SDCC – Progetto B1

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

Michele Salvatori
Facoltà di Ingegneria
Università di Roma Tor Vergata
Roma, Lazio, Italia
michelesalvv@gmail.com

ABSTRACT

L'obiettivo del progetto è quello di realizzare un'applicazione distribuita che implementi l'algoritmo di clustering ***k*-means** in versione distribuita secondo il paradigma di computazione **MapReduce**. L'applicazione è stata sviluppata utilizzando il linguaggio di programmazione `GO`.

1. INTRODUZIONE

L'algoritmo k-means permette di suddividere un insieme di punti in k cluster sulla base della **distanza euclidea**:

$$d(p,q) = \sqrt{\sum_i (p_i - q_i)^2}$$

La versione dell'algoritmo k-means offerta dal servizio è quella di **Lloyd** [1], descritta brevemente nel seguente pseudocodice:

```
choose k as the number of clusters
randomly choose k datapoints as centroids
repeat
  for each centroid do
    reassign each point to its closets centroid
    recalculate centroid as mean over all points assigned
  end for
until convergence
```

2. ARCHITETTURA E IMPLEMENTAZIONE

Il deployment del cluster di nodi mapper e reducer è stato fatto su singoli Docker Container, tutti in esecuzione su una singola istanza di AWS EC2 [*Figura 1*].

Per quanto riguarda il deployment sui Docker Container, è stato scelto di utilizzare la seguente immagine per ogni nodo del cluster, in quanto presenta già l'intero SDK di Golang:

- *docker.io/bitnami/golang:1.19* [2]

Tutti i nodi del cluster sono in grado di comunicare tra loro poiché interconnessi dalla rete virtuale creata di default dal tool di orchestrazione dei container `docker-compose`.

A livello software, la comunicazione nel cluster è stata implementata tramite l'utilizzo delle *chiamate a procedura remota (RPC)* offerte nativamente dal linguaggio Go tramite la libreria `net/rpc`.

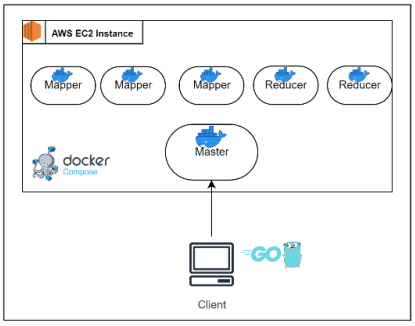

Figura 1 Infrastruttura di Rete

L'indirizzo IP dell'istanza EC2, che utilizzerà il client per usufruire del servizio, è stato assegnato staticamente: `3.229.222.181`.

Il paradigma di computazione MapReduce è stato implementato seguendo il modello utilizzato dal framework Hadoop MapReduce mostrata in [*Figura 2*], non utilizzando però nessun DFS sottostante.

2.1. Configurazione Cluster

Attraverso un file di configurazione `config.yml`, all'utente viene offerta la possibilità di cambiare alcuni parametri per l'esecuzione dell'algoritmo tra cui il `master_address`, `master_port` (default: `9001`), `worker_port` (default: `9999`), e la `threshold` di convergenza dell'algoritmo (default: `0.002`).

Inoltre, è stato scelto di implementare anche una versione ottimizzata del paradigma di computazione MapReduce che fa utilizzo di un **Combiner**. L'utente, tramite lo stesso file di configurazione, potrà disabilitare tale ottimizzazione.

Infine, è importante osservare che il Master fa utilizzo di tutti i Mapper che sono online nel cluster. Ad ognuno di essi il assegnerà un diverso chunk del dataset sul quale effettuare la fase di `Map()`, garantendo così la parallelizzazione.

