

Classification Trees in APL: *Implementation and Application*

Alexander Skomorokhov and Vladimir Kutinsky

Institute of Nuclear Power Engineering

P.O. Box 5061, Obninsk-5

Kaluga Region, 249020, Russia

askom@obninsk.com, kutinskyv@obninsk.com

Abstract

This paper considers the problem of classification tree based data analysis. Among the topics discussed in the paper are: growing a classification tree using CART style exhaustive search for splits, selecting the right size for the tree using minimal cost-complexity cross-validation pruning, and examples of the application of classification trees.

The algorithms are implemented in Dyalog APL.

Application example is based on data from vibration monitoring equipment installed on a Nuclear Power Plant in Novovoronezh, Russia and includes classification of vibration spectra of steam generators or coolant pumps and classification of vibration spectra of steam generators of different coolant loops.

Keywords: Data Mining, Classification Trees, Pruning, Cross-validation, Vibration Monitoring.

Introduction

Tree-based models provide a number of benefits:

- A natural approach to join processing of both categorical and continuous variables;
- Important information is revealed as the decision rules are constructed;
- Easy interpretation of the results;
- Large datasets (both the number of cases and the number of predictors) can be dealt with.

What is a *Classification Tree*? A Classification Tree is a hierarchical set of classification rules, which, in case of numeric predictors only, are of the form:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
APL01, 06/01, New Haven, CT USA
©2001 ACM 1-58113-419-3 / 01/0006 \$5.00

if $x_1 \leq b_1$ and $x_3 > b_2$

then y is most likely to be c_2

where b_i are some thresholds.

Classification Trees can work with both numeric and factor prediction variables. In this case the classification rules are of the form:

if $x_1 \leq b_1$ and $x_3 \in \{S, D\}$

then y is most likely to be c_2

Another feature of Classification Trees is their hierarchical nature, i.e., the ability of classification trees to examine the effects of predictor variables one at a time, rather than just all at once, as it is in case of discriminant analysis.

Classification Trees are displayed graphically, making their interpretation much easier than a strictly numerical presentation.

In sum, Classification Trees are very attractive because they provide a simple and clear methodology for data analysis.

Growing the Tree

Input data

Let us consider a number of objects $A = \{a_1, a_2, \dots, a_N\}$, described by attributes $\{x_1, x_2, \dots, x_m, y\}$. The x_j are called classification or predictor variables and y is a response variable. Predictors can be both numeric and factor variables. Let us consider also a number of classes $C = \{c_1, c_2, \dots, c_k\}$. Response variable y indicates the class of each object in A .

For illustration purposes we will use an artificial data set represented in Table 1.

Table 1. Example data set

N	X1	X2	X3	X4	Y
1	5.1	0.5	A	3	1
2	5.1	4.5	B	3	1
3	4.4	4	C	3	1
4	5.4	3.5	C	4.5	1
5	5	3.5	D	4	1
6	7.7	5	A	1.5	2
7	6.1	3.5	B	3.5	2
8	6.8	0.5	B	2.5	2
9	5.6	4	B	3.5	2
10	6.7	5	A	1.5	2
11	6.4	1	D	0.5	3
12	6	4.5	C	2.5	3
13	6.7	2	C	3	3
14	5.5	4.5	D	5	3
15	5.5	0.5	D	0.5	3

There are a total of 15 objects (Table 1 rows), described with 4 attributes. Each object belongs to one of three classes, indicated by the variable *Y*. Predictors *X1*, *X2* and *X4* are continuous variables. Predictor *X3* is a categorical variable with possible levels taken from set {A,B,C,D}.

In APL we represent continuous and ordered variables as simple numeric vectors. For categorical predictors the natural representation is a nested character vector of value labels. The same is true for a response variable of class labels. But to simplify the code we enumerate the classes and use a simple numeric vector of class numbers instead. The data set, shown in Table 1, is a nested matrix called *data*:

```

pdata
15 5
  =data
  2
  disp data[1 2:]
  ↓
  | 5.1 0.5 |A| 3 1 | |
  |         | |   |   |
  | 5.1 4.5 |B| 3 1 |
  |         | |   |   |
  ↓
  
```

Note that scalars from set {A,B,C,D} are presented as one element character vectors, to allow long names if necessary.

Tree-based model

A Classification Tree constructed for this data is shown in Figure 1.

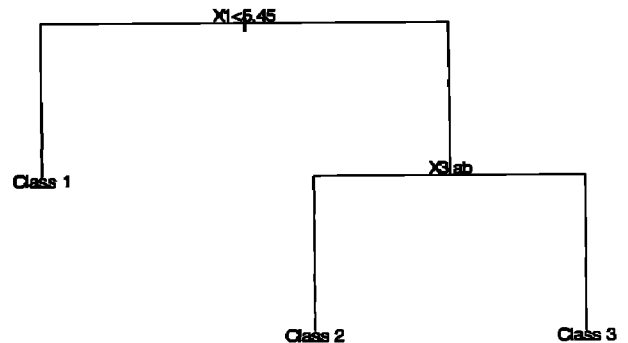


Figure 1: Classification Tree for artificial data set

This figure and other tree plots in this paper were created using the R Statistical system [2]. R is “GNU S” – a language and environment for statistical computing and graphics. This system is freely available for major platforms, including Windows and Linux operating systems. R is similar to the award-winning S system [8], which was developed at Bell Laboratories by John Chambers et al. It provides a wide variety of statistical and graphical techniques (linear and nonlinear modeling, statistical tests, time series analysis, classification, clustering, etc.) and a rich set of functions to create and explore tree-based models. We used R and S systems to validate the APL software developed in this paper.

Possible splits

Growing the tree proceeds sequentially. As the structure of trees is hierarchical, splits are selected one at a time, starting with the split at the root node and continuing with splits of the resulting child nodes, until splitting stops. Those child nodes which have not been split become terminal nodes.

The different methods of split selection are discussed in the paper [1].

Let us start from a simple example of selecting the best split of a single continuous variable.

```

(x c)+(18)(4/1 2)
  >x c
  1 2 3 4 5 6 7 8
  1 1 1 1 2 2 2 2
  
```

Variable *c* represents the class of several objects, and variable *x* is an attribute to be used for classification. It is clear that the best partition is given by

$x \leq 4.5$. In our case, the variable x has $L=8$ levels and there are a total of $L-1$ possible splits. To select the best one we need a criterion to compare the different splits. That kind of criterion is based on the misclassification rate. The best is a split which produces pure nodes. A pure node is a node that contains objects of one class only.

