# Product Defect Diagnosis

Yang Liang
Weian Ou
Gang Tang

with the assistance of
Jacquelyn M. Ormiston

May 2, 2003

**Abstract:**
The Ford Motor Company uses hazard analysis to identify part, component, and assembly failure rates over time from warranty claims data. In this project, we developed a combined method of decision tree and survival analysis, diagnosed the causes behind those failure rates, and revealed where, when and how failures occur from the defective hardware warranty claims.

The project was completed for the Ford Motor Company under the direction of Dr. Michael Cavaretta (Staff Technical Specialist, Ford Research Laboratory), in partial fulfillment of the requirements of Michigan State University MTH 844, advised by Professor Gábor Francsics.

# Table of Contents

Introduction ..... 1

Description of the Model ..... 1
Two Approaches to Data Mining of Defect Diagnosis ..... 1
Artificial Neural Networks ..... 1
Decision Tree Methods ..... 2

Survival Analysis ..... 3
The Survival Curve ..... 4
The Hazard Function ..... 4
Estimation Methods ..... 5

Analysis ..... 6
Basic Structure of Our Program ..... 6

Important Classes and Methods in this Java Program ..... 9
Class: SAC45App ..... 9
Class: survivalAnalysis ..... 9
Class: SurvivalAnalysisClassifier ..... 10

Result ..... 10

Summary ..... 11

Future Work ..... 12

Acknowledgement ..... 12

References ..... 13

Appendix ..... 14

# Introduction

The Ford Motor Company uses hazard analysis to identify part, component and assembly failure rates over time from warranty claims data. In this project we diagnosed the causes behind those failure rates through C4.5 decision tree and survival analysis. A decision tree is a tree-like structure used to classify data and represent the knowledge, which form the structure. In this work, the tree is utilized to find out what factors influence the failure rate of the investigated product and to determine the significance of the factors to product defect. In addition, the survival analysis method was combined into the decision tree for product failure prediction at each node and formation of the decision tree structure. The new analysis revealed where, when and how failures occurred from the defective hardware warranty claims. With this information, the Ford Company may arrange their customers' service more efficiently and diagnose the hidden damaging factors in car manufacturing. This will also make it possible to use fewer company resources to achieve higher customer satisfaction.

# Description of the Model

This project used a data mining or Knowledge Discovery from Database method. Data mining is used in two ways:

- Automated prediction of trends and behaviors and
- Automated discovery of previously unknown patterns.

Automated prediction of trends and behaviors involves data mining, which automates the process of finding predictive information in large databases. Thus, the prediction of new question can be handily solved by the existing data. Expert systems and many pattern recognition applications fall into this area.

Automated discovery uses data mining tools to sweep through databases and identify previously hidden information. The present project, which is to seek hidden factors of creating abnormal product failure, belongs to this category.

## Two Approaches to Data Mining of Defect Diagnosis

To achieve the goals of data mining, different algorithms can be applied. There are two major approaches to the present question of product defect diagnosis:

1. *Artificial neural networks and*
2. *Decision trees.*

### Artificial Neural Networks

Artificial neural networks are nonlinear predictive models, which simulate the behavior of biological neurons and neural network [1]. A neural network is a powerful tool in

behavior prediction, pattern recognition, and artificial intelligence and is very convenient to achieve machine learning. Safer has used it in "The Application of Neural Networks to Predict Abnormal Stock Returns Using Insider Trading Data" (2002). Povinelli has used it in "A New Temporal Pattern Identification Method for Characterization and Prediction of Complex Time Series Events" (2003). However, neural networks are not very convenient to classify factors that may affect the failure of product from the most important ones to the least important ones [2].

**Decision Tree Methods**

For this project, decision trees are more natural candidates. Decision trees are tree shaped structures that represent sets of decisions or classifications. They are a clear structure from top (root) to bottom (leaves). Specific decision tree methods include classification and regression trees. These decisions generate rules for the classification of a dataset. Hence, the whole dataset (root) can be classified into several subsets (nodes) by certain attribute of the data (each case of the attribute can be a branch leading to a node). Under each node, according to the similar process, a sub-tree can then be generated. For our project, the factors (attributes) of product defect will be classified level by level, and the higher the tree level (closer to root), the more important a factor is.

There are many decision tree methods used for mining knowledge, i.e., C4.5 [3], CART, etc. A C4.5 algorithm was used in this project. The C4.5 method constructs a decision tree from a set of training objects. It performs a top-down irrevocable search and uses information gain ratio as the criterion for selecting the branching attribute. Each time, the method picks one attribute and computes the information gain ratio for this attribute. After we generated the information gain ratio for all the attributes of the cars, we split the data by choosing the attribute with highest gain ratio then continue by repeating the same procedure, choosing the attribute with highest gain ratio as compared to the remaining attributes.

Let the node of the tree contain a set $T$ of cases, with $|C_j|$ of the cases belonging to one of the predefined classes $C_j$. The information needed for classification in the current node is

$$Info(T) = -\sum_j \frac{|C_j|}{|T|} \log_2 \left[ \frac{|C_j|}{|T|} \right].$$

This value is defined as entropy. It measures the average amount of information needed to identify the class of cases. When the entropy is high, this indicates that the class is uniform and boring. Otherwise, the class is varied and interesting. Assume that the tree will divide the cases into $n$ subsets by using attribute $X$ as the branching attribute. Let $T_i$ denote the set of cases in subset $i$. The information required for the subset $i$ is $Info(T_i)$. The expected information required after choosing attribute $X$ as the branching attribute is the weighted average of the sub-tree information:

$$Info_X(T) = \sum_i \frac{|T_i|}{|T|} \times Info(T_i).$$

Now we can find the information gain, given by

$$Infogain(X) = Info(T) - Info_X(T).$$

C4.5 uses an information gain ratio. We can find this ratio by dividing the information gain by $split \quad Info$. This is given by

$$split \quad Info_X(T) = -\sum_{i=1}^{n} \frac{|T_i|}{|T|} \log_2 \left[ \frac{|T_i|}{|T|} \right].$$

This value represents the potential information generated by dividing $T$ into $n$ subsets. The gain ratio is used as the criterion.

As a smaller value in the information (low entropy) corresponds to a better classification, the attribute with the maximum information gain ratio is selected for the branching of the node. However, for a large data set with hundreds of attributes to be classified, the calculation could be impossible to be accomplished if without adequate optimization. In this case, we introduce a simple optimization algorithm for decision tree building:

> A node without sub-tree is the one whose possible sub-tree selected by $C4.5$ algorithm has an information-gain ratio less than some tolerance $\varepsilon$, where ε is the pre-determined positive small value.

In the product defect project, for example, a small information-gain ration less than $\varepsilon$ represents very similar failure rate of each branches, and hence, there is no important factor under this node that could be regarded as one capable of influencing product defect.

## Survival Analysis

Survival analysis arose from clinical trials and other medical studies. Interest of these researches often centers on an assessment of the times until the participants experience some specific event [4]. In our project, we viewed the new cars as the participants, which may experience some specific event, here stated as failure. Since this failure rate can be caused by many reasons, a good way to measure the effect of each possible reason along the time period is to determine the possibility of failure at each separate time frame during the whole warranty period. Using a survival function generated by the historical data, we projected the future possibility of a specific failure.

Censoring is the unique and the first process, which distinguishes survival analysis from other statistical methods [5]. A censored observation contains only partial information about the random variable of interest. For the present project, a fixed time censoring was applied. Suppose $T_1$, $T_2$, … $T_n$ are the time that products fail, we can only observe $Y_1$, $Y_2$, … $Y_n$ where

$$Y_i = \begin{cases} T_i, if\ T_i < T_c \\ T_c, otherwise \end{cases},$$

where $T_c$ is the censoring time limit. By doing this, the experiment can stop at a previously specified time $T_c$.

**The Survival Curve**

After censoring, the survival curve can be obtained by calculating the survival ratio of the investigated object. Take an example of a group of patients whose survival is observed as time elapses. Denote $s(t)$ as the survival function, and suppose the study was completed within 10 years. Displayed below in Figure 1 is the imagined $s(t)$ curve.