Il numero di nodi Reducer che utilizzerà sarà invece pari al numero di cluster che l'algoritmo vuole individuare nel dataset, ovvero k, in modo tale che il k-esimo reducer si occuperà di calcolare esclusivamente il valore aggiornato del centroide del k-esimo

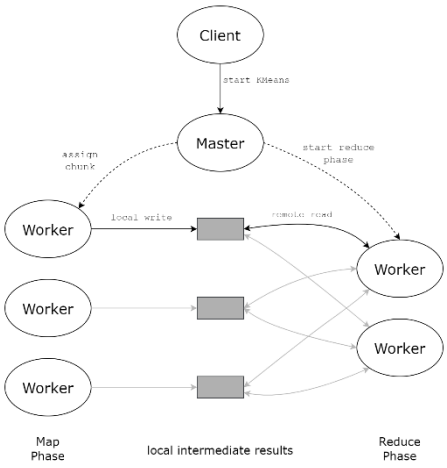

Figura 2 Execution Overview

cluster. Soltanto in questo modo si è in grado di garantire la parallelizzazione della fase di `Reduce()`.Nel caso in cui si utilizzi il Combiner, è necessario utilizzare un solo Reduce.

2.2. Connessione al Servizio

L'orchestrazione del cluster e l'inizializzazione dell'algoritmo viene effettuata da un nodo Master, la cui implementazione può essere visualizzata all'interno la directory `node/master` nella repository Github [3].

Ogni nodo Worker (Mapper/Reducer), al suo startup effettua la registrazione al servizio di clusterizzazione tramite la RPC `JoinMR()` esposta proprio dal nodo Master, comunicandogli le proprie informazioni che sono rappresentate dalla `struct WorkerInfo`.

Il Master, infatti, tiene traccia di ogni Mapper e Reducer tramite due `array` di strutture `WorkerInfo`, in modo da poter comunicare con loro una volta iniziato l'algoritmo. Di conseguenza la registrazione consiste nell'`append` delle informazioni dei singoli Worker nei rispettivi array, preceduti da controlli su eventuali Worker già registrati con lo stesso IP.

Analogamente, il master espone la RPC `ExitMR()`, per offrire la possibilità ai nodi worker di uscire dal cluster (*eg.* quando un worker cattura un segnale di interrupt).

2.3. Initialization Phase

Il client, per eseguire l'algoritmo invoca la `RPC KMeans()` esposta dal Master, alla quale passa in input la struttura `InputKMeans` che conterrà informazioni sul dataset da clusterizzare quali nome, dimensione, numero di entry e il valore scelto di k, ovvero quanti cluster l'algoritmo cercherà di individuare. L'RPC in questione andrà innanzitutto a verificare che le risorse del cluster siano disponibili, ovvero che ci sia almeno un nodo che esegua il task di Map, e almeno k nodi che eseguano il task di Reduce o, nel caso in cui sia attivo il Combiner, viene controllato che sia presente almeno un reducer.

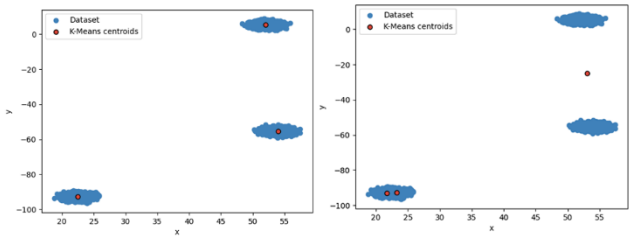

Figura 3 KMeans++ (sx) vs Starndard KMeans (dx)

Dopo aver caricato il dataset, il nodo master andrà ad effettuarne lo *splitting* suddividendolo in chunk di ugual dimensione, uno per ogni mapper collegato al servizio. Ogni chunk sarà rappresentato da un array di `Point`, struttura dati che a sua volta contiene all'interno un array di `[d]float64`, dove con `d` indichiamo la dimensione dei singoli punti.

Per il secondo step dell'algoritmo di Lloyd, la scelta dei k centroidi iniziali, sono state implementate due soluzioni alternative descritte di seguito.

