Implementazione dell'algoritmo K-means in Hadoop e OpenMP

Esame di PC

2°Parte

Cos'è K-means

- Il clustering è un'area dell'unsupervised learning usato nel data mining come tecnica di analisi esplorativa dei dati.
- Tra gli algoritmi di clustering il più famoso è senz'altro il K-means, la sua popolarità è dovuta alla sua semplicità e anche alla sua capacità di raggiungere buoni risultati nella pratica.
- Analizzando lo pseudo codice dell'algoritmo si ricaveranno le implementazioni in OpenMP e Hadoop.

Cos'è K-means

- Le parti più critiche sono:
  - la scelta dei centroidi iniziali;
  - l'assegnamento di un punto ad un cluster ;
  - l'aggiornamento dei centroidi.
- Le tre operazioni saranno implementate in OpenMP e Hadoop.

ALGORITHM 13.1. K-means Algorithm

K-MEANS $(\mathbf{D}, k, \epsilon)$:

1 $t = 0$
2 Randomly initialize k centroids: $\boldsymbol{\mu}_1^t, \boldsymbol{\mu}_2^t, \ldots, \boldsymbol{\mu}_k^t \in \mathbb{R}^d$
3 **repeat**
4 $\quad t \leftarrow t + 1$
5 $\quad C_j \leftarrow \emptyset$ for all $j = 1, \cdots, k$
$\quad$ `// Cluster Assignment Step`
6 $\quad$ **foreach** $\mathbf{x}_j \in \mathbf{D}$ **do**
7 $\quad\quad j^* \leftarrow \arg\min_i \left\{ \left\| \mathbf{x}_j - \boldsymbol{\mu}_i^t \right\|^2 \right\}$ `// Assign` $\mathbf{x}_j$ `to closest centroid`
8 $\quad\quad C_{j^*} \leftarrow C_{j^*} \cup \{\mathbf{x}_j\}$
$\quad$ `// Centroid Update Step`
9 $\quad$ **foreach** $i = 1$ *to* k **do**
10 $\quad\quad \boldsymbol{\mu}_i^t \leftarrow \frac{1}{|C_i|} \sum_{\mathbf{x}_j \in C_i} \mathbf{x}_j$
11 **until** $\sum_{i=1}^{k} \left\| \boldsymbol{\mu}_i^t - \boldsymbol{\mu}_i^{t-1} \right\|^2 \leq \epsilon$

Implementazione 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.

Inizializzazione centroidi

- initCentroids() sceglie casualmente k righe della matrice dei dati iniziale e queste saranno i centroidi di partenza.I
- l costo di initCentroids() è O(kd).

```
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];
        }
    }
}
```

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.
- Il costo di questa operazione diventa O(kn).

```
#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);
    }
```

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

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

Aggiornamento centroidi

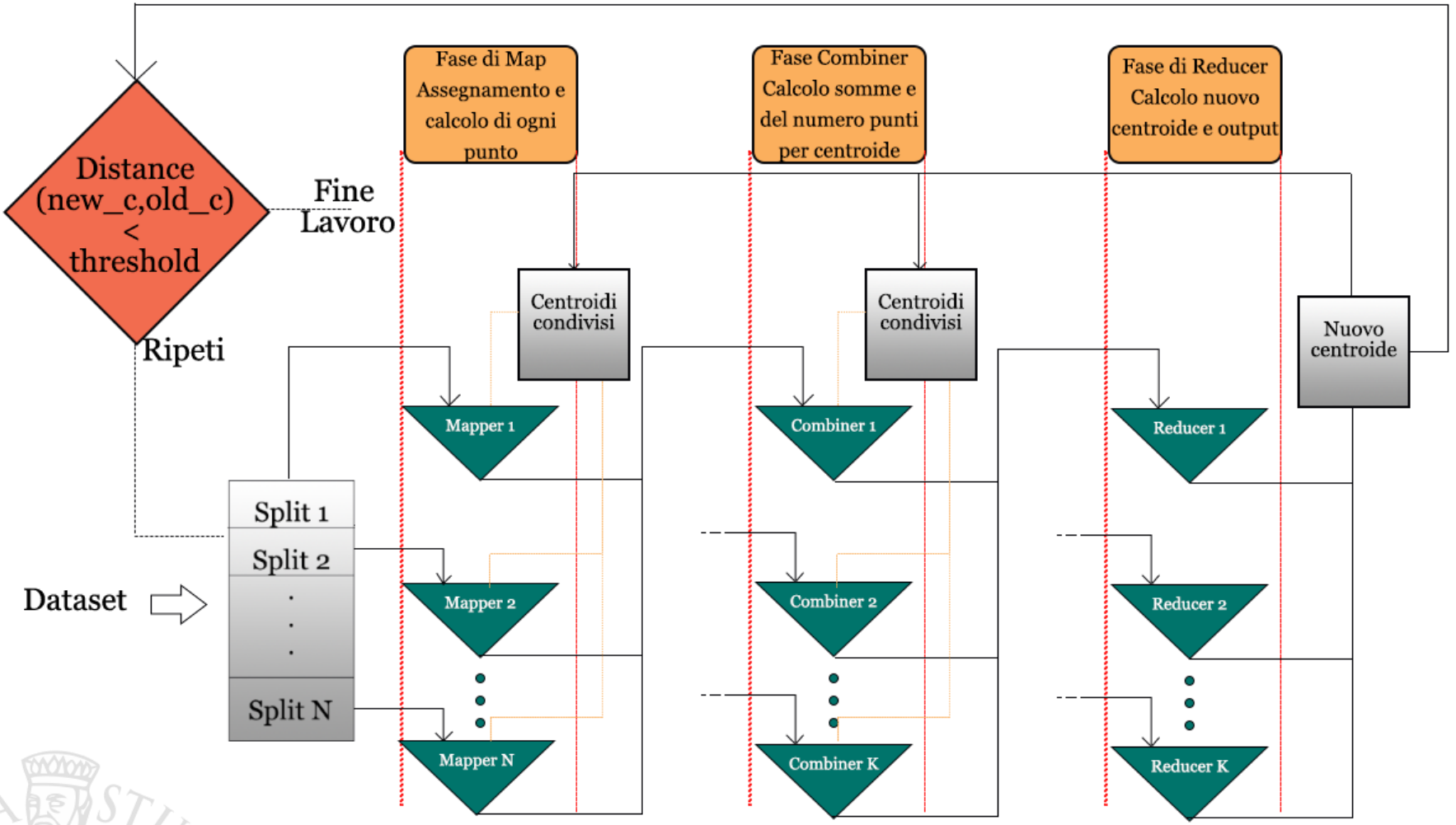
```
void KmeansOpenMP::centroidUpdate(int *labels,int* labelsCount,double **centroids,double **dataset,int nCol, int nRow){
    int pointForCluster;
#pragma omp parallel default(none) shared(centroids,labelsCount) firstprivate (nCol,nRow,labels,dataset)
    {
#pragma omp for collapse(2)
        for(int 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 causa di una dipendenza dovuta da labelsCount[c]++

Implementazione Hadoop



Inizializzazione centroidi

- 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.

