import java.io.BufferedWriter;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileWriter;

import java.io.IOException;

import java.text.DecimalFormat;

import java.text.NumberFormat;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Random;

import java.util.Scanner;

import java.util.stream.IntStream;

public class kNN2 {

    /\*\*

     \* @author Erdem Elik

     \* @version 1.1

     \*/

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

        NumberFormat formatter = new DecimalFormat("###.##");

        // Data Parsed

        final List<List<Float>> trainData = new ArrayList<>(parseData(new File("train\_data.txt")));

        final List<List<Float>> testData = new ArrayList<>(parseData(new File("test\_data.txt")));

        // Label Parsed

        final List<Integer> trainLabel = new ArrayList<>(parseLabel(new File("train\_label.txt")));

        final List<Integer> testLabel = new ArrayList<>(parseLabel(new File("test\_label.txt")));

        // Will contain labels and data paired

        Map<Integer, List<List<Float>>> trainPairsMap = new HashMap<>();

        Map<Integer, List<List<Float>>> testPairsMap = new HashMap<>();

        // Simply contains all "0" labeled trainData in index 0 and all "1" labeled in

        // index 1

        List<List<List<Float>>> trainDataGrouped = groupAllData(trainLabel, trainData);

        trainPairsMap.put(0, trainDataGrouped.get(0));

        trainPairsMap.put(1, trainDataGrouped.get(1));

        // Simply contains all "0" labeled testData in index 0 and all "1" labeled in

        // index 1

        List<List<List<Float>>> testDataGrouped = groupAllData(testLabel, testData);

        testPairsMap.put(0, testDataGrouped.get(0));

        testPairsMap.put(1, testDataGrouped.get(1));

        // Stores predicted labels

        List<Integer> euclideanPredictedLabels = new ArrayList<>(

                predictLabel(calculateEuclideanDistances(testData, trainData), trainLabel));

        List<Integer> manhattanPredictedLabels = new ArrayList<>(

                predictLabel(calculateManhattanDistances(testData, trainData), trainLabel));

        System.out.println("Euclidean: " + calculateAccuracy(euclideanPredictedLabels,

                testLabel) + "%");

        System.out.println("Manhattan: " + formatter.format(calculateAccuracy(manhattanPredictedLabels,

                testLabel)) + "%");

        String[] chromosomeSet = generateInitialPopulation(5, testData.get(0).size());

        formatter.format(calculateGeneticAlgorithm(testData, trainData, testLabel, trainLabel, chromosomeSet));

    }

    //////////////////////////////////////////////

    //////////////// DATA PARSING ////////////////

    //////////////////////////////////////////////

    /\*\*

     \* Parses file to float and creates a 2D List of features

     \*

     \* @param file Train and test data to parse

     \* @return List<List<Float>> parsed data

     \*/

    public static List<List<Float>> parseData(File file) throws FileNotFoundException {

        List<List<Float>> dataList = new ArrayList<>();

        Scanner dataScanner = new Scanner(file);

        while (dataScanner.hasNextLine()) {

            dataList.add(new ArrayList<>(Arrays.asList(

                    Arrays.stream(dataScanner.nextLine().split(" ")).map(Float::valueOf).toArray(Float[]::new))));

        }

        dataScanner.close();

        return dataList;

    }

    ///////////////////////////////////////////////

    //////////////// LABEL PARSING ////////////////

    ///////////////////////////////////////////////

    /\*\*

     \* Parses String labels to Integer and creates an List<Integer>

     \*

     \* @param file File containing labels

     \* @return A List of labels as Integer

     \*/

    public static List<Integer> parseLabel(File file) throws FileNotFoundException {

        List<Integer> labelArr = new ArrayList<>();

        Scanner labelReader = new Scanner(file);

        while (labelReader.hasNext()) {

            labelArr.add(Integer.parseInt(labelReader.next()));

        }

        labelReader.close();

        return labelArr;

    }

    ////////////////////////////////////////////////

    //////////////// END OF PARSING ////////////////

    ////////////////////////////////////////////////

    ///////////////////////////////////////////////

    //////////////// DATA GROUPING ////////////////

    ///////////////////////////////////////////////

    /\*\*

     \* Iterates through the dataSet, matching each index with the corresponding

     \* entry in "labelSet" to determine the appropriate group for adding Lists of

     \* floats.

