**20BDS0146**

**VENNELA G**

**MATHEMATICAL MODELLING FOR DATA SCIENCE**

**LAB ASSESSMENT-4**

**SLOT: E1+TE1**

Write the programs for implementing the Machine Learning Algorithms Mentioned in Questions 1 to 4 using any programming Language of your choice.

**1. Back Propagation Neural Networks Algorithm.**

**CODE:**

NeuralNetwork.java

**import** java.text.\*;

**import** java.util.\*;

**public** **class** NeuralNetwork {

**static** {

Locale.setDefault(Locale.ENGLISH);

}

**final** **boolean** isTrained = **false**;

**final** DecimalFormat df;

**final** Random rand = **new** Random();

**final** ArrayList<Neuron> inputLayer = **new** ArrayList<Neuron>();

**final** ArrayList<Neuron> hiddenLayer = **new** ArrayList<Neuron>();

**final** ArrayList<Neuron> outputLayer = **new** ArrayList<Neuron>();

**final** Neuron bias = **new** Neuron();

**final** **int**[] layers;

**final** **int** randomWeightMultiplier = 1;

**final** **double** epsilon = 0.00000000001;

**final** **double** learningRate = 0.9f;

**final** **double** momentum = 0.7f;

// Inputs for xor problem

**final** **double** inputs[][] = { { 1, 1 }, { 1, 0 }, { 0, 1 }, { 0, 0 } };

// Corresponding outputs, xor training data

**final** **double** expectedOutputs[][] = { { 0 }, { 1 }, { 1 }, { 0 } };

**double** resultOutputs[][] = { { -1 }, { -1 }, { -1 }, { -1 } }; // dummy init

**double** output[];

// for weight update all

**final** HashMap<String, Double> weightUpdate = **new** HashMap<String, Double>();

**public** **static** **void** main(String[] args) {

NeuralNetwork nn = **new** NeuralNetwork(2, 4, 1);

**int** maxRuns = 50000;

**double** minErrorCondition = 0.001;

nn.run(maxRuns, minErrorCondition);

}

**public** NeuralNetwork(**int** input, **int** hidden, **int** output) {

**this**.layers = **new** **int**[] { input, hidden, output };

df = **new** DecimalFormat("#.0#");

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

**if** (i == 0) { // input layer

**for** (**int** j = 0; j < layers[i]; j++) {

Neuron neuron = **new** Neuron();

inputLayer.add(neuron);

}

} **else** **if** (i == 1) { // hidden layer

**for** (**int** j = 0; j < layers[i]; j++) {

Neuron neuron = **new** Neuron();

neuron.addInConnectionsS(inputLayer);

neuron.addBiasConnection(bias);

hiddenLayer.add(neuron);

}

}

**else** **if** (i == 2) { // output layer

**for** (**int** j = 0; j < layers[i]; j++) {

Neuron neuron = **new** Neuron();

neuron.addInConnectionsS(hiddenLayer);

neuron.addBiasConnection(bias);

outputLayer.add(neuron);

}

} **else** {

System.out.println("!Error NeuralNetwork init");

}

}

// initialize random weights

**for** (Neuron neuron : hiddenLayer) {

ArrayList<Connection> connections = neuron.getAllInConnections();

**for** (Connection conn : connections) {

**double** newWeight = getRandom();

conn.setWeight(newWeight);

}

}

**for** (Neuron neuron : outputLayer) {

ArrayList<Connection> connections = neuron.getAllInConnections();

**for** (Connection conn : connections) {

**double** newWeight = getRandom();

conn.setWeight(newWeight);

}

}

// reset id counters

Neuron.counter = 0;

Connection.counter = 0;

**if** (isTrained) {

trainedWeights();

updateAllWeights();

}

}

// random

**double** getRandom() {

**return** randomWeightMultiplier \* (rand.nextDouble() \* 2 - 1); // [-1;1[

}

**public** **void** setInput(**double** inputs[]) {

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

inputLayer.get(i).setOutput(inputs[i]);

}

}

**public** **double**[] getOutput() {

**double**[] outputs = **new** **double**[outputLayer.size()];

**for** (**int** i = 0; i < outputLayer.size(); i++)

outputs[i] = outputLayer.get(i).getOutput();

**return** outputs;

}

**public** **void** activate() {

**for** (Neuron n : hiddenLayer)

n.calculateOutput();

**for** (Neuron n : outputLayer)

n.calculateOutput();

}

**public** **void** applyBackpropagation(**double** expectedOutput[]) {

// error check, normalize value ]0;1[

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

**double** d = expectedOutput[i];

**if** (d < 0 || d > 1) {

**if** (d < 0)

expectedOutput[i] = 0 + epsilon;

**else**

expectedOutput[i] = 1 - epsilon;

}

}

**int** i = 0;

**for** (Neuron n : outputLayer) {

ArrayList<Connection> connections = n.getAllInConnections();

