Application of Genetic Programming to Induction of Linear Classification Trees

Martijn Bot

April 19, 1999

Final term project report

vrije Universiteit
Faculteit der Wiskunde en Informatica
Studierichting Bedrijfswiskunde & Informatica
De Boelelaan 1081a
1081 HV Amsterdan

Work conducted at:
Centrum voor Wiskunde en Informatica
Kruislaan 413
1098 SJ Amsterdam

Foreword

In the last semester of the study of Business Mathematics and Informatics at the *vrije* Universiteit a final term project has to be performed outside of the university. This paper is the report of my project.
The project was a research project at the CWI in Amsterdam, in the research group Software Engineering 4, which is concerned with Evolutionary Algorithms and Applied Algorithmics. Specifically, I applied Genetic Programming to the problem of inducing linear classifiers from data.
I would like to thank dr. W.B. Langdon, dr. Arno Siebes and dr. Wojtek Kowalczyk for their invaluable assistance and critiques.

Abstract

...

Contents

1 Introduction **1**

2 Background **2**
2.1 Genetic Programming . . . 2
2.2 Strong typing . . . 3
2.3 System . . . 3
2.3.1 Selection method . . . 4
2.4 Special techniques . . . 5
2.4.1 Avoiding bloat . . . 5
2.4.2 Limited Error Fitness . . . 6
2.4.3 Pareto Scoring with Fitness Sharing . . . 6

3 Experiments **9**
3.1 Settings . . . 9
3.2 Databases . . . 9
3.2.1 Artificial databases . . . 10
3.2.2 MLR-databases . . . 11
3.3 Mutation rate . . . 12
3.4 Avoiding bloat . . . 12
3.5 Limited Error Fitness . . . 14
3.6 Pareto scoring . . . 17
3.6.1 Pareto scoring with Fitness Sharing . . . 17
3.6.2 Results . . . 18
3.6.3 Pareto scoring with domination-based ranking . . . 20
3.6.4 Results . . . 21
3.6.5 Paretofronts . . . 22
3.7 Treesizes . . . 22
3.8 Execution speed . . . 24
3.9 Effects of noise . . . 26

4 Conclusions **28**
4.1 Bloat . . . 28
4.2 Accuracy . . . 28
4.3 Performance . . . 28

Chapter 1

Introduction

In many databases, some classification can be made of the tuples. For example, a bank can classify its customers in good and bad clients and a medical doctor can classify (diagnose) his patients in patients having a certain illnes and patients not having this illness. When a classification is a linear combination of (some of) the available variables, it is called a linear classifier.

In this paper I apply Genetic Programming [Koza, 1992] to the problem of inducing such linear classifiers from data. I will compare the classification accuracy with that achieved by several other techniques, such as Discriminant Analysis, the OC1-algorithm and the Stroganoff system.

This research took place at the "Centrum voor Wiskunde en Informatica", CWI.
CWI is the national research institute for Mathematics and Computer Science in the Netherlands. It's mission is twofold:

- to perform frontier research in mathematics and computer science;
- to transfer new knowledge in these fields to society in general and trade and industry in particular

CWI is funded for 70 percent by NWO, the National Organisation for Scientific Research. The remaining 30 percent is obtained through national and international programmes and contract research commissioned by industry.

In chapter 2, some theoretical background will be given on Genetic Programming as I applied it and on the different techniques I tried. Chapter 3 contains the results of the experiments. Chapter 4 gives the conclusions.

Chapter 2

Background

2.1 Genetic Programming

Genetic Programming (GP) is a technique based on biological evolution. It enables computers to solve problems without being explicitly programmed. Randomly, a number of individual programs is generated which try to solve some problem. They will all probably perform very badly (since they were generated at random), but still by chance some will perform better than others. The individuals with the best performance get the largest chance of reproduction and the worst individuals will be removed from the population.
When reproducing, mutation or crossover may occur. Mutation means that part of the program is removed and a new part is generated and inserted at that point. Crossover occurs between two individuals. In each individual, a crossover point is randomly chosen and the corresponding parts of the two individuals are swapped. (See figure 2.3.)
After reproduction, the individuals in the new generation are evaluated and the process starts over again. In this way each generation will (hopefully) get slightly closer to an optimal solution.

Performance is measured by a fitness function f, which assigns a fitness score to each individual. The best programs get lowest fitness (closest to 0). Selection for reproduction is often done proportional to fitness score:

$$\mathcal{P}(\text{individual } x_i \text{ is selected}) = \frac{f(x_i)}{\sum_{j=1}^{M} f(x_j)}$$

with x_i individual i in the population and M the size of the population.

The programs have the form of a parse tree. Nodes in the tree are procedures, functions, variables and constants. The subtrees of a node represent the arguments to the procedure or function of that node. (Since variables and constants don't take arguments, they're always leaves in the tree.) The leaves are called terminals, the inner nodes functions.
An example is the following program, which is supposed to classify all customers of a bank into good or bad clients:

```
if income > $100,000
    the client is a good client
else
    the client is a bad client
```

The parse tree of this program looks like figure 2.1.
There are two functions in this tree:

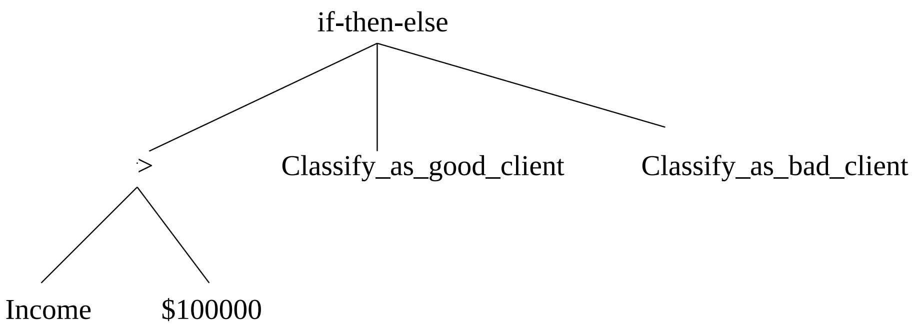

Figure 2.1: Parse tree

- `if-then-else` takes 3 parameters (the condition and the procedures to perform if the condition is true resp. false)
- `>` takes 2 parameters and checks if the first is greater than the second. It returns true or false, so it can be used in the condition-childnode of the `if-then-else` node.

`Income` is a variable the value of which can be found in the database for each client. 100000 is a constant. `Classify_as_good_client` and `Classify_as_bad_client` are procedures that give a final classification.
Since the `if-then-else` function can take another `if-then-else` function as its first child and all variables and constants can occur, arbitrarily complicated classifications can be made.
The fitness function simply counts the number of errors that are made in this classification, i.e. the number of misclassified customers in the database.

2.2 Strong typing

The `if-then-else` function needs a first parameter which returns true or false and two other parameters which are procedures. If the tree is generated completely at random, there is no guarantee that this will always be the case. For this reason, I will use the concept of Strong Typing [Montana, 1995].

Strong Typing is a way to guarantee correct parse trees. It allows functions to specify of which datatype each parameter must be and which datatype it returns itself. When generating a random tree, only nodes of the correct datatype are inserted at each child node. In the example above, `if-then-else` node takes a boolean as first parameter and voids (procedures that don't return any value) as second and third parameter. It returns void.

2.3 System

I used the system GPsys by Qureshi [Qureshi, 1997], which is a Strongly Typed, steady state GP-system implemented in Java. The advantage of this system over other systems is that it's much more easily extendible, due to its clear object-oriented structure and the fact that it's written in Java.

My Strong Typing system includes three datatypes:

- `Variable` (contains an integer, returns a double)
- `Constant` (contains and returns a double)

- `Classification` (returns an integer).

`Variable` is an integer which takes values in the range [0, `NR_VARIABLES_IN_DATABASE−1`]. It represents the number of a variable in the database. When evaluated, it looks up the value in the database and returns it as a double. It can only be used as a terminal.
`Constant` is a double within a certain range, for example [`MIN_VALUE_IN_DATABASE`, `MAX_VALUE_IN_DATABASE`]. It too can only be used as a terminal.
`Classification` can be either a terminal or a function. There's one terminal that returns this datatype which just contains an integer value in the range [0, `MAX_CLASSIFICATIONS−1`]. All functions also have this return type. The functions are called `CheckCondition1Var`, `CheckCondition2Vars`, etc. For each variable in their name, they have 2 children: a `Constant` and a `Variable`. After those comes one more `Constant` and two `Classifications`. `CheckCondition2Vars` is shown graphically in figure 2.2.