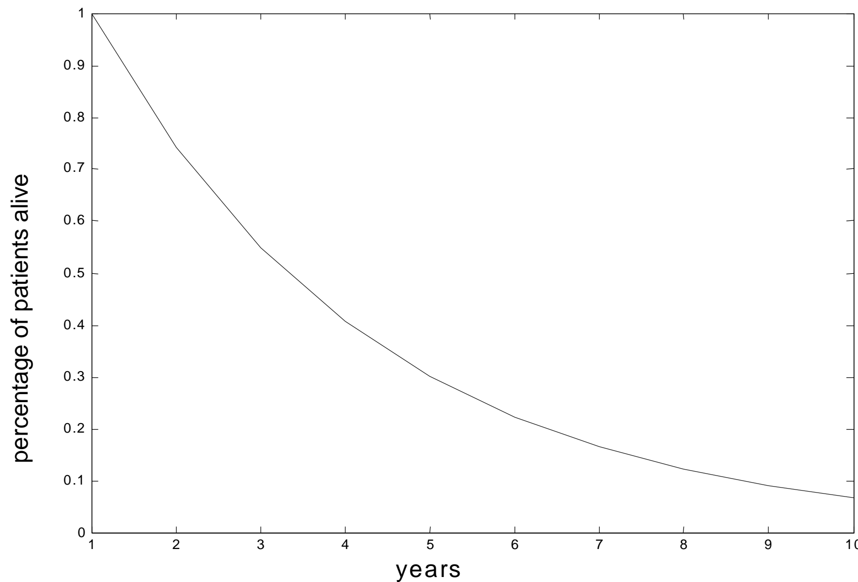


**Figure 1. The imagined survival function s(t) of patients' survival over a period of 10 years is a downward sloping curve, which decreases at a decreasing rate.**

**The Hazard Function**

Related to survival function $s(t)$, is the hazard function $h(t)$, which is the proportion of persons who fail at time t from among those who have not failed previously. It can be expressed by the following equation:

$$h(t) = -\frac{d}{dt}[\log s(t)] .$$

**Estimation Methods**

Methods of estimation of $s(t)$ from observed data include both parametric methods and non-parametric methods. For one sample data, the methods include exponential, Gamma, Weibull, Rayleigh, etc.

Non-parametric algorithms can be good choices when an unknown set of survival data is being investigated. These include life tables, Kaplan-Meier estimator, regression, etc. However, some of the non-parametric models lack the capability of predicting a future survival curve.

Parametric methods can give parameters with physical or realistic meaning and are also very precise, provided that we know which parametric model is best fitted for the observed sample. Among the parametric models, the Weibull model can be a very useful tool, in which *s(t)* is given by

$$s(t) = e^{-(\lambda t)^{\alpha}} \qquad \alpha > 0, \lambda > 0 ,$$

and the related hazard function is

$$h(t) = \alpha\lambda(\lambda t)^{\alpha-1} .$$

With the parameters identified, one can easily predict the future survival rate or hazard rate by this equation.

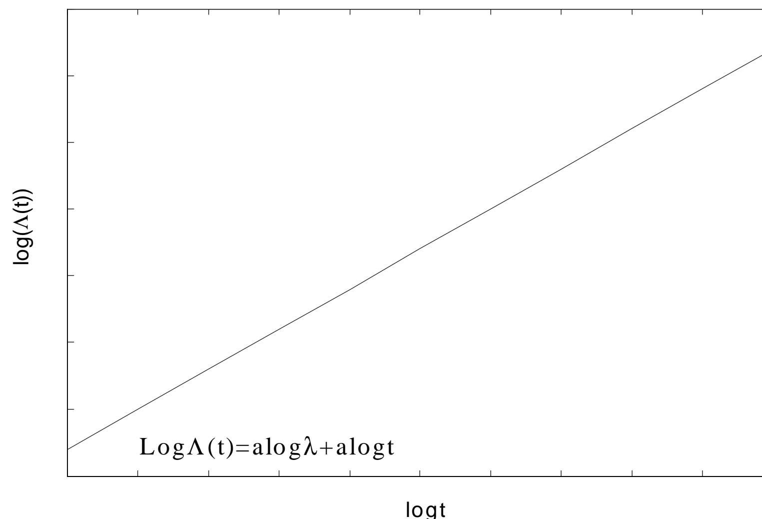


**Figure 2. This graphical method of examining goodness of fit for the Weibull model shows that the fit was very good, since the graph is indeed linear.**

In order to examine how well the observed data matchs with the assumed model, a goodness of fit algorithm must be applied. Consider the example of the Weibull method, which denotes $\Lambda(t) = -\log[S(t)]$. A direct graphical method is useful for this purpose, which is actually linear if the model matches with the data, so the quality of the fit is very clear. This is demonstrated in Figure 2.

# Analysis

With the basic model discussed before, we constructed our own program for the project. The program is a fundamental fulfillment of the algorithm, and it requires a strict input data format. In the following parts, the basic structure of the program will be introduced. Three flow charts were also provided for better understanding of the program. Sampled data was also tested and the related explanation was given. However, due to the fact that product defect data is commercially classified material, the actual product defect result is not listed in this report. Instead, an example of patients' survival data is showed for demonstration.

## Basic Structure of Our Program

Using the existing historical data, we built a tree to determine what error or problem occurred in the cars. The next step was to determine possible problems of new cars in the future. Here we needed survival analysis to determine the possible failure rate of the car in each attribute at any time during the warranty period. Then, by use of decision trees, we made a projection of when, where and how new cars will fail.

Following the construction of the tree, we created a flow chart of the program's basic structure, as shown in Figure 3. The input file is data file, which is in ARFF format. There are also some parameters that should be defined during the input. The data file is a specially formatted file, which stores the description of each attribute in data set, and the data set. The parameters include censoring time; estimating time, gain ratio tolerance, maximum bin number, minimum case number, input file name and output file name. Details of parameters analysis are shown in Figure 4. The output files are decision tree files, which are generated from the data file under specified parameters. They are the ASCII file with tree-like structure and some other information, such as defined parameters, and survival rate and error at each node.

The following is the basic routine of our program.

**Step 1**. Create the data set by selecting the cases from the data file according to specified parameters. Make the attribute set, which includes all of attributes in the data file.

