```java
import java.util.Vector;
import java.util.HashSet;
import java.util.Collections;
import java.util.Comparator;
import java.util.StringTokenizer;

/**
 * Reference implementation of Rough Sets rule generation using boolean condition
 * and decision attributes.
 */
public class RoughtSets {
   public static int DECISION_ATTRIBUTE_IDX=8;
   public static int LAST_CONDITION_ATTRIBUTE_IDX=DECISION_ATTRIBUTE_IDX-1;

   /**
    * Computes all equivalence classes. The result is a vector of patterns instead
    * of a vector of sets. A pattern is represented by the vector of values for the
    * different attributes considered for computing the equivalence classes (a subset of
    * a).
    * @param data The data array containing condition and decision attribute values
    *             for the e-mails.
    * @param attributes Array containing the indices of attributes considered for
     computing
    *                   classes
    * @return A vector with the patterns (values for all condition attributes considered)
    *         of the equivalence classes.
    */
   public static Vector<Vector<Boolean>> computeclasses(Boolean data[][], int
   attributes[]){
       Vector<Vector<Boolean>> result=new Vector<Vector<Boolean>>();

       //For each data row
       for (int i=0;i<data.length;i++){
          //Compute a combination of attributes
          Vector<Boolean> v=new Vector<Boolean>();
          for (int j=0;j<attributes.length;j++)
              v.add(data[i][attributes[j]]);

          //See if the combination has been computed and included before
          boolean included=false;
          for (int j=0;j<result.size() && !included;j++){
               Vector<Boolean> v2=result.get(j);
               boolean elEqual=true;
               for (int k=0;k<v.size() && elEqual;k++){
                    elEqual=v.get(k).equals(v2.get(k)); //Lo normal
               }
               if (elEqual) included = true;
          }

          //If the combination was not found before, include it now.
          if (!included){
             result.add(v);
          }
       }

       return result;
   }

   /**
    * Show an equivalence Class set using stdout
    * @param classes Equivalence classes patterns computed
    * @param attributes Attributes considered for computing the equivalence classes
    */
   public static void printEqClassSet(Vector<Vector<Boolean>> classes, int attributes[]){
      int j=0;
      for (Vector<Boolean> current:classes){
          System.out.println("Class: "+(j++));
          for (int i = 0 ; i<attributes.length; i++){
             System.out.print("a"+(i+1)+"="+(current.get(i)?"true, ":"false, "));
          }
          System.out.println();
      }
   }

   /**
    * Compute the lower approximation for a concept. The concept should be included in the
```

```java
     * data matrix. The lower approximation is computed as a vector of equivalence class patterns
     * instead of a set of objects (e-mails).
     * @param data Data matrix containing all atributes considered and also the concept
     * @param attributes Attributes (column indices) considered to compute the lower approximation
     * @param concept Concept column index to compute the lower approximation
     * @return set of equivalence classes patterns that conforms the lower approximation
     */
    public static Vector<Vector<Boolean>> lowerAprox(Boolean data[][], int attributes[], int concept){
        Vector<Vector<Boolean>> eqClasses=computeclasses(data, attributes);
        Vector<Vector<Boolean>> result=new Vector<Vector<Boolean>>();

        //printEqClassSet(eqClasses,attributes);

        //For each equivalence class
        for (int i=0;i<eqClasses.size();i++){
            Vector<Boolean> currentclass=eqClasses.get(i);
            //Check if all instances included in the equivalance class are positive

            boolean shouldBeIncluded=true;
            boolean onePositive=false;
            //for each instance see if it belongs to the class. If yes check if
            //the class can bee included in the lowerApprox
            for (int j=0;j<data.length && shouldBeIncluded;j++){
                boolean belongs=true;
                for (int k=0;k<attributes.length && belongs;k++){
                    belongs=(currentclass.get(k)==data[j][attributes[k]]);
                }

                //If current instance belongs current class, then, the current class
                //can be included into lowerApprox if its decission attribute is positive
                if (belongs){
                   shouldBeIncluded=(
                      data[j][concept]==null || data[j][concept]
                   );
                   onePositive=onePositive || data[j][concept]!=null && data[j][concept];
                }
            }

            //after checking all instances of the same class
            //if the class still should be included, then include in the result of the lowerApprox
            if (shouldBeIncluded && onePositive) result.add(currentclass);
        }

        return result;
    }

    /**
     * Compare two lower approximations to test if they are equivalent. This function is
     * used to compute reducts.
