

Università degli Studi di Salerno



Dipartimento di Informatica

Corso di Sistemi Operativi Avanzati

**Comparative experimental analysis of k-means, spherical k-means and bisec
k-means algorithms applied to document clustering**

Professori

Prof. Cattaneo Giuseppe
Dr. Roscigno Gianluca

Studenti

Farisano Gino
Luciano Giuseppe

1.	Il problema	1
1.1	Introduzione	1
1.2	Term Frequency e Term Frequency Inverse Document Frequency	1
1.3	Analisi dei cluster.....	2
1.3.1	Nozioni di base e definizioni.....	3
1.3.2	Misure di distanza	4
1.3.3	Metodi di raggruppamento.....	6
1.4	Metodi non gerarchici	7
1.4.1	KMeans.....	7
1.4.2	Bisecting KMeans	8
2.	Sequenziale	10
2.1	Introduzione	10
2.2	Dataset.....	10
2.3	Configurazione Hardware	10
2.4	KMeans	11
2.4.1	Note implementative	11
2.4.2	Tempi.....	11
2.4.3	Profile	12
2.5	Bisecting KMeans.....	14
2.5.1	Note implementative	14
2.5.2	Tempi.....	15
2.5.3	Profile	15
2.6	Note	16
3.	Hadoop.....	17
3.1	Introduzione all'architettura	17
3.1.1	Distributed Cache	18
3.2	Formato dei dati in input.....	18
3.3	La classe Document	18
3.4	KMeans	18
3.4.1	Pseudo codice	19

3.5	Bisecting KMeans.....	21
3.5.1	Problematiche	21
3.5.2	Pseudo codice	21
4.	Test e benchmarking su Hadoop	23
4.1	Introduzione	23
4.2	Configurazione Hadoop	23
4.3	KMeans - dataset da 5,041 GB	25
4.3.1	Test su 8 macchine (9 centroidi, 6 iterazioni).....	25
4.3.2	Test su 16 macchine (9 centroidi, 6 iterazioni).....	25
4.3.3	Test su 31 macchine (9 centroidi, 6 iterazioni).....	25
4.4	KMeans - dataset da 45,369 GB.....	26
4.4.1	Test su 16 macchine (9 centroidi, 6 iterazioni).....	26
4.4.2	Test su 31 macchine (9 centroidi, 6 iterazioni).....	26
4.4.3	Ulteriori test KMeans (2centroidi, 2 iterazioni).....	26
4.5	BKMeans - dataset da 5,041 GB.....	27
4.5.1	Test su 8 macchine (9 centroidi, 2 iterazioni).....	27
4.5.2	Test su 16 macchine (9 centroidi, 2 iterazioni).....	27
4.5.3	Test su 31 macchine (9 centroidi, 2 iterazioni).....	27
4.5.4	Test su 31 macchine – blocchi da 64 MB (9 centroidi, 2 iterazioni)	27
4.6	BKMeans - dataset da 45,369 GB.....	27
4.6.1	Test su 16 macchine (2 centroidi, 2 iterazioni).....	27
4.6.2	Test su 31 macchine (2 centroidi, 2 iterazioni).....	27
4.7	SpeedUp e efficienza	27
4.7.1	KMeans (2 centroidi, 2 iterazioni) – dataset 45,369 GB	28
4.7.2	BKMeans (2 centroidi, 2 iterazioni) – dataset 45,369 GB	28
4.8	Conclusioni sui risultati.....	29
4.9	Grafici KMeans.....	30
4.9.1	CPU	30
4.9.2	Memoria.....	31
4.9.3	Lettura da disco	31

4.9.4	Scrittura su disco	32
4.9.5	Thoughtput pacchetti di rete ricevuti	32
4.9.6	Throughput pacchetti di rete inviati	33
4.9.7	Job1 – Centroidi Random	34
4.9.8	Job2 - KMeans	35
4.9.9	Job3 – KMeans	36
4.10	Grafici Bisecting KMeans	37
4.10.1	CPU	37
4.10.2	Memoria.....	37
4.10.3	Lettura da disco.....	38
4.10.4	Scrittura su disco	38
4.10.5	Throughput pacchetti di rete ricevuti	39
4.10.6	Throughput pacchetti di rete inviati	39
4.11	Conclusioni	40
5.	Esperimenti sul clustering	i
5.1	Dataset TF	ii
5.1.1	Bisecting KMeans con metrica euclidea	ii
5.1.2	Bisecting KMeans con metrica del coseno	iii
5.1.3	KMeans con metrica euclidea	iv
5.1.4	KMeans con metrica del coseno	v
5.2	Dataset TFIDF.....	vi
5.2.1	Bisecting KMeans con metrica euclidea	vi
5.2.2	Bisecting KMeans con metrica del coseno	vii
5.2.3	KMeans con metrica euclidea	viii
5.2.4	KMeans con metrica del coseno	ix
5.3	Centroidi scelti non a caso	x
5.3.1	Daset TF e KMeans con metrica del coseno	x
5.4	Analisi dei risultati	x
6.	Riferimenti.....	xi

1. Il problema

1.1 Introduzione

Questo documento presenta i risultati di alcune tecniche di clustering, il KMeans e il bisesting KMeans, nel document clustering [14].

In particolare la prima fase del document clustering consiste nell'estrazione delle caratteristiche (insieme di parole più significative per un dato documento); la fase successiva, invece, attraverso l'uso di metodi gerarchici o non gerarchici, consente di raggruppare i documenti tra loro più simili in uno stesso gruppo utilizzando delle misure di distanza oppure di similarità.

Ciò premesso, il nostro obiettivo in questo lavoro è stato duplice: testare la bontà (qualità) di clusterizzazione del KMeans e bisecting KMeans e verificare se tali algoritmi, in un'implementazione distribuita, “scalassero”. A tal proposito i test eseguiti sono stati condotti su una, 8, 16 e 31 macchine utilizzando per l'implementazione distribuita il framework Hadoop. Il dataset utilizzato, di circa 5 GB, è l'output prodotto dal sottogruppo1 [1] del nostro team.

1.2 Term Frequency e Term Frequency Inverse Document Frequency

Per estrarre informazione dal testo vi sono diversi modelli matematici utilizzabili, tra cui il VSM (Vector Space Model) [4], ampiamente utilizzato in contesti in cui è necessario classificare documenti. In particolare, una collezione di d documenti descritti da t termini può essere rappresentata come una matrice A $t \times d$ chiamata comunemente *term-document*. Le colonne del vettore (*document-vector*) rappresentano i documenti nella collezione e le righe del vettore, chiamate *term-vector* rappresentano una particolare parola. Ciò premesso, il valore di ogni componente $a_{i,j}$ rappresenta il grado di relazione tra il termine i e il documento j . Tale valore, chiamato *term-weighting* può essere:

- **Binario:** $a_{i,j} = 1$ se il termine appare nel documento, 0 altrimenti.
- **Term frequency:** $a_{i,j} = tf_{i,j}$ dove $tf_{i,j}$ indica quante volte il termine i compare nel documento j .
- **Term frequency - Inverse Document Frequency:** questo schema, a differenza di quanto accade per il TF, include un *global-weight* di una parola. Ad esempio, se una specifica parola è presente in tutti i documenti è giusto attribuirle un peso minore, invece, se appare solo in un certo tipo di documenti mentre negli altri no, quel termine è molto rappresentativo ed è quindi giusto attribuirgli un peso maggiore. Per capire meglio questo concetto, consideriamo il seguente esempio: data la seguente tabella

	Doc1	Doc2	Doc3	Doc4
Term1	2	1		3
Term2		2		1

Img1: Term-document Matrix

Term1 ricorre in 3 documenti mentre Term2 in 2. Quindi il peso globale di un termine è dato da:

$$\log(\text{numero totale di documenti} / \text{numero di documenti in cui il termine da analizzare appare})$$

Pertanto avremo Term1 = $\log(4/3)=0.124$. Allo stesso modo il peso globale di Term2 = $\log(4/2)=0.30$. Così il peso globale rappresenta l'importanza generale del termine e decresce all'aumentare del numero di documenti che contengono il termine in esame. Lo schema TF-IDF quindi mira a bilanciare le occorrenze locali/globali e può essere definito come $a_{i,j} = tf_{i,j} \cdot \log(N/df_i)$ dove $tf_{i,j}$ è il TF nel documento d_j , df_i denota il numero di documenti in cui il termine i compare e N il numero totale di documenti.

In [1] dopo aver estratto le features dai documenti utilizzando tecniche quali:

- Rimozione delle stop word
- Stemming
- Lemmatisation
- Pruning
- Ontologie
- ...

sono stati utilizzati gli schemi TF e TFIDF per produrre l'input utilizzato dagli algoritmi discussi in questo lavoro.

1.3 Analisi dei cluster

L'analisi dei cluster [8] è una metodologia che permette di raggruppare in sottoinsiemi, detti cluster, entità appartenenti ad un insieme più ampio. I vari metodi attraverso cui si attua l'analisi dei cluster hanno in comune uno scopo generale: ottenere raggruppamenti in base alla somiglianza in modo che gli elementi di uno stesso gruppo siano tra loro il più possibile simili e gli elementi appartenenti a gruppi distinti siano tra loro il più possibile diversi. In altre parole tale analisi ha lo scopo di distribuire le osservazioni in gruppi in modo tale che il grado di naturale associazione sia alto tra i membri dello stesso gruppo e basso tra i membri di gruppi diversi. In questo modo si otterrà quindi un'alta omogeneità all'interno dei gruppi e un'alta eterogeneità tra gruppi distinti.

Ad esempio, prendendo in considerazione il dataset da noi analizzato (documenti) e fissando a 2 il numero di cluster da ottenere come risultato dell'esperimento, l'output

desiderato è una suddivisione dei paper in letterari e scientifici. Ovviamente, incrementando il numero di cluster da ottenere in output, il risultato desiderato è un'ulteriore affinamento della classificazione (ad esempio, paper di medicina e biologia ecc. - paper di matematica, fisica, informatica e ingegneria - paper di letteratura, storia e arte).

Al fine di comprendere meglio quanto detto vediamo nel prossimo paragrafo una descrizione formale del problema del clustering.

1.3.1 Nozioni di base e definizioni

Sia $I = \{I_1, I_2, \dots, I_n\}$ un insieme di n individui (entità o unità) appartenenti ad una popolazione ideale Π e assumiamo che esista un insieme di caratteristiche (features) $C = \{C_1, C_2, \dots, C_p\}$ che sono osservabili e sono possedute da ogni individuo in I .

Denotiamo con il simbolo x_{ij} il valore della misura della caratteristica j -esima relativa all'individuo I_i e con $X_i = (x_{i1}, x_{i2}, \dots, x_{ip})$ ($i = 1, 2, \dots, n$) il vettore di cardinalità $1 \times p$ di tali misure. Quindi, in generale, l'iniziale naturale collezione dei dati su cui il ricercatore deve operare consiste di un insieme di n vettori di misure $\{X_1, X_2, \dots, X_n\}$ che descrive l'insieme I degli individui a cui è associata una matrice di misure X di cardinalità $n \times p$, ossia

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix} = (X_1, X_2, \dots, X_n)^T$$

Il problema dell'analisi dei cluster consiste nel determinare m sottoinsiemi, detti cluster, di individui in I , con m intero minore di n , tali che I_i appartenga soltanto ad un unico sottoinsieme. Gli individui che sono assegnati allo stesso cluster sono detti simili mentre gli individui che sono assegnati a differenti cluster sono detti dissimili [8].

È importante osservare come la scelta delle variabili (delle caratteristiche da osservare) sia strettamente condizionata dallo scopo dell'indagine e presupponga l'esistenza, seppure ad uno stato iniziale, di un modello logico; essa, infatti, riflette il giudizio del ricercatore sull'importanza delle proprietà utili a descrivere il fenomeno in funzione del quale deve essere svolta l'analisi dei cluster. In particolare, nell'esperimento condotto si è scelto di utilizzare come features le parole appartenenti ai diversi documenti (a cui è associato un numero indicante il term frequency o il tf-idf) che, presentando tutte la stessa unità di misura, non sono state standardizzate. Per quanto riguarda il numero di queste, invece, sarebbe stato opportuno effettuare un taglio di alcune di loro; infatti, includere variabili (nel nostro caso parole) con un piccolo potere discriminante può appiattire le differenze tra gruppi, mentre includere variabili con un elevatissimo potere discriminante può rendere inutile l'inclusione di altre variabili strettamente collegate al fenomeno. Nel nostro caso, però, un taglio alle caratteristiche avrebbe prodotto "pochi dati" (circa 1 GB) e l'utilizzo di Hadoop sarebbe stato inutile in quanto tale sistema funziona bene con tanti dati, i c.d Big Data.

1.3.2 Misure di distanza

Per risolvere il problema del clustering è opportuno definire i termini somiglianza e differenza cioè, occorre precisare cosa significa la somiglianza di due individui I_i e I_j assegnati allo stesso cluster e la differenza di due individui assegnati a differenti cluster. La somiglianza può essere definita mediante un coefficiente di similarità $s_{ij} = s(X_i, X_j)$ oppure mediante una misura di distanza $d_{ij}=d(X_i, X_j)$ tra due individui I_i e I_j ($i \neq j$). Mentre i coefficienti di similarità assumono valori compresi tra 0 e 1, le misure di distanza possono assumere qualsiasi valore reale maggiore o uguale a zero. Un criterio per risolvere il problema di clustering potrebbe essere quello di assegnare due individui I_i e I_{ji} ($i \neq j$) allo stesso cluster se il coefficiente di similarità tra i punti X_i e X_j è prossimo ad 1 oppure se la distanza tra i punti X_i e X_j è sufficientemente piccola e a differenti cluster se il coefficiente di similarità tra i punti è prossimo ad 0 oppure se la distanza tra i punti è sufficientemente grande.

Più formalmente:

Definizione 1. Una funzione a valori reali $d(X_i, X_j)$ è detta funzione distanza se e soltanto se essa soddisfa le seguenti condizioni:

- (i) $d(X_i, X_j)=0$ se e solo se $X_i = X_j$, con X_i e X_j
- (ii) $d(X_i, X_j) \geq 0$ per ogni X_i e X_j
- (iii) $d(X_i, X_j) = d(X_j, X_i)$ per ogni X_i e X_j
- (iv) $d(X_i, X_j) \leq d(X_i, X_k) + d(X_k, X_j)$ per ogni X_i, X_j e X_k

La proprietà (i) implica che X_i è a distanza zero da se stesso e che ogni due punti a distanza nulla debbono essere identici. La proprietà (ii) afferma che la funzione distanza è non negativa. La proprietà (iii) impone la simmetria richiedendo che la distanza tra X_i e X_j deve essere la stessa della distanza tra X_j e X_i . La proprietà (iv), nota come diseguaglianza triangolare, richiede che la distanza tra X_i e X_j debba essere sempre minore o uguale della somma delle distanze di ognuno dei due vettori considerati da qualunque altro terzo vettore X_k [8].

In generale, le distanze tra tutte le possibili coppie di unità sono inserite in una matrice simmetrica D di cardinalità $n \times n$:

$$D = \begin{pmatrix} 0 & d_{12} & \cdots & d_{1n} \\ d_{21} & 0 & \cdots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \cdots & 0 \end{pmatrix},$$

dove $d_{ij} = d(X_i, X_j)$ ($i, j = 1, 2, \dots, n$). Dalla Definizione 1 segue che i termini sulla diagonale principale sono tutti uguali a zero mentre i termini simmetrici sono uguali a due a due. È sufficiente, pertanto, considerare solamente la matrice triangolare al di sopra o al di sotto della diagonale principale di D .

Esistono innumerevoli funzioni distanza (che rispettano almeno le quattro proprietà della Definizione 1) ognuna con i propri pregi e difetti. In particolare, in questo lavoro sono state utilizzate la distanza euclidea e il coseno di similitudine [8].

1.3.2.1 Distanza euclidea

La più familiare misura di distanza è la metrica euclidea, così definita:

$$d(X_i, X_j) = \left[\sum_{k=1}^p (x_{ik} - x_{jk}) \right]^{1/2}$$

dove x_{ij} è il valore della k-esima caratteristica dell'individuo I_i . Nella fattispecie ogni documento (paper) è stato rappresentato con un'hash-map (una matrice sparsa avrebbe richiesto molti valori nulli) in modo tale da controllare in tempo $\theta(1)$ la presenza o meno della parola che si intende ricercare.

Di seguito mostriamo lo pseudocodice della metrica implementata:

```

distanza = 0
Per ogni <w1,v1> in documento1:
    se parola1 in documento2 {
        v2 = documento2.get(parola1) //il valore restituito è zero se la parola non è presente nel
        documento
        distanza += |v1 - v2|
    } altrimenti {
        distanza += v1
    }
Per ogni <w2, v2> in documento2 non presente in documento1:
    distanza += v2

return distanza

```

1.3.2.2 Coseno di similitudine

Il coseno di similitudine [6,7], o *cosine similarity*, è una tecnica euristica per la misurazione della similitudine tra due vettori, effettuata calcolando il coseno tra di loro; è usata generalmente per il confronto di testi nel *data mining* e nell'analisi del testo. In base alla definizione del coseno quindi, dati due vettori, si otterrà sempre un valore di similitudine compreso tra -1 e +1, dove -1 indica una corrispondenza esatta ma opposta (ossia un vettore contiene l'opposto dei valori presenti nell'altro) e +1 indica due vettori uguali.

Nel caso dell'analisi del testo, poiché le frequenze dei termini sono sempre valori positivi, si otterranno valori che vanno da 0 a +1, dove +1 indica che le parole contenute nei due testi sono le stesse (ma non necessariamente nello stesso ordine) e 0 che non c'è nessuna parola che appare in entrambi.

Il metodo proposto è descritto come segue: in primo luogo la similarità del coseno tra i termini del cluster e quelli del documento da processare sono calcolati utilizzando l'equazione:

$$d(A, B) = 1 - \cos(\theta) = 1 - \frac{A \cdot B}{|A||B|} = 1 - \frac{\sum_{k=1}^n A(k)B(k)}{\sqrt{\sum_{k=1}^n A(k)^2} \sqrt{\sum_{k=1}^n B(k)^2}}$$

dove A e B indicano rispettivamente il documento da analizzare e un cluster; poi, il documento avente il valore di similarità massima rispetto ai cluster testati è aggiunto al rispettivo cluster.