```
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);
    }
}
```

Assegnamento cluster

- 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’’
- Il metodo di Map processa i punti del dataset in modo parallelo per coppie <*key, value*>
- *value* sarà un punto del dataset il quale verrà restituito alla Classe Combiner con il centroide associato.
- Per i tipi di \texttt{key} e \texttt{value} sono state usate le classi messe a disposizione da Hadoop.

Assegnamento cluster

```
@Override
public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    String line = value.toString();
    List<Double> coordinates = new ArrayList<~>();
    StringTokenizer tokenizer = new StringTokenizer(line, s1: ",");
    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");
}
```

Aggiornamento centroidi

- L' aggiornamento dei centroidi è stato demandato alle classe **Reduce** e **Combiner**
- Loro lavorano in parallelo le coppie *<key, List<value>>* dove questa volta le chiavi saranno i centroidi e avranno associate delle liste di punti.
- Il **Combiner** prende una la lista di punti associati ad un cluster e ne calcola la somma parziale delle coordinate e il numero di punti associati.
- Il **Reduce** calcola i nuovi centroidi sommando separatamente tutte le somme parziali delle coordinate e quelle dei numero di punti associati al centroide poi, per ottenere il valore del nuovo centroide, divide la somma ottenuta per il numero totale dei punti nel cluster.

Combiner

```
@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<~>();
        StringTokenizer tokenizer = new StringTokenizer(line, s1: ",");
        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()));

}
```

Reduce

```
@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<~>();
        StringTokenizer tokenizer = new StringTokenizer(line, s1: ",");
        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()));
}
```

Driver

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

    do{
        centroids.clear();
        centroids.addAll(UpdateCentroids.centroids);
        ClusterAssignment.centroids.clear();
        ClusterAssignment.centroids.addAll(centroids);
        UpdateCentroids.centroids.clear();
        UpdateCentroids.centroids.addAll(centroids);

        initJob();
        deleteOutputDirIfExists();

        if(!job.waitForCompletion( verbose: true))
            logger.info("job failed");
        n_iter++;

        logger.info("\n#iter = "+n_iter+"\n");
        logger.info("\tOld_centers: "+ centroidsToString(centroids)+"\n");
        logger.info("\tNew_centers: "+ centroidsToString(UpdateCentroids.centroids)+"\n");

    }while(compare(centroids,UpdateCentroids.centroids));
    logger.info("Finished in "+ n_iter + " iterations");
}
```

Analisi delle performance

- Per condurre le analisi delle prestazioni è stata usata una macchina MSI con le seguenti specifiche hardware:
  - Processore: Intel® Core™ i7-10750H CPU @ 2.60GHz 6 core
  - Ram: 16GB DDR4 3200Mhz
  - Hardisk: SSD 1TB

Analisi delle performance

- 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;

Speedup

- Come è possibile vedere dalla figura 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.