**K-MEANS**

**­­**­Per l’implementazione del k-means svolgono in ruolo di primaria importanza due aspetti: la scelta dei centroidi iniziali e la definizione di distanza tra i nodi e i cluster.

Data la necessità di trovare (almeno) 4 cluster all’interno del grafo, lo step iniziale consiste nella scelta di 4 centroidi. Tali centroidi, in ogni esecuzione dell’algoritmo, vengono scelti in maniera randomica attraverso una procedura tale da garantire che essi non abbiano collegamenti tra loro. In particolare, nel grafo in esame, la proprietà di non vicinanza tra i centroidi è facilmente rispettabile grazie alla bassa densità della rete. Si noti che la scelta randomica porta ad avere risultati molto differenti in varie esecuzioni dello stesso algoritmo.  
Per quanto concerne la distanza, si è deciso di definire come cluster a distanza minima da un nodo il cluster contenente il maggior numero di vicini di tale nodo. Si è scelto di ricorrere a questa definizione, invece di assegnare il nodo al primo cluster contenente almeno un vicino, per evitare il più possibile di sovrappopolare i primi cluster a discapito degli ultimi (dove per primo si intende il primo cluster presente nella sequenza di “if” relativa alla classificazione del nodo).

Una volta definiti questi aspetti, si è passati all’implementazione dell’algoritmo, in cui è stato definito un quinto “cluster”, privo di centroide, popolato da tutti i nodi non inseribili negli altri gruppi. In particolare, nel caso di un grafo non connesso, esso viene popolato da tutti i nodi delle componenti che non contengono un centroide, mentre nel grafo in esame (il quale è connesso), risulta di fondamentale importanza per una corretta implementazione della versione parallela, trattata in maggiore dettaglio in seguito.

La versione Naive di questo algoritmo (funzione k\_means nel file kmeans.py) è risultata essere inaccettabile per scopi pratici dato il troppo lungo tempo di esecuzione. Infatti, l’algoritmo, che è stato testato sul server online di Google Colaboratory, ha classificato solo 5000 elementi su 22470 in ben 6 ore (l’esecuzione è stato successivamente interrotta manualmente), mostrando inoltre un sempre maggiore rallentamento nella classificazione degli elementi (all’inizio classifica 250 elementi in pochi minuti, dopo 1000 elementi impiega circa 15 minuti ogni 250 elementi e andando avanti i tempi risultano essere sempre maggiori). Inoltre, i risultati parziali risultano essere insoddisfacenti data la tendenza a sovrappopolare alcuni cluster e lasciare quasi vuoti altri, come viene qui mostrato.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
| Politician | 289 | 0 | 1135 | 0 |
| Company | 289 | 18 | 528 | 0 |
| Government | 217 | 0 | 1974 | 2 |
| TV-show | 80 | 0 | 468 | 0 |

Questo esperimento ha quindi messo in luce due grandi problematiche da risolvere: il tempo di esecuzione e lo sbilanciamento dei cluster.

Come prima cosa, risulta di maggiore importanza ridurre il più possibile il tempo di esecuzione in modo da velocizzare l’analisi del problema. È stata quindi implementata una versione parallela dell’algoritmo (funzione parallel\_k\_means in kmeans.py) e i relativi esperimenti sono stati effettuati con numero di job pari a 4.  
La prima problematica riscontrata in questa versione è stata l’impossibilità di assegnare tutti i nodi ai cluster con una sola esecuzione dell’algoritmo. Infatti, dividendo i nodi in 4 gruppi, c’è un’altissima probabilità di avere nodi non connessi con gli altri nodi del suo gruppo, causata soprattutto dalla presenza di 2658 nodi aventi un solo arco. Di conseguenza, come accennato in precedenza, è stato inserito un quinto cluster che contiene tutti i nodi non assegnabili all’interno del suo gruppo e, dato l’elevato numero di elementi presenti in esso e la non convenienza nell’ottenere un numero maggiore di quattro cluster per motivi di accuratezza, si è scelto di effettuare la chiamata parallela alla funzione di clusterizzazione in maniera iterativa, classificando in ogni ciclo i nodi rimanenti dalla precedente esecuzione.   
Nel caso in cui il numero di elementi non ancora assegnati risulta essere inferiore a 10\*j, con j numero di job, si preferisce non eseguire l’algoritmo in maniera parallela ma nella versione naive, dato che con un basso numero di elementi la parallelizzazione non comporta vantaggi.

