PC-2019/20 Implementazione dell'algoritmo K-means in Hadoop e OpenMP

Matteo Marulli
E-mail address
matteo.marulli@stud.unifi.it

Matteo Gemignani
E-mail address
matteo.gemignani@stud.unifi.it

Abstract

L'algoritmo di clustering K-means è uno dei più popolari algoritmi di cluster analysis usato per scoprire gruppi nei dati nel data mining e machine learning. Non è infrequente usare il Kmeans su grandi quantità di dati e in questo report proponiamo due versioni di K-means scritta una in OpenMP e l'altra in Hadoop. Per ogni versione descriveremo l'implementazione e l'analisi delle prestazioni.

1. Introduzione

Il clustering è un area dell'unsupervised learning usato nel data mining come tecnica di analisi esplorativa dei dati molto utile quando i dati del dataset sono privi di una label.

Gli algoritmi che fanno parte di questa area ci consentono di organizzare i dati all'interno di gruppi senza avere alcuna precedente conoscenza delle appartenenze dei dati a tali gruppi.

I dati che finiscono in un gruppo sono molto simili tra di loro ma diversi dai dati che sono presenti in un altro gruppo.

Tra gli algoritmi di clustering il più famoso è senz'altro il K-means, la sua popolarità è dovuto alla sua semplicità e anche alla sua capacità di raggiungere buoni risultati nella pratica.

Non è raro usare il K-means su grandi quantità di dati per questo partiremo dallo pseudo codice dell'algoritmo per analizzarlo e da questa analisi ricaveremo le implementazioni di OpenMP e Hadoop.

1.1. L'algoritmo

In questa sezione è riportato lo pseudo-codice del K-means.

Algorithm 1: K-means

Input: $\mathbb{D} \in \mathrm{R}^{n \times d}, k \in \mathrm{N} - \{0, 1\}, \epsilon \in \mathrm{R}$
$t = 0$
inizializza in modo casuale k centroidi iniziali: $\boldsymbol{\mu}_1^t, \boldsymbol{\mu}_2^t, ..., \boldsymbol{\mu}_k^t \in \mathrm{R}^d$
repeat
  $t \leftarrow t + 1$
  $\mathrm{C}_j \leftarrow \emptyset$ per ogni $j = 1, \cdots, k$
  `//Assegnamento ai cluster`
  foreach $\boldsymbol{x}_j \in \mathbb{D}$ **do**
    $j^* \leftarrow \text{argmin}_i \{||\boldsymbol{x}_j - \boldsymbol{\mu}_j^t||^2\}$
    `// Si assegna` $\boldsymbol{x}_j$ `al centroide più vicino`
    $\mathrm{C}_{j^*} \leftarrow \mathrm{C}_{j^*} \cup \{\boldsymbol{x}_j\}$
  end
  `// Aggiornamento centroidi`
  foreach $i = 1..., k$ **do**
    $\boldsymbol{\mu}_i^t \leftarrow \frac{1}{|\mathrm{C}_j|} \sum_{\boldsymbol{x}_j \in \mathrm{C}_i} \boldsymbol{x}_j$
  end
until $\sum_{i=1}^{k} ||\boldsymbol{\mu}_i^t - \boldsymbol{\mu}_i^{t-1}||^2 \leq \epsilon$;

Le parti più critiche sono la scelta dei centroidi iniziali, l'assegnamento di un punto ad

un cluster e l'aggiornamento dei centroidi.

Le tre operazioni saranno implementate in OpenMP e Hadoop.

1.2. Implementazione in OpenMP

Fra l'implementazione sequenziale e quella parallela non ci sono differenze tranne l'impostazione del numero dei Threads nel main che, nel caso dell'implementazione sequenziale viene impostata a 1.