**for** (Connection con : connections) {

**double** ak = n.getOutput();

**double** ai = con.leftNeuron.getOutput();

**double** desiredOutput = expectedOutput[i];

**double** partialDerivative = -ak \* (1 - ak) \* ai

\* (desiredOutput - ak);

**double** deltaWeight = -learningRate \* partialDerivative;

**double** newWeight = con.getWeight() + deltaWeight;

con.setDeltaWeight(deltaWeight);

con.setWeight(newWeight + momentum \* con.getPrevDeltaWeight());

}

i++;

}

// update weights for the hidden layer

**for** (Neuron n : hiddenLayer) {

ArrayList<Connection> connections = n.getAllInConnections();

**for** (Connection con : connections) {

**double** aj = n.getOutput();

**double** ai = con.leftNeuron.getOutput();

**double** sumKoutputs = 0;

**int** j = 0;

**for** (Neuron out\_neu : outputLayer) {

**double** wjk = out\_neu.getConnection(n.id).getWeight();

**double** desiredOutput = (**double**) expectedOutput[j];

**double** ak = out\_neu.getOutput();

j++;

sumKoutputs = sumKoutputs

+ (-(desiredOutput - ak) \* ak \* (1 - ak) \* wjk);

}

**double** partialDerivative = aj \* (1 - aj) \* ai \* sumKoutputs;

**double** deltaWeight = -learningRate \* partialDerivative;

**double** newWeight = con.getWeight() + deltaWeight;

con.setDeltaWeight(deltaWeight);

con.setWeight(newWeight + momentum \* con.getPrevDeltaWeight());

}

}

}

**void** run(**int** maxSteps, **double** minError) {

**int** i;

// Train neural network until minError reached or maxSteps exceeded

**double** error = 1;

**for** (i = 0; i < maxSteps && error > minError; i++) {

error = 0;

**for** (**int** p = 0; p < inputs.length; p++) {

setInput(inputs[p]);

activate();

output = getOutput();

resultOutputs[p] = output;

**for** (**int** j = 0; j < expectedOutputs[p].length; j++) {

**double** err = Math.pow(output[j] - expectedOutputs[p][j], 2);

error += err;

}

applyBackpropagation(expectedOutputs[p]);

}

}

printResult();

System.out.println("Sum of squared errors = " + error);

System.out.println("##### EPOCH " + i+"\n");

**if** (i == maxSteps) {

System.out.println("!Error training try again");

} **else** {

printAllWeights();

printWeightUpdate();

}

}

**void** printResult()

{

System.out.println("NN example with xor training");

**for** (**int** p = 0; p < inputs.length; p++) {

System.out.print("INPUTS: ");

**for** (**int** x = 0; x < layers[0]; x++) {

System.out.print(inputs[p][x] + " ");

}

System.out.print("EXPECTED: ");

**for** (**int** x = 0; x < layers[2]; x++) {

System.out.print(expectedOutputs[p][x] + " ");

}

System.out.print("ACTUAL: ");

**for** (**int** x = 0; x < layers[2]; x++) {

System.out.print(resultOutputs[p][x] + " ");

}

System.out.println();

}

System.out.println();

}

String weightKey(**int** neuronId, **int** conId) {

**return** "N" + neuronId + "\_C" + conId;

}

/\*\*

\* Take from hash table and put into all weights

\*/

**public** **void** updateAllWeights() {

// update weights for the output layer

**for** (Neuron n : outputLayer) {

ArrayList<Connection> connections = n.getAllInConnections();

**for** (Connection con : connections) {

String key = weightKey(n.id, con.id);

**double** newWeight = weightUpdate.get(key);

con.setWeight(newWeight);

}

}

// update weights for the hidden layer

**for** (Neuron n : hiddenLayer) {

ArrayList<Connection> connections = n.getAllInConnections();

**for** (Connection con : connections) {

String key = weightKey(n.id, con.id);

**double** newWeight = weightUpdate.get(key);

con.setWeight(newWeight);

}

}

}

// trained data

**void** trainedWeights() {

weightUpdate.clear();

weightUpdate.put(weightKey(3, 0), 1.03);

weightUpdate.put(weightKey(3, 1), 1.13);

weightUpdate.put(weightKey(3, 2), -.97);

weightUpdate.put(weightKey(4, 3), 7.24);

weightUpdate.put(weightKey(4, 4), -3.71);

weightUpdate.put(weightKey(4, 5), -.51);

weightUpdate.put(weightKey(5, 6), -3.28);

weightUpdate.put(weightKey(5, 7), 7.29);

weightUpdate.put(weightKey(5, 8), -.05);

weightUpdate.put(weightKey(6, 9), 5.86);

weightUpdate.put(weightKey(6, 10), 6.03);

weightUpdate.put(weightKey(6, 11), .71);

weightUpdate.put(weightKey(7, 12), 2.19);

weightUpdate.put(weightKey(7, 13), -8.82);

weightUpdate.put(weightKey(7, 14), -8.84);

weightUpdate.put(weightKey(7, 15), 11.81);

weightUpdate.put(weightKey(7, 16), .44);

