**Code for Data Clustering Using Firefly Algorithm in Distributed Environment**

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class FireflyKMeans {

// Helper class for representing data points

public static class DataPoint {

double[] features;

DataPoint(String[] values) {

features = new double[values.length];

for (int i = 0; i < values.length; i++) {

features[i] = Double.parseDouble(values[i]);

}

}

public double distance(DataPoint other) {

double sum = 0;

for (int i = 0; i < features.length; i++) {

sum += Math.pow(features[i] - other.features[i], 2);

}

return Math.sqrt(sum);

}

}

public static class FireflyMapper extends Mapper<Object, Text, IntWritable, Text> {

private List<DataPoint> initialPopulation = new ArrayList<>();

private int populationSize = 10; // Adjust as necessary

@Override

protected void setup(Context context) throws IOException, InterruptedException {

// Initialize random population

for (int i = 0; i < populationSize; i++) {

initialPopulation.add(generateRandomDataPoint());

}

}

private DataPoint generateRandomDataPoint() {

Random random = new Random();

double[] features = new double[10]; // Adjust to the number of features in your dataset

for (int i = 0; i < features.length; i++) {

features[i] = random.nextDouble(); // Generate random values

}

return new DataPoint(features);

}

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

String[] fields = value.toString().split(",");

DataPoint dataPoint = new DataPoint(fields);

DataPoint bestFirefly = null;

double bestIntensity = Double.MAX\_VALUE;

// Firefly algorithm to find the best solution

for (DataPoint firefly : initialPopulation) {

double intensity = dataPoint.distance(firefly);

if (intensity < bestIntensity) {

bestIntensity = intensity;

bestFirefly = firefly;

}

}

context.write(new IntWritable(1), new Text(bestFirefly.toString()));

}

}

public static class FireflyReducer extends Reducer<IntWritable, Text, IntWritable, Text> {

public void reduce(IntWritable key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

// Combine firefly results

for (Text value : values) {

context.write(key, value); // In a real implementation, you'd combine the results more meaningfully

}

}

}

public static class KMeansMapper extends Mapper<Object, Text, IntWritable, Text> {

private List<DataPoint> centroids = new ArrayList<>();

private int k = 5; // Number of clusters, adjust as necessary

@Override

protected void setup(Context context) throws IOException, InterruptedException {

// Initialize centroids randomly

for (int i = 0; i < k; i++) {

centroids.add(generateRandomDataPoint());

}

}

private DataPoint generateRandomDataPoint() {

Random random = new Random();

double[] features = new double[10]; // Adjust to the number of features in your dataset

for (int i = 0; i < features.length; i++) {

features[i] = random.nextDouble(); // Generate random values

}

return new DataPoint(features);

}

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

String[] fields = value.toString().split(",");

DataPoint dataPoint = new DataPoint(fields);

DataPoint nearestCentroid = null;

double nearestDistance = Double.MAX\_VALUE;

// Find the nearest centroid

for (int i = 0; i < centroids.size(); i++) {

double distance = dataPoint.distance(centroids.get(i));

if (distance < nearestDistance) {

nearestDistance = distance;

nearestCentroid = centroids.get(i);

}

}

context.write(new IntWritable(centroids.indexOf(nearestCentroid)), value);

}

}

public static class KMeansReducer extends Reducer<IntWritable, Text, IntWritable, Text> {

public void reduce(IntWritable key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

// Combine K-Means results

for (Text value : values) {

context.write(key, value); // In a real implementation, you'd update centroids and check for convergence

}

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

// Job 1: Firefly Algorithm

Job job1 = Job.getInstance(conf, "firefly algorithm");

job1.setJarByClass(FireflyKMeans.class);

job1.setMapperClass(FireflyMapper.class);

job1.setReducerClass(FireflyReducer.class);

job1.setOutputKeyClass(IntWritable.class);

job1.setOutputValueClass(Text.class);

FileInputFormat.addInputPath(job1, new Path(args[0]));

FileOutputFormat.setOutputPath(job1, new Path(args[1]));

job1.waitForCompletion(true);

// Job 2: K-Means Clustering

Job job2 = Job.getInstance(conf, "kmeans clustering");

job2.setJarByClass(FireflyKMeans.class);

job2.setMapperClass(KMeansMapper.class);

job2.setReducerClass(KMeansReducer.class);

job2.setOutputKeyClass(IntWritable.class);

job2.setOutputValueClass(Text.class);

FileInputFormat.addInputPath(job2, new Path(args[1]));

FileOutputFormat.setOutputPath(job2, new Path(args[2]));

System.exit(job2.waitForCompletion(true) ? 0 : 1);

// Calculate and print time complexity estimates

int populationSize = 10;

int k = 5;

int d = 10;

long job1Complexity = TimeComplexityCalculator.calculateJob1TimeComplexity(populationSize, d);

long job2Complexity = TimeComplexityCalculator.calculateJob2TimeComplexity(k, d);

System.out.println("Estimated time complexity for Job 1: O(" + job1Complexity + ")");

System.out.println("Estimated time complexity for Job 2: O(" + job2Complexity + ")");

}

// Helper class for representing data points

public static class DataPoint {

double[] features;

DataPoint(String[] values) {

features = new double[values.length];

for (int i = 0; i < values.length; i++) {

features[i] = Double.parseDouble(values[i]);

}

}

public double distance(DataPoint other) {

double sum = 0;

for (int i = 0; i < features.length; i++) {

sum += Math.pow(features[i] - other.features[i], 2);

}

return Math.sqrt(sum);

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append("(");

for (int i = 0; i < features.length; i++) {

sb.append(features[i]);

if (i < features.length - 1) {

sb.append(", ");

}

}

sb.append(")");

return sb.toString();

}

}

}

Similar by replacing the path of the dataset file, we can run the code for any size of the dataset.