     \* //FIXME: HARDCODED

     \*

     \* @param labelSet contains the labels

     \* @param dataSet  contains the data

     \* @return 3D List of float with index 0 being the "0" labeled group and index 1

     \*         being the "1" labeled group

     \*/

    public static List<List<List<Float>>> groupAllData(List<Integer> labelSet, List<List<Float>> dataSet) {

        List<List<Float>> ListOf0 = new ArrayList<>();

        List<List<Float>> ListOf1 = new ArrayList<>();

        List<List<List<Float>>> ListOfBoth = new ArrayList<>();

        for (int i = 0; i < labelSet.size() && i < dataSet.size(); i++) {

            if (labelSet.get(i) == 0)

                ListOf0.add(dataSet.get(i));

            else if (labelSet.get(i) == 1) {

                ListOf1.add(dataSet.get(i));

            }

        }

        ListOfBoth.add(ListOf0);

        ListOfBoth.add(ListOf1);

        return ListOfBoth;

    }

    //////////////////////////////////////////////////////////////

    //////////////// DATA IS READY FOR PROCESSING ////////////////

    //////////////////////////////////////////////////////////////

    //////////////////////////////////////////////

    //////////////// CALCULATIONS ////////////////

    //////////////////////////////////////////////

    /\*\*

     \* Calculates the Euclidean distance of every test pattern to all training

     \* patterns

     \*

     \* @param testSet  Test Patterns

     \* @param trainSet Train Patterns

     \* @return 2D List of Euclidean distances

     \*/

    private static List<List<Float>> calculateEuclideanDistances(List<List<Float>> testInput,

            List<List<Float>> trainInput) {

        List<List<Float>> distances = new ArrayList<>();

        List<Float> innerArr;

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

            innerArr = new ArrayList<>();

            for (int j = 0; j < trainInput.size(); j++) {

                double sum = 0;

                for (int k = 0; k < testInput.get(i).size(); k++) {

                    sum += Math.pow(testInput.get(i).get(k) - trainInput.get(j).get(k), 2);

                }

                innerArr.add((float) Math.sqrt(sum));

            }

            distances.add(innerArr);

        }

        return distances;

    }

    /\*\*

     \* Calculates the Manhattan distance of every test pattern to all training

     \* patterns

     \*

     \* @param testInput 2D List of test patterns

     \* @param trainMap  Map of the train patterns

     \* @return 2D List of Manhattan distances

     \*/

    private static List<List<Float>> calculateManhattanDistances(List<List<Float>> testInput,

            List<List<Float>> trainInput) {

        List<List<Float>> distances = new ArrayList<>();

        List<Float> innerArr;

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

            innerArr = new ArrayList<>();

            for (int j = 0; j < trainInput.size(); j++) {

                double sum = 0;

                for (int k = 0; k < testInput.get(i).size(); k++) {

                    sum += Math.abs(testInput.get(i).get(k) - trainInput.get(j).get(k));

                }

                innerArr.add((float) sum);

            }

            distances.add(innerArr);

        }

        return distances;

    }

    /\*\*

     \* Compares predictions and actual labels to calculate the accuracy of the model

     \*

     \* @param predictions Prediction labels

     \* @param actuals     Test labels

     \* @return Accuracy of the predictions

     \*/

    public static Double calculateAccuracy(List<Integer> predictions, List<Integer> actuals) {

        try {

            return (double) IntStream.range(0, actuals.size()).filter(i -> actuals.get(i).equals(predictions.get(i)))

                    .count() / actuals.size() \* 100;

        } catch (IndexOutOfBoundsException e) {

            return null;

