K-Means clustering con Apache Hadoop MapReduce

Filippo Mameli
filippo.mameli@stud.unifi.it
6222254

Abstract

La crescita dei volumi di dati da processare, ha reso necessario lo sviluppo di tecniche efficienti per la parallelizzazione dei metodi per l'analisi dei gruppi(clustering). In questo documento si presenta l'algoritmo di clustering K-Means utilizzando il modulo MapReduce del framework Apache Hadoop. Lo strumento migliora la scalabilità e le prestazioni dell'algoritmo su dati di grandi dimensioni distribuendo il carico computazionale e gli elementi da analizzare sull'hardware a disposizione.

Permessi di distribuzione

L'autore di questa relazione permette che questo documento possa essere distribuito a tutti gli studenti UNIFI dei corsi futuri.

1. Introduzione

Insieme allo sviluppo dell'information technology, è aumentata anche la quantità di dati da processare. In molti casi risulta impossibile assumere che una singola macchina possa mantenere nella memoria principale tutti i dati da analizzare. Per questo problema sono stati creati nuovi metodi per processare insiemi di dati contenenti numerosi oggetti.

L'analisi dei gruppi (o clustering) ha delle tecniche che si fondano su assunzioni inattuabili su dataset di grandi dimensioni, per questo motivo uno dei più noti algoritmi di clustering K-Means è stato modificato basandosi sul modello di MapReduce, in modo tale da risolvere i problemi legati al volume di dati.

In questo documento si presenta una possibile implementazione di K-Means basata su MapReduce utilizzando il framework Apache Hadoop. Nella Sezione 2 si introduce l'algoritmo generico di clustering per poi analizzare nella Sezione 3 la versione basata su MapReduce. Nella Sezione 4 si mostrano invece i risultati derivanti dai test.

2. K-Means

K-Means è un algoritmo di clustering molto noto. Questo prende in input un parametro K e un numero n di oggetti. L'obiettivo dell'algoritmo è quello di dividere gli n oggetti in K cluster in modo tale da massimizzare l'omogeneità degli elementi appartenenti ad un cluster. La similarità ad un cluster viene calcolata utilizzando il centroide. Questo è ricavato dalla media degli elementi appartenenti al cluster.

Il primo passo dell'algoritmo è la creazione di K centroidi in modo casuale. Questi rappresentano i centri dei cluster.

Ciascuno degli n oggetti viene assegnato al centroide più simile calcolando la distanza tra l'oggetto e il centro del cluster. Dopo l'assegnazione, i centri vengono ricalcolati utilizzando la media degli elementi del cluster.

L'algoritmo continua a iterare sui nuovi centri e si ferma quando un determinato criterio di arresto viene soddisfatto.

3. K-Means basato su MapReduce

L'algoritmo K-Means utilizzando il modello di MapReduce è suddiviso in tre parti principali:

- Map
- Combine
- Reduce

Nella parte di Map si assegna a ognuno degli n oggetti uno dei K centroidi. In modo da ridurre il carico di dati passati alla funzione di Reduce, la funzione di Combine calcola i risultati intermedi degli elementi che hanno lo stesso centroide. I dati poi passano alla funzione di Reduce che calcola la media degli elementi del centroide e assegna i nuovi valori.

Alla fine della funzione di Reduce si controlla se il criterio di arresto viene soddisfatto. Se la condizione non è vera si continua a iterare ripartendo da Map operando sui centri riassegnati. L'algoritmo si interrompe nel caso in cui il requisito di arresto è soddisfatto.

3.1. Implementazione con Apache Hadoop

Il programma sviluppato processa l'algoritmo di K-Means su una serie di punti che appartengono ad uno spazio N-Dimensionale. I dati in input sono un insieme di file di testo che in ogni riga ha le coordinate di un punto. Ad esempio:

$$1.17; 2.5; 3.895$$

Il numero dei K cluster viene specificato dall'utente ed è utilizzato per creare un file sequenziale. Il file contiene i centri generati in modo casuale e viene inizializzato prima dell'avvio del processo di MapReduce.

