Learning on Expression Array Data
Project Report for CS786S

Bronislava Brejová and Tomáš Vinař

Spring term 2000

Contents

1 Introduction

In this project we have explored possibilities of using machine learning algorithms in biological data, specifically in processing expression array data. This data have specific properties that require customized algorithms and methods. We have focused our attention to data preprocessing, especially discretization of continuous attributes.

1.1 Biology Background

Genetic information of living organisms is stored in DNA molecules. Some regions of DNA molecules are called coding regions, or genes, because they encode the chemical composition of proteins used in cells of the organism. Proteins are synthestized in a cell during the process called translation, following the information encoded in genes. Proteins then influence all other chemical processes in a cell, including generation of new proteins.

Today all genes are known for some species, such as yeast, and a large portion of all genes is known for other organisms, such as human. Although we might know the location of a gene and the composition of the corresponding protein, we often do not know the function of the protein, i.e. in which processes in a cell the protein participates and what is its specific role in these processes. Finding functions of genes is a very important biological question.

One of the recently discovered tools that help to approach this question is the expression array technology. This technology allows to measure expression levels of genes, i.e. how much each gene is used to produce new proteins at a given moment. Important thing is that we can measure and compare expression of all genes at a given moment by a single experiment. High expression level of a gene indicates that this gene may have some role in processes occuring in the organism under given experimental conditions. Performing multiple experiments under different conditions can therefore provide an avaluable insight into functions of genes.

However expression array experiments produce huge amounts of data that cannot be any longer processed by hand. In a simple organism, such as yeast, there are 6 000 genes and the human genome contains tens of thousands of genes. Thus a single experiment creates thousands of values and typically several experiments are done under different conditions or in a time series. Biologists need software tools to find relevant patterns and to discover potential functions of genes from this kind of data.

There are several factors that make analysis of expression array data difficult.

- The obtained values are not an absolute values, reflecting the number of proteins produced from a gene, but they are only relative ratios, comparing this amount to the amount of the same protein generated in some reference sample.
- The technology used is still very inaccurate. Changes in expression with ratio as high as 2 are considered to be reproducible in other experiments. However such great changes in expression are relatively rare. Smaller ratios are often caused only by noise.
- The values obtained from two different experiments cannot be directly compared. Values measured at the experiment depend on many factors, some of them are irreproducible settings done by human and never recorded.

Gene	Function	0 min.	7 min.	14 min.	21 min.	...
YBR166C	METABOLISM	0.33	-0.17	0.04	-0.07	...
YOR357C	CELLULAR-ORG	-0.64	-0.38	-0.32	-0.29	...
YLR292C	PROTEIN-DEST	-0.23	0.19	-0.36	0.14	...
YGL112C	CELLULAR-ORG	-0.69	-0.89	-0.74	-0.56	...
YIL118W	UNKNOWN	0.04	0.01	-0.81	?	...
YDL120W	HOMEOSTASIS	0.11	0.32	0.03	0.32	...
YHL025W	METABOLISM	-0.47	1	-0.51	-0.25	...
YGL248W	CELLULAR-ORG	-0.25	0.26	0.01	-0.06	...
...	...	...	...	...	...	...

Figure 1: An example of expression array data.

- Often it happens, that small portions, or even large parts of an expression array have been processed improperly and therefore it is impossible to read some of values. However, experiments are too expensive to repeat until all values can be read, therefore we need to deal with the problem of missing values.

For all these reasons, data processing of expression array data is a very specific task. It is also challenging and important problem, which needs to be handled.

1.2 Problem Definition

Machine learning problem we try to solve is the prediction of a gene function from expression values measured in several experiments. For every gene we have (see Figure 1):

- **Gene name.** This is an identifier of the gene.
- **Gene function.** Genes are divided into functional classes. This function is determined by human expert. Determining gene function is usually a hard research problem and involves many of experiments. Some of genes in our data set have undetermined gene function. This means, that research in this area is still in progress.
- **Experiment results.** Each experiment forms one attribute. Each value means level of expression of particular gene in particular experiment. Some of the experiments were performed in time series (see Figure 1). There can be missing values.

Our task is to predict gene function from expression levels measured under different conditions. The set of genes with function already determined serves as a training set. After learning process is finished, the resulting system can be used for predicting functions of genes, for which function is not yet determined.

Such a prediction system can be used by researchers in following ways:

- Predicted function can serve as a good starting point for research in the area. On basis of such a prediction, researches can decide, which conjectures about function of particular gene to test first.
- Internal structure of induced classifier (for example rules produced by Elem2 system), if presented and analyzed in suitable way, can show researchers hidden dependencies in expression array data. Such dependencies cannot be found simply by hand due to enormous amount of the data.

1.3 Project Overview

First we will try to describe briefly the past work in the area from two different points of view: from the view of bioinformatics and from the view of machine learning. Then we will describe our approach and show, how we tried to link ideas from both fields together, and how our approach differs from the past work.

In our work we focused on discretization. In Section 4 we will describe discretization algorithm, which we used to perform our experiments. For the experiments we needed to write numerous small utility programs, each of them alone not very interesting. In Section 5 we will describe, how these utilities hang together and how to perform experiments using our programs. Finally, we will give an overview of experimental results and conclusions.

We would like to point out, that in this project, the main point was not to implement some known or novelty algorithm or system. The main idea was to try to use known techniques on a new data set, identify special features of our data set and try to improve on results by using techniques focused on these features. In part 6.2.2 we also describe some experiments, which were not successful.

We also tried (where possible) to compare, how the problem stated here is seen from the bioinformatics point of view (with its current techniques) and from the machine learning point of view.

2 Past Work

In this section we outline papers that were motivation and basis of our work, both in the field of machine learning and more specifically discretization, and in bioinformatics.

2.1 Past Work in Bioinformatics

Analysis of expression array data is a new and dynamically developing area of research in bioinformatics. There are different kinds of biological questions one can try to answer using these data. Examples include classification of healthy and tumorous tissues based on expression levels [GST+99], modeling interactions among a group of related proteins using differential equations, boolean networks [AMK99], linear models, etc, discovering relationships of different tissues in an organism [PSG+00] and so on. Research of some of these questions involves methods of machine learning.

We will concentrate on one particular question, namely prediction of gene functions based on expression patterns. The most common approach taken in bioinformatics is clustering of genes (see for example [ESBB98, BK00, ZZ00]). This is an unsupervised method working as follows. Each gene is represented by a vector of real values containing results of individual experiments. In the first step distance in some appropriate metrics is computed between each two genes. We can take simply Euclidean distance of vectors or some form of correlation coefficients. Based on this distance table items with small distance are grouped together to form clusters. There are different clustering algorithms, most of them working on some kind of greedy heuristics, grouping together the closest pairs of genes.

After clustering the list of the genes is reordered so that genes that are in one cluster are adjacent. Such a reordered list is then displayed for a user in a graphical form – each vector is depicted as one line, individual expression levels are depicted using a color scale – the brighter is the color, the greater is the value. Clusters are also shown in the figure.

An experienced biologist can spot in such a figure clusters of genes that have interesting properties. It was observed in [ESBB98] that many clusters consist of genes with the same function.

Other interesting approach is to use principal component analysis (PCA) to reduce dimensionality of the data (see [RSA00]). PCA is a traditional statistical method that replaces original variables with a smaller number of new variables so that the new variables are linear combinations of the original variables and they capture as much variance in the data as possible. Using this approach Raychaudhuri, Stuart, Altman [RSA00] were able to reduce the number of attributes from 7 to 2 and still capture most of the variance. Again, their method was based mostly on visualization – they have displayed genes as a points in 2D space (since they had only 2 attributes) and they have tried to observe interesting properties of this diagram.

To conclude, the work done in gene function prediction was based mainly on unsupervised approaches, in which the information about known functions of some genes is not taken into account. The emphasis was placed on convenient graphical representation of data to user. Still ideas in the mentioned articles inspired us in our work, since they show that genes with the same function have often similar expression patterns and therefore expression array data can be used in machine learning to predict the function.

2.2 Past Work in Machine Learning and Discretization

Machine learning is a well-established area with a wealth of results. The purpose of this short overview is to provide a short introduction to those concepts which particularly closely related to our work.

Classification problem is problem in which we are given an example consisting of values of several condition attributes and based on this information we want to classify this example to one of the classes, i.e. predict a value of unknown class attribute. Supervised machine learning program is a program that gets several classified examples and based on them it induces a general classification procedure which can be used to classify other examples.

We will use only two types of classification procedures – decision trees and decision rules. Decision tree is a tree which has a condition in each inner node and class in each leaf. During classification we start from root and evaluate condition in each node. The answer determines to each child of the node we should move next. The class in a leaf gives the predicted class. We will use C4.5 decision trees constructing program by Quinlan [Qui93].

Decision rules are logical rules. During classification we search for a rule that fits our data and this rule determines the answer. In case that more than one rule fits our data, we need a rule quality measure that helps us to choose the best answer. In this project we use rule induction system Elem2 by An and Cercone [AC98].

Algorithms for decision rule and decision tree induction are suitable especially in cases when eeac attribute is symbolic, i.e. it can have only one of a small number of different values. However in practice one often encounters attribute that have real values. Therefore there were developed approaches how to discretize such attributes. Discretization of a continuous attribute is done by dividing the range of the attribute to intervals and assigning each interval some symbolic value.

There are several methods for discretization of continuous attributes. They can be divided into supervised and unsupervised methods. Supervised methods take into account classification of training examples. Unsupervised methods are based only on values of a continuous attribute. Below we list some discretization techniques.

- **Equal width intervals** is a simple method unsupervised that divides entire possible range of a continuous attribute into specified intervals of equal length.
- **Equal size bins** method divides the range into several intervals so that each interval contains the same number of values of the continuous attribute.
- **MDLP method** is a supervised method by Fayyad and Irani [FI93] based on entropy minimization. In each step all possible cut points are evaluated and the one which achieves the smallest entropy of the class attribute is chosen. The range is then divided into two parts and for each part discretization continues recursively. Subrange is not divided any further if it satisfies Minimum Description Length Principle.
- **EDA-DB method** is a supervised method used in Elem2 [AC99]. It ranks possible cut points according to the entropy measure, similarly as in the previous method. Then it chooses best m of them so that the chosen cut points are reasonably distributed throughout the entire range. The number of cut points is chosen as $m = \max\{2, k \log_2 l\}$ where k is the number of classes and l is the number of distinct values of the given attribute.
- **eOPT method** is a method that chooses the set of cut points so that the overall entropy is minimized. The main difference betweenIt MDLP and eOPT method is that MDLP always chooses one most promising point. In eOPT all points are chosen at once in order to minimize entropy. This can be done in polynomial time using dynamic programming algorithm introduced in [FKS95].

In contrast to these methods, C4.5 does not discretize attributes beforehand but chooses division points locally, evaluating the cut points only of subset of examples corresponding to the currently processes node of a decision tree.

There are several surveys and experimental studies regarding discretization. Elomaa and Rousu [ER99] mention entropy based approaches and they also list several other evaluation criteria that can be used

instead of entropy. They study which of them can be minimized using dynamic programming algorithm. They also perform an empirical study of influence of different evaluation criteria and splitting algorithms (such as binary recursive strategy or dynamic programming) on prediction accuracy of C4.5. It turns out that in case of C4.5 splitting strategy is not so important, whereas evaluation criterion has an impact. The best results were achieved by an entropy-based measure, which takes into account also the number of cut points. An and Cercone [AC99] study effect of discretization on Elem2 system, finding out that their EDA-DB method performs better than MDLP method. Dougherty, Kohavi and Sahami [DKS95] overview many discretization methods and compare influence of equal width, simple greedy method and MDLP method on C4.5, concluding that entropy has the best results.

