Parallel K-Means

RELAZIONE PROGETTO CLOUD COMPUTING

Emilio Paolini

Filippo Guggino

Leonardo Bacciottini

Luca Barsellotti

Sommario

Introduzione 3

Hadoop 3

- Mapper 3
- Combiner 4
- Reducer 4
- Sampling 5

Spark 7

Performance 8

Introduzione

Obiettivo del progetto è l'implementazione dell'algoritmo **k-means** sui framework **Hadoop** e **Spark**. Il k-means è un algoritmo di **clustering**, una tecnica appartenente alla categoria dell'unsupervised learning, ossia quella classe di problemi di cui non conosciamo a priori l'esistenza di classi tra gli oggetti da raggruppare. In particolare, in un problema di clustering, quello che vogliamo ottenere alla fine sono gruppi omogenei di oggetti, ossia gruppi in cui gli oggetti appartenenti allo stesso gruppo sono "simili" (il concetto di similitudine viene definito dall'algoritmo adottato), mentre oggetti appartenenti a cluster diversi sono "differenti". L'algoritmo k-means utilizza il concetto di distanza per risolvere il problema di clustering: inizialmente vengono estratti casualmente k punti (k è il numero di cluster da creare) che sono chiamati **centroidi**, poi ad ogni iterazione dell'algoritmo ciascun oggetto è assegnato al centroide piú vicino (ogni oggetto viene rappresentato come un vettore di R^N) e i centroidi sono aggiornati in modo che ognuno di essi rappresenti la media, attributo per attributo, degli oggetti appartenenti a tale cluster. I **criteri di stop** che abbiamo utilizzato sono il numero di iterazioni dell'algoritmo e la differenza tra le somme delle distanze quadrate di ogni punto dal suo centroide più vicino , la quale deve essere minore di una determinata soglia. Se quest'ultimo criterio è soddisfatto prima del numero massimo di iterazioni, l'algoritmo viene interrotto, altrimenti continua fino ad arrivare al numero massimo di iterazioni.

Tipicamente l'algoritmo di k-means viene eseguito su una singola macchina che lavora su un dataset locale. Obiettivo del progetto è estendere l'algoritmo base di k-means per fare in modo che possa essere eseguito su Hadoop e Spark, utilizzando **MapReduce**. Nel seguito verrà presentato lo pseudocodice del **mapper**, del **combiner** (usato per migliorare le performance) e del **reducer**, come è stata effettuata l'**estrazione casuale** dei primi k centroidi in modalitá distribuita, la sua implementazione su Hadoop e Spark e infine il confronto tra i due framework.

Hadoop

Mapper

Input: centroidi, <chiave,valore>=<offset dell'oggetto nel file, punto rappresentante l'oggetto>

Output: <chiave',valore>= <indice del cluster piú vicino, punto rappresentante l'oggetto>

Codice:

```
1: minDistance = +inf, index =-1
2: for centro : centroidi then
      distanzaCorrente = distanzaEuclidea(punto,centroidi(i))
      if distanzaCorrente<minDistance then
            minDistance = distanzaCorrente
```

```
            index = i
        endif
    endfor
3: emetti <index,punto>
```

Combiner

Input: <chiave,valore>=<indice del cluster, lista di punti assegnati al cluster>

Output: <chiave',valore'>= <indice del cluster, (somma dei punti, numero di punti)>

Codice:

```
1: crea un nuovo punto newCentroid con coordinate 0
2: numeroDiPunti = 0
3: for punto : lista di punti then
      newCentroid = newCentroid + punto (somma dimensione a dimensione)
      numeroDiPunti ++
    endfor
4: index = chiave
5: emetti <index, (newCentroid, numeroDiPunti)>
```

Reducer

Input: <chiave,valore>=<indice del cluster, lista di somme parziali del cluster>

Output: <chiave,valore>= <indice del cluster, nuovo centroide>

Codice:

```
1: crea un nuovo punto newCentroid con coordinate 0
2: numeroTotaleDiPunti = 0
3: for elemento : lista somma parziali then
      newCentroid += elemento[punto] //(somma dimensione a dimensione)
      numeroTotaleDiPunti += elemento[numeroPunti]
    endfor
4: newCentroid = newCentroid / numeroTotaleDiPunti // effettua la divisione di ogni     dimensione del newCentroid per il numero di punti del cluster
```

```
5: index = chiave
6: emetti <index, newCentroid>
```

