# Algoritmo di clustering k-means in stile MapReduce e in Go

Progetto B1

***Melissa Petrolo***
Corso di Laurea Magistrale in Ingegneria Informatica
*Universita degli Studi di Roma "Tor Vergata"*
*0286160*
emelis.ptr@gmail.com

***Abstract*—Questo documento riporta l'implementazione dell'algoritmo k-means secondo la computazione MapReduce con i relativi benchmark dell'applicazione.**

***Keywords — K-means, MapReduce, llyod, k-means++, docker, container, ec2, benchmark***

## I. Introduzione

Lo scopo del progetto e realizzare nel linguaggio di programmazione Go un'applicazione distribuita che implementi l'algoritmo di clustering k-means in versione distribuita secondo il paradigma di computazione MapReduce.

L'applicazione distribuita soddisfa i requisiti elencati di seguito.

- Realizza un'architettura master-worker, in cui il master distribuisce il carico di lavoro tra i nodi worker, che implementano i mapper ed i reducer;
- Implementa k-means in modo distribuito secondo il paradigma di computazione MapReduce;
- Per il deployment dell'applicazione si utilizzano container Docker;
- Deployment dell'applicazione su una istanza EC2.

## II. Architettura

Per l'esecuzione dell'applicazione si utilizza una piattaforma software, Docker, che permette di creare, testare e distribuire applicazioni con la massima rapidità.

All'interno del file docker-compose vengono specificati i servizi utili per la creazione dei container:

- Master: distribuisce il carico di lavoro ai worker;
- Worker: esegue la computazione dell'algoritmo k-means;
- Registry: permette di registrare l'indirizzo IP e porta dei worker, in modo tale che il master possa accedervi e recuperare le informazioni;
- Benchmark: esegue benchmark in base al numero di punti, al numero di worker e reducer.

## III. Implementazione

### A. MapReduce

Map/Reduce supporta due modalità operative, sequenziali e distribuite. Nel primo caso, le funzioni di map e reduce vengono eseguite una alla volta. Quando tutte le attività della funzione map sono terminate, viene eseguita la prima funzione di reduce. La modalità distribuita esegue molti thread di lavoro che prima eseguono attività di mapping in parallelo e quindi riducono le attività.

L'algoritmo k-means è un algoritmo di clustering che permette di suddividere un insieme di punti in k gruppi o cluster sulla base della distanza euclidea. In questo caso, l'algoritmo è implementato secondo il paradigma di computazione MapReduce.

L'implementazione si suddivide in due fasi:

- Mapper
- Reducer

#### 1) Mapper

Nella fase di Map, ciascun mapper riceve in input un sottoinsieme di punti e per ogni punto determina la distanza euclidea con ognuno dei centroidi, determinando la distanza minima e assegnandolo al cluster più vicino.

La distanza euclidea tra due osservazioni rappresenta la radice quadrata della somma dei quadrati dei rispettivi valori:

$$d = (i,j) = \sqrt{\sum_{k=1}^{n} \left(X_{ik} - X_{jk}\right)^2}$$

$$dove\ i = (X_{i1}, X_{i2}, \dots, X_{in}) e\ j = \left(X_{j1}, X_{j2}, \dots, X_{jn}\right)\ sono\ due\ vettori\ n - dimensionali.$$

#### 2) Reducer

Nella fase di Reduce, ciascun reducer determina per ogni cluster il nuovo centroide calcolando la media dei punti che appartengono a quel cluster.

### B. Algoritmi

Gli algoritmi di clustering sviluppati sono:

- Llyod

- Standard K-means
- Kmeans++

*1)* ***Llyod***

1. *Inizializzare k;*
2. *Suddividere i punti dell'insieme X nei k clusters;*
3. *Scegliere k punti uniformemente dall'insieme dei k clusters, determinando i centroidi di ogni cluster;*
4. *Applicare la fase di Map e Reduce;*
5. *Determinare i nuovi centroidi dal risultato della fase di Reduce;*
6. *Ripetere step 4 fino allo step 5 fino alla convergenza;*

*Tabella 1 - Llyod*

*2)* ***Standard K-means***

1. *Inizializzare k;*
2. *Scegliere k punti uniformemente dall'insieme dei punti X determinando i centroidi di ogni cluster;*
3. *Applicare la fase di Map e Reduce;*
4. *Determinare i nuovi centroidi dal risultato della fase di Reduce;*
5. *Ripetere step 3 fino allo step 4 fino alla convergenza;*

*Tabella 2 – Standard K-means*

*3)* ***K-means++***

1. *Inizializzare k;*
2. *Scegliere un punto M scelto uniformemente dall'insieme dei punti X;*
3. *Applicare la fase di Map e Reduce per determinare un nuovo centroide;*
4. *Aggiungere il punto x all'insieme M;*
5. *Ripetere step 3 fino a 4 per un totale di k-1 centroidi;*
6. *Applicare la fase di Map e Reduce dell'algoritmo Llyod con questi centroidi.*

*Tabella 3 – K-means++*

### *C. Master-worker*

È stata realizzata un'architettura master-worker.

