, vemos que la primera iteración siempre tarada menos que los otros, esto es debido a que incialmente el centroide del un cluster corresponde a un documento, por lo tanto las operaciones tardan menos. Después en iteraciones posteriores, el centroide se calculará apartir de los documentos asignados al cluster, consecuentemente la lista de pares (doc , freq) será más extensa, y esto añade tiempo de cómputo al calcular la similaridad.