Sviluppo dell'algoritmo k-Means per sistema Hadoop

Davide Tarasconi

27 marzo 2019

1 Introduzione

Il progetto prevedeva l'utilizzo del sistema *Hadoop* per sviluppare e testare l'algoritmo **k-Means** sfruttando il paradigma *MapReduce* fornito proprio dal sistema.

1.1 K-Means

E' un algoritmo di clustering che prevede di minimizzare la somma delle distanze tra gli elementi del dataset e i rispettivi centri a cui sono associati in modo deterministico. Una volta assegnati tutti gli elementi, viene effettuata la media per aggiornare i centri da usare nel prossimo ciclo. Quando i centri sono uguali a quelli del passo successivo possiamo terminare l'esecuzione perché ci troviamo in una situazione di convergenza. Con questo algoritmo se conosciamo i cluster possiamo trovarne i centri, mentre se conosciamo i centri possiamo derivarne il cluster assegnandogli gli elementi più vicini. E' sensibile all'inizializzazione e pessime scelte di centri possono portare a risultati inattesi, per questo motivo si fanno varie prove per scegliere le configurazioni migliori.

1.1.1 Vantaggi

- Implementazione facile e veloce
- Ricerca di centri che minimizzano la varianza

1.1.2 Svantaggi

- Forte sensibilità alla scelta iniziale dei centri

- Forte influenza outlier sulle medie
- Numero fisso di cluster

2 Architettura

Andiamo ad analizzare le principali architetture utilizzate.

2.1 Hadoop

Hadoop è un framework open source sviluppato da Apache che permette di eseguire applicazioni distribuite che elaborano grandi quantità di dati (petabyte di dati) sfruttando cluster costituiti da commodity hardware consentendo di migliorarne la scalabilità. Hadoop implementa il paradigma MapReduce che permette di suddividere l'applicazione in piccoli frammenti ognuno dei quali verrà eseguito su un nodo qualsiasi del cluster. Hadoop è formato da quattro moduli principali:

- **Hadoop Common**
- **Hadoop Distribuited File Systems (HDFS)**
- **Hadoop YARN**
- **Hadoop MapReduce**

2.1.1 Hadoop Common

E' uno strato di software comune che fornisce supporto agli altri moduli

2.1.2 Hadoop Distribuited File Systems (HDFS)

HDFS è stato progettato per immagazzinare grandi quantità di file di grandi dimensioni all'interno di un sistema cluster in modo affidabile e scalabile. Sfrutta un'architettura *master/slave* composta da:

- **NameNode:** applicazione *master* eseguita sul server, si occupa della gestione del namespace del filesystem regolando l'accesso ai file, gestendone i nomi e i nomi dei blocchi in cui vengono suddivisi e determinando la distribuzione dei blocchi e le strategie di replica per garantire l'affidabilità del sistema. Si occupa dell'esecuzione delle operazioni nel filesystem come apertura/chiusura file, ecc.

- **DataNode:** ne esiste uno per ogni nodo del cluster e si occupa della gestione dello spazio di storage di quel nodo. Eseguono le operazioni di lettura e scrittura richieste dal cliente e si occupano della creazione e gestione dei blocchi. E' lo *slave* del sistema.
- **SecondaryNameNode o CheckPointNode:** scarica periodicamente l'immagine del *NameNode* per aiutarlo a essere più efficiente.
- **BackupNode:** nodo di failover per avere sempre un nodo secondario sincronizzato al *NameNode*.

2.1.3 Hadoop YARN

E' stato introdotto dalla seconda versione di Hadoop e permette di dividere in due demoni separati le funzionalità del gestore di risorse e le funzionalità di job scheduling/monitoring così da avere un *ResourceManager globale*(RM) e un *ApplicationMaster*(AM) per applicazione che si occupa di richiedere le risorse al RM.

2.1.4 Hadoop MapReduce

MapReduce è un paradigma che permette di eseguire grandi elaborazioni distribuite come una sequenza di operazioni su dati in forma *chiave/valore (k/v)*. E' composto da due fasi più una opzionale:

- **Map:** divide i dati in input in frammenti che verranno passati ai task map presenti all'interno del cluster. Questi elaboreranno i dati e forniranno una coppia chiave/valore intermedia che fungerà da input per la fase successiva.
- **Reduce:** le coppie k/v vengono instradate nel relativo reducer presente nel cluster che le elaborerà fornendo in output una singola coppia k/v.
- **Combiner:** è opzionale e lavora in modo simile ad un task *Reduce* ma può essere eseguito un numero indefinito di volte.

