```java
/*
 *  Written by Masaki Uto (2017/04/1)
 *  The following libraries are required.
 *  1. commons-math3-3.2.jar
 *  2. uncommons-maths-1.2.2.jar
 */
import java.io.IOException;
import java.util.Arrays;
import java.util.Random;

import org.apache.commons.math3.linear.MatrixUtils;
import org.apache.commons.math3.linear.RealMatrix;
import org.uncommons.maths.random.GaussianGenerator;

public class MaximumLikelihood {
    // サンプルサイズの指定
    final int N = 1000;
    // W0, W1, W2, sigma の真値設定
    final double[] W = { 1.0, -1.0, -0.5, 1.5 };
    // 収束条件
    final double cond = 0.001;
    // 学習率（eta: 最急上昇法、etaNewton:ニュトン法で使用）
    final double eta = 0.0001;
    final double etaNewton = 1.0;
    // 推定アルゴリズムの指定: steepest or newton
    final String method = "newton";

    // 説明変数XはN(0, 1)から生成
    private double[][] X;
    private double[] Y, E;
    // W0, W1, W2, sigma のパラメータ推定値
    private double[] eW;

    MaximumLikelihood() {
        init();
        if(method.equals("steepest")){
            runSteepest();
        } else if(method.equals("newton")){
            runNewton();
        }
    }

    void init() {
        X = new double[N][2];
        Y = new double[N];
        E = new double[N];
        // データの生成
        generate();
        // 推定値の初期値設定
        eW = new double[4];
        eW[3] = 1.0;
    }

    void generate() {
        Random rand = new Random();
        GaussianGenerator gg = new GaussianGenerator(0, 1, rand);
```

```
57          GaussianGenerator gge = new GaussianGenerator(0, W[3], rand);
58          for (int n = 0; n < N; n++) {
59              X[n][0] = gg.nextValue();
60              X[n][1] = gg.nextValue();
61              E[n] = gge.nextValue();
62              Y[n] = W[0] + W[1] * X[n][0] + W[2] * Math.pow(X[n][1], 2) + E[n];
63          }
64      }
65
66      double[] getGradient() {
67          double[] grad = new double[4];
68          for (int n = 0; n < N; n++) {
69              double Z = Y[n] - eW[0] - eW[1] * X[n][0] - eW[2] * Math.pow(X[n][1], 2);
70              grad[0] += Z / Math.pow(eW[3], 2);
71              grad[1] += X[n][0] * Z / Math.pow(eW[3], 2);
72              grad[2] += Math.pow(X[n][1], 2) * Z / Math.pow(eW[3], 2);
73              grad[3] += Math.pow(Z, 2) / Math.pow(eW[3], 3);
74          }
75          grad[3] -= N / eW[3];
76          return grad;
77      }
78
79      double[][] getInvHesse() {
80          double[][] Hesse = new double[4][4];
81          for (int n = 0; n < N; n++) {
82              double Z = Y[n] - eW[0] - eW[1] * X[n][0] - eW[2] * Math.pow(X[n][1], 2);
83              Hesse[0][0] += -1.0 / Math.pow(eW[3], 2);
84              Hesse[0][1] += -X[n][0] / Math.pow(eW[3], 2);
85              Hesse[0][2] += -Math.pow(X[n][1], 2) / Math.pow(eW[3], 2);
86              Hesse[0][3] += -2 * Z / Math.pow(eW[3], 3);
87              Hesse[1][1] += -Math.pow(X[n][0], 2) / Math.pow(eW[3], 2);
88              Hesse[1][2] += -X[n][0] * Math.pow(X[n][1], 2) / Math.pow(eW[3], 2);
89              Hesse[1][3] += -2 * X[n][0] * Z / Math.pow(eW[3], 3);
90              Hesse[2][2] += -Math.pow(X[n][1], 4) / Math.pow(eW[3], 2);
91              Hesse[2][3] += -2 * Math.pow(X[n][1], 2) * Z / Math.pow(eW[3], 3);
92              Hesse[3][3] += -3 * Math.pow(Z, 2) / Math.pow(eW[3], 4);
93          }
94          Hesse[3][3] += N / Math.pow(eW[3], 2);
95          Hesse[1][0] = Hesse[0][1];
96          Hesse[2][0] = Hesse[0][2];
97          Hesse[3][0] = Hesse[0][3];
98          Hesse[2][1] = Hesse[1][2];
99          Hesse[3][1] = Hesse[1][3];
100         Hesse[3][2] = Hesse[2][3];
101
102         double[][] invHesse = new double[4][4];
103         RealMatrix mA = MatrixUtils.createRealMatrix(Hesse);
104         RealMatrix m = MatrixUtils.blockInverse(mA,0);
105         for (int i = 0; i < m.getRowDimension(); i++) {
106             for (int j = 0; j < m.getColumnDimension(); j++) {
107                 invHesse[i][j] = m.getEntry(i, j);
108             }
109         }
110         return invHesse;
111     }
112
113     void runSteepest() {
114         printCurrentParam();
```

```
115         while (true) {
116             double[] g = getGradient();
117             boolean flag = true;
118             for (int p = 0; p < W.length; p++) {
119                 double delta = g[p] * eta;
120                 eW[p] += delta;
121                 if (Math.abs(delta) > cond) flag = false;
122             }
123             printCurrentParam();
124             if (flag) break;
125         }
126     }
127 
128     void runNewton() {
129         printCurrentParam();
130         while (true) {
131             double[] g = getGradient();
132             double[][] invH = getInvHesse();
133             boolean flag = true;
134             double[] delta = new double[W.length];
135             for (int p = 0; p < W.length; p++) {
136                 for (int q = 0; q < W.length; q++) {
137                     delta[p] += invH[p][q] * g[q];
138                 }
139                 if (Math.abs(delta[p]) > cond) flag = false;
140             }
141             for (int p = 0; p < W.length; p++) {
142                 eW[p] += - delta[p] * etaNewton;
143             }
144             printCurrentParam();
145             if (flag) break;
146         }
147     }
148 
149     void printCurrentParam() {
150         System.out.println(Arrays.toString(eW).replace("[","").replace("]",""));
151     }
152 
153     public static void main(String[] args) throws IOException {
154         new MaximumLikelihood();
155     }
156 }
157 
```