For categorical predictors the consideration is a bit more complex. We have discussed this topic in another paper [4]. Let us consider an example of categorical variable x with a set of values $\{A,B,C,D\}$. The split condition may be $x \in \{A,B\}$ (" x belongs to a subset $\{A,B\}$ of possible values"). The left child node is the set of cases for which this condition is true. The right child node condition may be considered either as the left node condition negation or as $x \in \{D,C\}$, because set $\{D,C\}$ is a complement of set $\{A,B\}$.

To try different splits with a categorical predictor we have to create a set of possible subsets of its values. The number of all possible subsets is equal to 2^L , where L is the number of categorical variable levels. We do not need an empty subset and a subset equal to a whole set of levels. We also do not need to keep both subsets, if one of a pair is a compliment of another. For instance, splits $x \in \{A\}$ and $x \in \{B,C,D\}$ produce the same child nodes, but in different left to right order.

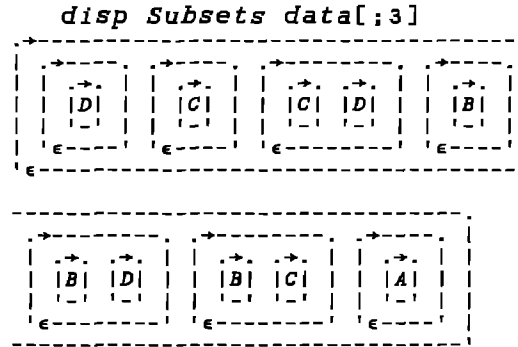
The total number of useful subsets is equal to $2^{L-1} - 1$. These subsets may be generated using function *Subsets*:

```
[0] z←Subsets x;i;n
[1] x←Unique x
[2] i←(nρ2)⍒12*n+ρx
[3] i←(-(+i)∈0,n,n-1)/i
[4] i←c[1]i
[5] z←''
[6] L:z+z,c+i
[7] i←i-z,-z
[8] →(0<ρi)/L
[9] z+z/'c x
```

A well-known APL idiom generates a list of unique values of a vector:

```
[0] u←Unique x
[1] u←((x∧x)=ρx)/x
```

The function *Subsets* takes a nested character vector of a categorical predictor as its right argument and returns a nested vector of possible subsets. For instance:



Deviance

Let us count the number of objects of each class at some node N_i and denote it as n_{ij} , for a class j . The total number of objects for this node is $n_i = \sum_j n_{ij}$.

Now we may estimate probabilities p_{ij} of each class at this node as $p_{ij} = \frac{n_{ij}}{n_i}$. The purity of a node i may be

then characterized by entropy:

$$H_i = -2 \sum_j p_{ij} \log_2 p_{ij}$$

which equals zero for a pure node (note that $0 \times \log_2 0$ gives 0) and takes some positive value for a mixture of classes. That means that we have to select a split that minimizes entropy. It may be illustrated with entropy values for different probabilities of 2 classes at a given node:

```
p+(.5 .5)(.7 .3)(.9 .1)(.999 .001)
  (-2*x+/w*2*o w)**p
2 1.76258179 0.9379911872 0.02281551547
```

The most pure distribution occurs when $p_{i1}=0.999$ and minimum entropy when $p_{i2}=0.001$.

In the construction of tree models, another criterion, known as *deviance* [3], is used more often. For a given node i , deviance is given by:

$$D_i = -2 \sum_j n_{ij} \log p_{ij}$$

Deviance has a value of zero value for a pure node. It is very close to entropy but uses multiplier n_{ij} instead of p_{ij} . The meaning of this difference will be clear a bit later.

Now consider a node i splitting into nodes j and k . If the corresponding values of deviance are denoted as D_i , D_j and D_k then the reduction in deviance is $D_i - D_j - D_k$. The goal is to select a split, which maximizes the reduction in deviance.

Deviance is calculated with the following APL function:

```
[0] d←Deviance class;n;p
[1] n←+/ (Unique class)∘.=class
[2] p←n÷ρclass
[3] d←~2×+/n×2∘p
```

The right argument is a vector of class indices and estimates the number of objects of each class (vector *n*) and corresponding probabilities (vector *p*).

Let us now return to our previous example:

```
      c
1 1 1 1 2 2 2 2

+d←→Deviance""(1~1+ρc){(α+w)(α+w)}""c
0          13.7931939
0          11.01955001
0          7.219280949
0          0
7.219280949 0
11.01955001 0
13.7931939 0
```

The APL expression above calculates the deviance of child nodes for each possible split from $A=\{x_1\}$, $B=\{x_2-x_3\}$ to $A=\{x_1-x_7\}$, $B=\{x_3\}$, where A and B stand for child node subsets after a split. The first 4 splits give pure left node (class 1) and the mix of classes in the right node. The last 4 splits perform vice versa. Only the fifth split (equivalent to split condition $x \leq 4.5$) creates two pure child nodes. The reduction in deviance for each possible split is given by the APL expression:

```
(Deviance c)←+/d
2.206806096 4.980449991 8.780719051 16
8.780719051 4.980449991 2.206806096
```

The maximum value of 16 is achieved with the optimal $x \leq 4.5$ split.

Split selection

In order to find the best split, it is necessary to review all possible splits for each predictor variable at each node and choose the one that produces the largest improvement in *goodness of classification*. So, as the best split we choose the one that leads to the minimal value of deviance (or equivalently, the largest reduction in it).

The APL function *Partition* returns the best split of a given predictor *x* (right argument) and a given vector of class numbers *class* (left argument). The result is an optimal threshold and a corresponding value of deviance for continuous predictors, or a subset of levels and deviance for categorical predictors.