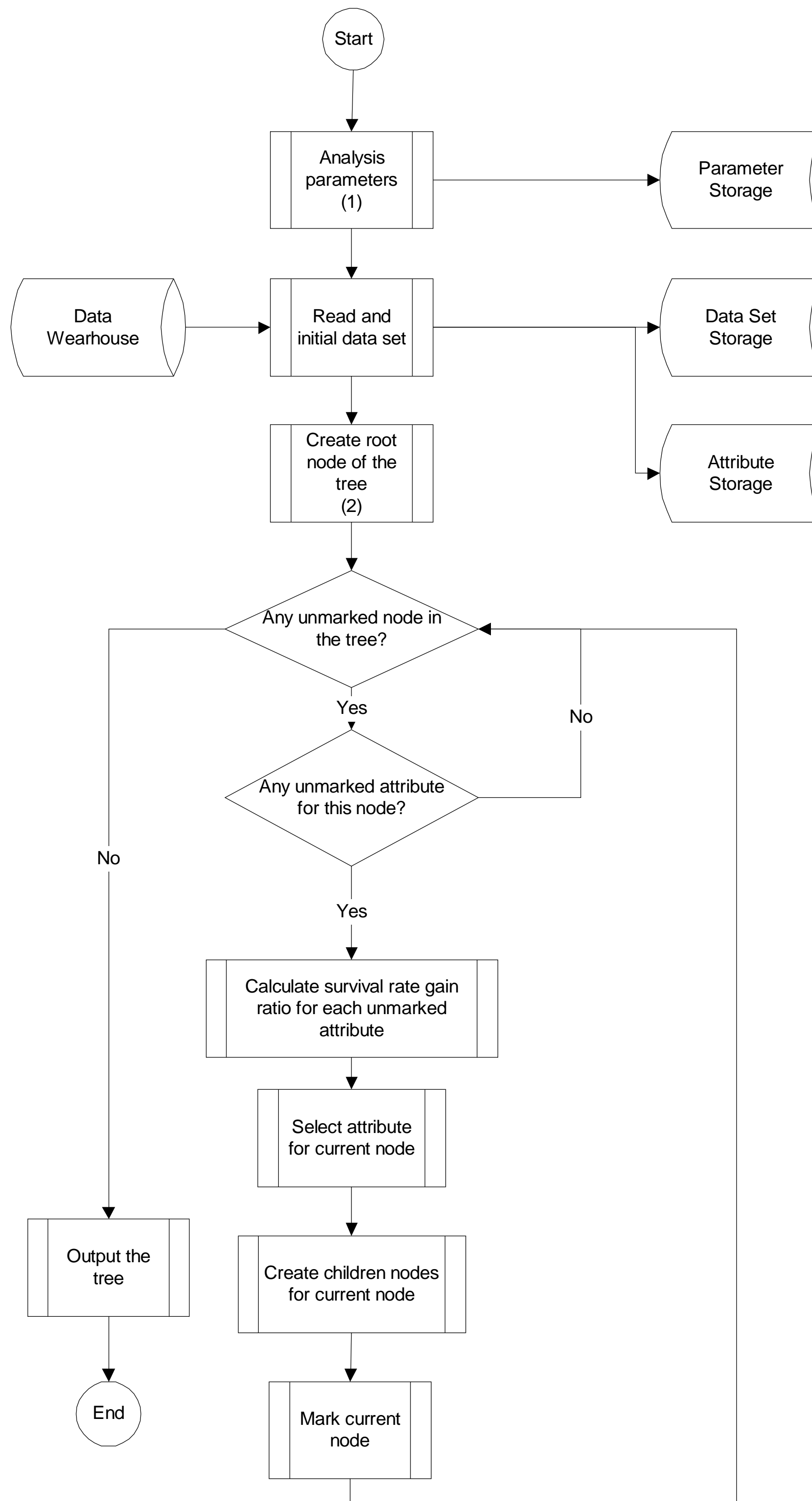


**Figure 3. The flow chart of the program illustrates the basic guidelines for generating an output file.**

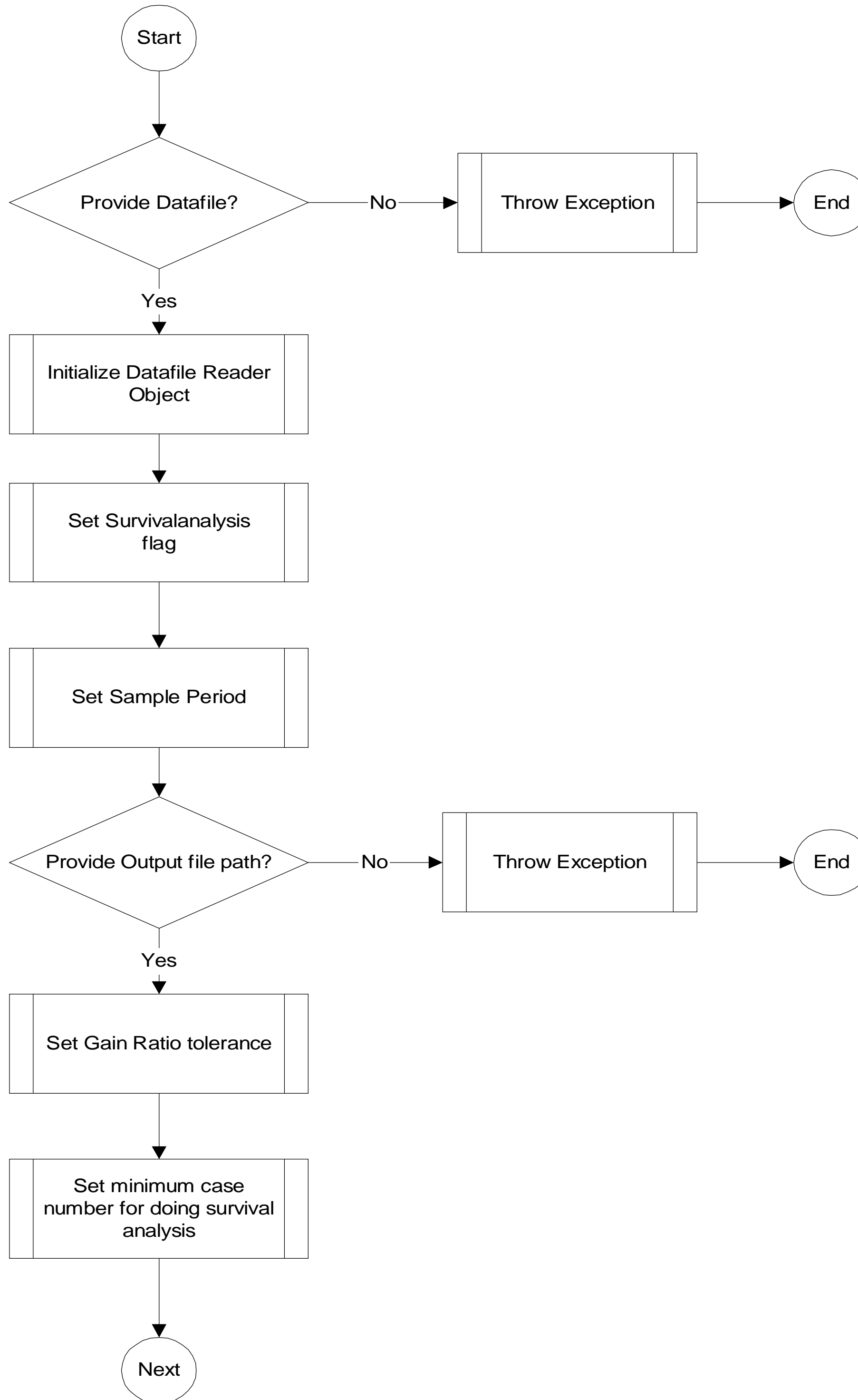


**Figure 4. The flow chart of the program illustrates the basic guidelines for generating an output file (Analysis parameters (1)).**

**Step 2**. Set the root node. By using Weibull model on the data set, compute the survival rates at the future time point specified in parameters for different attributes.

**Step 3**. For each unmarked node in the tree, make an unselected attribute set, which includes all the unselected attributes for the current node.

**Step 4**. Repeat Step 3 for every attribute in the unselected attribute set. Then use the survival rate as the key value to determine which attribute will give the largest information gain ratio. Select this attribute as the next branch from current node. Delete the selected attribute from the unselected attribute set for the next generation node of the current node. Compute and record the survival rate and its error for current node.

**Step 5**. Repeat Step 3 and Step 4 for each leaf in the current tree, until no branch needs to be split from each node.

**Step 6**. Output the result.

## Important Classes and Methods in this Java Program

Following the basic structure of the program, several classes and methods were created in Java (with Weka library [6]) to build up the routine:

**Class: SAC45App**
This class is the entrance of this application. It reads the data file, gets parameters, filters and modifies the data and last, it outputs the result.

**Method: setParams**
In this function, we read and set the parameters needed by two other classes: *survivalAnalysis* and *SurvivalAnalysisClassifier*.

**Method: run**
This function filters the data file, and creates an instance of class *SurvivalAnalysisClassifier* to make the decision tree and outputs the result.

**Class: survivalAnalysis**
This class is used to compute the survival rate at specified time point according to given cases and sample period. We constructed the Weibull model in this class for given data and also gave the estimated root of mean square error.

**Method: computeSurvivalRate**
This function calculates the survival rate at the projected time point according to the given events and when the events occurred. The survival model we used in this function was the Weibull model, which is explained in the survival analysis section.

**Class: SurvivalAnalysisClassifier**
This class is used to create the decision tree on given cases in the C4.5 method combining survival analysis.

**Method: buildClassifier**
The main purpose of this function is to check the situation of currently working node and to calculate the survival rate and its absolute error for this node. If there is an unmarked attribute, it will call *makeTree*, which is used to divide the node into branches.

**Method: computeSurvivalGainRatio**
This function computes survival gain ratio for given attribute and cases. It calls *computeSurvivalRateGain*, which is used to compute the information gain according to the projected survival rate and *computeSurvivalSplitInfo*, which is used to compute the associated splitting information gain.

**Method: makeTree**
In *makeTree* , we use *computeSurvivalGainRatio* to calculate the gain ratio by using survival rate generated from the survival analysis for each attribute. The attribute, which has the maximum gain ratio with its survival rate, will be used to split current cases into sub-groups as the children nodes.

## Result

The result generated by running our program is shown in Table 1.

The first attribute to branch from the root node is *rx*. The reason for this is that *rx* has the highest information gain ratio in all the attributes. There are two branches under the first node. They are separately *rx*=1 and *rx*=2. There are no sub-branches under the first level-one node. Under the second node connecting the branch whose *rx* value is 2, there are sub-trees divided by another attribute *number*. This attribute has the highest information gain ratio within the sub group data of *rx*=2. At the same time, attribute *size* has the highest information gain ratio under branch *number* = '(-inf-2.75)'.