2.3.1. *k*-means standard

Nella versione standard dell'algoritmo, gli iniziali k centroidi vengono semplicemente scelti in modo randomico. Tuttavia il k-means risulta essere **molto sensibile** [4] alla scelta dei primi k centroidi. Infatti, se l'algoritmo parte da centroidi che sono troppo vicini l'uno con l'altro, c'è una buona probabilità che la convergenza dell'algoritmo oscilli intorno ai cluster ottimi o comunque che vengano individuati cluster ottimi localmente ai centroidi iniziali, ma non globalmente su tutti i punti del dataset. In [Figura 3] possiamo vedere un esempio di questo scenario in cui sono state effettuate due esecuzioni consecutive dell'algoritmo sullo stesso dataset. Possiamo vedere che nella seconda esecuzione l'algoritmo non converge globalmente.

È possibile osservare questo comportamento specialmente su dataset piccoli perché se la selezione randomica è particolarmente sfortunata, ovvero vengono scelti dei centroidi di partenza che appartengono allo stesso cluster (sono troppo vicini tra loro), l'algoritmo avrà a disposizione pochi punti per convergere ed aggiustare il valore dei centroidi. Di conseguenza tenderà ad oscillare vicino ad una falsa convergenza.

2.3.2. *k*-means++

La versione k-means++ porta una miglioria sulla scelta dei k centroidi iniziali. L'idea alla base è quella di selezionare i centroidi iniziali non randomicamente ma in modo che essi siano il più distanti l'uno dall'altro. Così facendo, aumenta la probabilità che vengano scelti dei centroidi di partenza che appartengono a cluster differenti. L'algoritmo, fin dall'inizio, andrà così a coprire la maggior parte del *data space*. Di seguito viene riportato lo pseudocodice che implementa questa versione dell'algoritmo, di cui possiamo riconoscere i seguenti step:

1. Il primo centroide viene scelto casualmente tra i punti del dataset

2. Per ogni punto del dataset viene calcolata la distanza con il centroide appena scelto. Il punto più distante sarà il secondo centroide.

3. Per ogni punto del dataset viene calcolata la distanza euclidea con il centroide ad esso più vicino. Una volta calcolate tutte le distanze, il punto che avrà una distanza maggiore dal suo centroide più vicino, diventerà il successivo centroide.

4. Si itera il passo 3 finché non vengono selezionati *k* centroidi

```
first_centroid = random(points)
foundend_centroids++

for each point do
          dist = euclideandistance(point, first_centroid)
          if dist>=maxdist
                    secondcentroid = point
          founded_centroids++
end for

while founded_centroids < k do
          distances[] = calculate_distances(points, centroids)
          next_centroid = farthestpoint(distances[])
          founded_centroids++
end
```

L'implementazione completa di questo algoritmo è offerto dalla funzione `startingCentroidPlus()`, visionabile all'interno del file `master_impl.go`[5].

2.4. KMeans Execution

Dopo aver formato i vari chunk da distribuire ai mapper ed aver scelto i *k* centroidi iniziali, il nodo Master inizia ad eseguire un ciclo `while()` in cui, ad ogni iterazione, vengono comunicati ai mapper i chunk sui quali lavorare e gli attuali centroidi da utilizzare per calcolare le distanze euclidee dei punti.

L'invio dei dati verso i mapper viene effettuato lanciando una go-routine per ogni mapper registrato al servizio. All'interno della routine avviene la chiamata alla RPC `Map()` esposta dai mapper tramite il metodo `Call()` di `net/rpc`, in cui passiamo una struttura `MapperInput`, appositamente creata, che conterrà il chunk di punti e gli attuali centroidi sui quali lavorare.