Gli argomenti da indicare all'avvio del programma sono:

- La cartella di input contenente i file di testo
- La cartella di output
- Il numero di K centri
- Il numero di coordinate del punto
- Il valore che indica il limite per la convergenza dell'algoritmo

3.1.1 Center e Point

Center e Point sono le classi su cui il processo di MapReduce lavora. Point è una classe che ha come unico campo la lista delle coordinate che descrivono il punto e implementa l'interfaccia `WritableComparable`.

Center è una sottoclasse di Point, ma ha anche come campo un identificativo e il numero dei punti appartenenti al cluster associato al centro.
I metodi di Center sono:

divideCoordinates Divide i valori delle coordinate per il numero di punti associati al centro

isConverged Controlla se la distanza del centro rispetto ad un altro è minore del valore limite di convergenza

addNumOfPoints Incrementa di un intero il numero di punti associati al centro

Codice 1. Center

```
16 public class Center extends Point {

18    private IntWritable index;

20    private IntWritable numberOfPoints;
```

3.1.2 Map

La funzione di Map ha una parte di inizializzazione in cui ricava dal file sequenziale i centri che verranno poi inseriti in un ArrayList globale (Codice 2).

Codice 2. Setup di Map

```
    protected void setup(Context context)
        throws IOException,
        InterruptedException {
28      Configuration conf = context.
            getConfiguration();
        Path centersPath = new Path(conf.get(
            "centersFilePath"));
30      SequenceFile.Reader reader = new
            SequenceFile.Reader(conf,
            SequenceFile.Reader.file(
            centersPath));
        IntWritable key = new IntWritable();
32      Center value = new Center();
        while (reader.next(key, value)) {
34        Center c = new Center(value.
              getListOfCoordinates());
          c.setNumberOfPoints(new
              IntWritable(0));
```

```
            c.setIndex(key);
            centers.add(c);
        }
        reader.close();
        logger.fatal("Centers: " + centers.
            toString());
    }
```

Le istruzioni che esegue la funzione di Map sono mostrate nel Codice 3. Nella prima parte del metodo si ricava le coordinate del punto dal file di input. Per ogni centro poi si calcola la distanza dal punto e infine si salva il centro più vicino all'elemento.

Codice 3. Funzione Map

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

        Center minDistanceCenter = null;
        Double minDistance = Double.MAX_VALUE
            ;
        Double distanceTemp;
        for (Center c : centers) {
            distanceTemp = Distance.
                findDistance(c, p);
            if (minDistance > distanceTemp) {
                minDistanceCenter = c;
                minDistance = distanceTemp;
            }
        }
        context.write(minDistanceCenter, p);
    }
```

3.1.3 Combine

La funzione Combine opera sulle coordinate dei punti che hanno associato lo stesso centro. Crea un oggetto Point che ha come valori le somme delle coordinate di tutti i punti e salva il numero di punti processati per aggiornare il valore `NumberOfPoints` del centro.

Per esempio, con i punti (1,2,3) e (4,5,6) associati ad uno stesso centro si crea un oggetto Point che ha come coordinate (5,7,9) e si aggiorna il valore `NumberOfPoints` del centro a 2.

Codice 4. Funzione Combine

```
    public void reduce(Center key, Iterable<
        Point> values, Context context)
        throws IOException,
            InterruptedException {
        Configuration conf = context.
            getConfiguration();

        Point sumValues = new Point(conf.
            getInt("iCoordinates", 2));
        int countValues = 0;
        Double temp;
        for (Point p : values) {
            for (int i = 0; i < p.
                getListOfCoordinates().size();
                i++) {
                temp = sumValues.
                    getListOfCoordinates().get(i
                    ).get() + p.
                    getListOfCoordinates().get(i
                    ).get();
                sumValues.getListOfCoordinates
                    ().get(i).set(temp);
            }
            countValues++;
        }
        key.setNumberOfPoints(new IntWritable
            (countValues));
        context.write(key, sumValues);
    }
```

3.1.4 Reduce

La funzione Reduce è molto simile a Combine. Infatti fa un ulteriore somma dei risultati parziali ricavati dalla funzione di Combine. Nel metodo `reduce` si popolano due HashMap: `newCenters` e `oldCenters`. Il primo contiene i centri che hanno come valori le somme delle coordinate dei punti a loro collegati e il numero di punti associati al cluster. Il secondo HashMap contiene i centri con i valori non aggiornati.

Codice 5. Funzione Reduce