Di seguito mostriamo lo pseudocodice della metrica implementata:

```

A = B = dividendo = 0
For <w1,v1> in documento1:
    A += v1*v1
    dividendo += v1 * documento2.getOrDefault(w1,0)

For <w2, v2> in documento2:
    B += v2*v2

return 1 - (dividendo / (sqrt(A))*(sqrt(B)))

```

1.3.3 Metodi di raggruppamento

Una volta scelta la misura di distanza (o di similarità) si pone il problema di procedere alla scelta di un idoneo algoritmo di raggruppamento delle unità osservate. I metodi di raggruppamento si distinguono in tre tipi [8]:

- metodi di enumerazione completa
- metodi gerarchici
- metodi non gerarchici

1.3.3.1 Metodi di enumerazione completa

Supponiamo di considerare un insieme $\{I_1, I_2, \dots, I_n\}$ di n individui e sia m il numero di cluster. Il numero di partizioni di un'insieme di n individui in m cluster non vuoti può essere valutato calcolando il numero di modi in cui è possibile sistemare n individui in m gruppi con nessun gruppo vuoto. Formalmente:

$$\binom{n}{m} = \frac{n!}{(n-m)! m!}$$

1.3.3.2 Metodi gerarchici

I metodi gerarchici sono distinti in **agglomerativi** che procedono per aggregazioni successive delle unità partendo da n gruppi formati da un singolo individuo e **divisivi** che,

invece, partono da un solo gruppo formato da tutti gli individui e procedono per divisioni successive fino a giungere a gruppi formati da una sola unità. I metodi gerarchici hanno due vantaggi: quello di fornire una visione completa dell'insieme in termini di distanza (o di coefficienti di similarità) e quello di non comportare la scelta a priori del numero di cluster oppure la scelta a priori di parametri per la determinazione automatica del loro numero. Invece, uno svantaggio di tali metodi è che essi non consentono di riallocare gli individui che sono stati già classificati ad un livello precedente dell'analisi (peculiarità dei metodi non gerarchici).

A tal proposito vediamo nel prossimo i metodi non gerarchici ponendo attenzione agli algoritmi KMeans e bisecting KMeans.

1.4 Metodi non gerarchici

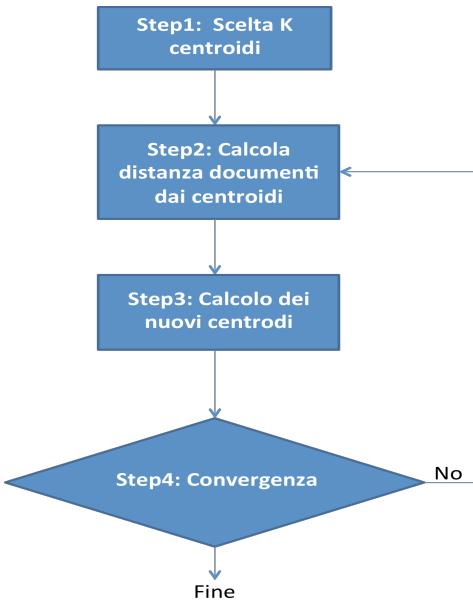
L'obiettivo dei metodi non gerarchici è quello di ottenere un'unica partizione degli n individui di partenza in cluster e, a differenza dei metodi gerarchici, in tali tecniche è consentito riallocare gli individui già classificati ad un livello precedente dell'analisi. In particolare gli algoritmi di tipo non gerarchico procedono, data una prima partizione, a riallocare gli individui nel gruppo con centroide più vicino, fino a che per nessun individuo si verifica che sia minima la distanza rispetto al centroide di un gruppo diverso da quello a cui esso appartiene.

1.4.1 KMeans

Il metodo non gerarchico più utilizzato prende il nome di KMeans ed è dovuto a Hartigan e Wong (1979) [13]. Tale metodo richiede che il numero di cluster sia specificato a priori e consiste dei passi descritti nel seguente algoritmo:

Algoritmo 1.

- Step1: Fissare a priori (o scegliere a caso) k punti di riferimento iniziali, i centroidi, che inducono ad una prima partizione provvisoria;
- Step2: Considerare tutti gli individui e attribuire ciascuno di essi al cluster individuato dal centroide da cui ha distanza minore;
- Step3: Calcolare il baricentro (il centroide) di ognuno dei k gruppi così ottenuti. Tali centroidi costituiscono i punti di riferimento per i nuovi cluster;
- Step4: Valutare la distanza di ogni unità da ogni centroide ottenuto al passo precedente. Se la distanza minima non è ottenuta in corrispondenza del centroide del gruppo di appartenenza, allora si ritorna allo Step2.



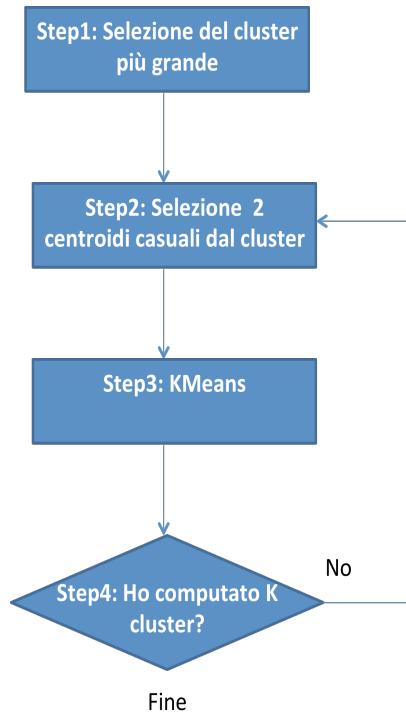
I vantaggi del metodo KMeans sono la velocità di esecuzione dei calcoli e l'estrema libertà che viene lasciata agli individui di raggrupparsi e allontanarsi. Uno svantaggio è invece che la classificazione finale può essere influenzata dalla scelta iniziale dei k vettori delle caratteristiche come punti di riferimento, dall'ordine in cui sono presi tali vettori e naturalmente dalle proprietà geometriche dei vettori delle misure.

1.4.2 Bisecting KMeans

Il Bisecting KMeans [9,11] è un algoritmo di clustering ibrido che combina tecniche gerarchiche e non gerarchiche, consentendo una clusterizzazione migliore e più veloce rispetto ad altri algoritmi di clustering dell'una o l'altra famiglia. In particolare il bisecting a partire da un unico grande cluster (nel nostro caso, tutti i documenti) lavora nel seguente modo:

Algoritmo 2.

1. Scegliere un cluster da dividere
2. Scegliere a caso due documenti dal cluster che si intende dividere
3. Eseguire il KMeans #iterazioni volte sul cluster individuato al punto 1
4. Eseguire punto 1, punto 2 e punto 3 finché non è stato raggiunto il numero di cluster desiderato.



Ci sono differenti modi di scegliere quale cluster selezionare. Ad esempio è possibile selezionare il cluster con il più alto numero di documenti, quello con il più alto grado di similarità o un criterio basato su entrambe; in ogni caso come descritto in [11] la differenza tra i diversi metodi è piccola e, per tale motivo, la nostra implementazione utilizza la prima metodologia.

1.4.2.1 Iterazioni

Per quanto riguarda il numero di iterazioni eseguite per ogni bisezione (punto 3), al fine di equiparare il bisection e il KMeans, è stata utilizzata la seguente equazione [9]:

$$\text{KMeansIteration} = \log_2(\text{cluster}) * \text{BKMeansIteration}$$

Ad esempio se il numero di cluster desiderati sono 9 e le iterazioni del KMeans eseguite sono 6 abbiamo $\text{BKMeansIteration} = 6/3 = 2$.

2. Sequenziale

2.1 Introduzione

In questa sezione presenteremo i tempi impiegati dagli algoritmi sequenziali implementati che, successivamente, saranno confrontati con i tempi delle implementazioni parallele. In particolare i dataset sui quali sono stati condotti gli esperimenti sono:

1. Dataset TF - 5,041 GB
2. Dataset TF replicato 9 volte - 45,369 GB

È interessante notare come il Dataset TF replicato 9 volte, come vedremo nel prosieguo della trattazione, sia stato utilizzato al fine di dimostrare come 5,051 GB fossero insufficienti per incrementare lo speedup tra 16 e 31 macchine e solo un incremento di dati consentisse agli algoritmi di scalare correttamente.

2.2 Dataset

Il dataset utilizzato è stato generato partendo dal SequenceFile Hadoop. Da questo è stato creato un file testuale, le cui righe hanno questa struttura:

nome_documento;parola:valore;parola:valore;

Ovviamente ogni riga è un documento diverso.

2.3 Configurazione Hardware

La macchina su cui è stato eseguito l'esperimento sequenziale presenta le seguenti caratteristiche:

- CPU: Intel Celeron G530 Dual Core, 2,4Ghz;
- Memory RAM: 4096Mb DDR a 667Mhz;
- Hard disk: 500Gb a 5400rpm;
- Ethernet Network adapter a 1Gbps;
- Windows 7 Enterprise 64 bit.

Sulla macchina era inoltre installata una virtual machine VMWare, così configurata:

- 2 Vcore;
- Memoria RAM da 3072Mb di;
- Hard disk da 120Gb
- Sistema operativo: Canonical Ubuntu 14.10 LTS 64 bit.
- Java version "1.8.0 _20", con Java HotSpot (TM) 64 - Bit Server VM (build 25.20 - b23 , mixed mode)

Il codice Java del sequenziale è stato eseguito sulla VM.

2.4 KMeans

2.4.1 Note implementative

Durante gli esperimenti ci si è resi conto che, dopo la prima iterazione di KMeans, il tempo di esecuzione cresceva esponenzialmente all'aumentare delle iterazioni dell'algoritmo. Questo fenomeno era dovuto al fatto che ogni centroide generato dopo la prima iterazione era l'unione di tutte le parole dei documenti ad esso vicino. Quindi, i centroidi, sempre più grandi (in media 250.000 parole nell'ultima iterazione dell'algoritmo) allungavano di molto il tempo di calcolo al crescere del numero di iterazioni.

Per ovviare a questo problema quindi, al termine di ogni iterazione del KMeans, è stato scelto di limitare ad un numero pari a 200 il numero di parole appartenenti ad ogni centroide (considerando solo i termini più ricorrenti) così da evitare fenomeni gravitazionali¹ (in modo tale da migliorare la classificazione) e limitare il tempo di computazione. È stato scelto il numero di 200 features, dopo vari esperimenti effettuati.

A tal proposito, all'unico scopo di comprendere il tempo necessario per completare l'esecuzione dell'algoritmo senza limitare il numero di centroidi, su un MacBook Pro (processore 2,4 GHz Intel Core i5 e 8 GB RAM 1600 MHz DDR3) l'esecuzione del KMeans (9 cluster e 6 iterazioni) su un dataset da 1.5GB ha impiegato:

```
real 1045m22.882s (più di 17 ore)
user 919m9.887s.
```

Tale tempo sarebbe sicuramente aumentato sul PC utilizzato per condurre gli esperimenti in laboratorio e diventato improponibile sui dataset da 5 GB e 45 GB utilizzati per confrontare l'implementazione sequenziale con quella distribuita.

2.4.2 Tempi

Riportiamo di seguito i tempi dell'implementazione sequenziale del KMeans (Algoritmo 1). Da notare come i centroidi siano salvati in memoria centrale e le features dei documenti siano lette dal disco; trovato il cluster più vicino quindi, il rispettivo centroide viene modificato “online”. Infine, letti tutti i documenti sono calcolate le medie per tutti i centroidi.

Dataset TF – 5,041 GB (200 parole per centroide, 9 cluster, 6 iterazioni):

```
real 98m
user 79m
```

¹ I cluster molto grandi tendono ad attrarre al loro interno tutti gli elementi da classificare [8]

Dataset TF - 5,041 GB (400 parole per centroide, 9 cluster, 6 iterazioni):

real 115m

user 93m

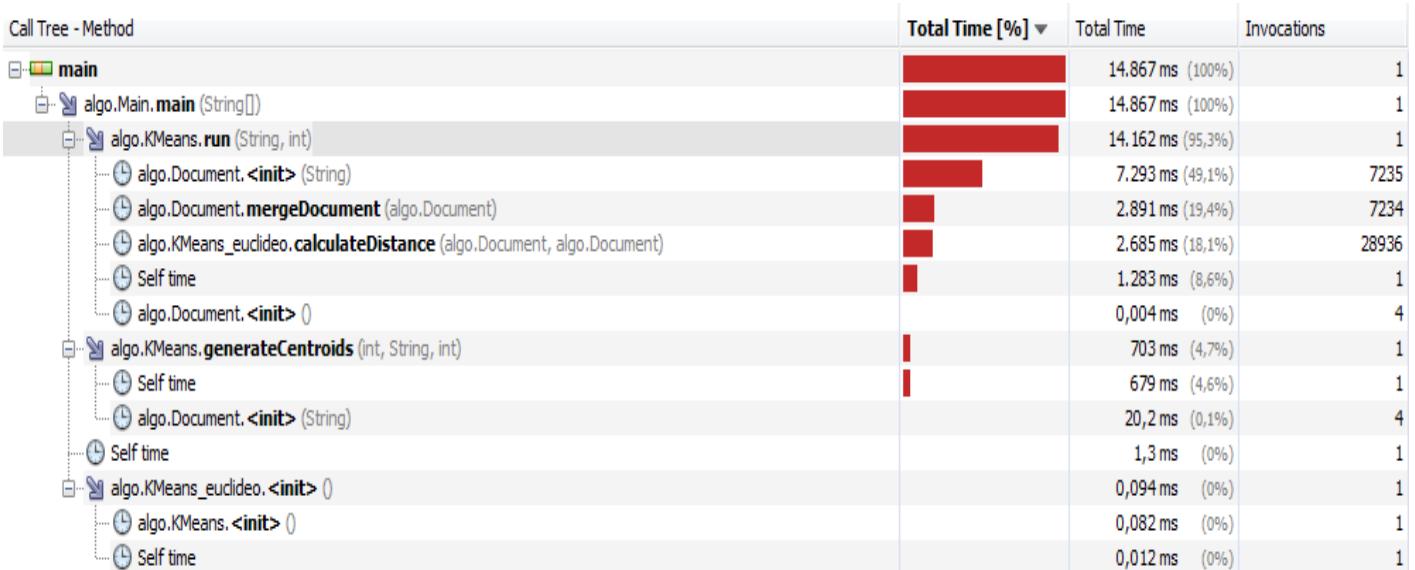
Dataset TF replicato 9 volte – 45,369 GB (200 parole per centroide, 2 cluster, 2 iterazioni):

real 238m

user 163m

2.4.3 Profile

CPU: dall'immagine sottostante si evince come la maggior parte del tempo di CPU sia utilizzato dal **costruttore** di **Document**. Questa classe è utilizzata per effettuare il parsing da una semplice stringa a un'hash-map. Seguono i metodi **merge**, utilizzato per unire i documenti e calcolare i nuovi centroidi, e **calculateDistance**, utilizzato per calcolare la distanza tra il documento ed il centroide. Quest'ultimi sono invocati ad ogni lettura di una linea di testo dal file di input.



Memoria: gli oggetti più allocati sono array di char per mantenere in memoria i termini di ogni documento, i double utilizzati per memorizzare il TF o il TF-IDF dei termini.

Class Name - Allocated Objects	Bytes Allocated [%] ▾	Bytes Allocated	Objects Allocated
char[]		527.926.608 B (41,5%)	37.852.450 (13,6%)
java.lang.Double		210.730.416 B (16,6%)	83.418.873 (30%)
int[]		126.320.024 B (9,9%)	35.534.793 (12,8%)
sun.misc.FDBigInteger		119.693.696 B (9,4%)	35.534.775 (12,8%)
java.lang.String		71.647.344 B (5,6%)	28.358.034 (10,2%)
java.util.ArrayList\$SubList		39.803.840 B (3,1%)	9.451.722 (3,4%)
java.util.ArrayList\$SubList\$1		39.780.680 B (3,1%)	9.451.722 (3,4%)
java.util.HashMap\$Node		33.658.944 B (2,6%)	9.992.477 (3,6%)
sun.misc.FloatingDecimal\$ASCIIToBinaryBuffer		31.807.264 B (2,5%)	9.444.483 (3,4%)
java.lang.String[]		27.739.696 B (2,2%)	9.451.725 (3,4%)
java.util.ArrayList		23.876.064 B (1,9%)	9.451.728 (3,4%)
java.util.HashMap\$Node[]		19.205.136 B (1,5%)	58.666 (0%)
java.util.HashMap\$EntryIterator		275.040 B (0%)	65.106 (0%)
java.nio.HeapCharBuffer		235.296 B (0%)	46.672 (0%)
java.lang.Object[]		89.480 B (0%)	12.092 (0%)
java.util.HashMap		36.576 B (0%)	7.248 (0%)
java.lang.StringBuffer		24.936 B (0%)	9.845 (0%)
algo.Document		18.096 B (0%)	7.242 (0%)
java.util.HashMap\$EntrySet		12.224 B (0%)	7.238 (0%)
java.lang.Integer		12.000 B (0%)	7.032 (0%)
byte[]		3.184 B (0%)	30 (0%)

2.5 Bisecting KMeans

2.5.1 Note implementative

Come si evince dall'algoritmo presentato nel paragrafo 1.4.2, ogni iterazione del bisecting KMeans (vedi punto 3 Algoritmo 2) richiede un tempo lineare sul numero di documenti presenti nel cluster che si intende dividere. Ovviamente non è stato possibile mantenere in memoria la suddivisione dei documenti per cluster (ciò sarebbe stato possibile solamente avendo a disposizione almeno 8 GB di RAM) pertanto si è deciso, in un primo momento, di procedere ad un'implementazione disk-based dell'algoritmo (Algoritmo 3).

Algoritmo 3.

1. Scegliere un cluster da dividere dove ogni cluster è rappresentato da un file (nella prima iterazione vi è un solo file contenente tutti i documenti)
2. Scegliere a caso due documenti dal cluster che si intende dividere
3. Eseguire il KMeans #iterazioni volte sul cluster individuato al punto 1 mantenendo la lista dei documenti appartenente ad ogni cluster in file distinti
4. Eseguire punto 1, punto 2 e punto 3 finché non è stato raggiunto il numero di cluster desiderato.

In un secondo momento, al fine di uniformare l'implementazione sequenziale e Hadoop (che richiede necessariamente la lettura dell'interno input ad ogni iterazione del KMeans) si è deciso di passare da un'implementazione disk-based ad un'implementazione "hadoop-based". Nella fattispecie ogni iterazione del KMeans, invece di richiedere tempo lineare sul numero di documenti presenti nel cluster da dividere, richiede tempo lineare sull'intero dataset.

Algoritmo 4.

1. Scorri l'intero dataset e controlla se l'item appartiene al cluster che si intende dividere
2. Scegliere a caso due documenti dagli item individuati al punto 1
3. Eseguire il KMeans #iterazioni volte sugli item individuato al punto 1
4. Eseguire punto 1, punto 2 e punto 3 finché non è stato raggiunto il numero di cluster desiderato.

Infine, anche per quanto riguarda il bisecting è stato scelto di limitare il numero di parole di ogni centroide a 200.

2.5.2 Tempi

Riportiamo di seguito i tempi dell'implementazione sequenziale del KMeans (Algoritmo 4) che, come evidenziato nel sottoparagrafo “Note implementative”, presenta le stesse peculiarità dell'implementazione Hadoop dell'algoritmo.

Dataset TF - 5,041 GB (200 parole per centroide, 9 cluster, 2 iterazioni per cluster):

real 156m

user 91m

Dataset TF replicato 9 volte - 45,369 GB (200 parole per centroide, 2 cluster, 2 iterazioni per cluster):

real 427m

user 163m

2.5.3 Profile

CPU: come per il KMeans.

