K-Means in Hadoop MapReduce

Øyvind Samuelsen
School of Science
RMIT
Melbourne, Australia
s3801950@student.rmit.edu.au

K-means is one of the most popular algorithms used in scientific and industrial applications [1]. With use of Hadoop MapReduce one can compute clusters more efficiently, and several optimizations can be used to further improve the computing speed. In this paper I will present an implementation, optimizations, scaling issues, and test results for a K-means algorithm on a Hadoop MapReduce system.

I. Implementation

The codebase is based on the algorithm from Week 7 tutorial, “mapreduce-kmeans.zip”, with missing code added. Based on the NYC TLC taxi data distributed on Canvas (due to missing location data on S3), methods to extract the pickup location (latitude and longitude) was implemented and saved in the same manner as the dummy data from the tutorial code. In this way all the map and reduce code could be left unmodified in the basic implementation.

The basic implementation works as follows: raw location-data is read and saved on HDFS as <Centroid, DataPoint>, where Centroid is a default (0,0) location. A user specified amount of Centroids is generated or chosen from a predefined set of random data points.

Before each Map task is run all centroids are read from HDFS and stored in a global variable used by each Map method. The Map method takes as input <Centroid, DataPoint>, computes the nearest Centroid for this DataPoint, and writes back to HDFS <NearestCentroid, DataPoint>.

The Reduce method takes as input <Centroid, DataPoints>, in other words all DataPoints that are closest to this Centroid. Reduce then computes the new Centroid, writes DataPoints back to HDFS with the new Centroid value, then all Centroids.

The MapReduce job finishes when all Centroids are converged, eg. they do not update in the last iteration.

II. Optimizations

A. *In-Map Combiner*

The In-Map Combiner is implemented in the Map phase as a keyed hash-map <Centroid, Datapoints>. Instead of writing each new <Centroid, DataPoint> back to disk one at a time, a list of <Centroid, DataPoints> is saved with the help of DpArrayWritable class. In this way one avoids several small network transfers in change of few larger ones.

B. *Partitioner*

A simple Partitioner is implemented which assigns each reducer a set of <Centroid, DataPoints>. This is also the default setting for all implementations and tests in this paper.

C. *In-Map and Combiner Optimized*

An intermediate Centroid value can be computed after each Map task. Since the new Centroid value is the mean of all the closest DataPoints, and mean is not associative and commutative, the output of the Map and Combiner methods must be edited to output <Centroid, <CentSum, NumDataPoints>>. Here the CentSum is the sum of DataPoints belonging to Centroid, and NumDataPoints is the number of DataPoints used to calculate CentSum. The Reducer can then calculate the correct mean by summarizing CentSums and NumDataPoints. A reference to the old Centroid is needed to check if the new Centroid has a new value, or if the values have converged and the algorithm is finished.

An implementation of this is on GitHub, but unfortunately there are still some errors with it so I was not able to do any tests.

III. Scaling

There are several scaling issues related to this implementation of K-means. With larger datasets these issues will be more dominant.

A. *Number of centroids*

All centroids are loaded into memory in Map. Usually this is not an issue as a normal amount of centroids rarely exceeds the memory available.

With more centroids there would also normally be more iterations needed before the centroids converge, and the MapReduce job would take longer time to finish.

B. *Data size*

The basic implementation and the In-Map Combiner optimization are affected by the size of the datasets.

- When the dataset size increases with N datapoints in a node, the basic implementation will output N more intermediate <key, value> pairs. These pairs need to be transferred over the network, which is a bottleneck when the dataset is too big.
- The In-Map Combiner optimization has the advantage that the number of outputs in a Map node is only affected by the number of Centroids. However, the size of each output will increase

linearly with N. This is solved in the optimized In-Map Reducer.

- The In-Map Combiner has to store all datapoints in a hash-map in memory. When the amount of datapoints is too big the system would run out of available memory. This would also be solved in the optimized version, where only N centroids in each Map method is used to accumulate the sum of all belonging datapoints and the number of points.

C. *Number of nodes*

With more nodes added to the cluster, the MapReduce system could distribute the Map and Reduce jobs more efficiently, with more work done at the same time. In an ideal system one would have equal amounts of Reducers as nodes, such that each node could compute the mean for one centroid.

IV. Results

Test id	Algorithm	# Nodes	# Centroids	Data size	Runtime (minute)
1	Basic	2	4	1.8 GB	74
1	**In-Map-C**	**2**	**4**	**1.8 GB**	**67**
2	Basic	12	4	1.8 GB	66
2	**In-Map-C**	**12**	**4**	**1.8 GB**	**57**
3	Basic	12	10	1.8 GB	160
3	**In-Map-C**	**12**	**10**	**1.8 GB**	**123**

Test results given different optimizations

A. *Test 1*

With 2 nodes and 4 centroids the two algorithms had almost no difference in runtime. Still the In-Map Combiner finished 7 minutes faster, and is most likely because of fewer data-blocks needed to be transferred over the network.

B. *Test 2*

With 12 nodes in the cluster the In-Map Combiner performed better. Still the performance improvement is not as much as one could expect, and this is possibly mainly because of the limitation of 1 reducer. The improvement for Basic was a marginally less.

C. *Test 3*

With 10 centroids to compute the centers for one would expect the runtime to be higher, because of more iterations needed to converge, which is also the case. The difference in performance is more clearly here, with In-Map Combiner finishing 37 minutes earlier than Basic.

V. Influence factors

A. *1 reducer*

The sample code provided did only work correctly with 1 reducer, and after extensive debugging I did not manage to fix it. With 1 reducer the performance improvements from optimizations such as in-map combining is very limited. There would be no improvements with a custom partitioner and all reduce operations for each centroid would have to be queued and done sequentially. This is very unfortunate and the test results are not representative of a “normal” MapReduce program.

B. *Test data*

The NYC TLC data distributed on S3 does not always contain pickup locations due to privacy reasons. Therefore the tests were only run on one dataset. In order to compare the results a bit more systematic, *N* fixed initial centroids were chosen in this way: for the first *X* datapoints, find the maximum and minimum values. From this range randomly generate points, and save them for use in later iterations.

VI. End notes

In this paper I have presented an implementation of K-means in MapReduce and optimizations to improve the performance of the algorithm. Scaling issues such as number of initial centroids and dataset size were discussed, and finally a few test results were presented with influence factors discussed.

References

[1] P. Berkhin, “Survey of Clustering Data Mining Techniques,” Accrue Software, Inc., pp. 15,