In each line, the ratio in the parentheses by the attribute value is ratio of survival patients over the total patient. The first value following this ratio is the survival ratio at the projected time point, and the second following number in the parentheses is the survival ratio for the censoring time point. In our example they are the twentieth month and the thirtieth month. As we can see, the projected value is smaller than survival value at the censoring time point. The last value in the line is the error of the projection. The error here represents the root mean square error. It is the square root of the difference between real data point and our projected curve.

This result shows that the medicine makes the most different effect on patients. That is the reason why this *rx* attribute divides the root first. Also the sub groups with attributes *rx* = 2 and *number* ='(2.75-4.5)' have the highest survival rates.

**Table 1. Program result: a decision tree classified by attributes through survival analysis prediction.**

```
C4.5 Decision Tree on Survival Analysis Parameters:
Input file: C:\bladder1.arff
Output file: C:\output
Maximum bin number for discretizing numerical attribute: 4
C4.5 GainRatio Tolerance: 0.0010
Sampler period: 20
Projected period: 30

SurvivalAnalysisClassifier
Using Survival Analysis(sample period:20)

(99:151)Pro:0.10682927195855038(Ori:0.3443708609271523)Err:0.499919469654307
rx=1(66:94)Pro:0.04315922710875903(Ori:0.2978723404255319)Err:0.5818819485668938
rx = 2
|  (33:57)Pro:0.24344447686351672(Ori:0.42105263157894735)Err:0.411681254
5608825
|  number = '(-inf-2.75)'
| | (14:31)Pro:0.4360935967990892(Ori:0.5483870967741935)Err:0.28101966
194911615
| |  size ='(-inf-2.5)'(11:23)Pro:0.391049986053993(Ori:0.5217391304347826)
Err:0.3033058386213427
| |  size ='(2.5-4)'(2:6)Pro:0.4781043048830641(Ori:0.6666666666666667)Err:0.21 47
0936954700384
| |  size = '(4-inf)'(1:2)Pro:-1.0(Ori:0.5)Err:-1.0
|  number ='(2.75-4.5)'(4:10)Pro:0.4758998790331292(Ori:0.6)Err:0.31445706538
14034
|  number ='(4.5-inf)'(15:16)Pro:0.006378248946507072(Ori:0.0625)Err:0.7146814
377280586
```

## Summary

In this project, a combined survival analysis and decision tree method was established and tested for product defect analysis. The C4.5 model was modified and integrated by new survival analysis feature such that each nodes of the tree will be decided by the failure rate predicted by survival analysis. Furthermore, the more important a attribute is, the earlier that attributed is selected to divided the nodes into sub-trees. Therefore, the constructed decision tree reflects the attributes importance to product failure from top to bottom.

Also, a Weibull model was used in survival analysis for its simplicity and diversity. With the algorithm established, the resulting Java program was setup. They

involve 3 Java classes. The new algorithm program was tested by the data provided Ford motor company. The results showed that the damaging factors in manufacture were extracted and sorted by their importance. Namely, the new algorithm was verified to be successful. The Java program was also proved to work efficiently and correctly. Meanwhile, the algorithm was also tested to be powerful and promising for future improvement.

# Future Work

It is also noticeable that the present version of Java program is simply a primitive version of the established algorithm. There are several assumptions (or simplifications) for the Java program:

- Only Weibull model is used for all the databases, and therefore bathtub hazard functions can not be approximated by the present model.
- For each database analysis, only one survival model is needed for all failure rate prediction, which is actually not always true.
- All of the records of the produced cars are included in the analysis, and hence, the calculation burden is enhanced.
- Lots of pre-processing of the data is necessary for correct analysis, including deleting unnecessary attributes, converting attributes into designed format, etc.

Considering the simplifications, some improvements are necessary for more efficient analysis:

- Multiple survival models can be applied for different data, and the best-fitted one will be selected.
- Faster method of choosing tree nodes and attributes is desirable for large database. Also, for simplicity of the tree structure, combination of similar branches can be developed
- Improved data format is quite useful for better analysis.
- A friendly users' interface is highly recommended. This may includes some standard pre-processing routines

# Acknowledgement

The team acknowledges the Professors Charles R. MacCluer, and Gábor Francsics for their valuable advises and our liaison Dr. Michael Cavaretta for his help on this project.

## References

[1] Cox DR, Oakes D., "Analysis of survival data", London: Chapman and Hall, 1984.

[2] Lagakos, Stephen W., "Statistical analysis of survival data", In: Bailar, J. Medical uses of statistics. Boston: NEJM Books, 1992.

[3] Ruggieri, S., "Efficient C4.5", IEEE transactions on knowledge and data engineering, 14.2 (2002): 438-444.

[4] Subrahmanyam, M. and Sujatha, C., "Using Neural Networks For the Diagnosis of Localized Defects in Ball Bearings.", Tribology International 30.10 (1997): 739-752.

[5] Wang, S. S., "Artificial Intelligence and Expert Systems for Diagnostics", Proc. of the First International Conference for Machinery and Diagnostics and Exhibition, Las Vegas, 1989.

[6] The University of Waikato, Weka free license package 3.2.3.

## Appendix

This section includes database and programs.

