PC-2023/24 Final Project

Melina Berrio Pari
melina.berrio@edu.unifi.it

Abstract

Il progetto si concentra sull'implementazione e valutazione dell'algoritmo di clustering K-means in modalità parallela. Si tratta di un algoritmo di clustering partizionato, suddivide un insieme di punti in un numero specificato di cluster, "k", basato sulla distanza dai centroidi.

L'implementazione è realizzata in Python sfruttando la libreria "multiprocessing" questo approccio consente di distribuire il carico computazionale su più core del processore, accelerando il calcolo dell'assegnazione dei punti ai cluster.

La valutazione delle prestazioni comprende un confronto tra la versione parallela e quella sequenziale dell'algoritmo a livello dei test condotti su diversi numeri di core.

1. L'algoritmo k-means

K-Means esegue una assegnazione iterativa dei punti al "*centroide*" più vicino, il centroide rappresenta la posizione media di un cluster.

Pseudocodice:
Input: numero di centroidi *k*

1. Inizializzazione dei centroidi
2. Iterazione
   a. Assegnazione di ogni punto al centroide più vicino
   b. Calcolo del nuovo centroide per cluster

   Convergenza: il processo termina quando i centroidi non cambiano più.

Output: k cluster/sottoinsiemi

In questo analisi i centroidi verranno scelti casualmente, e i punti saranno in due dimensioni.

1.1.1 L'assegnamento di un punto ad un cluster

Si calcolano le distanze di ogni punto rispetto ad ogni centroide. Successivamente si assegna ogni punto al Cluster il cui centroide è a una distanza minima.

1.1.2 L'aggiornamento dei centroidi

Si ricalcola la posizione di ogni centroidi facendo la media di tutti gli elementi appartenenti al rispettivo cluster.

2. Implementazione Sequenziale

Si è creata una classe K-means di cui gli attributi sono:

- *n_clusters*: Il parametro k rappresentando il numero di clusters o centroidi.
- *clusters*: Un dizionario per salvare le coordinate dei punti una volta assegnati ai rispettivi clusters.
- *centroids*: Un dizionario dove se aggiornerà e salverà i centroidi finali.

Entrambi dizionari vanno inizializzati vuoti, per quanto riguarda la struttura si useranno oggetti di tipo array usando la libreria numpy.

Come funzioni di base abbiamo la funzione per calcolare la **distanza euclidea** tra due punti in uno spazio bidimensionale:

$$d(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Dove $P = (p_x, p_y)$ e $Q = (q_x, q_y)$

La cui equivalente usando la libreria numpy sarebbe:

```
np.sqrt(((P - Q) ** 2).sum())
```

Addizionalmente si è considerato **np.linalg.norm,** Questa funzione, sebbene comunemente utilizzata per operazioni vettoriali su array di grandi dimensioni, è altrettanto efficace per calcoli ripetuti tra singole coppie di punti, grazie al supporto di librerie C sottostanti.

```
np.linalg.norm(P - Q)
```

La seconda funzione di base, **nearest_centroid** calcola la distanza Euclidea tra un punto specifico "*x*" e un centroide "*value*" questo per tutti i centroidi affinché si possa trovare la distanza più piccola segnalando cosi il centroide più vicino.

```
np.argmin([self.euclidean_distance(point,
centroid) for centroid in
self.centroids.values()])
```

2.1. Inizializzazione dei centroidi

Si usa una seed=15 e poi si sceglie casualmente n posizioni del dataset iniziale successivamente si trovano i punti nelle posizioni suddetti, che saranno i centroidi iniziali. Costo computazionale O(n).

```
idx = np.random.default_rng(seed=
15).choice(len(X), self.n_clusters,
replace=False)

self.centroids = {kk: X[idx[kk]] for kk in
range(self.n_clusters)}
```

2.2. Parte Iterativa

Mentre non si verifichi convergenza, cioè quando centroidi non variano più, la esecuzione del ciclo dovrà continuare, il ciclo è composto per le seguenti funzioni:

recomputes_clusters che ottiene il centroide più vicino per ogni punto con la funzione nearest_centroid assegnando cosi il punto al Cluster. Costo computazionale O(kn).

```
self.clusters = {i: [] for i in
range(self.n_clusters)}

for point in X:
  self.clusters[self.nearest_centroid(point)
  ].append(point)
```

recomputes_centroids che calcola la media dei punti per cluster. Costo computazionale O(n).

```
for i in range(self.n_clusters):
    self.centroids[i] =
np.mean(self.clusters[i], axis=0)
```