1.2.1 Inizializzazione centroidi

Per OpenMp è stato scelto di creare i centroidi iniziali con questa strategia:

```
void KmeansOpenMP::initCentroids(double**
     dataMatrix, int nRow,int nCol){
   centroids = new double *[k];

   for (int i = 0; i < k; i++) {
      centroids[i] = new double[nCol];
   }
   srand(seed);
   for(int i = 0;i<k;i++){
      int row = int(rand()%nRow);
      for(int j=0;j<nCol;j++){
         centroids[i][j]= dataMatrix[row
             ][j];
      }
   }
}
```

`initCentroids()` sceglie casualmente k righe della matrice dei dati iniziale e queste saranno i centroidi di partenza.

Il costo di `initCentroids()` è $\mathbf{O}(kd)$.

1.2.2 Assegnamento cluster

L'assegnamento di un punto ad un cluster è uno dei passi più costosi dell'algoritmo. Per ogni punto bisogna calcolare k distanze, cioè una per ogni centroide, quindi il costo di questa operazione diventa $\mathbf{O}(n\dot{k})$.

```
int KmeansOpenMP::clusterAssignment(
    double* point, double** centroids,
    int nCol){
   int index = 0;
   double distance = 0;
   double oldDistance = DBL_MAX;
   for(int i =0; i<k;i++){
      double* centroid_i = centroids[i];
      distance = getDistance(point,
          centroid_i, nCol);
      if(distance<oldDistance){
         index = i;
         oldDistance = distance;
      }
   }
   return index;
}
```

Nel codice `clusterAssignment()` viene invocata la funzione che calcola la distanza tra due punti che ha costo $\mathbf{O}(d)$, pertanto il costo dell'intero passaggio di assegnamento di un punto ad un cluster costa $\mathbf{O}(knd)$

```
double KmeansOpenMP::getDistance(double *
    point, double *centroid, int nCol) {
   double distance = 0;
   for (int i = 0; i < nCol; i++) {
      distance += pow(point[i] - centroid
          [i], 2);
   }
   return sqrt(distance);
}
```

Usando OpenMp la funzione clusterAssignment è stata parallelizzata nel seguente modo

```
#pragma omp parallel for schedule(static,
    nRow/nThreads) default(none) shared(
    labels) firstprivate(nThreads,
    centroids,nCol,nRow,dataset)
      for (int indexPoint = 0; indexPoint
          <nRow; indexPoint++){
         labels[indexPoint] =
             clusterAssignment(dataset[
             indexPoint], centroids, nCol
             );
      }
```

Il codice è stato parallelizzato dividendo in modo equo il carico di lavoro tra i thread in modo statico.

I thread condividono l'array `labels` per riporre il risultato dell'assegnamento al cluster per ogni punto.

Inoltre, i thread, hanno una copia locale, inizializzata al valore del thread principale, delle variabili che identificano il numero dei

threads, i centroidi, il numero righe, il numero di colonne, e il dataset.

Per fare ciò è stata utilizzata la direttiva `fristprivate` cosi da evitare una copia locale inizializzata a 0 portando i thread ad non entrare nei cicli `for`.

1.2.3 Aggiornamento centroidi

L'aggiornamento dei centroidi è stato parallelizzato in base al numero di cluster:

```
void KmeansOpenMP::centroidUpdate(int *
    labels,int* labelsCount,double **
    centroids,double **dataset,int nCol,
    int nRow){
   int pointForCluster;
#pragma omp parallel default(none) shared
    (centroids,labelsCount) firstprivate
    (nCol,nRow,labels,dataset)
{
#pragma omp for collapse(2)
   for(int c=0; c<k;c++){
      for(int i=0;i<nCol;i++){
         centroids[c][i]=0; // azzera il
             centroide c-esimo
      }
   }
#pragma omp for
   for(int i=0;i<k;i++){
      labelsCount[i]=0;
   }

#pragma omp for
   for(int c=0;c<k;c++){
      for(int j=0;j<nRow;j++){
         if(c==labels[j]){
            labelsCount[c]++; // conta il
                numero di elementi nel
               cluster c-esimo
            for(int i=0;i<nCol;i++){
               centroids[c][i]+=dataset[j
                   ][i]; // aggiornamento
                    centroide c-esimo
                   componente per
                   componente
            }

         }
      }
   }
#pragma omp for collapse(2)
   for(int c=0; c<k;c++){
      for(int i=0;i<nCol;i++){
         centroids[c][i]=centroids[c][i]/
             labelsCount[c]; // dividi
             ogni componente per il
             numero di oggetti nel
             cluster
      }
   }
}
}
```