---

```
/**
 * Title:        C4.5 with Survival Analysis application
 * Description:  create application of SurvivalnAlaysisClassifier
 * Input: Parameters, see below
 * Output: the tree (output.txt) with survival rate
 * @version 1.0
 *
 * Parameter:
 *-t <source filename>
 *    Train file(arff) full path and name. must define
 *
 *-r <output filename>
 *    Output file full path and name. must define
 *
 *-s
 *    Using survival analysis, or pure c4.5
 *
 *-p <number>
 *    Sample period, must define
 *
 *-j <number>
 *    Projected period, must define
 *
 *-b <number>
 *    Discretizing the real value into <number> bins, default 4.
 *
 *-e <number>
 *    Gain ratio tolerance which should be very small, default 0.001.
 *
 *-n <number>
 *    Minimum case number to do the survival analysis, default 4
 */

import weka.filters.*;
import weka.core.*;
import java.io.*;
import java.util.*;


public class SAC45App {
```

```java
/** train data */
private Instances m_Data;

/** classifier */
private SurvivalAnalysisClassifier m_Classifier;

/** filter */
DiscretizeFilter m_Filter = new DiscretizeFilter();

/** train file path */
static String m_sTrainFile;

/** output file path */
String m_sOutputFile;

/** using survival analysis, or only c4.5 */
static boolean m_bUseSA = false;

/** maximum bin number */
private int m_iMaxBinNum = 4;

/** projected period */
private int m_iProjectedPeriod;

/** gain ratio tolerance */
double m_dGainRatioTol = 0.001;

/** sample period */
private int m_iSamplePeriod = 0;

/** minimum cases number to do the survival analysis*/
private int m_iMinCases = 4;

public SAC45App(Reader rd, boolean bUseSA) throws Exception {
  m_Data = new Instances(rd);
  m_Classifier = new SurvivalAnalysisClassifier(bUseSA);
}

public SAC45App(Reader rd) throws Exception {
  m_Data = new Instances(rd);
}


/**
 * print parameters
 */
```

```
  public String toString() {
    StringBuffer sb = new StringBuffer();
    sb.append("C4.5 Decision Tree on Survival Analysis\n");
    sb.append("Parameters:\n");
    sb.append("Input file: " + m_sTrainFile + "\n");
    sb.append("Output file: " + m_sOutputFile + "\n");
    sb.append("Maximum bin number for discretizing numerical attribute: " +
m_iMaxBinNum + "\n");
    sb.append("C4.5 GainRatio Tolerance: " + m_dGainRatioTol + "\n");
    if (m_bUseSA) sb.append("Sampler period: " + m_iSamplePeriod + "\n");
    if (m_bUseSA) sb.append("Projected period: " + m_iProjectedPeriod + "\n");
    sb.append("\n");
    return sb.toString();
  }

  /**
   * set parameters
   */
  private void setParams(String[] args) throws Exception {

  // set the sample period
    String speriod = Utils.getOption('p',args);
    try {
      int iperiod = Integer.parseInt(speriod);
      if (iperiod > 0) {
        m_iSamplePeriod = iperiod;
        SurvivalAnalysisClassifier.setSamplePeriod(iperiod);
      }
      else {
        throw new Exception("the sample period must be greater than 0.");
      }
    } catch (Exception e){
      throw new Exception("check the sample period.");
    }

    // set the projected period
    String sproperiod = Utils.getOption('j',args);
    try {
      int iperiod = Integer.parseInt(sproperiod);
      if (iperiod > 0) {
        m_iProjectedPeriod = iperiod;
        survivalAnalysis.setProjectedPeriod(iperiod);
      }
      else {
        throw new Exception("the projected period must be greater than 0.");
      }
```

```
    } catch (Exception e){
      throw new Exception("check the projected period.");
    }

  // set the output file path
    m_sOutputFile = Utils.getOption('r',args);
    if (m_sOutputFile.length() == 0)
      throw new Exception("No output file specified.");
  // set maximum bin number
    try {
      int ibins = Integer.parseInt(Utils.getOption('b',args));
      if (ibins > 0) m_iMaxBinNum = ibins;
    } catch (Exception e){
      System.out.println("Not specifying maximum bin number, using default(" +
this.m_iMaxBinNum + ").");
    }

  // set the gain ratio tolerance
    try {
      double dgainrat = Double.parseDouble(Utils.getOption('e',args));
        if ( dgainrat < 1.0 && dgainrat >= 0.0)
          m_dGainRatioTol = dgainrat;
        else
          System.out.println("The specified gain ratio tolerance number is outbounded, using
default(" + SurvivalAnalysisClassifier.getTolerance() + ").");
    } catch (Exception e) {
      throw new Exception("check the gain ratio tolerance number.");
    }

  // set minimum case number
    try {
      int icase = Integer.parseInt(Utils.getOption('n',args));
      if (icase > 0) {
        SurvivalAnalysisClassifier.setMinCases(icase);
        m_iMinCases = icase;
      }
    } catch (Exception e){
      System.out.println("Not specifying minimum case number, using default(" +
SurvivalAnalysisClassifier.getMinCase() + ").");
    }

  }

//  /**
//   * Classify given instance
//   */
```

```
//  public void classifyInstance(Instance data) throws Exception {
//    m_Filter.input(data);
//    Instance filtereddata = m_Filter.output();
//    double predicted = m_Classifier.classifyInstance(filtereddata);
//    System.err.println("Instance Classified as:" +
//              m_Data.classAttribute().value((int)predicted));
//  }

  public static void main(String[] args) {
    try {
      // get train file name
      String sTrainFileName = Utils.getOption('t',args);
//      sTrainFileName = "c:\\bladder1.arff";
      // whether use survival analysis.
      if (sTrainFileName.length() != 0) {
        Reader reader;
        reader = new FileReader(sTrainFileName);
//        String sIdxFldNo = Utils.getOption('c',args);
        m_bUseSA = Utils.getFlag('s', args);
        SAC45App SAC45App1 = new SAC45App(reader);
        m_sTrainFile = sTrainFileName;
        SAC45App1.setParams(args);
        SAC45App1.run();
      } else {
        throw new Exception("No train file provided.");
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  private void run() throws Exception{
    m_Data.setClassIndex(m_Data.numAttributes() - 1);
    m_Filter.inputFormat(m_Data);
    String sattRange = "first-";
    sattRange += Integer.toString(m_Data.numAttributes () - 2);
    String[] sOptions = {"-B",Integer.toString(m_iMaxBinNum) ,"-O"};
    m_Filter.setOptions(sOptions);      // must before setAttributeIndices
                                        // or set indices will be disabled
    m_Filter.setAttributeIndices(sattRange);
    Instances filteredData = Filter.useFilter(m_Data,m_Filter);
//    if (filteredData.numClasses () > 2) {
//      throw new Exception("More than two class values.");
//    }
//    System.out.print(filteredData.toString ());
```

```
    SurvivalAnalysisClassifier.setGainRatioTolerance(m_dGainRatioTol);
    SurvivalAnalysisClassifier.setMinCases(m_iMinCases);
    if (filteredData.numClasses() > 2) {
      Instances[] splitData = this.split(filteredData);
      SurvivalAnalysisClassifier[] SAClassifiers = new
SurvivalAnalysisClassifier[splitData.length];
      for (int i=0; i < splitData.length; i++)
      {
        SAClassifiers[i] = new SurvivalAnalysisClassifier(m_bUseSA);
        SAClassifiers[i].buildClassifier(splitData[i]);
        Writer mr;
        mr = new FileWriter(m_sOutputFile + i + ".txt");
        mr.write(this.toString());
        mr.write("Part: " + filteredData.classAttribute().value(i*2) + "\n\n");
        mr.write(SAClassifiers[i].toString());
        mr.close();
      }
    }
    else {
      m_Classifier = new SurvivalAnalysisClassifier(m_bUseSA);
      m_Classifier.buildClassifier(filteredData);
      Writer mr;
      mr = new FileWriter(m_sOutputFile + ".txt");
      mr.write(this.toString());
      mr.write(m_Classifier.toString ());
      mr.close ();
    }
  }
  /**
   * split the multi-parts into part groups
   */
  private Instances[] split(Instances data) {
    Attribute att;
    att = data.classAttribute();
    int ivalues = att.numValues()/2;
    Instances[] splitData = new Instances[ivalues];
    FastVector fv = new FastVector(2);
    fv.addElement("0");
    fv.addElement("1");
    Attribute newatt = new Attribute("event",fv);
    FastVector atts = new FastVector(data.numAttributes());
    for (int i = 0; i < data.numAttributes() - 1; i++) {
      atts.addElement(data.attribute(i));
    }
    atts.addElement(newatt);
    for (int j = 0; j < ivalues; j++) {
```

```
      splitData[j] = new Instances(att.value(j*2),atts,(int)data.numInstances()/ivalues);
      splitData[j].setClassIndex(data.classIndex());
    }
    Enumeration instEnum = data.enumerateInstances();
    while (instEnum.hasMoreElements()) {
      Instance inst = (Instance) instEnum.nextElement();
      double dclsval = inst.classValue();
      inst.setClassValue((int)dclsval % 2);
      splitData[(int)dclsval/2].add(inst);
    }
    return splitData;
  }
}
```