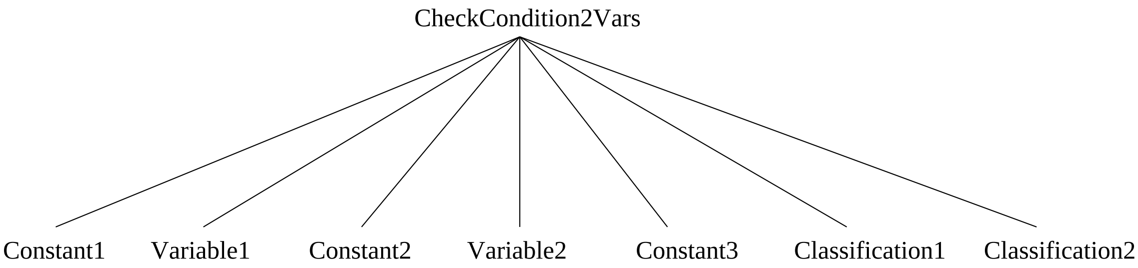

Figure 2.2: CheckCondition2Vars function

Its meaning is the following:

```
if Constant1 * Variable1 + Constant2 * Variable2 <= Constant3
    return Classification1
else
    return Classification2
```

The two `Classification` children can be either a `Classification` terminal or a `Classification` function again.

2.3.1 Selection method

GPsys implements a tournament selection. This means that a number of individuals are randomly selected from the population. Each individual is scored by the fitness function and the tournament is ranked accordingly.
With some probability p_{mut}, mutation is chosen and with chance $1 - p_{mut}$, crossover is chosen. If it is mutation, the best individual from the selection is copied into the slot of the worst individual. Then this copy is mutated randomly (a node is chosen at random, deleted and a new subtree is generated at that point). If crossover was chosen, the two best individuals from the tournament sexually reproduce: a subtree from the "father" is chosen at random, a node from the "mother" is chosen at random and the subtree at that node is replaced with the subtree from the father. See figure 2.3. The worst individual in the tournament is overwritten with the new child. In this way, the best individuals are always kept in the population. (In some systems, the mutated individual overwrites its parent instead of the worst individual. In this system, best performance never decreases over time, contrary to those systems.)

The advantage of this scheme is that later on in evolution, when many individuals are close to another in fitness score, there's still a big advantage of being slightly better than the others. If reproduction probability is proportional to fitness score and no tournaments are used, the

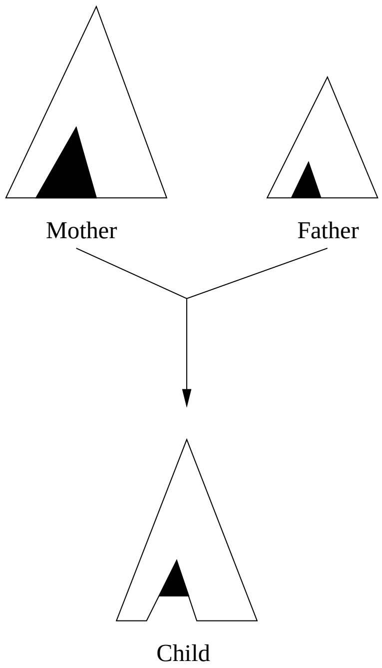

Figure 2.3: Crossover

pressure on becoming even better is very small later in evolution. With tournament selection, there's always such pressure.

The random choice of a node for mutation or crossover is implemented as follows. Starting at the root, an unweighed coin is "thrown" to see if we should go a level deeper. If not, the current node is returned. Otherwise, one of the children is randomly chosen, the coin is thrown again and the process starts over again (until we reach a terminal). In this way, the inner nodes have the highest chance of getting chosen.

2.4 Special techniques

In this section I'll describe some of the techniques I've tried in order to improve performance of the GP.

2.4.1 Avoiding bloat

In GP (and other fitness based search techniques with a variable length representation), individuals tend to become larger over time [Koza, 1992] [Blickle and Thiele, 1994] [Nordin and Banzhaf, 1995] [McPhee and Miller, 1995] [Soule *et al.*, 1996] [Greeff and Aldrich, 1997] [Soule, 1998]. Being large has an advantage, because chances of destructive crossover are smaller.
An individual may have some subtrees that don't do anything, for example

```
if x3 <= 5.2
    classification = 0
else
    classification = 0
```

Such subtrees can protect from destructive crossover, because if there are many of them, the chance of one of them being chosen for crossover is high and if indeed such a subtree is chosen

for crossover, performance of the individual won't be affected by the crossover. So individuals which contain large amounts of such inoperative code are less likely to have damaged offspring and therefore have an evolutionary advantage.
Disadvantages of large individuals are more executing time for fitness measuring and lower understandability for humans.

There are different ways to avoid this problem. A common way is to put a penalty on being large. The fitness function would become like this:

$$\text{Fitness} = \#\text{Errors} + a \cdot \#\text{Size of tree}$$

In this formula, a is a value which sets the relative importance of the penalty in comparison to the number of errors. Possible values include 0.1, 0.25, 0.5, 1, etc. The size of the tree could be measured by the number of nodes or by the depth of the tree.

2.4.2 Limited Error Fitness

Limited Error Fitness (LEF) [Gathercole and Ross, 1997] is a modification to the standard supervised learning in GP. An error limit is introduced. When evaluating an individual, if the number of errors it has made so far is higher than the error limit, all remaining cases are counted as errors. The fitness score is still the total number of errors. In this way, poor individuals will not be evaluated on the entire training set, thereby saving CPU time. Good invididuals, making fewer errors than the error limit, will still be evaluated on the entire training set.
The error limit is modified over time, depending on the number of errors the best individual makes and the number of generations no improvement has been made on the best individual. If no improvement has been made for some generations, the error limit is raised (lowered) if the best individual scores more (fewer) errors than the error limit. Also, the easiest training case(s) (the cases that most individuals classify correctly) is moved to the end of the training set.

In this way, the population is kept in a constant flux instead of converging early. Once the best individual stops improving for some time, the fitness function is adjusted by altering the error limit and re-ordering the training set. The most difficult cases are emphasized at the cost of the easier ones.
The time saved by evaluating fewer cases is mostly spent again, because LEF often needs more generations to find the best solution. So, on average, LEF runs take about as long as runs without LEF [Gathercole and Ross, 1997].

2.4.3 Pareto Scoring with Fitness Sharing

Pareto Scoring [Goldberg, 1989] is a way of combining multiple goals in the fitness function. For example, a car should be as beautiful, cheap, fast and economical as possible. Obviously, not all these criteria can be fulfilled at the same time, so some tradeoff has to be made.
One way of doing this could be a linear combination of the different goals. The problem with this is the settings of the weights: what's more important (and how much): beauty or price?
Pareto Scoring allows different combinations of the criteria. If an individual is no worse than another on every dimension, and better in at least one dimension, this individual is said to dominate the other. For example in figure 2.4, both point 1 and point 2 dominate point 3, but point 1 and point 2 don't dominate each other (assuming smaller is better). The Pareto optimal front is the set of points that aren't dominated by any other point.
Now invididuals can be scored on the number of other individuals they dominate: the more, the better. In this way, some individuals will be better in terms of price, others in economy, etc., but no individual on the Pareto optimal front will be worse than another in all respects. Finally, a human will have to make the decision which of the cars to choose.

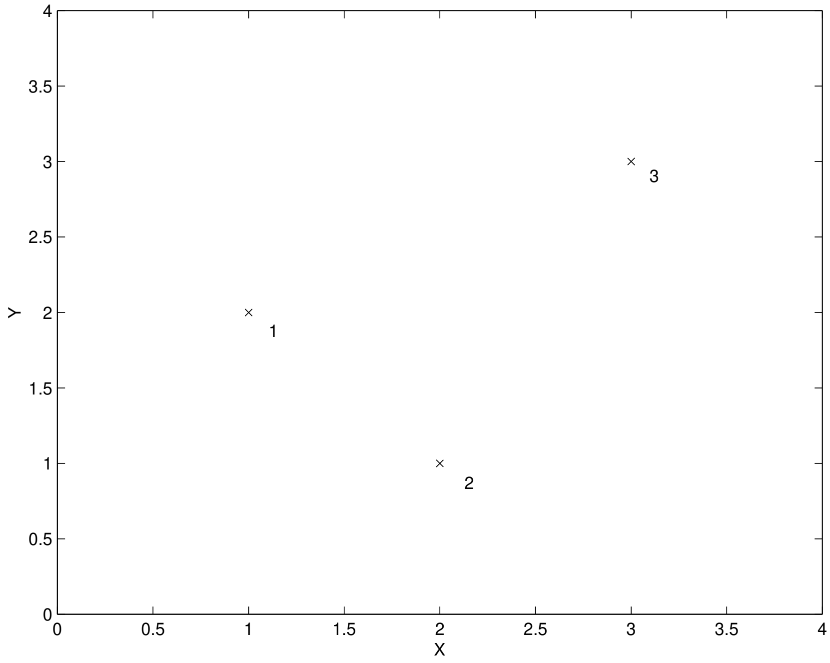

Figure 2.4: Points in Pareto score calculation

Formally, a multi-objective optimization problem is formulated a follows:

$$\min_{x \in F} f_1(x), f_2(x), \ldots, f_M(x)$$

where $f_1(x), f_2(x), \ldots, f_M(x)$ are the objective functions to be minimized simultaneously, x is the decision variable and F is the feasible region. Solution x_P is on the Pareto optimal front if there exists no $x \in F$ which satisfies

$$\forall m \in 1, \ldots, M : f_m(x) < f_m(x_P)$$

When Pareto Scoring is used in this way, the GP tends to converge to one (suboptimal) value. Fitness Sharing [Goldberg, 1989] is a way to ensure that different local minima are found. It works by putting a penalty on individuals that have many “neighbours”, other individuals that are nearly the same. For a given individual, the degree of sharing is determined by the summing the sharing function values contributed by all other individuals in the population (or, for reasons of efficiency, a sample of it). The closer the individidual, the higher (closer to 1) the degree of sharing. The shape of the sharing function is often triangular, with value 0 for farther away individuals.
For each non-dominated individual, the sharing sum is calculated. The one with the lowest sum is the best among them and will be allowed to reproduce.
In my system, I'll apply Pareto scoring with the following two dimensions:

- Number of errors (the smaller, the better)
- Number of nodes in the tree (the smaller, the better)

By using Fitness Sharing, I hope to get a diverse population with both small trees and good scoring trees. In the end, the individual which scores fewest errors is returned. Hopefully, this will also be a small tree. Presumably, if such small and accurate a tree exists and is not too hard to find, the Pareto system will find it.

In the literature, genotype-based Pareto scoring and phenotype-based Pareto scoring are distinguished. Genotype-based Pareto scoring works by looking at the "genes", i.e. the internal representation. Phenotype-based Pareto scoring looks at the solutions produced by the system and compares various dimensions of the solutions.
My system is a combination of the two, because one dimension is genotypic (the number of nodes) and the other is phenotypic (the number of errors). In the literature (REFERENCE???), phenotypic Pareto scoring seems to work best. It may be interesting to see how this combination works.

Another way of fitness sharing (REFERENCE???) is by counting the number of individuals in the population (or the sample) that dominate or are equal in size and errors to each of the non-dominated individuals in the tournament. The one which is dominated by or equal to the smallest number of individuals wins the tournament and gets to reproduce. This is also a form of niching, since if there many individuals which score the same on all dimensions, they will get a bad score. So, there's a diversive pressure on the population, causing the population to spread out over the Pareto front.

Chapter 3

Experiments

In this chapter I'll give an overview of the experiments that were performed. In the first section, the settings that were used in most experiments are given. Later on, when other settings were used, this is always explicitly noted. In the second section, the databases that were used in the experiments are described. In the following sections the different experiments are described along with their results.

3.1 Settings

In table 3.1, a Koza-style tableau is given with the standard settings for the experiments.

Objective	Classify all training cases correctly
Terminal set	Variable, Constant, Classification
Function set	CheckCondition1Var {,CheckCondition2Vars, ...}
Fitness Cases	Various databases
Fitness Scaling	1.0 for each error made
Selection	Tournament of size 7
Hits	n/a
Wrapper	n/a
Parameters	Population size = 250, Nr of runs = 25, G=1000
Success Predicate	All cases correctly classified

Table 3.1: Tableau

When crossvalidating, I do 10-fold crossvalidation. In each fold, I do 4 runs. Of those 4 runs, the best one is taken. These best runs are averaged.

3.2 Databases

In order to test the GP-algorithm, I used several databases. Some, I created myself, others were taken from the Machine Learning Repository[1]. In the following paragraphs a description is given of these databases.

[1] `http://www/ics.uci.edu/~mlearn/MLRepository.html`

3.2.1 Artificial databases

Axis-parallel tree with 6 variables, 3 dummies

A database of 200 tuples was created with 6 variables. Classification was done according to the axis-parallel tree in figure 3.1. With a chance of 10%, the classification was switched. This resulted in 15 switches (due to the randomness of the switching).

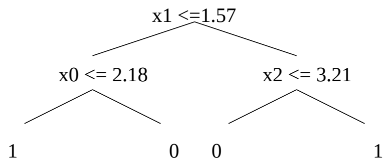

Figure 3.1: Axis-parallel tree with 3 variables

In figure 3.2 a plot is drawn in which the two planes are drawn with all points. The points with the x inside and connected by lines are the switches. Both the points and the switches are well distributed across the whole space.

Figure 3.2: Graphical display of the tree in figure 3.1

Axis-parallel tree with 100 variables, 95 dummies

A database of 1000 tuples was created with 100 variables, 95 of which were dummies. The tree is shown in figure 3.3. There were 61 switches.

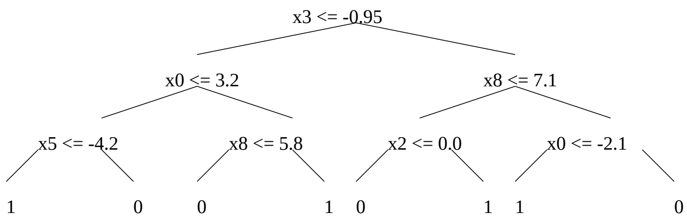

Figure 3.3: Axis-parallel tree with 5 variables

Oblique tree with 100 variables, 97 dummies

A database of 1000 tuples was created with 100 variables, 97 of which were dummies. The tree is shown in figure 3.4. There were 52 switches.
A validation file was made in the same way with 45 switches.

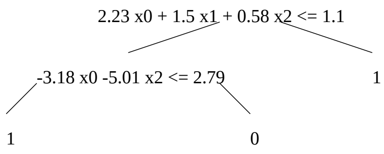

Figure 3.4: Oblique tree with 3 variables

In figure 3.5 a plot is drawn in which the two planes are drawn with all points. The points with the x inside and connected by lines are the switches. Both the points and the switches are well distributed across the whole space.

Figure 3.5: Graphical display of the tree in figure 3.4

Oblique tree with 100 variables, 93 dummies

A database of 1000 tuples was created with 100 variables, 93 of which were dummies. The tree is shown in figure 3.6. There were 61 switches.
A validation file was made in the same way with 51 switches.

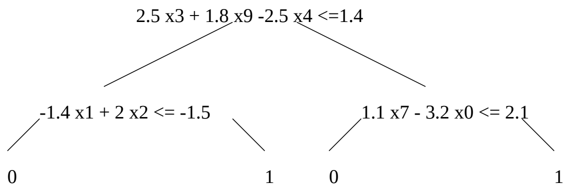

Figure 3.6: Oblique tree with 7 variables

3.2.2 MLR-databases

Of each database, a short description is given. For more details, see the website.

Glass

The glass database contains 214 instances of 9 continous variables each plus a classification (the type of glass). There are 7 classes (1...7), one of which (4) isn't used. I randomly split the database in a training set of 120 cases and a validation set of 94 cases.

Ionosphere

The ionosphere database contains 351 instances of 34 continous variables plus a class attribute. Classification is binary, in good and bad cases. The database was split in a training set of 175 cases and a validation set of 176 cases.

Pima

The Pima database contains 768 instances of 8 continous variables plus a class attribute. Classification is binary, either the Pima-Indians are tested positive of negative for diabetes. The database was split in a training set of 384 cases and a validation set of 384 cases.

Segmentation

The training database consists of 210 and the validation database of 2100 cases. There are 19 attributes plus a class variable. There are 7 classes.

3.3 Mutation rate

An important parameter is the mutation rate p_{mut}. In many GP-problems, it is set very low. 0.01 is not uncommon. Using the Pima dataset with LEF and Fitness Sharing Pareto (see paragraphs 3.5 and 3.6.1), I did runs with different settings for p_{mut}. The results are shown in figure 3.7. Average performance of the best individual of each of the 25 runs if averaged and a 95% confidence interval is given.
Clearly, the mutation rate should not be set at 0 or 1. One reason for this could be that finding the correct constants requires some mutation, because if p_{mut} is close to 0, only the constants that were generated in the initial population generation could be used.
Between $p_{mut} = 0.1$ and $p_{mut} = 0.9$ the differences are very small (not even nearly significant). The only significant differences are between $p = 0.0$ and $p = 0.1 \ldots 0.9$ and between $p = 1.0$ and $p = 0.1 \ldots 0.9$.
It seems that the exact amount of mutation and crossover doesn't matter, but both should be present. In the following experiments I will use $p_{mut} = 0.5$, because since the fractions between 0.1 and 0.9 don't produce statistically different results, a value in the middle seemed logical.