La barriera di sincronizzazione che deve essere interposta tra la fine della fase di Map e l'inizio della fase di Reduce, è implementata tramite l'utilizzo di una mappa di **canali** Go. Ad ogni go-routine che invoca la fase di Map sui singoli mapper, viene associato un canale di questa mappa, sul quale verrà inserita la risposta una volta ricevuta dal mapper. Nella barriera di sincronizzazione, quindi, basterà iterare sulla mappa di canali ed attendere una risposta da ognuno di essi. L'implementazione della barriera di sincronizzazione è osservabile all'interno della funzione `waitMapperResponse()` [6].

Quando il master potrà iniziare la fase di Reduce, in modo analogo eseguirà *k* go-routine che andranno ad invocare l'RPC `Reduce()` dei nodi reducer necessari. Ai reducer non verranno inviati i risultati proveniente dalla fase di Map, bensì gli verrà comunicato la lista di mapper utilizzati nella fase precedente e l'indice del cluster sul quale dovranno lavorare (univoco per ogni reducer). In seguito, vedremo come i reducer effettuano una `read` remota dai mapper per ottenere i vari cluster calcolati.

- Se invece l'ottimizzazione del Combiner è attiva, verrà comunicata solamente la lista dei mapper ad un unico reducer.

Ogni risposta proveniente da un reducer viene mappata in una struttura `ReducerResponse`, contenente il centroide calcolato, o la lista di centroidi calcolati nel caso di utilizzo del Combiner, e alcuni metadati relativi al reducer stesso. Analogamente a quanto avviene per la fase di Map, ogni singola risposta viene ricevuta su un diverso canale Go.

Il Master, dopo aver ricevuto i nuovi centroidi calcolati dai reducer, controllerà se l'algoritmo ha raggiunto una convergenza. In caso contrario, procederà con una nuova iterazione del ciclo `while`.

2.4.1. Convergenza Algoritmo

È stato scelto di implementare la convergenza dell'algoritmo andando a verificare che non ci siano stati progressi rilevanti sul calcolo dei nuovi centroidi. Questo viene fatto comparando ogni centroide con il rispettivo centroide calcolato nell'iterazione precedente. Di seguito viene riportata l'effettiva implementazione:

```
if (checkConvergence(reducersReplies, prevCentroids)){
   break
}
. . .
func checkConvergence(actual, prev []Point) bool {
 for i, point := range actual {
   for j := 0; j < dimension; j++ {
     diff = point.Values[j] - prev[i].Values[j]
       if diff > cfg.Parameters.CONV_THRESH {
          return false
       }
    }
  }
  return true
}
```

È stata anche aggiunto un limite al numero di iterazioni eseguibili per evitare che l'algoritmi eseguiti infinitamente, soprattutto se viene utilizzato l'algoritmo standard [2.3.1] per la scelta dei centroidi di partenza.

2.5. Map Phase

Ricevuto l'input, rappresentato da `MapperInput` descritta precedentemente, ogni mapper va ad attribuire ad ogni punto del suo chunk il centroide più vicino tra quelli ricevuti in input. Ogni mapper formerà il nuovo cluster utilizzando *k* `slice` di `Point`, uno per ogni cluster, ai quali aggiungerà i punti che avranno la minor distanza dal *k*-esimo centroide.

```
var clusters = make([][]utils.Point, k)
for _, point := range *chunk {
  for i := 0; i < k; i++ {
    centroid := input.Centroids[i]
    euDistance := euclideanDistance(point, centroid, dimension)
    if euDistance <= minDistance || i == 0 {
      minDistance = euDistance
      centroidIndex = i
    }
  }
  clusters[centroidIndex]=append(clusters[centroidIndex], point)
  centroidIndex, minDistance = 0 }}
```

Dopo aver terminato la sua esecuzione, il mapper restituirà il controllo al master senza inviare in risposta i cluster appena computati, che manterrà in locale.