Sampling

Per effettuare l'estrazione dei primi k centroidi, è stato utilizzato un algoritmo di sampling noto come **Reservoir Sampling**. Tale algoritmo viene utilizzato per campionare *k* elementi a caso da un insieme di *n* elementi, dove *n* non è noto (infatti essendo il nostro dataset distribuito tra i vari nodi del cluster tramite hdfs, nessun nodo sa a priori la dimensione del dataset). È possibile dimostrare che in tale algoritmo ogni elemento viene estratto con probabilità *k/n*. Per fare ciò tipicamente l'algoritmo di sampling effettua i seguenti passaggi:

Input: k, insieme di oggetti x1, x2, x3…

Output: k elementi

Codice:

```
1: crea un insieme vuoto ExtractedSet
2: copia i primi k elementi in ExtractedSet
3: for i=k+1,k+2,k+3… then
      Estrai un numero casuale rand nell'intervallo (0,i)
      if rand<k then
            ExtractedSet[rand] = oggetto[i]
      endif
    endfor
```

Questo codice peró va bene solamente se il dataset si trova solamente su una macchina, mentre se il dataset è distribuito, dobbiamo implementare una versione distribuita di tale algoritmo. Ció è stato fatto utilizzando MapReduce; ancora una volta, per motivi di ottimizzazione è stato utilizzato un combiner. Gli pseudo codici dei task Map, Combine e Reduce sono i seguenti:

Mapper:

Input: k,<chiave,valore>=<offset dell'oggetto nel file, stringa contenente un punto>

Output: <chiave,valore>=<numero estratto, (stringa contenente il punto, offset)>

Codice:

```
1: if offset<k then
      randomValue = offset
    else
      randomValue = rand(0,offset)
```

```
    endif
2: if randomValue<k then
      emetti <randomValue, (stringa contenente il punto, offset)>
    endif
```

Combiner/Reducer

Input: k,<chiave,valore>=<numero estratto, lista di (stringhe,offset) con lo stesso numero estratto>

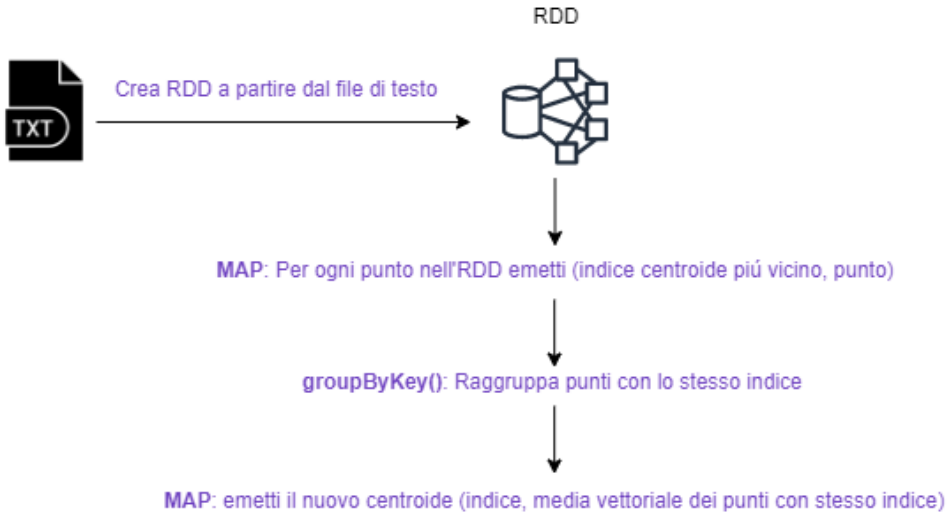
Output: <chiave,valore>=<numero estratto, (stringa contenente il punto, offset)>

Codice:

```
1: (stringaTmp, offsetTmp) = NULL
2: for (stringa,offset) : lista di stringhe di (punti,offset) then
      if (stringaTmp,offsetTmp) == NULL then
            (stringaTmp,offsetTmp) = (stringa, offset)
       endif
      if offset > offsetTmp then
            (stringaTmp,offsetTmp) = (stringa, offset)
       endif
    endfor
3: emetti <numero estratto, (stringaTmp, offsetTmp)>
```