3 Our approach

In this project we tried to apply machine learning methods to the bioinformatics domain. We have used C4.5 and Elem2 programs together with our own discretization program to predict function of a gene from its expression profile. In experiments we focused our attention to comparison of different discretization methods and different number of cut points used in discretization.

We found our project different from the most of the work done to date at least in the following points:

- We have applied supervised learning methods in context of gene function prediction from expression array data. Research so far focused on clustering and other unsupervised approaches.
- We have used eOPT discretization together with Elem2 learning algorithm. This combination was not explored before.
- We studied performance of learning algorithms on discretized data with different numbers of cut points. In this way we tried to give an evidence, that in some cases (for example in case of gene function prediction from expression profile) it is useful to keep number of cut points small.

4 eOPT Discretization Algorithm

4.1 Minimizing Entropy

The input of the algorithm contains n real-valued numbers $v_1 \leq v_2 \leq \ldots \leq v_n$ (we will call them points). For every point v_i we are also given its class c_i (symbolic value). Denote number of classes c. Finally, we are given integer k.

We want to find a set of k cut points $P(n,k) = \{p_1, p_2, \ldots, p_k\}$. These cut points divide points $v_1, \ldots, v_n$ into $k+1$ bins $B_0, \ldots, B_k$, where bin B_i contains all points v such that $p_i < v \leq p_{i+1}$ (let $p_0 = -\infty$ and $p_{k+1} = \infty$).

We want to choose the cut points so that the following function is minimized:

$$\mathit{impurity}(P(n,k)) = \sum_{i=0}^{k} \frac{|B_i|}{n} \mathit{entr}(B_i), \tag{1}$$

where $\mathit{entr}(B_i)$ is an entropy of class distribution in B_i, i.e. if $P(j \mid B_i)$ is a frequency of points with class j in bin B_i then

$$\mathit{entr}(B_i) = -\sum_{j=1}^{c} P(j \mid B_i) \log P(j \mid B_i). \tag{2}$$

Entropy based discretization used together with decision tree based learning methods has proved to be successful in past (see for example [DKS95, FI93, AC99, ER99]). However, most of the approaches so far used heuristic, or recursive partitioning to minimize (1). We present here a dynamic programming algorithm, which for given number of cut points computes global optimum[1].

[1] The dynamic programming algorithm for this problem was already presented in [FKS95], however, we did not know about this article, when we started working on the implementation. Our work is independent from Fulton et al.

4.2 Algorithm Description

Our algorithm is based on dynamic programming technique. We want to compute $P(n,k)$, set of k cut points for a set of points $v_1 \leq \ldots \leq v_n$, such that $impurity(P(n,k))$ is minimized. We will show, how to compute such a set efficiently, providing that we have already computed cut point sets for all smaller values of n and k and we will present efficient dynamic programming algorithm for this task.

To keep things simple, from now on we will asume, that values $v_1, \ldots, v_n$ are all distinct. Algorithm described here can be easily extended to handle cases, where this assumption is not met. We will deal with the problem in section 4.4.

Let v_i be the largest cut point from $P(n,k)$. We can compute impurity for $P(n,k)$ as follows:

$$impurity(P(n,k)) = \begin{cases} \frac{i}{n} impurity(P(i,k-1)) + \frac{n-i}{n} entr(\{v_j \mid v_i < v_j \leq v_n\}), & \text{if } k > 0, \\ entr(\{v_1, \ldots, v_n\}), & \text{if } k = 0. \end{cases} \quad (3)$$

Notice, that if $P(n,k)$ has an optimal impurity, then also $P(i,k-1)$ must have an optimal impurity. If there would exist set of cut points $P'(i,k-1)$ with impurity less than $P(i,k-1)$, according to (3) the set of cut points $P'(i,k-1) \cup \{v_i\}$ would have a smaller value of impurity than $P(n,k)$.

Therefore, we can compute optimal impurity by examining all possible cut points as follows (denote $e(n,k)$ the optimal impurity for points $v_1, \ldots, v_n$ using k cut points):

$$e(n,k) = \begin{cases} \min_{1 \leq i \leq n} (\frac{i}{n} e(i,k-1) + \frac{n-i}{n} entr(\{v_{i+1}, \ldots, v_n\}), & \text{if } k > 0, \\ entr(\{v_1, \ldots, v_n\}), & \text{if } k = 0. \end{cases} \quad (4)$$

To complete our algorithm, for every n and k we need to store the index i for which the minimum was obtained. Denote this value by $idx(n,k)$. Notice, that $idx(n,k)$ is in fact the number of the last cut point for a set of k cut points with the minimum impurity. Therefore we can restore all cutpoints of the set after computation (last but one cut point is $idx(idx(n,k),k-1)$ etc.). We can summarize this approach in the following pseudocode:

for all points v_i **do** {compute $e(*,0)$}
  $e(i,0) = entr(\{v_1, \ldots, v_i\})$
  $idx(i,0) = -1$
end for

for $p = 0$ to k **do** {now proceed with dynamic programming}

  for all points v_i **do** {compute $e(i,p)$}
    $mincost = \infty$; $minarg = -1$
    for all points v_{last}, where $last \leq i$ **do** {try all possible last cut points}
      $cost = \frac{last}{n} e(last, p-1) + \frac{n-last}{n} entr(\{v_j \mid v_{last} < v_j \leq v_i\})$
      if $cost < mincost$ **then**
        $mincost = cost$
        $minarg = last$
      end if
    end for
    $e(i,p) = mincost$
    $idx(i,p) = minarg$
  end for{computation of $e(i,p)$}
end for{dynamic programming}

{recover set of cut points with minimum cost}
$id = n$; $i = p$
while $idx(id,i) \neq -1$ **do**
  $id = idx(id,i)$; $i = i - 1$
  v_{id} is a cut point
end while

Notice, that we can compute entropy of any subset in $O(n)$ time, therefore the time complexity of the whole algorithm is $O(n^3 p)$.

4.3 Efficient Computation of Entropy

We can speed up the algorithm using more efficient computation of entropy. Notice, that we are computing entropy only for intervals of points.

We will first afford $O(cn)$ time preprocessing. In this step we will compute values $inclass(i, j)$ – the number of points from set $\{v_1, \ldots, v_i\}$, which have $c_i = j$ (are in class j). Define $inclass(0, j) = 0$.

Then we can compute an entropy of the set $\{v_i, \ldots, v_{i'}\}$ according to the following formula (notice, that number of points in this set, which are in class c is exactly $inclass(i') - inclass(i-1)$):

$$entr(()\{v_i, \ldots, v_{i'}\}) = -\sum_{j=1}^{c} \frac{inclass(i', c) - inclass(i-1)}{i' - i + 1} \log \frac{inclass(i', c) - inclass(i-1)}{i' - i + 1} \tag{5}$$

However, this computation can be now easily performed in $O(c)$ time and therefore the whole dynamic programming algorithm after incorporating this speed-up will take $O(n^2 cp)$. Since the number of classes is usually much less than the number of points, this will speed up our algorithm significantly.

4.4 Boundary Points

In the previous section we have described how to compute set of k cut points in $O(n^2 cp)$ time using a dynamic programming technique. This method is quite efficient, however, the number of points involved (n) can be quite large and dynamic programming algorithm can take a long time. Fortunately, not all points need to be considered as a potential cut points.

Assume, that all points have different values (we will get rid of this assumption later in this section). The result of Fayyad and Irani [FI92] allows us to reduce the number of potential cut points involved in dynamic programming substantially:

Definition 1 *Point v_i is a* boundary point *if at least one of the following conditions is met:*

1. *v_i is the last point (i.e. $i = n$)*
2. *$c_i \neq c_{i+1}$*

Lemma 1 (Fayyad, Irani [FI92]) *Let P be a set of k cut points with the minimum value of impurity. Then there exist a set P' with the same number of cut points and the same impurity, such that all points in P' are* boundary points.

Lemma 1 simply tells us that when computing optimal set of cut points, we need only consider boundary points. Boundary points can be easily detected in $O(n)$ time in preprocessing step. Dynamic programming algorithm is then performed only on a set of boundary points (no substantial changes to the algorithm are necessary).

Notion of boundary points can also help us to solve the problem of several points with the same value. In such case we will use only *the last point* of every group of points with the same value as a candidate for boundary point. Choosing other points does not make sense, since all points in one group must fall into one bin.

Then our condition for testing, whether the point is a boundary point or not looks as follows: For point v_i denote G_{i-} the group of points with the same value as v_i and G_{i+} the group of points with the least higher value. Then v_i is *not a boundary point* if all points in $G_{i-} \cup G_{i+}$ belong to the same class. Testing this condition can be easily performed.

4.5 Implementation, Data Structures

We implemented the dynamic programming algorithm described above in the program Discretize. We will here shortly describe our implementation by explaining data structures and their roles in functions in the program.

- **Input values.** Array input_vals stores values read from the input file. For each row in the input file we store:
  - **cols** – the number of attributes in the row (should be the same for all rows and we use it only for the purpose of checking the correctness of the input);
  - **cl** – class, to which row belongs
  - **id** – identification of the row (as appears in the input)
  - **array val** – array of values of attributes
  - **array missing** – signal, if value is missing.

  This array is not passed to the dynamic programming procedure. For dynamic programming, new structure is prepared, containing only relevant values. In this way we were able to separate dynamic programming from reading input and data representation, which can be very useful, if this method should be incorporated in the other program, or if the data representation is changes to handle more complex data sets (e.g. some attributes continuous, some symbolic).

- **Boundary point information.** Function **prepare_boundary** takes one column of the input file which needs to be discretized. It computes, which points are boundary points and prepares data structure which is passed as an input to the procedure **compute_cut_points** (dynamic programming). This process is repeated for every column separately.

  The boundary point information array contains only data needed for dynamic programming, i.e. for every boundary point:
  - **val** – real value of the boundary point,
  - **order** – the order of the boundary point in a list of all points sorted by increasing value
  - **array inclass** – containing *inclass* values needed for computation of entropy (see Section 4.3)

  Boundary points are stored in order with increasing value.

- **Dynamic programming table.** This table is created locally in function performing the computation of cut points **compute_cut_points**. For every p and i, where p is the number of cut points and i is the index of a boundary point, it stores:
  - **entr** – the optimal impurity for the set of all points with value less or equal to the value of ith boundary point, where the cut point set contains p points.
  - **idx** – the largest cut point used in such optimal cut point set.

  After computation is completed, we can recover the optimal set using **idx** values, as it is described in Section 4.2.

4.6 Testing

We have tested our implementation extensively. Here we are presenting samples of testing data.

In the left column below you can see the input file `test.txt`. The first column of this file is ID attribute, the second column is class attribute, the third and fourth columns are continuous attributes to be discretized. Notice that the third column has 4 boundary points, whereas the fourth column only 2.

In the right column below you can see the transcript of the testing session. The program always outputs for each continuous column the number of non-missing values in that column and number of boundary points

(recall that the last point is always a boundary point). For eOPT discretization it also outputs the found cut points in decreasing order.