     * @param data Data considered for computing the lower approximations. The data
     *             is unique for both lower approximations but the attribute list may
     *             differ.
     * @param la1 First lower approximation to compare given in the form of set of
     *            equivalence classes patterns
     * @param at1 List of attribute indices considered for computing the first approximation
     * @param la2 Second lower approximation to compare given in the form of set of
     *            equivalence classes patterns
     * @param at2 List of attribute indices considered for computing the second
     *            approximation
     * @param concept Attribute (column) index considered as decision attribute (concept)
     * @return A value indicating if the lower approximations are equal
     */
    public static boolean equalLowerAprox(Boolean data[][], Vector<Vector<Boolean>> la1, int at1[], Vector<Vector<Boolean>> la2, int at2[], int concept){
        boolean equal=true;

        HashSet<Integer> dataRows1=new HashSet<Integer>();
        HashSet<Integer> dataRows2=new HashSet<Integer>();
        HashSet<Integer> allRows=new HashSet<Integer>();

        //For each instance in the first lower approximation
```

```
145         for (int i=0;i<la1.size();i++){
146             Vector<Boolean> la1current=la1.get(i);
147 
148             //For each instance j, see if it fists with i class pattern to build a vector
                of
149             //instances that fits with i pattern
150             for (int j=0;j<data.length;j++){
151                 boolean instanceFits=true;
152                 for (int k=0; k<at1.length && instanceFits;k++){
153                     instanceFits=(data[j][at1[k]]==la1current.get(k));
154                 }
155                 //If instance fits, it is included into the vector of instances that
                    matches la1
156                 if (instanceFits){
157                    dataRows1.add(j);
158                    allRows.add(j);
159                 }
160             }
161         }
162 
163         //For each instance in the first lower approximation
164         for (int i=0;i<la2.size();i++){
165 
166             Vector<Boolean> la2current=la2.get(i);
167 
168             //For each instance j, see if it fists with i class patern to build a vector of
169             //instances that fits with i pattern
170             for (int j=0;j<data.length;j++){
171                 boolean instanceFits=true;
172                 for (int k=0; k<at2.length && instanceFits;k++){
173                     instanceFits=(data[j][at2[k]]==la2current.get(k));
174                 }
175                 //If instance fits, it is included into the vector of instaces that matches
                    la2
176                 if (instanceFits){
177                    dataRows2.add(j);
178                    allRows.add(j);
179                 }
180             }
181         }
182 
183         equal=dataRows1.equals(dataRows2);
184 
185         //Check equivalence for concepts that handle null values
186         if (!equal){
187              equal=true;
188              /*Compute those ones that does not fit and see if they have null for the
189                concept/decision attribute */
190              for (Integer j:allRows){
191                 if (!dataRows1.contains(j) || !dataRows2.contains(j))
192                     equal=(data[j][concept]==null);
193              }
194         }
195 
196         return equal;
197     }
198 
199     /**
200      * Used internally to store reducts into a variable avoiding repeating them
201      * @param reduct Set of attribute (column) indices composing the reduct
202      * @param reducts Vector to store the reducts. This is a Input/Output parameter
203      */
204     private static void addReduct(int reduct[], Vector<String> reducts){
205             String currentReduct=new String();
206 
207             for (int k=0;k<reduct.length;k++){
208                currentReduct+=("a"+(reduct[k]+1)+(k==reduct.length-1?"":", "));
209             }
210 
211             boolean found=false;
212             for (int i=0;i<reducts.size() && !found;i++){
213                 found=reducts.get(i).equals(currentReduct);
214             }
215             if (!found)
216                 reducts.add(currentReduct);
217     }
```

```
218
219     /**
220      * Print a list of reducts using stdout
221      * @param reducts List of reducts to print
222      */
223     public static void printReducts(Vector<String> reducts){
224         for (int i=0;i<reducts.size();i++){
225             System.out.println("Reduct: "+reducts.get(i));
226         }
227     }
228
229     /**
230      * This internal function is able to recursively find reducts for a given data and a
         given concept.