2.2 Java

Linguaggio di programmazione ad oggetti sviluppato da Oracle. Hadoop sfrutta la versione 1.8.0_161 per permettere agli utenti di sviluppare applicazioni distribuite.

3 Sviluppo

L'applicazione che implementa l'algoritmo **k-Means** è stata sviluppata in Java sfruttando il framework di Hadoop per implementare il paradigma *MapReduce*.
L'applicazione è stata strutturata nel seguente modo:

- Una classe per gestire gli elementi del dataset, per assegnarli al centro più vicino e per calcolare i nuovi centri (`Element.java`).
- Una classe per eseguire l'applicazione (`KMeansDriver.java`).
- Una classe per il task *Map* (`KMeansMapper.java`).
- Una classe per il task *Combine* (`KMeansCombiner.java`).
- Una classe per il task *Reduce* (`KMeansReducer.java`).

3.1 Element.java

Classe che permette la gestione di ogni singola tupla del dataset e di gestire i centri impostando il numero di parametri, memorizzando il numero di record appartenenti ad un determinato centro ed eseguendo il calcolo della media degli elementi associati per calcolare i nuovi centri.
Per permettere il passaggio degli oggetti `Element` tra un task e l'altro abbiamo ereditato la classe `WritableComparable` che fornisce i metodi di scrittura, lettura e confronto per gli oggetti in questione. la classe è composta dagli attributi:

- `protected ArrayList<DoubleWritable> parameters` : lista contenente i parametri del dataset da utilizzare per calcolare i nuovi centri e calcolare la distanza.
- `public DoubleWritable instanceNum` : numero di istanze appartenenti al centro che si sta analizzando, utilizzato per calcolare la media, cioè il nuovo centro.

I metodi più importanti di questa classe sono quello per il **calcolo della distanza** e quello per il **calcolo della media**.

3.1.1 Calcolo della distanza

E' presente un metodo statico per calcolare la distanza euclidea in base al numero di elementi della tupla (parametri) utilizzati.

```
public static double distance(Element c1, Element c2) {
    double res = 0;

    for(int i = 0; i < c1.getParam().size(); i++){
        res += Math.pow((c1.getParam().get(i).get() -
            c2.getParam().get(i).get()), 2);
    }

    return Math.sqrt(res);
}
```

Nel nostro caso calcoliamo la distanza tra centro e l'elemento in esame.

3.1.2 Calcolo della media

Dopo aver contato gli elementi appartenenti al centro in esame e aver effettuato la somma dei parametri passiamo al calcolo della media per trovare il nuovo centro.

```
public void mean(){
    for(int i = 0; i < parameters.size(); i++){
        parameters.get(i).set(parameters.get(i).get() /
            this.getInstance());
    }
    this.instanceNum.set(1);
}
```

3.2 KMeansDriver.java

E' la classe principale che si occupa della configurazione e dell'esecuzione dei task *Map*, *Combine* e *Reduce*. Permette di definire l'output del task Map e l'output finale dell'elaborazione, dato che in questo caso abbiamo due tipi di output:

1. **Output centri**: consiste nel file sequenziale contenente i nuovi centri trovati alla fine di un ciclo di elaborazione.

2. **Output dataset**: dopo aver finito l'elaborazione viene eseguito il task di Map per avere un output aggiuntivo composto dagli elementi del dataset associati al corrispondente centro di appartenenza.

Il file sequenziale risultante non è leggibile, per questo motivo viene creato un file di testo che ne traduce il contenuto per renderlo comprensibile.

Questa traduzione è situata nel filesystem Hadoop nella stessa cartella del file sequenziale, ovvero `centers/`. in questa cartella abbiamo:

- `cent.seq` : file sequenziale
- `cent.txt` : file di testo

Possiamo scegliere se far terminare l'esecuzione dopo un numero preciso di cicli oppure quando grazie all'utilizzo di contatori notiamo che i centri convergono in un determinato valore a meno di una soglia determinata (0.01).

```
if(job_conf.getCounters()
        .findCounter(KMeansReducer.CONVERGENCE.CONVERGE).getValue()
           == 0)
{
   converge = true;
}
```

3.3 KMeansMapper.java

Implementa l'interfaccia Mapper che fornisce il metodo che verrà eseguito come task *Map* e due metodi per eseguire pre e post-elaborazioni. In questo caso utilizziamo il metodo `setup` per caricare i centroidi prima dell'esecuzione del task, salvandoli in un vettore `private static Vector<Element> centroids`. Dopodiché verrà eseguito un task *Map* per ogni tupla presente all'interno del dataset e poi controlleremo qual è il centro più vicino al record che stiamo analizzando.

```
for(Element c : centroids){

   //computation of the distance
   dis = Element.distance(c, element);

   //check if it is the lower distance
   if(dis < minDis)
   {
      cent = c;
      minDis = dis;
      index = i;
   }
   i++;
}
```

