5. SOFTWARE DEFECT PREDICTION - PREDICTIVE CLASSIFICATION FRAMEWORK

5.1. INTRODUCTION

Considering the fast development of the product improvement application transform in the present days, programming advancement application holds a few deserts in presenting/executing programming items. They are expensive and powerful movement programming advancement in testing the result of the product. Customarily a portion of the information digging systems were produced to distinguish programming deformity forecast from different information set applications from recorded information. One pass calculation is one of the systems for getting to administrations and different gimmicks of the preparing units continuously programming application improvement including the peculiarities of programming application like item cost and testing item. Programming quality and testing proficiency are the fundamental peculiarities in programming deformity forecast. So in this paper we propose to create prescient arrangement calculation to reduce expenses of the product testing advancement and expensive estimation for programming application process. This method proposed to create programming quality and testing proficiency in by developing prescient modules exhibit code properties in discharged thing sets. In this strategy, use information affiliation guideline digging occasions for identifying backing and trust for every information thing present progressively programming application improvement with characteristic representation. This methodology is helpful to designers to locate programming deserts and support venture administration in distributing testing strategies with assets adequately.

The content of this chapter is published in International Journal of Applied Engineering Research ISSN 0973- 4562 Volume 9, Number 23 (2014) pp. 20879-20890. The reprint is appended toward end of this thesis.

Software engineering is the process to develop, design and maintenance of the software product with updated features of the earlier application development. These features termed as software development and information technology from various updated and outdated application events from other equality applications. Software development is one of the computer programming, documenting and testing involved in real time application development. Software development may include various features. In
recent research, development processes, reuse, reengineering maintenance, prototyping and other activities in software development application process. Programming improvement technique is edge work used to create structure, arrange, and control the procedure of creating data frameworks with practical skeleton engendering with applicable information. Included matter with proselyte the peculiarities in practical in qualities and shortcomings. Figure 5.1 shows technique for advancement programming application including all the utilitarian, programming and outline usage with testing feasibilities. Consider the above examination; we watch that product testing as one of the key elements for exploring the gimmicks of machine program that creates surprising results.

Figure 5.1 Software Development Process
The achievement of the product relies upon the expense and timetable, as well as on quality, and quality may rely on programming deformity forecast. Programming imperfection is a slip, defect, mix-up, and blame in machine program that delivers unforeseen results with relative programming improvement application process. Programming deformity expectation is the procedure of model that causes with viable modules in programming advancement process administration operations. Marian Jureczko (2013) to lessen programming absconds, progressively, application process administration with their delegate accomplishments of item discharge quality is a primary sacred occasion in programming forecast. Programming imperfection is condition in programming discharged item does not rely on programming prerequisites and end client desires. For available weak of the product deformity forecast continuously application advancement, information mining methods are acquainted with produce proficient and fantastic information administration operations in discharge programming item handling. Information mining is the methodology of examining information from alternate point of view information dissection and after that outlining the information with helpful data. Distinctive information digging methods are presented for illuminating programming deformity to handle continuously programming application advancement with relative information administration operations. Dulal Chandra Sahana (2013) According to the affiliation text mining in information, mining system occasions in programming application improvement characteristics they incorporate all the information measures that attains relations in semantic information representation of authentic information utilizing affiliation digging for concentrating the peculiarities of the examined information movement. We proposed affiliation mining, distinctive guidelines are sorted out to foresee programming deformity expectations with partner relations in diverse methods for transforming guideline set eras in semantic information representation of information dissection. It uncovers examples and patterns that are characterized in verifiable database process. Affiliation guideline mining is not just limiting the administrations with reliance dissection in the connection application improvement furthermore find co-happening examples of the traits in information bases I. Gondra, (2008). It takes after affiliation arrangement for running across fulfilled example grouping and different gimmicks of the obliged information base outline process. One pass calculation is one of the affiliated calculations for depicting these occasions focused around arrangement and acquainted order for higher grouping exactness with
progressive information occasion associations. One pass calculation portrays administration of recorded information from different information transactions; calculation takes after productive process on assessment and imperfection forecast for catching programming deformity forms in authentic information when we apply pattern assessment and other gimmick exercises in pertinent information movement. This methodology takes after affiliation tenet digging for computing least backing and least trust in every information thing present in chronicled information with successive property check in information examination of the delivered information movement. Gondra (2008) Using this methodology we diminish the product item cost and booking occasions of chronicled information, in illustrative occasions to other progressing occasions with business movement era. Programming quality is one of the key viewpoints in programming improvement process, because of this application, structure of programming items we propose to create prescient grouping calculation for expanding proficient gimmick areas on every information set representation of most punctual information things with continuous property representation in authentic information.

We utilize deformity information sort as prescient grouping for partitioning information sets into number of information things. With incorporating quality representation continuously, programming handling units in information consistence
with backing and certainty of the continuous information sets to expand programming item confirmation.

The basic idea of the association rule mining that originates from retrieving data items from historical data, for example customers buy three products p1, p2, and p3 and organized with probability of all the product information in finding efficient data extraction processes using support and confidence with other format results. In our technique the following procedure will occur and calculating the patterns of attributes in data bases T.R. Gopalakrishnan Nair (2012). Association rule mining explores high confidence associations among multiple variables as it may overcome some constraints introduced by other techniques. The success of the software prediction assessment with association rule mining in various fields motivates us to apply software defect data set.

Remaining part of Chapter proposes as follows: Section 5.2 explains all the literature review of the previous used techniques with feasibility access on advanced techniques. Section 5.3 explains Back ground work propagation of software product assurance with different values. Section 5.4 explains Research approach of predictive classification procedures. Section 5.5 explains experimental evaluation of each subordinate software product development.

5.2 RELATED WORK

To enhance the product quality, gainfulness programming designers with information mining, calculations for different programming applications with application programming client interface systems by a complex library or schema with lacking documentation G. Fung (2005). Programming deformity expectation is a device and after that it creates among testing exercises and programming improvement process applications. Defect indicators used to make as assets of the product measurements and other gimmick exercises in business occasion movement. Martin Shepperd et al. (2013) presents K- implies and neural information mining strategies on distinctive information sets and afterward illustrative those terms in reasonable module representation with investigating information into diverse groups with deficiency tolerance. In their study, they have displayed relative results performed on same information sets. Grouping is the procedure of displaying occasions that depicts programming models. In these
models, a preparation information set and other test information sets with proficient information extraction on chronicled information. The proposed expectation characterization calculation helps distinctive affiliation tenets with quantitative and qualitative information thing accomplishment in sensible information representation of the handling of properties presentation of the consecutive preparing of every information thing designated to other information characteristics Qinbao Song (2006). The proposed work concurs with retraining information from different information things introduced in the business occasions. To begin with, it recovers information from different information sets display in the concentrated information sets with handling occasion administration. It assesses all the information things from concentrated information sets with nature of programming application improvement. Affiliation forecast calculation does not accomplish representation of all the preparing units.

**Background Work**

Association rule mining is to find the association rules that satisfy data set report generation in minimum support and minimum confidence with data results in comparative data process. Historical datasets are reported with proceeding events in real time software application development. In this technique historical data can be divided into two ways with further report processing of all the commercial event software applications. It divides into two ways of evaluation and defect prediction using the services of software product evaluation with relative and successive coordination with semantic and other features are activated historical data representation. In this technical aspect, representation of the data computation there is a relative conceptual presentation of each and verified results in commercial progression. Data sets are loaded into memory process for computing additional requirements of association rule mining with conceptual and physical processes with equality sharing between each item set presents historical data Dulal Chandra Sahana (2013) derived Procedure of the one pass algorithm can be satisfied following consequences, upload data sets and then training some datasets into necessary action in to test data for accessing services from testing data then apply one pass algorithm on historical data sets. Compare data sets into one pass procedure in real time
processing operations. Association rules are applied in following terminology events with reserved data items in calculating minimum support and confidence for accessing services with relevant features of each data item.