Purtroppo non è stato possibile parallelizzare l'istruzione più costosa(Ciclo `for` Rig. 18 del codice precedente) causa di una dipendenza dovuta da `labelsCount[c]++`.

1.3. Implementazione in Hadoop

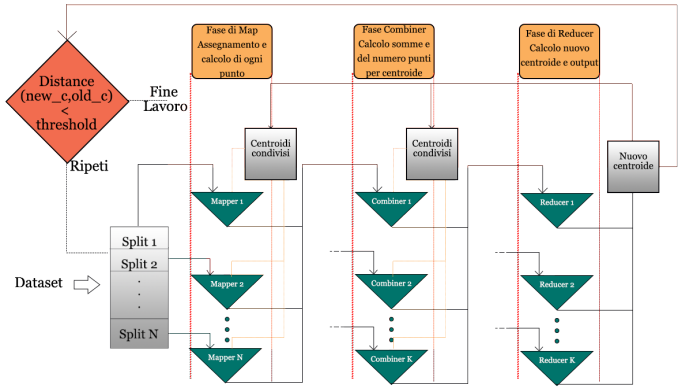

Figure 1. Flowchart dell'implementazione di K-means in Hadoop

Per Hadoop è stata scelta un'altra strategia di inizializzazione dei centroidi

```
private void initCentroids() {
      for(int i=0;i<this.k;i++){
         double[] center= new double[this
             .nb_dimensions];
         for(int j=0;j<this.nb_dimensions
             ;j++)
            center[j]=Math.floor(
                -1000+2000*random.
                nextDouble())/100 ;
         this.centroids.add(center);
      }
   }
```

Questa volta, invece di estrarre casualmente dei punti dal dataset, è stato deciso di generarli in modo aleatorio, evitando di leggere tutto il

dataset in modo sequenziale.

Per quanto riguarda le operazioni di assegnamento di un punto ad un cluster e l'aggiornamento dei centroidi, sono state rivisitate in modo da applicare il paradigma di programmazione di Hadoop: il "MapReduce", come mostrato in figura 1.

Il metodo di Map processa i punti del dataset in modo parallelo per coppie `<key, value>`, ugualmente i metodi Reduce e Combine lavorano in parallelo le coppie `<key, List<value>>` dove questa volta le chiavi saranno i centroidi e avranno associate delle liste di punti.

Per i tipi di `key` e `value` sono state usate le classi messe a disposizione da Hadoop, evitando la scrittura di una possibili nuove classi, `Point` e/o `Centroids`, che avrebbero dovuto implementare l'interfaccia `WritableComparable` e suoi relativi metodi.

1.3.1 Map

```
@Override
  public void map(Object key, Text value
      , Context context) throws
      IOException, InterruptedException
      {
    String line = value.toString();
    List<Double> coordinates = new
        ArrayList<Double>();
    StringTokenizer tokenizer = new
        StringTokenizer(line, ",");
    while(tokenizer.hasMoreTokens())
       coordinates.add(Double.
           parseDouble(tokenizer.
           nextToken()));

    double distance = Double.MAX_VALUE;
    double distanceTmp;
    long index = -1;
    for(int i = 0; i<centroids.size();i
        ++){
       distanceTmp = getDistance(
           coordinates, centroids.get(i
           ));
       if(distanceTmp<distance){
          distance = distanceTmp;
          index= i;
       }
    }
   if(index!=-1)
       context.write(new LongWritable(
           index), value);
   else
       logger.fatal("\n\nNessun cluster
           vicino trovato? min = "+
           distance+" coordinate = "+
           coordinates+"\n\n\n");
  }
```

Il metodo `map` esegue l'assegnamento di un punto ad cluster come la funzione `clusterAssignment` di OpenMP. Come possiamo vedere inizialmente il punto è salvato in una variabile `Text` e quindi per per essere utilizzato, come un vettore di numeri reali, deve essere letto e trasformato in un `Double`.