Per garantire l'efficienza computazionale durante la fase di convergenza, l'uso di una soglia di tolleranza permette di limitare il numero di calcoli necessari. Nella funzione **has_converged**, abbiamo impostato una tolleranza di 0,0001. Questo significa che il processo di iterazione si interrompe non appena le differenze tra i centroidi della iterazione corrente e quelli della precedente scendono al di sotto di questa soglia. Di conseguenza, non è necessario calcolare continuamente le distanze tra ogni coppia di centroidi con elevata precisione, riducendo così il costo computazionale e accelerando il processo di convergenza.

```
if self.euclidean_distance(
self.centroids[i], old_centroids[i]) >
tolerance:return False
```

3. Implementazione Parallela

L'assegnazione dei punti ai cluster può essere molto dispendiosa in termini di tempo. Utilizzando **Pool**, ho suddiviso il carico di lavoro tra più core della CPU, permettendo l'elaborazione simultanea di diverse porzioni del dataset, Questo approccio è fondamentale per migliorare l'efficienza dell'algoritmo di

clustering, ma comporta la necessità di gestire attentamente la variabile self.clusters per evitare problematiche di race condition.

self.*clusters* è un dizionario che mappa ciascun cluster (identificato da un indice) a una lista di punti appartenenti a quel cluster. Durante la fase di assegnazione dei punti ai cluster, ogni processo calcola le assegnazioni localmente per un sottoinsieme di dati, senza modificare direttamente **self.***clusters*. Questo isolamento dei calcoli riduce il rischio di conflitti di accesso concorrenti. I risultati di queste assegnazioni, ottenuti tramite la funzione **assign_chunk**, sono successivamente aggregati e combinati in modo sicuro nella struttura principale.

Per implementare il parallelismo, ho suddiviso i dati utilizzando:

```
np.array_split(self.data, self.n_cores)
```

Dove self.n_cores rappresenta il numero di core del processore da utilizzare. Succesivamente è stato creato un pool di processi:

```
with Pool(self.n_cores) as pool:
    results = pool.map(self.assign_chunk,
chunks)
```

La funzione **assign_chunk** viene applicata a ciascun blocco di dati in parallelo, per determinare il centroide più vicino a ogni punto appartenente al questa partizione.

```
def assign_chunk(self,chunk):
    return [self.nearest_centroid(point) for
point in chunk]
```

I risultati dell'elaborazione parallela sono poi raccolti e aggregati sequenzialmente. Questa aggregazione avviene senza sovrapposizione di accesso ai dati condivisi tra i diversi processi, poiché i dati sono aggiornati in modo ordinato e sequenziale. Ogni punto viene assegnato al cluster più vicino e memorizzato nella lista corrispondente, migliorando notevolmente l'efficienza dell'algoritmo e riducendo i tempi di elaborazione complessivi.

Si è deciso di non applicare il parallelismo alla fase di ricalcolo dei centroidi. Questa scelta è stata motivata dal numero ridotto di centroidi (3), che rende il ricalcolo computazionalmente poco oneroso. Inoltre, il parallelismo avrebbe introdotto un overhead non giustificato dai benefici in questo contesto specifico. Pertanto, la fase di ricalcolo è stata mantenuta sequenziale.

4. Applicazione

Il Dataset usato proviene da UCI machine learning repository, e contiene 581,012 records con caratteristiche cartografiche. Dopo una breve preparazione di dati che ha incluso la selezione di solo 2 variabile, rimozione di duplicati, la standardizzazione e la riformulazione in 3 classi, il dataset è stato ridotto a con 562,873 records. Per l'analisi, sono stati considerati campioni del dataset a tre livelli: 25% (140,718 records), 50% (281,436 records) e 100%.