![Figure 5.3 One pass framework for data set computation](image)

5.3 RESEARCH APPROACH

According to the utilization data sets from historical datasets, each data item calculated association of memorials and processing features of the extracted data sets. These procedures are evaluated into number of item sets with including each item set representation in proceeding on each data item. One pass algorithm achieves software product cost and scheduling of services in recommended data events. Consider the representation of software quality assurance we propose to develop Predictive Classification algorithm for increasing the services of all the features presented in data modulation. We propose to develop predictive classification algorithm to decrease cost of the software testing development. This algorithm consists 3 phases:

- Extraction phase
- Learning Phase
- Prediction Result Analysis Phase

**In Extraction phase the process will include following procedure**

Initialize Datasets, Transactions, Training set (TS), Testing set (TS1)
Extract data set from D = d0,d1,d3, ......................... dn
Perform transactions $T = t_0, t_1, t_2, \ldots \ldots \ldots, t_n$ on retrieving data sets 
Calculate each transaction $r$ with processing operations on training and testing extraction datasets.

**Learning Phase performs following procedures.**
Training data from each Data item $D = d_0, d_1, \ldots \ldots, d_n$
Test training data from Extracted data sets $D = d_0, d_1, \ldots \ldots, d_n$.
Learner = Build Classifier ($d'$, scheme. Algorithm)
Calculate Support($S$) and Confidence($C$) for each attribute in data item including transactions

**Prediction results**
$(\text{Predictor, bestAttrs}) = \text{Learning (historical Data)} \ d = \text{select best Attrs from new Data}$
Classify data with when transaction occurs Support as $P(d_0Ud_1)$ for each attribute in data item.
Classify data with when transaction occurs Confidence as $P(d_0/d_1)$ for each attribute in data item.
Result = Predict ($d$, predictor).

**Algorithm 1: Predictive Classification algorithm for software defect predictions.**

Calculation 1 takes after productive methodology for ascertaining least and most extreme information utility focused around affiliation principle mining operations which incorporates clear clarification with respect to information, investigation and information Visualization which incorporates preparing of information things with quantitative and qualitative information things for expanding limit religious in information representation in present procedure B. Dhanalaxmi (2012).

Information Source and Extraction: Data we utilize SEL (Software Engineering Laboratory) information sets which is sub situated of online information sets like NASA and other stockpiling and recovery of programming designing information things. Qinbao Song (2006) The SEL builds information to furnish ventures with respect to programming necessities with determined characters changes and blunders being developed of all the product applications. With the end goal of programming imperfection expectation which incorporates the methodology of concentrated information from distinctive tables of SEL information research facilities. The
deformity is exceptionally basic in concentrating information sets from distinctive gimmick forms.

Examination Approach: For this methodology we utilize acceptance technique on intersection information occasions with results in all the introduced information which prepares effective work spread S. Lessmann (2008). We utilize affiliation principle mining system to recover preparing information sets from information angles to introduce in the business work engendering occasion administration. For deformity affiliation procedure of concentrated information sets standard learning is straight advance while imperfection forecast the ensuing of a tenet must be absconding amendment exertion.

5.4 EXPERIMENTAL RESULTS

In this section we present experimental results for defect transactional data operations when performing the operations of the association which includes data processing unit as product releasing cost and product formative cost and scheduling the product with consequence event management operations. Ms. Puneet Jai Kaur (2013) Extracted defect association correction effort data set with different minimum support and minimum confidence threshold for associating rule mining based defect correction and effort predictions. For example, we describe to take five different data sets as training and as test data sets are used to defect software effort prediction.

Input: Data extraction with rule ranking RR, data sets with attributes, defects. Output: Predicted effort for correcting defect attributes.

```
Initialize the Simulation ------> O, Effect<------ O
For each data item e € attribute Do
  Defect <- e;
  Apply each rule on data item r € rules
  Defect <----- Consequence(r);
  Each data item attribute calculate the support and confidence
  Effect< ----- Effect U Defect;
  Simulation <----- Sim U D (attrib) / attri.
  Defect<---Effect (attrib). End if End for.
Release defect factor.
```

Figure 5.4 Effort prediction procedure

Perform operation with each data item then calculate the performance of accuracy and quality of software product in commercial event management system application.
progress. Our proposed work follows the following procedure events in query evaluation.

**DEFECT ASSOCIATION PREDICTION:** When mining defect association rules minimum support 10,20 , and 30 four minimum confidence values.

Defectf {DataV alueg}
DefectfNullg@\(32:5\%; 79:9\%\).
Defectf {Comput:g ^ DefectfIni:g}
DefectfEx:Interfaceg@\(34:3\%; 75:1\%\).
The first defect rule that defects the Data value occurred in the defect process. The second rule defines example interfacing the systematic data representation in calculating support and confidence.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Accuracy</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. Supp=20%</td>
<td>Mean=96.59%</td>
<td>FP= 13.17%</td>
</tr>
<tr>
<td>Min. Conf=30%</td>
<td>Min = 95.83%</td>
<td>FN=2.84%</td>
</tr>
</tbody>
</table>

The quality of the defect association depends upon the association rules which includes perfect process of explored with impact of min.supp and min.conf on each association rule progressed with commercial data aggregation on each data item presented in attributing representation.

**DEFECT EFFORT PREDICTION:** We use same configurations for extracting and mining defect isolation which includes defect rules for accessing services. Qinbao Song (2006) the distribution effort formation of all the taken five data sets gives minimum support And confidence effort with equal formalism with sufficient data processing units.

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td>Apriori</td>
<td>93.80</td>
</tr>
<tr>
<td>PART</td>
<td>69.52</td>
</tr>
<tr>
<td>C4.5</td>
<td>68.59</td>
</tr>
<tr>
<td>Naive Byes</td>
<td>67.95</td>
</tr>
</tbody>
</table>
We observe that each data set the average number of rules decreases then min.supp can be increased from 10 to 40 percent with equal sharing of data between each data item present in requested protocol hierarchy B. Dhanalaxmi (2012). As shown in the Table 5.2 achieves efficient comparison results like apriori association rule mining gives higher performance in comparison working procedure with equality sharing with other PART, C4.5, and other Naive Bayes methodologies.

The process of executing informative representation of each data item which includes process applications in real time processing units.

The reason association rule mining based prediction performs so much better than other methods in that it explores high confidence associations among multiple variables and discovers interesting rules, i.e., rules that are useful, strong, and significant. G. Fung (2005) Concluded experimental result show efficient data set
representation in increasing the quality of software in processing utilization event management operations.