}

**public** **void** printWeightUpdate() {

System.out.println("printWeightUpdate, put this i trainedWeights() and set isTrained to true");

// weights for the hidden layer

**for** (Neuron n : hiddenLayer) {

ArrayList<Connection> connections = n.getAllInConnections();

**for** (Connection con : connections) {

String w = df.format(con.getWeight());

System.out.println("weightUpdate.put(weightKey(" + n.id + ", "

+ con.id + "), " + w + ");");

}

}

// weights for the output layer

**for** (Neuron n : outputLayer) {

ArrayList<Connection> connections = n.getAllInConnections();

**for** (Connection con : connections) {

String w = df.format(con.getWeight());

System.out.println("weightUpdate.put(weightKey(" + n.id + ", "

+ con.id + "), " + w + ");");

}

}

System.out.println();

}

**public** **void** printAllWeights() {

System.out.println("printAllWeights");

// weights for the hidden layer

**for** (Neuron n : hiddenLayer) {

ArrayList<Connection> connections = n.getAllInConnections();

**for** (Connection con : connections) {

**double** w = con.getWeight();

System.out.println("n=" + n.id + " c=" + con.id + " w=" + w);

}

}

// weights for the output layer

**for** (Neuron n : outputLayer) {

ArrayList<Connection> connections = n.getAllInConnections();

**for** (Connection con : connections) {

**double** w = con.getWeight();

System.out.println("n=" + n.id + " c=" + con.id + " w=" + w);

}

}

System.out.println();

}

}

Neuron.java

**import** java.util.\*;

**public** **class** Neuron {

**static** **int** counter = 0;

**final** **public** **int** id; // auto increment, starts at 0

Connection biasConnection;

**final** **double** bias = -1;

**double** output;

ArrayList<Connection> Inconnections = **new** ArrayList<Connection>();

HashMap<Integer,Connection> connectionLookup = **new** HashMap<Integer,Connection>();

**public** Neuron(){

id = counter;

counter++;

}

**public** **void** calculateOutput(){

**double** s = 0;

**for**(Connection con : Inconnections){

Neuron leftNeuron = con.getFromNeuron();

**double** weight = con.getWeight();

**double** a = leftNeuron.getOutput(); //output from previous layer

s = s + (weight\*a);

}

s = s + (biasConnection.getWeight()\*bias);

output = g(s);

}

**double** g(**double** x) {

**return** sigmoid(x);

}

**double** sigmoid(**double** x) {

**return** 1.0 / (1.0 + (Math.exp(-x)));

}

**public** **void** addInConnectionsS(ArrayList<Neuron> inNeurons){

**for**(Neuron n: inNeurons){

Connection con = **new** Connection(n,**this**);

Inconnections.add(con);

connectionLookup.put(n.id, con);

}

}

**public** Connection getConnection(**int** neuronIndex){

**return** connectionLookup.get(neuronIndex);

}

**public** **void** addInConnection(Connection con){

Inconnections.add(con);

}

**public** **void** addBiasConnection(Neuron n){

Connection con = **new** Connection(n,**this**);

biasConnection = con;

Inconnections.add(con);

}

**public** ArrayList<Connection> getAllInConnections(){

**return** Inconnections;

}

**public** **double** getBias() {

**return** bias;

}

**public** **double** getOutput() {

**return** output;

}

**public** **void** setOutput(**double** o){

output = o;

}

}

Connection.java

**public** **class** Connection {

**double** weight = 0;

**double** prevDeltaWeight = 0; // for momentum

**double** deltaWeight = 0;

**final** Neuron leftNeuron;

**final** Neuron rightNeuron;

**static** **int** counter = 0;

**final** **public** **int** id; // auto increment, starts at 0

**public** Connection(Neuron fromN, Neuron toN) {

leftNeuron = fromN;

rightNeuron = toN;

id = counter;

counter++;

}

**public** **double** getWeight() {

**return** weight;

}

**public** **void** setWeight(**double** w) {

weight = w;

}

**public** **void** setDeltaWeight(**double** w) {

prevDeltaWeight = deltaWeight;

deltaWeight = w;

}

**public** **double** getPrevDeltaWeight() {

**return** prevDeltaWeight;

}

**public** Neuron getFromNeuron() {

**return** leftNeuron;

}

**public** Neuron getToNeuron() {

**return** rightNeuron;

}

}

OUTPUT



**2. Dimensionality Reduction with Principal Component Analysis**

CODE:

package org.apache.spark.examples.mllib;

import java.util.Arrays;

import java.util.List;

import org.apache.spark.SparkConf;

import org.apache.spark.SparkContext;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.mllib.linalg.Matrix;

import org.apache.spark.mllib.linalg.Vector;

import org.apache.spark.mllib.linalg.Vectors;

import org.apache.spark.mllib.linalg.distributed.RowMatrix;

public class JavaPCAExample {

public static void main(String[] args) {

SparkConf conf = new SparkConf().setAppName("PCA Example");

