K-Means Clustering on MapReduce

Prepared by Yanbo Xu

Out April 3, 2013
Due Wednesday, April 17 2013 via Blackboard

1 Important Note

You are expected to use Java for this assignment.

Yanbo Xu (yanbox@cs.cmu.edu) is the contact TA for this homework. Please post clarification questions to the Google Group:
machine-learning-with-large-datasets-10-605-in-spring-2013

2 Overview

K-Means iteratively improves the partition of the data into K sets:

- Predefine the number of clusters, K
- Initialize K cluster centroids
- Iteration until the centroids no longer change
  - Associate each data instance with the closest centroid (we consider them in a Euclidean space in this assignment)
  - Recalculate centroids as an average of the associated data instances

3 K-Means Clustering on MapReduce

To parallelize K-Means on MapReduce, we are going to share some small information, i.e. the cluster centroids, across the iterations. This will result

in a duplication, but very minimal comparing with the large amount of data.

Therefore, before starting, a file is created accessible to all processors (through FileSystem in Configuration()) that contains the initial K cluster centroids. This file will be updated after each iteration to contain the latest cluster centroids calculated by Reducer. Then

1. The **Mapper** reads this file to get the centroids from last iteration. It then reads the input data and calculates the Euclidean distance to each centroid. It associates each instance with the closest centroid, and outputs (data instance id, cluster id).

2. Since this is a lot of data, we use a Combiner to reduce the size before sending it to Reducer. The **Combiner** calculates the average of the data instances for each cluster id, along with the number of the instances. It outputs (cluster id, (intermediate cluster centroid, number of instances)).

3. The **Reducer** calculates the weighted average of the intermediate centroids, and outputs (cluster id, cluster centroid).

The **main** function runs multiple iteration jobs using the above Mapper + Combiner + Reducer. You can use the following sample codes[1] to implement the multiple iterations in **main**:

```
int iteration = 0;

// counter from the previous running import job
long counter = job.getCounters().findCounter(Reducer.
Counter.CONVERGED).getValue();

iteration++;
while (counter > 0) {
   conf = new Configuration();
   conf.set("recursion.iter", iteration + "");
   job = new Job(conf);
```

[1] http://codingwiththomas.blogspot.com/2011/04/controlling-hadoop-job-recursion.html

```
    job.setJobName("KMeans " + iteration);

    //...
    // job.set Mapper, Combiner, Reducer...
    // ...

    // always take the output from last iteration as the input
    in = new Path("files/kmeans/iter_" + (iteration - 1) + "/");
    out = new Path("files/kmeans/iter_" + iteration);

    //...
    // job.set Input, Output...
    // ...

    // wait for completion and update the counter
    job.waitForCompletion(true);
    iteration++;
    counter = job.getCounters().findCounter(Reducer
    .Counter.CONVERGED).getValue();
}
```

Regarding the Counter, you can define an enum in **Reducer**:

```
public enum Counter{
   CONVERGED
}
```

and then update the counter:

```
context.getCounter(Counter.CONVERGED).increment(1);
```

4 The Data

We are going to cluster the Census data that was collected by the U.S. Census Bureau in 1990. Many of the less useful attributes in the original data set have been dropped, the few continuous variables have been discretized and the few discrete variables that have a large number of possible values have been collapsed to have fewer possible values. As a result, the data contains

68 categorical attributes.

The data appears at `/afs/cs.cmu.edu/project/bigML/census/`, and are also available in the public s3 bucket: `s3://bigml-shared/census`. The first row contains the list of attributes. The first attribute is a case id and should be ignored during analysis. The data is comma delimited with one case per row. Again, a small toy data is provided for debugging.

```
USCensus1990.small.txt
USCensus1990.full.txt
```

For the convenience of grading, the initial cluster centroids for K=8 and K=12 were already randomly generalized. Please use the following files as your starting points. Each row starts with a cluster id, and follows by the centroid's case id and values of the 68 attributes. Each row is still comma delimited.

```
centroids8.small.txt
centroids12.small.txt
centroids8.full.txt
centroids12.full.txt
```

5 Deliverables

Submit a compressed archive (zip, tar, etc) of your code, along with the controller and syslog files from AWS. Please include a pdf document with answers to the questions below.

1. Run K=8 and K=12 clusters on the small data, report the cluster centroids, the number of iterations for convergence, and the wall time respectively.

2. Run K=8 and K=12 clusters on the full data, report the cluster centroids, the number of iterations for convergence, and the wall time respectively.

   For each iteration, we compared each instance to each possible centroid, which may result in a large computation cost. We can reduce

the number of distance comparison by applying the Canopy Selection, described as in `www.kamalnigam.com/papers/canopy-kdd00.pdf`.

Please read the paper, and answer:

3. What distance metric would you choose for the canopy clustering? Why?
4. Can you implement the Canopy Selection on MapReduce? If yes, please describe the workflow.
5. Describe the workflow to combine the Canopy Selection with K-Means on MapReduce.

BONUS question: Implement the Canopy Selection with K-Means on MapReduce. Run K=12 on both small and full data. Report the cluster centroids, the number of iterations for convergence, and the wall time respectively.

6 Marking breakdown

- Code correctness: [**45 points**].
- Question 1, 2 [**10+10 points**]
- Question 3, 4, 5 [**5 + 15 + 15 points**]
- BONUS question: [**20 points**]