Infine passeremo al task di *Combine* l'identificatore del centro trovato e la tupla analizzata.

```
//pass value to Combiner
context.write(idx, element);
```

3.4 KMeansCombiner.java

La classe implementa l'interfaccia `Reducer` che fornisce il metodo principale che verrà eseguito come task *Combine* e, come *Mapper*, fornisce metodi per pre e post-elaborazione. L'output di questo task deve essere coerente con il suo input (cioè avere le stesse classi per la coppie k/v in ingresso e uscita) dato che può essere eseguito un numero imprecisato di volte. Permette di raggruppare piccoli insiemi di elementi aventi la stessa chiave e, in questo caso, di calcolare somme parziali da passare al task *Reduce*.

3.5 KMeansReducer.java

Reduce come *Combine* implementa l'interfaccia `Reducer`. In questo caso sfruttiamo entrambi i metodi di pre e post-elaborazione, infatti:

- fase di `setup` (pre-elaborazione): leggiamo i centri dal file sequenziale per confrontarli con i nuovi e controllare la convergenza a meno di una soglia.
- fase di `cleanup` (post-elaborazione): aggiorniamo il file sequenziale con i nuovi centri.

Il metodo di `reduce` che verrà eseguito durante il task ha un funzionamento simile a quello del `Combiner` ma in questo caso non troviamo le somme parziali ma le somme totali su cui eseguire i calcoli per trovare la media e quindi il nuovo centro.
Dopo aver trovato il centro lo confrontiamo con il vecchio e controlliamo che la distanza sia minore della soglia imposta. Se la è per tutti i nuovi centri allora stiamo convergendo, in caso contrario incrementiamo un contatore che verrà letto in `KMeansDriver.java` e che fermerà il ciclo nel caso in cui questo sia zero (vedi sopra).

Contatore:

```
/**
* Counter used to check convergence
*/
public enum CONVERGENCE{
   CONVERGE
}
```

Calcolo e convergenza:

```
newCenter.mean();

//Result desn't converge if distance between old and new centroids
    is greater than threshold
if(Element.distance(OldCenter.get(key.get()), newCenter) > 0.01)
{
   context.getCounter(CONVERGENCE.CONVERGE).increment(1);
}

Centri.put(new IntWritable(iKey), newCenter);

context.write(new IntWritable(iKey), newCenter);
```

4 Funzionamento

Dopo aver effettuato il login al cluster Hadoop `bd301.hpc.unipr.it` creiamo una cartella KMeansProject in cui caricare il dataset, il file Manifest (che indicherà al sistema la classe principale in cui è presente il metodo `main`) e i file di codice, e spostarsi al suo interno:

1. `mkdir KMeansProject`

Poi esportiamo il CLASSPATH:

2. export CLASSPATH="$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.9.1.jar:$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-common-2.9.1.jar:$HADOOP_HOME/share/hadoop/common/hadoop-common-2.9.1.jar:~/KMeansProject/KMeans/*:$HADOOP_HOME/lib/*"

Dopodiché si devono compilare i file di codice e creare il file .jar:

3. `javac -d .  Element.java KMeansMapper.java KMeansCombiner.java KMeansReducer.java KMeansDriver.java`

4. `jar cfm KMeans.jar Manifest.txt KMeans/*.class`

Ora dobbiamo creare nel filesystem di Hadoop la cartella in cui andremo a caricare il dataset:

5. `hdfs dfs -mkdir kmeansDirectory`

6. `hdfs dfs -put "nome_file_dataset" kmeansDirectory`

Per verificare:

- `hdfs dfs -ls kmeansDirectory`

Una volta compilati i file e caricato il dataset possiamo eseguire il programma:

7. `$HADOOP_HOME/bin/hadoop jar KMeans.jar "input_dir" "output_dir" "num_centers" "n_parameters" "loop" "max_num" "split_char"`

($HADOOP_HOME/bin/hadoop jar KMeans.jar kmeansDirectory output_mean 5 3 20 50 t **oppure** $HADOOP_HOME/bin/hadoop jar KMeans.jar kmeansDirectory output_mean 5 3 20 50 \,)

Dove:

- `Input_dir`: cartella del dataset
- `output_dir`: cartella in cui verrà messo il risultato, va indicata ma viene creata automaticamente
- `num_centers`: numero di centri da usare
- `n_parameters`: numero di parametri da usare
- `loop`: numero di cicli che devono essere effettuati
- `max_num`: numero massimo utilizzato per la scelta casuale dei centri (grandezza dataset). Se 0 si usano le prime n righe del dataset
- `split_char`: carattere usato per la concatenazione dei parametri nel dataset, serve per poter separare i dati. Usare 't' nel caso di tabulazioni mentre gli altri caratteri devono essere preceduti da '\' (es. \,)

Quando l'esecuzione è ultimata è possibile scaricare il file contenente i centri e quello contenente il risultato dell'elaborazione:

- `hdfs dfs -get output_mean/part-r-00000` (risulatato elaborazione, file di testo contenente tuple composte dall'identificatore del centro di appartenenza e dalla rispettiva tupla del dataset)

- `hdfs dfs -get centers/cent.txt` (risultato centri)

Prima di rieseguire nuovamente il programma assicurarsi di eliminare il precedente risultato:

- `hdfs dfs -rm -r output_mean`

5 Analisi

L'algoritmo è stato testato utilizzando tre dataset differenti.

5.1 Music rating study survey answer

Dataset che fa parte della collezione di dati Yahoo! Webscope dataset ydata-ymusic-user-artist-ratings-v1_0 [http://research.yahoo.com/Academic_Relations]. In particolare noi abbiamo usato il file ydata-ymusic-rating-study-v1_0-survey-answers.txt contenente le risposte date ad un questionario inerente alla frequenza di recensione di canzoni ascoltate, per un totale di 5400 righe.
Questi dati hanno permesso di testare le funzionalità del programma dato che il numero di parametri richiesto è differente da quello delle altre prove. In questo caso abbiamo infatti 7 parametri (uno per ogni risposta) contro i 3 degli altri dataset. E' stato utile perché ha permesso anche di testare la robustezza del programma, in quanto nel caso di selezione di centri uguali, questi vengono aggregati in un unico centro, andando così a diminuire il numero reale di centroidi selezionati.

Centri	**Valori**
1	4.569 - 4.011 - 2.958 - 2.550 - 4.225 - 4.775 - 1.822
2	4.705 - 4.740 - 4.412 - 3.025 - 4.483 - 4.859 - 1.766
3	4.738 - 1.179 - 1.574 - 2.635 - 4.125 - 4.851 - 1.728
4	4.832 - 4.961 - 4.939 - 4.663 - 4.956 - 4.993 - 1.548

Tabella 1: Risultati test con 5 centroidi richiesti ma due centri iniziali sono uguali

5.2 Mall Customer Segmentation Data

Questo dataset è stato scaricato all'indirizzo https://www.kaggle.com/vjchoudhary7/customer-segmentation-tutorial-in-python (Mall.csv) e contiene informazioni riguardanti un punteggio di spesa assegnato ai clienti di un centro commerciale in base ai loro acquisti. Questi dati però sono stati pre-elaborati per far in modo che presentassero soltanto i valori numerici.
Abbiamo effettuato tre prove.

5.2.1 Test 1

Test generico con tre centri casuali per verificare il funzionamento. Il risultato è il seguente:(Fig. 1)

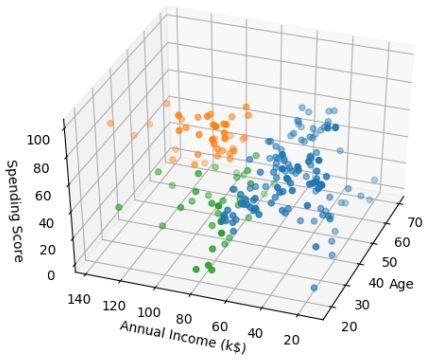

Figura 1: Mall Test 1 - test generico scegliendo tre centri casuali

5.2.2 Test 2

Nel secondo test abbiamo utilizzato 4 centri selezionando una persona di giovane età con basso guadagno annuale e una con alto guadagno annuale, e una persona di maggiore età con con basso guadagno annuale e una con alto guadagno annuale. Abbiamo effettuato più prove notando che il tempo medio di elaborazione per un solo file si aggira sui 125 secondi mentre suddividendo il dataset in più file cresce fino a 298 secondi.
Il risultato ottenuto è questo: (Fig. 2)