---

```
/**
 * survival analysis, provided the method to calculate the survival rate and RMS error.
 * The weibull model is used in the class.
 *
 * @version $Revision: 1.0 $
 */


import weka.core.*;

public class survivalAnalysis {

  public survivalAnalysis() {
  }

  /* the estimated survival rate */
  private double m_dSurvivalRate = -1;

  /* the specified model number*/
  private int m_iModel = 1;

  /* the estimated period */
  private static double D_PROJECTEDPERIOD = 36;

  /* the norm 2 error */
  private double m_Norm2Error = -1;

  /** sample period */
  private double m_dSamplePeriod = -1;

  /** the unit size in period */
```

```
  private double m_dTimeUnit = 1.0;

  /** weibull's params */
  private double m_alpha = 0;

  /** weibull's params */
  private double m_lambda = 0;

  /** the matrix vertical size */
  private int m_iVectorSize = -1;

  /** The time limit used to select the m_dTimeUnit. >=8, 2 or 1. */
  public static final double TIMESIGN=20.0;

  /** Return the survival rate */
  public final double getSurvivalRate() {
    return m_dSurvivalRate;
  }

  /** Return the prjected period */
  public final double getProjectedPeriod() {
  return D_PROJECTEDPERIOD;
  }

  /** Return the norm 2 error */
  public double getNorm2Error() {
  return m_Norm2Error;
  }

  /** set projected period */
  public static void setProjectedPeriod(int properiod) {
  D_PROJECTEDPERIOD = (double)properiod;
  }
  /**initial model, routine will select the best-fit model to calculate the failure rate
   * if no suitable model selected, return false.
   */
  public boolean initModel(Instances data, double dSamplePeriod) throws Exception{
    if (dSamplePeriod > 0) {
      m_dSamplePeriod = dSamplePeriod;
    }
    return initModel(data);
  }

  public boolean initModel(Instances data) throws Exception{
//    try {
      //check truecases and total cases in the dataset
```

```
      if (data.numInstances() == 0) {
        return false;
      }
      if (m_dSamplePeriod <= 0 ) m_dSamplePeriod = D_PROJECTEDPERIOD;
      double[] times = new double[data.numInstances()];
      boolean[] events = new boolean[data.numInstances()];
      int iEventAtt = data.numAttributes () - 1;
      int iTimeAtt = iEventAtt - 1;
      int iCntEnt = 0;
      double iMeaTm = 0;  // to get the arrange of time
      for (int i = 0; i < times.length; i++) {
        times[i] = data.instance(i).value(iTimeAtt);
//        events[i] = (boolean)data.instance(i).value(iEventAtt);
        if (data.instance(i).value(iEventAtt) == 1) {
          if (iMeaTm < times[i]) iMeaTm = times[i];
          events[i] = true;
          iCntEnt +=1;
        } else {
          events[i] = false;
        }
      }
      if (iCntEnt == 0) {
      // no event happened.
        m_dSurvivalRate = 1;
        return true;
      } else if (iCntEnt == events.length) {
      // all fail.
        m_dSurvivalRate = 0;
        return true;
      }
      if (iMeaTm >= TIMESIGN) m_dTimeUnit = 2.0;
      computeSurvivalRate(times,events);
      return true;
/*    } catch(Exception e) {
//      return false;
    }
*/
  }


  /** parameters: time - time peroid in [0, D_PROJECTEDPERIOD]
   *            event - 0 or 1, 1 means event happened, 0 means nothing happened
   *
   * what needs to do: calculate alpha and lambda, then compute m_dSurvivalRate at
D_PROJECTEDPERIOD,
   *       and return m_dSurvivalRate.
```

```
  *       If you can get the norm 2 error or other errors, that will be very appreciated.
  *
  */
 private void computeSurvivalRate(double time[], boolean event[]){
   //I would like to change the event[] an array of boolean
   /* Here is an assumption: If time<=36 month, then the corresponding event
     indicate a Yes-product defect; If time>36, then set the time to 36 and
     set a No-no product defect to event;*/

//   FinalPeriod;
   int LengthOfArray = time.length;//I don't know how to find a length of an array
//   int TotalProduct;
   int TotalProduct = time.length;  //TG: suppose given are total
   m_iVectorSize = (int)(m_dSamplePeriod/m_dTimeUnit + 1);
   double[] TimeMark = new double[m_iVectorSize];
   double[] OurTime = new double[m_iVectorSize];
   double[] matrixout = new double[2];
   double[][] matrix1 = new double[m_iVectorSize][2];
   double[][] matrix2 = new double[2][2];
   double[][] matrix3 = new double[m_iVectorSize][2];
   double[] matrix4 = new double[2];
   double elma, elmb, elmc, elmd, mid1;
   double lambda, alfa;
   int i,j,k;
   TotalProduct=LengthOfArray;//In case that all of the products record
   for(i = 0; i < m_iVectorSize; i++){
     TimeMark[i]=0.0;
     OurTime[i]=i*m_dTimeUnit;
     }

   // organize the event into the vector
   int iMod;
   for (i = 0; i < time.length; i++) {
     iMod = (int)((time[i]+1)/m_dTimeUnit);
     if (iMod < m_iVectorSize & iMod > 0) {
//       if (event[i]) TimeMark[iMod] +=1;
       if (event[i]) TimeMark[iMod] +=1.0/TotalProduct;
     }
   }

   TimeMark[0] = 1;

   for (i = 1; i < m_iVectorSize && TimeMark[i] == 0; i++) {}
   int iNonZeroIdx = i;

   for(i = 1; i < m_iVectorSize; i++){
```

```
      TimeMark[i] =-1.0*TimeMark[i]+TimeMark[i-1];
      } //So this is the survival curve
    double[] dBUTimeMark = new double[m_iVectorSize];
    for(i = 0; i < m_iVectorSize; i++) dBUTimeMark[i] = TimeMark[i];
    for(i = iNonZeroIdx; i < m_iVectorSize; i++){
      TimeMark[i]= Math.log(Math.log(TimeMark[i])*(-1));
//      TimeMark[i]= Math.log(Math.log(TimeMark[i]));
      matrix3[i][0]=1;
      matrix3[i][1]=Math.log(OurTime[i]);
    }  //matrix3: A

    for(i=0; i<2;i=i+1){
      for(j=0; j<2; j=j+1){
        matrix2[i][j]=0;
        for(k = iNonZeroIdx; k < m_iVectorSize; k++){
          matrix2[i][j]=matrix3[k][i]*matrix3[k][j] + matrix2[i][j];
        }
      }
    }//matrix2: (A'*A)

    for(i=0; i<2;i=i+1){
      matrixout[i]=0.0;
      for(k=iNonZeroIdx;k<m_iVectorSize;k=k+1){
        matrixout[i] += matrix3[k][i]*TimeMark[k];
      }
    }//matrixout: vector A'*TimeMark

    //now is to find out the least square root by solving matrix equ.
    elma=matrix2[0][0];
    elmb=matrix2[0][1];
    elmc=matrix2[1][0];
    elmd=matrix2[1][1];
    mid1=elma*elmd-elmc*elmb;
    if (mid1 == 0) {
      m_dSurvivalRate = 0;
      return;
    }
    matrix2[0][0]=elmd/mid1;
    matrix2[0][1]=-1.0*elmb/mid1;
    matrix2[1][0]=-1.0*elmc/mid1;
    matrix2[1][1]=elma/mid1;
    //matrix2: inv(A'A)

    for(i = 0; i < 2; i++){
      matrix4[i]=0.0;
      for(k = 0; k < 2; k++){         //matrix4: inv(a'a)*(a'*timemark)
```

```
        matrix4[i]=matrix4[i]+matrix2[i][k]*matrixout[k];
        //output
      }
    } //matrix4: inv(A'A)*matrixout

    lambda=Math.exp(matrix4[0]);
    alfa=matrix4[1];
    m_dSurvivalRate=Math.exp(-1.0*lambda * Math.pow(D_PROJECTEDPERIOD-
iNonZeroIdx+1,alfa));
//    m_dSurvivalRate=Math.exp(-1.0*lambda *
Math.pow(D_PROJECTEDPERIOD,alfa));
//    m_dSurvivalRate=Math.exp(1.0*lambda *
Math.pow(D_PROJECTEDPERIOD,alfa));
    m_alpha = alfa; m_lambda = lambda;
    computeNorm2Error(dBUTimeMark, iNonZeroIdx);
  }

  /**
   * compute the norm2 error
   * parameter: devents, the timemark vector of survival rate at each time
   */
  private void computeNorm2Error(double[] devents, int inon) {
    double derr = 0.0;
    for(int i = inon;i < m_iVectorSize; i++) {
      derr += Math.pow(devents[i] - Math.exp(-1.0*m_lambda * Math.pow(i - inon +
1,m_alpha)),2);
    }
    m_Norm2Error = Math.sqrt(derr);
  }
}
```