SparkContext sc = new SparkContext(conf);

JavaSparkContext jsc = JavaSparkContext.fromSparkContext(sc);

List<Vector> data = Arrays.asList(

Vectors.sparse(5, new int[] {1, 3}, new double[] {1.0, 7.0}),

Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),

Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0)

);

JavaRDD<Vector> rows = jsc.parallelize(data);

RowMatrix mat = new RowMatrix(rows.rdd());

Matrix pc = mat.computePrincipalComponents(4);

RowMatrix projected = mat.multiply(pc);

Vector[] collectPartitions = (Vector[])projected.rows().collect();

System.out.println("Projected vector of principal component:");

for (Vector vector : collectPartitions) {

System.out.println("\t" + vector);

}

jsc.stop();

}

}

**3. Gaussian Mixture Model based clustering.**

CODE:

package org.openimaj.ml.gmm;

import java.util.Arrays;

import java.util.EnumSet;

import org.apache.commons.math.util.MathUtils;

import org.openimaj.math.matrix.MatrixUtils;

import org.openimaj.math.statistics.MeanAndCovariance;

import org.openimaj.math.statistics.distribution.AbstractMultivariateGaussian;

import org.openimaj.math.statistics.distribution.DiagonalMultivariateGaussian;

import org.openimaj.math.statistics.distribution.FullMultivariateGaussian;

import org.openimaj.math.statistics.distribution.MixtureOfGaussians;

import org.openimaj.math.statistics.distribution.MultivariateGaussian;

import org.openimaj.math.statistics.distribution.SphericalMultivariateGaussian;

import org.openimaj.ml.clustering.DoubleCentroidsResult;

import org.openimaj.ml.clustering.kmeans.DoubleKMeans;

import org.openimaj.util.array.ArrayUtils;

import org.openimaj.util.pair.IndependentPair;

import Jama.Matrix;

import gnu.trove.list.array.TDoubleArrayList;

public class GaussianMixtureModelEM {

public static enum CovarianceType {

Spherical {

@Override

protected void setCovariances(MultivariateGaussian[] gaussians, Matrix cv) {

double mean = 0;

for (int i = 0; i < cv.getRowDimension(); i++)

for (int j = 0; j < cv.getColumnDimension(); j++)

mean += cv.get(i, j);

mean /= (cv.getColumnDimension() \* cv.getRowDimension());

for (final MultivariateGaussian mg : gaussians) {

((SphericalMultivariateGaussian) mg).variance = mean;

}

}

@Override

protected MultivariateGaussian[] createGaussians(int ngauss, int ndims) {

final MultivariateGaussian[] arr = new MultivariateGaussian[ngauss];

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

arr[i] = new SphericalMultivariateGaussian(ndims);

}

return arr;

}

@Override

protected void mstep(EMGMM gmm, GaussianMixtureModelEM learner, Matrix X, Matrix responsibilities,

Matrix weightedXsum,

double[] norm)

{

final Matrix avgX2uw = responsibilities.transpose().times(X.arrayTimes(X));

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

final Matrix weightedXsumi = new Matrix(new double[][] { weightedXsum.getArray()[i] });

final Matrix avgX2uwi = new Matrix(new double[][] { avgX2uw.getArray()[i] });

final Matrix avgX2 = avgX2uwi.times(norm[i]);

final Matrix mu = ((AbstractMultivariateGaussian) gmm.gaussians[i]).mean;

final Matrix avgMeans2 = MatrixUtils.pow(mu, 2);

final Matrix avgXmeans = mu.arrayTimes(weightedXsumi).times(norm[i]);

final Matrix covar = MatrixUtils.plus(avgX2.minus(avgXmeans.times(2)).plus(avgMeans2),

learner.minCovar);

((SphericalMultivariateGaussian) gmm.gaussians[i]).variance = MatrixUtils.sum(covar)

/ X.getColumnDimension();

}

}

},

Diagonal {

@Override

protected void setCovariances(MultivariateGaussian[] gaussians, Matrix cv) {

for (final MultivariateGaussian mg : gaussians) {

((DiagonalMultivariateGaussian) mg).variance = MatrixUtils.diagVector(cv);

}

}

@Override

protected MultivariateGaussian[] createGaussians(int ngauss, int ndims) {

final MultivariateGaussian[] arr = new MultivariateGaussian[ngauss];

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

arr[i] = new DiagonalMultivariateGaussian(ndims);

}

return arr;

}

@Override

protected void mstep(EMGMM gmm, GaussianMixtureModelEM learner, Matrix X, Matrix responsibilities,

Matrix weightedXsum,

double[] norm)

{

final Matrix avgX2uw = responsibilities.transpose().times(X.arrayTimes(X));