5.5 CONCLUSION

Software quality and maintenance are main achieving terms in recent research approaches. Conventional methods are used in one pass algorithm for finding association rules with minimum support and minimum confidence. In this chapter we propose to develop predictive classification algorithm to decrease cost of the software testing development and cost estimation for software application process. With this proposed software application model format, we increase software quality assurance with all the properties in the data set representation in association rule mining. It further supports the conclusion that a sufficient number of rules are a precondition for the high prediction accuracy to obtained in the context of defect isolation effort prediction.
import java.io.*;
import java.util.*;
/**
* Set of utilities to support various Association Rule Mining (ARM)
* algorithms.
*/
public class AssocRuleMining {
    protected class RuleNode {
        /** Antecedent of AR. */
        protected short[] antecedent;
        /** Consequent of AR. */
        protected short[] consequent;
        /** The confidence value associate with the rule represented by this
         * node. */
        double confidenceForRule = 0.0;
        /** Link to next node */
        RuleNode next = null;
        /** Three argument constructor
         * @param ante the antecedent (LHS) of the AR.
         * @param cons the consequent (RHS) of the AR.
         * @param confValue the associated confidence value. */
        protected RuleNode(short[] ante, short[] cons, double confValue) {
            antecedent = ante;
            consequent = cons;
            confidenceForRule = confValue;
        }
    } // Data structures
    /** The reference to start of the rule list. */
    protected RuleNode startRulelist = null;
    /** 2-D aray to hold input data from data file. Note that within the data
     * array records are numbered from zero, thus record one has index 0 etc. */
    protected short[][] dataArray = null;
    /** 2-D array used to renumber columns for input data in terms of
     * frequency of single attributes (reordering will enhance performance
     * for some ARM algorithms). */
    protected int[][] conversionArray = null;
    /** 1-D array used to reconvert input data column numbers to their
     * original numbering where the input data has been ordered to enhance
     * computational efficiency. */
    protected short[] reconversionArray = null;
    // Constants
    /** Minimum support value */
    private static final double MIN_SUPPORT = 0.0;
    /** Maximum support value */
    private static final double MAX_SUPPORT = 100.0;
    /** Maximum confidence value */
    private static final double MIN_CONFIDENCE = 0.0;
    /** Maximum confidence value */
    private static final double MAX_CONFIDENCE = 100.0;
    // Command line arguments with default values and associated fields.
    /** Command line argument for data file name. */
    protected String fileName = null;
    /** Command line argument for number of columns. */
    protected int numCols = 0;
    /** Command line argument for number of rows. */
    protected int numRows = 0;
    /** Command line argument for % support (default = 20%). */
    protected double support = 20.0;
    /** Minimum support value in terms of number of rows. <P>Set when input
     * data is read and the number of records is known, */
    protected double minSupport = 0;
    /** Command line argument for % confidence (default = 80%). */
    protected double confidence = 80.0;
    /** The number of one itemsets (singletons). */
    protected int numOneItemSets = 0;
    // Flags
    /** Error flag used when checking command line arguments (default =
protected boolean errorFlag = true;
/** Input format OK flag (default = <TT>true</TT>). */
protected boolean inputFormatOkFlag = true;
/** Flag to indicate whether system has data or not. */
private boolean haveDataFlag = false;
/** Flag to indicate whether input data has been sorted or not. */
protected boolean isOrderedFlag = false;
/** Flag to indicate whether input data has been sorted and pruned or not. */
protected boolean isPrunedFlag = false;

// Other fields
/** The input stream. */
protected BufferedReader fileInput;
/** The file path */
protected File filePath = null;

/* ------ CONSTRUCTORS ------ */
/** Processes command line arguments */
public AssocRuleMining(String[] args)
{
    // Process command line arguments
    for(int index=0;index<args.length;index++) idArgument(args[index]);
    // If command line arguments read successfully (errorFlag set to "true")
    // check validity of arguments
    if (errorFlag) CheckInputArguments();
    else outputMenu();
}

/* ------ METHODS ------ */
/* ---------------------------------------------------------------- */
/*                                                                  */
/*                        COMMAND LINE ARGUMENTS                    */
/*                                                                  */
/* ---------------------------------------------------------------- */
/* IDENTIFY ARGUMENT */
/** Identifies nature of individual command line arguments:
-C = confidence, -F = file name, -S = support. */
protected void idArgument(String argument) {
    if (argument.charAt(0) == '-') {
        char flag = argument.charAt(1);
        argument = argument.substring(2,argument.length());
        switch (flag) {
            case 'C':
                confidence = Double.parseDouble(argument);
                break;
            case 'F':
                fileName = argument;
                break;
            case 'S':
                support = Double.parseDouble(argument);
                break;
            default:
                System.out.println("INPUT ERROR: Unrecognised command " +
                        "line argument " + flag + argument);
                errorFlag = false;
                break;
        }
    } else {
        System.out.println("INPUT ERROR: All command line arguments " +
                        "must commence with a '-' character (" +argument + ")");
        errorFlag = false;
    }
}

/* CHECK INPUT ARGUMENTS */
/** Invokes methods to check values associate with command line arguments */
protected void CheckInputArguments()
{
    // Check support and confidence input
    checkSupportAndConfidence();
}
// Check file name
checkFileName();

// Return
if (errorFlag) outputSettings();
else outputMenu();
}

 /**< CHECK SUPPORT AND CONFIDENCE */
/** Checks support and confidence input % values, if either is out of
 bounds then <TT>errorFlag</TT> set to <TT>false</TT>. */
protected void checkSupportAndConfidence()
{
  // Check Support
  if ((support < MIN_SUPPORT) || (support > MAX_SUPPORT))
  {
    System.out.println("INPUT ERROR: Support must be specified " + "as a percentage (" + MIN_SUPPORT + ", " + MAX_SUPPORT + ")");
    errorFlag = false;
  }

  // Check confidence
  if ((confidence < MIN_CONFIDENCE) || (confidence > MAX_CONFIDENCE))
  {
    System.out.println("INPUT ERROR: Confidence must be " + "specified as a percentage (" + MIN_CONFIDENCE + ", " + MAX_CONFIDENCE + ")");
    errorFlag = false;
  }
}

 /**< CHECK FILE NAME */
/** Checks if data file name provided, if not <TT>errorFlag</TT> set
to <TT>false</TT>. */
protected void checkFileName()
{
  if (fileName == null)
  {
    System.out.println("INPUT ERROR: Must specify file name (-F)");
    errorFlag = false;
  }
}

 /**< --------------------------------------------------------------- */
 /**< */
 /**< READ INPUT DATA FROM FILE */
 /**< */
 /**< --------------------------------------------------------------- */

 /**< INPUT DATA SET */
/** Commences process of getting input data (GUI version also exists). */
public void inputDataSet()
{
  // Read the file
  readFile();
  // Check ordering (only if input format is OK)
  if (inputFormatOkFlag)
  {
    if (checkOrdering())
    {
      System.out.println("Number of records = "+ numRows);
      countNumCols();
      System.out.println("Number of columns = " + numCols);
      minSupport = (numRows * support)/100.0;
      System.out.println("Min support = " + twoDecPlaces(minSupport) + " (records)"");
    }
    else
    {
      System.out.println("Error reading file: " + fileName + " at");
      closeFile();
      System.exit(1);
    }
  }
}
/* READ FILE */
/** Reads input data from file specified in command line argument <TT>fileName</TT>. Note that it is assumed that no empty records are included. Proceeds as follows:
<OL>
<LI>Gets number of rows (lines) in file, checking format of each line (space separated integers), if incorrectly formatted line found <TT>inputFormatOkFlag</TT> set to <TT>false</TT>.<LI>Dimensions input array.<LI>Reads data
</OL> */
protected void readFile()
{
try{
   // Dimension data structure
   inputFormatOkFlag=true;
   numRows = getNumberOfLines(fileName);
   if (inputFormatOkFlag)
       dataArray = new short[numRows][];
   // Read file
   System.out.println("Reading input file: " + fileName);
   readInputDataSet();
   }
else
   System.out.println("Error reading file: " + fileName + "a");
catch(IOException ioException)
   { System.out.println("Error reading File");
   closeFile();
   System.exit(1);
   }
}/* GET NUMBER OF LINES */
/** Gets number of lines/records in input file and checks format of each line.
* @param nameOfFile the filename of the file to be opened.
* @return the number of rows in the given file.
* */
protected int getNumberOfLines(String nameOfFile) throws IOException
{
int counter = 0;
// Open the file
if (filePath==null) openFileName(nameOfFile);
else openFilePath();
// Loop through file incrementing counter
// get first row.
String line = fileInput.readLine();
while (line != null)
{
   checkLine(counter+1,line);
   StringTokenizer dataLine = new StringTokenizer(line);
   int numberOfTokens = dataLine.countTokens();
   if (numberOfTokens == 0) break;
   counter++;
   line = fileInput.readLine();
}
// Close file and return
   closeFile();
   return(counter);
}/* CHECK LINE */
/** Check whether given line from input file is of appropriate format (space separated integers), if incorrectly formatted line found <TT>inputFormatOkFlag</TT> set to <TT>false</TT>.
* @param counter the line number in the input file.
*/
/**
 * Reads input data from file specified in command line argument. */
 public void readInputDataSet() throws IOException {
     readInputDataSet(fileName);
 }