We have tested the program for 1,2,3 and 4 cut points with eOPT algorithm and for 2 cut points with equal size and equal width algorithms.

```
 1 1 10.0 10.0
 2 1 10.1 10.0
 3 1 10.2 11.0
 4 2 11.0 11.0
 5 3 11.1 11.0
 6 1 12.0 12.0
 7 1 12.1 12.0
 8 3 13.0 12.0
 9 3 13.1 12.0
```

```
Script started on Sat Aug 12 18:20:37 2000
bash-2.01$ ./discretize test1.cut 1 <test.txt >test1.out
Column 1: [n: 9 nbp: 5] 12.100000
Column 2: [n: 9 nbp: 3] 10.000000
bash-2.01$ ./discretize test2.cut 2 <test.txt >test2.out
Column 1: [n: 9 nbp: 5] 11.000000 10.200000
Column 2: [n: 9 nbp: 3] 11.000000 10.000000
bash-2.01$ ./discretize test3.cut 3 <test.txt >test3.out
Column 1: [n: 9 nbp: 5] 12.100000 11.100000 10.200000
Column 2: [n: 9 nbp: 3] 12.000000 11.000000 10.000000
bash-2.01$ ./discretize test4.cut 4 <test.txt >test4.out
Column 1: [n: 9 nbp: 5] 12.100000 11.100000 11.000000 10.200000
Column 2: [n: 9 nbp: 3] 12.000000 12.000000 11.000000 10.000000
bash-2.01$ ./discretize test2s.cut 2 1 <test.txt >test2s.out
Column 1:
Column 2:
bash-2.01$ ./discretize test2w.cut 2 2 <test.txt >test2w.out
Column 1:
Column 2:
bash-2.01$
bash-2.01$ exit

Script done on Sat Aug 12 18:21:08 2000
```

Below we can see cut points as generated on the shown input file with different parameters. The first line of each file contains the cut points for the first continuous attribute and the other line for the second continuous attribute.

The left column contains results of eOPT for 1,2,3 and 4 cut points. Notice that when we choose the number of cut points greater than the number of boundary points in some columns, the last value will be repeated more than once. The right column contains cut points for equal size and equal width methods. Equal size method in our implementation does not work very well for inputs with large groups of the points with the same value, as is the case in the second column of this small input file. The equal size method created only 2 diferent values instead of 3, because it simply chooses cut points to be the real values of the 3rd and 6th line. The problem is that all the points between 3rd and 5th line have equal values and therefore they all belong to the same group as the first two points. The real input data do not contain such big groups of equal values and therefore this problem is negligible and all groups have approximately the same size.

```
test1.cut:
 12.100000
 10.000000
test2.cut:
 10.200000 11.000000
 10.000000 11.000000
test3.cut:
 10.200000 11.100000 12.100000
 10.000000 11.000000 12.000000
test4.cut:
 10.200000 11.000000 11.100000 12.100000
 10.000000 11.000000 12.000000 12.000000
```

```
test2s.cut:
 10.200000 12.000000
 11.000000 12.000000
test2w.cut:
 11.033333 12.066667
 10.666667 11.333333
```

Finally here we see the output files with real values replaced by symbolic values.

```
test1.out:    test2.out:    test3.out:    test4.out:    test2s.out:   test2w.out:
1 1 0 0       1 1 0 0       1 1 0 0       1 1 0 0       1 1 0 0       1 1 0 0
2 1 0 0       2 1 0 0       2 1 0 0       2 1 0 0       2 1 0 0       2 1 0 0
3 1 0 1       3 1 0 1       3 1 0 1       3 1 0 1       3 1 0 0       3 1 0 1
4 2 0 1       4 2 1 1       4 2 1 1       4 2 1 1       4 2 1 0       4 2 0 1
5 3 0 1       5 3 2 1       5 3 1 1       5 3 2 1       5 3 1 0       5 3 1 1
6 1 0 1       6 1 2 2       6 1 2 2       6 1 3 2       6 1 1 1       6 1 1 2
7 1 0 1       7 1 2 2       7 1 2 2       7 1 3 2       7 1 2 1       7 1 2 2
8 3 1 1       8 3 2 2       8 3 3 2       8 3 4 2       8 3 2 1       8 3 2 2
9 3 1 1       9 3 2 2       9 3 3 2       9 3 4 2       9 3 2 1       9 3 2 2
```

5 Programs Supporting Experiments

During the work on this project we have done a large number of experiments. The first ones were done mostly by hand, preparing data in spreadsheet and comparing results by hand. However as we started to perform 10-fold evaluation and we repeated experiments with both Elem2 and C4.5, explored different discretization techniques and different numbers of cut points, it became necessary to automatize some tasks involved in performing experiments and we wrote a couple of programs. These programs themselves are not so interesting. We find much more important the overall architecture, the way in which these programs fit together to help to conduct different experiments relatively easily.

The most important program we have written during the work on this project is Discretize, which performs eOPT discretization, as well as equal width and equal size discretization of continuous attributes. This program was already described in Section 4.

In order to carry out numerous experiments we needed also other auxiliary programs converting data from one format to another, facilitating 10-fold evaluation or summarizing results. These programs are mostly very simple and their roles evolved over time as we were changing the design of experiments. Since these programs are short and simple and since we needed to rewrite their parts as our plans changed, they are not particularly nicely structured or documented. We include them for completeness and document at least their function and usage, if not their inner architecture.

This section is organized as follows. First we explain how a typical experiment was done with our set of programs, including flowcharts depicting a flow of data between the programs. We also discuss some decisions that had to be made during devising experiments. Then we explain usage of individual programs and we explain one of them, called Experiment in detail.

5.1 Organization of an Experiment

Assume that we have an input file containing data and we have chosen some method of discretization and the number of points to be used in discretization. We will now describe how to conduct an experiment with our programs and discuss issues we needed to solve. We will start with most general view of the system and go increasingly into more and more details.

5.1.1 The Most General View

The most general view of experiment is depicted in Figure 2. Our experiments were based on 10-fold evaluation, but we have found out that if we divide input into 10 parts in two different ways, the overall success rates changed considerably. In order to capture the performance more reliably, we decided to do several experiments with the same data set, but with different partition into 10 folds. This was achieved by taking the original input file and creating several input files, each with a different random permutation of lines (records). This was done using program RandomOrder. Each of these files was than processed separately by a 10-fold evaluation (to be described below). Finally we computed the avarage of success rates from all experiments, using our program ForGNU, together with Unix command grep. Grep extracts only lines with relevant information out of many result files in multiple directories and the output of grep is then used as an input for ForGNU. Program ForGNU is able to summarize even results obtained using different discretization techniques and different number of points. It organizes all results according to parameters

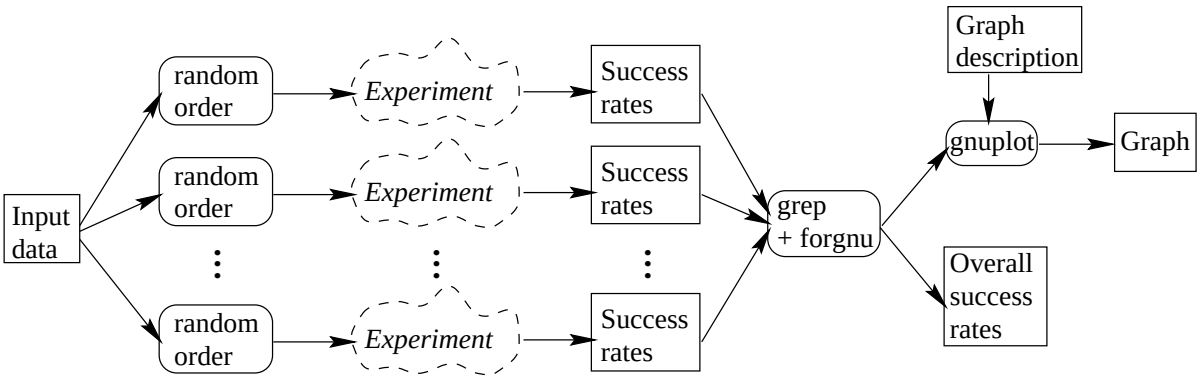

Figure 2: Experiment flowchart - most general level

used in each particular experiment and writes them to separate files. These files are then used as an input for graph-plotting program Gnuplot. In this way when we add new experiment it is easy to rerun ForGNU and Gnuplot to obtain new graphs, without the necessity to load new data manually into spreadsheet. The results produced by ForGNU are in a simple text format and can be also easily used to produce tables and so on.

5.1.2 Performing 10-fold Evaluation

Now we want to perform a 10-fold evaluation on one given data set. Realize, that it would not be correct to discretize entire set in advance, because discretization (at least supervised discretization) uses values of decision attribute and in a real classification situation we do not have values of decision attribute. Therefore we need to discretize only training set and then to use generated set of cut points to discretize the testing set.

The experiment goes as follows (see Figure 3): we start with program Experiment which divides the input data into 10 parts and creates 10 pairs of testing and training data sets.

Now for each data set we run discretization, learning using Elem2 or C4.5 and testing of the resulting rules/decision tree on a testing set. Finally the success rates of individual "folds" of 10-fold evaluation are averaged, producing the final answer for this particular experiment. We have written our own program, called Summary, that summarizes results of 10-fold evaluation for Elem2, using data stored in `.result` files. For results of C4.5 we have used program Average, provided with C4.5 distribution. However this program had to be slightly modified, because it does only a plain average of success rates in individual programs and does not take into account that sizes of testing sets are not neccessarily equal. They may differ by one if the total size of input file is not divisible by 10. We have modified Average so that it weights each success rate with the size of the corresponding testing set. Results of the old and new version of Average differ in the first decimal place for smaller data sets.

5.1.3 One Fold of 10-fold Evaluation

Figures 4 and 5 depict in more detail one fold of 10-fold evaluation. The first call of Experiment produces testing and training data for each fold in a format accepted by Discretize program. Now the training set is discretized using the chosen discretization method and the chosen number of cut points.

There are two possibilities how to supply discretized data to Elem2. Either you replace each value by a symbol, creating true symbolic attribute out of continuous attribute, or you can leave real values as they are and specify only cut points. This second method is in general preferrable because in this case Elem2 can still compare values using operators $\leq$, $\geq$, which is not possible for symbolic attributes. However most of our experiments were done using the first method. The reasons are the following.