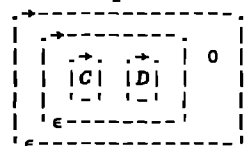
```
[0] z←class Partition x;y:por;p
[1] z←10
[2] por←Unique x
[3] :If 1=ρx
[4]   por←por[⊖por]
[5]   ⚡(1<ρpor)!'por+.5×2+/por'
[6] :Else
[7]   por←Subsets por
[8] :EndIf
[9] :For p :In por
[10]  y←Deviance(x Compare p)/class
[11]  y←y+Deviance(-x Compare p)/class
[12]  z←z,y
[13] :EndFor
[14] p←z\|/z
[15] z←por[p],z[p]
```

To unify processing of continuous and categorical predictors we use a simple utility function, *compare*:

```
[0] l←x Compare p
[1] :If 1=ρx
[2]   l←x≤p
[3] :Else
[4]   l←x∈p
[5] :EndIf
```

The type of a predictor (left argument) is distinguished using its depth and the right argument is either a threshold or a subset of values. The following examples illustrate the use of *Partition*:

```
c←1 1 1 1 2 2 2 2
x←1 2 3 4 5 6 7 8
c Partition x
4.5 0
s←,'ABAACDDC'
disp c Partition s
```



In the first example the best threshold is equal to 4.5 and in the second one (categorical predictor) the best subset is {D,C}. Both splits lead to a perfect classification and therefore have deviance equal to 0.

The following function *Split* finds the best split among all predictors and all splits for each predictor.

```
[0] z←class Split data;x;i
[1] :If 1=ρUnique class
[2]   z←4ρ0
[3] :Else
[4]   z←(cclass)Partition""c[1]data
[5]   i←z[;2]\|/z[;2]
[6]   z←i,z[i;1],2ρ0
[7] :EndIf
[8] i←+/ (Unique class)∘.=class
[9] z←z,i+(i\|/i)÷Unique class
[10] z←z,ρclass-i
```

It takes the whole data set (matrix) as the right argument and the classification vector as the left argument. If the current node is a pure one (a check at line [1]), then the first 4 items of a result are set to 0 to indicate a terminal node (leaf). Otherwise, we find the best split for each predictor (line [4]) and define which predictor has to be used to perform a split (lines [5-6]). Finally, we add to the result a class label for the current node (lines [8-9]) and a misclassification rate (line [10]). A class label is assigned to a class with the highest probability (Bayes decision rule) or, equivalently, to a class mostly represented at this node.

Recursive partitioning

Now we are ready for the job of growing a classification tree. This construction is known as *recursive partitioning*. A recursive function *GrowTree* performs this task:

```
[0] tree←(i)GrowTree data;cl;i;j; left;
      right;por;sp
[1] ⍠(0=⊂NC'i')/'i+1'
[2] cl←data[;2÷pdata]
[3] sp←cl Split ~1+[2]data
[4] tree←0 7ρ0
[5] :If sp[1]=0
[6]   tree←tree,[1]i,sp
[7] :Else
[8]   sp[3 4]←0 1+2×i
[9]   tree←tree,[1]i,sp
[10]  (j por)←sp[1 2]
[11]  left←(data[;j]Compare por)÷data
[12]  right←(-data[;j]Compare por)÷data
[13]  tree←tree,[1](2×i)GrowTree left
[14]  tree←tree,[1](1+2×i)GrowTree right
[15]:EndIf
```

The right argument is a matrix of data. An optional left argument is used in recursive calls and gives a current node a number. It takes value 1 for a root node (line [1]), when the function is called for the first time. The result is a classification tree represented as a 7-column matrix. The meaning of the result columns will be explained a bit later. In line [2] we take a response variable (class labels) given in the last column of data matrix. Line [3] finds the best split as it was discussed above. If the current node is a terminal one (check in line [5]) then its data is directly concatenated to a matrix of the tree (line [6]). Otherwise, the numbers for child nodes are assigned in line [8]. The root node takes the number one. Each subsequent level of a binary tree has twice as many possible nodes (2, 4, 16, 32,...), which are enumerated from left to right. In line [10] from a result of function *Split* we take the best predictor index *j* and the threshold value *por* (or a subset of values for a categorical predictor). In lines [11-12] we perform an actual split creating data

subsets for the left and right child nodes. And finally, the function is called recursively for each subset of split data (lines [13-14]).

Let us now build a tree-based model for the artificial data set described above.

```
pt←GrowTree data
5 7
  =t
3
  disp vt
.-----
|1 1      5.45  2 3 1 10|
|2 0      0      0 0 1 0|
|3 3      D C   6 7 2 5|
|6 0      0      0 0 3 0|
|7 0      0      0 0 2 0|
'-----'
```

Each row of the result corresponds to a node. We constructed a tree of 5 nodes. The depth of the result is 3 because it contains nested vectors of categorical predictor subsets. If only continuous predictors are used, then the result is a simple numeric matrix. To display the output we call a utility function, *disp*, to format the matrix of the resulting tree.

Let us discuss the structure of a result in more detail. The first three columns are the number of node, the number of a predictor to be used for the split and a threshold value (or levels subset) to apply to that predictor. A zero value for a predictor index (and for a threshold) indicates a terminal node. For instance, in the root (row 1) we should split data on predictor X_1 and use threshold value 5.45. Columns 4 and 5 give us the numbers of the child nodes. At node 3 (row 3) we perform split on condition $X_j \in \{D,C\}$ and the child nodes are 6 and 7 (rows 4 and 5), which are both terminal nodes. The last two columns (6 and 7) are the class label and the number of classification errors for each node. Now we can draw the tree. The result is shown in Figure 1.

Using the Tree

The main use of a tree-based model is prediction or classification of new data points. APL function *Predict* performs this task:

```
[0] z←data Predict tree;por;i;j;k
[1] ⍠(1=ρpdata)/'data+,[.5]data'
[2] z←10
[3] :For k :In 1+pdata
[4]   i←1
[5]   :While -tree[i;2]=0
[6]     (j por)←tree[i;2 3]
[7]     :If 1=⊂data[;j]
[8]       j←data[k;j]>por
[9]     :Else
```

```
[10]      j←-data[k;j]εpor
[11]      :EndIf
[12]      i←tree[i;4+j]
[13]      i←tree[i;1]i
[14]      :EndWhile
[15]      z←z,tree[i;6]
[16]      :EndFor
```

The right argument is a tree object built using the function *GrowTree*. The function starts from a root of the tree and selects a branch (child node) in lines [8] (continuous predictor) or in line [10] (categorical predictor). Then, it goes to a proper child node (lines [12-13]) and repeats the process. It stops when a terminal node is reached (line [5]) and, finally, the class label of this node is assigned to a result in line [15]. The following examples illustrate the use of *Predict*:

```
1      5.1 0.5 (,'A') 3 Predict t
1      5.5 0.5 (,'D') 0.5 Predict t
3
      data[;5]
1 1 1 1 1 2 2 2 2 2 3 3 3 3
      data Predict t
1 1 1 1 1 2 2 2 2 2 3 3 3 3
      data[;5]∧.=data Predict t
1
```