Poi, esattamente come in C++, si calcola la distanza tra i centroidi e il punto estratto, scegliendo il centroide più vicino.

Infine, si passa il risultato al Combine scrivendo sul file `contex` l'indice del centroide più vicino e la lista delle coordinate del punto.

1.3.2 Combiner

La classe Combiner ha lo scopo di ridurre il trasferimento dati da Map a Reduce dal momento che i dati sono salvati sul disco locale dell'host.

Il Combiner prende una la lista di punti associati ad un cluster e ne calcola la somma parziale delle coordinate e la quantità.

```

  public static int nb_dimension;

  @Override
  public void reduce(LongWritable key,
      Iterable<Text> values, Context
      context) throws IOException,
      InterruptedException {
     double[] sum = new double[
         nb_dimension];
    int numPoint = 0;

    for(int i = 0; i<nb_dimension;i++)
```

```
            sum[i] = 0;

        for(Text record: values){
            String line = record.toString();
            List<Double> coordinates = new
                ArrayList<Double>();
            StringTokenizer tokenizer = new
                StringTokenizer(line, ",");
            while(tokenizer.hasMoreTokens())
                coordinates.add(Double.
                    parseDouble(tokenizer.
                    nextToken()));

            for(int i = 0; i<nb_dimension;i
                ++)
                sum[i] += coordinates.get(i);
            numPoint++;
        }
        StringBuilder stringBuilder = new
            StringBuilder();
        for(int i = 0; i<nb_dimension;i++)
            {
            stringBuilder.append(sum[i]);
            stringBuilder.append(",");
        }
        stringBuilder.append(numPoint);
        context.write(key, new Text(
            stringBuilder.toString()));

    }
```

Infine passa al Reduce, sempre scrivendo sul file `context`, una coppia chiave valore, dove la chiave è ancora il centroide e il valore è un testo con la semisomma della cordinate e il numero di punti associati al centroie.

1.3.3 Reduce

Il Reduce calcola i nuovi centroidi sommando separatamente tutte le somme parziali delle coordinate e quelle dei numero di punti associati al centroide.

Fatto ciò, per ottenere il valore del nuovo centroide, il Reducer divide la somma ottenuta per il numero totale dei punti nel cluster.

Infine scrive sul file `context` la coppia indice del centroide e coordinate del centroide.

```
public static int nb_dimension;
    public static List<double[]> centroids
        = new ArrayList<>();
    public static DecimalFormat dFormater
        = new DecimalFormat("#.###");

    @Override
    public void reduce(LongWritable key,
        Iterable<Text> values, Context
        context) throws IOException,
        InterruptedException {
        double[] center = new double[
            nb_dimension];
        int numPoint = 0;

        for(int i = 0; i<nb_dimension;i++)
            center[i] = 0;

        for(Text record: values){
            String line = record.toString();
            List<Double> coordinates = new
                ArrayList<Double>();
            StringTokenizer tokenizer = new
                StringTokenizer(line, ",");
            for(int i = 0; i<nb_dimension;i
                ++)
                center[i] += Double.
                    parseDouble(tokenizer.
                    nextToken());
            numPoint += Integer.parseInt(
                tokenizer.nextToken());
        }

        for(int i = 0; i<nb_dimension;i++)
            center[i] /= numPoint;

        StringBuilder center_sb = new
            StringBuilder();
        for(int i = 0; i< nb_dimension;i++)
            {
            center_sb.append(dFormater.
                format(center[i]));
            if(i< nb_dimension-1)
                center_sb.append(" ");
        }
        centroids.set((int)key.get(),
            center);
        context.write(key, new Text(
            center_sb.toString()));
    }
```

1.3.4 Driver

La classe driver contiene il metodo `run` che permette di eseguire il Kmeans.