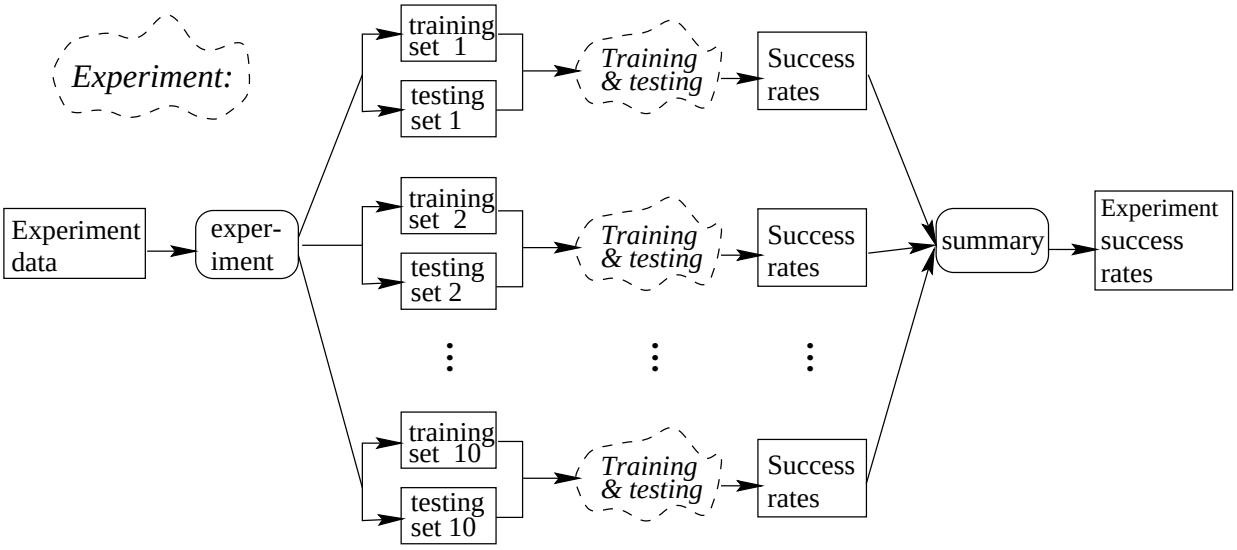

Figure 3: Experiment flowchart - one experiment

- C4.5 does not accept continuous attributes with specified cut points, only a symbolic attribute. If we want to compare C4.5 with Elem2, it is fair to use the same kind of input for both.
- Some data sets contain missing values. Simply skipping records or attributes with missing values would reduce the size of available data drastically. Elem2 does not support missing data and therefore we have replaced each missing piece of data by a special symbol `?` that is included in a list of possible values for each symbolic attribute. Since the overall number of missing values is not too large, it does not influence results too much. Special symbol cannot be included in a continuous attribute. Therefore the continuous attribute needs to be converted to a symbolic attribute first.

  Other idea how to fill in missing values is to use interpolation, since in our particular data sets some attributes represent values as they change in a time series. Thus one can interpolate value in some time point from values of other time points in the time series. However this approach has some drawbacks. First, not all attributes are a part of a time series. Second, in our data sets values of two (or more) attributes cannot be directly compared together. Each attribute corresponds to one biological experiment and with the current technology it is impossible to control experiment conditions so that the values obtained in two experiment would be directly comparable. This is sometimes solved with some kind of normalization of columns, but this method is not enough reliable.

The first kind of discretization, in which continuous attributes are replaced by symbolic, is depicted in Figure 4. After we run Discretize, we obtain training set with real values already replaced by symbols. We also obtain a file describing cut points used in discretization. We also need to convert testing data to symbolic form. This is done by a simple program Discrtst which takes testing data and cut points generated by Discretize and produces testing data with symbolic attributes. Now both testing and training data are converted to formats needed for Elem2 and C4.5. We also need to create files describing individual attributes (`.fmf` file for Elem2, `.names` file for C4.5). This done using program Experiment, this time with a different arguments as in the case of 10-fold evaluation.

Finally, we take these files in appropriate formats and use learning programs to learn rules and to classify testing data. In this way we obtain final results, showing number of successful and unsuccessful attempts for the testing set.

On some data we also performed the second kind of discretization, in which continuous attributes are feeded directly to Elem2, together with cut points generated by Discretize. However here we cannot use C4.5. Other than that the experiment is done in a very similar way as in the first kind of discretization (see Figure 5).

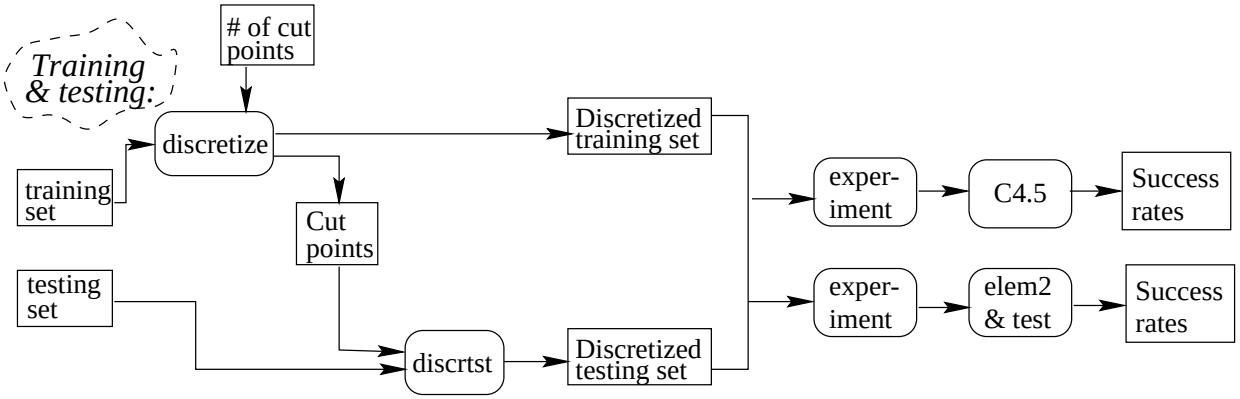

Figure 4: Experiment flowchart - one fold of 10-fold evaluation. Continuous attributes are converted to symbolic.

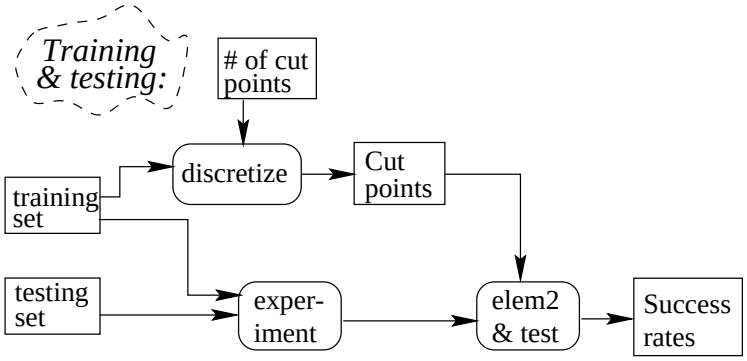

Figure 5: Experiment flowchart - one fold of 10-fold evaluation. Continuous attributes are supplied directly to Elem2 together with cut points.

5.2 Usage of Our Programs

In this subsection we describe function and usage of auxiliary programs used in conducting an experiment.

5.2.1 Program Experiment

Simple program for conversion between different data formats and for creating input of 10-fold evaluation.

Input file: The program reads a file from stardant input. Each line is considered to be one record, empty lines are skipped. Line consists of entries separated by a single space/tabulator/comma. Two consecutive separators denote a missing value. Missing value can be also denoted by question mark. All lines should have the same number of entries, including missing entries. Program handles different end of line charcters (DOS, UNIX).

Output files: The program can output files in a format suitable for program Discretize, for Elem2 and for C4.5.

For purpose of n-fold evaluation the file is randomly permutated and divided into n parts. Then n testing files are created, each containing one of n parts of input. Each testing file has corresponding training file containing the $(n-1)$ parts omitted from testing. In a special case when $n = 1$, all input is written to a training file.

Program also generates `.names`/`.fmf` files for C4.5 and Elem2 containing information about columns. It describes all possible classes as found in the file and also all possible values of symbolic attributes. It applies a simple heuristics for distinguishing between symbolic and continuous attribute – it assumes that symbolic attribute contains at least one decimal point '.' Columns containing at least one period are assumed to be continuos. This heuristics works perfectly for our data. The column containing decision atribute and one optional ignored attributed (such as ID) are given as command-line arguments.

Usage: `experiment <stem> <program> <nflod> <class col> <id col>`

- `<stem>` is the stem of filenames of output files. They will have names `<stem><xx>.<ext>` where `<xx>` is number from 0 to $(n-1)$ and `<ext>` is an extension indicating the type of the file.
- `<program>` determines the format of output. 0 means Discretize, 1 means Elem2 and 2 means C4.5.
- `<class col>` is a number of a column containing class attribute (0 means the first column).
- `<id col>` is a number of a column with ignored (ID) attribute (0 means first column, -1 means ID not available).

Note: program Experiment will be described in more details in Subsection 5.3.

5.2.2 Program Discretize

Usage: `discretize <cutpoints> <points> [<method>]`

- `<cutpoints>` is a filename of a file into which found cut points will be written.
- `<points>` is the number of cut points that have to be generated for each column.
- `<method>` is the method of discretization to be used. 0 means eOPT (default), 1 means equal size discretization, 2 means equal width discretization.

Input file: Input file is read from standard input. It should contain one record in each line with values separated by tabulators. Two consecutive tabulators denote a missing value. The first column should be integer ID (ignored) attribute, the second one should be class attribute (represented by integer) and the other columns contain continuous attributes to be discretized

Output files: The program creates two outputfiles. Discretized data are written to standard output in the same format as was the input file, only with real numbers replaced by symbols $0, 1, 2, \ldots$

The program also creates a file containing cut points for each column. The filename is supplied as a command line option. The file contains on each line cutpoints for one column separated by spaces.

Note: eOPT algorithm for minimizing overall entropy and its implementation is described in detail in Section 4. Other than that the program contains only simple input, output and discretization by equal width and equal size methods. Both these methods are very simple.

5.2.3 Program Discrtst

Program converts real values in an input file into symbolic values provided file with cut points generated by program Discretize. It means all real values are replaced with symbols $0, 1, \ldots$ It is used mainly for discretization of test data sets (training data sets are discretized directly by Discretize).

Usage: `discrtst <cutpoints>`
where `<cutpoints>` is a name of file containing cut points (produced by Discretize).

Input files: Cutpoints are read from file specified on command line. It should contain on each line cutpoints for one attribute sepataed by spaces.

Input file itself is read from standard input. Format should be the same as for Discretize.

Output file: Output is written to the standard output in the same format as input file, only real values are replaced by integers $0, 1, 2...$

5.2.4 Program Randomorder

The program takes an input file, permutes the lines randomly and writes it to output. Random number generator is initialized with current time. Therefore the program generates a different order each time it is called. The program assumes that the input is in the format suitable for discretize, i.e. lines contain records separated by tabs, it handles missing values. First value in a row should be ID, the second one is a class attribute (both integers), the rest of the columns contain continuous attributes. Input is taken from standart input, output is written to standart output.

5.2.5 Program Summary

The program computes the average success rate of ELEM2 for 10-fold evaluatio n.

Usage: `experiment <filestem>`

Input files: Reads files `<filestem>00.result`, ..., `<filestem>09.result`, produced by program Test.exe provided with Elem2. In each of them it locates the table of results at the end of file (giving for each two classes frequency how often the first one was classified as the other one). These frequencies are summed together over all 10 files.

Output files: The result is written to standard output in a human readable form.

5.2.6 Program ForGNU

The program takes as an input overall results of several experiments, computes averages and writes them to separate files, one file for each combination of parameters. The program is written specifically for our directury structure and naming scheme for storing results of individual expreiments.

Input file: Input is read from standard input. It is assumed that the input was created by grep command, which gathers lines containing string "All:" from several result files over several directories. Each line of the input cosists of filename, including path and a line containing summary of results. The overall success rate in per cents is given at the end of each line in parenthesis. This number is extracted. Also we extract parts of filename and path, which describe the parameters of experiment. These parameters include the name of testing data file, number of discretization points, method of discretization and learning program used (Elem2/C4.5). For each combination of parameters there can be multiple values obtained from experiments differening only in permutation of input methods (and therefore in resulting division for n-fold evaluation).