Another task of interest is the estimation of a misclassification rate for a given tree-based model. The following simple function performs this task:

```
[0] n←NErrors tree
[1] tree←(tree[;2]=0)+tree
[2] n←+/tree[;7]

      NErrors t
0
```

We may be interested in extracting a rule or a set of rules from a tree-based model. For a given terminal node we have an associated class label and a set of conditions to reach this node from the root of a tree. This may be expressed as a rule:

Class C IF *Condition 1* AND...AND *Condition N*

Functions *GetRule* and *GetIF* allow us to extract rules:

```
[0] r←t GetRule n
[1] r←φt GetIF n
[2] ((pr)→r)←-3+(pr)→r
[3] r←'Class ',(⍎t[t[;1]n;6]),' IF ',r

[0] r←t GetIF n;i;x;v;0
[1] r←''
[2] i←(=[1]t[;4 5])i~n
[3] →(1=t[t[;1]n;1])/0
[4] (x v)+t[l/i;2 3]
[5] o←(1+2=≡v)→'≤>' '≡'
[6] x←('V',⍎x),o[1+>/i],⍎v
[7] r←r,←x,' AND'
[8] r←r,t GetIF t[l/i;1]
```

The left argument of the function *GetRule* is a tree object and the right argument is a number of a terminal node associated with the rule to be extracted. An illustration is shown below:

```
      t GetRule 6
Class 3 IF V1>5.45 AND V3= D C
      t GetRule 7
Class 2 IF V1>5.45 AND V3≠ D C
```

In order to document a classification tree model, it is useful to format a matrix of a tree. APL function *Print* does this job:

```
[0] f←Print t;i
[1] i←[2⊙1+t[;1]
[2] f←⍎t[;1]
[3] t[;1]←f,⍎''
[4] f←⍎t,' *'[1+t[;2]=0]
[5] f←(-2×i)φ((2→pf)+2×[ /i])+[2]f
```

Our format is similar to the style used in S-PLUS and R. This means that we show a depth of each node and mark terminal nodes with asterisks:

```
      Print t
1) 1      5.45  2 3 1 10
2) 0      0      0 0 1 0 *
3) 3      D C    6 7 2 5
6) 0      0      0 0 3 0 *
7) 0      0      0 0 2 0 *
```

Cost-Complexity Analysis Overfitting

As it is well known, real data are always “noisy” and the distributions for the classes overlap. While we grow a tree using a training set we may adapt a model too well to the particular training set we have. But this model may work badly when we use it for prediction on new data points. This problem is known as *overfitting*. Let us illustrate this problem using our artificial data set. It has four predictors, but two of them (X_2 and X_4) are useless (they were generated at random). The only two predictors X_1 and X_3 perform perfect classification in accordance with the tree built above. It may be expressed with a set of 3 simple rules:

```
Class=1 IF ( $X_1 \leq 5.45$ )
Class=2 IF ( $X_1 > 5.45$ ) AND ( $X_3 \in \{A, B\}$ )
Class=3 IF ( $X_1 > 5.45$ ) AND ( $X_3 \in \{D, C\}$ )
```

Let us now bring in errors or noise to our data set:

```
data2←data
data2[1;1]←6.1
data2[15;3]←←,'A'
```

The first data point belongs to class 1. After we changed the value of predictor X_1 to 6.1 the first rule

does not work and this point will be misclassified. For point 15 (Class 3) we changed X_2 value from 'D' to 'A'. Rule 3 does not work now and this point will be misclassified to class 2. The following calculation shows prediction using the "old" tree, but the new data set indicates that two errors were created:

```

data2[;5]
1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
data2 Predict t
2 1 1 1 1 2 2 2 2 2 3 3 3 3 2
data2[;5]*data2 Predict t
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
    
```

Let us now build the new tree using the new data:

```

t2←GrowTree data2
NErrors t2
0
Print t2
1) 1 5.45 2 3 1 10
2) 0 0 0 0 1 0 *
3) 3 D C 6 7 2 6
6) 0 0 0 0 3 0 *
7) 1 5.55 14 15 2 2
14) 0 0 0 0 3 0 *
15) 2 2 30 31 2 1
30) 1 6.45 60 61 1 1
60) 0 0 0 0 1 0 *
61) 0 0 0 0 2 0 *
31) 0 0 0 0 2 0 *
    
```

We may see that the number of errors is equal to 0 again, but at the cost of a doubly complex tree (6 terminal nodes in comparison to 3 terminal nodes of an "old" tree). Even more important is the fact that the irrelevant noisy predictor X_2 is now involved in the classification (node 15). Let us now examine an entirely new data point, which fits to rule 2) and therefore has to be predicted as a class 2 instance:

```

6.4 1.5 ('A') 3.62 Predict t
2
6.4 1.5 ('A') 3.62 Predict t2
1
6.4 2.5 ('A') 3.62 Predict t2
2
    
```

That is true if we use the initial tree model t . But this point is misclassified with the new model $t2$, because the noisy predictor X_2 has a value of 1.5 and the irrelevant rule at node 15 is fired. As predictor X_2 is a noisy one it may take any value. The third expression of the example above shows that the classification is changed if $X_2=2.5$. It really looks like a random game and we have to conclude that the tree model $t2$ is overfitted.

Cutting the tree

At this stage, a question arises, "How do we cope with the problem of overfitting?" One way is to stop a tree growing before it reaches its maximal size. The

usual stop criteria are a small number of points at some node or a small reduction in deviance. Another approach is to remove unnecessary nodes after a tree of maximal size has been built. We may limit the number of rules per class or the number of predictors used for classification. Some insights of that kind may come from previous research, diagnostic information from other analyses, or even intuition.

In the next section we consider using special formal procedures for selecting the "right-sized" tree, known as *cost-complexity pruning* and *cross-validation*. Here we discuss a method and a function to reduce a tree size to a given number of nodes.

The raw result of the *GrowTree* function is a maximal classification tree. Now we want to get a smaller subtree with fewer nodes from the maximal tree. The problem is that there can be many different trees of the same size. These trees of the same size may differ in the proportion of misclassified cases. The goal is to select the best subtree of a given size.