231      * As a reducts should be minimal, when a candidate for reduct is found, we should
         recursively
232      * test if we can found a reduct drooping some attributes from candidate. Recursion is
233      * a good choice for doing that. However, we are aware that recursion could be avoided
234      * using a Stack data structure.
235      * @param data Data considered to compute the reducts. Should contain A (condition
         attribute matrix)
236      *             and X (decision attribute matrix).
237      * @param currentAttributes Attributes (columns) indices considered to compute the
238      *                          reducts. Initially this should be initialized to handle all
239      *                          condition attribute (A) indices.
240      * @param concept The concept (decision attribute or X) index considered to compute the
         reducts.
241      * @param reducts Is a input/output parameter that contains the reducts found
         recursively.
242      *                Initially, it should be contain a void vector. When the function
         ends,
243      *                this vector will have all reducts found.
244      * @return True if a reduct is found. This is useful to take advantage of recursion to
245      *         compute reducts.
246      */
247     private static boolean _computeReducts(Boolean data[][], int currentAttributes[], int
         concept, Vector<String> reducts){
248      Vector<Vector<Boolean>> lowerAproxCurrentAttributes=lowerAprox(data,
         currentAttributes, concept);
249      Vector<Vector<Boolean>> lowerAproxProposal;
250      boolean foundReduct=false;
251
252      if (currentAttributes.length==0){
253         return false;
254      }else if (currentAttributes.length==1){
255         addReduct(currentAttributes,reducts);
256         return true;
257      }
258
259      //Drop one attribute and, check if there is reduct using this subset of features. If
260      //found a reduct with a subset of attributes of currentAttributes param,
         currentAttributes
261      //is not a reduct.
262      for (int i=0;i<currentAttributes.length;i++){
263         int proposal[]=new int[currentAttributes.length-1];
264         for (int k=0;k<currentAttributes.length-1;k++){
265                if (k<i) proposal[k]=currentAttributes[k];
266                else if (k>=i) proposal[k]=currentAttributes[k+1];
267         }
268
269         lowerAproxProposal=lowerAprox(data, proposal, concept);
270
271         //proposal is a reduct candidate if the lower approximations of the concept
            computed with  proposal attributes
272         //is equivalent to the one computed by the initial attribute set.
273         if (equalLowerAprox(data, lowerAproxCurrentAttributes, currentAttributes,
            lowerAproxProposal, proposal, concept)){
274
275               //If proposal is a reduct candidate, and we can not find reducts by dropping
                  attributes, then proposal is a reduct.
276               if (!_computeReducts(data, proposal, concept, reducts))
                  addReduct(proposal,reducts);
277
278               //Both if proposal is finally a reduct, or if we can found a subset of
                  features included in proposal
279               //that is a reduct, we found a reduct. Therefore, the original attribute set
```

```
                (currentAttributes parameter)
280             //is not a reduct.
281             foundReduct=true;
282         }
283       }
284
285       // If we were not able to found a reduct using a subset of features included in
          currentAttributes,
286       //then currentAttributes is a reduct.
287       if (!foundReduct){
288           addReduct(currentAttributes,reducts);
289       }
290
291       return foundReduct;
292     }
293
294     /**
295      * Computes the reducts for a given data (condition and decision attributes), the
296      * indices for condition attributes and the index for the decision or concept
297      * attribute.
298      * @param data Data considered to compute the reducts. Should contain A (condition
          attribute matrix)
299      *             and X (decission attribute matrix).