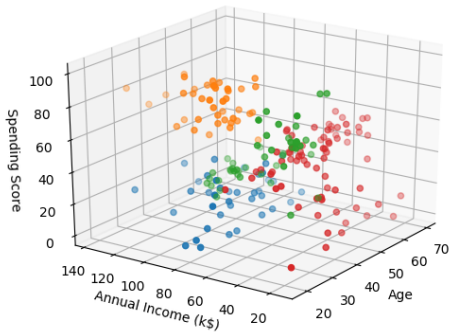

Figura 2: Mall Test 2 - posizionamento dei quattro centri iniziali in testa al dataset ed esecuzione fino a convergenza

5.2.3 Test 3

In questo ultimo test abbiamo usato sempre quattro centri selezionando casualmente un giovane con basso punteggio e uno con alto punteggio ed un signore con basso punteggio e uno con alto punteggio. In questo modo si è potuto notare che mantenendo invariato il numero di centri ma modificandone i valori i cluster risultanti cambiano.(Fig. 3)

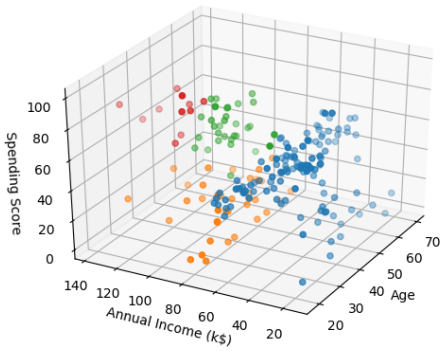

Figura 3: Mall Test 3 - posizionamento dei quattro centri iniziali in testa al dataset ed esecuzione fino a convergenza

5.3 Clustering benchmark datasets - Aggregation

Scaricato sempre da Kaggle come il precedente (https://www.kaggle.com/vasopikof/clustering-benchmark-datasets#Aggregation.txt). In questo caso abbiamo scelto l'insieme di dati Aggregation.txt per effettuare ulteriori test. In uno di questi test sono state scelte le prime cinque tuple come centroidi iniziali e sono state fatte varie prove suddividendo il dataset in più file, notando un miglioramento rispetto al singolo file (Tabella 2).

N. File	Tempo di esecuzione (secondi)
1	359.28
2	307.99
3	307.48
4	311.57

Tabella 2: Aggregation Test 1 - test valutazione tempi di esecuzione suddividendo il dataset in più file scegliendo sempre le prime cinque tuple come centri iniziali ed eseguendo fino alla convergenza

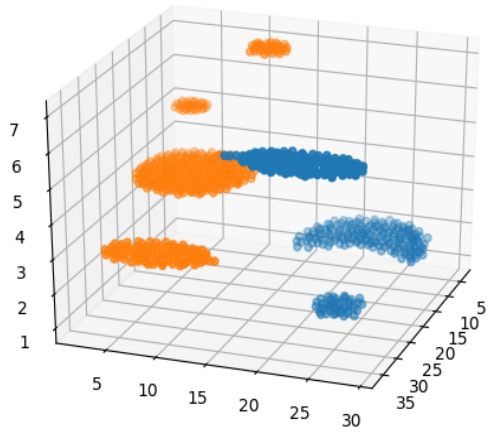

Figura 4: Aggregation Test 1 - due centri scelti casualmente ed esecuzione fino alla convergenza

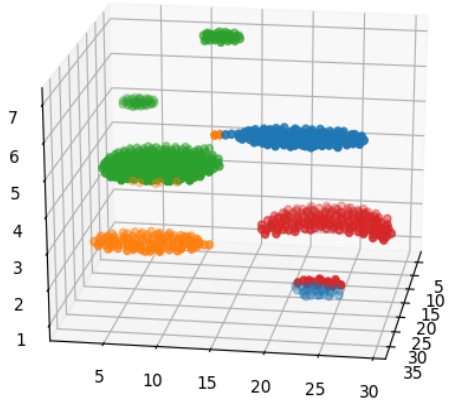

Figura 5: Aggregation Test 2 - quattro centri scelti casualmente ed esecuzione fino alla convergenza

6 Conclusioni

L'utilizzo di Hadoop riduce il tempo di elaborazione dei dataset rispetto ad una elaborazione sequenziale poiché gazie al paradigma MapReduce permette la suddivisione di un file in più segmenti che vengono elaborati in parallelo (Map) ed il loro risultato viene unito grazie ad una ulteriore elaborazione (Reduce) per fornire il risultato finale. Questo procedimento permette di elaborare file di grosse dimensioni in tempi ragionevoli se il sistema è composto dal giusto numero di nodi.
Il numero di nodi del sistema influenza la velocità di esecuzione, più nodi ci sono, maggiori sono le risorse di elaborazione a disposizione. Nel nostro caso abbiamo usato un sistema composto da un NameNode e tre DataNode.