The function *Cut* takes the tree of maximal size as its right argument and searches for the tree of size n (left argument) such that its cost (misclassification rate) is the lowest among all trees of the same size.

```

[0] z←n Cut tree;cnt;memo;tmp;i;j;k
    ;row;e
[1] n←n[+/tree[;2]=0
[2] :If n=1
[3] z←tree[;1;]
[4] z[1;2 3 4 5]←0
[5] :Return
[6] :EndIf
[7] z←0 7ρ0
[8] cnt←0 ρ memo←0 3ρ0 ρ k←1
[9] :While cnt≤n
[10] z←z,[1]tree[k;]
[11] :For i :In tree[k;4 5]
[12] j←tree[;1]i
[13] row←tree[j;]
[14] :If row[2]=0
[15] z←z,[1]row
[16] cnt←cnt+1
[17] :Else
[18] e←tree[;1]row[4 5]
[19] e←row[7]-+/tree[e;7]
[20] memo←memo,[1]i e j
[21] :EndIf
[22] :EndFor
[23] :If n=cnt+ρmemo
[24] tmp←tree[memo[;3];]
[25] tmp[;2 3 4 5]←0
[26] z←z,[1]tmp
[27] :Return
[28] :EndIf
[29] memo←memo[ρmemo[;2];]
[30] k←1+memo[;3]
[31] memo←1+[1]memo
[32] :EndWhile
    
```

The algorithm for selecting a smaller size tree with minimal costs is the following: having at a certain stage a tree of size n , we choose which of $(1:n)$ nodes (suppose they are not terminal nodes yet) to split next in order to get a tree of size $(n+1)$. The choice is in favor of that node that, after having been split, gives the maximum decrease in deviance. Starting from the root node (size = 1), the function examines all candidate nodes grown till the moment. If they are not terminal nodes (line [15]) it calculates the decrease in deviance for each of them (line [20]) and stores the results in a temporary variable *memo* (line [21]). When all nodes are examined, the function ranks the order of the matrix *memo* and picks the node with the maximum value of decrease in deviance (line [32]). The process continues till the desired tree size n (left argument) is reached (line [24]). Here, *cnt* equals to the number of terminal nodes obtained to this moment and $\uparrow\rho memo$ equals to the number of candidate nodes. Once no further split is needed, the nodes in *memo* are regarded as terminal ones (line [26]) and the process ends. The result of the function is the tree of size n with the lowest cost.

```
Print 3 Cut t2
1) 1 5.45 2 3 1 10
   2) 0 0 0 0 1 0 *
   3) 3 D C 6 7 2 6
   6) 0 0 0 0 3 0 *
   7) 0 0 0 0 2 2 *
```

An example shown above demonstrates that the reduced tree *t2* fits exactly to a tree *t*, built on “clean” data before errors were inserted. But we have now two misclassified points at node 7. This is unavoidable, because the noisy data does not fit the ideal rules.

Pruning the tree

In the previous section we learned how to find trees smaller than the maximal tree size. Now we have to snip off the least important splits on a regular basis. This process is called *cost-complexity pruning*.

Let us denote the cost of the subtree T' as $C(T')$. We will estimate a cost as the total misclassification rate of a tree. The size of a tree is equal to the number of terminal nodes and is denoted as $size(T')$. Then the cost-complexity measure is given by:

$$C_k(T') = C(T') + k \times size(T'),$$

where k is the complexity parameter.

Let us assign the initial value to k as zero. Now for every tree (including the first, containing only the root node), compute the value of the function above. Increase the complexity parameter continuously until the

value of the function for the largest tree exceeds the value of the function for a smaller-sized tree. Take the smaller-sized tree to be the new largest tree and continue increasing the complexity parameter. Stop the process when the root node becomes the largest tree.

Thus, we get a sequence of largest trees. The sequence has a number of interesting properties. It is nested, i.e. every tree contains all the nodes of the next smaller tree in the sequence. Initially, many nodes are pruned going from one tree to the next smaller tree in the sequence, but fewer nodes tend to be pruned as the root node is approached. Second, for every tree in the sequence, there is no other tree of the same size with lower cost. And finally, this is the very sequence of trees out of which we will choose the “right” tree size at the stage of cross validation.

To perform the task of pruning a classification tree we have a function *Prune* which takes a tree to prune as its right argument. The left optional argument *st* stands for “step” and indicates the value of increase of the complexity parameter k , line [15].

```
[0] z+{st}Prune tree;prime;min;max;k;sizes
[1] ±(0=NC'st')/'st+0.1'
[2] max++/tree[;2]=0
[3] sizes+~max
[4] prime+sizes Cut''ctree
[5] prime+(NErrors''prime),[1.5]sizes
[6] k+0
[7] z+φ0,-1+[1]prime
[8] :While max≠1
[9]     min+prime[;1]+k*sizes
[10]    min+min\|/min
[11]    :If min<max
[12]        max+min
[13]        z+z,[1]max,prime[max;1],k
[14]    :EndIf
[15]    k+k+st
[16]:EndWhile
```

In the line [4] we call the function *Cut* to search for trees with minimal costs for all tree sizes. This tree sequence is used later (lines [6-16]) to produce the final sequence of the cost-complexity optimal trees. The result is the 3-columns matrix. The first column contains tree sizes. The second column contains the corresponding costs and the third column contains the corresponding values of complexity parameter k :

```
Prune t2
6 0 0
4 1 0.5
3 2 1
1 10 4
```


Cross-validation and choosing the final tree

Cross-validation is a general statistical approach to select a model of optimal complexity. The main idea is to separate datasets used for model learning and testing. The quality of fitting on a training set increases with the increase in model complexity. That is not true for independent data of a testing set. Usually, the error rate of prediction reaches a small value at some "reasonable" complexity level and then decreases very slowly or even increases. Thus, we may select the simplest model with an acceptable error rate.

The cost-complexity pruning considered in the previous section gives us the right sequence of trees, ordered by complexity. We may now calculate the misclassification rate for each tree of this sequence on an independent dataset and select the optimal one.

In the present work we will use so-called *V-fold Cross-validation*. This type of cross-validation is useful when no separate test sample is available and the learning sample is too small to have the test sample taken from it. A specified parameter, *V*, determines the number of random subsamples, as equal in size as possible, that are formed from the learning sample. The classification tree of the specified size is computed *V* times, each time leaving out one of the subsamples from the computations. This subsample is used as a test sample for cross-validation. So that each subsample is used (*V* - 1) times in the learning sample and just once in the test sample. The misclassification rates computed for each of the *V* test samples are then averaged and used for an optimal tree selection.