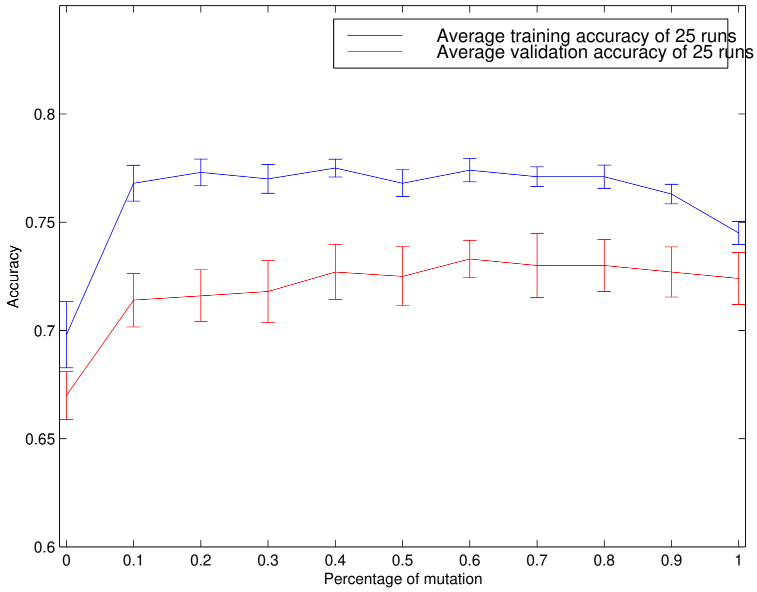

Figure 3.7: Mutation rate

3.4 Avoiding bloat

Treesize can be very large when there's no penalty on large trees. In this section, I'll try to determine a good penalty on size or depth.
I've run several penalties on the Pima database, using LEF and Fitness Sharing Pareto scoring. In figure 3.8 accuracy is plotted against the penalty on the number of nodes. From the best individual of each of the 25 runs the average is taken and a 95% confidence interval is given.
Fitness is calculated as follows (x is the value on the x-as):

$$\text{Fitness} = \#\text{Errors} + x \cdot \#\text{nodes}$$

In figure 3.9 the same is done for a depth-penalty.

$$\text{Fitness} = \#\text{Errors} + x \cdot \text{depth}$$

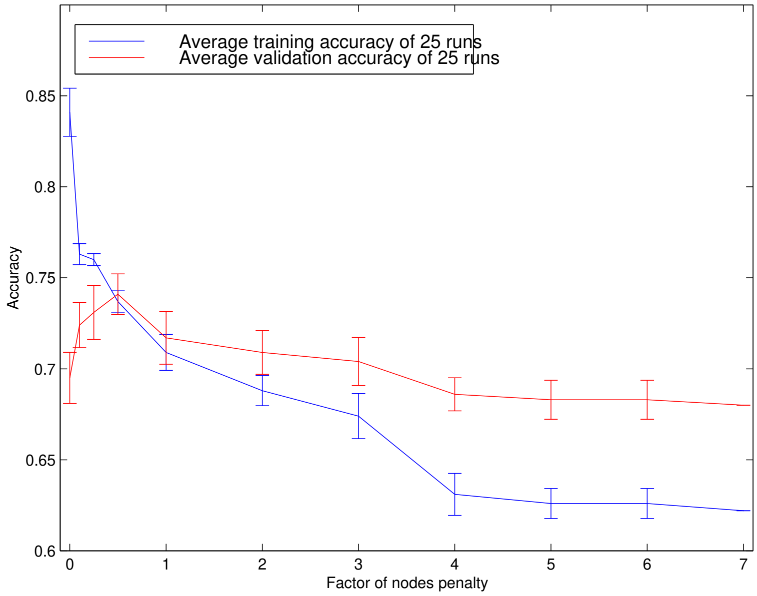

Figure 3.8: Nodes penalty

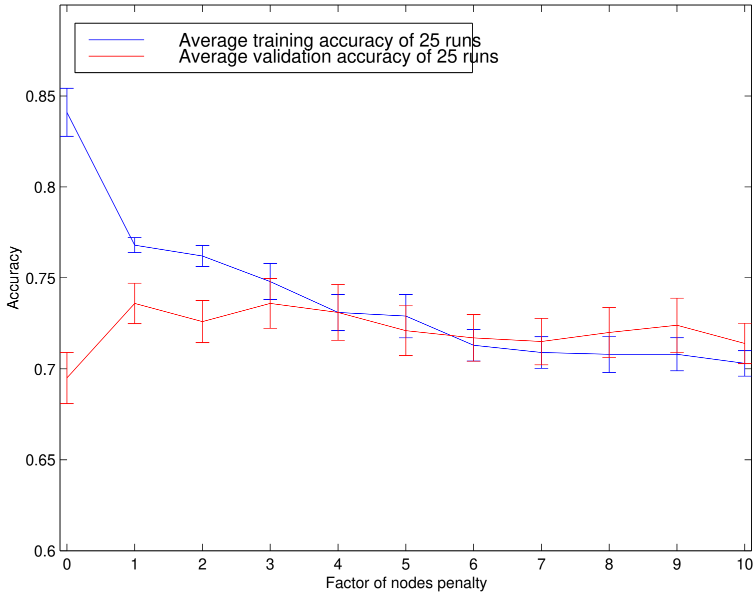

Figure 3.9: Depth penalty

In figure 3.10 the average treesizes of the best individual of each of the 25 runs are plotted.
The higher the penalty, the smaller the trees become, as expected. At small values, an increasing penalty creates better scoring trees on the validation set, but they're a bit worse on the training set. This means that there's less overgeneralization.
At higher values, the pressure on creating small trees gets larger (compared to the pressure on creating trees which make few errors) and performance drops.
At higher nodes-penalties, more and more trees are of size 1(they simply classify all cases as 0). At the highest nodes-penalty (7), all trees are. This is the reason why the confidence interval

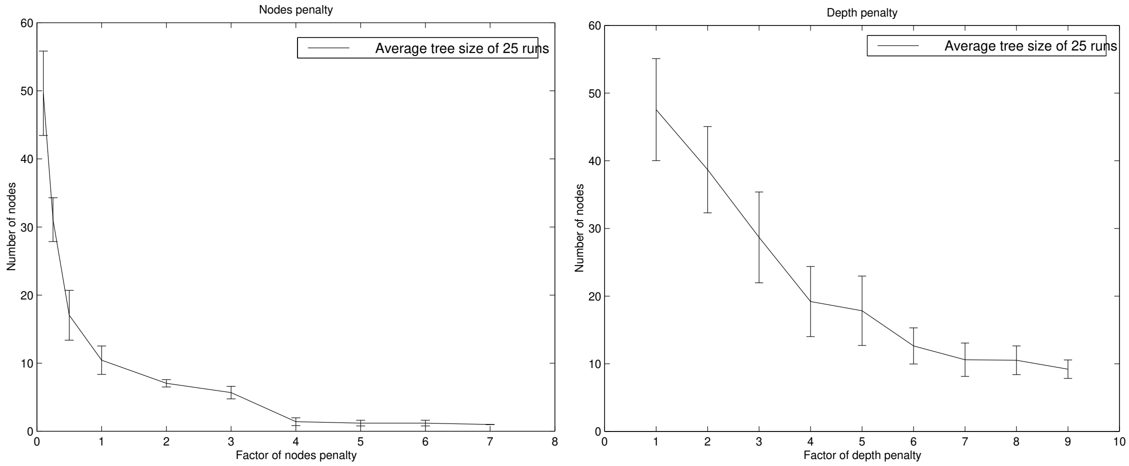

Figure 3.10: Average tree size of the best individual of each of the 25 runs

here is empty and the validation accuracy at higher penalties is better than training accuracy (the fraction of 0 cases is higher in this particular validation set than in this training set; no cross validation was used here [SHOULD I??? CAN THE EFFECTS OF INCREASING PENALTY BE COMPARED WITH THE SAME SPLIT???]).
The depth-penalty is less harsh for the trees than the nodes-penalty. With this I mean that tree sizes drop more gradually and performance goes down more slowly with an increasing penalty. Also, the differences between the higher (≥ 3) penalties are not significant.
Obviously, the penalty shouldn't be taken too large.
In the subsequent experiments I'll use (somewhat arbitrarily because not all differences are significant) a nodes-penalty of 0.25 or a depth-penalty of 2. These penalties have approximately the same effect on this dataset.