Call Tree - Method	Total Time [%]	Total Time	Invocations
main			
org.kmeans.bisec.Main.main (String[])			
org.kmeans.bisec.Bisec.kmeans (int, java.util.ArrayList)			
org.kmeans.bisec.Document.<init> (String, int)	61.579 ms (48,7%)	61.579 ms (48,7%)	65115
org.kmeans.bisec.Document.mergeDocument (org.kmeans.bisec.Document)	16.549 ms (13,1%)	16.549 ms (13,1%)	43158
org.kmeans.distances.CosiceDistance.distance (org.kmeans.bisec.Document, org.kmeans.bisec.Document)	12.262 ms (9,7%)	12.262 ms (9,7%)	194841
Self time	10.147 ms (8%)	10.147 ms (8%)	9
org.kmeans.bisec.Document.normalizzaX (int)	439 ms (0,3%)	439 ms (0,3%)	18
org.kmeans.bisec.Document.<init> ()	0,019 ms (0%)	0,019 ms (0%)	18
org.kmeans.bisec.Bisec.getRandomCenters (int, int)	25.447 ms (20,1%)	25.447 ms (20,1%)	3
org.kmeans.bisec.Document.<init> (String, int)	22.230 ms (17,6%)	22.230 ms (17,6%)	17174
org.kmeans.distances.CosiceDistance.distance (org.kmeans.bisec.Document, org.kmeans.bisec.Document)	1.625 ms (1,3%)	1.625 ms (1,3%)	28711
Self time	1.576 ms (1,2%)	1.576 ms (1,2%)	3
Self time	22,2 ms (0%)	22,2 ms (0%)	1
org.kmeans.bisec.Bisec.maxCluster ()	0,030 ms (0%)	0,030 ms (0%)	3
Self time	0,029 ms (0%)	0,029 ms (0%)	3
org.kmeans.bisec.Document.getSize ()	0,001 ms (0%)	0,001 ms (0%)	5
org.kmeans.bisec.Bisec.<init> (String, org.kmeans.distances.DistanceStrategy, int)	0,011 ms (0%)	0,011 ms (0%)	1
org.kmeans.distances.CosiceDistance.<init> ()	0,005 ms (0%)	0,005 ms (0%)	1
org.kmeans.bisec.Document.getSize ()	0,000 ms (0%)	0,000 ms (0%)	2

Memoria: come per il KMeans.

Class Name - Allocated Objects	Bytes Allocated [%] ▾	Bytes Allocated	Objects Allocated
char[]		4.768.930.504 B (40%)	410.685.987 (15,4%)
java.lang.Double		1.408.797.432 B (11,8%)	557.666.519 (20,9%)
int[]		1.370.997.976 B (11,5%)	385.644.306 (14,4%)
sun.misc.FDBigInteger		1.299.036.800 B (10,9%)	385.644.211 (14,4%)
java.lang.String		777.576.264 B (6,5%)	307.792.155 (11,5%)
java.util.ArrayList\$SubList\$1		431.990.320 B (3,6%)	102.598.853 (3,8%)
java.util.ArrayList\$SubList		431.942.080 B (3,6%)	102.598.853 (3,8%)
java.util.HashMap\$Node		359.016.032 B (3%)	106.584.970 (4%)
sun.misc.FloatingDecimal\$ASCIIToBinaryBuffer		345.327.168 B (2,9%)	102.520.152 (3,8%)
java.lang.String[]		302.753.944 B (2,5%)	102.598.864 (3,8%)
java.util.ArrayList		259.167.840 B (2,2%)	102.598.880 (3,8%)
java.util.HashMap\$Node[]		175.085.968 B (1,5%)	637.127 (0%)
java.util.HashMap\$EntryIterator		2.014.840 B (0%)	478.671 (0%)
java.nio.HeapCharBuffer		1.806.624 B (0%)	357.744 (0%)
java.util.HashMap		398.400 B (0%)	78.742 (0%)
java.util.ArrayList\$Itr		320.704 B (0%)	95.030 (0%)
org.kmeans.bisec.Document		198.528 B (0%)	78.719 (0%)
java.lang.StringBuffer		184.656 B (0%)	73.127 (0%)
java.util.TreeMap\$Entry		141.040 B (0%)	33.666 (0%)
java.util.HashMap\$EntrySet		124.400 B (0%)	73.990 (0%)
java.lang.Object[]		96.848 B (0%)	11.204 (0%)
java.util.AbstractMap\$SimpleImmutableEntry		84.648 B (0%)	33.666 (0%)
java.lang.Integer		67.824 B (0%)	40.499 (0%)
byte[]		10.608 B (0%)	48 (0%)
java.util.HashMap\$TreeNode		8.736 B (0%)	1.436 (0%)

2.6 Note

Il test su 45,369 GB è stato eseguito al solo scopo di dimostrare l'aumento dello speedup tra 16 e 31 macchine. Pertanto sia per quanto riguarda il KMeans che il bisecting le iterazioni eseguite sono le medesime. L'utilizzo di 9 cluster e 2 iterazioni, in particolare, avrebbe richiesto troppo tempo in quanto il dataset di 45,369 GB necessitava di 26 letture (2 per ognuna delle 9 bisezioni del bisecting più 8 per la scelta dei centroidi).

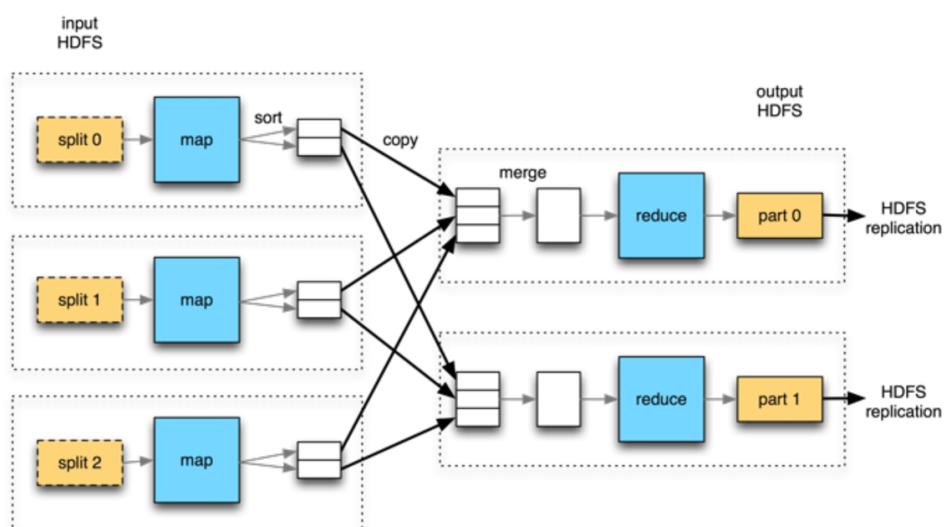
3. Hadoop

3.1 Introduzione all'architettura

Hadoop è un framework sviluppato dall'Apache Software Foundation, nato come versione open source del progetto map-reduce di Google. Consente l'elaborazione distribuita di grandi insiemi di dati attraverso gruppi di computer: è stato progettato infatti per scalare da singoli server a migliaia di macchine, ognuna delle quali offre calcolo e storage locale. Piuttosto che fare affidamento su hardware per fornire alta disponibilità, la libreria stessa è progettato per rilevare e gestire gli errori a livello di applicazione, in modo da fornire un servizio altamente disponibile su di un cluster di computer, ciascuno dei quali può essere soggetto a guasti.

Il progetto prevede questi moduli:

- **Hadoop Common**: programmi di utilità che supportano gli altri moduli Hadoop.
- **Hadoop Distributed File System (HDFS)**: un file system distribuito che fornisce accesso ad alto throughput ai dati
- **Hadoop YARN**: un framework per la pianificazione del lavoro e la gestione delle risorse del cluster.
- **Hadoop MapReduce**: un sistema basato su YARN per l'elaborazione distribuita di grandi insiemi di dati che supporta il paradigma map-reduce e permette di eseguire calcolo distribuito in maniera semi-trasparente: il programmatore, per ogni Job, può definire la funzione *di map*, *combiner* e *reduce*. Poi sarà l'infrastruttura che, alla luce di vari fattori, tra cui la distribuzione dei dati sul HDFS, deciderà come distribuire il carico di lavoro e risolvere eventuali failures.



3.1.1 Distributed Cache

La distributed cache è una funzionalità di Hadoop che permette di copiare un file su tutte le macchine del cluster così da poterlo usare come se fosse un file in locale.

Per implementare il KMeans/Bisecting in Hadoop si deve tener conto del modo in cui i dati sono distribuiti sui nodi, infatti, ogni macchina mantiene una porzione del dataset totale e deve poter calcolare qual è il cluster di ogni documento appartenente al blocco che memorizza. Per tale motivo, al fine di mantenere la medesima lista di centroidi (aggiornata ad ogni iterazione dell'algoritmo) su ogni nodo del cluster, è stato scelto di utilizzare la Distributed Cache in modo tale da conservare i medesimi dati di sintesi su tutte le macchine.

3.2 Formato dei dati in input

L'output prodotto dal sottogruppo 1 su circa 140 GB scaricati dal sito web *arxiv* [2] è stato utilizzato come input per gli algoritmi implementati. In particolare quest'ultimo è rappresentato come due SequenceFile² (ognuno di circa 5 GB) contenenti, per ogni PDF una coppia <TextWritable PDFName, MapWritable PDFValue> dove TextWritable e MapWritable sono particolari strutture dati Hadoop che incapsulano rispettivamente una stringa e una mappa.

Solo replicando 9 volte il dataset iniziale, la differenza nei tempi utilizzando 16 e 31 macchine ha prodotto lo scaling sperato.

3.3 La classe Document

La classe **Document** è la classe ausiliaria più importante del codice Hadoop realizzato. Come suggerisce il nome, quest'ultima contiene il documento letto dal HDFS: ogni parola (e il rispettivo valore) è memorizzato in una struttura MapWritable. La classe è inoltre utilizzata per mantenere i dati di sintesi di ogni centroide; è infatti presente una variabile count che mantiene il conteggio del numero dei documenti aggregati, permettendo la computazione della media nella fase di reduce.

3.4 KMeans

Per quanto riguarda il KMeans sono stati implementati due Job Hadoop:

² Un SequenceFile è un flat file contenente coppie key/value binarie [3] ampiamente utilizzato in Hadoop sia per l'input che per l'output ed è consigliato per evitare il problema dei piccoli file [4]. Al suo interno sono ammesse diverse strutture dati che, al fine di consentire il transito delle informazioni sulla rete, forniscono dei metodi per serializzare, deserializzare i dati memorizzati al loro interno.

1. Scelta dei centroidi casuali (Job1):
 - la fase di Map seleziona a caso dei documenti con una probabilità molto alta.
 - la fase di Combine screma ulteriormente la lista di documenti selezionati nella fase di Map (la probabilità di selezionare un documento decresce).
 - la fase di Reduce seleziona esattamente un numero di documenti pari al numero di cluster che si intende produrre in output.
2. Calcolo centroidi (Job2):
 - la fase di Map calcola qual è il cluster più vicino al documento in input, data la lista dei centroidi memorizzati nella distributed cache.
 - la fase di Combine calcola uno pseudo-centroide: dato un cluster K e i dati presenti su ogni singola macchina si aggregano tutti i documenti appartenenti ad ogni centroide.
 - la fase di Reduce combina i risultati degli pseudo-centroidi così da calcolare i nuovi centroidi.

Alla fine di entrambi i Job viene generato un file testuale da ogni reducer. Questi file vengono aggregati per generare un nuovo file, che contiene i centroidi calcolati.

Alla luce di quanto detto, per quanto concerne il KMeans il Job1 è eseguito una sola volta (prima fase dell'algoritmo) mentre, per quanto riguarda il Job2, quest'ultimo viene reiterato tante volte quanto sono le iterazioni scelte.

Il codice del KMeans è stato scritto da zero a partire dallo pseudo codice riportato di seguito:

3.4.1 Pseudo codice

3.4.1.1 Mapper

Mapper(raw <K,V>):

```

indexVicino = -1; distMin = inf;
for (Centroids as ind => centroide) {
    dist = distanza(raw, centroide)
    if (dist < lenMin) {
        indexVicino = ind
        distMin = dist
    }
}
send(indexVicino, raw.value())

```

3.4.1.2 Combiner

Reducer(List<K,V> input):

```
Document aggreg = new Document(K);
for (Document raw in input){
    aggreg.add(raw)
}
send(aggreg.key(); aggreg.value())
```

3.4.1.3 Reducer

Reducer(List<K,V> input):

```
Document aggreg = new Document(K);
for (Document raw in input) {
    Aggreg.add(raw)
}
aggreg.media()
send(aggreg.key(); aggreg.value())
```

3.5 Bisecting KMeans

Per il Bisecting KMean, l'argoritmo Hadoop implementato presenta le medesime peculiarità dell'Algoritmo 4 (versione sequenziale) presentato nel paragrafo 2.5 :

1. Si sceglie il cluster che ha più documenti.
2. Dato il cluster scelto, si selezionano due documenti a caso come centroidi iniziali.
3. Viene interato il KMeans su questo cluster.
4. Si ritorna al punto 1, finchè non sono stati computati K cluster.

Anche il Bisecting KMeans è stato scritto da zero. La realizzazione del codice non è stata tuttavia immediata poichè ci siamo imbattuti in diverse problematiche che riassumiamo nel paragrafo successivo.

3.5.1 Problematiche

Durante la stesura del codice del Bisecting KMeans sono state riscontrate alcune difficoltà, elencate di seguito:

- **Selezionare i documenti di un determinato cluster:** alla luce di quanto descritto nella sezione “Bisecting KMeans”, l’algoritmo del bisecting, scelto un cluster da dividere, effettua una bisezione di quest’ultimo mantenendo le partizioni individuate separatamente. Tuttavia in ambiente Hadoop non è possibile leggere solamente una porzione dell’input, pertanto ad ogni iterazione del bisecting è stato necessario leggere l’intero dataset e, per ogni documento è stato necessario verificare l’appartenenza del documento al cluster che si intendeva dividere. Quindi, nella funzione Map è stato inserito un filtro che, a partire del file dei centroidi salvato sulla Distributed Cache calcola l’appartenenza del documento al cluster desiderato.
- **Trovare il cluster con più documenti:** la metrica scelta per dividere un cluster, come evidenziato nella sezione “Bisecting KMeans”, è rappresentata dal cluster con più documenti. Al fine di mantenere tale informazione, a partire dal numero di documenti di ogni cluster (fase di reduce), è stato necessario anteporre al file dei centroidi conservati nella Distributed Cache il numero totale di documenti presenti in ogni cluster. Tale numero è stato quindi utilizzato per individuare quale cluster dividere.

3.5.2 Pseudo codice

Per creare il Bisecting KMeans è stato necessario modificare la classe del Mapper in modo tale da considerare solamente i documenti appartenenti al cluster più numeroso. Inoltre è stato necessario modificare il main del programma così da aggiornare il file che contiene i

centroidi: eliminare dal file il centroide del cluster diviso ed inserire i due nuovi centroidi individuati.

3.5.2.1 Mapper

Mapper(raw <K,V>):

```
indexVicino = -1; distMin = inf;
for Centroids as ind =>centroide {
    dist = distanza(raw, centroide)
    if dist < lenMin {
        indexVicino = ind
        distMin = dist
    }
}
if (indexVicino == indiceClusterDaDividere ) {
    //KMeans sui due cluster random
    indexCentroide = -1; distMin = inf;
    for (RndCentroids as ind => centroide) {
        dist = distanza(raw, centroide)
        if dist < lenMin {
            indexCentroide = ind
            distMin = dist
        }
    }
    send(indexCentroide, raw)
}
```

4. Test e benchmarking su Hadoop

4.1 Introduzione

I test effettuati su Hadoop sono stati eseguiti inizialmente sul dataset TF da 5,041 GB, che è stato utilizzato come input sia dagli algoritmi KMeans che Bisecting KMeans. Il test avrebbe dovuto dimostrare l'aumento dello speedup confrontando i tempi dell'implementazione sequenziale con i risultati ottenuti dall'algoritmo su 8, 16 e 31 macchine (non siamo riusciti a far partire un'altra macchina).

Purtroppo, i risultati dell'esecuzione su 16 e 31 macchine non hanno fornito i risultati sperati: il tempo di esecuzione è stato il medesimo. Pertanto si è deciso, in un secondo momento, di aumentare la taglia del dataset che da 5,041 GB è passata a 45,369 GB (non avendo ulteriori PDF da analizzare si è optato per la replicazione del dataset iniziale).

4.2 Configurazione Hadoop

Le macchine sulle quali sono stati eseguiti i test presentano le seguenti caratteristiche:

- CPU: Intel Celeron G530 Dual Core, 2,4Ghz;
- Memory RAM: 4096Mb DDR a 667Mhz;
- Hard disk: 500Gb a 5400rpm;
- Ethernet Network adapter a 1Gbps;
- Windows 7 Enterprise 64 bit.

Le macchine del cluster sono inoltre collegate tramite switch a 100Mbps e su ognuna è installata una virtual machine VMWare, così configurata:

- 2 Vcore;
- Memoria RAM da 3072Mb di;
- Hard disk da 120Gb
- Sistema operativo: Canonical Ubuntu 14.10 LTS 64 bit.
- Java version "1.8.0 _20", con Java HotSpot (TM) 64 - Bit Server VM (build 25.20 - b23 , mixed mode)
- Hadoop 2.5.1

È inoltre mostrata di seguito la configurazione Hadoop utilizzata. Si è scelto nella fattispecie di utilizzare un solo container per macchina poiché, in seguito ai primi test effettuati (che utilizzavano due container, ognuno con 1.5GB di RAM), sono stati riscontrati problemi in quanto ogni container saturava l'intera memoria a disposizione causando eccezioni di Heap Overflow.

Yarn-site.xml

Proprietà	Valore	Descrizione
yarn.nodemanager.resource.memory-mb	2560	Amount of physical memory, in MB, that can be allocated for containers

mapred-site.xml

Proprietà	Valore	Descrizione
mapreduce.reduce.memory.mb	2560	Larger resource limit for reduces
mapreduce.map.memory.mb	2560	Larger resource limit for maps
mapreduce.map.java.opts	-Xmx 2048M	Larger heap-size for child jvms of maps
mapreduce.redece.java.opts	-Xmx 2048M	Larger heap-size for child jvms of reduces

hdfs-site.xml

Proprietà	Valore	Descrizione
dfs.replication	2	Default block replication. The actual number of replications can be specified when the file is created. The default is used if replication is not specified in create time
dfs.blocksize	128M	The default block size for new files, in bytes. You can use the following suffix (case insensitive): k(kilo), m(mega), g(giga), t(tera), p(peta), e(exa) to specify the size or provide complete size in bytes

4.3 KMeans - dataset da 5,041 GB

4.3.1 Test su 8 macchine (9 centroidi, 6 iterazioni)

Il tempo di esecuzione dell'esperimento è stato:

54m24.227s

Sono stati eseguiti in totale 7 job, il primo che genera i centroidi casuali (durata: 7 minuti e 19 secondi) e gli altri che effettuano la medesima iterazione del KMeans (durata: 14 minuti e 20 secondi il secondo job, 5 il terzo e i rimanenti 6 minuti).

In sostanza tutti i job del KMeans durano all'incirca lo stesso tempo, ad eccezione del primo che presenta dei centroidi con più termini; infatti nella prima iterazione i centroidi hanno una lunghezza variabile e sono molto lunghi (in genere 1000 parole) mentre, nelle altre iterazioni sono esattamente 200 quindi, la prima iterazione del KMeans consuma molto più tempo delle successive.