```
    public void reduce(Center key, Iterable<
        Point> values, Context context)
```

```
30          throws IOException,
                InterruptedException {
        Configuration conf = context.
            getConfiguration();
32
        Center newCenter = new Center(conf.
            getInt("iCoordinates", 2));
34      boolean flagOld = false;
        if (newCenters.containsKey(key.
            getIndex())) {
36          newCenter = newCenters.get(key.
                getIndex());
            flagOld = true;
38      }

40      int numElements = 0;
        Double temp;
42      for (Point p : values) {
            for (int i = 0; i < p.
                getListOfCoordinates().size();
                i++) {
44              temp = newCenter.
                    getListOfCoordinates().get(i
                    ).get() + p.
                    getListOfCoordinates().get(i
                    ).get();
                newCenter.getListOfCoordinates
                    ().get(i).set(temp);
46          }
            numElements += key.
                getNumberOfPoints().get();
48      }
        newCenter.setIndex(key.getIndex());
50      newCenter.addNumberOfPoints(new
            IntWritable(numElements));

52      if (!flagOld) {
            newCenters.put(newCenter.getIndex
                (), newCenter);
54          oldCenters.put(key.getIndex(), new
                 Center(key));
        }
56
        context.write(newCenter.getIndex(),
            newCenter);
58  }
```

Nella fase di cleanup si aggiorna concretamente i valori dei centri e si confrontano le nuove coordinate con quelle vecchie per vedere se la condizione di convergenza è soddisfatta. Per aggiornare i valori basta dividere le coordinate rispetto al numero dei punti del cluster. Questa operazione è eseguita richiamando la funzione `divideCoordinates`.

La condizione di convergenza si divide in due vincoli:

1. Il centri che convergono (la distanza tra il vecchio valore e quello nuovo non supera il limite di threshold) sono maggiori o uguali al 90%.

2. La media dei delle distanze tra i vecchi valori e quelli nuovi non supera il limite. La media è calcolata in questo modo:

$$\sqrt{\frac{\sum distanza^2}{K}}$$

Se uno di questi attributi viene soddisfatto allora un contatore globale identificato da `CONVERGE_COUNTER.CONVERGED` viene incrementato.

Codice 6. Cleanup

```
    protected void cleanup(Context context)
        throws IOException,
        InterruptedException {
62      Configuration conf = context.
            getConfiguration();
        Path centersPath = new Path(conf.get(
            "centersFilePath"));
64      SequenceFile.Writer centerWriter =
            SequenceFile.createWriter(conf,
              SequenceFile.Writer.file(
                  centersPath),
66            SequenceFile.Writer.keyClass(
                  IntWritable.class),
              SequenceFile.Writer.valueClass(
                  Center.class));
68      Iterator<Center> it = newCenters.
            values().iterator();
        Center newCenterValue;
70      Center sameIndexC;
        Double avgValue = 0.0;
72      Double threshold = conf.getDouble("
            threshold", 0.5);
        int k = conf.getInt("k", 2);
74      while (it.hasNext()) {
            newCenterValue = it.next();
76          newCenterValue.divideCoordinates()
                ;
            sameIndexC = oldCenters.get(
                newCenterValue.getIndex());
78          if (newCenterValue.isConverged(
                sameIndexC, threshold))
              iConvergedCenters++;
80          avgValue += Math.pow(Distance.
                findDistance(newCenterValue,
                sameIndexC), 2);
```

```
            centerWriter.append(newCenterValue
                .getIndex(), newCenterValue);
        }
        avgValue = Math.sqrt(avgValue / k);
        logger.fatal("Convergence value: " +
            avgValue);
        int percentSize = (newCenters.size()
            * 90) / 100;
        logger.fatal("Percent value: " +
            percentSize);
        if (iConvergedCenters >= percentSize
            || avgValue < threshold)
            context.getCounter(
                CONVERGE_COUNTER.CONVERGED).
                increment(1);
        centerWriter.close();
        logger.fatal("Converged centers: " +
            iConvergedCenters);
    }
```

3.1.5 Driver

La classe principale che contiene il main è `KMeans`. Questa contiene la configurazione e gestisce i job che eseguono il processo di MapReduce. La prima parte del metodo inizializza tutte le variabili necessarie per l'esecuzione dei job utilizzando gli argomenti dati in input.

Codice 7. Setup di main