3.5 Limited Error Fitness

In tables 3.2 and 3.3 an overview is given of several experiments comparing GP with and without LEF. The numbers in the tables are the mean number of errors over the 25 runs, standard deviation and the accuracy of the best individual of all 25 runs.
On oblique problems, LEF in the average consistently outperforms non-LEF, but never significantly. The best individual is sometimes found in non-LEF, sometimes in LEF.
In some cases, the depth penalty is slightly better, in other cases the nodes penalty. The differences are not significant. This may depend on the solution to be found: if the "real" tree is a full tree, the depth penalty may be better; if it's a sparse tree, the nodes penalty may be better. This is because a depth-penalty makes no difference between full and sparse trees whereas a nodes-penalty does reward sparse trees. Although the depth-penalty is the same for full and sparse trees, trees will tend to be fuller because of the bloat phenomenon.
Generalization is reasonably good. There is some overtraining, both with and without LEF, but only the glass-runs with LEF and nodes/4 and the pima-runs with LEF and depth$\times$2 have significant overtraining.
In order to find how well the algorithm scores in comparison to complete random search, I randomly generated $25 \times 250 \times 1000 = 6,250,000$ individuals (in the same way as individuals are generated in a normal run) and scored them on the Pima problem. In 50,000 runs of 125 individuals, average performance of the best individual from each run was 63.8% $\pm$ 1.2 on the trainingset and 67.1 $\pm$ 1.8 on the validation set. The best individual scored 72.5% on the trainingset and 76.6% on the validation set with a very small tree:

Problem	without LEF		with LEF	
	Mean	Best	Mean	Best
100 vars 95 dummies	81.6% ± 2.8	88.8%	82.5% ± 2.2	88.0%
100 vars 97 dummies	88.1% ± 2.3	93.9%	89.3% ± 3.5	94.2%
Validation	89.0% ± 2.2	94.3%	89.9% ± 3.1	94.7%
100 vars 93 dummies	77.9% ± 4.3	88.1%	79.0% ± 5.4	87.9%
Validation	74.0% ± 4.5	86.5%	75.0% ± 6.2	84.9%
Glass	63.9% ± 4.1	69.1%	72.8% ± 2.5	78.3%
Validation	58.4% ± 7.1	64.5%	62.2% ± 6.1	54.3%
Ionosphere	92.8% ± 1.7	95.4%	91.6% ± 2.0	95.4%
Validation	93.3% ± 2.2	93.1%	91.3% ± 4.0	94.3%
Pima	72.6% ± 1.2	75.3%	72.6% ± 1.4	75.5%
Validation	70.6% ± 2.2	71.1%	70.8% ± 3.9	78.1%
Segmentation	67.4% ± 7.6	81.1%	68.8% ± 7.6	82.4%
Validation	64.5% ± 7.5	77.5%	65.1% ± 6.7	74.8%

Table 3.2: Comparison between standard GP and LEF, nodes/4. Mean number of errors of the best individual of the runs, standard deviation of the errors and the accuracy of the best individual of all runs

Problem	without LEF		with LEF	
	Mean	Best	Mean	Best
100 vars 95 dummies	83.6% ± 3.6	89.1%	82.9% ± 3.0	89.3%
100 vars 97 dummies	89.0% ± 3.2	94.4%	89.9% ± 3.3	94.1%
Validation	89.5% ± 3.1	94.4%	90.4% ± 2.8	94.2%
100 vars 93 dummies	80.6% ± 5.2	91.4%	80.5% ± 5.7	92.9%
Validation	75.3% ± 6.7	90.1%	75.2% ± 7.1	92.7%
Glass	66.7% ± 3.1	70.1%	65.8% ± 4.5	73.2%
Validation	57.5% ± 6.0	61.7%	56.1% ± 9.1	66.0%
Ionosphere	92.7% ± 2.2	95.4%	93.5% ± 2.3	95.4%
Validation	91.8% ± 3.6	94.3%	92.6% ± 3.1	96.0%
Pima	72.9% ± 1.0	76.0%	77.2% ± 2.2	82.3%
Validation	70.4% ± 1.8	68.8%	67.5% ± 3.0	70.0%
Segmentation	75.1% ± 5.3	84.3%	72.9% ± 7.6	88.1%
Validation	70.8% ± 4.9	75.0%	68.9% ± 7.2	84.4%

Table 3.3: Comparison between standard GP and LEF, depth×2. Mean number of errors of the best individual of the runs, standard deviation of the errors and the accuracy of the best individual of all runs

```
if (x1 >= 153.86)
     classification = 1
else classification = 0
```

This is only a bit worse than the GP-score with LEF.
In figure 3.11 average and best performance is plotted over (a long) time. The run is from the axis-parallel 100 variables 95 dummies database. It ran for 10000 generations.
Wihout LEF, average and best performance improve at the same time, sometimes with big leaps. See figure 3.12, left subplot. Average and best performance are quite close together.
With LEF, another pattern can be seen. While the best individual doesn't improve, the effects of the error limit can be seen (especially in the beginning, when the error limit hasn't reached the score of the best individual yet) in the rising average. Then suddenly, a (small) improvement

is made in the best individual. In the right subplot in figure 3.12 a closer look is made at what happens then. Average fitness decreases after the improvement in the best individual, for a few generations. Then average fitness improves again, settling at a slightly higher value than before, due to the lower error limit. (The average fitness in generations 3800–4100 is 439 and the average fitness in generations 4200–4500 is 456. This is a significant difference according to the t-test.)
In [Gathercole and Ross, 1997] the same thing was noticed: sudden backfalls in fitness accompanied with improvement just after that. They found sudden backfalls in best performance as well; a reason why I didn't found this may be that in my system, the best individual always remains in the population (this is called elitism).
With LEF, the population is much more diverse: average and best fitness are much less close together than without LEF. (Note the different scaling on the y-axis in figure 3.12.) The correlation between average and best fitness without LEF is 0.97. With LEF it is -0.70.
The reason why with LEF average performance doesn't follow best performance is probably the error limit. Because the error limit goes down some time after an improvement in the best individual, most other individuals will score worse than before, when there was a higher error limit.

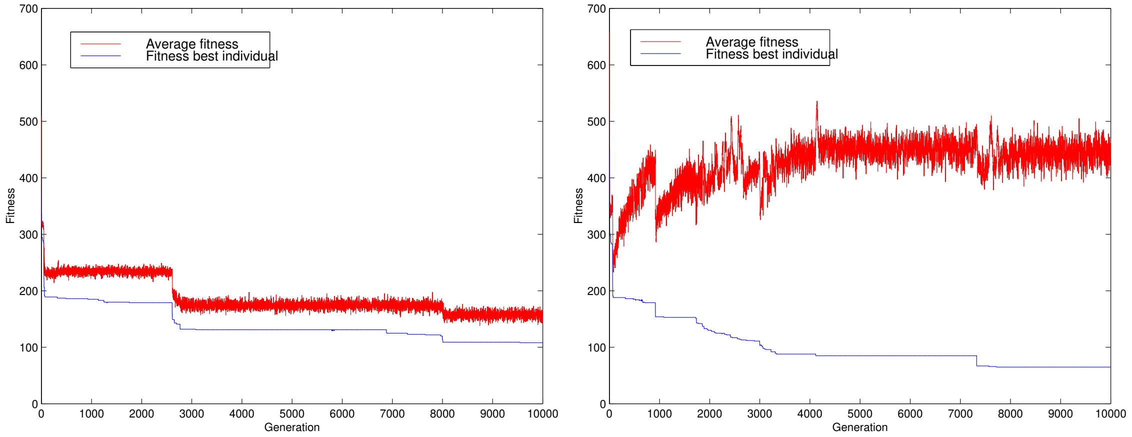

Figure 3.11: Population without (left plot) and with (right plot) LEF

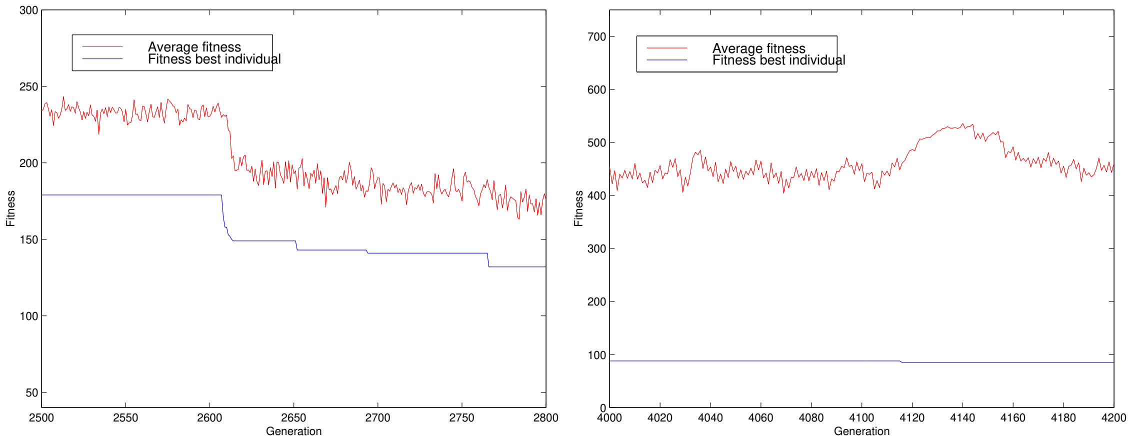

Figure 3.12: Zooming in on generations 2500-2800 resp. the dip at generation 4116

In figure 3.13 the tree size of the best of generation individual is plotted without and with LEF.