4.3.2 Test su 16 macchine (9 centroidi, 6 iterazioni)

Il tempo di esecuzione dell'esperimento è stato:

25m17.349s

quindi lo speedup è pari a 2,16. Tale valore è in linea con il risultato atteso, infatti, raddoppiando il numero di macchine vi è un incremento delle prestazioni pari al doppio.

4.3.3 Test su 31 macchine (9 centroidi, 6 iterazioni)

Il tempo di esecuzione dell'esperimento è stato:

28m5.608s

In questo caso il tempo di esecuzione è stato peggiore di quello ottenuto su 16 macchine quindi, l'esperimento non ha prodotto i risultati attesi.

Si è deciso di **aumentare il dataset in input**, così da utilizzare a pieno tutte le macchine. Infatti con questo dataset, vi sono circa 40 blocchi (da 128MB) distribuiti sulle 31 macchine, mentre portano il dataset a 45GB si arriva ad avere 360 blocchi (da 128MB).

È stato anche effettuato un **test utilizzando blocchi da 64 MB**, quindi il dataset è stato diviso in 80 blocchi, in modo da aumentare i blocchi per macchina ma, anche in questo caso, il risultato ottenuto è peggiore rispetto all'esecuzione dell'algoritmo su 16 macchine. Si è provato inoltre ad incrementare il numero di cluster pensando che i risultati negativi fossero da imputare al collo di bottiglia creato dai nove reduce creati (ricordiamo al tal proposito che per ogni cluster al massimo era possibile creare un reduce) ma, anche in questo caso, i risultati ottenuti non sono stati quelli sperati.

4.4 KMeans - dataset da 45,369 GB

Alla luce dei pessimi risultati ottenuti confrontando l'esecuzione del KMeans su 16 e 31 macchine è stato scelto di replicare il dataset di input fino a raggiungere 45,369 GB di dati. Anche in questo caso sono stati utilizzati 9 centroidi ed eseguite sei iterazioni dell'algoritmo.

4.4.1 Test su 16 macchine (9 centroidi, 6 iterazioni)

Il tempo di esecuzione dell'esperimento è stato:

195m26.908s

Il primo Job che sceglie i centroidi casuali ha impiegato 11minuti, il primo job di KMeans eans 48 minuti, gli altri job di KMeans variavano tra i 25 e i 30 minuti.

4.4.2 Test su 31 macchine (9 centroidi, 6 iterazioni)

Il tempo di esecuzione dell'esperimento è stato:

115m15.270s

Il primo Job che sceglie i centroidi ha impiegato 8 minuti, il primo job di KMeans 21 minuti, gli altri job di KMeans variavano tra i 13 e i 15 minuti.

In questo caso lo speedup è di 1,7 quindi vi è stata una diminuzione del tempo di esecuzione passando da 16 a 31 macchine quindi, il risultato ottenuto è in linea con il risultato atteso.

4.4.3 Ulteriori test KMeans (2centroidi, 2 iterazioni)

Durante la trattazione è stato più volte sottolineato come il KMeans sia un algoritmo iterativo per cui, ad eccezione della fase in cui vengono scelti i documenti in maniera casuale (prima fase dell'algoritmo), nelle altre fasi il medesimo frammento di codice viene iterato più volte. A tal proposito, al fine di confrontare il tempo di esecuzione del sequenziale con l'implementazione Hadoop e considerato che in tale paragrafo non siamo interessati a testare la bontà di classificazione dell'algoritmo, si è scelto di eseguire il KMeans su Hadoop con due centroidi e due iterazioni in modo tale da non eseguire sul sequenziale un esperimento che probabilmente sarebbe durato diversi giorni.

I risultati ottenuto su 8, 16 e 31 macchine sono rispettivamente:

8 macchine	109m19.274s
16 macchine	52m12.180s
31 macchine	29m4.945s

4.5 BKMeans - dataset da 5,041 GB

Anche per quanto riguarda il bisecting sono stati eseguiti in primo luogo gli esperimenti sul dataset da 5,041 GB. Lo speedup tra 16 e 31 macchine anche in questo caso non ha prodotto i risultati attesi pertanto è stato necessario replicare il dataset al fine di dimostrare l'aumento di prestazioni tra le due esecuzioni.

4.5.1 Test su 8 macchine (9 centroidi, 2 iterazioni)

109m51.021s

4.5.2 Test su 16 macchine (9 centroidi, 2 iterazioni)

56m50.030s

4.5.3 Test su 31 macchine (9 centroidi, 2 iterazioni)

55m54.020s

4.5.4 Test su 31 macchine - blocchi da 64 MB (9 centroidi, 2 iterazioni)

55m53.909s

4.6 BKMeans - dataset da 45,369 GB

Riportiamo di seguito i risultati ottenuti eseguendo il bisecting sul dataset da 45,369. Anche in questo caso, l'algoritmo è stato eseguito su due cluster e le iterazioni eseguite sono 2. Come evidenziato nella sezione “Note” del paragrafo “Sequential implementation” l'esecuzione dell'algoritmo con 9 centroidi e 2 iterazioni avrebbe richiesto la lettura di 45,369 GB di dati ben 26 volte; pertanto, l'implementazione sequenziale avrebbe richiesto troppo tempo per completare l'intero task.

4.6.1 Test su 16 macchine (2 centroidi, 2 iterazioni)

43m56.636s

Sono stati eseguiti 3 Job: uno per la scelta dei centroidi a caso della durata di 12 minuti più due di KMeans dalla durata di circa 15 minuti ciascuno.

4.6.2 Test su 31 macchine (2 centroidi, 2 iterazioni)

28m4.936s

Sono stati eseguiti 3 Job: uno per la scelta dei centroidi a caso di 8 minuti e due di KMeans dalla durata di circa 9.30 minuti ciascuno.

4.7 SpeedUp e efficienza

Di seguito viene calcolato lo **SpeedUp** dividendo il tempo di computazione ottenuto

dall'implementazione sequenziale per il tempo ottenuto con la computazione distribuita. Formalmente, la metrica utilizzata per quantificare l'entità del miglioramento tra l'implementazione sequenziale e quella parallela è quella seguente [12]:

$$S = \frac{T_{\text{old}}}{T_{\text{new}}}$$

dove:

- S è il risultato dello SpeedUp
- T_{old} è il tempo di esecuzione del sequenziale
- T_{new} è il tempo di esecuzione della versione parallela, con il miglioramento.

È stata inoltre calcolata l'**efficienza**. La formula utilizzata è la seguente:

$$\text{Efficienza} = \frac{\text{SpeedUp}}{\text{NumeroSlave}}$$

4.7.1 KMeans (2 centroidi, 2 iterazioni) - dataset 45,369 GB

Tempo sequenziale: 238m 27.659s

Numero di slave	Tempo parallelo (minuti)	SpeedUp	SpeedUp incrementale	Efficienza
8	109	2,18		0.27
16	52	4,57	(da 8 a 16) 2.09	0.28
31	29	8,20	(da 16 a 31) 1.79	0.26

4.7.2 BKMeans (2 centroidi, 2 iterazioni) - dataset 45,369 GB

Tempo sequenziale: 427min 35.550s

Numero di slave	Tempo parallelo (minuti)	SpeedUp	SpeedUp incrementale	Efficienza
16	43	9,93		0.62
31	28	15,25	(da 16 a 31) 1.53	0.49

4.8 Conclusioni sui risultati

Come si evince dai tempi non vi è molto speedUp tra il codice sequenziale e quello parallelo anche se vi è un miglioramento incrementando la taglia del dataset in input. Lo speedUp incrementale, invece, con il dataset da 45,369 GB ha prodotto i risultati sperati.

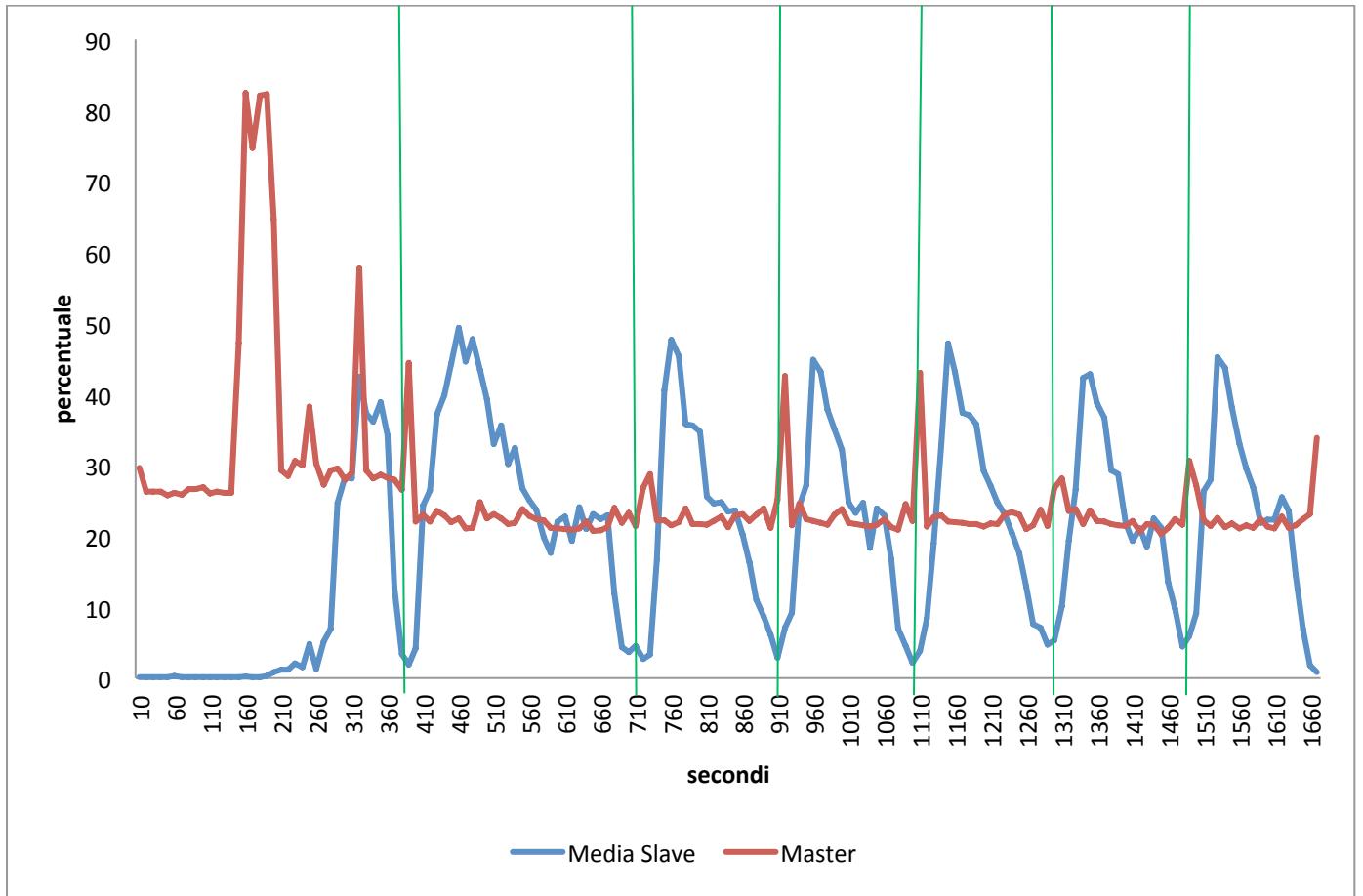
Per quanto riguarda l'efficienza, il KMeans ha prodotti indici piuttosto bassi (in media 0.2) perché l'algoritmo ben si presta ad essere eseguito su una singola macchina. Aumentando la taglia del dataset probabilmente l'implementazioni sequenziale avrebbe mostrato i suoi limiti (saturazione della memoria e limitata capacità in lettura) con il conseguente incremento dell'indice "Efficienza".

Il bisecting, invece, meglio si presta ad essere parallelizzato. Molti dei limiti riscontrati con l'implementazione sequenziale sono risolti utilizzando la distribuzione dei dati sul cluster: il dataset di input richiede molte letture ed Hadoop velocizza molto tale operazione. L'efficienza infatti è in media 0,5.

4.9 Grafici KMeans

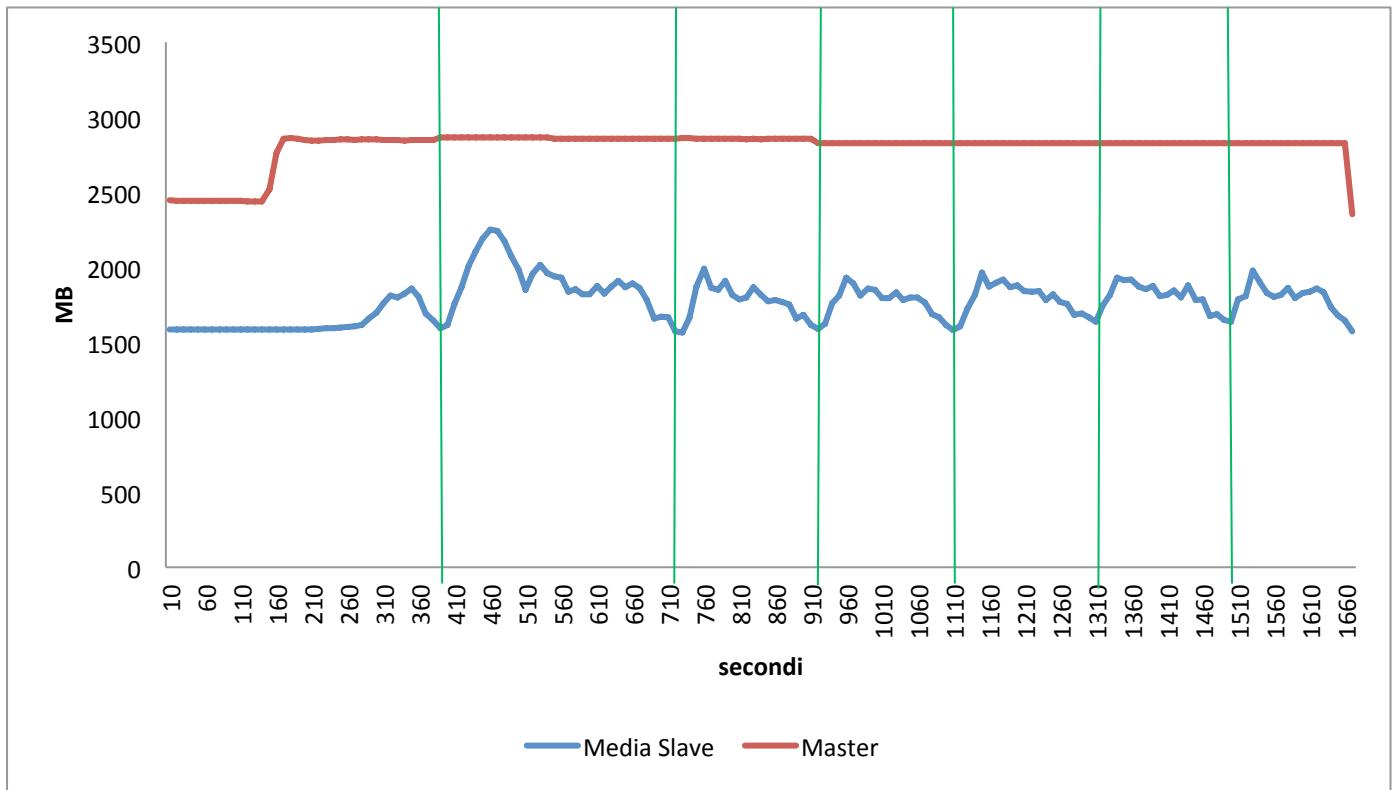
Di seguito vengono mostrati i grafici dell'esperimento su 16 macchine per il KMeans (9 centroidi e 6 iterazioni) sul dataset da 45,369 GB.

4.9.1 CPU



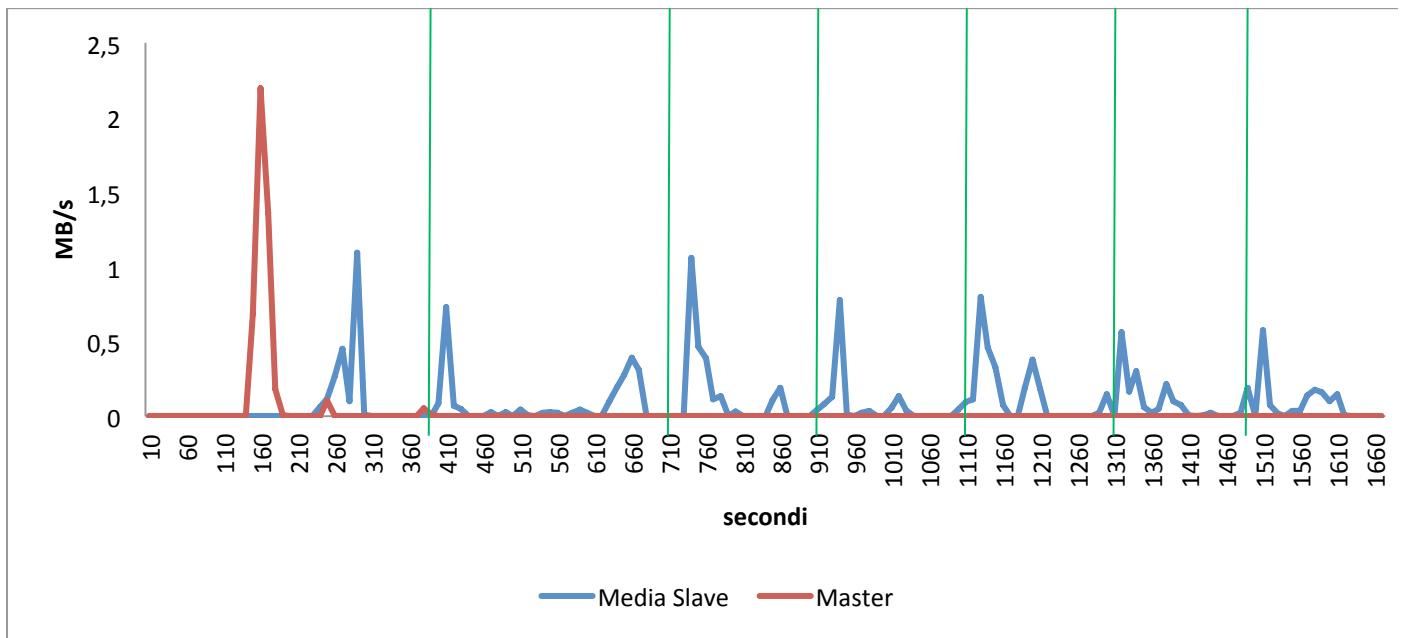
Come si evince dal grafico, l'andamento della curva blu (*media degli slave*) fa individuare i vari job eseguiti. I più lunghi sono il primo: 4 minuti e 30 secondi (scelta dei centroidi casuali) e il secondo: 6 minuti (più lungo a causa della maggiore lunghezza dei centroidi), mentre gli altri job (che sono iterazioni di kmeans) hanno la stessa all'incirca la stessa durata: 3 minuti e 20 secondi.