```
        Path input = new Path(args[0]);
        Path output = new Path(args[1]);
        Path centers = new Path(input.
            getParent().toString() + "centers/
            c.seq");

        conf.set("centersFilePath", centers.
            toString());
        conf.setDouble("threshold", Double.
            parseDouble(args[4]));

        int k = Integer.parseInt(args[2]);
        conf.setInt("k", k);
        int iCoordinates = Integer.parseInt(
            args[3]);
        conf.setInt("iCoordinates",
            iCoordinates);

        Job job;

        FileSystem fs = FileSystem.get(output
            .toUri(),conf);
        if (fs.exists(output)) {
            System.out.println("Delete old
                output folder: " + output.
                toString());
            fs.delete(output, true);
        }

        createCenters(k, conf, centers);
```

Il processo di MapReduce vero e proprio si mostra nel Codice 8. Si inizializza tutte le classi utilizzate e gli elementi di input e di output. Dopo ogni `waitForCompletion` si controlla che il contatore contenente il valore che indica la condizione di convergenza. Se questa è diversa da 1 si deve continuare il processo e si cancella la cartella di output generata. Se è 1 invece si esce dal ciclo e si passa alla scrittura del risultato finale.

Codice 8. Setup di main

```
        long isConverged = 0;
        int iterations = 0;
        while (isConverged != 1) {
            job = Job.getInstance(conf, "K
                means iter");
            job.setJarByClass(KMeans.class);
            job.setMapperClass(Map.class);
            job.setCombinerClass(Combine.class
                );
            job.setReducerClass(Reduce.class);

            FileInputFormat.addInputPath(job,
                input);
            FileOutputFormat.setOutputPath(job
                , output);
            job.setMapOutputKeyClass(Center.
                class);
            job.setMapOutputValueClass(Point.
                class);

            job.waitForCompletion(true);

            isConverged = job.getCounters().
                findCounter(Reduce.
                CONVERGE_COUNTER.CONVERGED).
                getValue();

            fs.delete(output, true);
            iterations++;
        }
```

L'output finale è un file che viene generato con la classe Map su tutti i valori senza che i centri siano modificati (Codice 9).

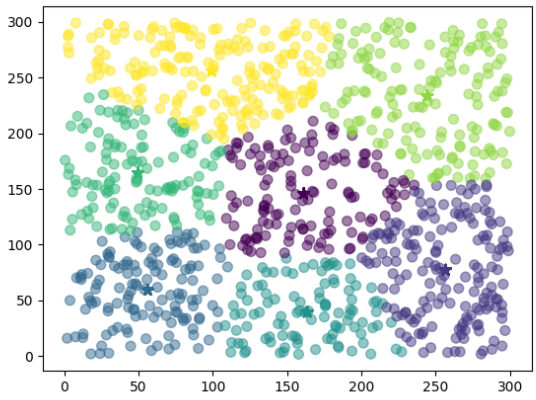

Figura 1. K = 7 e 1000 punti

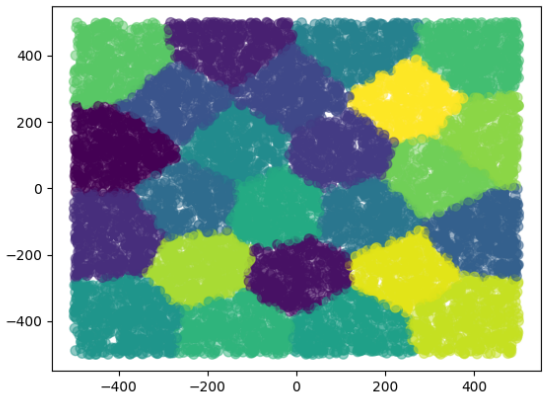

Figura 2. K = 25 e 10000 punti

Codice 9. Setup di main