With LEF, changes in tree size are smaller, but the trees get much larger.
Like Gathercole and Ross found, improvement in the best individual goes hand in hand with an increase in size, as can be seen at generations 3000 and 7000. The correlation between tree size and best fitness is very high: -0.89. The same thing occurs without LEF, see generations 2500 and 2800. This correlation is much smaller though: -0.37.
These correlations may be explained by crossover: two existing fit individuals are combined into a larger, fitter tree. By mutation, the new tree gets smaller while remaining the same functionality. Effectively, some bloat is removed from the individual by the parsimony pressure.

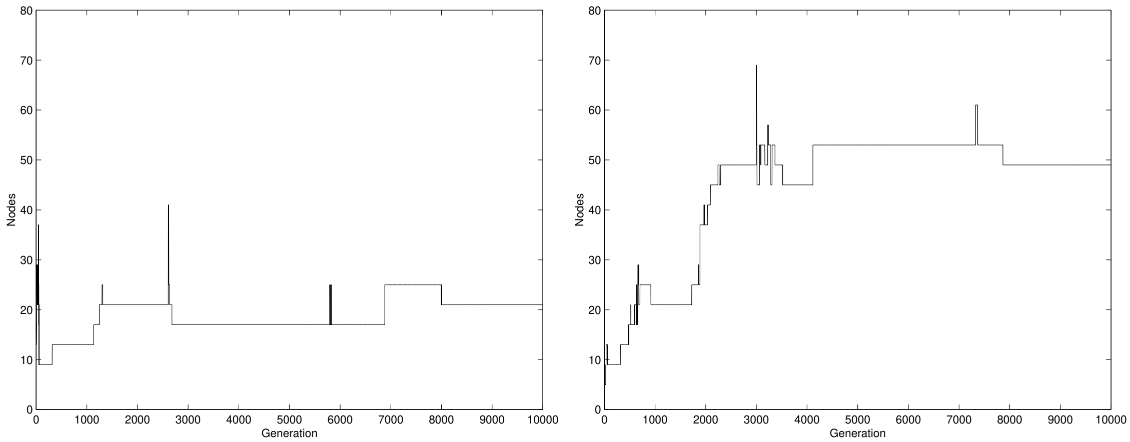

Figure 3.13: Tree size of best of generation individual without (left plot) and with (right plot) LEF

3.6 Pareto scoring

In this section I'll explain the experiments I've done with respect to Pareto scoring (see section 2.4.3). First, I'll use Pareto scoring with fitness sharing, then with counting the number of dominating or equal individuals in the sample.

3.6.1 Pareto scoring with Fitness Sharing

The distance between 2 individuals is based on the shape of the two trees. It is calculated as follows:

$$d_{i,j} = |\#nodes_i - \#nodes_j| + |depth_i - depth_j|$$

The fitness share is

$$s(d) = \begin{cases} 1 - \frac{d}{10} & d <= 10 \\ 0 & d > 10 \end{cases}$$

The value 10 seemed reasonable because one extra `CheckCondition` function node adds 5, 7 or 9 extra nodes to the tree (and sometimes increases depth by 1).
Note that in the distance measure the number of errors is not included. This is because we don't want to force the system in producing some bad individuals. If the number of nodes is included in the distance measure, the diversive pressure of the fitness sharing causes both well scoring and badly scoring individuals to emerge. When the number of errors is not included, there may (and probably will) still be some small individuals that potentially score badly, but if there are individuals that are both small and score well, these will be preferred. Including the number of errors in the distance measure might degrade these individuals.

3.6.2 Results

In table 3.4 an overview is given of the results from the experiments. The LEF-results are these from table 3.3, the runs with penalty 2×depth added to the number of errors.

Problem	without Pareto		with Pareto	
	Mean	Best	Mean	Best
100 vars 97 dummies	89.9% ± 3.3	94.1%	92.4% ± 2.3	94.7%
Validation	90.4% ± 2.8	94.2%	92.7% ± 2.4	95.2%
100 vars 93 dummies	80.5% ± 5.7	92.9%	85.7% ± 3.6	93.7%
Validation	75.2% ± 7.1	92.7%	82.0% ± 5.3	94.1%
Glass	65.8% ± 4.5	73.2%	70.6% ± 3.9	67.5%
Validation	56.1% ± 9.1	66.0%	60.7% ± 7.7	71.3%
Ionosphere	93.5% ± 2.3	95.4%	95.8% ± 1.2	98.9%
Validation	92.6% ± 3.1	96.0%	92.5% ± 2.7	87.5%
Pima	77.2% ± 2.2	82.3%	76.8% ± 1.5	80.5%
Validation	67.5% ± 3.0	70.0%	72.5% ± 3.3	74.2%
Segmentation	72.9% ± 7.6	88.1%	83.4% ± 4.8	91.0%
Validation	68.9% ± 7.2	84.4%	78.2% ± 5.3	82.9%

Table 3.4: Comparison between LEF with and without Pareto Scoring. Mean number of errors of the best individual of the runs, standard deviation of the errors and the accuracy of the best individual of all runs

In figure 3.14 the population is drawn at the end of a run of the 100 vars 93 dummies oblique database. White noise is added to the x-value of each individual in order to show how many individuals there are on overlapping points. (#Errors can get so high because of the LEF-way of counting the non-evaluated cases as false (and by producing lousy children of course).)
The left figure shows the population after a run without Pareto Scoring and the right one with Pareto Scoring. Clear is that with Pareto Scoring there's a lot more diversity in the population in terms of size and accuracy.

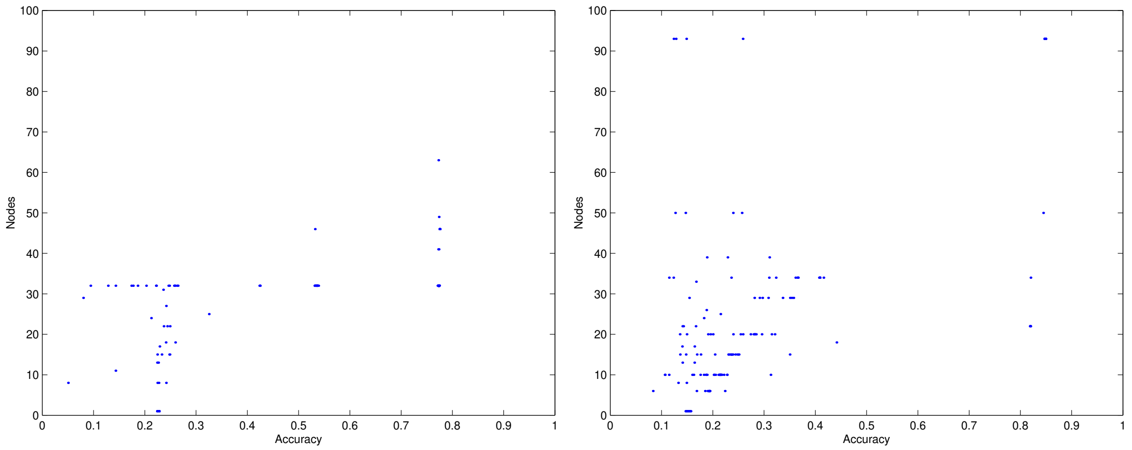

Figure 3.14: Population without (left plot) and with (right plot) Pareto Scoring

In figures 3.15 and 3.16 the distribution of errors made by the best individual in each of the 25 runs is shown, without (left subplot) and with (right subplot) Pareto scoring. Dataset is the Pima-set; the non-Pareto set is with penalty nodes/4 and in all cases LEF was used. Clearly, Pareto scoring is much more accurate both on the training data and on the validation data than non-Pareto. The

difference between average training accuracy and validation accuracy is about the same, so the amount of overtraining is also approximately the same.
In figure 3.17, training and validation accuracies are plotted. The correlations between training and validation accuracy are 0.63 and 0.19, respectively. Without Pareto, the individual with the best training accuracy also has the best validation accuracy, whereas with Pareto, the individual with the best validation accuracy has far from the best training accuracy. This suggests that the best individual from the Pareto run is not necessarily the very best individual, but it is still better than the one from the runs without Pareto scoring.
So, Pareto scoring would seem preferable since training and validation accuracy are generaly better. The amount of overtraining is approximately the same.
Accuracy distribution is somewhat skewed to the left. This means that in order to get a good individual, many runs are necessary.