4.9.2 Memoria



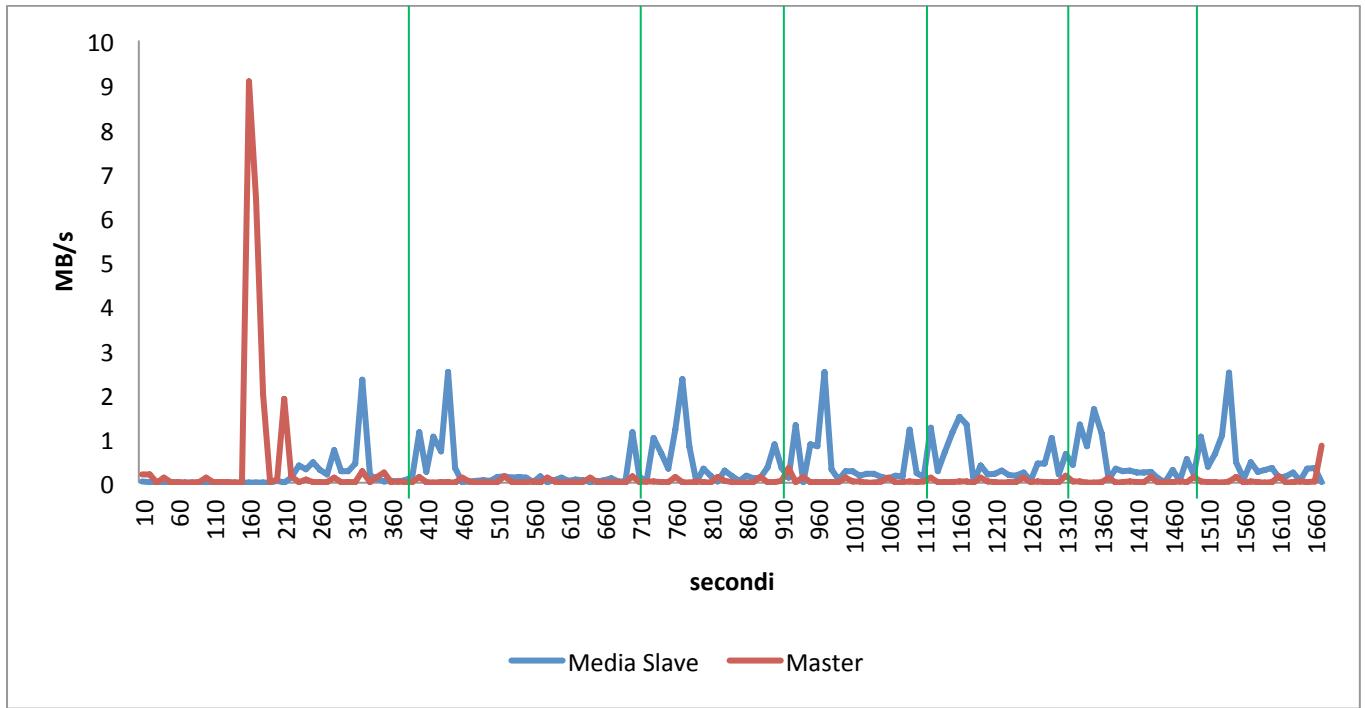
L'uso della memoria da parte degli slave è in media maggiore ai 1.5GB, ed arriva a superare i 2GB. Per questa ragione non è stato possibile avviare due container per macchina in quanto non c'era abbastanza memoria a disposizione.

4.9.3 Lettura da disco



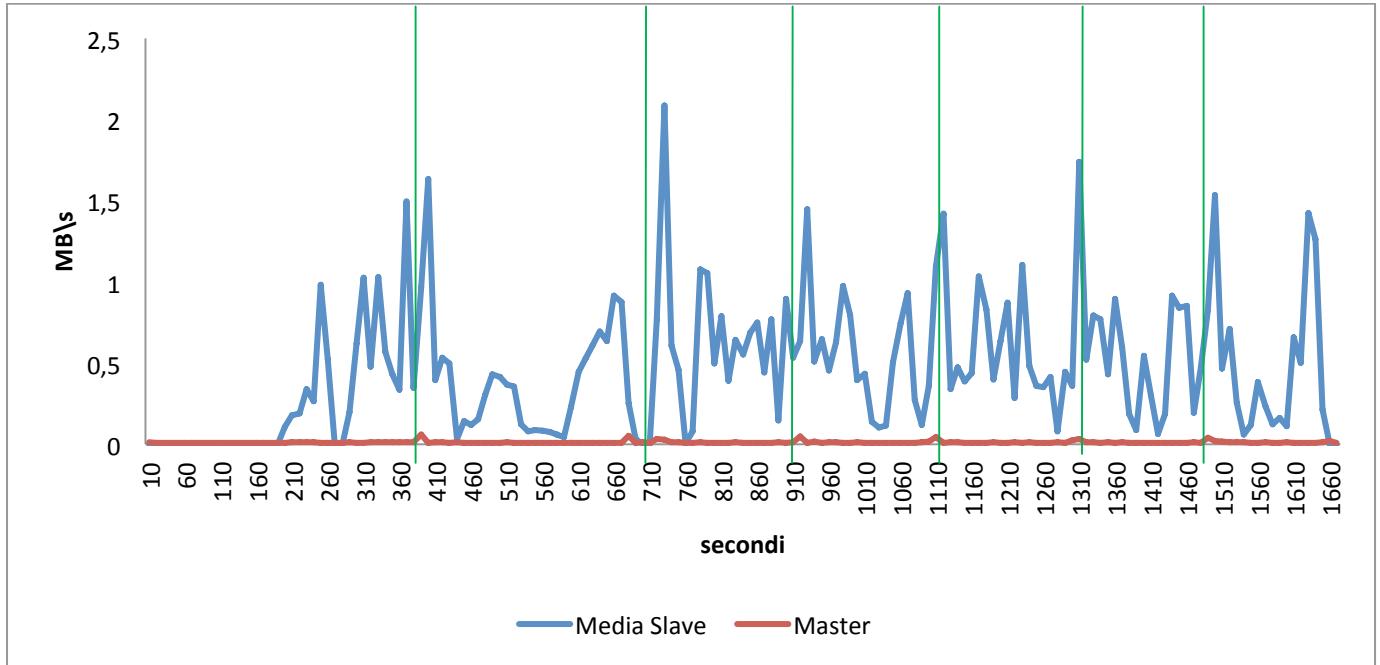
Dal grafico si può vedere come la lettura da disco venga effettuata all'inizio di ogni job, cioè quando il map va a leggere i blocchi dall'HDFS

4.9.4 Scrittura su disco



Come si evince dal grafico il disco viene scritto poche volte, soprattutto durante la fase iniziale dei job, cioè la fase di map.

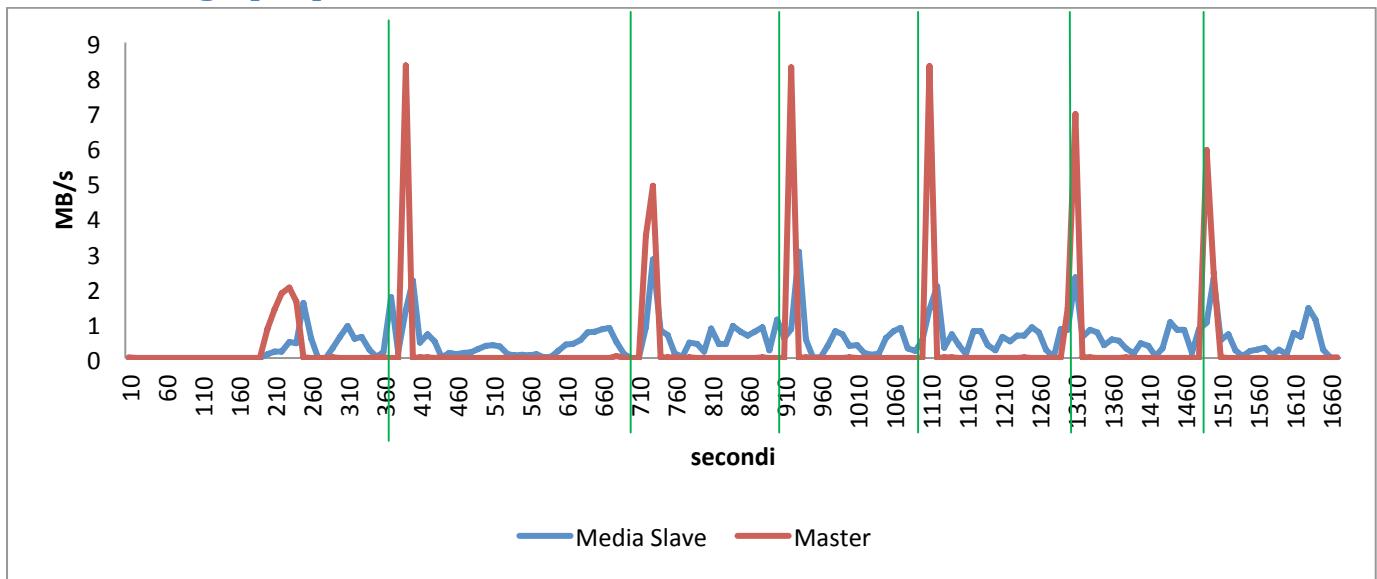
4.9.5 Thoughtput pacchetti di rete ricevuti



Il grafico mostra come gli slave ricevano molti pacchetti durante l'esecuzione dei job: blocchi del sequence file e dati dei pseudo-centroidi calcolati.

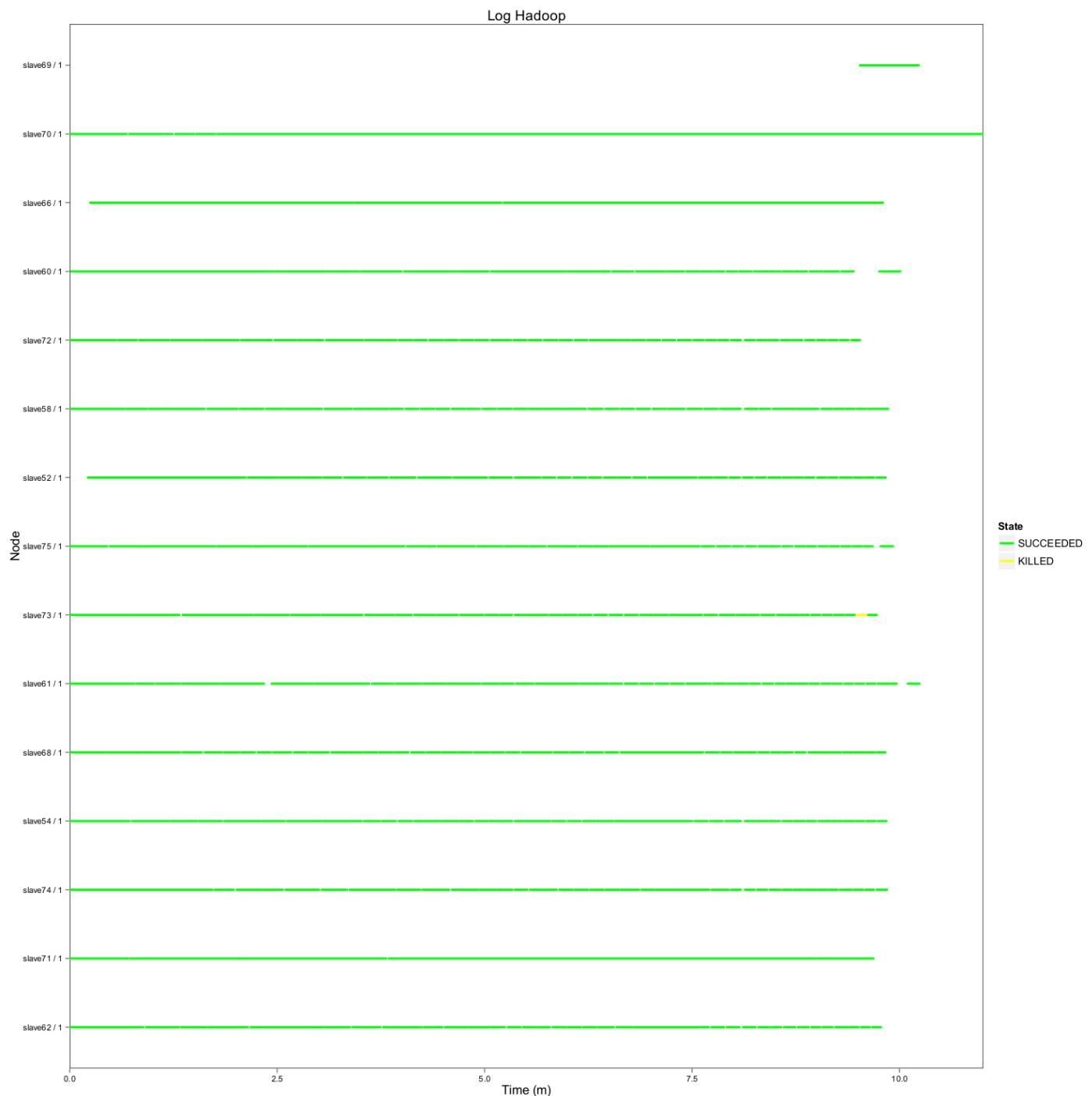
Il master, invece, non riceve quasi nulla.

4.9.6 Throughput pacchetti di rete inviati



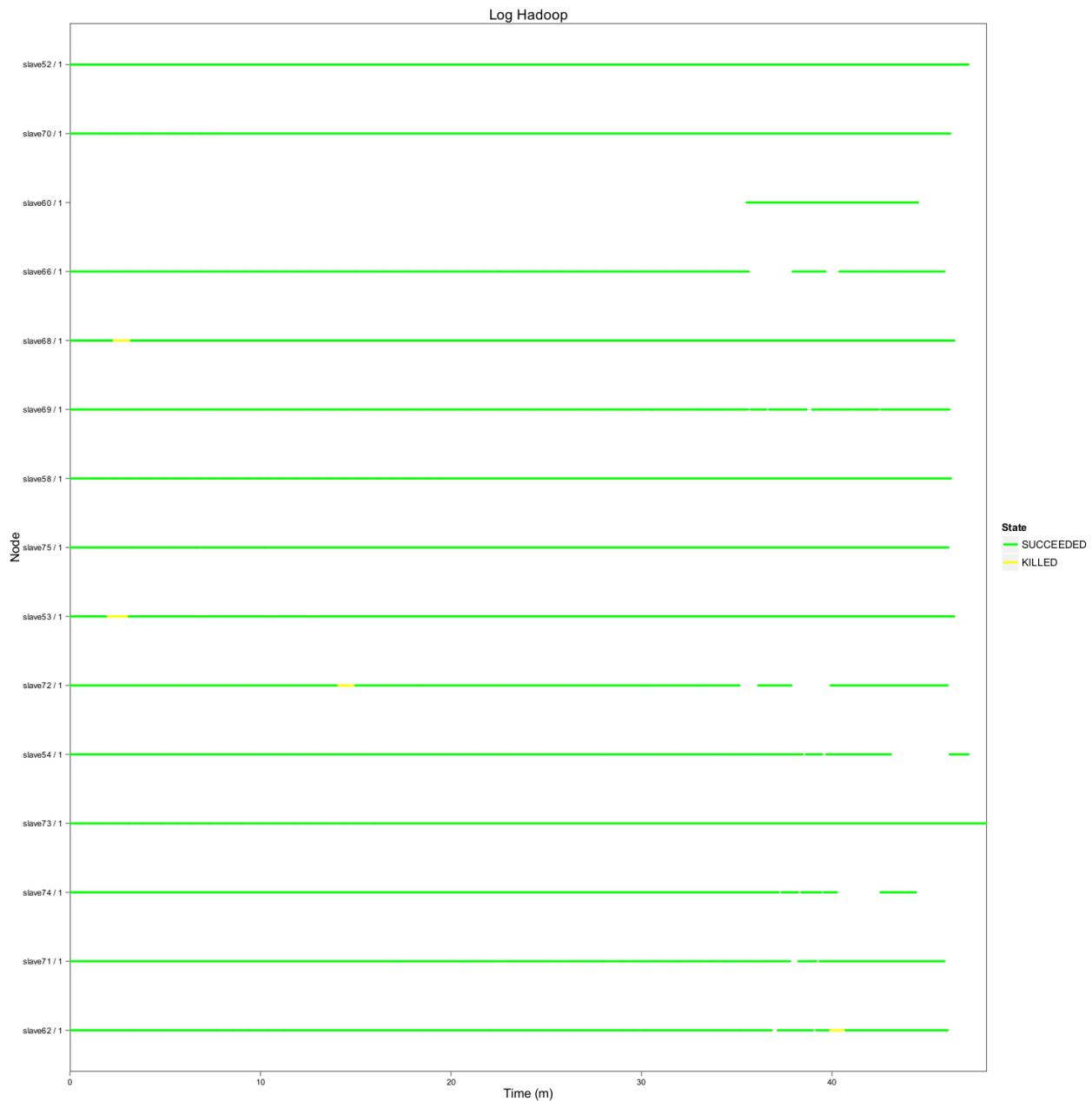
Dal grafico si nota come il master comunica con gli slave all'inizio di ogni job.