*1) Master*

Il master attraverso la funzione *DialHTTP* si connette a un server rpc "Registry" all'indirizzo di rete specificato in ascolto sul percorso RPC HTTP predefinito.

Tramite la funzione *Call* richiama la funzione denominata, ne attende il completamento recuperando l'indirizzo IP e porta di ogni worker che è stato registrato.

Recuperate le informazioni dei worker, il master si connette tramite rpc all'indirizzo di rete di ogni worker e assegna ad ognuno di loro una porzione di punti e/o cluster per eseguire la funzione di *Call* sul mapper e reducer.

*2) Worker*

Il worker tramite la funzione *Register* pubblica i metodi del destinatario in DefaultServer, in modo tale da eseguire la funzione di map e reduce.

Anche il worker si connette tramite la funzione *DialHTTP* all'indirizzo del registry ed effettua la funzione di *Call* per registrare la sua porta e il suo indirizzo IP.

Tramite la funzione *HandleHTTP* registra un gestore HTTP per i messaggi RPC in DefaultServer in DefaultRPCPath e un gestore di debug in DefaultDebugPath. È ancora necessario richiamare la funzione *http.Serve()*, per accettare connessioni http sul *listener*.

Il worker, durante la funzione *Call* del master esegue i metodi di map e reducer e ritorna, attraverso un canale, i risultati ottenuti al master.

*3) Registry*

Il registry memorizza solamente le informazioni del worker, affinché il master possa recuperarle. Il registry aspetta finché tutti i worker sono stati registrati. Tramite la funzione *Register* pubblica i metodi su una porta aperta.

## IV. TECNOLOGIE E LIBRERIE

### *A. Tecnologie*

Le piattaforme utilizzate per la progettazione dell'applicazione sono:

- Go: linguaggio di programmazione usato per l'implementazione dell'applicazione;
- Docker: supporto alla virtualizzazione;
- Docker-compose: orchestrazione dei container, per creare una rete che permette di far comunicare i singoli container;
- Ec2: sviluppo dell'applicazione su un'istanza della piattaforma aws.
- go-RPC: per la comunicazione dei diversi container.

### *B. Librerie*

Le librerie importate e utilizzate nell'applicazione, non considerando le librerie predefinite di Go, sono elencate all'interno della seguente tabella.

| | |
|---|---|
| Net/rpc | Il pacchetto rpc consente di accedere ai metodi esportati di un oggetto attraverso una rete o un'altra connessione di I/O. Un server registra un oggetto, rendendolo visibile come servizio con il nome del tipo dell'oggetto. Dopo la registrazione, i metodi esportati dell'oggetto saranno accessibili da remoto. Un server può registrare più oggetti (servizi) di tipi diversi, ma è un errore registrare più oggetti dello stesso tipo. |
| testing | Il test dei pacchetti fornisce supporto per il test automatizzato dei pacchetti Go. È destinato ad essere utilizzato di concerto con il comando "go test", che automatizza l'esecuzione di qualsiasi funzione del modulo |
| errorgroup | Il gruppo errgroup del pacchetto fornisce la |

| | |
|---|---|
| | sincronizzazione, la propagazione degli errori e l'annullamento del contesto per gruppi di routine che lavorano su sotto attività di un'attività comune. |
| godotenv | Utilizzato per caricare in env o come comando bin. |
| plot | Package plot fornisce un'API per l'impostazione di grafici e primitive per disegnare su grafici. |

*Tabella 4 - Librerie*

## V. Analisi delle prestazioni

Le valutazioni sulle prestazioni sono state effettuate su un processore Intel® Core™ i7-6500U CPU @ 2.50GHz 2.60GHz.

Benchmark permette di valutare le prestazioni dell'applicazione al variare del numero di punti, di mapper e reducer. Si utilizza il package di testing per implementare la funzione di Benchmark.

Le metriche prese in considerazione sono:

- Il numero totale di allocazioni di memoria;
- Latenza;
- Il numero totale di bytes allocati;
- Numero di iterazioni dell'applicazione fino alla sua convergenza.