2.5.1. Combiner

Nel caso in cui l'esecuzione avvenisse utilizzando anche il Combiner, dopo aver individuato i nuovi cluster il mapper va anche a calcolare le k somme locali dei punti appena individuati. Per ogni cluster viene utilizzato uno `slice` di `float`, `centroidValues`, che conterrà alla fine le somme calcolate sulle d coordinate di ogni punto appartenente al cluster in considerazione. Di seguito è riportata parte dell'implementazione del Combiner.

```
var combined_values = make([]Point, k)
for j, cluster := range *clusters {
  centroidValues := make([]float64, dimension)
  for _, point := range cluster {
    for i := 0; i < dimension; i++ {
          centroidValues[i] += point.Values[i]
    }
  }
   combined_values[j].Values = centroidValues
}}
```

2.6. Reduce Phase

Come spiegato in 2.4, l'i-esimo reducer ha il compito di calcolare il nuovo centroide dell'i-esimo cluster, che ottiene andando ad invocare l'RPC `GetClusters()`[7] da ogni Mapper che ha partecipato nella fase precedente.

Nel caso in cui si utilizzi il combiner, oltre alle somme locali, il reducer riceverà in risposta da ogni mapper il **numero di punti** che ha assegnato ai k cluster. Successivamente andrà ad aggregare le varie somme ricevute da tutti i mapper ed eseguirà il calcolo del nuovo centroide i, dividendo le somme aggregate per il numero totale di punti che ogni mapper ha associato all'i-esimo cluster. Di seguito viene riportata una versione dell'implementazione semplificata per motivi di leggibilità. La versione completa è osservabile in `workers/reducer_rpc.go`[8].

```
aggregatedSums = make([]Point,k)
for i, mapper := range in.Mappers {
  mapper_response := retrieveData(mapper)
  numOfPoints := mapper_response.ClusterCardinality
  localSums := mapper_response.Sums
  aggregate(&aggregatedSums,localSums,numOfPoints)
}
newCentroids := recenter(aggregatedSums)
```

I nuovi centroidi verranno inviati in risposta al Master, il quale provvederà a comunicarli al Client al raggiungimento della convergenza.

3. TESTING E ANALISI DELLE PERFORMANCE

In questa sezione del documento valuteremo le performance ottenute, in termini di tempo di esecuzione dell'algoritmo, al variare del numero di Mapper utilizzati e dei vari parametri di configurazione, tra cui la possibilità di utilizzare o meno l'ottimizzazione del combiner. In particolare, i test sono stati effettuati su dataset composti da `1000`, `10000`, `100000`, `500000`, `1000000` e `5000000` punti.

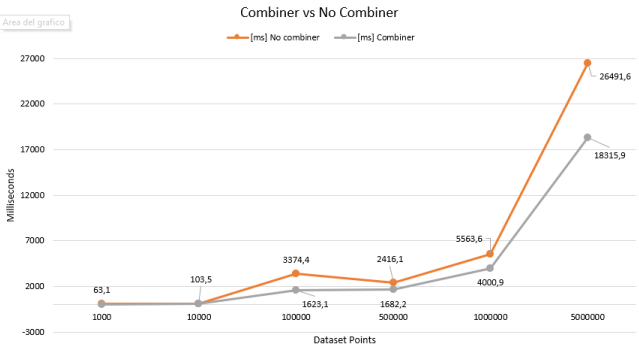

Figura 4 Analisi ottimizzazione Combiner

L'intero cluster esegue su un'istanza EC2 ad esclusione del Client che viene utilizzato in locale dall'utente. Tuttavia, i tempi di esecuzioni vengono calcolati dal Master e resi disponibili al Client. Ciò significa che nei tempi di esecuzione non è incluso il tempo necessario al trasferimento dei risultati tra il server EC2 e l'utente locale, ma sono invece inclusi il tempo necessario ad effettuare lo *splitting* del dataset e la scelta dei primi k centroidi iniziali, in quanto sono passi fondamentali dell'algoritmo, anche se non eseguiti in modo parallelo.