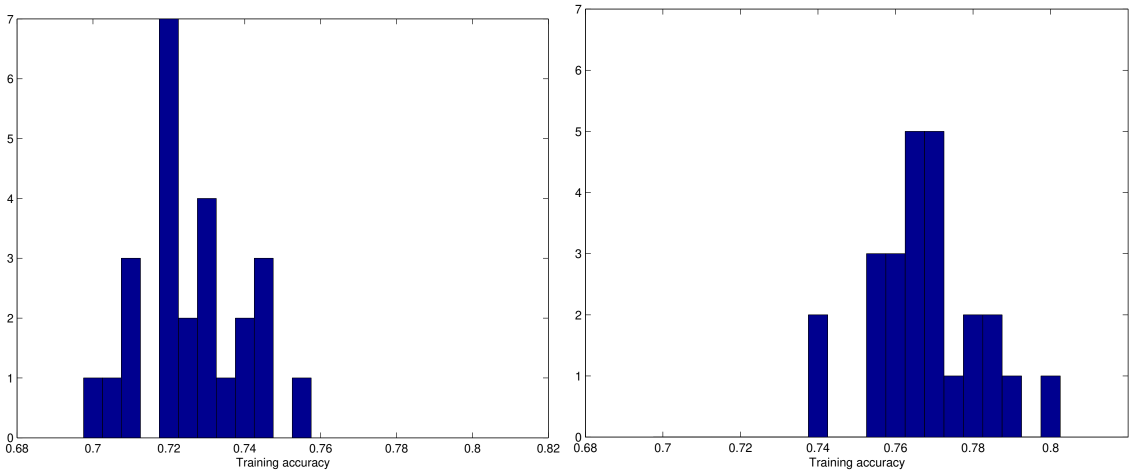

Figure 3.15: Training accuracy distribution without (left plot) and with (right plot) Pareto Scoring

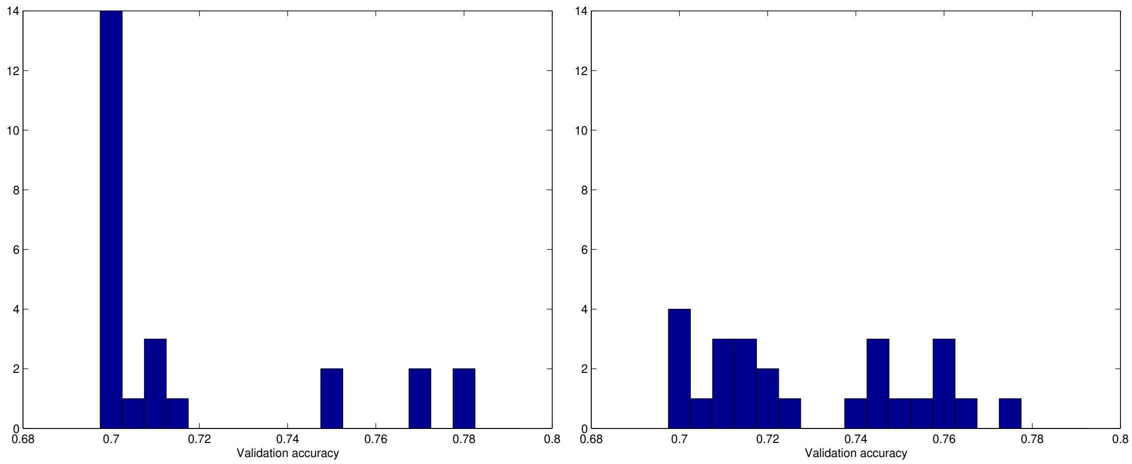

Figure 3.16: Validation accuracy distribution without (left plot) and with (right plot) Pareto Scoring

In figure 3.18 a plot is drawn of average fitness in the population and fitness of the best individual in each generation. The run is one on the Pima database, with LEF, for 1000 generations. The

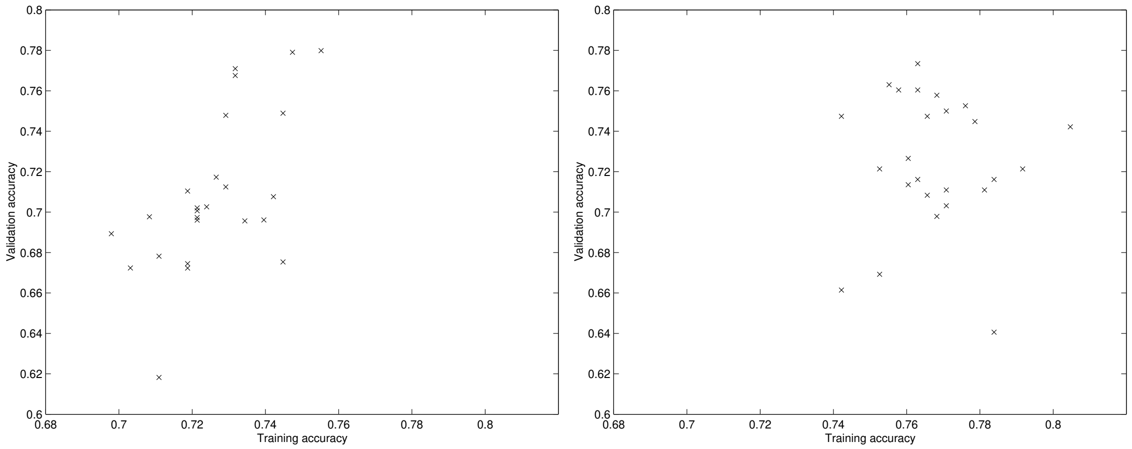

Figure 3.17: Training accuracy versus validation accuracy without (left plot) and with (right plot) Pareto Scoring

effects of LEF are the same as without Pareto: many improvements in the best individuals are accompanied by an increase in average errors, due to the lower error limit. While best score is stable, average errors mostly decrease because of the fitness pressure.

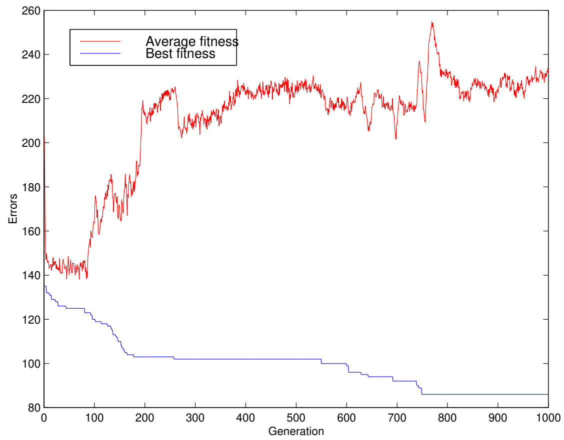

Figure 3.18: Average and best fitness over time in one Pareto Scoring run with the Pima database

From all this it may be concluded that the Pareto scoring method works better than applying penalties on size. Performance on both training and validation set generally increases.

3.6.3 Pareto scoring with domination-based ranking

The scoring mechanism now works as follows. In each tournament, first all non-dominated individuals are found, just like in the previous section. Then, a choice is made between those individuals by sampling a number of individuals from the population (50 in this case) and counting the number of individuals that are dominated by or equal (in size and errors) to each non-dominated individual from the tournament. The one that dominates most individuals from the sample gets to reproduce.

This is also a form of niching, because if there are many individuals of the same size and errors, they won't win many tournaments. This causes the population to drift out over the space.

3.6.4 Results

In table 3.5 an overview is given of the results.
On some problems, Domination-type Pareto works almost as well as Fitness Sharing Pareto, but on other problems, performance is much worse. Especially on problems with more than 2 classes, it produces very small trees, as can be seen in table 3.6. Performance on those problems is bad, probably because the pressure on small trees is too high so not all classes can be represented in the tree. On the other problems, performance is better, but Fitness Sharing Pareto is still better.

Problem	Fitness Sharing Pareto		Domination Pareto	
	Mean	Best	Mean	Best
100 vars 97 dummies	92.4% ± 2.3	94.7%	89.8% ± 2.8	94.8%
Validation	92.7% ± 2.4	95.2%	90.4% ± 2.7	95.5%
100 vars 93 dummies	85.7% ± 3.6	93.7%	78.4% ± 5.4	93.4%
Validation	82.0% ± 5.3	94.1%	75.5% ± 6.2	93.5%
Glass	70.6% ± 3.9	67.5%	65.5% ± 3.8	71.7%
Validation	60.7% ± 7.7	71.3%	59.2% ± 7.1	63.8%
Ionosphere	95.8% ± 1.2	98.9%	93.3% ± 2.0	96.6%
Validation	92.5% ± 2.7	87.5%	93.0% ± 3.8	92.0%
Pima	76.8% ± 1.5	80.5%	74.2% ± 1.5	77.1%
Validation	72.5% ± 3.3	74.2%	71.0% ± 3.3	79.2%
Segmentation	83.4% ± 4.8	91.0%	74.1% ± 4.5	86.2%
Validation	78.2% ± 5.3	82.9%	71.0% ± 4.1	81.7%

Table 3.5: Comparison between Fitness Sharing and Domination Pareto scoring. Mean accuracy of the best individual of all 25 runs, standard deviation of the accuracies and the accuracy of the best individual of all runs

In figure 3.19 the population is drawn after a run of the 100 variables, 93 dummies database.

