Hadoop k-means

Willian Ver Valem Paiva & Alan Guitard

January 15, 2017

Contents

1 Introduction

Ce projet est destiné à mettre en oeuvre un programme en map/reduce pattern pour calculer le K-means d'un ensemble de données à N dimensions en utilisant la récursivité pour construire un arbre hiérarchique. Il a aussi pour objectif de réaliser un deuxième programme map/reduce pattern qui étiquette les clusters créés à chaque niveau de la hiérarchie.

2 Mise en oeuvre

2.1 Le K-means

Le programme K-means travaille en exécutant les étapes suivantes :

1. création d'un dossier temporaire d'input dans les hdfs
2. copie du fichier input dans le premier niveau du dossier temporaire d'input
3. création d'un dossier temporaire d'output dans les hdfs
4. lancement d'un job pour tous les fichiers de chaque dossier d'un même niveau du dossier temporaire d'input en cours de traitement
   (a) examine chaque fichier pour trouver le K-means

(b) une fois qu'il est trouvé, création d'un fichier pour chaque cluster, doté de son numéro de cluster mis à la fin de chaque ligne et transfert dans le dossier temporaire d'outputs

5. prise de l'output des K-means et copie dans le niveau suivant du dossier temporaire d'input

6. une fois les K-means finis, concaténation des fichiers au dernier niveau du dossier temporaire d'input, et versement dans le fichier d'output défini dans les paramètres.

2.1.1 étapes 1 et 3

Pour créer les dossiers, nous utilisons la méthode mkdirs de FileSystem

```
1 fs.mkdirs(new Path(TEMPIN+"/N1"));
```

2.1.2 étape 2

Cette étape consiste à copier le fichier d'input dans le dossier temporaire d'input, à l'intérieur du sous-dossier N1 qui représente le niveau 1 de l'arbre hiérarchique, et pour cela, nous utilisons le simple code :

```
1 copyFile(args[0],TEMPIN+"/N1/1",fs,conf);
```

Cette partie en particulièr contenait un problème dont nous ne nous sommes pas rendu compte avant la fin du projet. Car nous avons toujours testé le programme avec le fichier *worldcitiespop.txt* et ça n'avait jamais posé de problème ; mais une fois que le code a été testé avec le gros fichier 52Go du cluster de machines de l'Université, nous avons constaté que cette approche n'était vraiment pas la bonne. Et de ce fait, les performances de notre programme en ont souffert...

2.1.3 étape 4 et 5

C'est là le coeur du programme ; à cette étape, nous lançons un K-means pour chaque fichier à l'intérieur d'un dossier du niveau hiérarchique de l'arbre. Exemple :

- à la racine l'arbre, nous lisons les fichiers à l'intérieur du dossier :

  `/tempin/N1`

- au niveau 2, nous lisons les fichiers à l'intérieur du dossier :

  `/tempin/N2`

- etc. . . , jusqu'à ce que nous atteignions le niveau donné par le paramètre.

Le K-means va copier l'output dans un fichier pour chaque cluster au niveau suivant de l'arbre hiérarchique. Exemple :

- Un K-means 2 au dossier racine donnera comme output :

  `/tempin/N2/1`
  `/tempin/N2/2`

- et un K-means 2 dans le dossier de niveau 2 donnera comme output :

  `/tempin/N3/1`
  `/tempin/N3/2`
  `/tempin/N3/3`
  `/tempin/N3/4`

- et ainsi de suite, jusqu'à atteindre le niveau donné dasn le paramètre +1

2.1.4 étape 6

A cette étape, nous concaténons tous les fichiers du dernier niveau du dossier d'input, pour constituer un fichier d'output (nommé comme le paramètre donné à l'exécution). Pour ce faire, nous utilisons une des premières leçons que nous avons eu en programmation *Hadoop*, en employant le code suivant :

```
1  Path outPut = new Path(args[1]);
2  OutputStream os = fs.create(outPut);
3  for(Path path:pths){
4    FSDataInputStream is = fs.open(path);
5    //copy the intout in to the output using the conf format
6    IOUtils.copyBytes(is,os,conf,false);
7    is.close();
8  }
```

2.2 Etiquettage

La partie "étiquettage" du projet a été réalisée en développant une classe appelée *LabelKey/qui est /writable* et *comparable* et aussi en développant un *partitioner* qui sépar les clés par le numéro de niveau de l'arbre. Le *LabelKey* prend simplement les colonnes de clusters et crée une chaîne en tant que clé. Exemple :

```
text,0,1,2 --> key = "0,1,2"
```

Dans ce cas, il créerait une taille qui représente le niveau ; dans cet exemplen la clé serait de taille 3.

2.2.1 Le *mapper*

le *mapper* dans ce programme lit les lignes et pour chaque ligne, écrit dans le *context* la [key, label, value] mais il fait ceci depuis le premier jusqu'au dernier niveau.

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
  String[] values = value.toString().split(",");
  String sk = "";
  for(int x:labels){
    int l = Integer.parseInt(values[x].replaceAll("\\s+",""));
    if(sk.equals("")){
      sk += ""+l;
    }else{
      sk += ","+l;
    }
    if(values[measureCol].equals("")){
      values[measureCol] = "0";
    }
    context.write(new LabelKey(new Text(sk)),new Text(values[labelCol]+","+values[measureCol]));
  }

}
```

et de cette façon, crée un fichier pour chaque niveau de l'arbre hiérarchique.

2.2.2 Le *reducer*

Dans le *reducer* nous prenons simplement la valeur la plus grande et utilisons son étiquette pour créer l'output :

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
  String[] values = value.toString().split(",");
  String sk = "";
  for(int x:labels){
    int l = Integer.parseInt(values[x].replaceAll("\\s+",""));
    if(sk.equals("")){
      sk += ""+l;
    }else{
      sk += ","+l;
    }
    if(values[measureCol].equals("")){
      values[measureCol] = "0";
    }
    context.write(new LabelKey(new Text(sk)),new Text(values[labelCol]+","+values[measureCol]));
  }

}
```