Output files: The program creates several output files with `.dat` extension. Each of them corresponds to one combination of input file, learning program used and discretization method used. Each line of the file contains overall average result for each number of cut points in discretization. These files are a suitable input for GNUplot graph plotting program, but they can be used also for making tables and so on.

5.3 Design of Program Experiment

The program Experiment does not contain any sofisticated algorithms or data structures. Its main advantage is the versatility. Using various switches it can be used to perform different task.

First of all it can take input in several different versions of format. For example entries in one line can be separated by spaces (as in inputs for Elem2), tabulators (as in inputs for Discretize) or commas (as in inputs for C4.5). This was useful when we tried to use inputs from UCI repository, that were separated by commas, whereas uour original inputs were separated by either spaces or tabulators. It also handles both DOS and UNIX end of lines, since we have moved files from one operating system to another. The last important feature is that the column containing class and ID attributes can be specified as a commnad line parameters. This is were handy in case of files from UCI repository, which contained their class and ID attributes at different positions.

Input is read line by line and each line is then dividedinto series of items separated by space. These items are stores as strings, and pointers to these strings are kept in an array. It is very convenient to store items as strings because in this way we can handle even symbolic attributes with non-numeric values. We also do not nned to worry about such things as how many decimal points are needed to write given real number and so on, because all items are written the output exactly in the same form as they was read.

For n-fold evaluation the file needs to be divided into n parts. This is done by assigning number from 0 to $n-1$ to each record cyclically in order in which they appear. In this way size of any two parts differ by at most 1. However if the lines in the input are sorted in some special order, parts may become biased – some of them can contain only walues with some special properties. Therefore before dividing lines into parts we permute them randomly. We use the same seed for random number generator in order to make the same permutation in subsequent runs of Experiment.

The most complex part of the program are output routines, since they have to write the output in different formats. Since all three supprted formats are in principle very similar, we implemented all formats in one code. In some places there are case distinctions where for each format different action is taken, but we have tried to place as many differences as possible into some arrays of constants, so that the code remains relatively simple. We have for axample an array containing suffixes required for each particular format, characters working as separators of items in each format and many constants determining how the `.fmf/.names` file should look like.

5.3.1 Testing Session for Program Experiment

In the left column below we see a simple input file with entries separated by spaces. Column 0 contains class attribute, column 1 contains another symbolic attribute with one missing value, column 2 contains continuous attribute and column 3 contains ID of a record. From this file we want to create inputs for Elem2 and C4.5. The right column shows typescript of a session running Experiment with various settings. Recall the usage of Experiment:

```
experiment <stem> <program> <nflod> <class col> <id col>
<program> is 0 for discretize, 1 for elem2 and 2 for c4.5
<id col> can be -1, i.e.  unspecified
```

```
testinp.txt:
class1␣a␣10␣1
class1␣b␣11.3␣2
class2␣␣10.5␣3
class2␣a␣15␣4
```

```
Script started on Fri Aug 11 20:26:05 2000
bash-2.01$ ./experiment testA 1 3 0 3 <testinp.txt
Warning: not all values are defined.
bash-2.01$ ./experiment testB 1 1 0 -1 <testinp.txt
Warning: not all values are defined.
bash-2.01$ ./experiment testC 2 1 0 3 <testinp.txt
Warning: not all values are defined.
bash-2.01$ ./experiment testD 2 1 0 -1 <testinp.txt
Warning: not all values are defined.
bash-2.01$ exit

Script done on Fri Aug 11 20:26:15 2000
```

The first call of Experiment created inputs for 3-fold evaluation for Elem2. The column 3 was declared to contain ID attributed and was discarded. The class column was moved to the last column for compatibility with C4.5. Missing value was denoted by `?` and included in the list of symbols of a symbolic attribute. This might not be intended behaviour and therefore the program issues a warning. Only one `.fmf` file is shown, all other are the same.

```
testA00.dat:      testA01.dat:      testA02.dat:      testA00.fmf:
b 11.3 class1     a 15 class2       a 15 class2       <C 0 X01 S 3 ? a b>
? 10.5 class2     ? 10.5 class2     b 11.3 class1     <C 0 X02 R>
                  a 10 class1       a 10 class1       <D 0 FUNCTION S 2 class1 class2>
testA00.tst:
a 15 class2       testA01.tst:      testA02.tst:
a 10 class1       b 11.3 class1     ? 10.5 class2
```

Next three calls of input created only 1-fold files, i.e. putting all data to training set and leaving the testing set empty. The first of these three calls created a file for Elem2, but this time we claimed that there is no ID column. Column 3 was therefore included in data as an ordinary symbolic attribute. The second two calls created inputs for C4.5. They always create an ID column – either by preserving the ID column as it appears in the input or by sequentially numbering the lines of the input.

```
testB00.dat:
a 10 1 class1
b 11.3 2 class1
? 10.5 3 class2
a 15 4 class2

testB00.fmf:
<C 0 X01 S 3 ? a b>
<C 0 X02 R>
<C 0 X03 S 5 ? 1 2 3 4>
<D 0 FUNCTION S 2 class1 class2>
```

```
testC00.data:
1,a,10,class1
2,b,11.3,class1
3,?,10.5,class2
4,a,15,class2

testC00.names:
class1,class2.
ID : ignore.
X01 : a,b.
X02 : continuous.
```

```
testD00.data:
0,a,10,1,class1
1,b,11.3,2,class1
2,?,10.5,3,class2
3,a,15,4,class2

testD00.names:
class1,class2.
ID : ignore.
X01 : a,b.
X02 : continuous.
X03 : 1,2,3,4.
```

6 Results of Experiments

6.1 Experiments With Expression Array Data

For our experiments we have used publicly available data that were used in [ESBB98]. This data set contains expression levels of yeast (Saccharomyces cerevisiae). Measurement for 2467 genes over 79 experiments are available. Some subsets of these 79 experiments correspond to measurements in a time series at different

stages of cell live cycle. Other experiments show changes in gene expression level during sudden changes in environment, such as heat shock or cold shock.

For our purposes we have created two subsets of this data sets. The purpose of the **small data set** was to create a subset of genes that do not contain missing values, and that has relatively small number number of classes. This set contains 127 genes and 79 continuous attributes. There are no missing values. The classes (corresponding to gene function) are based on information Saccharomyces Genome Database [SGD]. The data set contains 4 different classes: protein degradation, glycolysis, protein synthesis, and cell cycle. The sizes of these classes are 14, 72, 19, 22.

Large data set was prepared based of information about gene functions taken from MIPS database [MIP]. We have selected all genes that have known function in this database. The resulting set contains 1398 genes, 79 continuous attributes and 12 classes (sizes of classes: minimum 2, maximum 287, average 116). There are missing values.

For experiments we used two learning programs: Elem2 [AC98] and C4.5 [Qui93]. We have run Elem2 with `-p` switch enabling pruning rules to avoid overfitting. Default quality rule measure was used. For discretization we used the original discretization methods built in learning programs, as well as eOPT method and two widely used simple unsupervised methods: equal width intervals and equal size bins. In the further discussion and in graphs we will use short names for these discretization methods, were equal width is called width, equal size bins is called size, original discretization built in learning programs is called C4.5 alone or Elem2 alone and our dynamic programming method is called eOPT.

In all tests we have performed 10-fold evaluation. In experiments with small data set we have also conducted 5 different experiments with different division of data into folds. This could not be done with large data set due to time constraints. Results were averaged. All custom discretization methods were tried for different numbers of cut points. The results of these experiments are shown in simple graphs, where x axis corresponds to the number of cut points and y axis corresponds to the overall success rate in per cent, e.g. what is the percentage of cases where the method determined the correct function of a gene.

The discretized values were passed to C4.5 as symbolic values. Elem2 accepts discretized values as symbolic values (these experiments are referred to as eOPT, size and width) or as real values with specified cut points (referred to as eOPTr, sizer, widthr).

6.1.1 Summary of Experiments on Small Data Set

Figures 6, 7, and 8 show results of our experiments on the small data set. C4.5 performed with the original discretization little better (70.0%) than Elem2 with the original discretization (69.5%).

We have compared performance of Elem2 and C4.5 with/without eOPT discretization. As it can be seen from Figure 6, eOPT discretization does not help to C4.5. However, the situation is different with Elem2. Here eOPT discretization helps. Both eOPT and eOPTr together with Elem2 were able to outperform C4.5.

The difference between Elem2+eOPT (i.e. values passed as symbolic values) and Elem2+eOPTr (i.e. passed cut points) is quite notable. On small values, the performance is almost the same, however with larger number of points eOPT performance quickly drops. This effect is probably caused by the fact, that when passing discretized values as symbolic values, it is impossible to use comparisons $<, \leq, >, \geq$ in building rules. This does not matter with small possible number of values, however, with larger numbers of values the difference is considerable. This is probably also the reason, why C4.5 performs poorly with our custom discretization. This effect can be seen in all performed experiments on all data sets.

Also interesting effect is that the best results are obtained for relatively small number of points. Best performance was experienced for 10 cut points for combination Elem2+eOPTr (71.8%) and almost as good results were obtain even for 1 cut point.

We have also compared our results with commonly used simple unsupervised discretization methods (Figures 7, 8). We were surprised that these methods were able outperform eOPT in both cases (Figure 7 shows result for passing discretized values as symbolic values, 8 show results with specifying cut points).

The reason for such a behaviour can be the fact, that the best performing method (width) takes into account also how large are values. In this way, even small number of important large values can get its own discrete value.

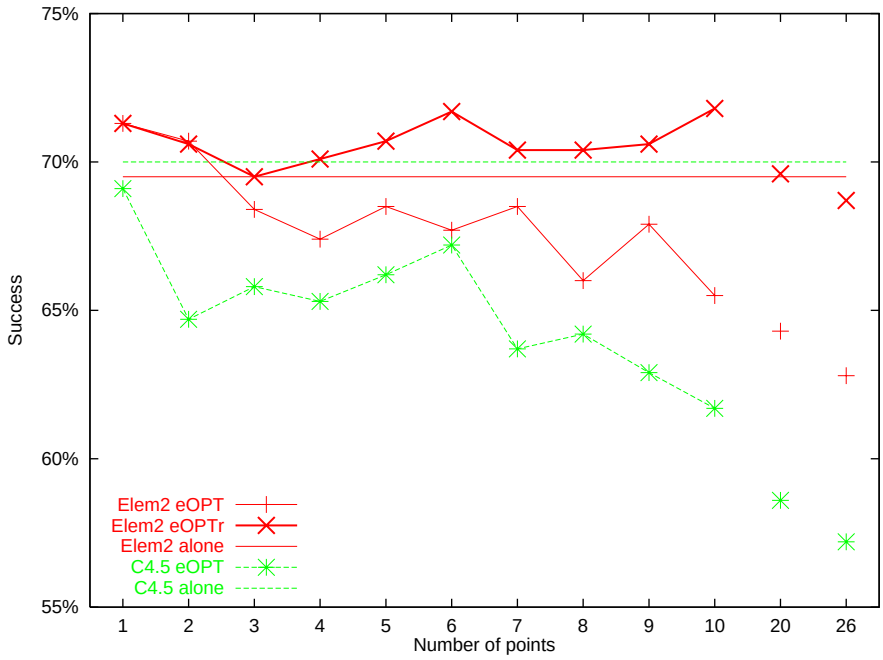

Figure 6: **Small data set:** comparison of Elem2, C4.5 with original and eOPT discretization passed as symbolic values (Elem2/C4.5 + eOPT) and cut points (Elem2+eOPTr).