/** Reads input data from given file. */
 protected void readInputDataSet(String fName) throws IOException {
     int rowIndex=0;
     // Open the file
     if (filePath==null) openFileName(fName);
     else openFilePath();
     // Get first row.
     String line = fileInput.readLine();
     // Process rest of file
     while (line != null) {
         // Process line
         if (!processInputLine(line,rowIndex)) break;
         // Increment first (row) index in 2-D data array
         rowIndex++;
         // get next line
         line = fileInput.readLine();
     }
     // Close file
     closeFile();
 }

/**
 * Reads input data segment from a given file and places content into to
 * the data array structure commencing at the given row index, continues until
 * the end index is reached.
 */
 protected void readInputDataSetSeg(String fName, int startRowIndex,
 int endRowIndex) throws IOException {
     int rowIndex=startRowIndex;
     // Open the file
     if (filePath==null) openFileName(fName);
     else openFilePath();
     // get first row.
     String line = fileInput.readLine();
     for (int index=startRowIndex;index<endRowIndex;index++) {
         // Process line
         processInputLine(line,index);
         // get next line
         line = fileInput.readLine();
     }
 }

/**
 * The current line from the input file. */
 protected void checkLine(int counter, String str) {
     for (int index=0;index <str.length();index++)
         if (!Character.isDigit(str.charAt(index)) &&
             !Character.isWhitespace(str.charAt(index)))
             System.err.println("FILE INPUT ERROR:
 charcater on line " + counter +
 " is not a digit or white space");
     inputFormatOkFlag = false;
     haveDataFlag = false;
     break;
 }

/* READ INPUT DATA SET */
/** Reads input data from file specified in command line argument. */
 public void readInputDataSet() throws IOException {
     readInputDataSet(fileName);
 }

/* READ INPUT DATA SET */
/** Reads input data from given file. */
 protected void readInputDataSet(String fName) throws IOException {
     int rowIndex=0;
     // Open the file
     if (filePath==null) openFileName(fName);
     else openFilePath();
     // Get first row.
     String line = fileInput.readLine();
     // Process rest of file
     while (line != null) {
         // Process line
         if (!processInputLine(line,rowIndex)) break;
         // Increment first (row) index in 2-D data array
         rowIndex++;
         // get next line
         line = fileInput.readLine();
     }
     // Close file
     closeFile();
 }

/* READ INPUT DATA SEGMENT */
/** Reads input data segment from a given file and places content into to
 * the data array structure commencing at the given row index, continues until
 * the end index is reached.
 */
 protected void readInputDataSetSeg(String fName, int startRowIndex,
 int endRowIndex) throws IOException {
     int rowIndex=startRowIndex;
     // Open the file
     if (filePath==null) openFileName(fName);
     else openFilePath();
     // get first row.
     String line = fileInput.readLine();
     for (int index=startRowIndex;index<endRowIndex;index++) {
         // Process line
         processInputLine(line,index);
         // get next line
         line = fileInput.readLine();
     }
 }

/* READ INPUT DATA SEGMENT */
/** Reads input data segment from a given file and places content into to
 * the data array structure commencing at the given row index, continues until
 * the end index is reached.
 */
 protected void readInputDataSetSeg(String fName, int startRowIndex,
 int endRowIndex) throws IOException {
     int rowIndex=startRowIndex;
     // Open the file
     if (filePath==null) openFileName(fName);
     else openFilePath();
     // get first row.
     String line = fileInput.readLine();
     for (int index=startRowIndex;index<endRowIndex;index++) {
         // Process line
         processInputLine(line,index);
         // get next line
         line = fileInput.readLine();
     }
 }

/* READ INPUT DATA SEGMENT */
/** Reads input data segment from a given file and places content into to
 * the data array structure commencing at the given row index, continues until
 * the end index is reached.
 */
 protected void readInputDataSetSeg(String fName, int startRowIndex,
 int endRowIndex) throws IOException {   
     int rowIndex=startRowIndex;
     // Open the file
     if (filePath==null) openFileName(fName);
     else openFilePath();
     // get first row.
     String line = fileInput.readLine();
     for (int index=startRowIndex;index<endRowIndex;index++) {
         // Process line
         processInputLine(line,index);
         // get next line
         line = fileInput.readLine();
     }
 }

/* READ INPUT DATA SEGMENT */
/** Reads input data segment from a given file and places content into to
 * the data array structure commencing at the given row index, continues until
 * the end index is reached.
 */
 protected void readInputDataSetSeg(String fName, int startRowIndex,
 int endRowIndex) throws IOException {   
     int rowIndex=startRowIndex;
     // Open the file
     if (filePath==null) openFileName(fName);
     else openFilePath();
     // get first row.
     String line = fileInput.readLine();
     for (int index=startRowIndex;index<endRowIndex;index++) {
         // Process line
         processInputLine(line,index);
         // get next line
         line = fileInput.readLine();
     }
 }

/* READ INPUT DATA SEGMENT */
/** Reads input data segment from a given file and places content into to
 * the data array structure commencing at the given row index, continues until
 * the end index is reached.
 */
 protected void readInputDataSetSeg(String fName, int startRowIndex,
 int endRowIndex) throws IOException {   
     int rowIndex=startRowIndex;
     // Open the file
     if (filePath==null) openFileName(fName);
     else openFilePath();
     // get first row.
     String line = fileInput.readLine();
     for (int index=startRowIndex;index<endRowIndex;index++) {
         // Process line
         processInputLine(line,index);
         // get next line
         line = fileInput.readLine();
     }
 }

/* READ INPUT DATA SEGMENT */
/** Reads input data segment from a given file and places content into to
 * the data array structure commencing at the given row index, continues until
 * the end index is reached.
 */
 protected void readInputDataSetSeg(String fName, int startRowIndex,
 int endRowIndex) throws IOException {   
     int rowIndex=startRowIndex;
     // Open the file
     if (filePath==null) openFileName(fName);
     else openFilePath();
     // get first row.
     String line = fileInput.readLine();
     for (int index=startRowIndex;index<endRowIndex;index++) {
         // Process line
         processInputLine(line,index);
         // get next line
         line = fileInput.readLine();
     }
 }
/* PROCESS INPUT LINE */

/** Processes a line from the input file and places it in the <TT>dataArray</TT> structure.
 * @param line the line to be processed from the input file
 * @param rowIndex the index to the current location in the <TT>dataArray</TT> structure.
 * @return true if successfull, false if empty record. */
private boolean processInputLine(String line, int rowIndex) {
    // If no line return false
    if (line==null) return(false);
    // Tokenize line
    StringTokenizer dataLine = new StringTokenizer(line);
    int numberOfTokens = dataLine.countTokens();
    // Empty line or end of file found, return false
    if (numberOfTokens == 0) return(false);
    // Convert input string to a sequence of short integers
    short[] code = binConversion(dataLine,numberOfTokens);
    // Dimension row in 2-D dataArray
    int codeLength = code.length;
    dataArray[rowIndex] = new short[codeLength];
    // Assign to elements in row
    for (int colIndex=0;colIndex<codeLength;colIndex++)
        dataArray[rowIndex][colIndex] = code[colIndex];
    // Return
    return(true);
}