        }

    }

    /////////////////////////////////////////////////////

    //////////////// END OF CALCULATIONS ////////////////

    /////////////////////////////////////////////////////

    /////////////////////////////////////////////////////////////

    //////////////// PREDICTIONS & FILE PRINTING ////////////////

    /////////////////////////////////////////////////////////////

    /\*\*

     \* Finds the label of the nearest neighbour

     \*

     \* @param inputToPredict List containing distances

     \* @param trainLabels    List of TRAINING labels

     \* @return List of predictions

     \*/

    public static List<Integer> predictLabel(List<List<Float>> inputToPredict, List<Integer> trainLabels) {

        List<Integer> predictions = new ArrayList<>();

        for (List<Float> innerArr : inputToPredict) {

            int minIndex = trainLabels.get(innerArr.indexOf(Collections.min(innerArr)));

            predictions.add(minIndex);

        }

        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output2.txt"))) {

            writer.write(predictions.toString());

        } catch (IOException e) {

            e.printStackTrace();

        }

        return predictions;

    }

    //////////////////////////////////////////////////////////

    //////////////// BINARY GENETIC ALGORITHM ////////////////

    //////////////////////////////////////////////////////////

    public static Double shortcut(List<List<Float>> testData, List<List<Float>> trainData, List<Integer> testLabel,

            List<Integer> trainLabel) {

        return calculateAccuracy(predictLabel(calculateEuclideanDistances(testData, trainData), trainLabel), testLabel);

    }

    /\*\*

     \* //TODO:

     \*

     \* @param testSet

     \* @param trainSet

     \* @param chromosomeSet

     \*/

    public static Double calculateGeneticAlgorithm(List<List<Float>> testSet, List<List<Float>> trainSet,

            List<Integer> testLabel, List<Integer> trainLabel, String[] chromosomeSet) {

        final List<List<Float>> localTestSet = new ArrayList<>(testSet);

        final List<List<Float>> localTrainSet = new ArrayList<>(trainSet);

        List<List<Integer>> indices = new ArrayList<>(findIndicesofOnes(chromosomeSet));

        List<List<Float>> modifiedTestSet = new ArrayList<>();

        List<List<Float>> modifiedTrainSet = new ArrayList<>();

        String bestChromosome = chromosomeSet[listOfResults.indexOf(Collections.max(listOfResults))];

        String worstChromosome = chromosomeSet[listOfResults.indexOf(Collections.min(listOfResults))];

        System.out.println("Best is " + bestChromosome + " with " + Collections.max(listOfResults) + "%");

        double generationSuccessPercentage = 0;

        while(generationSuccessPercentage < 80.0) {

            generationSuccessPercentage = shortcut(modifiedTestSet, modifiedTrainSet, testLabel, trainLabel);

            String[] newGen = chromosomeMutator(bestChromosome, worstChromosome, 4, 2);

            retrieveDataFromBinaryString(trainLabel, findIndicesofOnes(newGen));

        }

        return generationSuccessPercentage;

    }

    public static List<List<Float>> newGenerationAccuracyCalculator(List<List<Integer>> indices, List<List<Float>> modifiedData, String[] chromosomeSet, List<Integer> testLabel, List<Integer> trainLabel, List<List<Float>> constantTestSet, List<List<Float>> constantTrainSet) {

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

            List<List<Float>> modifiedTestSet = new ArrayList<>();

            List<List<Float>> modifiedTrainSet = new ArrayList<>();

            Double result = 0.0;

            modifiedTestSet = retrieveDataFromBinaryString(indices.get(i), constantTestSet); // Ready for distance measuring

            modifiedTrainSet = retrieveDataFromBinaryString(indices.get(i), constantTrainSet); // Ready for distance measuring

            List<Double> listOfResults = new ArrayList<>();

            result = shortcut(modifiedTestSet, modifiedTrainSet, testLabel, trainLabel);

            listOfResults.add(result);

            System.out.println("Binary String: " + chromosomeSet[i] + " causes accuracy of " + result + "%");

        }

        return

    }

    public static String[] chromosomeMutator(String chromosome1, String chromosome2, int offSpringCount, int mutationCount) {