La versione parallela è risultata essere molto più veloce rispetto alla naive, richiedendo comunque un tempo non accettabile, di circa 8 ore e 30 minuti, con l’esecuzione di 4 cicli di clusterizzazione, in cui dopo il primo sono rimasti non classificati ben 9118 elementi, dopo il secondo 1251 e dopo il terzo 5. Di seguito vengono riportati i risultati.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
| Politician | 3851 | 410 | 1108 | 399 |
| Company | 3994 | 317 | 1295 | 889 |
| Government | 4748 | 608 | 977 | 547 |
| TV-show | 1995 | 146 | 744 | 442 |

Si può subito notare come l’accuratezza risulta essere bassissima, non riuscendo ad individuare in maniera corretta i cluster. Inoltre, pur sembrando essere meno affetto dal problema dello sbilanciamento delle dimensioni dei cluster rispetto al naive, tale problema si ripresenta allo stesso e identico modo, in quanto si è potuto notare che, nel primo ciclo, ogni job tende a popolare quasi solo un cluster, ma con buona probabilità che essi popolino cluster diversi. Ciò, quindi, causa l’assenza di categorie dominanti all’interno dei cluster.  
Prima di cercare di risolvere questa problematica, si è comunque deciso di lavorare per diminuire ulteriormente i tempi di esecuzione.

Da un’analisi dell’algoritmo, si è ipotizzato che il principale collo di bottiglia è dato da questa istruzione:

Immagine che contiene testo

Descrizione generata automaticamente

Tale istruzione consente di ricavarsi una lista contenente i nodi non ancora aggiunti in un cluster che hanno almeno un vicino in essi e viene eseguita ogni qual volta si deve scegliere un nodo da classificare e quindi n volte, con n numero di nodi. Inoltre, la complessità di questa istruzione risulta essere

*O(n\*(somma delle lunghezze dei cluster)\*4\*min(len(G.neighbors(el)),len(clusteri)),*

dove le operazioni più costose risultano essere il ciclo for e l’unione tra set e di conseguenza, all’aumentare delle dimensioni dei cluster, l’algoritmo diventa sempre più lento, come già notato in precedenza durante l’esecuzione dell’algoritmo naive.

Per eliminare questo collo di bottiglia si è deciso di rimuovere dalla variabile *samples* i nodi nel momento in cui vengono assegnati ad un cluster per ridurre in maniera progressiva la sua dimensione (la remove su un set ha complessità O(1)) e di introdurre una nuovo set: *neighbors\_clusters.* Questo set contiene tutti i nodi già classificati e i loro vicini. Quindi, ogni qual volta un nodo viene aggiunto in un cluster, si inserisce tale nodo nel set ed si effettua l’unione tra il vicinato del nodo e il set (complessità *O(len(neighbors\_cluster)+len(G.neighbors(nodo))*, che verso il termine dell’algoritmo tende ad O(n)).

L’operazione riportata in precedenza è stata invece sostituita da:



avente complessità pari a *O((min(len(samples), len(neighbors\_clusters))^2)*, ma dato che all’aumentare della dimensione di neighbors\_cluster si ha una riduzione della dimensione di samples, tale operazione non risulta essere onerosa. In particolare, si può notare come risulta completamente equivalente alla precedente istruzione, restituendo lo stesso e identico risultato.  
Infatti, le nuove versioni dell’algoritmo (optimized\_k\_means e parallel\_opt\_k\_means, si noti che sono state chiamate optimized per convenzione, seppur non introduco effettive modifiche nella logica dell’algoritmo) risultano essere estremamente più rapide, con risultati equivalenti in termini di accuratezza e sbilanciamento nelle dimensioni dei cluster (non possono essere esattamente uguali i risultati per la scelta randomica dei centroidi). In particolare, la versione normale impiega 1 minuto e 30 secondi circa per essere eseguita, mentre la versione parallela circa 5 minuti. Si noti che la versione parallela risulta molto più lenta in quanto deve essere eseguita più volte per assegnare tutti gli elementi ad un cluster e in ogni ri-esecuzione si deve ricreare il set *neighbors\_clusters* a partire dai cluster in input, operazione che risulta essere sempre più espansiva ogni qual volta si effettua un ulteriore chiamata a funzione. I risultati di queste versioni non sono stati aggiunti alla documentazione perché equivalenti ai precedenti e non introducono possibili nuove considerazioni.