/* CHECK DATASET ORDERING */
/** Checks that data set is ordered correctly.
 * @return true if appropriate ordering, false otherwise. */
protected boolean checkOrdering() {
    boolean result = true;
    // Loop through input data
    for(int index=0;index<dataArray.length;index++) {
        if (!checkLineOrdering(index+1,dataArray[index])) {
            haveDataFlag = false;
            result=false;
        }
    }
    // Return
    return(result);
}

/* CHECK LINE ORDERING */
/** Checks whether a given line in the input data is in numeric sequence.
 * @param lineNum the line number.
 * @param itemSet the item set represented by the line
 * @return true if OK and false otherwise. */
protected boolean checkLineOrdering(int lineNum, short[] itemSet) {
    for (int index=0;index<itemSet.length-1;index++) {
        if (itemSet[index] >= itemSet[index+1]) {
            System.err.println("FILE FORMAT ERROR:
" + "Attribute data in line "+ lineNum + " not in numeric order");
            return(false);
        }
    }
    // Default return
    return(true);
}
/* COUNT NUMBER OF COLUMNS */
/** Counts number of columns represented by input data. */
protected void countNumCols()
{
    int maxAttribute=0;
    // Loop through data array
    for(int index=0;index<dataArray.length;index++)
    {
        int lastIndex = dataArray[index].length-1;
        if (dataArray[index][lastIndex] > maxAttribute)
            maxAttribute = dataArray[index][lastIndex];
    }
    numCols = maxAttribute;
    numOneItemSets = numCols;   // default value only
}

/* OPEN FILE NAME */
/** Opens input file using fileName (instance field).
 @param nameOfFile the filename of the file to be opened. */
protected void openFileName(String nameOfFile)
{
    try
    {
        // Open file
        FileReader file = new FileReader(nameOfFile);
        fileInput = new BufferedReader(file);
    }
    catch(IOException ioException)
    {
        System.err.println("Error Opening File");
        System.exit(1);
    }
}

/* OPEN FILE PATH */
/** Opens file using filePath (instance field). */
protected void openFilePath() {
    try
    {
        // Open file
        FileReader file = new FileReader(filePath);
        fileInput = new BufferedReader(file);
    }
    catch(IOException ioException)
    {
        System.err.println("Error Opening File");
        System.exit(1);
    }
}

/* CLOSE FILE */
/** Close file fileName (instance field). */
protected void closeFile() {
    if (fileInput != null) {
        try {
            fileInput.close();
        }
        catch (IOException ioException) {
            System.err.println("Error Closing File");
            System.exit(1);
        }
    }
}

/* BINARY CONVERSION. */
/** Produce an item set (array of elements) from input line.
 @param dataLine row from the input data file
 @param numberOfTokens number of items in row
 @return 1-D array of short integers representing attributes in input row */
protected short[] binConversion(StringTokenizer dataLine,
int numberOfTokens)
{
    short number;
    short[] newItemSet = null;

    // Load array
    for (int tokenCounter=0; tokenCounter < numberOfTokens; tokenCounter++)
    {
        number = new Short(dataLine.nextToken()).shortValue();
        newItemSet = realloc1(newItemSet, number);
    }

    // Return itemSet
    return(newItemSet);
}

/* ---------------------------------------------------------------- */
/*                                                                  */
/* REORDER DATA SET ACCORDING TO ATTRIBUTE FREQUENCY */
/*                                                                  */
/* ---------------------------------------------------------------- */

/* REORDER INPUT DATA: */
/** Reorders input data according to frequency of single attributes. <P> Example, given the data set:
<pre>
1 2 5
1 2 3
2 4 5
1 2 5
2 3 5
</pre>
This would produce a countArray (ignore index 0):
<pre>
+---+---+---+---+---+---+
|   | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+
|   | 2 | 1 | 4 | 3 | 5 |
+---+---+---+---+---+---+
</pre>
Which sorts to:
<pre>
+---+---+---+---+---+---+
|   | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+
|   | 5 | 4 | 3 | 2 | 1 |
+---+---+---+---+---+---+
</pre>
Giving rise to the conversion Array of the form (no index 0):
<pre>
+---+---+---+---+---+---+
|   | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+
|   | 5 | 4 | 3 | 2 | 1 |
+---+---+---+---+---+---+
</pre>
Note that the second row here are the counts which no longer play a role in the conversion exercise. Thus to the new column number for column 1 is column 3 (i.e., the first vale at index 1). The reconversion array of the form:
<pre>
+---+---+---+---+---+---+
|   | 2 | 5 | 1 | 3 | 4 |
+---+---+---+---+---+---+
</pre>
*/

public void idInputDataOrdering()
{
    // Count singles and store in countArray;
    int[][] countArray = countSingles();

    // Bubble sort count array on support value (second index)
    orderCountArray(countArray);
}
// Define conversion and reconversion arrays
defConvertArrays(countArray);

// Set sorted flag
isOrderedFlag = true;
}

/* COUNT SINGLES */

/** Counts number of occurrences of each single attribute in the
input data.
@return 2-D array where first row represents column numbers
and second row represents support counts. */
protected int[][] countSingles()
{
  // Dimension and initialize count array
  int[][] countArray = new int[numCols+1][2];
  for (int index=0;index<countArray.length;index++)
  {
    countArray[index][0] = index;
    countArray[index][1] = 0;
  }

  // Step through input data array counting singles and incrementing
  // appropriate element in the count array
  for(int rowIndex=0;rowIndex<dataArray.length;rowIndex++)
  {
    if (dataArray[rowIndex] != null)
    {
      for (int colIndex=0;colIndex<dataArray[rowIndex].length;colIndex++)
      {
        countArray[dataArray[rowIndex][colIndex]][1]++;  
      }
    }
  }

  // Return
  return(countArray);
}

/* ORDER COUNT ARRAY */

/** Bubble sorts count array produced by <TT>countSingles</TT> method
so that array is ordered according to frequency of single items.
@param countArray The 2-D array returned by the <TT>countSingles</TT>
method. */
private void orderCountArray(int[][] countArray)
{
  int attribute, quantity;
  boolean isOrdered;
  int index;
  do {
    isOrdered = true;
    index = 1;
    while (index < (countArray.length-1))
    {
      if (countArray[index][1] >= countArray[index+1][1]) index++;
      else {
        isOrdered=false;
        // Swap
        attribute = countArray[index][0];
        quantity = countArray[index][1];
        countArray[index][0] = countArray[index+1][0];
        countArray[index][1] = countArray[index+1][1];
        countArray[index+1][0] = attribute;
        countArray[index+1][1] = quantity;
        // Increment index
        index++;
      }
    }
  } while (isOrdered==false);
}

/* ORDER FIRST N ELEMENTS IN COUNT ARRAY */
/** Bubble sorts first N elements in count array produced by <TT>countSingles</TT> method so that array is ordered according to frequency of single items. <P>Used when ordering classification input data.
*param countArray The 2-D array returned by the <TT>countSingles</TT> method.
*param endIndex the index of the Nth element. */
protected void orderFirstNofCountArray(int[][] countArray, int endIndex)
{
  int attribute, quantity;
  boolean isOrdered;
  int index;
  do {
    isOrdered = true;
    index = 1;
    while (index < endIndex)
    {
      if (countArray[index][1] >= countArray[index+1][1])
        index++;
      else
      {
        isOrdered=false;
        // Swap
        attribute = countArray[index][0];
        quantity = countArray[index][1];
        countArray[index][0] = countArray[index+1][0];
        countArray[index][1] = countArray[index+1][1];
        countArray[index+1][0] = attribute;
        countArray[index+1][1] = quantity;
        // Increment index
        index++;
      }
    }
  } while (isOrdered==false);
}