We use a function, *Samples*, to generate random subsamples, which we need to perform the cross-validation procedure. It returns *n* (left argument) random subsamples of about the same size as its result. The right argument *size* indicates the size of the main sample.

```
[0] z←n Samples size;p;r
[1] z←size?size
[2] p←⌊size÷n
[3] r←size-n×p
[4] p←(npp)+n÷rρ1
[5] p←εpp⌈n
[6] z←pz
```

Here is an example of its work:
`disp 2 Samples 10`

```
→ +-----+
| 7 9 10 5 1 | | 6 2 4 8 3 |
← +-----+
ε
```

The next function *CrossValidate* performs *V*-fold Cross-validation. Its left argument *n* is the *V* parameter and the right argument is the tree object.

```
[0] z←n CrossValidate tree;data;e;ee
      ;m;rnd;s;sz;t;test;train
[1] (tree data)+tree
[2] sz←⊙,1+[2]Prune tree
[3] rnd←n Samples+ρdata
[4] z←0 3ρ0
[5] :For n :In sz
[6]   e←10
[7]   :For m :In rnd
[8]     test←data[m;]
[9]     train←data[εrnd-⊖m;]
[10]    t←GrowTree train
[11]    t←n Cut t
[12]    ee←+/test[;2>ρtest]*test Predict t
[13]    e←e,ee÷ρtest
[14]   :EndFor
[15]   m←+/e÷ρe
[16]   s←+/(e-m)*2
[17]   s←(s÷-1+ρe)*0.5
[18]   z←z,[1]n,m,s
[19]:EndFor
```

First, the function prunes the tree (line [2]). Then, for every tree in the obtained sequence (line [5]), the function grows a classification tree *n* times (line [7]). Every time a new training set is used (line [10]). Prediction accuracy is estimated on a new test set (line [12]). In lines [15-17], a mean value and a standard deviation of errors are calculated and passed to the result (line [18]). The final result is the matrix whose columns contain tree sizes, corresponding cross-validation costs and standard errors accordingly.

Let us apply function *CrossValidate* to a tree created with use of noisy data:

```
3 CrossValidate t2 data2
1 0.8          0
3 0.6          0.2
4 0.6666666667 0.3055050463
6 0.7333333333 0.2309401077
```

Results are shown in Fig.2, where we see the minimum misclassification rate for an optimal size-3 tree.

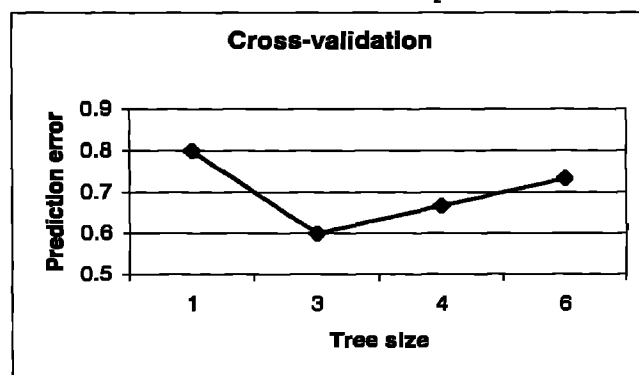


Figure 2: Cross validation results for artificial data

Thus, the automatic selection of an optimal size of a tree allows us to avoid the loss in the predictive accuracy produced by an effect of "overfitting".

Diagnostic Data Mining

As a Data Mining tool, tree-based modeling is increasingly used for summarizing large multivariate data sets. In this paper we demonstrate possibilities of this technique in application to monitoring the vibration of mechanical equipment in Nuclear Power Plants (NPP). The basic information unit of vibration is the spectrum of a signal measured by vibration sensors placed on mechanical equipment. Changes in vibration behavior and characteristics of a spectrum indicate mechanical changes in the monitored equipment. The data used in the paper were measured at a Nuclear Power Plant in Novovoronezh, Russia. We discussed the application to these data of Pattern Recognition techniques in the papers [6, 7].

Problem description

A typical NPP vibration monitoring system uses many sensors, performs measurements on a regular basis, and calculates high-resolution spectra. In our system there are 32 vibration sensors and each spectrum is estimated for 400 frequencies in the range 0-50Hz. The database size is growing rapidly and there is the need for an exploratory technique for uncovering structure in the data.

The system sensors and measured spectra differ in many aspects:

- Sensor type and measured signal may be pressure, absolute or relative displacement
- Sensor location, like steam generator or main coolant pump, and different coolant loops
- Measurement direction, as across or along a pipeline
- Displacement direction, as vertical movement of a reactor or steam generator movement to and from a reactor

All the above factors and any combination of them may be used for data categorization, such as

- Spectra of steam generator or coolant pump vibrations
- Spectra of steam generator vibrations for different coolant loops
- Spectra of absolute and relative displacements

The main classification of interest is if a spectrum from a particular sensor belongs to a normal or abnormal class of vibrations. But discovering the common features and differences for other categories, as we mentioned above, may give very important information about the system.

We successfully applied tree-based technique to a concise description of any reasonable category of data and large database summarization. Interesting patterns have been uncovered and used for malfunction diagnostics. In this paper we briefly consider only two examples of this research:

1. Classification of vibration spectra of steam generators or coolant pumps
2. Classification of vibration spectra of steam generators of different coolant loops

For graphic representation of data, we used our implementation of AP207 emulator for Dyalog APL [5]. In this paper we also described the syntax of associated utilities.