Le valutazioni sono state eseguite tenendo conto di un numero di worker pari a 3 e un numero di cluster pari a 5. Nella tabella seguente vengono riportati i risultati delle diverse prestazioni.

| **Llyod** | | | | | | | |
|---|---|---|---|---|---|---|---|
| Latenza | | Memoria allocata | | Bytes allocati | | Numero iterazioni | |
| Cluster | 5 | Punti | 200 | Mapper | 2 | Reducer | 2 |
| 1m50.5651021s | | 10446957 allocs/op | | 171802056 bytes/op | | 17 | |
| Cluster | 5 | Punti | 200 | Mapper | 5 | Reducer | 2 |
| 4m2.9829753s | | 1046328 allocs/op | | 170171360 bytes/op | | 15 | |
| Cluster | 5 | Punti | 500 | Mapper | 5 | Reducer | 2 |
| 2m51.4353833s | | 6030391 alloc/op | | 922528416 bytes/op | | 10 | |
| Cluster | 5 | Punti | 500 | Mapper | 10 | Reducer | 5 |
| 5m7.0041064s | | 5938643 alloc/op | | 903445720 bytes/op | | 9 | |
| Cluster | 5 | Punti | 800 | Mapper | 10 | Reducer | 10 |
| 5m41.6744888s | | 14772226 | | 2144435096 | | 10 | |
| | | alloc/op | | bytes/op | | | |

*Tabella 5 – Performance Llyod*

| **Standard K-means** | | | | | | | |
|---|---|---|---|---|---|---|---|
| Latenza | | Memoria allocata | | Bytes allocati | | Numero iterazioni | |
| Cluster | 5 | Punti | 200 | Mapper | 2 | Reducer | 2 |
| 1m50.9605014 s | | 1044689 allocs/op | | 171801352 bytes/op | | 17 | |
| Cluster | 5 | Punti | 200 | Mapper | 5 | Reducer | 2 |
| 4m3.4888661 s | | 1046325 allocs/op | | 170171976 bytes/op | | 15 | |
| Cluster | 5 | Punti | 500 | Mapper | 5 | Reducer | 2 |
| 3m21.4700887s | | 5959099 alloc/op | | 909405296 bytes/op | | 12 | |
| Cluster | 5 | Punti | 500 | Mapper | 10 | Reducer | 5 |
| 5m36.4919774s | | 5938651 alloc/op | | 903445424 bytes/op | | 10 | |
| Cluster | 5 | Punti | 800 | Mapper | 10 | Reducer | 10 |
| 5m42.3385246s | | 14772232 alloc/op | | 2144435696 bytes/op | | 10 | |

*Tabella 6 – Performance Standard K-means*

| **K-means++** | | | | | | | |
|---|---|---|---|---|---|---|---|
| Latenza | | Memoria allocata | | Bytes allocati | | Numero iterazioni | |
| Cluster | 5 | Punti | 200 | Mapper | 2 | Reducer | 2 |
| 1m38.5738274s | | 1072560 allocs/op | | 181671784 bytes/op | | 15 | |
| Cluster | 5 | Punti | 200 | Mapper | 5 | Reducer | 2 |
| 3m32.7970837s | | 1061626 allocs/op | | 175356432 bytes/op | | 13 | |
| Cluster | 5 | Punti | 500 | Mapper | 5 | Reducer | 2 |
| 5m36.2310378s | | 6198016 alloc/op | | 923242496 bytes/op | | 21 | |
| Cluster | 5 | Punti | 500 | Mapper | 10 | Reducer | 5 |
| 9m6.8351506s | | 6110487 alloc/op | | 903597352 bytes/op | | 17 | |
| Cluster | 5 | Punti | 800 | Mapper | 10 | Reducer | 10 |

| | | | |
|---|---|---|---|
| 5m11.8577959s | 15352244 alloc/op | 2362312448 bytes/op | 9 |

*Tabella 7 – Performance K-means++*

## VI. Conclusione

Prendendo in considerazioni i cluster con un numero di punti pari a 800, un numero di mapper pari a 10 e un numero di reducer pari a 10, è possibile notare che l'algoritmo Llyod e l'algoritmo Standard K-means non presentano grandi differenze dovuto al fatto che i centroidi di ogni cluster vengono scelti inizialmente in modo casuale.

Salvo alcune eccezioni, nella maggior parte dei casi si nota una differenza con l'algoritmo K-means++ con una latenza minore rispetto agli altri due ma con un'allocazione di memoria e bytes maggiore, come mostrato nei grafici seguenti.

