CP465 Database II Code

Data Mining Algorithms
Instructor: Ilias Kotsireas
Students: Chris Ye, Evgheni Naida
***Student ID: 104160010*, 090305930**

2013/11/30

November 30, 2013

Code

Table of Contents

ID3:

```
package ID3;

import java.io.*;
import java.util.*;

/**
 * A simple implementation of the id3 algorithm This is a modified version to
 * make my code closer to the standard id3 algorithm
 *
 * @version Nov, 27 2013
 * @author Chris Ye
 * @Email yexx0010@mylaurier.ca
 * @StudentId 104160010
 */


public class ID3 {

	int numAttributes; // The number of attributes including the output
                                        // attribute
	String[] attributeNames; // The names of all attributes. It is an array of dimension numAttributes. The last
attribute
                                                //is the output attribute
	private int atriClass;
	/*
	 * Possible values for each attribute is stored in a vector. domains is an
	 * array of dimension numAttributes. Each element of this array is a vector
	 * that contains values for the corresponding attribute domains[0] is a
	 * vector containing the values of the 0-th attribute, etc.. The last
	 * attribute is the output attribute
	 */
	@SuppressWarnings("rawtypes")
	Vector[] domains;

	/**
	 * This Class is to represent a data point that consisting of numAttributes values of attributes
	 */
	class DataPoint {

		/*
		 * The values of all attributes stored in this array. i-th element in this array is the index to the
element in the vector domains
		 * representing the symbolic value of the attribute. For example, if attributes[2] is 1, then the
actual value of the 2-nd attribute is
		 * obtained by domains[2].elementAt(1). This representation makes comparing values of attributes
easier - it involves only integer
		 * comparison and no string comparison. The last attribute is the output attribute
		 */
		public int[] attributes;

		public DataPoint(int numattributes) {
			attributes = new int[numattributes];
		}
	};
```

```
	/**
	 * The class to represent a node in the decomposition tree.
	 */
	class TreeNode {
		public double entropy; // The entropy of data points if this node is a leaf node
		@SuppressWarnings("rawtypes")
		public Vector data; // The set of data points if this is a leaf node
		public int decompositionAttribute; // If this is not a leaf node, the attribute that is used to divide the set of data points
		public int decompositionValue; // the attribute-value that is used to divide the parent node
		public TreeNode[] children; // If this is not a leaf node, references to the children nodes
		public TreeNode parent; // The parent to this node. The root has parent == null

		@SuppressWarnings("rawtypes")
		public TreeNode() {
			data = new Vector();
		}

	};

	/* The root of the decomposition tree */
	TreeNode root = new TreeNode();

	/**
	 * This function returns an integer corresponding to the symbolic value of the attribute. If the symbol does not exist in the domain,
	 * the symbol is added to the domain of the attribute
	 */
	@SuppressWarnings("unchecked")
	public int getSymbolValue(int attribute, String symbol) {
		int index = domains[attribute].indexOf(symbol);
		if (index < 0) {
			domains[attribute].addElement(symbol);
			return domains[attribute].size() - 1;
		}
		return index;
	}

	/* Returns all the values of the specified attribute in the data set */
	@SuppressWarnings({ "rawtypes", "unchecked" })
	public int[] getAllValues(Vector data, int attribute) {
		Vector values = new Vector();
		int num = data.size();
		for (int i = 0; i < num; i++) {
			DataPoint point = (DataPoint) data.elementAt(i);
			String symbol = (String) domains[attribute].elementAt(point.attributes[attribute]);
			int index = values.indexOf(symbol);
			if (index < 0) {
				values.addElement(symbol);
			}
		}

		int[] array = new int[values.size()];
		for (int i = 0; i < array.length; i++) {
			String symbol = (String) values.elementAt(i);
			array[i] = domains[attribute].indexOf(symbol);
		}
		values = null;
```

```
		return array;
	}

	/**
	 * Returns a subset of data, in which the value of the specfied attribute of
	 * all data points is the specified value
	 */
	@SuppressWarnings({ "rawtypes", "unchecked" })
	public Vector getSubset(Vector data, int attribute, int value) {
		Vector subset = new Vector();

		int num = data.size();
		for (int i = 0; i < num; i++) {
			DataPoint point = (DataPoint) data.elementAt(i);
			if (point.attributes[attribute] == value)
				subset.addElement(point);
		}
		return subset;

	}

	/**
	 * Calculates the entropy of the set of data points. The entropy is calculated using the values of the output attribute which is the last
	 * element in the array attributes
	 */
	@SuppressWarnings("rawtypes")
	public double calculateEntropy( Vector data) {