Di seguito sono riportate le caratteristiche dell'istanza EC2 utilizzata:

```
EC2 t3.Medium Instance: Intel Xeon, @2 vCPU up to
3.1GHz, RAM 4GB, Amazon Linux 5.10, Docker-compose
@1.29.2
```

3.1. Generazione Dataset

Al fine di generare un dataset che presenti almeno un cluster, è stato realizzato uno script `Python`[9] in cui è viene utilizzata la funzione `make_blobs()`[11] della libreria `scikit-learn`. Al termine della generazione viene anche mostrato e salvato il dataset in formato `png`. Nel `README` presente nella repository del progetto vengono descritte le istruzioni per utilizzare lo script.

3.2. Validazione Risultati

Al raggiungimento della convergenza dell'algoritmo, all'utente verranno forniti i valori dei k centroidi in formato `csv`, oltre alla visualizzazione sulla console. Per poter validare i risultati dell'algoritmo, viene offerto all'utente un ulteriore script `Python`[10] che costruisce uno *scatter plot* partendo dai punti del dataset originale, ai quali sovrappone i punti dei centroidi risultanti dal KMeans, evidenziandoli in rosso, così come è possibile vedere in [Figura 3].

3.3. Analisi Performance

Una prima analisi che è stata effettuata riguarda l'efficienza dell'ottimizzazione del Combiner. Il confronto è stato effettuato tra i tempi medi ottenuti in `10 run` dell'algoritmo con Combiner, e i tempi medi ottenuti in `10 run` senza utilizzare il Combiner. Osservando i risultati [Figura 4], traiamo beneficio dal Combiner soprattutto se utilizzato su grandi dataset, sui quali riusciamo a risparmiare quasi `10s`. Questo è spiegabile dal fatto che viene utilizzato un solo Reducer che dovrà semplicemente lavorare su k

Dataset Points	1 Mapper	2 Mapper	3 Mapper	4 Mapper	6 Mapper	8 Mapper	10 Mapper
1000	16,3	25,3	30,3	31,4	52,2	59,8	77,4
10000	69,6	59,4	66,9	81,1	89,2	118,7	123
500000	1708,8	1620,2	1682,2	1712,2	1719,5	1683,1	1699,8
1000000	4350,3	4039,2	4000,9	3940,9	3930	3962,7	4046
5000000	20630,8	20245,1	18315,9	17554,7	17050,5	15758,4	19347,4

Figura 5 Tempi di esecuzione al variare del numero di Mapper

somme già computate. Di conseguenza, il Master riceverà i nuovi centroidi da un solo Reducer, diminuendo anche la quantità di dati che vengono scambiati, e quindi non dovrà effettuare nessun'operazione di aggregazione di risultati prima di verificare la convergenza dell'algoritmo.

Successivamente sono state analizzati i tempi di esecuzione ottenuti al variare del numero di nodi che vanno ad eseguire la fase di Map. I risultati complessivi sono riportati nella tabella in [Figura 5] dove possiamo notare come su dataset di piccole dimensioni, ad esempio `1000` e `10000` punti, i tempi di esecuzione aumentino al crescere del numero dei Mapper. Questo perché si ottiene un grande beneficio dai paradigmi di computazione parallela, come il MapReduce, solamente su grandi quantità di dati in quanto altrimenti si andrebbe a pagare, in termini di tempi di esecuzione, l'overhead di comunicazione che risulta essere maggiore rispetto alla quantità di dati che effettivamente sono utili ai fini dell'algoritmo.

Scenario totalmente opposto si ha su dataset elevate dimensioni. In Figura 6 possiamo vedere come, su un dataset composto da **5 milioni** di punti, la computazione parallela porta un risparmio di poco più di `2s` solamente aggiungendo due nodi Mapper. Si ottiene un esecuzione `5s` più veloce incrementando il numero di nodi mapper fino ad 8. Tuttavia, eseguendo l'algoritmo su 10 Mapper, si torna ad avere tempi di esecuzione maggiori, sia a causa di comunicazione eccessiva non necessaria, sia a causa dell'elevato numero di container Docker in esecuzione sulla macchina.