SG-MP Classification

In this example we are interested in the differences between spectra of a Steam Generator (SG) and the Main Coolant Pump (MP) vibrations. A set of 1098 spectra of both classes were divided into a training and testing sets:

```

      ps
1098 400
      pcl
1098
      Unique cl
1 2
      ptrain
500 401
      ptest
598 401
    
```

The tree-based model was built using only data of the training set:

```

t←GrowTree train
NErrors t
0
    
```

The resulting tree is shown in Fig.3. This model allows the right classification, but looks too complicated (11 terminal nodes). Using the function *Cut* we created a sequence of rooted subtrees of sizes from 1 to 8. Then we calculated the misclassification rate using each subtree for prediction on independent test data.

```

p←(ctest[;1400]) Predict“(18)Cut”ct
(18).[.5]+/“(ctest[;401])#p
1 2 3 4 5 6 7 8
282 88 39 38 36 36 23 22
    
```

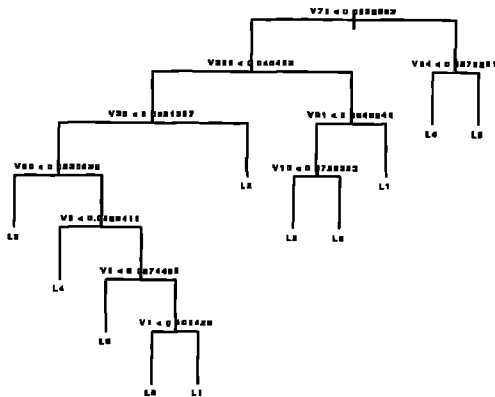


Figure 3: Full Tree for SG-MP Classification

This model gives correct classification, but looks too complicated (11 leaf nodes). Using the function *Cut* we created a sequence of subtrees of sizes 1 to 8. Then we calculated the misclassification rate using each subtree for prediction on independent test data.

```
p+(c test[;1400]) Predict"(18)Cut"ct
(18),[.5]+/"(c test[;401])#p
1 2 3 4 5 6 7 8
282 88 39 38 36 36 23 22
```

This result allowed us to select an optimal tree with only 3 terminal nodes:

```
t3+3 tr.Cut t
Print t3
1) 196 0.04816263452 2 3 1 242
2) 69 0.06912294284 4 5 1 60
3) 0 0 0 0 2 4 *
4) 0 0 0 0 1 16 *
5) 0 0 0 0 2 0 *
100*(+/cl*s Predict t3)+pcl
5.373406193
```

As it is shown above, the prediction error of an optimal tree is about 5.4%. It is more than acceptable for our goal of dataset summarization. And only 2 of 400 predictors (spectrum values at frequencies 69 and 196) have been used for classification. The following simple rules describe the main structure in data:

```
>(c t3) GetRule"3 4 5
Class 2 IF V196>0.048
Class 1 IF V196<=0.048 AND V69<=0.069
Class 2 IF V196<=0.048 AND V69>0.069
```

Let us consider median spectra of each class for the above rules interpretation:

```
med+{0.5*+ /w[(f w)][{0.5*0 1+p w}]}
m1+med" c[1](cl=1)+s1
m2+med" c[1](cl=2)+s1
```

plot (1400) m1 m2

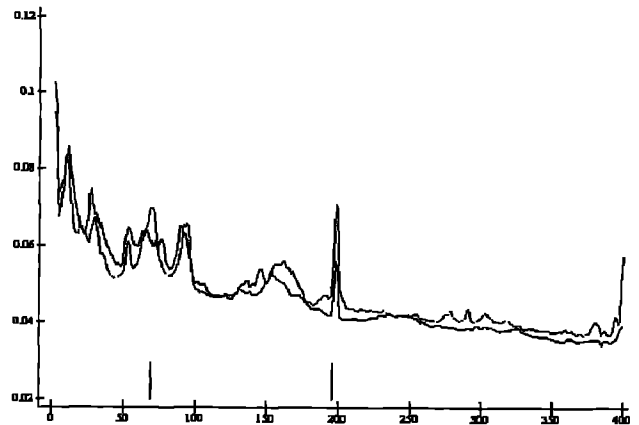


Figure 4: Median spectra for classes SG and MP

Two vertical lines in the Fig 4 indicate frequencies used in the classification tree. It may be seen, for instance, that class MP has a higher level for a large peak at frequency 196. This is the main frequency of a pump wheel rotation. A sensor located on the coolant pump detects the associated vibration more easily.

Different coolant loops classification

Let us now find rules to distinguish spectra from sensors located at the different coolant loops. There are 6 coolant loops at the reactor we work with. In general, there should not be any difference between vibration of the mechanical equipment of the same type, but located at different coolant loops. Therefore, we expect a large tree with only a few objects at each terminal node. Let us check if that is the case. We used for constructing the tree a dataset of 574 spectra from 6 classes:

```
pS
574 400
pcl
574
Unique cl
1 2 3 4 5 6
+/(16)*.=cl
117 107 66 93 125 66
```

The resulting tree is a complicated one:

```
t+GrowTree s,cl
Print t
1) 71 0.06326232251 2 3 5 449
2) 399 0.04046299416 4 5 1 247
4) 35 0.069135 8 9 2 76
8) 60 0.0530531941 16 17 3 10
16) 0 0 0 0 3 0 *
17) 9 0.082941066 34 35 4 4
34) 0 0 0 0 4 0 *
35) 1 0.087449465 70 71 6 2
70) 0 0 0 0 6 0 *
71) 1 0.10142617 142 143 1 1
142) 0 0 0 0 5 0 *
```

```

143)      0 0      0 0 1 0 *
9)        0 0      0 0 2 0 *
5)        91 0.06462439149 10 11 1 65
10)       10 0.07983529657 20 21 6 1
20)       0 0      0 0 5 0 *
21)       0 0      0 0 6 0 *
11)       0 0      0 0 1 0 *
3)        84 0.0575261014 6 7 5 87
6)        0 0      0 0 4 0 *
7)        0 0      0 0 5 0 *
    
```

We used the whole dataset for learning. Let us apply the cross validation procedure to define the right size of a tree:

```

+cv+5 CrossValidate t (s,cl)
1 0.8013424867 0.03797851893
2 0.6063463005 0.06012736467
3 0.4059344012 0.05174932396
4 0.2595728452 0.014051453
5 0.1620594966 0.02463777562
6 0.03485888635 0.01743974679
7 0.02613272311 0.01506165904
8 0.02091533181 0.01457146716
11 0.01743707094 0.01742018558
0 1 plot 1 2/c[1]cv[;1 2]
    
```

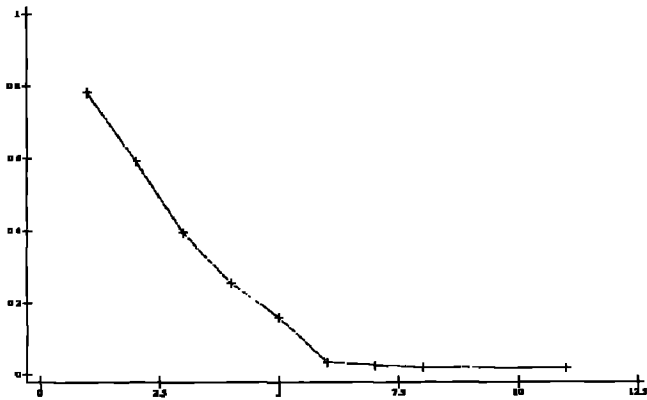


Figure 5. Cross validation results

We see that the misclassification rate almost stops decreasing after tree size reaches 6. The tree of that reduced size is shown in Figure 6 (a plot was created in the R statistical system):