		int numdata = data.size();
		if (numdata == 0)
			return 0;

		int attribute = atriClass;
		int numvalues = domains[attribute].size();
		double sum = 0;
		for (int i = 0; i < numvalues; i++) {
			int count = 0;
			for (int j = 0; j < numdata; j++) {
				DataPoint point = (DataPoint) data.elementAt(j);
				if (point.attributes[attribute] == i)
					count++;
			}
			double probability = 1. * count / numdata;
			if (count > 0)
				sum += -probability * Math.log(probability);
		}
		return sum;

	}

	/**
	 * This function checks if the specified attribute is used to decompose the data set in any of the parents of the specfied node
	 * in the decomposition tree. Recursively checks the specified node as well as all parents
	 */
	public boolean alreadyUsedToDecompose(TreeNode node, int attribute) {
		if (node.children != null) {
```

```
                if (node.decompositionAttribute == attribute)
                        return true;
                }
                if (node.parent == null)
                        return false;
                return alreadyUsedToDecompose(node.parent, attribute);
        }

        /**
         * This function decomposes the specified node according to the id3 algorithm. Recursively divides all
children nodes until it is not
         * possible to divide any further I have changed this code from my earlier version. I believe that the code
in my earlier version prevents
         * useless decomposition and results in a better decision tree! This is a more faithful implementation of the
standard id3 algorithm
         */
        public void decomposeNode(TreeNode node) {

                double bestEntropy = 0;
                boolean selected = false;
                int selectedAttribute = 0;

                int numdata = node.data.size();
                int numinputattributes = numAttributes - 1;
                node.entropy = calculateEntropy(node.data);
                if (node.entropy == 0)
                        return;

                /*
                 * In the following two loops, the best attribute is located which causes maximum decrease in entropy
                 */
                for (int i = 0; i < numinputattributes; i++) {

                        if (atriClass == i) {
                                continue;
                        }

                        int numvalues = domains[i].size();
                        if (alreadyUsedToDecompose(node, i))
                                continue;
                        // Use the following variable to store the entropy for the test node created with the
attribute i
                        double averageentropy = 0;
                        for (int j = 0; j < numvalues; j++) {
                                @SuppressWarnings("rawtypes")
                                Vector subset = getSubset(node.data, i, j);
                                if (subset.size() == 0)
                                        continue;
                                double subentropy = calculateEntropy(subset);
                                averageentropy += subentropy * subset.size(); // Weighted sum
                        }

                        averageentropy = averageentropy / numdata; // Taking the weighted average
                        if (selected == false) {
                                selected = true;
                                bestEntropy = averageentropy;
                                selectedAttribute = i;
                        } else {
```

```
                        if (averageentropy < bestEntropy) {
                                selected = true;
                                bestEntropy = averageentropy;
                                selectedAttribute = i;
                        }
                }

        }

        if (selected == false)
                return;

        // Now divide the dataset using the selected attribute
        int numvalues = domains[selectedAttribute].size();
        node.decompositionAttribute = selectedAttribute;
        node.children = new TreeNode[numvalues];
        for (int j = 0; j < numvalues; j++) {
                node.children[j] = new TreeNode();
                node.children[j].parent = node;
                node.children[j].data = getSubset(node.data, selectedAttribute, j);
                node.children[j].decompositionValue = j;
        }

        // Recursively divides children nodes
        for (int j = 0; j < numvalues; j++) {
                decomposeNode(node.children[j]);
        }

        // There is no more any need to keep the original vector. Release this memory
        node.data = null; // Let the garbage collector recover this memory

    }

    /**
    * Function to read the data file. The first line of the data file should contain the names of all attributes.
