Nicolas Escobar (escobarn)
Harsh Reddy (hagandav)

Cloud Computing - Project 8 Report

Harp Mini Batch Kmeans

Deliverables

Zip your source code and report as username_mbkmeans.zip. Please submit this file to the Canvas Assignments page.

Evaluation

The point total for this project is 6, where the distribution is as follows:

- Completeness of your code (5 points)
- In the report, describe your implementation and the output (1 point)

Harp MiniBatch Kmeans Implementation Report and Output

The application has three Java classes:

1. **KmeansMiniBatchMapCollective**: Job configuration based on parameters received
2. **KmeansMiniBatchMapper**: Execution of MiniBatch Kmeans algorithm and communication between workers based on Harp Allreduce computation model.
3. **KMeansMiniBatchConstants**: Constants that will be used in the Mapper application

The flow is described below:

1. KmeansMiniBatchMapCollective

main(String[] args)

- Main runs and creates a new instance of the KmeansMiniBatchMapCollective class
- The instance along with the arguments received in main are passed to the run method

run(String[] args)

- This method checks that the number of arguments is correct
- It then stores the arguments on local variables that are passed to the launch method

launch(int batchSize, int numOfTasks, int numOfIterations, int numOfCentroids, String workDir)

- The *launch* method will create the Path variables to store the input (data) and the output data (out) on HDFS.
- The out data folder will be created (and deleted if it exists)
- The Job configureMBKmeansJob is executed to configure the MapReduce Job on Hadoop (input/output path, input format, Jar, Mapper class, etc.)
- The result of the job execution is stored on a Job variable

configureMBKMeansJob(batchSize, numOfMapTasks, numOfCentroids, configuration, workDirPath, dataDir,cenDir, outDir, JobID, numOfIterations)

- This method performs the following job configurations:

a. Job configurations constants
b. Job input format
c. Jar by Class
d. Mapper definition
e. Number of mappers and reducers
f. Framework usage (map-collective)

- Once this is defined, the Mapper class defined in setMapperClass will be executed (KmeansMiniBatchMapper)

2. KmeansMiniBatchMapper

setup(Context context)

- In this method we get the values from Configuration instance which are defined by KMeansMiniBatchConstants and store the results in private class variables (batch size, number of mappers, vector size, number of iterations and number of centroids)

mapCollective(KeyValReader reader, Context context)

- This method is called for each file
- The runMBKmeans is called with the list of files and the configuration variable

runMBKmeans(List<String> fileNames, Configuration conf, Context context)

- This is the main method that implements the Mini Batch Kmeans algorithm
- The **loadData** method is called to load data points from HDFS file
- New centroid Table is created and passed to the **generateCentroids** which will generate random centroids from the data points that were loaded from HDFS
- The generated centroids (centroidsTable) are **broadcasted** to workers by the Master worker
- Then we implement the for loop up to the **number of iterations** defined in the arguments
    - For each iteration we execute **loadSampleData** which will load sample data based on batch size argument defined and store this in *sampleDataPoints* double Array
    - We then run **nearestCentroid** which will receive current and previous Centroid tables as well as sample data points to calculate nearest centroids
    - The **allreduce** method is executed to collect all centroid calculations from workers
    - We then execute the **calculateMBCentroids** method which will perform the final Centroid calculations
    - Finally, the Master will run **outputCentroids** method to write results to HDFS

loadData(List<String> fileNames, Configuration conf)

- This method will iterate over the list of files
- Then for each file:
    - And for each line:

        - We define the size of vectorSize (number of dimensions) which will be the length of the vector after the split (“,”)
        - We create a double vector of size[vectorSize] which will store the data points for each line
        - The created vector will be added to a DoubleArray which will store the list of data points for the line and added to an ArrayList<DoubleArray> which will contain the list of all data points for all lines and files

- The list of double arrays (data) is returned

generateCentroids(Table<DoubleArray> cenTable, ArrayList<DoubleArray> dataPoints)

- For the number of centroids that were passed as argument:
    - Generate a random integer between 0 and the size of the data point ArrayList
    - Choose random data point from dataPoints using random integer
    - Create a centroid array from selected data point
    - New Partition is created from new random centroid and added to centroids Table that was passed as an argument to the function

loadSampleData (ArrayList<DoubleArray> dataPoints)