---

```
/**
 * Title:        C4.5 with Survival Analysis class
 * Description:  Uses the projected survival rate as the class information in c4.5 to make
the tree.
 * @version 1.0
 */

import java.io.*;
import java.util.*;
import weka.core.*;
import weka.classifiers.*;


public class SurvivalAnalysisClassifier extends DistributionClassifier {
```

```
  /** whether it's root node. */
  private boolean m_bRoot = false;

  /** whether it's leaf. */
  private boolean m_bLeaf = true;

  /** Current node's children. */
  private SurvivalAnalysisClassifier[] m_Successors;

  /** Cases in current node. */
  private int m_iCases;

  /** Class percentage if node is leaf.
  TG: number of true Class value at each node*/
  private int m_iTrueCases = -1;

//  /** Class attribute of dataset. */
//  private Attribute m_ClassAttribute;

  /** marked attribute index. */
  private boolean[] m_bMarkedAttributes;

  /** Whether calculate quantity when compute SurvivalRatio
   * TG: I think ofcourse yes, because this is what survivalratio based on.
   */
  private static boolean B_QUANTITY = true;

  /** Attribute(s) belong to current node. */
  private Attribute[] m_Attributes = null;

//  /** Current node distribution. */
//  private double m_dDistribution = -1;

  /** Survival rate for current node. */
  private double m_dSurvivalRate = -1;

  /** norm2 error for survival analysis */
  private double m_dSurvivalError = -1;

  /** Optimizing Tolerance. */
  private static double D_TOLERANCE = 0.001;

  /** Whether this class is called alone.*/
  private boolean invokedStandalone = false;
```

```
/** minimum number to do the survival analysis */
private static int I_MINCASE = 4;

/** Whether use Survival Analysis. */
private boolean m_bUseSA = true;

/** sample period */
private static int I_SAMPLEPERIOD = -1;

/** projected period */
private static int I_PROJECTEDPERIOD = -1;

public SurvivalAnalysisClassifier(boolean bUseSA) {
  m_bUseSA = bUseSA;
}

public SurvivalAnalysisClassifier() {}

/** get minimum number of cases making tree */
public static int getMinCase(){
  return I_MINCASE;
}
/** get optimizing tolerance */
public static double getTolerance() {
  return D_TOLERANCE;
}

public static void setGainRatioTolerance(double dtol) {
  D_TOLERANCE = dtol;
}

public static void setSamplePeriod(int isampleperiod){
  I_SAMPLEPERIOD = isampleperiod;
}

public static void setProjectedPeriod(int iprojectedperiod) {
  I_PROJECTEDPERIOD = iprojectedperiod;
}

public static void setMinCases(int icase) {
  I_MINCASE = icase;
}
/**
 * Builds survival analysis decision tree classifier.
 *
 * @param data the training data
```

```
 * @param bAtts the already marked attributes array
 * @exception Exception if classifier can't be built successfully
 */
public void buildClassifier(Instances data, boolean[] bAtts) throws Exception{
  m_iCases = data.numInstances ();
  if (m_iCases == 0) {
    m_iTrueCases = 0;
    return;
  }
  m_iTrueCases = countTrueCases(data);
  if (m_iCases < I_MINCASE) {
    return;
  }
  if (m_bUseSA) {
    survivalAnalysis mSA = new survivalAnalysis();
    if (mSA.initModel(data,I_SAMPLEPERIOD)) {
      m_dSurvivalRate = mSA.getSurvivalRate ();
      m_dSurvivalError = mSA.getNorm2Error();
    }
    else {
      m_dSurvivalRate = mSA.getSurvivalRate ();
      return;
    }
  } else {
    m_dSurvivalRate = 1.0 - (double)m_iTrueCases/m_iCases;
  }
  boolean bAllMarkedAtt = true;
  m_bMarkedAttributes = new boolean[bAtts.length];
  for (int i = 0; i < bAtts.length; i ++) {
    m_bMarkedAttributes[i] = bAtts[i];
    bAllMarkedAtt &= bAtts[i];
  }
  if (bAllMarkedAtt) {
    return;
  }
  makeTree(data);
}

/**
 * Builds survival analysis decision tree classifier.
 *
 * @param data the training data
 * @exception Exception if classifier can't be built successfully
 * This method is called when there is not any node in the tree,
 * which means it will create the root of the tree.
 * node other than root node must call buildClassifier(Instances, boolean)
```

```
  */
  public void buildClassifier(Instances data) throws Exception {

    if (invokedStandalone) {
      throw new Exception("This Classifier can't run standalone.");
    }

    if (m_bUseSA) {
      data = modifyOutBoundedCases(data);
    }
    m_iCases = data.numInstances ();

    if (data.numClasses() > 2) {
      throw new Exception("Class has more than 2 different values.");
    }

    if (m_iCases < I_MINCASE) {
    // the cases is too few to make correct project,
    // suggest to stop
//      throw new Exception("Too few cases.");
      System.out.print(data.relationName() + ": Too few cases in this branch.");
      return;
    }
    else {
      m_bMarkedAttributes = new boolean[data.numAttributes () - 2];  // they should be
false.
      m_bRoot = true;
      m_iTrueCases = countTrueCases(data);
      data.setClassIndex (data.numAttributes() - 1);
      if (m_bUseSA) {
        survivalAnalysis mSA = new survivalAnalysis();
        if (mSA.initModel(data,I_SAMPLEPERIOD)) {
          m_dSurvivalRate = mSA.getSurvivalRate();
          m_dSurvivalError = mSA.getNorm2Error();
        }
        else {
          throw new Exception("Sorry, we can't make survival analysis on this data.");
        }
      } else {
        m_dSurvivalRate = 1.0 - (double)m_iTrueCases/m_iCases;
      }
    }
    makeTree(data);
  }

  /**
```

```
   * If there is zero GainRatio of any attribute, which hasn't marked in the
m_bMarkedAttributes
   *    but occured in this node,
   * then mark it.
   */
  private void optimizeSelectingAttribute(double[] survivalGR) {
    for (int i = 0; i < survivalGR.length; i++) {
      if (survivalGR[i] == 0) {
        m_bMarkedAttributes[i] = true;
      }
    }
  }

  /**
   * Modify the cases which
   */
  private Instances modifyOutBoundedCases(Instances data) {
    int itimeatt = data.numAttributes() - 2;
    int iclsatt = itimeatt + 1;
    for (int i = data.numInstances()-1; i >= 0; i--) {
      if (data.instance(i).value(itimeatt) > I_SAMPLEPERIOD) {
        // modify the data with time larger than sample period, to survive
        data.instance(i).setValue(itimeatt, I_SAMPLEPERIOD);
        data.instance(i).setClassValue(0);
      } else if(data.instance(i).value(itimeatt) <= 0){
        // delete the data with time less than or equal to zero.
        data.delete(i);
      } else if(data.instance(i).value(itimeatt) < I_SAMPLEPERIOD &&
data.instance(i).value(iclsatt) == 0) {
        data.delete(i);
      }
    }

    return data;
  }

  /**
   * Method building tree based on survival analysis.
   *
   * @param data the training data
   * @exception Exception if decision tree can't be built successfully
   */
  private void makeTree(Instances data) throws Exception {
    // Check if no instances have reached this node.
    if (data.numInstances() == 0) {
      return;
```

```
    }

    // Compute attribute with maximum survival rate.
    double[] survivalGainRatio = new double[m_bMarkedAttributes.length];
    for (int i = 0; i < m_bMarkedAttributes.length; i ++) {
      if (!m_bMarkedAttributes[i]) {
        if (m_bUseSA){
          survivalGainRatio[i] = computeSurvivalGainRatio(data, data.attribute(i));
        } else {
          survivalGainRatio[i] = computeGainRatio(data, data.attribute(i));
        }
      }
    }
//    optimizeSurvivalDistribution(survivalGainRatio, D_TOLERANCE);
//    optimizeSelectingAttribute(survivalGainRatio);
    // here we should select which attributes in current nodes,
    m_Attributes = new Attribute[1];
    m_Attributes[0] = data.attribute(Utils.maxIndex(survivalGainRatio));
//    if (!m_bUseSA) {
//      m_dSurvivalRate = survivalGainRatio[Utils.maxIndex(survivalGainRatio)];
//    }
    // Make leaf if survival gain ratio is zero.
    // Otherwise create successors.
    if (survivalGainRatio[m_Attributes[0].index()] < D_TOLERANCE) {
      m_Attributes = null;
    } else {
      m_bLeaf = false;
      m_bMarkedAttributes[m_Attributes[0].index()] = true;
      Instances[] splitData = splitData(data, m_Attributes[0]);
      m_Successors = new SurvivalAnalysisClassifier[m_Attributes[0].numValues()];
      for (int j = 0; j < m_Attributes[0].numValues(); j++) {
            m_Successors[j] = new SurvivalAnalysisClassifier(m_bUseSA);
            m_Successors[j].buildClassifier(splitData[j],m_bMarkedAttributes);
      }
    }
  }

  /**
   * Computes survival gain ratio.
   */
  private double computeSurvivalGainRatio(Instances data, Attribute att) throws
Exception {
    Instances[] splitData = splitData(data,att);
    double survivalRateGain = computeSurvivalRateGain(splitData);
    double dsplitInfo = computeSurvivalSplitInfo(splitData);
    if (dsplitInfo !=0) {
```

```
      return survivalRateGain/dsplitInfo;
    }
    else {
      return (double)0;
    }
  }

  /**
   * Computes survival rate gain.
   */
  private double computeSurvivalRateGain(Instances[] splitData) throws Exception{

    double survivalRateGain;
    if (m_dSurvivalRate != -1) {
      survivalRateGain = computeSurvivalInfo(m_dSurvivalRate);
    }
    else {
      throw new Exception("computeSurvivalRateGain(): No survival rate for current
node.");
    }