     * The number of attributes is inferred from the number of words in this line. The last word is taken as the
name
     * of the output attribute. Each subsequent line contains the values of attributes for a data point. If any
line starts with
     * Blank lines are ignored.
     */
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public int readData(String filename) throws Exception {

        FileInputStream in = null;

        try {
                File inputFile = new File(filename);
                in = new FileInputStream(inputFile);
        } catch (Exception e) {
                System.err.println("Unable to open data file: " + filename + "\n"
                                + e);
                return 0;
        }

        @SuppressWarnings("resource")
        BufferedReader bin = new BufferedReader(new InputStreamReader(in));
```

```
String input;
while (true) {
	input = bin.readLine();
	if (input == null) {
		System.err.println("No data found in the data file: "
				+ filename + "\n");
		return 0;
	}
	if (input.startsWith("//"))
		continue;
	if (input.equals(""))
		continue;
	break;
}

StringTokenizer tokenizer = new StringTokenizer(input, ", ");
numAttributes = tokenizer.countTokens();
if (numAttributes <= 1) {
	System.err.println("Read line: " + input);
	System.err
			.println("Could not obtain the names of attributes in the line");
	System.err
			.println("Expecting at least one input attribute and one output attribute");
	return 0;
}

domains = new Vector[numAttributes];
for (int i = 0; i < numAttributes; i++)

	domains[i] = new Vector();
attributeNames = new String[numAttributes];

for (int i = 0; i < numAttributes; i++) {
	attributeNames[i] = tokenizer.nextToken();
}

while (true) {
	input = bin.readLine();
	if (input == null)
		break;
	if (input.startsWith("//"))
		continue;
	if (input.equals(""))
		continue;

	tokenizer = new StringTokenizer(input, ", ");
	int numtokens = tokenizer.countTokens();
	if (numtokens != numAttributes) {
		System.err.println("Read " + root.data.size() + " data");
		System.err.println("Last line read: " + input);
		System.err
				.println("Expecting " + numAttributes + " attributes");
		return 0;
	}

	DataPoint point = new DataPoint(numAttributes);
	for (int i = 0; i < numAttributes; i++) {
		point.attributes[i] = getSymbolValue(i, tokenizer.nextToken());
```

```
                }
                root.data.addElement(point);

        }

        bin.close();

        return 1;

    } // End of function readData



    /**
     * This function writes the decision tree in the form of rules to an output file called test_document.txt
     * The action part of the rule is of the form  outputAttribute = "symbolicValue" or outputAttribute =
{ "Value1", "Value2", .. }
     * The second form is wrote if the node cannot be decomposed any further into an homogeneous set
     */
    public void printTree(TreeNode node, String tab, FileOutputStream fop) {

        int outputattr = atriClass;
        String content;

        if (node.children == null) {
            int[] values = getAllValues(node.data, outputattr);
            if (values.length == 1) {
                content = tab + "\t" + attributeNames[outputattr] + " = \""
                        + domains[outputattr].elementAt(values[0]) + "\";" +
System.getProperty("line.separator");
                try {
                    fop.write(content.getBytes());
                } catch (IOException e) {
                    e.printStackTrace();
                }
                return;
            }

            content = tab + "\t" + attributeNames[outputattr] + " = {" +
System.getProperty("line.separator");
            try {
                fop.write(content.getBytes());
            } catch (IOException e) {
                e.printStackTrace();
            }

            for (int i = 0; i < values.length; i++) {
                content = "\"" + domains[outputattr].elementAt(values[i]) + "\" " +
System.getProperty("line.separator");
                try {
                    fop.write(content.getBytes());
                } catch (IOException e) {