4.9.7 Job1 – Centroidi Random



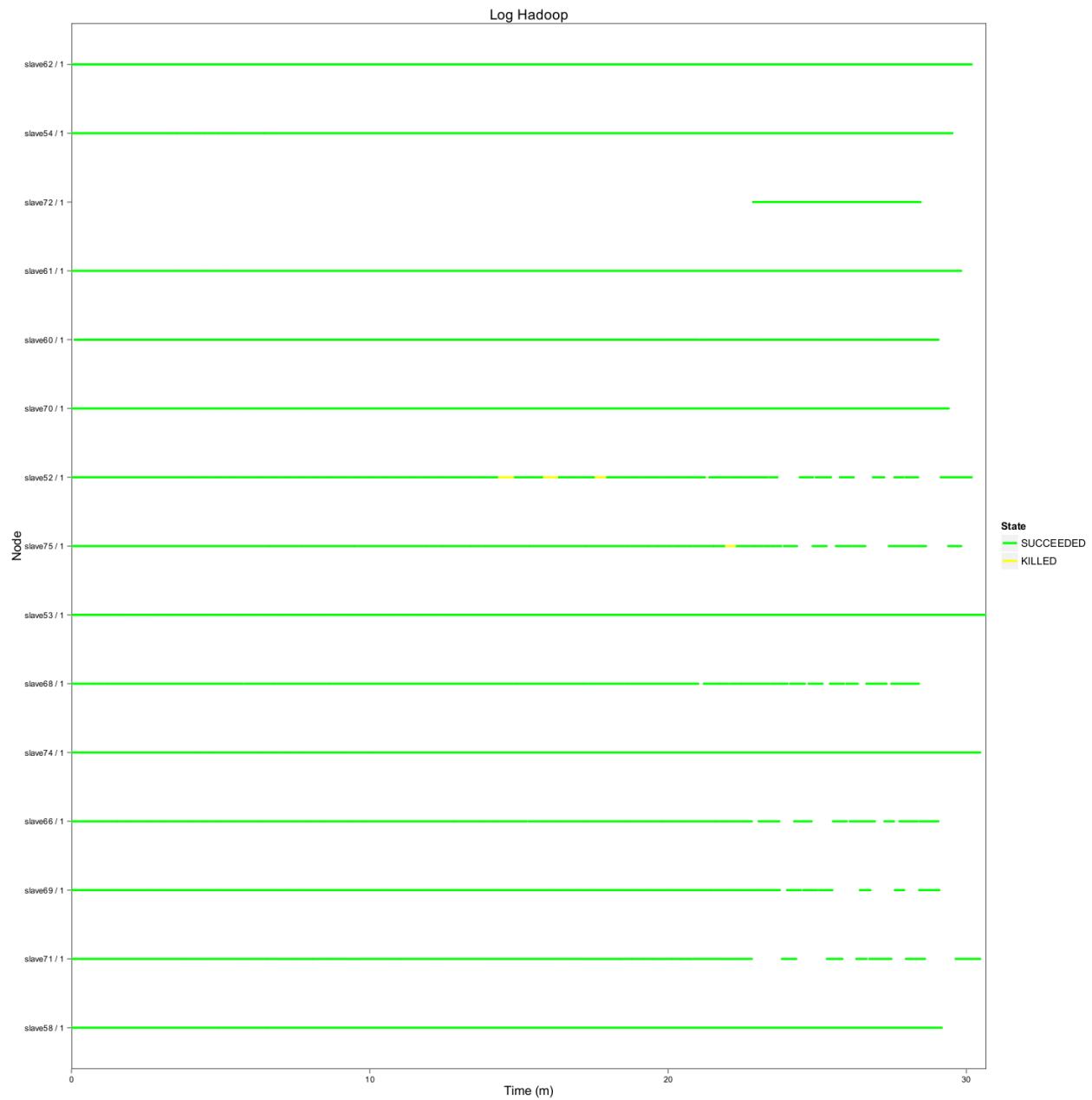
Si può notare come solo la macchine 70 ha lavorato per tutto il job, mentre il 69 ha lavorato solo durante la fase di reduce.

4.9.8 Job2 - KMeans



Dal grafico si nota che solo lo slave54 ha lavorato per tutto il tempo del job. Lo slave60 è stato utilizzato solo nella fase di reduce. Tutti gli altri hanno lavorato quasi sempre.

4.9.9 Job3 – KMeans

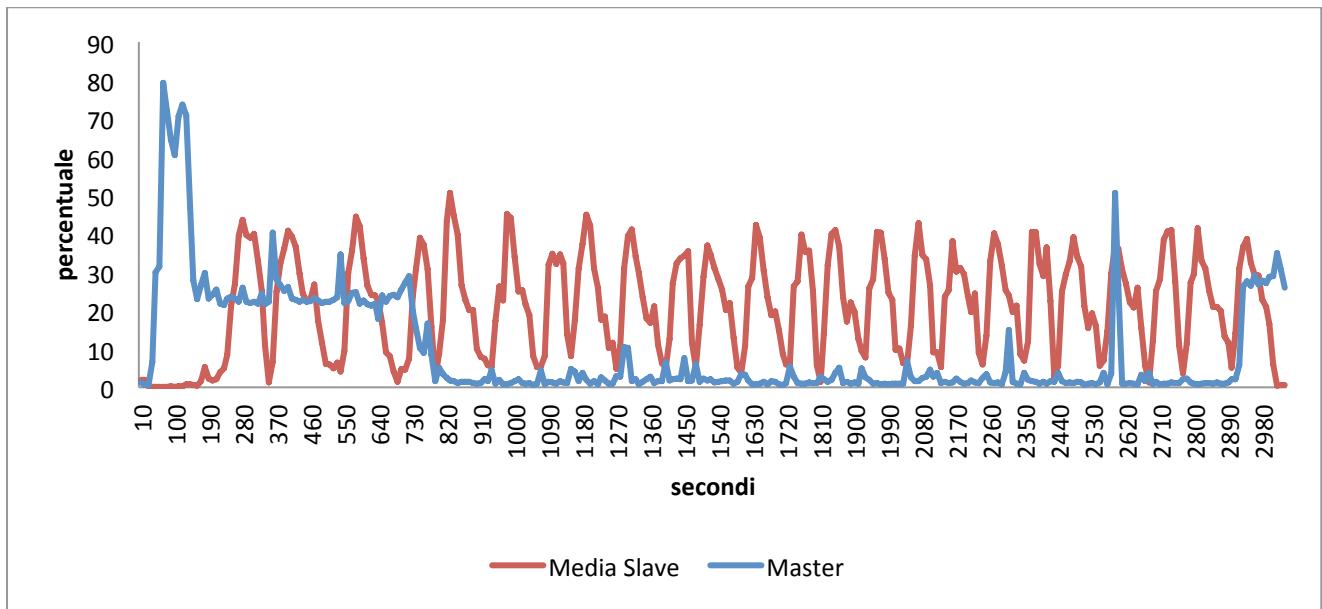


In questo caso è lo slave72 che ha lavorato solo nella fase di reduce, mentre lo slave53 ha lavorato solo per tutto il tempo del job.

4.10 Grafici Bisecting KMeans

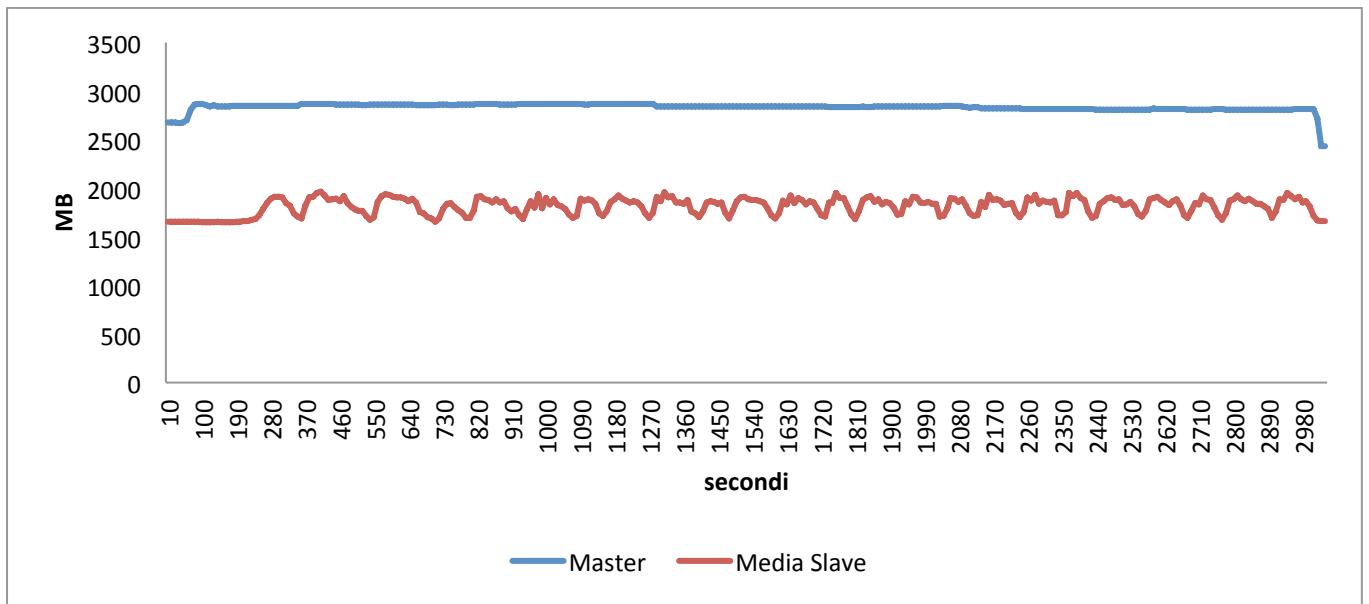
In questi grafici non sono state inserite le linee verdi per separare i vari Job al fine di non creare confusione nella visualizzazione. Dai grafici, però, si può notare come la forma a campana della media degli slave evidensi i job eseguiti. Ognuno mediamente dura 3 minuti.

4.10.1 CPU



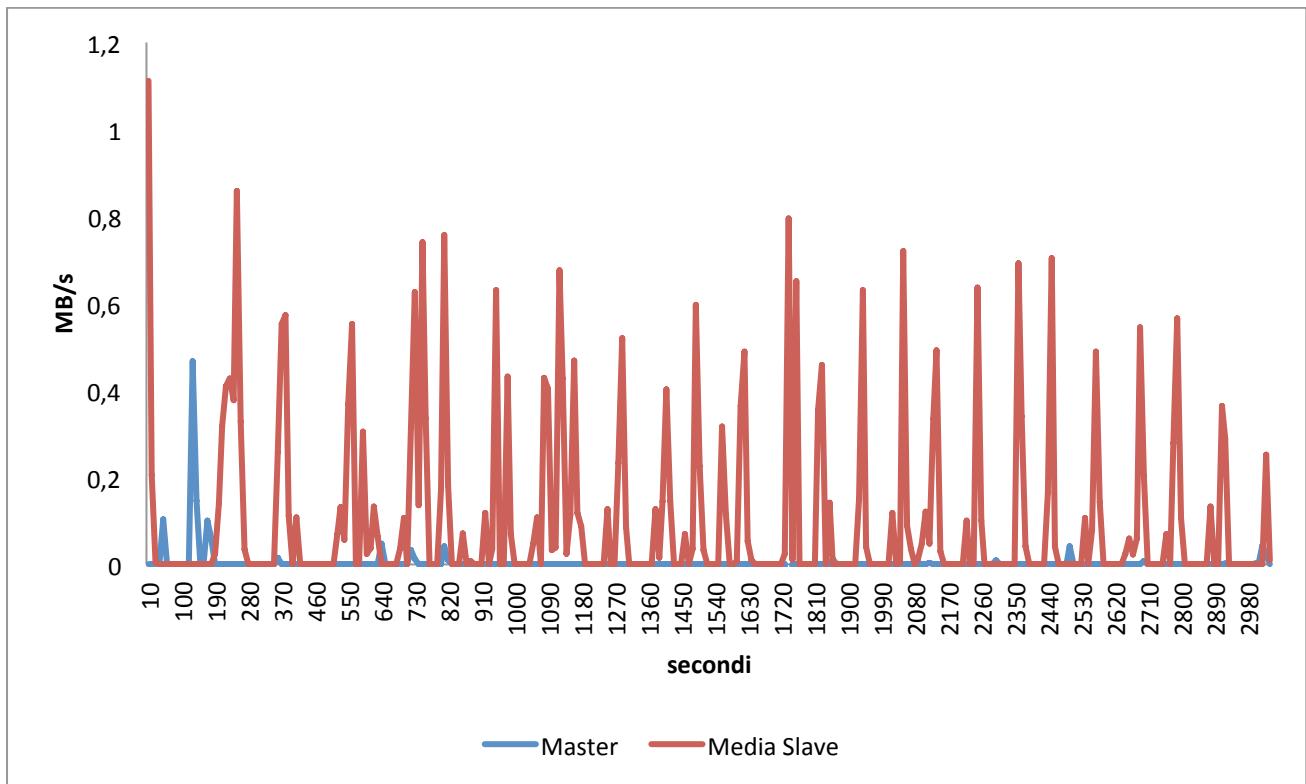
Dal grafico si può notare che ogni macchina slave al massimo sfrutta metà della CPU disponibile. È importante sottolineare che a causa della memoria utilizzata si è potuto instanziare un solo container.

4.10.2 Memoria



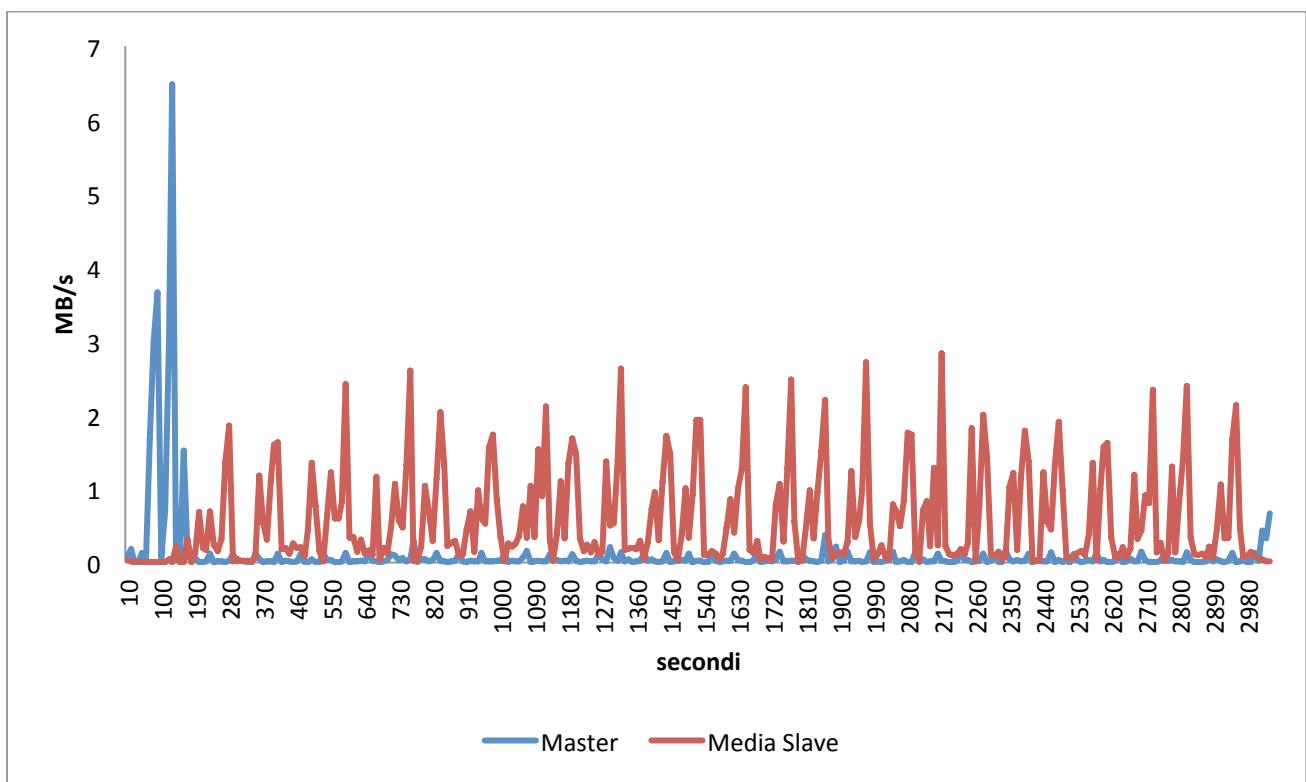
Si può vedere che in media gli slave utilizzano tra 1.5GB e 2GB di RAM, quindi non è stato possibile istanziare due container.

4.10.3 Lettura da disco

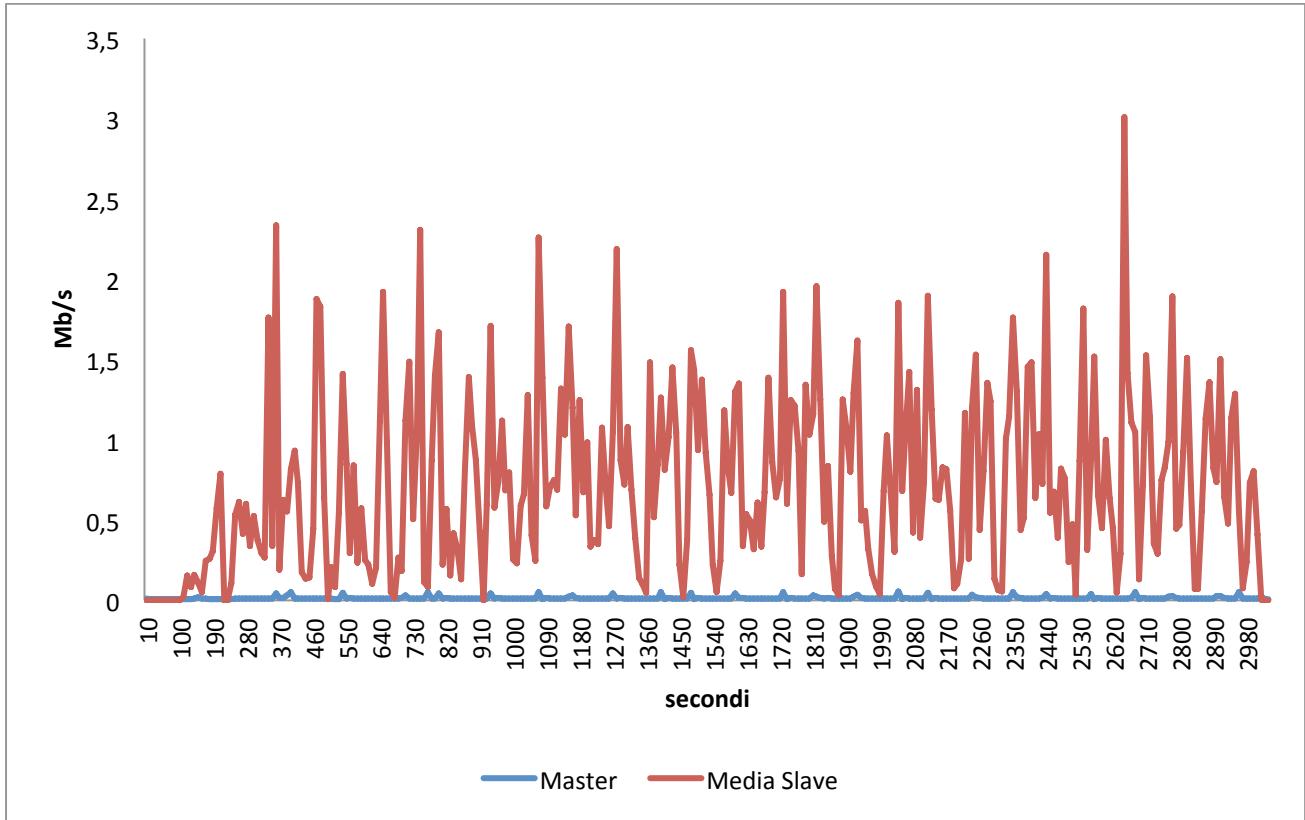


Nella fase map gli slave leggono dall' HDFS i dati da clusterizzare.