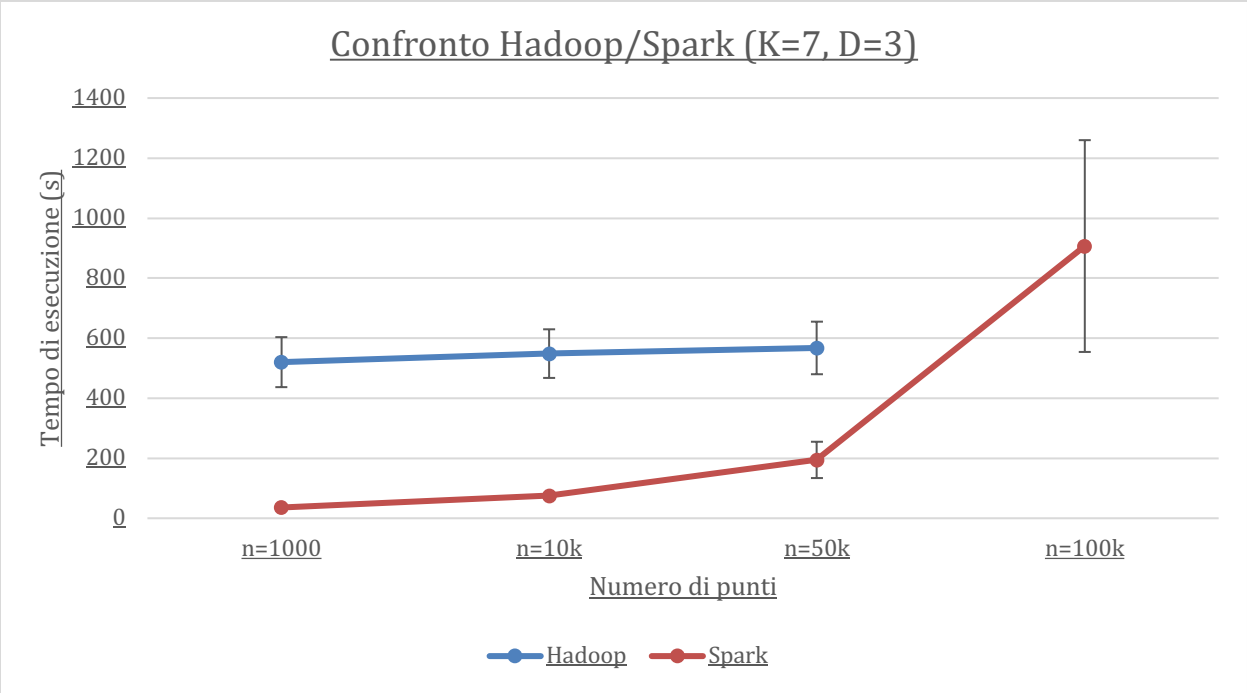
Spark

Per quanto riguarda l'implementazione su Spark, viene mostrata un'idea dell'applicazione attraverso la seguente immagine:



Per prima cosa viene creato il **Resilient Distributed Dataset (RDD)** utilizzando il metodo *textFile* dello *SparkContext*. Successivamente viene anche creato un sampling dei punti, in particolare vengono estratti *k* punti che rappresenteranno i centroidi di partenza. Dopodiché, viene applicata una prima *Map* che emette per ogni punto presente nel dataset, l'indice del centroide ad esso piú vicino ed il punto stesso. Poi i punti aventi lo stesso indice, vengono raggrupati insieme dalla *groupByKey* che quindi andrá a creare tuple del tipo (indice centroide, lista di punti). Infine, ognuna di queste tuple è passata ad un altro mapper che andrá a creare i nuovi centroidi, dati dalla media dei punti della tupla.

Performance



In questo grafico possiamo vedere un confronto tra le prestazioni di Hadoop e Spark nello stesso tipo di configurazione, con una dimensione dell'input variabile.

Non abbiamo riportato i grafici per le altre configurazioni testate (D=7, K=13) perché estremamente simili a questo, che esprime chiaramente alcuni risultati.

Le barre di errore visibili nel grafico sono intervalli di confidenza al 95%, ottenuti da 10 ripetizioni dell'algoritmo.

La prima osservazione riguarda la serie relativa ad Hadoop: si nota chiaramente che il tempo di esecuzione non varia rispetto alla dimensione degli input considerati. Il motivo di ciò è che l'overhead per la coordinazione e la distribuzione dell'algoritmo occupa un tempo maggiore rispetto alla risoluzione stessa, annullando di fatto il vantaggio.

Abbiamo visto che il tempo di esecuzione aumenta molto invece per N=100k in Hadoop, ma non abbiamo potuto riportarlo in grafico perché le risorse limitate delle macchine hanno reso impossibile l'esecuzione di questa configurazione per un numero adeguato di campioni.

In Spark invece osserviamo che in generale i tempi di esecuzione sono sempre inferiori (e di molto) a quelli di Hadoop, e che in particolare questi tendenzialmente aumentano in modo lineare con la dimensione dell'input (nel grafico lo si vede dal fatto che la curva cresce esponenzialmente come l'input). Ovviamente i dati a disposizione sono pochi per trarre delle conclusioni definitive, ma il risultato è soddisfacente in quanto k means è un algoritmo di complessità non polinomiale, e con questa soluzione parallela e iterativa siamo in grado di trovare una soluzione quasi ottima in tempi lineari, il che lo rende perfettamente utilizzabile in pratica.