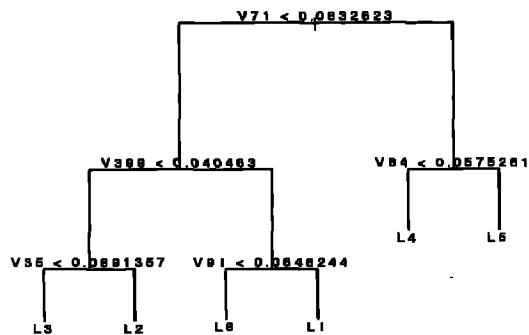


Figure 6: Reduced tree structure

This tree is unexpectedly simple and accurate. Error rate is only about 2%.

```

t6+6 Cut t
100*(NErrors t6)/+ps
1.916376307
    
```

Other news is even more important. The first split divides all data into two groups of coolant loops {1,2,3,6} at the left branch and {4,5} at the right one.

Let us have a look at median spectra of these two groups to interpret the observed classification:

```

m1+med[c[1](cl=1 2 3 6)]/s
m2+med[c[1](cl=4 5)]/s
plot (1400) m1 m2
    
```

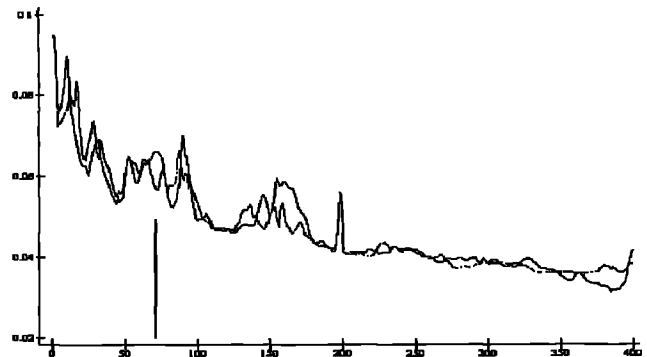


Figure 7: Median spectra for two groups of coolant loops

The vertical line in Figure 7 indicates the root split frequency. It may be seen that spectra at loops 4 and 5 have a peak at frequency 71 and the other loops have a spectrum dip at the same frequency. To see how significant this difference is, let us use a Box-Whisker plot for two subsets of the predictor 71 values:

```

x1+(cl=1 2 3 6)/s[;71]
x2+(cl=4 5)/s[;71]
bw_plot x1 x2
    
```

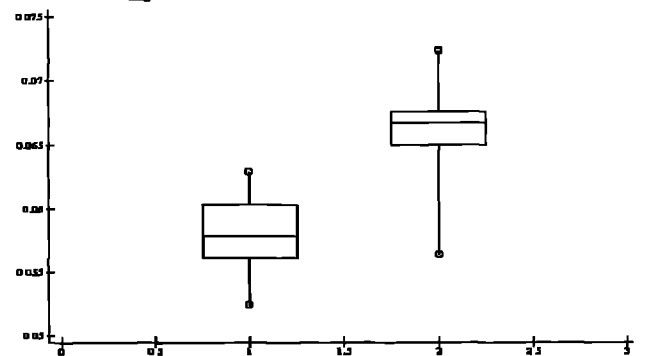


Figure 8: Box-Whisker plot for two groups of coolant loops

The difference is highly significant and therefore we may conclude that the important distinctive feature

of the vibration characteristics of coolant loops 4 and 5 has been uncovered.

Conclusion

A popular Data Mining technique known as Classification Trees has been considered in detail. All major parts of the technique starting from split selection and growing the tree and finishing with tree structure optimization were discussed and illustrated by numeric examples in APL.

It has been shown that Dyalog APL allows effective implementation of the software to build tree-based models and to use them for prediction. Use of control structures has simplified the code and improved readability significantly. The functions *Print* and *GetRule* may be mentioned as a good illustration of APL's power in array processing. The use of dynamic functions has been seen as very convenient for exploratory calculations in the APL session.

A case study of summarizing of large datasets and uncovering hidden patterns in the area of Nuclear Power Plant vibration monitoring has been discussed and illustrated by real life examples.

As future work directions, we would like to mention:

- Implementation in APL of Regression Trees technique
- Development of utilities for graphical representation of tree-based models
- Modifying existing software to allow direct building of a tree with data stored in a relational database.

References

- [1] W.Y. LOH AND Y.S. SHIN, "Split Selection Methods for Classification Trees", *Statistica Sinica*, 1997, Vol. 7, pp.815-840
- [2] *The R Project for Statistical Computing*. www.r-project.org
- [3] B.D. RIPLEY, *Pattern Recognition and Neural Networks*. University Press, Cambridge, 2000.
- [4] A.O. SKOMOROKHOV, "A Knowledge Discovery Method - APL Implementation and Application". *APL 2000 Conference Proceedings*, Berlin, Germany, *APL Quote Quad* Vol. 30, Num. 4W.N.
- [5] A.O. SKOMOROKHOV AND A.N. KORNILOVSKY, "Emulation of IBM APL2 auxiliary processor AP207 in Dyalog APL", *Vector*, Vol.14, No.1, p.64
- [6] A.O. SKOMOROKHOV AND M.T. SLEPOV, "Information Authenticity Control in Vibro-Diagnostics System of the Novovoronezh NPP". *Communications of Higher Schools: Nuclear Power Engineering*. - Obninsk, 1999.
- [7] A.O. SKOMOROKHOV AND M.T. SLEPOV, "Pattern Recognition in APL with Application to Reactor Diagnostics", *APL'98 Conference proceedings: APL Quote Quad*, Vol. 28, Num. 4, Rome, Italy, July 1998.
- [8] VENABLES AND B.D. RIPLEY, *Modern Applied Statistics with S-PLUS*, Springer, 1999
- [9] N. G. ZAGORUIKO, *Applied Methods of Data and Knowledge Analysis*, Novosibirsk, published by Institute of Mathematics, 1999.