/* DEFINE CONVERSION ARRAYS: */

/** Defines conversion and reconversion arrays.
*param countArray The 2-D array sorted by the <TT>orderCcountArray</TT> method. */
protected void defConvertArrays(int[][] countArray)
{
  // Dimension arrays
  conversionArray   = new int[numCols+1][2];
  reconversionArray = new short[numCols+1];
  // Assign values
  for(int index=1;index<countArray.length;index++)
  {
    conversionArray[countArray[index][0]][0] = index;
    conversionArray[countArray[index][0]][1] = countArray[index][1];
    reconversionArray[index] = (short) countArray[index][0];
  }
  // Diagnostic ouput if desired
  //outputConversionArrays();
}

/* RECAST INPUT DATA. */

/** Recasts the contents of the data array so that each record is ordered according to conversion array.
*param dataArray The 2-D array to be recasted.
*/
protected void recastInputData(int[][] dataArray)
{
  // Diagnostic output if desired
  //outputRecastInputData();
}

/* RULE OUT INDIVIDUALS. */

/** Rules out individuals from data array so that individual's record is given.
*param numTests The number of tests performed.
*param individual The test to rule out.
*/
protected void ruleOutIndividuals(int numTests, int individual)
{
  // Diagnostic output if desired
  //outputRuleOutIndividuals();
}
public void recastInputData()
{
    short[] itemSet;
    int attribute;

    // Step through data array using loop construct
    for(int rowIndex=0;rowIndex<dataArray.length;rowIndex++)
    {
        itemSet = new short[dataArray[rowIndex].length];
        // For each element in the itemSet replace with attribute number
        // from conversion array
        for(int colIndex=0;colIndex<dataArray[rowIndex].length;colIndex++)
        {
            attribute = dataArray[rowIndex][colIndex];
            itemSet[colIndex] = (short) conversionArray[attribute][0];
        }
        // Sort itemSet and return to data array
        sortItemSet(itemSet);
        dataArray[rowIndex] = itemSet;
    }
}

/* RECAST INPUT DATA AND REMOVE UNSUPPORTED SINGLE ATTRIBUTES. */

/** Recasts the contents of the data array so that each record is
    ordered according to ColumnCounts array and excludes non-supported
    elements. Proceed as follows:
    1) For each record in the data array. Create an empty new itemSet array.
    2) Place into this array any column numbers in record that are
       supported at the index contained in the conversion array.
    3) Assign new itemSet back into to data array */
public void recastInputDataAndPruneUnsupportedAtts()
{
    short[] itemSet;
    int attribute;

    // Step through data array using loop construct
    for(int rowIndex=0;rowIndex<dataArray.length;rowIndex++)
    {
        // Check for empty row
        if (dataArray[rowIndex]!= null)
        {
            itemSet = null;
            // For each element in the current record find if supported with
            // reference to the conversion array. If so add to "itemSet".
            for(int colIndex=0;colIndex<dataArray[rowIndex].length;colIndex++)
            {
                attribute = dataArray[rowIndex][colIndex];
                // Check support
                if (conversionArray[attribute][1] >= minSupport)
                {
                    itemSet = reallocInsert(itemSet,
                                (short) conversionArray[attribute][0]);
                }
            }
            // Return new item set to data array
            dataArray[rowIndex] = itemSet;
        }
    }

    // Set isPrunedFlag (used with GUI interface)
    isPrunedFlag=true;
    // Reset number of one item sets field
    numOneItemSets = getNumSupOneItemSets();
}

/* GET NUM OF SUPPORTED ONE ITEM SETS */
/** Gets number of supported single item sets (note this is not necessarily
the same as the number of columns/attributes in the input set).
@return Number of supported 1-item sets */
protected int getNumSupOneItemSets()
{
  int counter = 0;

  // Step through conversion array incrementing counter for each
  // supported element found
  for (int index=1; index < conversionArray.length; index++)
  {
    if (conversionArray[index][1] >= minSupport) counter++;
  }

  // Return
  return(counter);
}

/* RESIZE INPUT DATA */
/** Recasts the input data sets so that only N percent is used.
 * @param percentage the percentage of the current input data that is to form
 * the new input data set (number between 0 and 100). */
public void resizeInputData(double percentage)
{
  // Redefine number of rows
  numRows = (int) ((double) numRows*(percentage/100.0));
  System.out.println("Recast input data, new num rows = " + numRows);
  // Dimension and populate training set.
  short[][] trainingSet = new short[numRows][];
  for (int index=0; index < numRows; index++)
  {
    trainingSet[index] = dataArray[index];
  }
  // Assign training set label to input data set label.
  dataArray = trainingSet;

  // Determine new minimum support threshold value
  minSupport = (numRows * support)/100.0;
}

/** Reconverts given item set according to contents of reconversion array.
 * @param itemSet The given itemset.
 * @return The reconverted itemset. */
protected short[] reconvertItemSet(short[] itemSet)
{
  // If no conversion return original item set
  if (reconversionArray==null) return(itemSet);
  // If item set null return null
  if (itemSet==null) return(null);

  // Define new item set
  short[] newItemSet = new short[itemSet.length];

  // Copy
  for(int index=0; index < newItemSet.length; index++)
  {
    newItemSet[index] = reconversionArray[itemSet[index]];
  }

  // Return
  return(newItemSet);
}

/** Reconvert single item if appropriate.
 * @param item the given item (attribute).
 * @return the reconverted item. */
protected short reconvertItem(short item)
{
  // If no conversion return original item
  if (reconversionArray==null) return(item);
  // Otherwise return reconvert item
  return(reconversionArray[item]);
}
Methods for inserting rules into a linked list of rules ordered according to confidence (most confident first). Each rule described in terms of 3 fields: 1) Antecedent (an item set), 2) a consequent (an item set), 3) a confidence value (double). The support field is not used. */

/* INSERT (ASSOCIATION/CLASSIFICATION) RULE INTO RULE LINKED LIST (ORDERED ACCORDING CONFIDENCE). */

** Inserts an (association/classification) rule into the linked list of rules pointed at by <TT>startRulelist</TT>. The list is ordered so that rules with highest confidence are listed first. If two rules have the same confidence the new rule will be placed after the existing rule. Thus, if using an Apriori approach to generating rules, more general rules will appear first in the list with more specific rules (i.e., rules with a larger antecedent) appearing later as the more general rules will be generated first.