                    e.printStackTrace();
                }
                if (i != values.length - 1) {
                    content = " , " + System.getProperty("line.separator");
                    try {
                        fop.write(content.getBytes());
```

```
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
        }
        content = "};" + System.getProperty("line.separator");
        try {
                fop.write(content.getBytes());
        } catch (IOException e) {
                e.printStackTrace();
        }
        return;
}

int numvalues = node.children.length;
for (int i = 0; i < numvalues; i++) {
        content = tab + "if( "
                        + attributeNames[node.decompositionAttribute] + " == \""
                        + domains[node.decompositionAttribute].elementAt(i)
                        + "\") {" + System.getProperty("line.separator");
        try {
                fop.write(content.getBytes());
        } catch (IOException e) {
                e.printStackTrace();
        }

        printTree(node.children[i], tab + "\t", fop);
        if (i != numvalues - 1)
        {
                content =  tab + "} else " + System.getProperty("line.separator");
                try {
                        fop.write(content.getBytes());
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
        else {
                content = tab + "}" + System.getProperty("line.separator");
                try {
                        fop.write(content.getBytes());
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}

}

/**
 * This function creates the decision tree and prints it in the form of rules on the test_document file
 */
public void createDecisionTree() {
        FileOutputStream fop = null;
        File file;
        String filePath = "test_document.txt";
        try {
                file = new File(filePath);
                if (!file.exists()) {
```

```
                        file.createNewFile();
                }
                fop = new FileOutputStream(file);
        } catch (IOException e) {
                e.printStackTrace();
        }

        decomposeNode(root);
        printTree(root, "", fop);

        try {
                fop.flush();
                fop.close();
        } catch (IOException e) {
                e.printStackTrace();
        }
    }

    /** Here is the definition of the main function */
    public static void main(String[] args) throws Exception {

        ID3 me = new ID3();

        Scanner in = new Scanner(System.in);

        System.out.print("File Name: ");
        String str = in.nextLine();

        System.out.print("atrib Class: ");
        me.atriClass = in.nextInt();

        int status = me.readData(str);
        if (status <= 0) {
                return;
        }

        me.createDecisionTree();
        System.out.println("DONE!   check the 'test_document.txt' in root directory");
    }


}
```

Apriori:

```
/**
 * @author     Evgheni Naida
 * @version 2013-11-30
 *                    ID: 090305930
 *                    email: naid5930@mylaurier.ca
 *
 *     Mines Association rules of the database 'transactions.txt'
 * based on config file 'config.txt' which contains # of attributes,  # of transactions
 * and min support in %.
 *     The output 'output.txt' is produced, containing association rules, their frequency
 * and their support.
 *
 */




import java.io.*;
import java.text.DecimalFormat;
import java.util.*;



public class apriori {
    public static void main(String[] args) {
        AprioriAlgorithm apr = new AprioriAlgorithm();
        apr.Process();
    }
}




/**
 * @author     Evgheni Naida
 * @version 2013-11-30
 *                    ID: 090305930
 *                    email: naid5930@mylaurier.ca
 *
 *     AprioriAlgorithm class. Mines Association rules of the database 'transactions.txt'
 * based on config file 'config.txt' which contains # of attributes,  # of transactions
 * and min support in %.
 *     The output 'output.txt' is produced, containing association rules, their frequency
 * and their support.