Essendo stato raggiunto un tempo di esecuzione così breve, è ora possibile cercare di risolvere il problema dello sbilanciamento delle dimensioni tra cluster e migliorare l’accuratezza dell’algoritmo attraverso operazioni che, pur rallentando l’algoritmo, permettono di raggiungere un miglior trade-off tra tempo di esecuzione e accuratezza.

Si può notare che una scelta randomica dei centroidi può causare situazioni particolari in cui il grado di un centroide risulta molto maggiore rispetto agli altri. Ad esempio, si può avere la situazione in cui il centroide del cluster *x* ha grado 15, mentre gli altri hanno grado 1. Ciò comporta una probabilità di assegnare un elemento al cluster x pari a 15/18, la quale diventa sempre maggiore nel momento in cui si aggiungono altri nodi a questo cluster. Considerando inoltre che sono presenti 2658 nodi con grado 1 e allo stesso tempo vari nodi con grado > 200, tale problematica non può essere ignorata.

Per ovviare a questo problema, si è deciso di prendere come centroidi i nodi di grado più alto che non presentino collegamenti tra loro. Inoltre, per cercare di non sbilanciare troppo le probabilità di inserimento degli elementi in ogni cluster già all’inizio all’algoritmo si è deciso di inserire gli elementi presenti in *items\_to\_be\_clustered* in una priority queue, la cui priorità degli elementi è pari al loro grado, dando quindi maggiore priorità agli elementi con grado minore. Si può notare che, utilizzando una priority queue per l’inserimento degli elementi nei cluster, non si ha più bisogno di effettuare il cast a lista del set nell’operazione inserita nella precedente figura, permettendo quindi di ottenere circa la stessa velocità del precedente algoritmo, con un tempo di esecuzione di circa 1 minuto e mezzo. Inoltre, con questa implementazione, si è rimossa tutta la componente randomica dell’algoritmo, ottenendo quindi lo stesso risultato in ogni esecuzione, risultato qui riportato.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
| Politician | 96 | 1790 | 3800 | 82 |
| Company | 103 | 574 | 5811 | 7 |
| Government | 960 | 1015 | 4512 | 393 |
| TV-show | 304 | 416 | 2602 | 5 |

Possiamo subito vedere come lo sbilanciamento delle dimensioni sia ancora presente ma in maniera più ridotta rispetto alle precedenti versioni non parallele. Per quanto riguarda la precisione, essa risulta ancora essere molto bassa nonostante ci siano stati dei leggeri miglioramenti, clusterizzando correttamente 8566 elementi su 22470. In particolare, possiamo vedere come il cluster 3 contiene per ogni categoria il numero più alto di elementi, nonostante il centroide iniziale di tale cluster risulta essere quello col grado minore. Infatti, il centroide del cluster1 ha grado 709, quello del cluster2 417, quello del terzo 380 ed infine quello del quarto 468.

Per quanto concerne l’implementazione parallela di questa versione, anche in questo caso i tempi di esecuzione sono rimasti all’incirca inalterati, impiegando circa 5 minuti. Qui di seguito sono riportati i risultati.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
| Politician | 933 | 1654 | 2972 | 209 |
| Company | 2186 | 159 | 4098 | 52 |
| Government | 1776 | 1030 | 3467 | 607 |
| TV-show | 1043 | 181 | 2088 | 15 |

Anche in questo caso possiamo vedere come l’assegnazione dei cluster reali ai cluster ottenuti rimane equivalente, così come la tendenza del cluster 3 a contenere il maggior numero di elementi per ogni categoria. Da una prima analisi dei risultati, inoltre si può vedere come i TV-show risultano più difficilmente clusterizzabili attraverso il k-means rispetto alle altre categorie. Inoltre, le prestazioni risultano essere inferiori a quelli della versione non parallela, assegnando correttamente ai cluster 7543 elementi su 22470.