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

final Matrix weightedXsumi = new Matrix(new double[][] { weightedXsum.getArray()[i] });

final Matrix avgX2uwi = new Matrix(new double[][] { avgX2uw.getArray()[i] });

final Matrix avgX2 = avgX2uwi.times(norm[i]);

final Matrix mu = ((AbstractMultivariateGaussian) gmm.gaussians[i]).mean;

final Matrix avgMeans2 = MatrixUtils.pow(mu, 2);

final Matrix avgXmeans = mu.arrayTimes(weightedXsumi).times(norm[i]);

final Matrix covar = MatrixUtils.plus(avgX2.minus(avgXmeans.times(2)).plus(avgMeans2),

learner.minCovar);

((DiagonalMultivariateGaussian) gmm.gaussians[i]).variance = covar.getArray()[0];

}

}

},

Full {

@Override

protected MultivariateGaussian[] createGaussians(int ngauss, int ndims) {

final MultivariateGaussian[] arr = new MultivariateGaussian[ngauss];

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

arr[i] = new FullMultivariateGaussian(ndims);

}

return arr;

}

@Override

protected void setCovariances(MultivariateGaussian[] gaussians, Matrix cv) {

for (final MultivariateGaussian mg : gaussians) {

((FullMultivariateGaussian) mg).covar = cv.copy();

}

}

@Override

protected void mstep(EMGMM gmm, GaussianMixtureModelEM learner, Matrix X, Matrix responsibilities,

Matrix weightedXsum,

double[] norm)

{

final int nfeatures = X.getColumnDimension();

for (int c = 0; c < learner.nComponents; c++) {

final Matrix post = responsibilities.getMatrix(0, X.getRowDimension() - 1, c, c).transpose();

final double factor = 1.0 / (ArrayUtils.sumValues(post.getArray()) + 10 \* MathUtils.EPSILON);

final Matrix pXt = X.transpose();

for (int i = 0; i < pXt.getRowDimension(); i++)

for (int j = 0; j < pXt.getColumnDimension(); j++)

pXt.set(i, j, pXt.get(i, j) \* post.get(0, j));

final Matrix argcv = pXt.times(X).times(factor);

final Matrix mu = ((FullMultivariateGaussian) gmm.gaussians[c]).mean;

((FullMultivariateGaussian) gmm.gaussians[c]).covar = argcv.minusEquals(mu.transpose().times(mu))

.plusEquals(Matrix.identity(nfeatures, nfeatures).times(learner.minCovar));

}

}

},

Tied {

// @Override

// protected double[][] logProbability(double[][] x,

// MultivariateGaussian[] gaussians)

// {

// final int ndim = x[0].length;

// final int nmix = gaussians.length;

// final int nsamples = x.length;

// final Matrix X = new Matrix(x);

//

// final double[][] logProb = new double[nsamples][nmix];

// final Matrix cv = ((FullMultivariateGaussian)

// gaussians[0]).covar;

//

// final CholeskyDecomposition chol = cv.chol();

// Matrix cvChol;

// if (chol.isSPD()) {

// cvChol = chol.getL();

// } else {

// // covar probably doesn't have enough samples, so

// // recondition it

// final Matrix m = cv.plus(Matrix.identity(ndim, ndim).timesEquals(

// MixtureOfGaussians.MIN\_COVAR\_RECONDITION));

// cvChol = m.chol().getL();

// }

//

// double cvLogDet = 0;

// final double[][] cvCholD = cvChol.getArray();

// for (int j = 0; j < ndim; j++) {

// cvLogDet += Math.log(cvCholD[j][j]);

// }

// cvLogDet \*= 2;

//

// for (int i = 0; i < nmix; i++) {

// final Matrix mu = ((FullMultivariateGaussian) gaussians[i]).mean;

// final Matrix cvSol = cvChol.solve(MatrixUtils.minusRow(X,

// mu.getArray()[0]).transpose())

// .transpose();

// for (int k = 0; k < nsamples; k++) {

// double sum = 0;

// for (int j = 0; j < ndim; j++) {

// sum += cvSol.get(k, j) \* cvSol.get(k, j);

// }

//

// logProb[k][i] = -0.5 \* (sum + cvLogDet + ndim \* Math.log(2 \*

// Math.PI));

// }

// }

//

// return logProb;

// }

@Override

protected void setCovariances(MultivariateGaussian[] gaussians,

Matrix cv)

{

for (final MultivariateGaussian mg : gaussians) {

((FullMultivariateGaussian) mg).covar = cv;

}

}

@Override

protected MultivariateGaussian[] createGaussians(int ngauss, int ndims) {

final MultivariateGaussian[] arr = new MultivariateGaussian[ngauss];

final Matrix covar = new Matrix(ndims, ndims);

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

arr[i] = new FullMultivariateGaussian(new Matrix(1, ndims), covar);

}

return arr;

}

@Override

protected void mstep(EMGMM gmm, GaussianMixtureModelEM learner, Matrix X, Matrix responsibilities,

Matrix weightedXsum, double[] norm)