Le esecuzioni effettuate su altri dataset di dimensione minore sono state graficate e rese disponibili in `results/exectime.xls`[13].

4. CONCLUSIONI

Come visto dai risultati, l'algoritmo riesce a scalare bene su dataset di grandi dimensioni mentre per dataset composti da pochi punti la parallelizzazione della computazione non porta notevoli benefici. Questo era un comportamento comunque atteso essendo *MapReduce* un paradigma di computazione di tipo *Single Program Multiple Data*. Inoltre è noto come *MapReduce* non offre grandi prestazioni nell'esecuzione di algoritmi **iterativi**.

Alcune limitazioni riscontrate riguardano la potenza computazionale offerta dall'istanza EC2 utilizzata. Sono stati infatti eseguiti alcuni test provando ad eseguire l'algoritmo su dataset composti da *miliardi* di punti. Tuttavia, l'esecuzione falliva durante lo splitting del dataset effettuata dal nodo Master, poiché non è un operazione che viene parallelizzata. Si è scelto comunque di non utilizzare istanze EC2 più performanti per simulare uno scenario il più possibile vicino alla realtà in cui, a causa dei costi delle istanze, si cerca di evitare l'*overprovisioning* delle risorse.

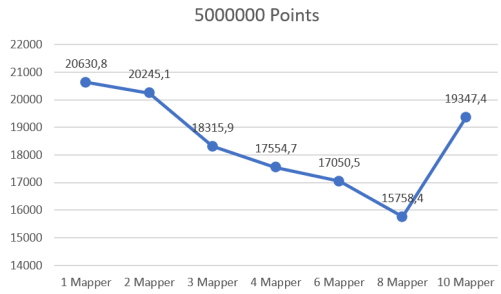

Figura 6 Tempi di esecuzione su 5000000 punti

Proprio il nodo Master è la causa di un'ulteriore limitazione di questa implementazione in quanto rappresenta un *Single Point of Failure*, essendo un componente centralizzato.

Infine, l'architettura non è completamente tollerante ai guasti poiché non sono stati implementati particolari meccanismi di elezione, ad esempio del nodo Master nel caso in cui questo fallisca. Inoltre, non vengono assegnati nuovi task a nuovi nodi worker nel caso in cui i precedenti subiscano un fallimento. Infatti, nel Master, è stata implementata solamente la cattura del segnale di `SIGINT` generato dai worker[12] per avvisarlo che, nell'iterazione successiva, lo specifico worker non avrebbe preso parte all'esecuzione dell'algoritmo, poiché uscito spontaneamente dal cluster.

5. RIFERIMENTI

[1] https://www.cs.utah.edu/~jeffp/teaching/cs5955/L10-kmeans.pdf

[2] https://hub.docker.com/r/bitnami/golang

[3] https://github.com/michsalvv/K-Means-on-MapReduce/tree/master/node/master

[4] https://stackoverflow.com/questions/33594749/proof-that-k-means-always-converges

[5] https://github.com/michsalvv/K-Means-on-MapReduce/blob/master/node/master/master_impl.go#L29

[6] https://github.com/michsalvv/K-Means-on-MapReduce/blob/master/node/master/master_impl.go#L195

[7] https://github.com/michsalvv/K-Means-on-MapReduce/blob/master/node/workers/mapper_rpc.go#L78

[8] https://github.com/michsalvv/K-Means-on-MapReduce/blob/master/node/workers/reducer_rpc.go

[9] https://github.com/michsalvv/K-Means-on-MapReduce/blob/master/datasets/dataset_gerator.py

[10] https://github.com/michsalvv/K-Means-on-MapReduce/blob/master/datasets/check_results.py

[11] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

[12] https://github.com/michsalvv/K-Means-on-MapReduce/blob/master/node/workers/worker.go#L61

[13] https://github.com/michsalvv/K-Means-on-MapReduce/blob/master/results/exectime.xlsx