 *
 */

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
import java.text.DecimalFormat;
import java.util.StringTokenizer;
import java.util.Vector;

/**
 * Generates Apriori itemsets
 */
class AprioriAlgorithm{

       //---------------I/O files
    String configFile="config.txt";         //configuration file
    String transactionsFile="transactions.txt";    //transaction file
    String outputFile="output.txt";                //output file



    int numItems;                                          //#of columns/items
    int numTransactions;                                //number of transactions
    double minSup;                                         //minimum support

    Vector<String> candidates=new Vector<String>(); //the current candidates

    String oneVal[];                                       //value of columns treated as TRUE, array
    String itemSeparator = ",";                    //default item separator



    //--------------------Generates Apriori Itemsets
    public void Process(){
       int itemsetNumber=0;                        //the current itemset
        int z = 0;                                         //frequency itemset counter
        getConfig();
        System.err.println("Apriori has started!!!!!!\n");


        do{                                                                    //loop until complete
            itemsetNumber++;                           //increment the itemset
            generateCandidates(itemsetNumber);     //generate candidates

            z = calculateFrequentItemsets(itemsetNumber);//determine frequent itemsets
            if(candidates.size()!=0){
                System.out.println(itemsetNumber + "-item itemsets: " + (z-1));//print the count of itemsets
            }

        }while(candidates.size()>1);//if <=1 frequent items, then end
        System.out.println("\r\nDone! \r\nFor details check the output file called:   " + outputFile);
    }




    /**
     * Gets Configutaion information from config.txt, allows changes in
     * separator, and TRUE column values
     */
    private void getConfig()
    {
```

```
        FileWriter fw;
        BufferedWriter file_out;

        String input="";

        //Separator change
        System.out.println("\nPress 'C' to change the item separator(default: ','), any other key to continue.  ");
        input=getInput();
        if(input.compareToIgnoreCase("c")==0){
            System.out.print("Enter the separating character for items (return for '"+itemSeparator+"'): ");
            input=getInput();
            if(input.compareToIgnoreCase("")!=0)
                itemSeparator=input;
        }

        try{
               //--------------------Read the config file
             FileInputStream file_in = new FileInputStream(configFile);  //get config file
             BufferedReader data_in = new BufferedReader(new InputStreamReader(file_in)); //read the data
             numItems=Integer.valueOf(data_in.readLine()).intValue();           //1st line, # of colmns
             numTransactions=Integer.valueOf(data_in.readLine()).intValue();//2nd line, # of tnscn
             minSup=(Double.valueOf(data_in.readLine()).doubleValue());         //3rd line, min supp
             minSup/=100.0;                                                                          //min
supp to decimal


            //---------------------Change the TRUE value of each column
            oneVal = new String[numItems];
            System.out.print("Press 'Y' to change what row value is recognized as TRUE (default: 'y'):");
            if(getInput().compareToIgnoreCase("y")==0){
                for(int i=0; i<oneVal.length; i++){
                    System.out.print("Enter value of column #" + (i+1) + ": ");
                    oneVal[i] = getInput();
                }
            }
            else for(int i=0; i<oneVal.length; i++) oneVal[i]="y"; //default is 'y'



            //---------------------Generating Output file
            fw= new FileWriter(outputFile);
            file_out = new BufferedWriter(fw);
            //create the header of the output file, containing important info about config
            file_out.write("Number of Transactions: " + numTransactions + "\r\n");
            file_out.write("Number of Items/Columns: " + numItems + "\r\n");
            file_out.write("Min Support: " + minSup + "\r\n");
            file_out.write("-------------------------------------------------------------\r\n");
            file_out.write("Column values considered as True:\r\n");
            for(int i=0; i<oneVal.length; i++){
                file_out.write("\tColumn" + (i+1) + ": " + oneVal[i] + "\r\n");
            }

            file_out.write("\r\n");
            file_out.close();
        }


        catch(IOException e){ System.out.println(e); }
```

```
}



/**
 * Generates all possible candidates for the n-th itemsets,
 * candidates are stored in the candidates vector
 *
 * @param: int n, current itemset
 */
private void generateCandidates(int set){
    String string1, string2;                    //strings that will be used for comparisons
    StringTokenizer st1, st2;                   //string tokenizers
    Vector<String> tmpCand =
                  new Vector<String>();                 //temp candidate vector


    if(set == 1){//first set case
        for(int i = 1; i <= numItems; i++){ tmpCand.add(Integer.toString(i));
        }
    }
    else if(set == 2){ //second set case