- The sample size variable is defined as batch size / number of mappers
- For each iteration from 0 to sampleSize we create a random integer (between 0 and data size) which we use to select a random data point from the data points set
- The method returns the sample data set to be used later in the MB Kmeans calculation

nearestCentroid(Table<DoubleArray> cenTable, Table<DoubleArray> previousCenTable, ArrayList<DoubleArray> sampleDataPoints)

- For each data point in the sample data set we do the following:
    - For each partition in the previous centroid table:
        - We calculate the Euclidean distance between the data point and the centroid
        - We keep the centroid’s partition where the distance between the data point and the centroid is the minimum
        - We check whether this partition ID exists already in the new Centroid Table. If it does then we get the centroid for that partition ID and add the specified data point. If it does not, then we add the new partition ID to the Centroid Table
- The resulting Centroid Table will contain the list of centroids with the accumulated data points. The last element of each partition will store the number of points clustered around each partition ID / centroid.

calculateMBCentroids(Table<DoubleArray> cenTable, Table<DoubleArray> prevCenTable)

- For each partition on the current Centroid Table we do the following:
    - Get previous iteration Centroid Partition for the ID of current partition
    - Accumulate counts for current partition ID: V[c] = V[c]+V-1[c]
    - For each element of current partition centroid:
        - Apply calculation to get learning rate and take gradient step
        - C[c] = (C-1[c]*V-1[c]+C[c])/V[c]

outputCentroids(Table<DoubleArray> cenTable,Configuration conf, Context context)

- For each centroid in the final cenTable we store the results in a String output
- We write the resulting output String as Text to HDFS. This will be the Mapper output.

3. KMeansMiniBatchConstants

- Definitions of all public static variables to store job configuration details such as Job ID, number of iterations, number of centroids, vector size, batch size, working directory and number of mappers.

Output Description

The resulting output contains a list of final clusters (data points).

We executed the algorithm with number of centroids = 5, so the output contained 5 different clusters where we noted the following observations for each one (see image below).

Cluster #1: LOW ageing (newer users) with MEDIUM revenue and voice/data consumption in postpaid, and MEDIUM revenue and voice/data consumption after migration to prepaid. In this case the revenue decreased after migration but consumption did not.

Cluster #2: LOW ageing (newer users) with VERY HIGH revenue and voice/data consumption in post-paid and VERY HIGH revenue and voice/data consumption after migration to pre-paid. This cluster represents healthy migrations as revenue increased afterwards.

Cluster #3: HIGH ageing (older users) with HIGH revenue and voice/data consumption in post-paid and MEDIUM revenue and voice/data consumption after migration to pre-paid. There was a significant revenue decrease in this segment after migration.

Cluster #4: LOW ageing (newer users) with LOW revenue and voice/data consumption in post-paid and VERY LOW revenue and voice/data consumption after migration to pre-paid. The customers that are part of this cluster were already low users in post-paid so the migration to pre-paid does not represent a big impact.

Cluster #5: HIGH ageing (older users) with HIGH revenue and voice/data consumption in post-paid and VERY LOW revenue and voice/data consumption after migration to prepaid. The customers in this cluster are migrating to pre-paid but they stop using their mobile number so it is almost as a cancelation of the service.

	AGEING	AVG_REV_3M_POS	AVG_MB_3M_POS	AVG_MOU_3M_POS	AVG_PAQUETIGOS_3M	RCHRG_QTY	MOST_FREQ_RCHRG_HIGH	MOST_FREQ_RCHRG_LOW	MOU_OUTGOING	MOU_INCOMING	AVG_REV_3M_PRE	AVG_MB_3M_PRE	AVG_MOU_3M_PRE
Cluster 1	819	84,769	1,531	139	24,364	4	5,966	5,195	123	23	73,450	1,846	121
Cluster 2	881	261,359	5,970	343	73,206	8	23,571	20,179	236	71	278,572	5,815	307
Cluster 3	1,160	186,941	4,178	259	55,176	6	10,291	8,763	196	43	82,193	2,168	135
Cluster 4	837	32,564	428	52	6,081	1	2,241	2,052	45	7	12,701	252	18
Cluster 5	1,129	101,201	1,811	129	18,079	2	3,532	3,226	106	17	9,467	208	16