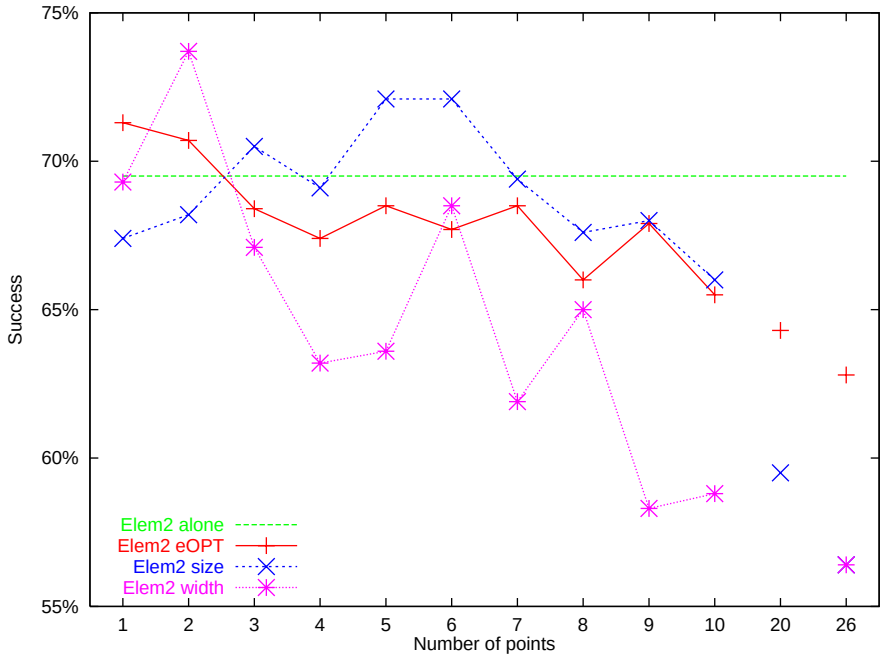

Figure 7: **Small data set:** comparison of Elem2 with different discretization methods (passed as symbolic values).

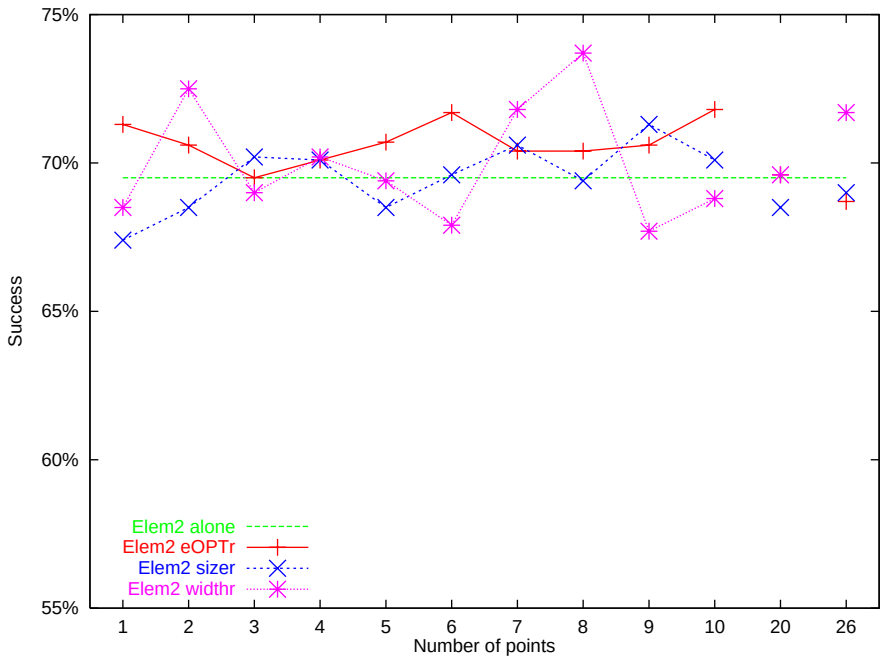

Figure 8: **Small data set:** comparison of Elem2 with different discretization methods (passed as cut points).

6.1.2 Summary of Experiments on Large Data Set

Since missing values are present in the large data set, we could not perform experiments with discretized values specified by cut points. For the same reason we were not able to run Elem2 alone.

In Figure 9 we can see comparison of C4.5, C4.5+eOPT and Elem2+eOPT. Both C4.5+eOPT and Elem2+eOPT perform for small number of points better than C4.5 alone. Performance quickly drops with more number of cut points. In this case (Figure 10) eOPT methods outperforms both simple methods.

6.1.3 Experiments on Data Set Modified with PCA

According to Raychaudhuri et al. [RSA00] principal component analysis (PCA) performs well on expression level data. It reduces dimensionality and concentrates almost all variance found in data into a small number of attributes. This may be usefull for machine learning programs. First of all, it can decrease their running time, because there is smaller number of attributes, and second, it helps them to concentrate on a small number of relevant attributes rather than finding the relevant pattterns among 79 columns.

We have performed PCA on the large data set using Cluster program package [Eis]. We have selected 11 of the new variables with the highest eigenvalue (variables with largest eigenvalue capture the greatest amount of variability). Another advantage of PCA is that the new data set does not contain missing values and therefore it can be directly used in Elem2 and also eOPTr discretization method can be applied. Results are shown in Figure 11.

We see that the best result is achieved with eOPTr discretization in combination with Elem2. This result is even better than the best result achieved at the original large data set (37.5% in PCA data set as opposed to 36.9% in the large data set).

In most other experiments C4.5 performed best with its built-in discretization. If there were any improvements, then usually only small and on a few number of cut points. Surprisingly on this data set the eOPT discretization improves performance of C4.5 considerably over a large range of cut points.

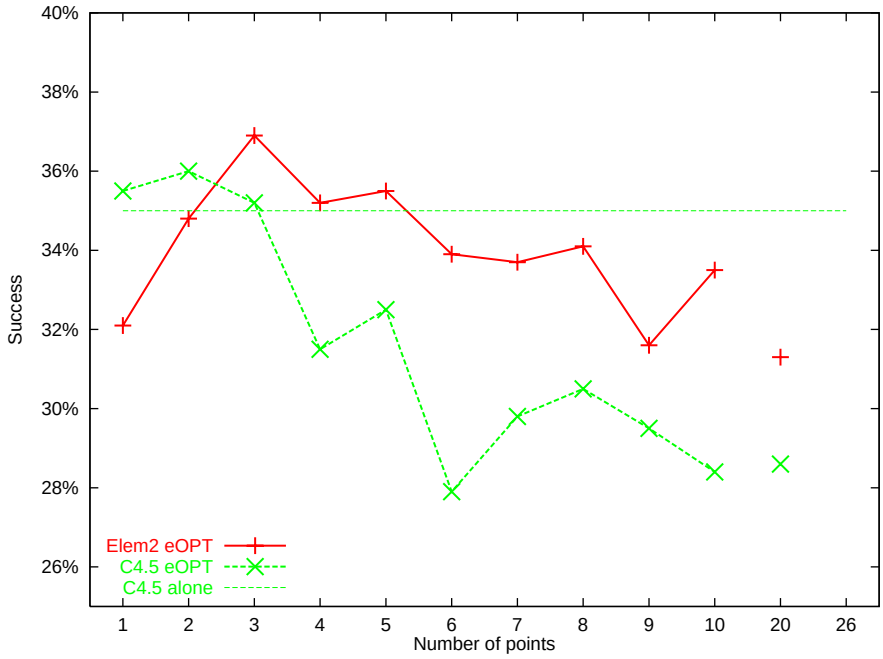

Figure 9: **Large data set:** comparison of C4.5 with original discretization, Elem2+eOPT and C4.5+eOPT.

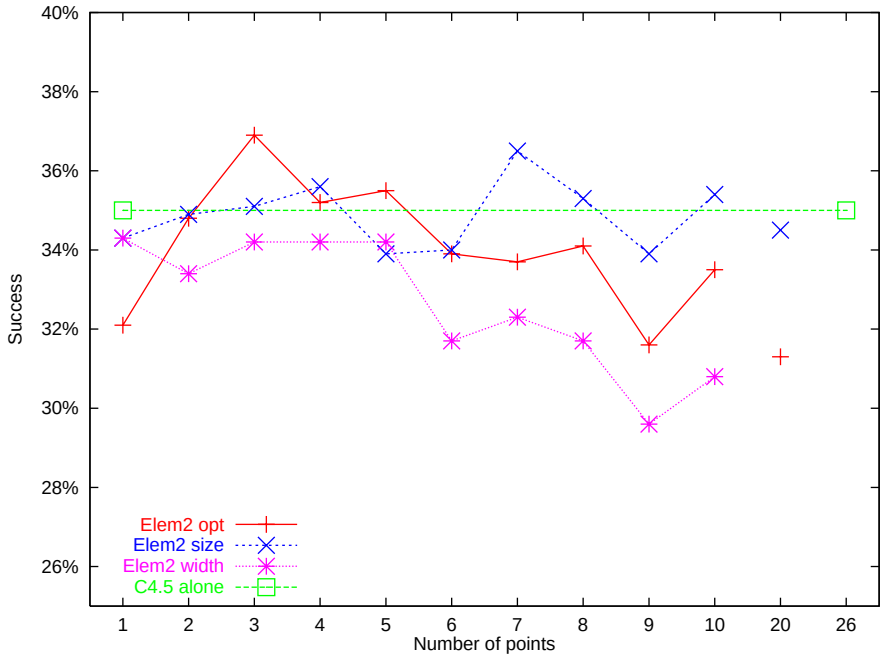

Figure 10: **Large data set:** comparison of Elem2 with different discretization methods (passed as symbolic values).

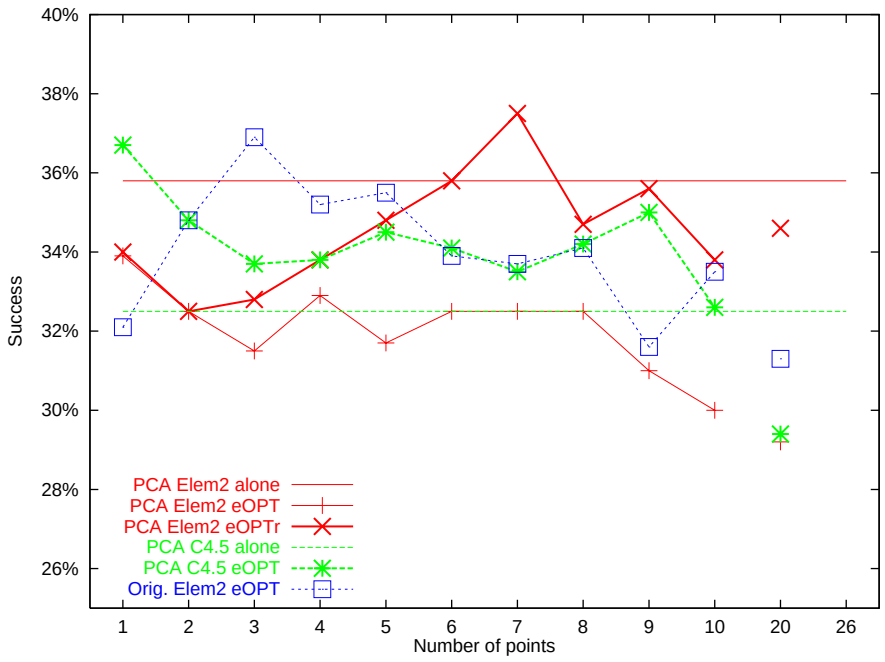

Figure 11: **PCA data set:** comparison of Elem2 and C4.5 on PCA data set. Built in, eOPT and eOPTr discretization methods were used. For comparison Elem2+eOPT on large data set is included.