4.10.4 Scrittura su disco

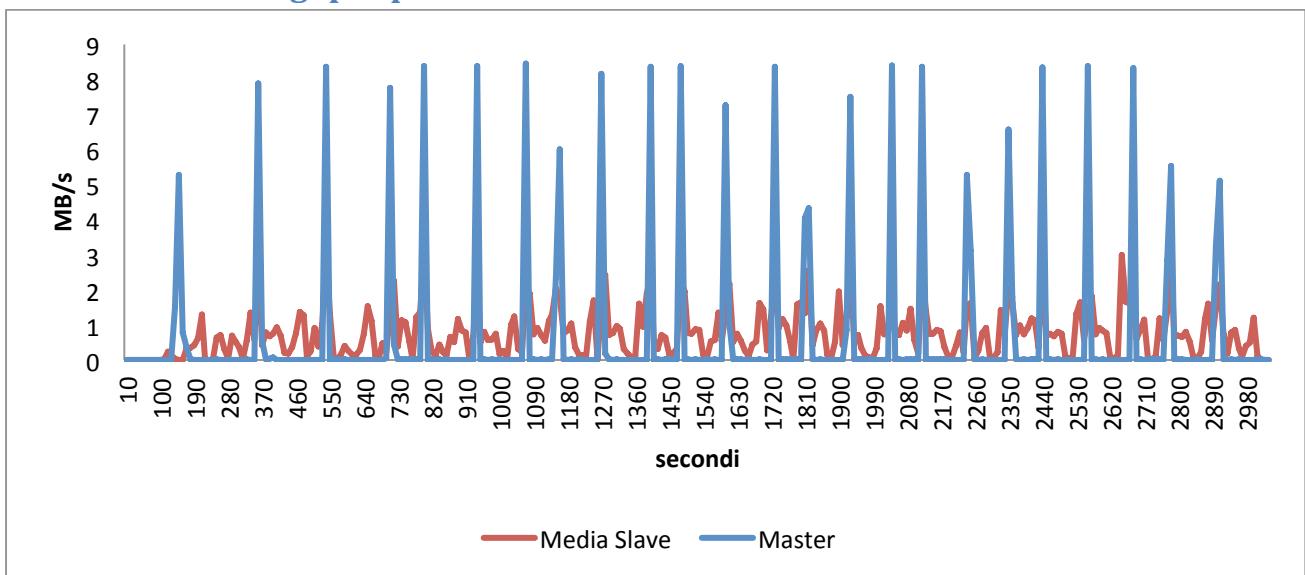


4.10.5 Throughput pacchetti di rete ricevuti



Come per il KMeans gli slave comunicano tra loro molto frequentemente durante l'esecuzione dei job.

4.10.6 Throughput pacchetti di rete inviati



Il master comunica con gli slave solo all'inizio dei job (i picchi della curva blu), mentre gli slave comunicano frequentemente durante i job.

4.11 Conclusioni

Il clustering è un argomento molto vasto che può essere utilizzato in svariati contesti tra i quali la classificazione di documenti.

Il lavoro ha mostrato come gli algoritmi implementati ben si prestino ad essere parallelizzati a patto di avere a disposizione una grossa quantità di dati; infatti se si hanno a disposizione solamente pochi GB di documenti, un'infrastruttura complessa quale Hadoop non ha motivo di essere utilizzata: l'overhead pagato per avviare e orchestrare il sistema non comporta alcun beneficio, in termini di tempo, al raggiungimento del risultato finale. Per tale motivo in questo lavoro abbiamo mostrato come su un dataset di 5 GB il tempo impiegato su 31 macchine fosse il medesimo di quello impiegato su 16 e solamente replicando 9 volte tale input l'aumento dello speedUp (soprattutto quello incrementale) fosse significativo.

Per quanto riguarda la qualità di classificazione, invece, dagli esperimenti condotti è emerso come utilizzando la metrica euclidea molto spesso i documenti di categorie differenti si addensino tutti nello stesso cluster mentre con la distanza del coseno i cluster tendono ad essere più omogenei: vi è prevalenza di documenti appartenenti ad una sola categoria o a categorie simili, come ad esempio lettere e arte, o matematica, fisica e ingegneria.

Tra il bisecting e il KMeans si può notare come il bisecting, anche con poche iterazioni separi meglio i documenti: infatti vengono computati centroidi che riescono ad individuare argomenti simili (ad esempio fisica e matematica) e per categorie molto omogenee (vedi lingue) una netta separazione dalle altre. Al contrario per il KMeans ci sono più centroidi per la stessa categoria pertanto molte volte in ogni cluster vi sono molte tipologie di documenti.

Per entrambi gli algoritmi testati, anche provando a moltiplicare per un fattore pari a 10 il numero di iterazioni e a considerare come centroide iniziale, per ogni cluster, un documento di una categoria differente, purtroppo non è stato possibile discriminare in maniera netta le varie tipologie di documenti. Pertanto sarebbe opportuno, al fine di giungere ad una soluzione migliore, scegliere in maniera più accurata le features iniziali (scartando quelle molto o poco significative) e soprattutto scegliere in maniera accurata i centroidi iniziali attraverso metodi gerarchici o utilizzando delle misure di non omogeneità.

5. Esperimenti sul clustering

Al fine di confrontare la qualità di classificazione degli algoritmi KMeans e bisecting riportiamo di seguito i risultati dell'analisi condotta su un sottoinsieme (1.5 GB di 5 GB) di 42.613 documenti scaricati³. Da notare come, la bontà di classificazione non sia stata misurata in termini di misure di non omogeneità⁴ poiché il cluster di appartenenza di ogni documento era noto a priori; in particolare i documenti scaricati da [2] afferiscono alle seguenti categorie:

1. Lettere: 6.752 documenti
2. Fisica: 7.479 documenti
3. Ingegneria: 9.741 documenti
4. Matematica: 8.056 documenti
5. Biologia: 2.116 documenti
6. Arte: 3.334 documenti
7. Economia: 765 documenti
8. Medicina: 2.973 documenti
9. Lingue: 1.397 documenti

I due algoritmi implementati (Algoritmo 1 e Algoritmo 4 con centroidi composti da 200 parole) sono stati eseguiti sui dataset TF e TFIDF utilizzando sia la metrica euclidea che la metrica del coseno. Per quanto riguarda il numero di iterazioni, alla luce di quanto detto in precedenza, ad ogni 6 iterazioni del KMeans sono state contrapposte 2 iterazioni del bisecting per ognuno dei nove cluster da ricercare.

Sono mostrati di seguito i risultati ottenuti. Da notare come ognuno dei nove plot in figura corrisponda ad un cluster e, per ogni plot siano mostrate delle barre indicanti il numero di documenti per ognuno delle 9 materie da classificare. È importante sottolineare come il risultato ottimo sia, per ognuno dei 9 plot, una barra riportante il numero totale dei documenti di una data categoria mentre le altre (le rimanenti 8) una barra di altezza zero

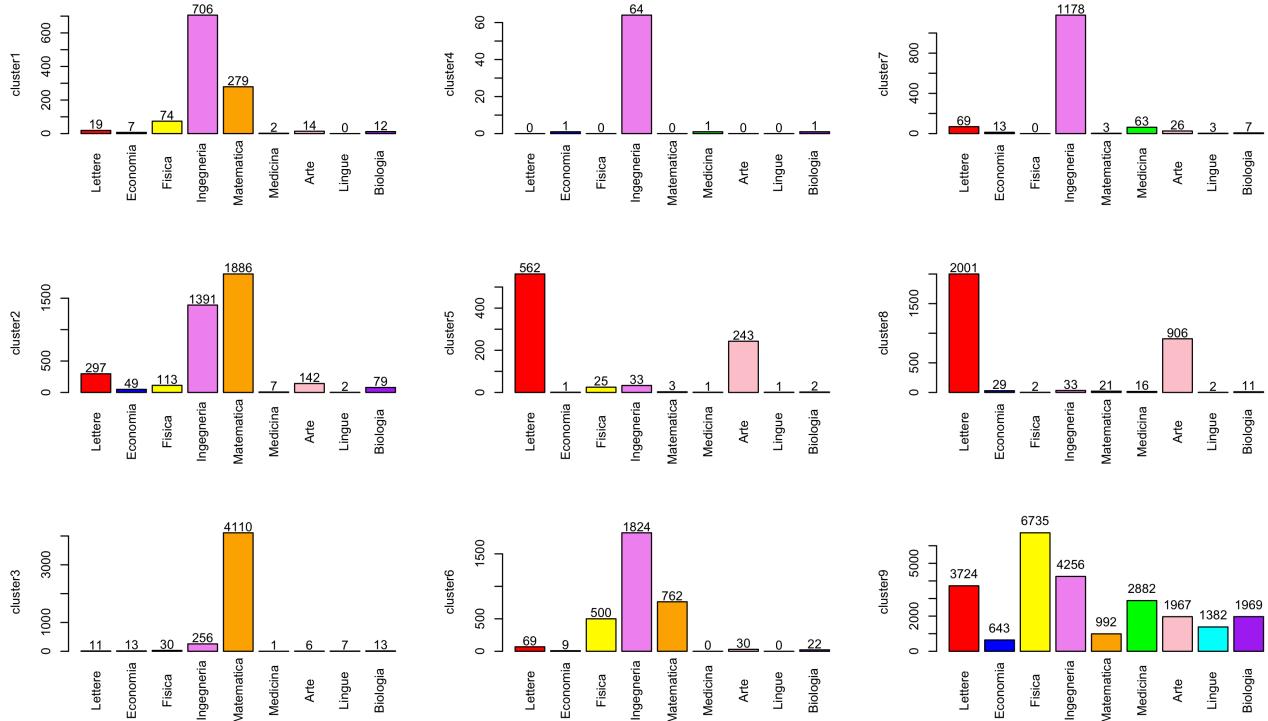
³ L'analisi condotta in questa sezione ha il solo obiettivo di testare la bontà di classificazione degli algoritmi implementati pertanto, i dataset TF e TFIDF utilizzati per tali esperimenti contengono solamente documenti di cui si conosce a priori l'insieme (la categoria) di riferimento.

⁴ Al termine della classificazione, gli individui appartenenti allo stesso cluster dovrebbero essere il più possibile omogenei tra di loro e il più possibile differenti da quelli appartenenti agli altri cluster individuati. Statisticamente i cluster dovrebbero essere individuati in modo da minimizzare la misura di non omogeneità statistica all'interno dei cluster (within) e massimizzare la misura di non omogeneità statistica tra i gruppi (between) [8].

5.1 Dataset TF

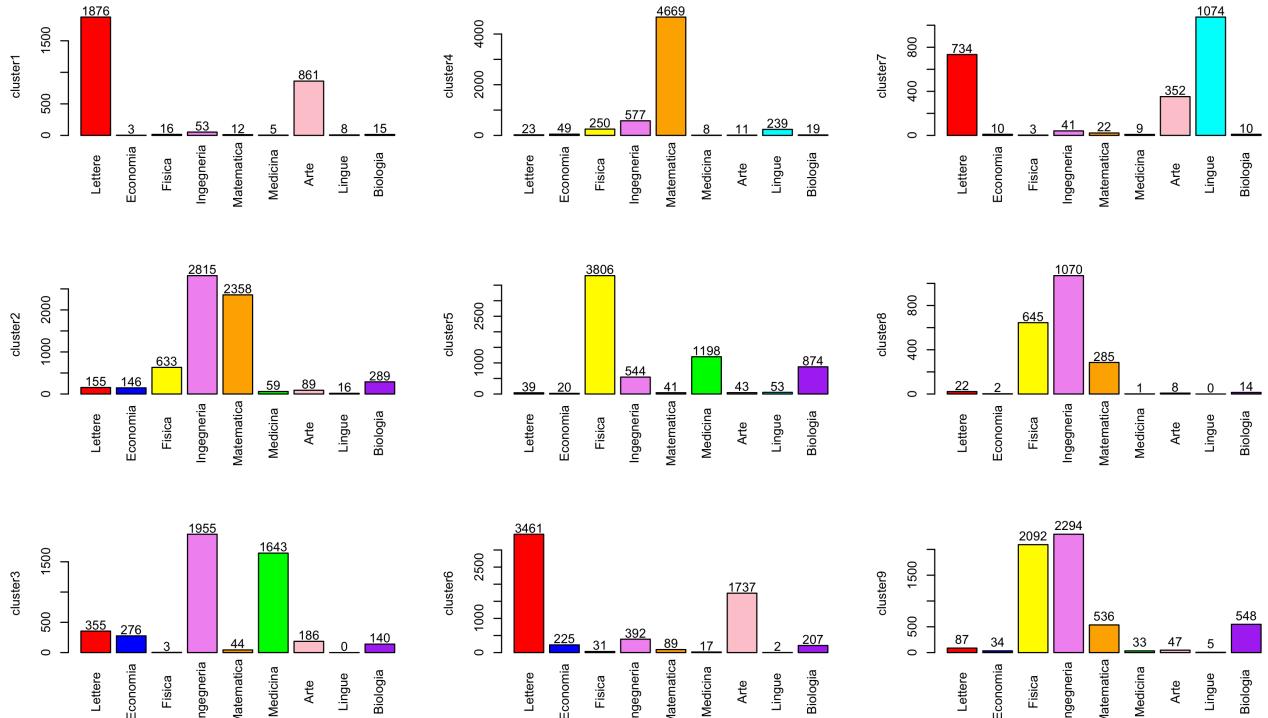
5.1.1 Bisecting KMeans con metrica euclidea

Come si evince dal grafico sottostante sono stati individuati due cluster con documenti di Ingegneria, due cluster con documenti di Lettere e Arte, due misti Ingegneria e Matematica ed uno solo con documenti di Matematica. C'è un cluster con documenti di fisica, ingegneria e matematica (simili tra loro come terminologia) ed uno misto con tutte le materie.



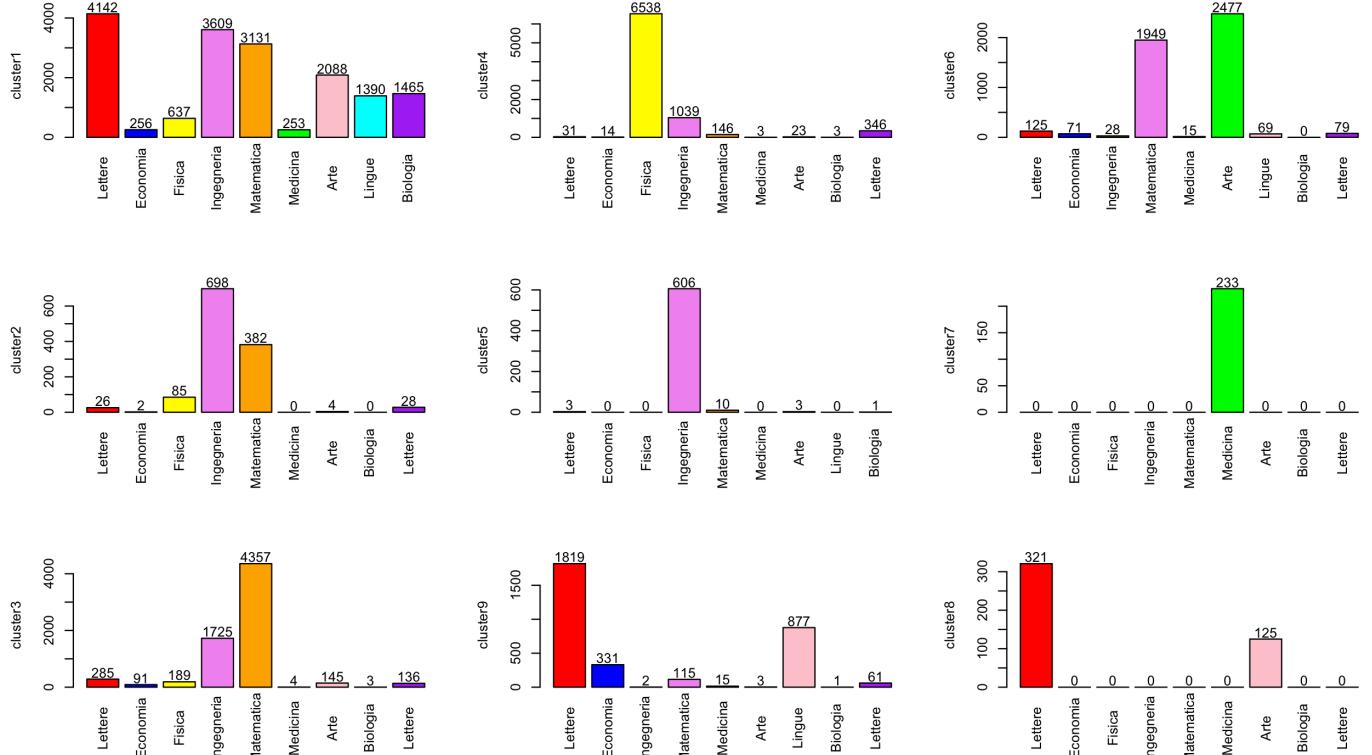
5.1.2 Bisecting KMeans con metrica del coseno

Con questa metrica, vengono individuati cluster che hanno diverse materie (e non cluster diversi con documenti della stessa materia): c'è un cluster prevalentemente letterario, uno di matematica, e poi ci sono cluster che hanno documenti di materie che usano terminologie simili: quali lettere, arte e lingue o fisica, ingegneria e matematica. Vi è un solo cluster che ha materie non certo simili quali ingegneria e medicina.



5.1.3 KMeans con metrica euclidea

Come nel caso del Bisecting KMeans, ci sono diversi cluster con le stesse materie: due a prevalenza di ingegneria, due con lettere e arte. Buona invece la classificazione per fisica.