{

// Eq. 15 from K. Murphy, "Fitting a Conditional Linear Gaussian

final int nfeatures = X.getColumnDimension();

final Matrix avgX2 = X.transpose().times(X);

final double[][] mudata = new double[gmm.gaussians.length][];

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

mudata[i] = ((FullMultivariateGaussian) gmm.gaussians[i]).mean.getArray()[0];

final Matrix mu = new Matrix(mudata);

final Matrix avgMeans2 = mu.transpose().times(weightedXsum);

final Matrix covar = avgX2.minus(avgMeans2)

.plus(Matrix.identity(nfeatures, nfeatures).times(learner.minCovar))

.times(1.0 / X.getRowDimension());

for (int i = 0; i < learner.nComponents; i++)

((FullMultivariateGaussian) gmm.gaussians[i]).covar = covar;

}

};

protected abstract MultivariateGaussian[] createGaussians(int ngauss, int ndims);

protected abstract void setCovariances(MultivariateGaussian[] gaussians, Matrix cv);

protected abstract void mstep(EMGMM gmm, GaussianMixtureModelEM learner, Matrix X,

Matrix responsibilities, Matrix weightedXsum, double[] inverseWeights);

}

public static enum UpdateOptions {

/\*\*

\* Update the means

\*/

Means,

/\*\*

\* Update the weights

\*/

Weights,

/\*\*

\* Update the covariances

\*/

Covariances

}

protected static class EMGMM extends MixtureOfGaussians {

EMGMM(int nComponents) {

super(null, null);

this.weights = new double[nComponents];

Arrays.fill(this.weights, 1.0 / nComponents);

}

}

private static final double DEFAULT\_THRESH = 1e-2;

private static final double DEFAULT\_MIN\_COVAR = 1e-3;

private static final int DEFAULT\_NITERS = 100;

private static final int DEFAULT\_NINIT = 1;

CovarianceType ctype;

int nComponents;

private double thresh;

private double minCovar;

private int nIters;

private int nInit;

private boolean converged = false;

private EnumSet<UpdateOptions> initOpts;

private EnumSet<UpdateOptions> iterOpts;

public GaussianMixtureModelEM(int nComponents, CovarianceType ctype, double thresh, double minCovar,

int nIters, int nInit, EnumSet<UpdateOptions> iterOpts, EnumSet<UpdateOptions> initOpts)

{

this.ctype = ctype;

this.nComponents = nComponents;

this.thresh = thresh;

this.minCovar = minCovar;

this.nIters = nIters;

this.nInit = nInit;

this.iterOpts = iterOpts;

this.initOpts = initOpts;

if (nInit < 1) {

throw new IllegalArgumentException("GMM estimation requires at least one run");

}

this.converged = false;

}

public GaussianMixtureModelEM(int nComponents, CovarianceType ctype) {

this(nComponents, ctype, DEFAULT\_THRESH, DEFAULT\_MIN\_COVAR, DEFAULT\_NITERS, DEFAULT\_NINIT, EnumSet

.allOf(UpdateOptions.class), EnumSet.allOf(UpdateOptions.class));

}

public boolean hasConverged() {

return converged;

}

public MixtureOfGaussians estimate(Matrix X) {

return estimate(X.getArray());

}

public MixtureOfGaussians estimate(double[][] X) {

final EMGMM gmm = new EMGMM(nComponents);

if (X.length < nComponents)

throw new IllegalArgumentException(String.format(

"GMM estimation with %d components, but got only %d samples", nComponents, X.length));

double max\_log\_prob = Double.NEGATIVE\_INFINITY;

for (int j = 0; j < nInit; j++) {

gmm.gaussians = ctype.createGaussians(nComponents, X[0].length);

if (initOpts.contains(UpdateOptions.Means)) {

// initialise using k-means

final DoubleKMeans km = DoubleKMeans.createExact(nComponents);

final DoubleCentroidsResult means = km.cluster(X);

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

((AbstractMultivariateGaussian) gmm.gaussians[i]).mean.getArray()[0] = means.centroids[i];

}

}

if (initOpts.contains(UpdateOptions.Weights)) {

gmm.weights = new double[nComponents];

Arrays.fill(gmm.weights, 1.0 / nComponents);

}

if (initOpts.contains(UpdateOptions.Covariances)) {

// cv = np.cov(X.T) + self.min\_covar \* np.eye(X.shape[1])

final Matrix cv = MeanAndCovariance.computeCovariance(X);

ctype.setCovariances(gmm.gaussians, cv);

}

// EM algorithm

final TDoubleArrayList log\_likelihood = new TDoubleArrayList();

// reset converged to false

converged = false;

double[] bestWeights = null;

MultivariateGaussian[] bestMixture = null;

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

// Expectation step

final IndependentPair<double[], double[][]> score = gmm.scoreSamples(X);

final double[] curr\_log\_likelihood = score.firstObject();

final double[][] responsibilities = score.secondObject();

log\_likelihood.add(ArrayUtils.sumValues(curr\_log\_likelihood));