        for(int i = 0; i < candidates.size(); i++){
            st1 = new StringTokenizer(candidates.get(i));
            string1 = st1.nextToken();
            for(int j = i+1; j < candidates.size(); j++){
                st2 = new StringTokenizer(candidates.elementAt(j));
                string2 = st2.nextToken();
                tmpCand.add(string1 + " " + string2);
            }
        }
    }
    else{//all other set cases
        for(int i=0; i<candidates.size(); i++){
            for(int j=i+1; j<candidates.size(); j++){
                string1 = new String();
                string2 = new String();
                st1 = new StringTokenizer(candidates.get(i));
                st2 = new StringTokenizer(candidates.get(j));

                for(int s = 0; s < set-2; s++){
                    string1 = string1 + " " + st1.nextToken();
                    string2 = string2 + " " + st2.nextToken();
                }
                if(string2.compareToIgnoreCase(string1)==0)//if same n-2 tokens, add
                    tmpCand.add((string1 + " " + st1.nextToken() + " " + st2.nextToken()).trim());
            }
        }
    }

    candidates.clear();                                 //del old cand
    candidates = new Vector<String>(tmpCand);//store new
    tmpCand.clear();
}
```

```
    /**
     * Identify frequency in the itemsets, based on min supp
     *
     * @param: int n, itemset to evaluate
     * @return: int z, count of frequent itemsets, that meet minSup condition
     */
    private int calculateFrequentItemsets(int n){
        Vector<String> frequentCandidates = new Vector<String>(); //the frequent candidates for the
                                                                  //current itemset, with proper support
        FileInputStream file_in;                                  //file input stream
        BufferedReader data_in;                                   //data input stream
        BufferedWriter file_out;                                  //output file
        FileWriter fw;                                                          //file writer obj
        StringTokenizer stCandidate, stTransaction;               //tokenizer
        boolean itemFound;                                                      //true, if transaction
matches itemset
        boolean transAttributes[] = new boolean[numItems];        //array, holding attribs of transaction after delim
        int count[] = new int[candidates.size()];                 //count intemFinds
        int z = 1;                                                              //frequency itemset
counter


        try{
                fw= new FileWriter(outputFile, true);
                file_out = new BufferedWriter(fw);

                file_in = new FileInputStream(transactionsFile);
                data_in = new BufferedReader(new InputStreamReader(file_in));


                //-----------Count the number of occurences
                for(int i=0; i<numTransactions; i++){ //iterate each transaction, store in array
                      stTransaction = new StringTokenizer(data_in.readLine(), itemSeparator); //read transaction
                    for(int j=0; j<numItems; j++){
                      transAttributes[j]=(stTransaction.nextToken().compareToIgnoreCase(oneVal[j])==0);
                    }
                    for(int c=0; c<candidates.size(); c++){ //check each candidate
                        itemFound = false;
                        stCandidate = new StringTokenizer(candidates.get(c));//see what items
                        while(stCandidate.hasMoreTokens()){//check if item is in transaction
                            itemFound = (transAttributes[Integer.valueOf(stCandidate.nextToken())-1]);
                            if(!itemFound) break;
                        }
                        if(itemFound) count[c]++; //count if found
                    }

                }


                //--------------------Write to File all the candidates with proper support
                file_out.write("---------------Frequency " + n + "-itemsets --------------\r\n" );
                for(int i=0; i<candidates.size(); i++){
                      double support = count[i]/(double)numTransactions;
                      if(support>=minSup){//each candidate with >min supp
                        frequentCandidates.add(candidates.get(i)); //add to vector
```

```
                        DecimalFormat format = new DecimalFormat("0.000"); //formatting
                        file_out.write("frequency-itemset " + z + ": [" + candidates.get(i)
                                    + "]" + "\t\t#ofT: " + count[i] + "\t\tsup: " + format.format(support) +
"\r\n" );
                        z++;
                    }
                }
                file_out.write("\r\n");
                file_out.close();
        }

        catch(IOException e) { System.out.println(e); } //Catch I/O error

        candidates.clear();//clear old candidates and store new candidates
        candidates = new Vector<String>(frequentCandidates);
        frequentCandidates.clear();
        return z;
    }



    /**
         * Gets user input from System.in
     * @return: String, user input
     */
    public static String getInput(){
        String input="";
        BufferedReader inpt = new BufferedReader(new InputStreamReader(System.in));

        try{ input = inpt.readLine();} //error handling
        catch (Exception e){ System.out.println(e);}
        return input;
    }



}
```