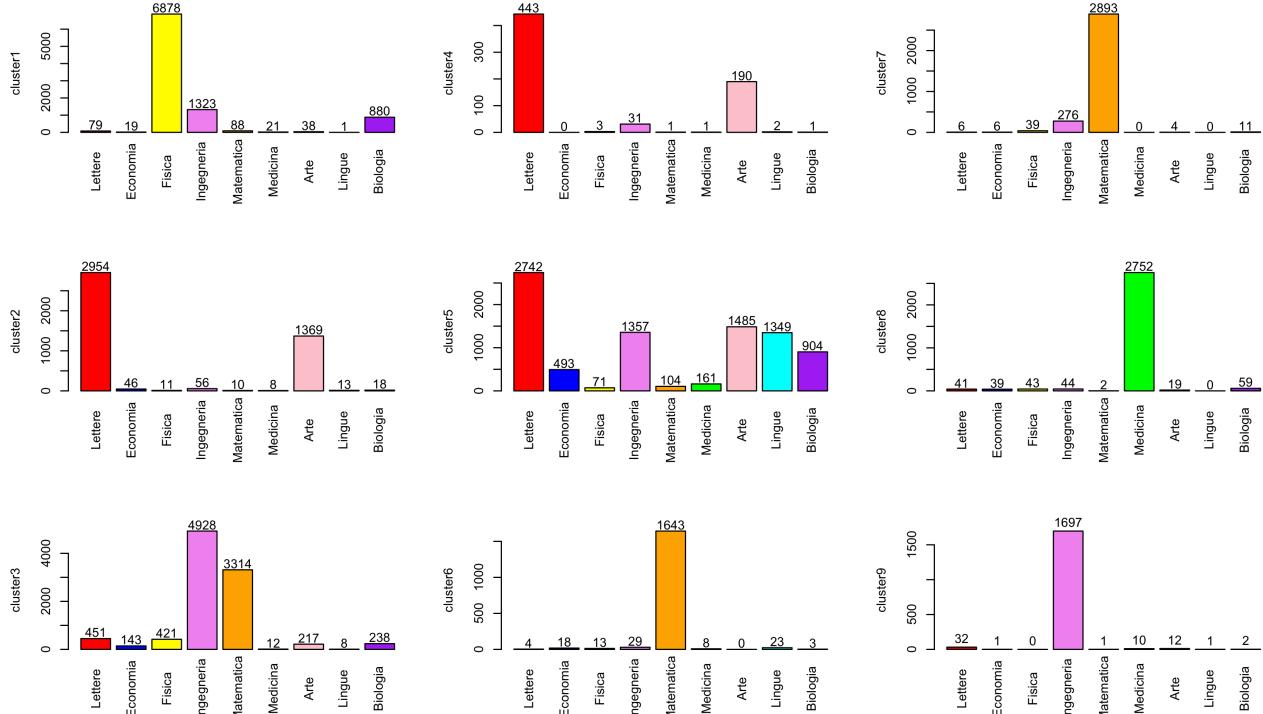


5.1.4 KMeans con metrica del coseno

Con la metrica del coseno vengono individuati cluster con argomenti diversi, da notare i cluster con prevalenza di:

- Fisica e Ingegneria
- Lettere e arte (due)
- Medicina
- Matematica e Ingegneria

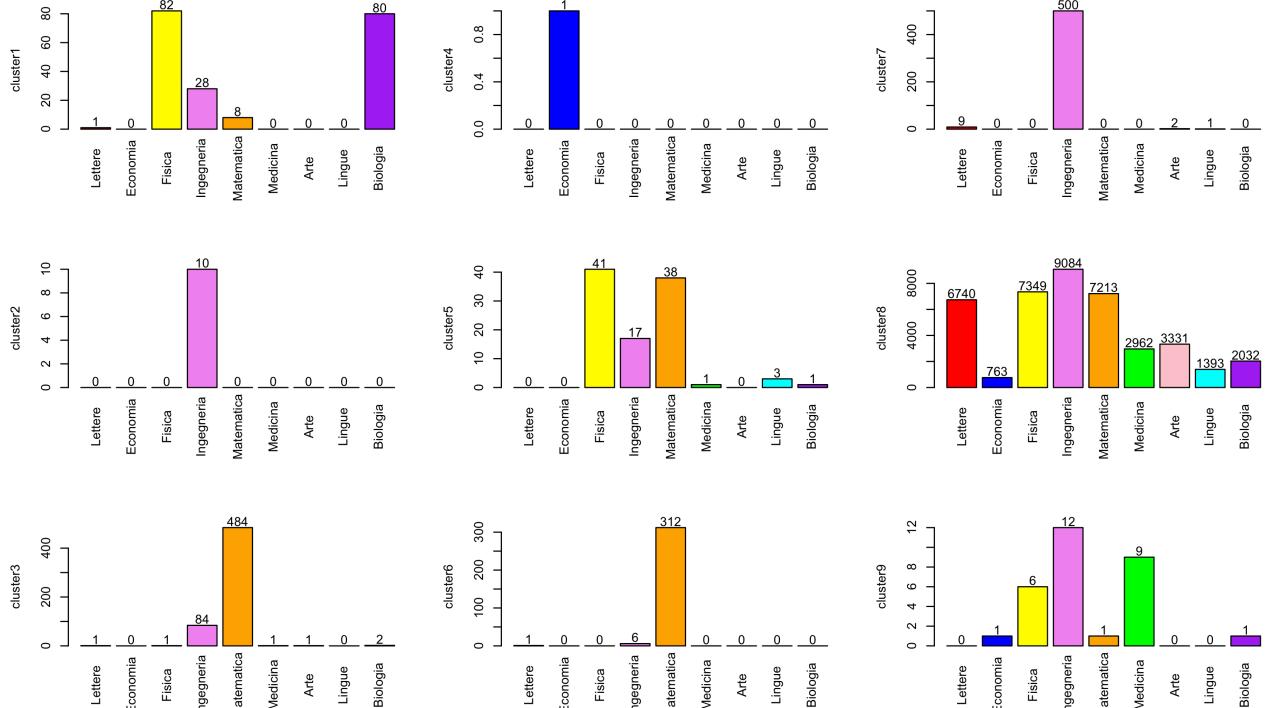
Vi è un solo cluster eterogeneo.



5.2 Dataset TFIDF

5.2.1 Bisecting KMeans con metrica euclidea

Il bisecting KMeans con il dataset TF-IDF riesce ad individuare due cluster che hanno solo documenti di ingegneria (anche se une ha solo 10 documenti) e due con soli documenti di matematica. C'è un cluster interamente di economia, due con materie matematiche: uno con fisica, ingegneria e biologia e uno con fisica, ingegneria e medicina.



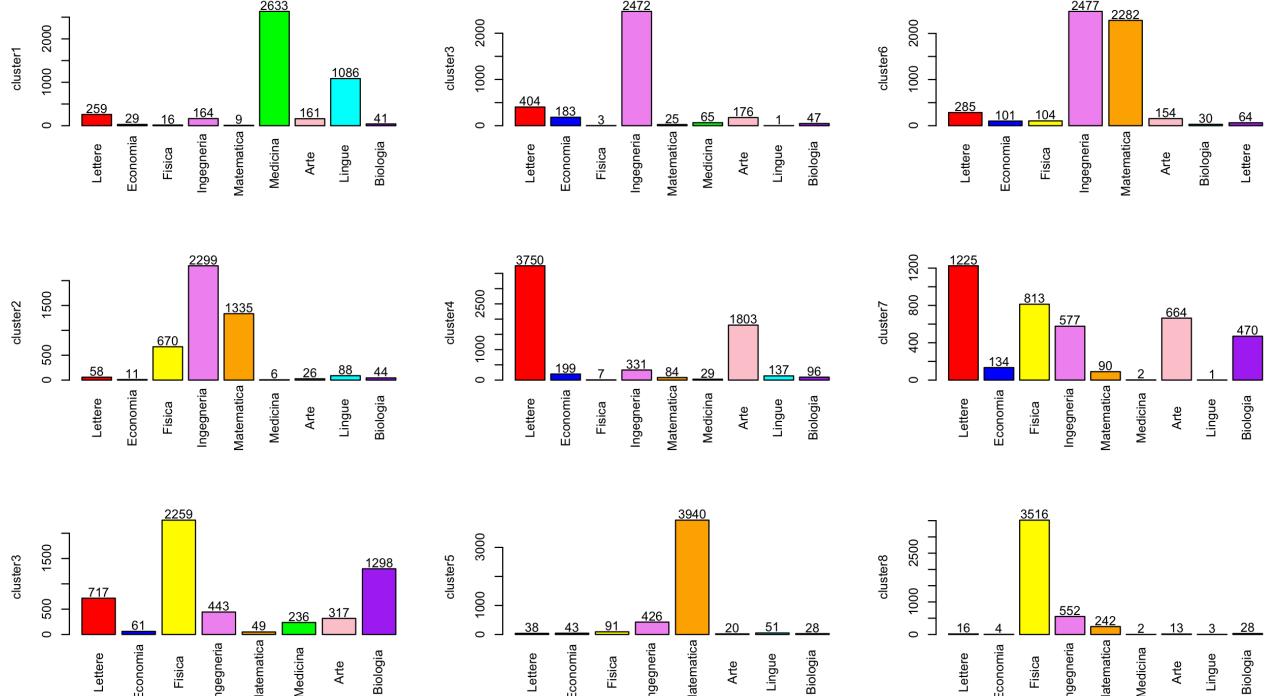
5.2.2 Bisecting KMeans con metrica del coseno

Vengono individuati cluster con due argomenti simili tra loro:

- Ingegneria e medicina
- Ingegneria e matematica
- Lettere e arte

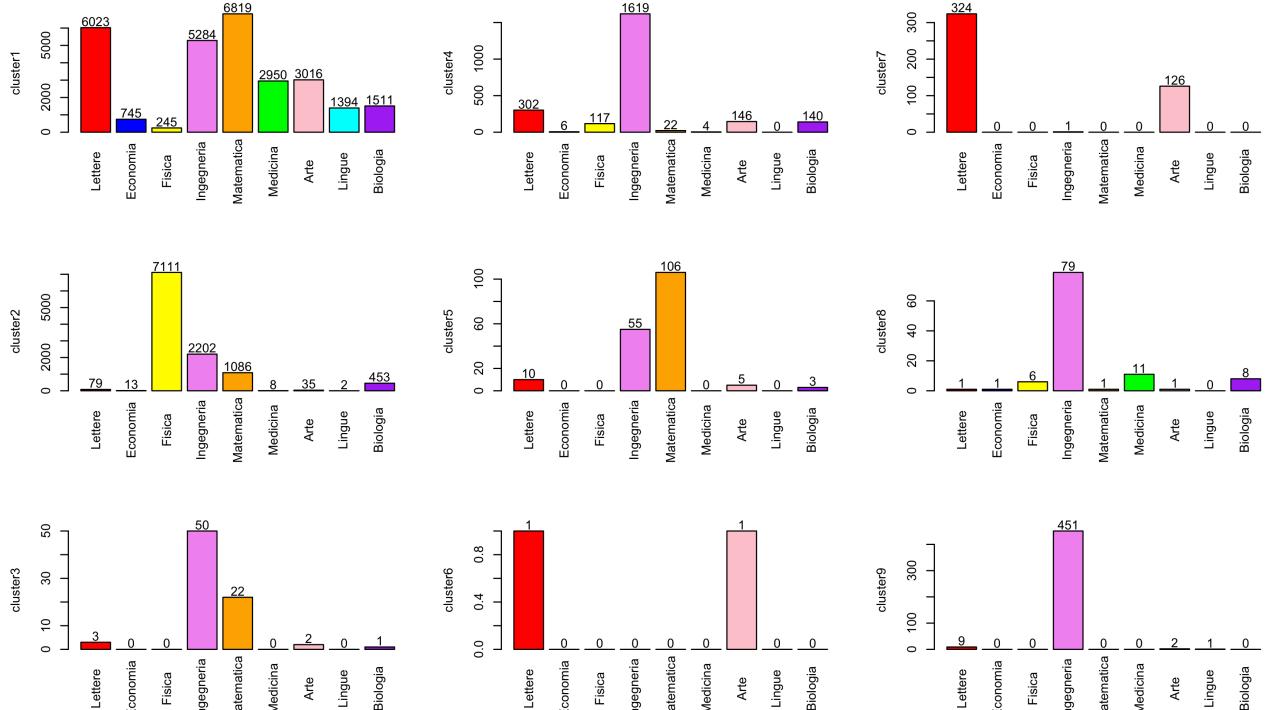
Poi vi sono cluster con una sola materia: medicina, ingegneria, matematica e fisica

Infine ci sono due cluster molto misti ed uno contenente medicina e lingua che è molto eterogeneo.



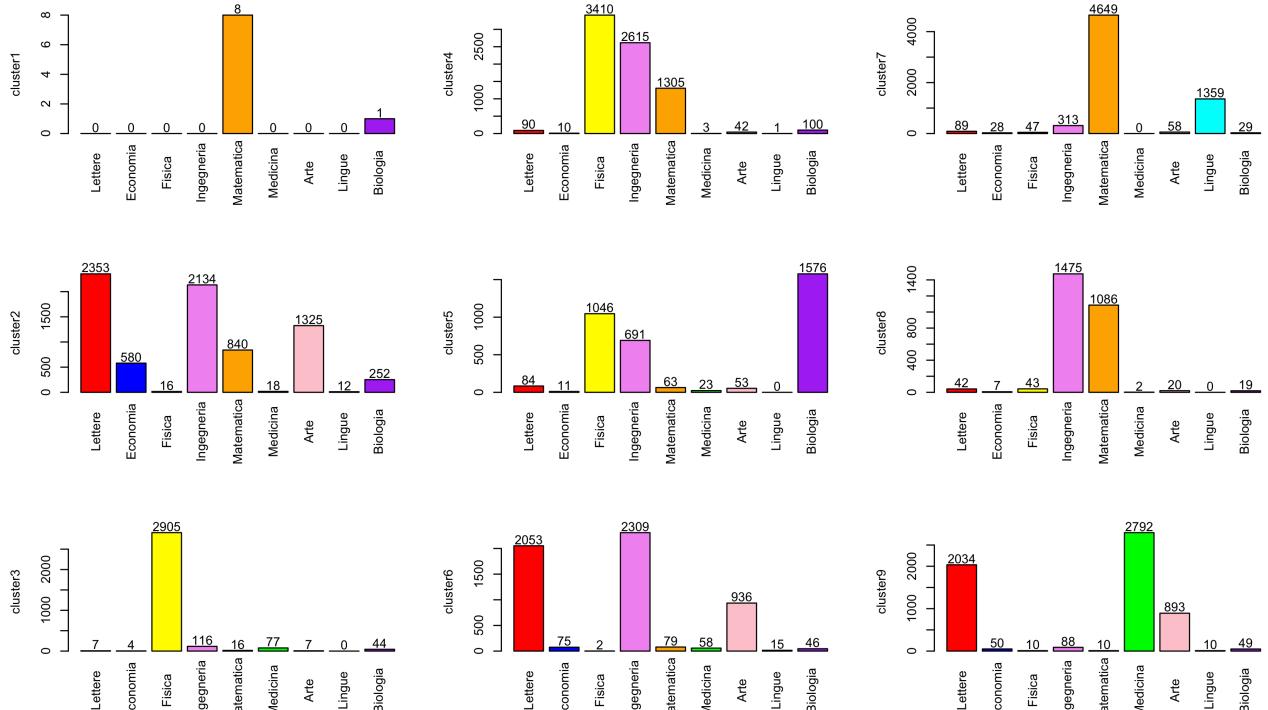
5.2.3 KMeans con metrica euclidea

Vengono individuati 4 cluster a prevalenza di ingegneria, uno a prevalenza di lettere e arte e due con materie matematiche. La clusterizzazione non ha prodotto tuttavia buoni risultati poiché il cluster1 è molto eterogeneo.



5.2.4 KMeans con metrica del coseno

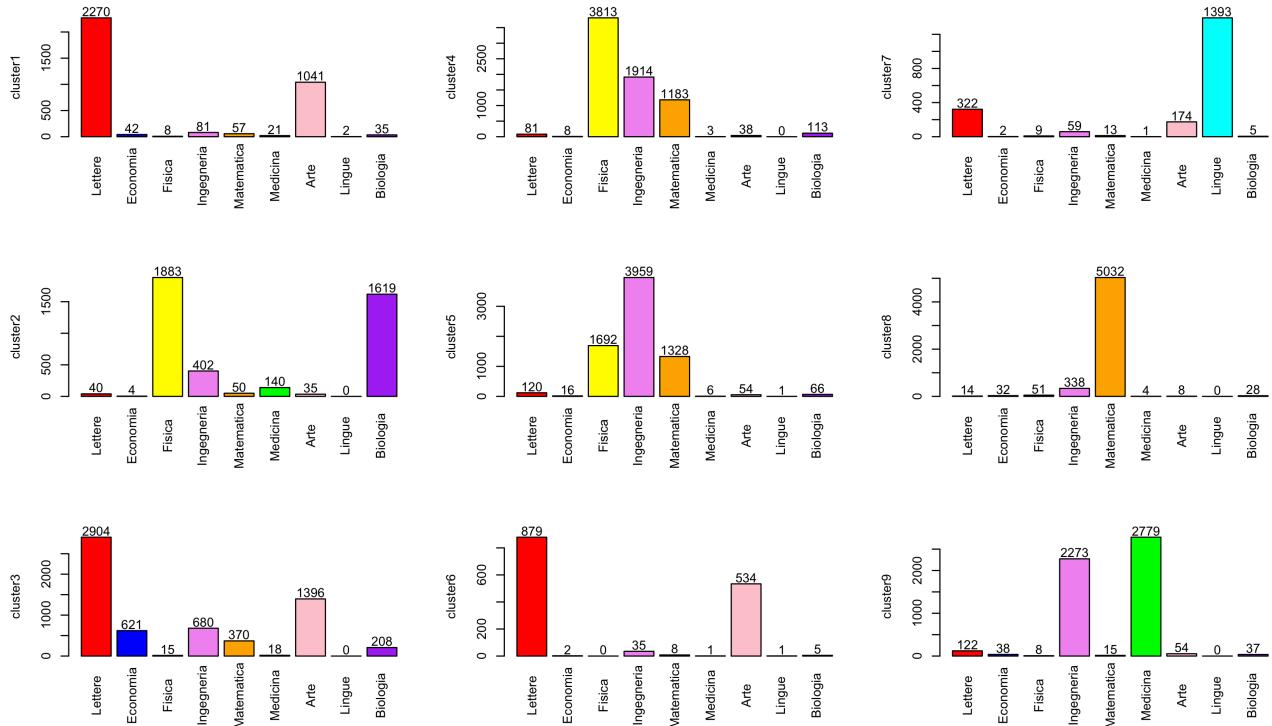
In questo caso ci sono solo 3 cluster che individuano un argomento bel preciso: due di medicina, uno di fisica e due a prevalentemente matematici. I restanti sono molto eterogenei.



5.3 Centroidi scelti non a caso

5.3.1 Daset TF e KMeans con metrica del coseno

Si può notare che in questo caso vengono individuati dei cluster più omogenei tra loro. Vi è un solo cluster eterogeneo, anche se confrontandolo con quelli visti precedentemente, lo è di meno.



5.4 Analisi dei risultati

Dagli esperimenti condotti è emerso come la **metrica euclidea** lavori meglio con il dataset TF-IDF, mentre la **metrica del coseno** lavori meglio con il dataset TF.

In generale si può dire che il Bisecting KMeans lavora meglio nell'individuazione di cluster con materie differenti, al contrario il kmeans individua cluster che hanno le stesse materie. È importante ricordare che il clustering è fortemente influenzato dalla scelta dei centroidi iniziali: per esempio con KMeans (paragrafo 5.1.4) vi sono due cluster matematici, poiché, probabilmente, sono stati scelti due centroidi di quella categoria. Invece nell'ultimo esperimento mostrato, con i centroidi non scelti a caso, vi è un clustering migliore.

6. Riferimenti

- [1] Features Extraction on Hadoop MapReduce Framework, Ciro Amati, Stefania Cardamone, Giovanni Grano.
- [2] Sito web: [Arxiv](#).
- [3] Javadoc di Hadoop: [SequenceFile](#).
- [4] Introduzione ad Apache Hadoop, Gianluca Roscigno (appunti del corso).
- [5] Text mining with lucene and hadoop:document clustering with feature extraction, Dipesh Shrestha.
- [6] Enhancing Text Document Clustering Using Non-negative Matrix Factorization and WordNet, Chul-Won Kim e Sun Park, Member, KIICE.
- [7] Pagina Wikipedia del [Coseno di Similitudine](#).
- [8] Metodi e tecniche per l'analisi dei dati, Amelia Giuseppina Nobile (appunti del corso).
- [9] Hybrid bisecting KMeans clustering algorithm, Keerthiram Murugesan, Jun Zhang.
- [10] Implementation of Space Optimized Bisecting KMeans (BKM) Based on Hadoop, Yanshen Yin, Chengguang Wei, Guigang Zhang, Chao Li.
- [11] A comparison of document clustering techniques, Micheal Steinbach, George Karypis, Vipin Kumar.
- [12] Performance and Benchmarking , Martin, Milo.
- [13] Algorithm AS 136: A K-Means Clustering Algorithm, J. A. Hartigan and M. A. Wong.
- [14] Document Clustering using K-Means and K-Medoids, Rakesh Chandra Balabantaray, Chandrali Sarma, Monica Jha.