// Check for convergence.

if (i > 0 && Math.abs(log\_likelihood.get(i) - log\_likelihood.get(i - 1)) < thresh) {

converged = true;

break;

}

// Perform the maximisation step

mstep(gmm, X, responsibilities);

// if the results are better, keep it

if (nIters > 0) {

if (log\_likelihood.getQuick(i) > max\_log\_prob) {

max\_log\_prob = log\_likelihood.getQuick(i);

bestWeights = gmm.weights;

bestMixture = gmm.gaussians;

}

}

// check the existence of an init param that was not subject to

// likelihood computation issue.

if (Double.isInfinite(max\_log\_prob) && nIters > 0) {

throw new RuntimeException(

"EM algorithm was never able to compute a valid likelihood given initial " +

"parameters. Try different init parameters (or increasing n\_init) or " +

"check for degenerate data.");

}

if (nIters > 0) {

gmm.gaussians = bestMixture;

gmm.weights = bestWeights;

}

}

}

return gmm;

}

protected void mstep(EMGMM gmm, double[][] X, double[][] responsibilities) {

final double[] weights = ArrayUtils.colSum(responsibilities);

final Matrix resMat = new Matrix(responsibilities);

final Matrix Xmat = new Matrix(X);

final Matrix weighted\_X\_sum = resMat.transpose().times(Xmat);

final double[] inverse\_weights = new double[weights.length];

for (int i = 0; i < inverse\_weights.length; i++)

inverse\_weights[i] = 1.0 / (weights[i] + 10 \* MathUtils.EPSILON);

if (iterOpts.contains(UpdateOptions.Weights)) {

final double sum = ArrayUtils.sumValues(weights);

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

gmm.weights[i] = (weights[i] / (sum + 10 \* MathUtils.EPSILON) + MathUtils.EPSILON);

}

}

if (iterOpts.contains(UpdateOptions.Means)) {

// self.means\_ = weighted\_X\_sum \* inverse\_weights

final double[][] wx = weighted\_X\_sum.getArray();

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

final double[][] m = ((AbstractMultivariateGaussian) gmm.gaussians[i]).mean.getArray();

for (int j = 0; j < m[0].length; j++) {

m[0][j] = wx[i][j] \* inverse\_weights[i];

}

}

}

if (iterOpts.contains(UpdateOptions.Covariances)) {

ctype.mstep(gmm, this, Xmat, resMat, weighted\_X\_sum, inverse\_weights);

}

}

@Override

public GaussianMixtureModelEM clone() {

try {

return (GaussianMixtureModelEM) super.clone();

} catch (final CloneNotSupportedException e) {

throw new RuntimeException(e);

}

}

}

OUTPUT

**4. Expectation–Maximization (EM) algorithm**

CODE:

import java.io.\*;

import java.util.\*;

import java.lang.\*;

import java.math.\*;

class prob\_set

{int zi1;

int zi2;

int zi3;

double x;

double pzi1=0.0;

double pzi2=0.0;

double pzi3=0.0;

}

public class EM {

public static void main(String args[])

{ double m1=5.1211;

double m2=10.222;

double m3=13.21;

double v1=0;

double v2=0;

double v3=0;

double pi1=0.333333;

double pi2,pi3;

pi2=pi3=pi1;

double denominator=0;

int wordcount=6000;

int k=0;

prob\_set p[]=new prob\_set[6000];

while(k<wordcount)

{p[k]=new prob\_set();

k++;}

try

{ FileReader fr=new FileReader("em\_data.txt");

Scanner scan=new Scanner(fr);

int y=0;

while(scan.hasNext())

{ p[y].x=Double.parseDouble(scan.next());

v1=v1+Math.pow((p[y].x-m1),2);

v2=v2+Math.pow((p[y].x-m2),2);

v3=v3+Math.pow((p[y].x-m3),2);

y++;

}

v1=1.0;v2=1.0;v3=1.0;

System.out.println("variance 1 :"+v1+" variance 2 :"+v2+" variance 3 :"+v3);

System.out.println("exp ="+Math.exp(1));

}

catch(Exception e){}

int iter=0;

while(iter<250)

{ int i=0;

while(i<6000)

{double g1=Math.exp(-Math.pow((p[i].x-m1),2)/2/Math.sqrt(v1));

g1=g1/(Math.sqrt(2));

g1=g1/(Math.sqrt(3.14159));

g1=g1/(Math.sqrt(v1));

p[i].pzi1=pi1\*g1;

double g2=Math.exp(-Math.pow((p[i].x-m2),2)/2/Math.sqrt(v2));

g2=g2/(Math.sqrt(2));

g2=g2/(Math.sqrt(3.14159));

g2=g2/(Math.sqrt(v2));

p[i].pzi2=pi2\*g2;

double g3=Math.exp(-Math.pow((p[i].x-m3),2)/2/Math.sqrt(v3));

g3=g3/(Math.sqrt(2));

g3=g3/(Math.sqrt(3.14159));