        String[] nextGenerationChromosomes = new String[offSpringCount];

        List<String> nextGenList = new ArrayList<>();

        Random rand = new Random();

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

            String c1 = chromosome1;

            String c2 = chromosome2;

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

                int minMax = Math.min(chromosome1.length(), chromosome2.length());

                int crossoverPointStart = rand.nextInt(5, minMax / 2);

                int crossoverPointEnd = rand.nextInt(crossoverPointStart,minMax);

                c1 = c1.substring(0, crossoverPointStart) + c2.substring(crossoverPointStart, crossoverPointEnd) + c1.substring(crossoverPointEnd);

                c2 = c2.substring(0, crossoverPointStart) + c1.substring(crossoverPointStart, crossoverPointEnd) + c2.substring(crossoverPointEnd);

                System.out.println(crossoverPointStart + " " + crossoverPointEnd);

            }

            nextGenList.add(c1);

            nextGenList.add(c2);

        }

        nextGenerationChromosomes = nextGenList.toArray(new String[0]);

        for(String str : nextGenerationChromosomes){

            System.out.println(str);

        }

        return nextGenerationChromosomes;

    }

    /\*\*

     \* Generates n-number of random binary strings of length 61 to be used as

     \* initial population

     \*

     \* @param initialPopulation starting population of the genetic algorithm

     \* @return String Array of length initialPopulation containing binary strings of

     \*         length 61

     \*/

    public static String[] generateInitialPopulation(int initialPopulation, int length) {

        String[] binaryStrings = new String[initialPopulation];

        Random random = new Random();

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

            StringBuilder binaryString = new StringBuilder();

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

                int randomBit = random.nextInt(2);

                binaryString.append(randomBit);

            }

            binaryStrings[i] = binaryString.toString();

        }

        return binaryStrings;

    }

    /\*\*

     \* //TODO:

     \*

     \* @param chromosomeSet

     \* @return

     \*/

    public static List<List<Integer>> findIndicesofOnes(String[] chromosomeSet) {

        List<List<Integer>> indices = new ArrayList<>();

        for (String binaryString : chromosomeSet) {

            List<Integer> innerIndices = new ArrayList<>();

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

                if (binaryString.charAt(i) == '1') {

                    innerIndices.add(i);

                }

            }

            indices.add(innerIndices);

        }

        return indices;

    }

    /\*\*

     \* //TODO:

     \*

     \* @param list

     \* @param testSet

     \* @param trainSet

     \* @return

     \*/

    public static List<List<Float>> retrieveDataFromBinaryString(List<Integer> indices, List<List<Float>> dataSet) {

        List<List<Float>> retrievedPattern = new ArrayList<>();

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

            List<Float> retrievedDataPoint = new ArrayList<>();

            for (int indexToLookUp : indices) { // Fix variable name from 'list' to 'indices'

                retrievedDataPoint.add(dataSet.get(i).get(indexToLookUp));

            }

            retrievedPattern.add(retrievedDataPoint);

        }

        return retrievedPattern;

    }

            /////////// FILE PRINT ///////////

            // try (

            //         BufferedWriter writer = new BufferedWriter(new FileWriter("aa\\modTest" + i + ".txt"))) {

            //     for (List<Float> innerList : modifiedTestSet) {

            //         for (Float value : innerList) {

            //             writer.write(value + " ");

            //         }

            //         writer.newLine();

            //     }

            //     writer.newLine();

            // } catch (IOException e) {

            //     e.printStackTrace();

            // }

            // try (

            //         BufferedWriter writer = new BufferedWriter(new FileWriter("aa\\modTrain" + i + ".txt"))) {

            //     for (List<Float> innerList : modifiedTrainSet) {

            //         for (Float value : innerList) {

            //             writer.write(value + " ");

            //         }

            //         writer.newLine();

            //     }

            //     writer.newLine();

            // } catch (IOException e) {

            //     e.printStackTrace();

            // }

            //////////////////////////////////

}