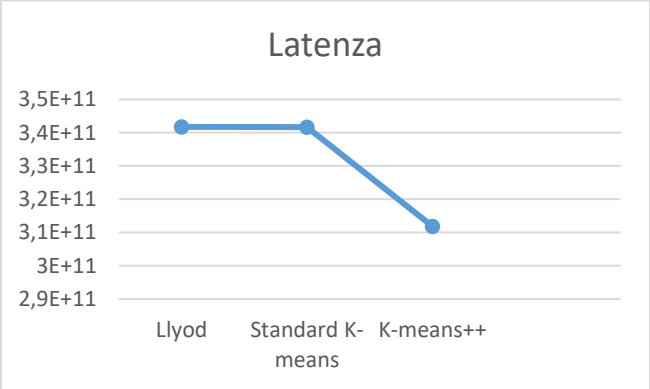


*Figura 1 - Latenza*

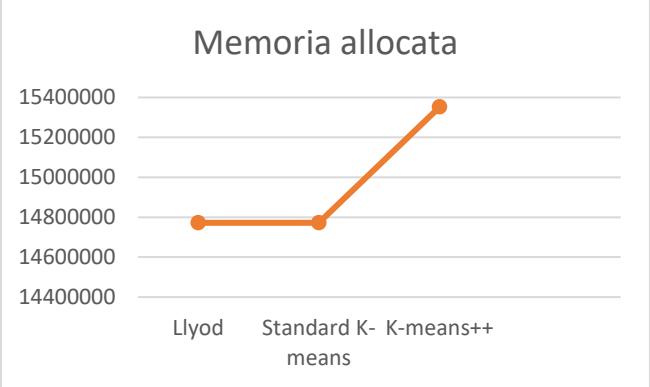


*Figura 2 – Memoria allocata*

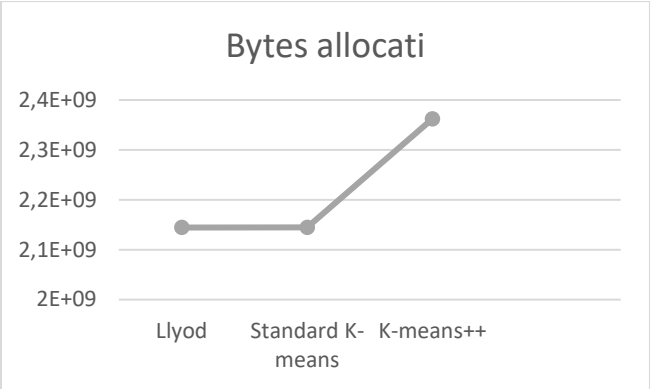


*Figura 3 – Bytes allocate*

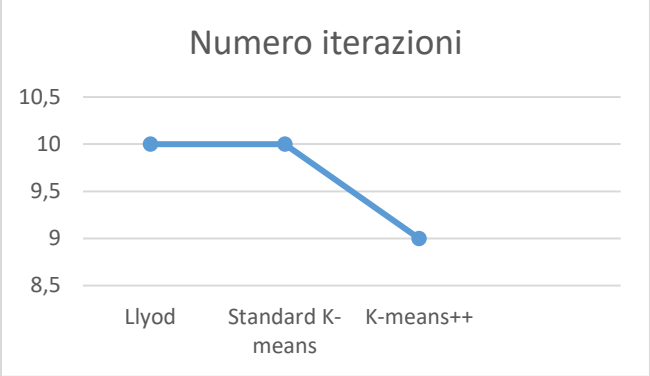


*Figure 1 – Numero iterazioni*

## Riferimenti bibliografici


[1] M. Bodoia. Mapreduce algorithms for k-means clustering, 2016. https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/bodoia.pdf.

[2] V. Cardellini. MapReduce and Hadoop, 2022. http://www.ce.uniroma2.it/courses/sabd2122/slides/MapReduce%26Hadoop.pdf..

[3] J. Leskovec, A. Rajaraman, and J. Ullman. Mining of Massive Datasets, chapter 2. Cambridge University Press, 3rd edition, 2020. http://infolab.stanford.edu/~ullman/mmds/ch2n. pdf.

[4] Leskovec, A. Rajaraman, and J. Ullman. Mining of Massive Datasets, chapter 7. Cambridge University Press, 3rd edition, 2020. http://infolab.stanford.edu/~ullman/mmds/ch7. pdf.

[5] https://pkg.go.dev

[6] https://gobyexample.com