    double[] survivalRates = new double[splitData.length];
    double[] survivalErrors = new double[splitData.length];
    for (int j = 0; j < splitData.length; j++) {
      // tg: here we may consider don't use survival analysis when the instances are few.
      // if (splitData[j].numInstances() > 10) {
      if (splitData[j].numInstances() > 0) {
      // tg: the following function are information gain combining survival rate calculation
        survivalAnalysis mSA = new survivalAnalysis();
        if (mSA.initModel(splitData[j],I_SAMPLEPERIOD)) {
          survivalRates[j] = mSA.getSurvivalRate();
          survivalErrors[j] = mSA.getNorm2Error();
          survivalRateGain -= ((double) splitData[j].numInstances() /
                      (double) m_iCases) * computeSurvivalInfo(survivalRates[j]);
        }
        else { //TG: what should we do if can't initModel()?
          survivalRates[j] = 0;
        }
      // tg: to check if they are similiar
//        System.out.println("f_rate[" + j + "]: " + survivalRate[j]);
      }
    }
    // tg: to check survivalRateGain with mSurvivalGain
//      System.out.println("mFRate: " + m_dSurvivalRate + " fRGain: " +
survivalRateGain);
    return survivalRateGain;
  }
```

```
/**
 * Optimize survival rate
 * paras: ddata, the survival rate array
 *        dvalue, the tolerance value which used to
 * now, doing nothing
 */

private void optimizeSurvivalDistribution(double[] ddata, double dvalue) {

}

/**
 * Classifies a given test instance using the decision tree.
 *
 * @param instance the instance to be classified
 * @return the classification
 */
public double classifyInstance(Instance instance) {

  if (m_Attributes == null) {
    return m_dSurvivalRate;
  }
  else {
    return m_Successors[(int) instance.value(m_Attributes[0])].
           classifyInstance(instance);
  }
}


/**
 * Computes class distribution for instance using decision tree.
 *
 * @param instance the instance for which distribution is to be computed
 * @return the class distribution for the given instance
 */
public double[] distributionForInstance(Instance instance) {

  if (m_Attributes == null) {
    double dtmp[] = new double[2];
    dtmp[0] = (double)1-m_dSurvivalRate;
    dtmp[1] = m_dSurvivalRate;
    return dtmp;
  }
  else {
    return m_Successors[(int) instance.value(m_Attributes[0])].
```

```
        distributionForInstance(instance);
    }
  }

  /**
   * Prints the decision tree using the private toString method from below.
   *
   * @return a textual description of the classifier
   */
  public String toString() {

    if ((m_dSurvivalRate < 0) && (m_Successors == null)) {
      return "\nSurvivalAnalysisClassifier: No model built yet.";
    }
    StringBuffer sbuf = new StringBuffer();
    sbuf.append("SurvivalAnalysisClassifier\n");
    if (m_bUseSA) {
      sbuf.append("Using Survival Analysis");
      if (I_SAMPLEPERIOD < 0)
        sbuf.append("(default sample period)\n");
      else
        sbuf.append("(sample period:" + I_SAMPLEPERIOD + ")\n");
    } else {
      sbuf.append("Not using survival analysis: \n");
    }
    return  sbuf.toString() + toString(0);
  }

  /**
   * Outputs a tree at a certain level.
   *
   * @param level the level at which the tree is to be printed
   */
  private String toString(int level) {
    StringBuffer text = new StringBuffer();
    if (m_Attributes == null) {
//      text.append(": "+ m_dSurvivalRate);
      text.append("(" + m_iTrueCases + ":" + m_iCases + ")Pro:" + m_dSurvivalRate);
      if (m_bUseSA) text.append("(Ori:" + (1.0-(double)m_iTrueCases/m_iCases) + ")");
      if (m_bUseSA) text.append("Err:" + m_dSurvivalError);
    }
    else {
      text.append("\n");
      for (int i = 0; i < level; i++) {
        text.append("|  ");
      }
```

```
//      text.append("fRate: " + m_dSurvivalRate);
//      text.append(": "+ m_dSurvivalRate);
      text.append("(" + m_iTrueCases + ":" + m_iCases + ")Pro:" + m_dSurvivalRate);
      if (m_bUseSA) text.append("(Ori:" + (1.0-(double)m_iTrueCases/m_iCases) + ")");
      if (m_bUseSA) text.append("Err:" + m_dSurvivalError);
      for (int j = 0; j < m_Attributes[0].numValues(); j++) {
        text.append("\n");
        for (int i = 0; i < level; i++) {
          text.append("|  ");
            }
        text.append(m_Attributes[0].name() + " = " + m_Attributes[0].value(j));
        text.append(m_Successors[j].toString(level + 1));
      }
    }
    return text.toString();
  }

  /**
   * Counts the number of true class value
   */
  public int countTrueCases(Instances data) throws Exception {
    int [] classCounts = new int[2];
    Enumeration instEnum = data.enumerateInstances();
    while (instEnum.hasMoreElements()) {
      Instance inst = (Instance) instEnum.nextElement();
      classCounts[(int) inst.classValue()]++;
    }
    return classCounts[1];
  }

  /**
   * Splits a dataset according to the values of a nominal attribute.
   *
   * @param data the data which is to be split
   * @param att the attribute to be used for splitting
   * @return the sets of instances produced by the split
   */
  private Instances[] splitData(Instances data, Attribute att) {

    Instances[] splitData = new Instances[att.numValues()];
    for (int j = 0; j < att.numValues(); j++) {
      splitData[j] = new Instances(data, data.numInstances());
    }
    Enumeration instEnum = data.enumerateInstances();
    while (instEnum.hasMoreElements()) {
      Instance inst = (Instance) instEnum.nextElement();
```

```
    splitData[(int) inst.value(att)].add(inst);
  }
  return splitData;
}

/**
 * compute split info
 * param: splitIns the Instance groups
 */
private double computeSurvivalSplitInfo(Instances[] splitIns) {
  double dsplitinfo = 0;
  int iins = 0;
  // calculate split info based on quantity
  if (B_QUANTITY) {
    for (int j = 0; j < splitIns.length; j++) {
      iins = splitIns[j].numInstances();
      if (iins> 0) {
        dsplitinfo -= (iins * Utils.log2(iins));
      }
    }
    dsplitinfo /= (double) m_iCases;
    dsplitinfo += Utils.log2(m_iCases);
    return dsplitinfo;
  }
  else {
    dsplitinfo = - Utils.log2((double)1.0/splitIns.length);
    return dsplitinfo;
  }
}

/**
 * Compute Info
 * Compute the projected entropy by using analysis method
 * @param dd the survival rate
 */
private double computeSurvivalInfo(double dd) throws Exception{
  if (0 <= dd && dd <= 1) {
    return -(1-dd)*Utils.log2(1-dd)-dd*Utils.log2(dd);
  }
  throw new Exception("computeSurvivalInfo():Please check, it's not in [0,1]. ");
}
/**
 * Computes information gain for an attribute.
 *
 * @param data the data for which info gain is to be computed
 * @param att the attribute
```

```
 * @return the information gain for the given attribute and data
 */
private double computeInfoGain(Instances data, Attribute att)
  throws Exception {
  Instances[] splitData = splitData(data,att);
  double infoGain = computeEntropy(data);
  for (int j = 0; j < splitData.length; j++) {
    if (splitData[j].numInstances() > 0) {
		infoGain -= ((double) splitData[j].numInstances() /
			(double) data.numInstances ()) *
		  computeEntropy(splitData[j]);
    }
  }
  return infoGain;
}

/**
 * Computes information gain ratio for an attribute.
 *
 * @param data the data for which info gain is to be computed
 * @param att the attribute
 * @return the information gain ration for the given attribute and data
 */
private double computeGainRatio(Instances data, Attribute att) throws Exception {
  double infoGain = computeInfoGain(data, att);
  double splitInfo = computeSplitInfo(data, att);
  if (splitInfo !=0) {
    return infoGain/splitInfo;
  } else {
    return 0;
  }
}

/**
 * Computes information gain for an attribute.
 *
 * @param data the data for which info gain is to be computed
 * @param att the attribute
 * @return the information gain for the given attribute and data
 */
private double computeSplitInfo(Instances data, Attribute att) throws Exception {
  double splitInfo = 0;
  Instances[] splitData = splitData(data,att);
  for (int j = 0; j < splitData.length; j++) {
    if (splitData[j].numInstances() > 0) {
		splitInfo -= (((double) splitData[j].numInstances() /
```

```
                        (double) m_iCases) * Utils.log2((double) splitData[j].numInstances() /
                        (double) m_iCases));
      }
    }
    return splitInfo;

  }

  /**
   * Computes the entropy of a dataset.
   *
   * @param data the data for which entropy is to be computed
   * @return the entropy of the data's class distribution
   */
  private double computeEntropy(Instances data) throws Exception {

    double [] classCounts = new double[data.numClasses()];
    Enumeration instEnum = data.enumerateInstances();
    while (instEnum.hasMoreElements()) {
      Instance inst = (Instance) instEnum.nextElement();
      classCounts[(int) inst.classValue()]++;
    }
    double entropy = 0;
    for (int j = 0; j < data.numClasses(); j++) {
      if (classCounts[j] > 0) {
        entropy -= classCounts[j] * Utils.log2(classCounts[j]);
      }
    }
    entropy /= (double) data.numInstances();
    return entropy + Utils.log2(data.numInstances());
  }
}
```