@param antecedent the antecedent (LHS) of the rule.
@param consequent the consequent (RHS) of the rule.
@param confidenceForRule the associated confidence value. */

protected void insertRuleIntoRuleList(short[] antecedent, short[] consequent, double confidenceForRule) {
    // Create new node
    RuleNode newNode = new RuleNode(antecedent, consequent, confidenceForRule);
    // Empty list situation
    if (startRulelist == null) {
        startRulelist = newNode;
        return;
    }
    // Add new node to start
    if (confidenceForRule > startRulelist.confidenceForRule) {
        newNode.next = startRulelist;
        startRulelist = newNode;
        return;
    }
    // Add new node to middle
    RuleNode markerNode = startRulelist;
    RuleNode linkRuleNode = startRulelist.next;
    while (linkRuleNode != null) {
        if (confidenceForRule > linkRuleNode.confidenceForRule) {
            markerNode.next = newNode;
            newNode.next = linkRuleNode;
            return;
        }
        markerNode = linkRuleNode;
        linkRuleNode = linkRuleNode.next;
    }
    // Add new node to end
    markerNode.next = newNode;
}

/* ----------------------------------------------- */
/*                                                 */
/*        ITEM SET INSERT AND ADD METHODS          */
/*                                                 */
/* ----------------------------------------------- */

/* REALLOC INSERT */

/** Resizes given item set so that its length is increased by one and new element inserted.
 * @param oldItemSet the original item set
 * @param newElement the new element/attribute to be inserted
 * @return the combined item set */

protected short[] reallocInsert(short[] oldItemSet, short newElement) {
    // No old item set
    if (oldItemSet == null)
short[] newItemSet = {newElement};
return(newItemSet);
}

// Otherwise create new item set with length one greater than old
// item set
int oldItemSetLength = oldItemSet.length;
short[] newItemSet = new short[oldItemSetLength+1];
// Loop
int index;
for (index=0;index < oldItemSetLength;index++)
{
if (newElement < oldItemSet[index])
{
newItemSet[index] = newElement;
// Add rest
for(int index2 = index+1;index2<newItemSet.length;index2++)
newItemSet[index2] = oldItemSet[index2-1];
return(newItemSet);
}
else newItemSet[index] = oldItemSet[index];
}
// Add to end
newItemSet[newItemSet.length-1] = newElement;
// Return new item set
return(newItemSet);

/* REALLOC 1 */
/** Resizes given item set so that its length is increased by one
and appends new element (identical to append method)
@param oldItemSet the original item set
@param newElement the new element/attribute to be appended
@return the combined item set */
protected short[] realloc1(short[] oldItemSet, short newElement)
{
// No old item set
if (oldItemSet == null) {
short[] newItemSet = {newElement};
return(newItemSet);
}

// Otherwise create new item set with length one greater than old
// item set
int oldItemSetLength = oldItemSet.length;
short[] newItemSet = new short[oldItemSetLength+1];

// Loop
int index;
for (index=0;index < oldItemSetLength;index++)
newItemSet[index] = oldItemSet[index];
newItemSet[index] = newElement;
// Return new item set
return(newItemSet);

/* REALLOC 2 */
/** Resizes given array so that its length is increased by one element
and new element added to front
@param oldItemSet the original item set
@param newElement the new element/attribute to be appended
@return the combined item set */
protected short[] realloc2(short[] oldItemSet, short newElement)
{
// No old array
if (oldItemSet == null) {
short[] newItemSet = {newElement};
return(newItemSet);
}

// Otherwise create new array with length one greater than old
int oldItemSetLength = oldItemSet.length;
short[] newItemSet = new short[oldItemSetLength+1];
// Loop
newItemSet[0] = newElement;
for (int index=0; index < oldItemSetLength; index++)
    newItemSet[index+1] = oldItemSet[index];
// Return new array
return(newItemSet);
}

/* --------------------------------------------- */
/*                                               */
/*            ITEM SET DELETE METHODS            */
/*                                               */
/* --------------------------------------------- */

/* REMOVE ELEMENT N */
/** Removes the nth element/attribute from the given item set. 
@param oldItemSet the given item set.
@param n the index of the element to be removed (first index is 0).
@return Revised item set with nth element removed. */
protected short[] removeElementN(short[] oldItemSet, int n) 
{
    if (oldItemSet.length <= n)
        return(oldItemSet);
    else
    {
        short[] newItemSet = new short[oldItemSet.length-1];
        for (int index=0; index<n; index++)
            newItemSet[index] = oldItemSet[index];
        for (int index=n+1; index<oldItemSet.length; index++)
            newItemSet[index-1] = oldItemSet[index];
        return(newItemSet);
    }
}

/* ---------------------------------------------------------------- */
/*                                                                  */
/*              METHODS TO RETURN SUBSETS OF ITEMSETS               */
/*                                                                  */
/* ---------------------------------------------------------------- */

/* COMPLEMENT */
/** Returns complement of first itemset with respect to second itemset. 
@param itemSet1 the first given item set.
@param itemSet2 the second given item set.
@return complement if <TT>itemSet1</TT> in <TT>itemSet2</TT>. */
protected short[] complement(short[] itemSet1, short[] itemSet2) 
{
    int lengthOfComp = itemSet2.length-itemSet1.length;
    // Return null if no complement
    if (lengthOfComp<1) return(null);
    // Otherwise define combination array and determine complement
    short[] complement = new short[lengthOfComp];
    int complementIndex = 0;
    for(int index=0; index<itemSet2.length; index++)
    {  
        // Add to combination if not in first itemset
        if (notMemberOf(itemSet2[index], itemSet1))
        {
            complement[complementIndex] = itemSet2[index];
            complementIndex++;
        }  
    }  
    // Return
    return(complement);
}

/* --------------------------------------- */
/*                                         */
/*             SORT ITEM SET               */
/*                                         */
/* --------------------------------------- */
/* SORT ITEM SET: Given an unordered itemSet, sort the set */
/** Sorts an unordered item set. */
protected void sortItemSet(short[] itemSet)
{
    short temp;
    boolean isOrdered;
    int index;
    do
    {
        isOrdered = true;
        index     = 0;
        while (index < (itemSet.length-1))
        {
            if (itemSet[index] <= itemSet[index+1])
                index++;
            else
            {
                isOrdered=false;
                // Swap
                temp = itemSet[index];
                itemSet[index] = itemSet[index+1];
                itemSet[index+1] = temp;
                // Increment index
                index++;
            }
        }
    } while (isOrdered==false);
}

/* NOT MEMBER OF */
/** Checks whether a particular element/attribute identified by a column number is not a member of the given item set. */
protected boolean notMemberOf(short number, short[] itemSet)
{
    // Loop through itemSet
    for(int index=0;index<itemSet.length;index++)
    {
        if (number < itemSet[index]) return(true);
        if (number == itemSet[index]) return(false);
    }
    // Got to the end of itemSet and found nothing, return true
    return(true);
}

/* COMBINATIONS */
/** Invokes <TT>combinations</TT> method to calculate all possible combinations of a given item set. */
protected array of arrays combinations (short[] itemSet)
{
    // Loop through itemSet
    for(int index=0;index<itemSet.length;index++)
    {
        if (number < itemSet[index]) return(true);
        if (number == itemSet[index]) return(false);
    }
    // Got to the end of itemSet and found nothing, return true
    return(true);
}
protected short[][] combinations(short[] inputSet)
{
    if (inputSet == null) return(null);
    else
    {
        short[][] outputSet = new short[getCombinations(inputSet)][];
        combinations(inputSet,0,null,outputSet,0);
        return(outputSet);
    }
}
/** Recursively calculates all possible combinations of a given item set.
@param inputSet the given item set.
@param inputIndex the index within the input set marking current element under consideration (0 at start).
@param sofar the part of a combination determined sofar during the recursion (null at start).
@param outputSet the combinations collected so far, will hold all combinations when recursion ends.
@return revised output index. */
private int combinations(short[] inputSet, int inputIndex,
short[] sofar, short[][] outputSet, int outputIndex)
{
    short[] tempSet;
    int index=inputIndex;
    // Loop through input array
    while(index < inputSet.length)
    {
        tempSet = realloc1(sofar,inputSet[index]);
        outputSet[outputIndex] = tempSet;
        outputIndex = combinations(inputSet,index+1,
copyItemSet(tempSet),outputSet,outputIndex+1);
        index++;
    }
    // Return
    return(outputIndex);
}
/* GET COMBINATIONS */
/** Gets the number of possible combinations of a given item set.
@param set the given item set.
@return number of possible combinations. */
private int getCombinations(short[] set)
{
    int counter=0, numComb;
    numComb = (int) Math.pow(2.0,set.length)-1;
    // Return
    return(numComb);
}
/* ---------------------------------------------------------------- */
/*                                                                  */
/*                            MISCELANEOUS                          */
/*                                                                  */
/* ---------------------------------------------------------------- */
/* COPY ITEM SET */
/** Makes a copy of a given itemSet.
@param itemSet the given item set.
@return copy of given item set. */
protected short[] copyItemSet(short[] itemSet)
{
    // Check whether there is a itemSet to copy
    if (itemSet == null) return(null);
    // Do copy and return
    short[] newitemSet = new short[itemSet.length];
    for(int index=0;index<itemSet.length;index++)
    {
        newitemSet[index] = itemSet[index];
    }
    // Return
    return(newitemSet);
}
/* OUTPUT METHODS */