300      * @param currentAttributes Attributes (columns) indices considered to compute the
301      *                          reducts. Initially this should be initialized to handle all
302      *                          condition attribute (A) indices.
303      * @param concept The concept (decision attribute or X) index considered to compute the
          reducts.
304      * @return Reducts for the given data sorted by size.
305      */
306
307     public static Vector<String> computeReducts (Boolean data[][], int currentAttributes[],
        int concept){
308        Vector<String> reducts=new Vector<String>();
309        _computeReducts(data, currentAttributes, concept, reducts);
310
311        Collections.sort(reducts,new Comparator<String>(){
312           public int compare(String x,String y){
313               return x.length()-y.length();
314           }
315        });
316        return reducts;
317     }
318
319     /**
320      * Check if a data row (object) fits to an equivalence class pattern
321      * @param data Data considered to check. Should contain A (condition attribute matrix)
322      *             and can contain X (decision attribute matrix).
323      * @param objIdx Object (row) index for the object to check if it belongs to class.
324      * @param attributes Attributes considered when we computed the equivalence classes
          set.
325      *                   The same ones should be used for comparison purposes.
326      * @param classPattern Equivalence class pattern that should be checked
327      * @return a boolean value indicating if the object belongs to class or not
328      */
329
330     private static boolean belongsClass(Boolean data[][], int objIdx, int attributes[],
        Vector<Boolean> classPattern){
331         boolean belongs=true;
332         for (int k=0;k<attributes.length && belongs;k++){
333             belongs=(classPattern.get(k)==data[objIdx][attributes[k]]);
334         }
335
336         return belongs;
337     }
338
339     /**
340      * Find a class pattern from a set of class patterns that fits with an specific object
341      * @param data Data considered to check. Should contain A (condition attribute matrix)
342      *             and can contain X (decission attribute matrix).
343      * @param objIdx Object (row) index for the target object.
344      * @param attributes Attributes considered when we computed the equivalence classes
          set.
345      *                   The same ones should be used for comparison purposes.
346      * @param classes The Vector for patterns to select the most appropriate one
347      * @return A pattern for an equivalence class that fits current object. Null if
```

```
348      *         we were not able to found a suitable pattern.
349      */
350     private static Vector<Boolean> findClass(Boolean data[][], int objIdx, int
        attributes[], Vector<Vector<Boolean>> classes){
351         boolean found=false;
352         Vector<Boolean> current=null;
353 
354         for (int i=0;i<classes.size() && !found ;i++){
355            current=classes.get(i);
356            found=belongsClass(data, objIdx,attributes, current);
357         }
358 
359         return found?current:null;
360     }
361 
362      /**
363       * Used internally to find the occurrences of a char into a string
364       * @param target the target string to count occurrences
365       * @param needle the character to check
366       * @return the number of occurrences of the char in the string
367       */
368      private static int countOccurrences(String target, char needle){
369          int count = 0;
370          for (int i=0; i < target.length(); i++)
371              if (target.charAt(i) == needle)
372                  count++;
373          return count;
374      }
375 
376      /**
377       * Shows a matrix of data (A and X)
378       * @param mat Matrix of data to show
379       */
380      public static void showMatrix(Boolean mat[][]){
381          for (int i=0;i<mat.length;i++){
382              for (int j=0;j<mat[0].length;j++)
383                  if (j<mat[0].length-1)
384                  System.out.print(mat[i][j]+", ");
385                  else
386                  System.out.print(mat[i][j]);
387               System.out.println();
388          }
389      }
390 
391     /**
392      * Generate a ruleset from the given data to guess a specific decision attribute.
393      * @param data Data considered to compute the reducts. Should contain A (condition
        attribute matrix)
394      *             and X (decission attribute matrix).
395      * @param attributes Attributes (columns) indices considered to compute the
396      *                   reducts. Initially this should be initialized to handle all
397      *                   condition attribute (A) indices.
398      * @param decisionAttr Decision attribute index in data to be considered in rule
        generation
399      * @return Array of strings where each string contains a rule.