```
        job = Job.getInstance(conf, "K means
            map");
72      job.setJarByClass(KMeans.class);
        job.setMapperClass(Map.class);
74
        FileInputFormat.addInputPath(job,
            input);
76      FileOutputFormat.setOutputPath(job,
            output);
        job.setMapOutputKeyClass(Center.class
            );
78      job.setMapOutputValueClass(Point.
            class);

80      job.waitForCompletion(true);

82      fs.delete(centers.getParent(), true);
        System.out.println("Number of
            iterations\t" + iterations);
```

4. Risultati sperimentali

Il programma è stato testato in locale e utilizzando i server con il servizio di Amazon Elastic Map Reduce (ERM). Oltre al programma Hadoop sono stati sviluppati degli script per la generazione di punti e per il plot dei risultati a due o tre dimensioni.

Nella Figura 1 si mostra il risultato del processo su 1000 punti e 7 cluster. Nella Figura 2 si aumenta il numero di punti a 10000 e i cluster a 25. In Figura 3 si mostra l'esempio di un risultato su 1000 punti a tre dimensioni con 5 cluster.

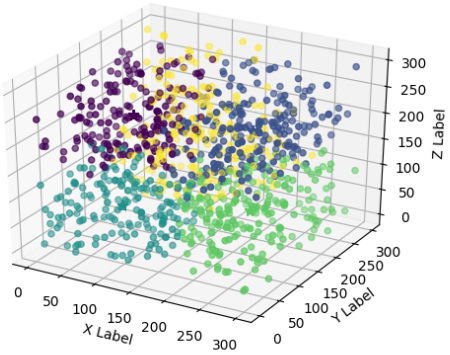

Figura 3. K = 5 e 1000 punti in 3d

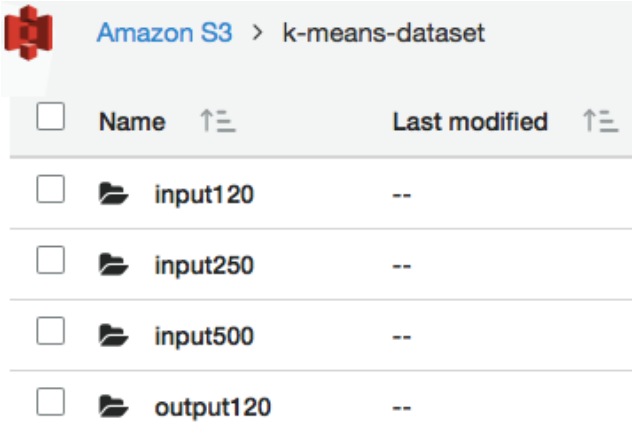

Figura 4. Dataset in S3

Usando il jar del progetto è stato possibile testare il processo anche su un ambiente completa-

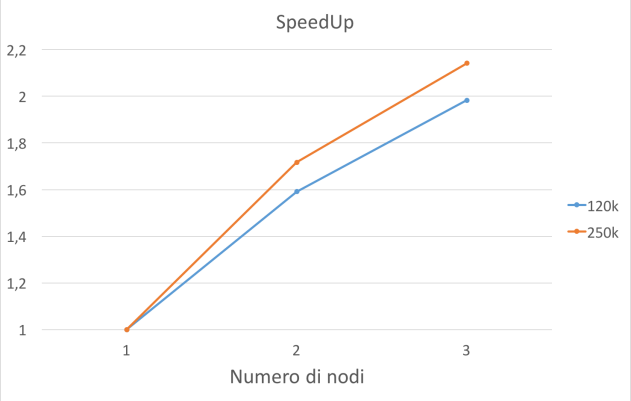

Figura 5. Speedup rispetto al numero di nodi

mente distribuito usufruendo del servizio di ERM. Caricando i dati su uno storage S3 (Figura 4) sono state testate varie configuarazioni su dataset diversi. Sono stati processati in particolare cartelle contenenti i dati su un totale di 120.000 punti e 250.000 punti su configurazioni con 1,2 e 3 nodi. Come si può vedere in Figura 5 lo speedup aumenta a seconda del numero di nodi, ma anche a seconda del carico di dati.

5. Codice

Tutto il codice si può trovare nella repository di Github:

`https://github.com/mameli/k-means-hadoop`