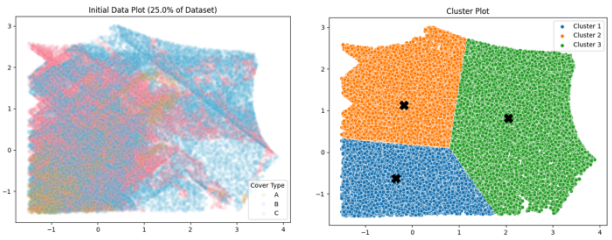

Figura 1. Dataset iniziale al 25 % e la sua Classificazione

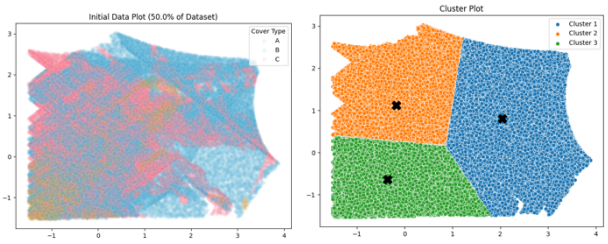

Figura 2. Dataset iniziale al 50 % e la sua Classificazione

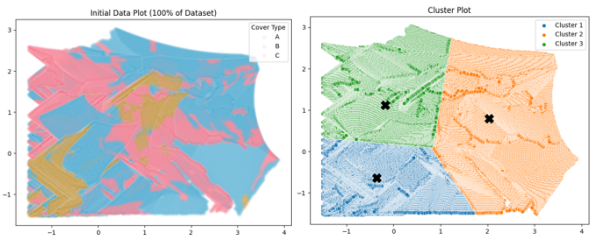

Figura 3. Dataset iniziale al 100 % e la sua Classificazione

I risultati dei tempi di esecuzione sono stati ottenuti utilizzando la libreria **time** per misurare il tempo di esecuzione sia in modalità sequenziale che parallela. I tempi sono stati registrati per diverse dimensioni del dataset e configurazioni di core, evidenziando come l'uso del parallelismo influisca sulle performance.

Tabella 1. Tempo di esecuzione (secondi) in modo sequenziale

25%	50%	100%
31.146655	59.806718	68.944432

Con dataset di dimensioni maggiori, i tempi di esecuzione sequenziali mostrano un incremento notevole, confermando l'elevato costo computazionale associato alla gestione di grandi volumi di dati.

Figura 4 . Tempi di esecuzione (secondi) in modo parallelo

Per ottenere un quadro più preciso delle prestazioni, eseguiremo un speed test.

$$SpeedUp = \frac{T_s}{T_p}$$

Tabella 2. Tempo di esecuzione (secondi) e Speedup per il 25% del dataset

Mode | Dataset	25%	Speedup
1- sequential	31.146655	
2 - cores	33.914527	0.92
4 - cores	28.989473	1.07
6 - cores	30.412248	1.02
8 - cores	36.54252	0.85
12 - cores	42.556131	0.73

Tabella 3. Tempo di esecuzione (secondi) e Speedup per il 50% del dataset

Mode | Dataset	50%	Speedup
1- sequential	59.806718	
2 - cores	50.769314	1.18
4 - cores	38.13253	1.57
6 - cores	36.694327	**1.63**
8 - cores	39.541635	1.51
12 - cores	47.729609	1.25

Tabella 4. Tempo di esecuzione (secondi) e Speedup per il 100% del dataset

Mode | Dataset	100%	Speedup
1- sequential	68.944432	
2 - cores	41.143057	1.68
4 - cores	27.890223	2.47
6 - cores	24.720345	**2.79**
8 - cores	25.153693	2.74
12 - cores	28.516521	2.42

L'analisi dei tempi di esecuzione e del speedup per diverse dimensioni del dataset evidenzia variazioni nell'efficacia della parallelizzazione. Per il 25% del dataset (Tabella 2), l'uso di più core ha avuto performance inferiori rispetto all'esecuzione sequenziale, con speedup

generalmente sotto 1 e un massimo di 1.07, quindi no speedup. La performance diminuisce con l'aumento dei core, probabilmente a causa di l'overhead della gestione dei processi.

Per dataset più grandi (50% e 100%, Tabella 3 e Tabella 4), la parallelizzazione mostra miglioramenti più significativi. Per il 50% del dataset, 6 core offrono il miglior speedup di 1.63, mentre per il 100%, il speedup raggiunge fino a 2.79 con 6 core, risultando migliore rispetto ai 12 core.

Questo indica che, mentre la parallelizzazione è molto efficace per dataset più grandi, il numero ottimale di core può variare a seconda della dimensione del dataset e del relativo overhead di gestione.