g3=g3/(Math.sqrt(v3));

p[i].pzi3=pi3\*g3;

denominator=p[i].pzi1+p[i].pzi2+p[i].pzi3;

p[i].pzi1=p[i].pzi1/denominator;

p[i].pzi2=p[i].pzi2/denominator;

p[i].pzi3=p[i].pzi3/denominator;

if(Double.isNaN(p[i].pzi1) || p[i].pzi1<0.00000000000001)

p[i].pzi1=0.00000000000001;

if(Double.isNaN(p[i].pzi2) || p[i].pzi2<0.00000000000001)

p[i].pzi2=0.00000000000001;

if(Double.isNaN(p[i].pzi3) || p[i].pzi3<0.00000000000001)

p[i].pzi3=0.00000000000001;i++;

}

int h=0;

while(h<6000)

{

pi1=pi1+p[h].pzi1;

pi2=pi3+p[h].pzi2;

pi3=pi3+p[h].pzi3;

m1=m1+(p[h].x)\*(p[h].pzi1);

m2=m2+(p[h].x)\*(p[h].pzi2);

m3=m3+(p[h].x)\*(p[h].pzi3);

v1=v1+(Math.pow((p[h].x-m1),2))\*(p[h].pzi1);

v2=v2+(Math.pow((p[h].x-m2),2))\*(p[h].pzi2);

v3=v3+(Math.pow((p[h].x-m3),2))\*(p[h].pzi3);

h++;}

m1=m1/pi1;

m2=m2/pi2;

m3=m3/pi3;

v1=v1/pi1;

v2=v2/pi2;

v3=v3/pi3;

pi1=pi1/(double)6000;

pi2=pi2/(double)6000;

pi3=pi3/(double)6000;

System.out.println("Mean 1="+m1);

System.out.println("Mean 2="+m2);

System.out.println("Mean 3="+m3);

System.out.println(p[0].pzi1);

System.out.println(p[0].pzi2);

System.out.println(p[0].pzi3);

iter++;

}

}

}

OUTPUT

**5. Steepest Gradient Descent Algorithm**

CODE:

import java.awt.geom.Point2D;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.function.DoubleBinaryOperator;  
import java.util.function.DoubleUnaryOperator;  
  
public class SimpleGradientDescent {  
  
 private static final double epsilon = Double.MIN\_VALUE;  
  
 public static void main(String[] args) {  
 new SimpleGradientDescent().run();  
 }  
  
 private void run() {  
 List<Point2D> data = loadData();  
 double alpha = 0.01;  
 int maxIterations = 10\_000;  
 Point2D finalTheta = singleVarGradientDescent(data, 0.1, 0.1, alpha, maxIterations);  
 System.out.printf("theta0 = %f, theta1 = %f", finalTheta.getX(), finalTheta.getY());  
 }  
  
 private Point2D singleVarGradientDescent(List<Point2D> data, double initialTheta0, double initialTheta1, double alpha, int maxIterations) {  
 double theta0 = initialTheta0, theta1 = initialTheta1;  
 double oldTheta0 = 0, oldTheta1 = 0;  
  
 for (int i = 0 ; i < maxIterations; i++) {  
 if (hasConverged(oldTheta0, theta0) && hasConverged(oldTheta1, theta1))  
 break;  
  
 oldTheta0 = theta0;  
 oldTheta1 = theta1;  
  
 theta0 = theta0 - (alpha \* gradientofThetaN(theta0, theta1, data, x -> 1.0));  
 theta1 = theta1 - (alpha \* gradientofThetaN(theta0, theta1, data, x -> x));  
 }  
 return new Point2D.Double(theta0, theta1);  
 }  
  
 private boolean hasConverged(double old, double current) {  
 return (current - old) < epsilon;  
 }  
  
 private double gradientofThetaN(double theta0, double theta1, List<Point2D> data, DoubleUnaryOperator factor) {  
 double m = data.size();  
 return (1.0 / m) \* sigma(data, (x, y) -> (hypothesis(theta0, theta1, x) - y) \* factor.applyAsDouble(x));  
 }  
  
 private double hypothesis(double theta0, double theta1, double x) {  
 return theta0 + (theta1 \* x);  
 }  
  
 private double sigma(List<Point2D> data, DoubleBinaryOperator inner) {  
 return data.stream()  
 .mapToDouble(theta -> {  
 double x = theta.getX(), y = theta.getY();  
 return inner.applyAsDouble(x, y);  
 })  
 .sum();  
 }  
  
 private List<Point2D> loadData() {  
 List<Point2D> data = new ArrayList<>();  
 data.add(new Point2D.Double(1, 2));  
 data.add(new Point2D.Double(2, 3));  
 data.add(new Point2D.Double(3, 4));  
 data.add(new Point2D.Double(4, 5));  
 data.add(new Point2D.Double(5, 6));  
 data.add(new Point2D.Double(6, 7));  
 return data;  
 }  
  
}

OUTPUT