6.2 Other Experiments

6.2.1 Experiments on Data From UCI Repository

We have performed several tests on four selected data sets from UCI repository (Glass, Wine, Bupa and Iris) to see, how eOPT method performs on other data than expression array data. Data were selected so that they have only continuous attributes.

On Glass data set (see Figure 12) eOPTr method together with Elem2 performed very well. Although Elem2 was outperformed by C4.5 when using original discretization method, Elem2 was able to gain better results using eOPTr.

Elem2+eOPTr also performed well with Wine data set (see Figure 13), however, results were not so good with other two examples (see Figure 14 and 15). Worse results could be caused by the fact, that design of our experiments assumes, that value sets in all variables are of a similar nature, and therefore it make sense to select the same number of cut points for all attributes. This assumption makes sense in expression array data but it is not valid for the last two data sets. In the Bupa data set the number of boundary points of individual attributes ranged from 14 to 75 with median being 56. This big range suggests that the attributes are very heterogenous in their nature. Iris data set had the number of boundary points of the attributes 8,9,19 and 24.

To obtain good results for data sets with heterogenous attributes, it would be probably necessary to design a stopping condition for our method, which can select the optimal number of cut points automatically and independently for every variable.

6.2.2 Unsuccessful experiments

Here we present a short description of other experiments, which we have performed. In design of these experiments were have made wrong assumptions or logical errors. Results of these experiments helped us to realize the error decision and correct the design.

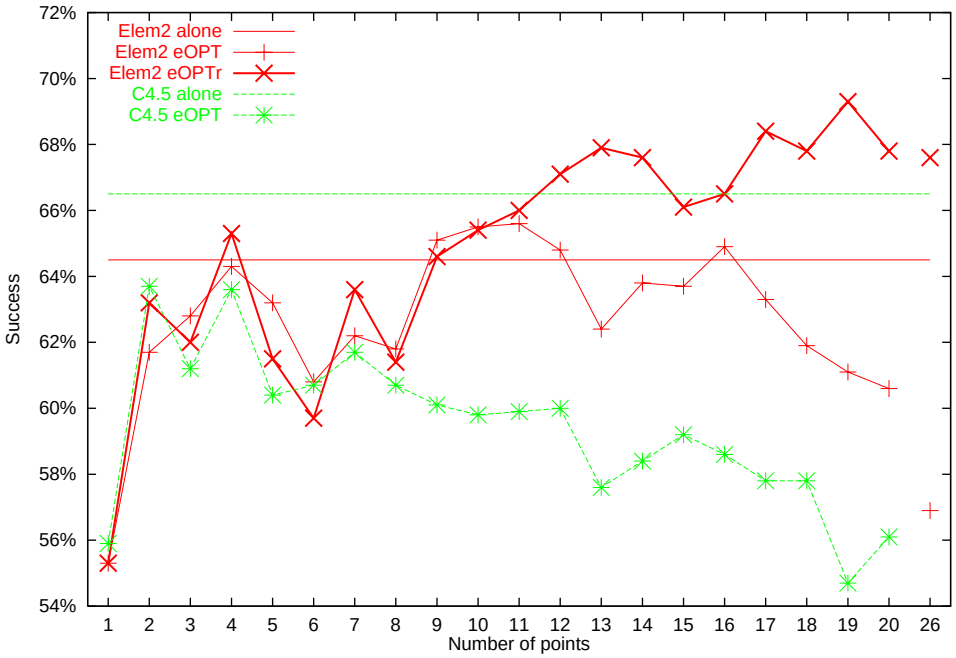

Figure 12: **Glass (UCI repository):** comparison of Elem2, C4.5 with original and eOPT discretization.

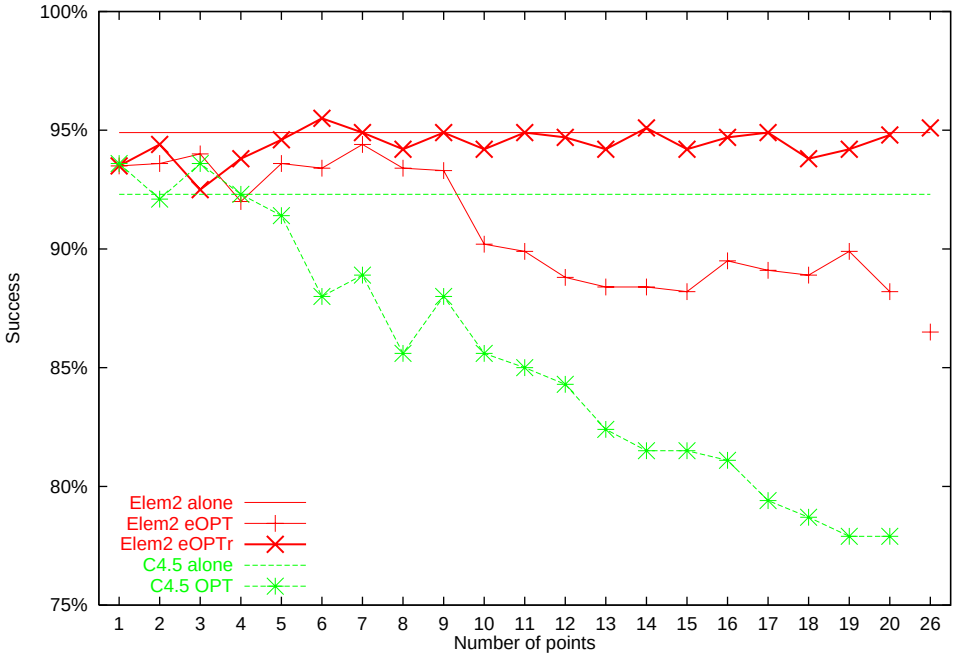

Figure 13: **Wine (UCI repository):** comparison of Elem2, C4.5 with original and eOPT discretization.

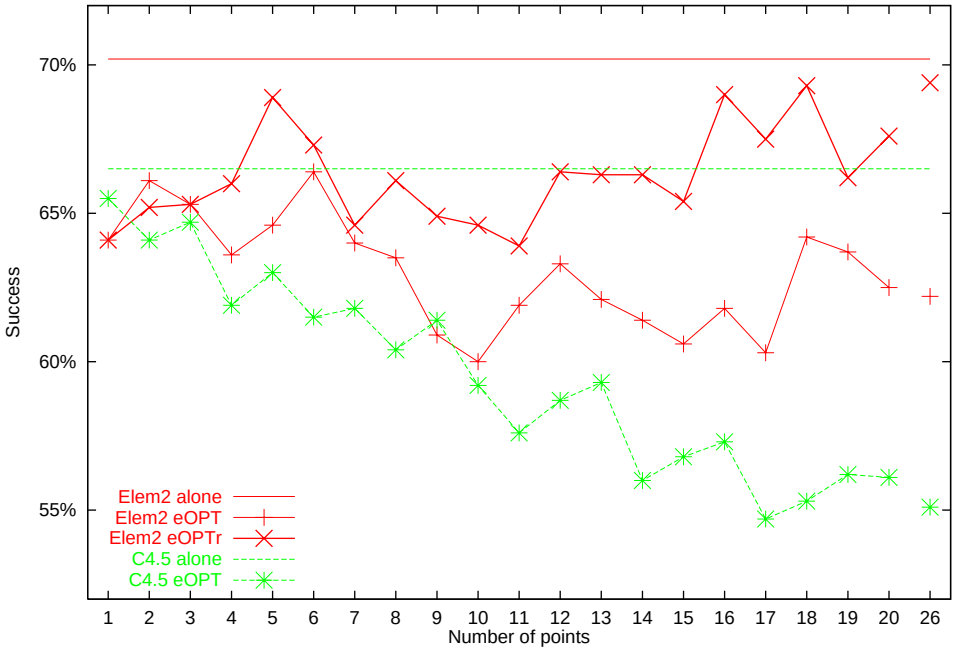

Figure 14: **Bupa (UCI repository):** comparison of Elem2, C4.5 with original and eOPT discretization.

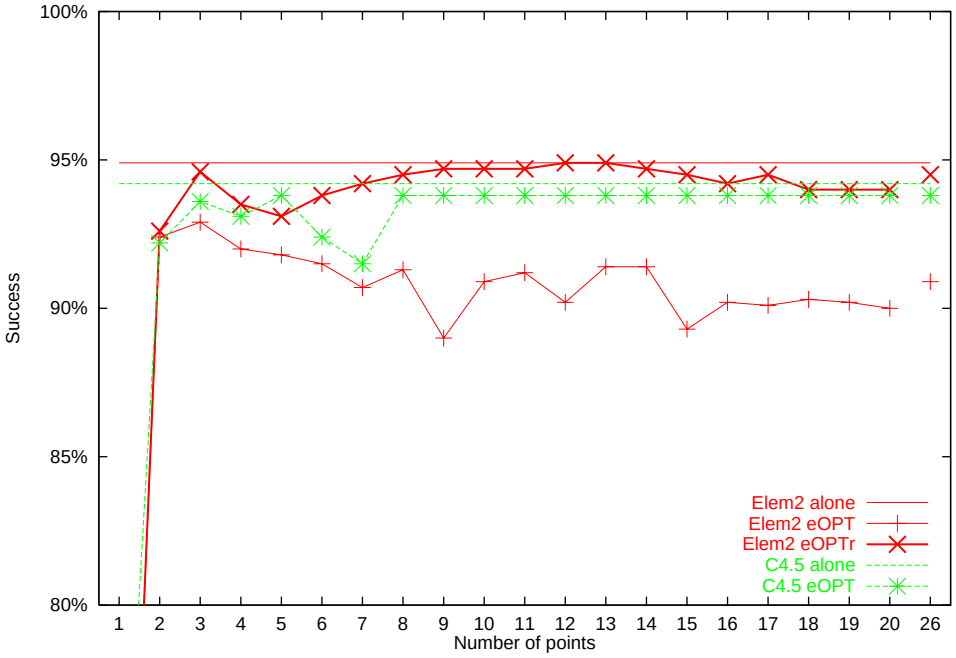

Figure 15: **Iris (UCI repository):** comparison of Elem2, C4.5 with original and eOPT discretization.

Experiment: Perform discretization before splitting data into 10 folds, i.e. on both training and testing data at the same time.

Results: Surprisingly high success values.

Reason: Since we have discretized all values at once, we also provided the class values of the testing set to the discretization method. Since the discretization method is supervised, the information contained in the testing set could influence the inference via our discretization.

Experiment: Perform eOPT discretization on all points, instead of boundary points only.

Results: Unexpected behaviour in some cases.

Reason: Sometimes, when there were several points of the same value, splitting to bins was done incorrectly. The algorithm considered also point inside a stretch of points with the same value as a possible cut point. During the entropy calculation these points with the same value were divided into two groups that could have smaller impurity as the entire group. Therefore the computation of entropy was incorrect and the set of cut points found by the algorithm as having the smallest entropy could have much larger entropy. The problem was corrected together with introduction of boundary points to the implementation.