/* OUTPUT DATA TABLE */
/** Outputs stored input data set; initially read from input data file, but may be reordered or pruned if desired by a particular application. */
public void outputDataArray()
{
    if (isPrunedFlag)
        System.out.println("DATA SET (Ordered and Pruned)\n" +  "-----------------------------");
    else {
        if (isOrderedFlag)
            System.out.println("DATA SET (Ordered)\n" +  "------------------");
        else System.out.println("DATA SET\n" +  "--------");
    }
    // Loop through data array
    for(int index=0;index<dataArray.length;index++)
    {
        outputItemSet(dataArray[index]);
        System.out.println();
    }
}

/** Outputs the given array of array of short integers. *<P> Used for diagnostic purposes. */
protected void outputDataArray(short[][] dataSet)
{
    if (dataSet==null)
    {
        System.out.println("null");
        return;
    }
    // Loop through data array
    for(int index=0;index<dataSet.length;index++)
    {
        outputItemSet(dataSet[index]);
        System.out.println();
    }
}

/* OUTPUT ITEMSET */
/** Outputs a given item set. */
protected void outputItemSet(short[] itemSet)
{
    // Check for empty set
    if (itemSet == null) System.out.print(" null ");
    // Process
    else {
        // Reconvert where input dataset has been reordered and possible pruned.
        short[] tempItemSet = reconvertItemSet(itemSet);
        // Loop through item set elements
        int counter = 0;
        for (int index=0;index<tempItemSet.length;index++)
        {
            if (counter == 0)
            {
                counter++;
                System.out.print(" {");
            }
            else System.out.print(" ");
                System.out.print(tempItemSet[index]);
        }
        System.out.print("} ");
    }
}
public void outputDataArraySize()
{
    int numRecords = 0;
    int numElements = 0;
    // Loop through data array
    for (int index = 0; index < dataArray.length; index++)
    {
        if (dataArray[index] != null)
        {
            numRecords++;
            numElements = numElements + dataArray[index].length;
        }
    }
    // Output
    System.out.println("Number of records = " + numRecords);
    System.out.println("Number of elements = " + numElements);
    double density = (double) numElements / (numCols * numRecords);
    System.out.println("Data set density = " + twoDecPlaces(density) + "%");
}

public void outputConversionArrays()
{
    // Conversion array
    System.out.println("Conversion Array = ");
    for (int index = 1; index < conversionArray.length; index++)
    {
        System.out.println("(" + index + ") " + conversionArray[index][0] + " = " + conversionArray[index][1]);
    }
    // Reconversion array
    System.out.println("Reconversion Array = ");
    for (int index = 1; index < reconversionArray.length; index++)
    {
        System.out.println("(" + index + ") " + reconversionArray[index]);
    }
}

protected void outputMenu()
{
    System.out.println();
    System.out.println("-F  = File name");
    System.out.println("-S  = Support (default 20%)");
    System.out.println();
    // Exit
    System.exit(1);
}

protected void outputSettings()
{
    System.out.println("SETTINGS
--------");
    System.out.println("File name                = " + fileName);
    System.out.println("Support (default 20%)    = " + support);
}
/* OUTPUT SETTINGS */
/** Outputs instance field values. */
protected void outputSettings2()
{
    System.out.println("SETTINGS\n--------");
    System.out.println("Number of records = " + numRows);
    System.out.println("Number of columns = " + numCols);
    System.out.println("Support (default 20%) = " + support);
    System.out.println("Confidence (default 80%) = " + confidence);
    System.out.println("Min support= " + minSupport + " (records)\n          Num one itemsets = " + numOneItemSets);
}

/* -------------------------------------- */
/* OUTPUT SUPPORT AND CONFIDENCE SETTINGS */
/* -------------------------------------- */
/** Outputs current support and confidence settings. */
public void outputSuppAndConf()
{
    System.out.println("Support = " + twoDecPlaces(support) + ", Confidence = " + twoDecPlaces(confidence));
}

/* ------------------------ */
/* OUTPUT RULE LINKED LISTS */
/* ------------------------ */
/** Outputs contents of rule linked list (if any) assuming that the list represents a set of ARs. */
public void outputRules()
{
    outputRules(startRuleList);
}

/** Outputs given rule list.
@param ruleList the given rule list. */
public void outputRules(RuleNode ruleList)
{
    // Check for empty rule list
    if (ruleList == null) System.out.println("No rules generated!");
    // Loop through rule list
    int number = 1;
    RuleNode linkRuleNode = ruleList;
    while (linkRuleNode != null)
    {
        System.out.print("(\n" + number + ") ");
        outputRule(linkRuleNode);
        System.out.println("\n(\n" + twoDecPlaces(linkRuleNode.confidenceForRule) + "%\n          number++;
        linkRuleNode = linkRuleNode.next;
    }
}

/** Outputs a rule assuming that the rule represents an ARs.
@param rule the rule to be output. */
private void outputRule(RuleNode rule)
{
    outputItemSet(rule.antecedent);
    System.out.print(" -> ");
    outputItemSet(rule.consequent);
}

/* OUTPUT RULE LINKED LIST WITH DEFAULT */
/** Outputs contents of rule linked list (if any), with reconversion, such that last rule is the default rule. */
public void outputRulesWithDefault()
{
    int number = 1;
    RuleNode linkRuleNode = startRuleList;
    while (linkRuleNode != null)
    {
        System.out.print("(\n" + number + ") ");
        outputRule(linkRuleNode);
        System.out.println("\n(\n" + twoDecPlaces(linkRuleNode.confidenceForRule) + "%\n          number++;
        linkRuleNode = linkRuleNode.next;
    }
}
// Output rule number
System.out.print("(" + number + ") ");
// Output antecedent
if (linkRuleNode.next==null) System.out.print("Default -> ");
else {
    outputItemSet(linkRuleNode.antecedent);
    System.out.print(" -> ");
}
// Output consequent
outputItemSet(linkRuleNode.consequent);
System.out.print(" "+
twoDecPlaces(linkRuleNode.confidenceForRule) + "%");
// Increment parameters
number++;
linkRuleNode = linkRuleNode.next;
}

/* --------------------------------*/
/*                                   */
/*        DIAGNOSTIC OUTPUT          */
/*                                   */
/* --------------------------------*/

/* OUTPUT DURATION */
/** Outputs difference between two given times. */
@param time1 the first time.
@param time2 the second time.
@return duration. */
public double outputDuration(double time1, double time2)
{
    double duration = (time2-time1)/1000;
    System.out.println("Generation time = " + twoDecPlaces(duration) + " seconds (" + twoDecPlaces(duration/60) + " mins)");
    // Return
    return(duration);
}

/* --------------------------------*/
/*                                  */
/*        OUTPUT UTILITIES          */
/*                                  */
/* --------------------------------*/

/* TWO DECIMAL PLACES */
/** Converts given real number to real number rounded up to two decimal places. */
@param number the given number.
@return the number to two decimal places. */
protected double twoDecPlaces(double number)
{
    int numInt = (int) ((number+0.005)*100.0);
    number = ((double) numInt)/100.0;
    return(number);
}