Il metodo `initJob` contiene la configurazione del Job che determina le istruzioni che

permettono l'esecuzione del processo di MapReduce.

```
1   private void run() throws IOException,
        ClassNotFoundException,
        InterruptedException {

3       do{
            centroids.clear();
5           centroids.addAll(UpdateCentroids
                .centroids);
            ClusterAssignment.centroids.
                clear();
7           ClusterAssignment.centroids.
                addAll(centroids);
            UpdateCentroids.centroids.clear
                ();
9           UpdateCentroids.centroids.addAll
                (centroids);

11          initJob();
            deleteOutputDirIfExists();
13
            if(!job.waitForCompletion(true))
15              logger.info("job failed");
            n_iter++;
17
            logger.info("\n#iter = "+n_iter+
                "\n");
19          logger.info("\tOld_centers: "+
                centroidsToString(centroids)
                +"\n");
            logger.info("\tNew_centers: "+
                centroidsToString(
                UpdateCentroids.centroids)+"
                \n");
21
        }while(compare(centroids,
            UpdateCentroids.centroids));
23      logger.info("Finished in "+ n_iter
            + " iterations");
    }
```

Infine, nel metodo `run`, viene effettuata una "pulitura" delle strutture dati `static` che contengono i centroidi nei metodi Map e Reduce; vengono svuotate ad ogni iterazione usando il metodo clear() per far posto ai nuovi centroidi che vengono aggiunti con l'istruzione addAll().

1.4. Avvio del programma in Hadoop

Per avviare il programma da terminale bisogna passare gli argomenti nel seguente ordine:

1. path dell'input
2. path dell'output
3. k che indica il numero di cluster desiderato
4. ϵ che indica la tolleranza

2. Analisi dei risultati

2.1. Hardware

Per condurre le analisi delle prestazioni è stata usata una macchina MSI con le seguenti specifiche:

- **Processore**: Intel® Core™ i7-10750H CPU @ 2.60GHz 6 core
- **Ram**: 16GB DDR4 3200Mhz
- **Hardisk**: SSD 1TB

2.2. Dataset di prova

Per testare il codice, è stato creato un dataset artificiale usando sklearn. Il metodo utilizzato è `make_blob` ed con i seguenti parametri:

- `n_samples`= 10.000.000;
- `n_features`= 10;
- `centers`= 10;

2.3. Parallelo Vs sequenziale

Viene riportata soltanto l'analisi delle prestazioni OpenMP in quanto per Hadoop servirebbero molti dati per poter osservare uno speedUP significativo.

Come è possibile vedere dalla Figura 2 la versione OpenMP si mantiene allineata con lo speedUP lineare fino a 3 thread per poi man mano distanziarsi leggermente fino a 6 thread.

Dopo 6 thread diverge completamente diventando sub-lineare, si può quindi concludere

che l'implementazione è al massimo 6 volte più veloce della versione sequenziale.

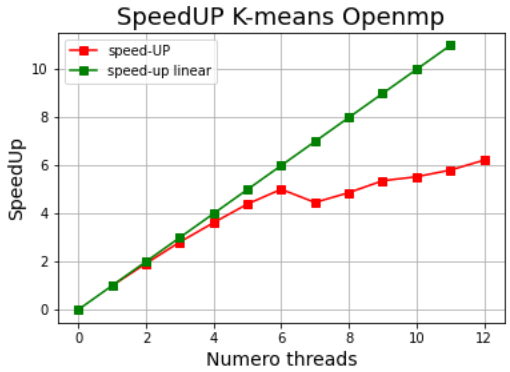

Figure 2. SpeedUP della versione OpenMP di Kmeans