400      */
401     public static String[] generateRules(Boolean data[][], int attributes[], int
        decisionAttr){
402        //attributes selected for MatX
403        int matXAttrs[]=new int [data[0].length-1];
404        for (int i=0;i<matXAttrs.length;i++) matXAttrs[i]=i;
405 
406        //Atrributes of Q
407        int qAttrs[]=new int[1];
408        qAttrs[0]=decisionAttr;
409 
410        Vector<Vector<Boolean>> classes=computeclasses(data,qAttrs);
411        HashSet<String> rules=new HashSet<String>();
412 
413        for (int i=0;i<data.length;i++){  //Para cada objeto
414           Vector<Boolean> elementClass=findClass(data, i, qAttrs, classes);
415 
416          Boolean matX[][]=new Boolean[data.length][data[0].length];
417           for (int j=0;j<data.length;j++)
418               for (int k=0;k<data[0].length;k++)
419                   if (k<data[0].length-1)
```

```
                    matX[j][k]=data[j][k];
                else if (j==i)
                        matX[j][k]=true;
                    else if (belongsClass(data,j,qAttrs,elementClass))
                        matX[j][k]=null;
                    else
                        matX[j][k]=false;

        Vector<String> matXReducts=new Vector<String>();
        _computeReducts(matX, matXAttrs, decisionAttr, matXReducts);

        //Select the shortest reduct
        Collections.sort(matXReducts,new Comparator<String>(){
           public int compare(String x,String y){
            return countOccurrences(x, ',')-countOccurrences(y, ',');
           }
           });
        String shortestReduct=matXReducts.get(0);

        StringTokenizer st = new StringTokenizer(shortestReduct,", ",false);

        String currentRule="if ";
        while (st.hasMoreTokens()) {
           String att=st.nextToken();
           int attIdx=Integer.parseInt(att.substring(1))-1;
           currentRule+=att+" = "+data[i][attIdx]+(st.hasMoreTokens()?" and ":" ");
        }
        currentRule+="then d1 = " + data[i][decisionAttr];
        rules.add(currentRule);
     }
     String ret[]=new String[rules.size()];
     int i=0;
     for(String j:rules){
        ret[i]=j;
        i++;
     }

     return ret;
  }

  /**
   * Launches a test of the class developed to test the data shown in the
   * manuscript
   * @param args Array of args received from command line (no one expected)
   */

  public static void main (String args[]){
      /*
      Example of data used in the paper. Columns 0-7 are the values for the
      condition attributes while column 8 contains the decision attribute (spam).
      Each row represents the different values for a concrete e-mail. In the paper
      this represents A and X matrixes.
      */
      Boolean sampleData[][]={
        {false, false, false, true,  true,  false, true,  false, true },
        {false, true,  true,  false, false, false, false, false, false},
        {true,  true,  true,  false, false, true,  false, false, false},
        {false, false, false, true,  false, false, false, true,  true },
        {false, false, false, false, false, false, true,  true,  true },
        {true,  true,  false, true,  false, true,  false, false, false}
      };

     //Show data
     System.out.println("Data");
     RoughtSets.showMatrix(sampleData);

     //Build the condition attribute index list
     int currentAttributes[]=new int[LAST_CONDITION_ATTRIBUTE_IDX+1];
     for (int i=0;i<currentAttributes.length;i++){
       currentAttributes[i]=i;
     }

     //Compute reducts
     Vector<String> reducts=RoughtSets.computeReducts(sampleData,
     currentAttributes,DECISION_ATTRIBUTE_IDX);
     //Show reducts for spam (last attribute of data)
```

```
495         System.out.println("Reducts for x: ");
496         RoughtSets.printReducts(reducts);
497 
498         //Compute rules
499         String rules[]=RoughtSets.generateRules(sampleData, currentAttributes,
            DECISION_ATTRIBUTE_IDX);
500         //Show rules
501         System.out.println("Rules: ");
502         for (String current:rules) System.out.println(current);
503     }
504 }
505 
```