Experiment: Pass discretized values only as symbolic values.

Results: We expected the results of customized discretization methods for high number of points to be approximately the same, as if we used the discretization algorithms built in the systems. However, success rate dropped significantly instead.

Reason: With symbolic values it is not possible to use $<, \leq, \geq, >$ in rules. This does not matter for small number points, but plays substantial role in the case of high number of cut points. When we passed real values together with the set of cut points, the problem was corrected.

Experiment: In 10-fold evaluation perform one experiment only.

Results: Unstable.

Reason: Results in 10-fold evaluation are usually unstable. After changing partition of data into 10 testing sets we can get success rates several per cents different form the original. The problem can be corrected by performing more experiments with different partitions of data into testing sets and averaging results.

6.3 Experiment Summary

Table 1 summarizes the best success rates achieved for individual discretization methods. We see that the best results for 3 out of 4 UCI data sets were obtained using combination of Elem2 together with eOPTr. Method eOPT also performs well on large expression array data set. We could not use eOPTr on this data set, because it contains missing values. On biological data simple unsupervised methods also perform well. On biological data we see that the best results were for most methods achieved in experiments with small number of cut points.

Data Set	Elem2			C4.5		
	alone	eOPT	other	alone	eOPT	other
Small set (symbolic)	69.5%	71.3% (1)	width **73.7% (2)** size 72.1% (5)	70.0%	69.1% (1)	width 72.7% (4) size 69.3% (6)
Small set (real)	69.5%	71.8% (10)	width **73.7% (8)** size 71.3% (9)	70.0%		
Large set (symbolic)		**36.9% (3)**	width 34.3% (1) size 36.5% (7)	35.0%	36.0% (2)	34.4% (3) size 31.5% (2)
Glass (symbolic)	64.5%	65.6% (11)		**66.5%**	63.7% (12)	
Glass (real)	64.5%	**69.13 (19)**		66.5%		
Wine (symbolic)	**94.9%**	94.4% (7)		92.3%	93.6% (1)	
Wine (real)	94.9%	**95.5% (6)**		92.3%		
Bupa (symbolic)	**70.2%**	66.4% (6)		66.5%	65.5% (1)	
Bupa (real)	**70.2%**	69.4% (26)		66.5%		
Iris (symbolic)	**94.9%**	92.9% (3)		94.5%	93.8% (5)	
Iris (real)	**94.9%**	**94.9% (12)**		94.5%		

Table 1: Comparison of best results obtained using individual discretization methods. For each method we show the best success rate as well as number of cut points where this maximum was achieved (not applicable for built in methods). The number of cut points is given in parenthesis. For each data set there are two lines, one for discretized values passed as symbolic attributes and one for passing cut points.

7 Conclusions

The purpose of the project was to explore possible use of machine learning methods on biological expression array data. We have focused on the problem of gene function classification from expression patterns of genes. Bioinformatics researchers mostly use unsupervised methods (clustering). After clustering is performed, data is processed by hand – biologists try to find clusters, to which they can assign some meaning.

7.1 Machine Learning Point of View

We tried to use automatic supervised methods for this task. We used two classical decision tree based machine learning programs to perform our experiments – C4.5 and Elem2. After initial experiments were performed, we focused our attention to discretization step. Our conjecture was, that since data contain lot of noise, a smaller number of cut points can eliminate some of the noise and therefore increase the learning performance.

We proposed an efficient entropy based eOPT discretization method, which selects a set of cut points to optimize an entropy based formula. The algorithm uses a dynamic programming to find the optimal value, in contrast to approaches taken in discretization implemented in C4.5 and Elem2, that use heuristic for this task. We have compared eOPT method to other commonly used methods – equal width and equal size bins. Our experiments show:

- In case of expression array data it is **profitable to use a small number of cut points**. Our results showed, that for our data best number of cut points was between 5 and 10 cut points. Also notice, that in the most cases even results obtained using only one cut point can be comparable to results with higher number of cut points.
- **Simple methods** (size and width) **perform comparably, and sometimes even better, than sofisticated discretization methods.** Therefore use of these methods in practice is valid. Also, these simple methods are easily understandable and results obtained can be usually easily interpreted. Their advantage is also their low running time.
- Discretized values need to be passed to the learning program **as a continuous values with specified set of cut points**, rather than symbolic values. This does not matter in the case of small number of cut points. However with larger number of cut points, learning algorithm can take huge advantage of

possibility to use $<, \leq, >, \geq$ comparisons. Unfortunately it is impossible to pass cut points in this way to C4.5 program.

- Elem2+eOPTr (eOPT method with passing discretized values as a set of cut points) performed very well and in the most cases this combination outperforms Elem2, as well as C4.5.
- eOPT discretization method did not work very well with C4.5. However, this can be caused by impossibility to pass cut points to the C4.5 method instead of symbolic values.
- Discretization can influence learning process very much. Therefore it is necessary to pay enough attention to that problem.

As we have seen, our work has interesting results. The following problems are possible topis for future research:

- **Stopping condition** for eOPT method needs to be developed. So far, we have tested eOPT method only with the number of cut points specified by the user. Although it is a good feature, automatic selection of the number of cut points is needed.
- We have explored only one possible task, in which machine learning methods can be applied to expression array data. It would be interesting to **apply machine learning methods to other tasks in processing of expression array data**, for example tissue classification (such as cancer class prediction task [GST+99]).
- We have performed some experiments on selected UCI repository data sets. However, design of our experiments did not allow us to perform more such experiments, because we could use only data sets without symbolic attributes. Even in the experiments presented, the performance is affected by the fact that we use the same number of cut points for all attributes. It would be interesting to perform experiments on more UCI repository data. However stopping condition needs to be developed for this task first.

7.2 Bioinformatics Point of View

We have performed experiments with supervised learning methods on expression array data. We have observed several interesting effects:

- Preprocessing method greatly influences learning process. Especially, the number of discrete levels chosen is very important and can have large impact on results. This is in contrast with analysis performed on the data to date, where people do not pay much attention to the disretization process.
- Simple methods (width and size) commonly used in discretization on expression array data can perform as well as more sophisticated methods (eOPT and Elem2 and C4.5 built-in methods, based on entropy measures). Which method can be used in future depends on the performance of a stopping criterion which needs to be developed for these methods.
- The results themselves are not very encouraging. Especially in the case of large data set, classifier with the 37% success rate is not very useful. However, at least class given by the classifier can be used as a working conjecture for the future biological research.

  The problem is that many genes have multiple functions. Therefore it would be helpful, if the classifier could be modified so that it gives multiple possible classes, together with ranking of their plausibility (or even without such ranking). In this case, success rate could be probably much higher.

Ackowledgments

We would like to thank Aijun An for providing Elem2 program for our experiments and for compiling it under Linux platform which simplified our experiments considerably.

References

[AC98] Aijun An and Nick Cercone. ELEM2: A learning system for more accurate classifications. In Robert E. Mercer and Eric Neufeld, editors, *Proceedings of the 12th Biennial Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence (AI-98)*, volume 1418 of *Lecture Notes in Artificial Intelligence*, pages 426–441, Berlin, June 18–20 1998. Springer.

[AC99] Aijun An and Nick Cercone. Discretization of continuous attributes for learning classification rules. In Ning Zhong and Lizhu Zhou, editors, *Proceedings of the 3rd Pacific-Asia Conference on Methodologies for Knowledge Discovery and Data Mining (PAKDD-99)*, volume 1574 of *Lecture Notes in Artificial Intelligence*, pages 509–514, Berlin, April 26–28 1999. Springer.

[AMK99] T. Akutsu, S. Miyano, and S. Kuhara. Identification of genetic networks from a small number of gene expression patterns under the boolean network model. In Russ B. Altman, Kevin Lauderdale, A. Keith Dunker, Lawrence Hunter, and Teri E. Klein, editors, *Pacific Symposium on Biocomputing 4*, pages 17–28, Mauna Lani, Hawaii, 4–9 January 1999. World Scientific.

[BK00] A. J. Butte and I. S. Kohane. Mutual information relevance networks: Functional genomic clustering using pairwise entropy measurements. In Russ B. Altman, A. Keith Dunker, and Lawrence Hunter, editors, *Pacific Symposium on Biocomputing 5*, pages 415–426, Honolulu, Hawaii, 4–9 January 2000. World Scientific.

[DKS95] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In Armand Preiditis and Stuart Rusell, editors, *Proceedings of the 12th International Conference on Machine Learning*, pages 194–202. Morgan Kaufmann, 1995.

[Eis] Michael Eisen. Cluster and TreeView: pair of programs for analyzing and visualizing the results of complex microarray experiments. Available on-line at `http://rana.stanford.edu/software/`.

[ER99] Tapio Elomaa and Juho Rousu. General and efficient multisplitting of numerical attributes. *Machine Learning*, 36(3):201–244, 1999.

[ESBB98] Michael B. Eisen, Paul T. Spellman, Patrick O. Brown, and David Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of National Academy of Sciences of USA*, 95:14863–14868, December 1998. Accompanying data available from `http://rana.stanford.edu/clustering`.

[FI92] Usama M. Fayyad and Keki B. Irani. On the handling of continuous-valued attributes for classification learning. *Machine Learning*, (8):87–102, 1992.

[FI93] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1022–1027. Morgan Kaufmann, 1993.

[FKS95] Truxton Fulton, Simon Kasif, and Steven Salzberg. Efficient algorithms for finding multi-way splits for decision trees. In Armand Preiditis and Stuart Rusell, editors, *Proceedings of the 12th International Conference on Machine Learning*, pages 244–251. Morgan Kaufmann, 1995.

[GST+99] T.R. Golub, D. K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. P. Mesirov, H. Coller, M. L. Loh, J. R. Downing, M. A Caligiuri, C. D. Bloomfield, and E. S. Lander. Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring. *Science*, 286, October 1999.

[MIP] MIPS: Saccharomyces cerevisiae – the yeast genome. Database available on-line at `http://www.mips.biochem.mpg.de/proj/yeast/`.

[PSG+00] N. Pollet, H. Schmidt, V. Gawantka, C. Niehrs, and M. Vingron. In silico analysis of gene expression patterns during early development of xenopus laevis. In Russ B. Altman, A. Keith Dunker, and Lawrence Hunter, editors, *Pacific Symposium on Biocomputing 5*, pages 440–451, Honolulu, Hawaii, 4–9 January 2000. World Scientific.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California, 1993.

[RSA00] Soumya Raychaudhuri, Joshua M. Stuart, and Russ B. Altman. Principal component analysis to summarize microarray experiments: Application to sporulation time series. In Russ B. Altman, A. Keith Dunker, and Lawrence Hunter, editors, *Pacific Symposium on Biocomputing 5*, pages 452–463, Honolulu, Hawaii, 4–9 January 2000. World Scientific.

[SGD] SGD: Saccharomyces genome database. Database available on-line at `http://genome-www.stanford.edu/Saccharomyces/`.

[ZZ00] J. Zhu and M. Q. Zhang. Cluster, function and promoter: Analysis of yeast expression array. In Russ B. Altman, A. Keith Dunker, and Lawrence Hunter, editors, *Pacific Symposium on Biocomputing 5*, pages 476–487, Honolulu, Hawaii, 4–9 January 2000. World Scientific.