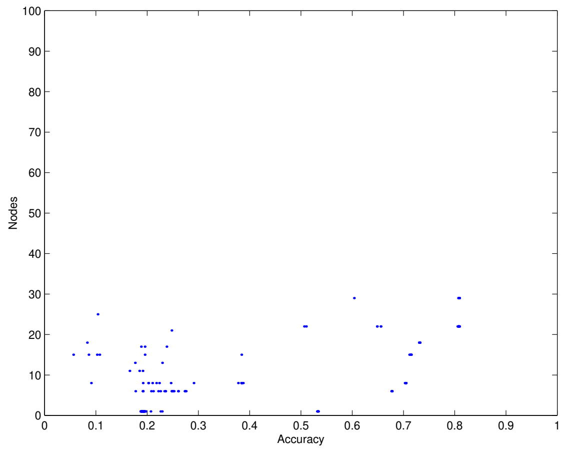

Figure 3.19: Accuracy and number of errors of the population after a run with Domination Pareto Scoring

The population is generally smaller than with Fitness Sharing Pareto and also smaller than when no Pareto scoring is used. (See figure 3.14.) The population is spread out quite well, just like Fitness Sharing Pareto, but over a smaller area. The spread over the y-axis (the number of errors) is better with Domination Pareto than Fitness Sharing Pareto, where the area between

200 and 500 errors is empty.

3.6.5 Paretofronts

In figure 3.20 the Pareto fronts are drawn after one run of both Fitness Sharing and Domination Pareto on the four Machine Learning databases. The number of individuals on the fronts varies between 156 and 207, so most individuals from the population (size 250) are on the Paretofront.
The overall shapes of the two methods are approximately the same, suggesting that both methods succeed reasonably well in finding a Pareto front. The Fitness Sharing lines are always above the Domination lines in the small and little accurate part of the graph. Fitness Sharing creates larger trees at the same (low) accuracy than Domination. A reason for this is that with Fitness Sharing a bonus is given for being different from the others in size, which causes more diversive pressure. Therefore, the pressure to be small is less with Fitness Sharing than Domination.
This is the reason why Domination trees are smaller then Fitness Sharing trees (see table 3.6). The larger variation in Fitness Sharing trees' sizes is also caused by this phenomenon. This larger variation in turn may explain the better performance of Fitness Sharing: better trees can be found when there's more variation in the population.
In the right half of the graph, the Fitness Sharing lines are always more to the right (more accurate) than the Domination lines, showing the better performance of Fitness Sharing Pareto.
Domination Pareto creates smoother, more spread out Pareto fronts, while Fitness Sharing Pareto is more bumpy. A striking feature is that most graphs contain some long straight edges without any points on them, especially with Fitness Sharing Pareto. For example with the Glass database, the Fitness Sharing system somehow wasn't able to find the individuals below the straight line that Domination Pareto was able to find. The larger average size of the population may have something to do with this, because that might make it harder to find smaller individuals below the current Pareto front.
In figures 3.21 and 3.22 the evolution of the Pareto fronts over time is drawn.
In some graphs (for example, the Fitnesss Sharing Glass run), it can be observed that the smallest individuals (1 node) actually start to perform worse. Why???

3.7 Treesizes

In this section I'll compare the treesize that are found by the various methods. In table 3.6 an overview is given of average and best treesizes of the various methods.

Problem	Without LEF		With LEF		With LEF and Pareto	
	Average	Best	Average	Best	Average	Best
100 vars 97 dummies	22.04 ± 13.16	19	28.72 ± 23.74	26	27.64 ± 14.80	19
100 vars 93 dummies	56.18 ± 31.84	28	80.60 ± 84.04	166	52.80 ± 24.06	40
Glass	25.76 ± 11.17	69	27.12 ± 8.25	38	38.96 ± 13.96	38
Ionosphere	36.28 ± 14.30	42	37.84 ± 19.95	33	49.44 ± 19.95	48
Pima	34.92 ± 19.10	78	32.84 ± 15.42	28	65.64 ± 22.03	84
Segmentation	58.64 ± 21.97	82	70.32 ± 28.07	97	77.60 ± 30.08	95

Table 3.6: Average treesize and standard deviation of the best individual of the 25 runs and the treesize of the best individual of all runs without LEF, with LEF and with Fitness Sharing Pareto (penalty = depth × 2 for the runs without Pareto)

The trees generated in the Fitness Sharing Pareto method are slightly bigger than with the other two methods, which have a penalty for size added to their fitness. In table 3.7 the treesizes in

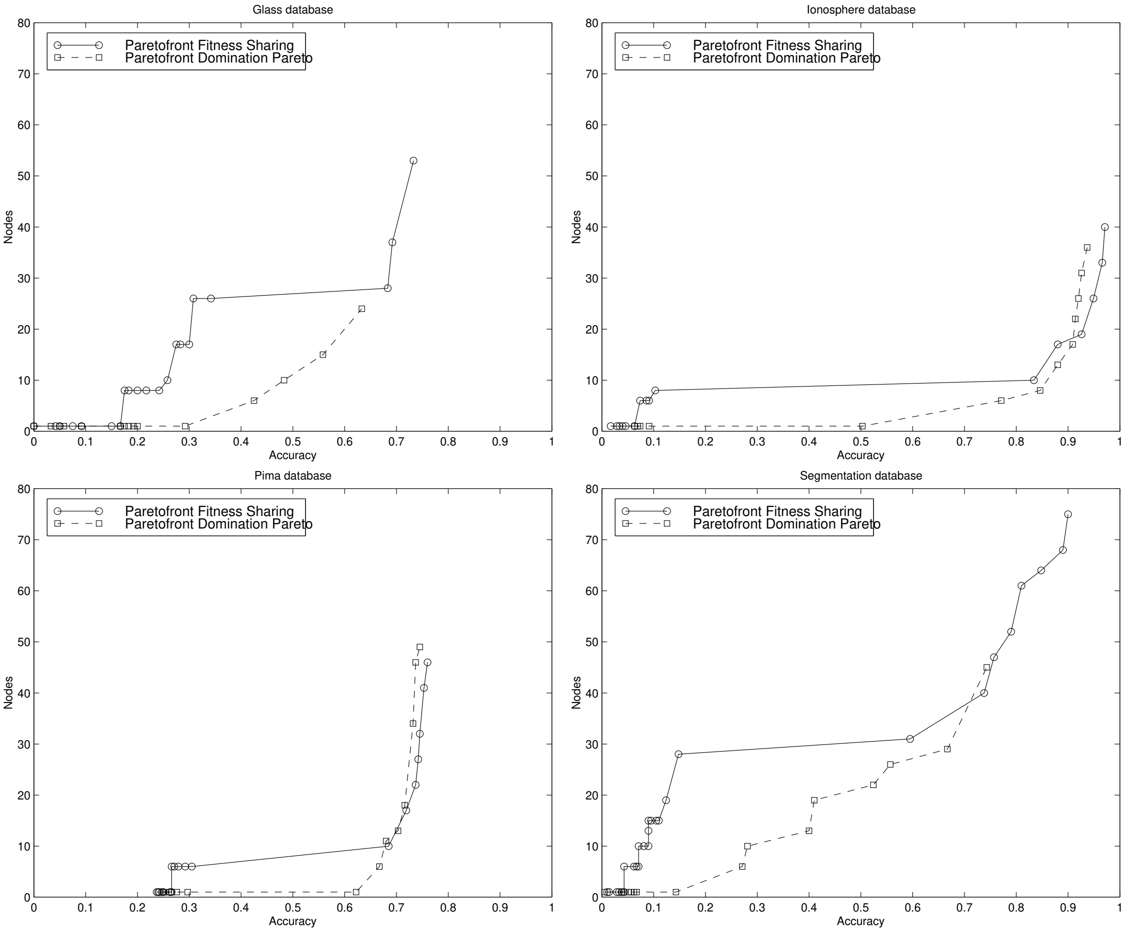

Figure 3.20: Paretofronts after one run of each of the Machine Learning databases (Glass, Ionosphere, Pima and Segmentation). The straight line is with Fitness Sharing and the dashed line is with Domination Pareto

Fitness Sharing Pareto are compared to treesizes in LEF with no penalty added to fitness and with Domination Pareto runs. Domination Pareto creates smaller trees than Fitness Sharing Pareto.

All methods succeed in creating much smaller trees than when no size-penalty at all is applied. The difference is almost a factor of 10.

Problem	Fitness sharing Pareto		With LEF, no penalty		Domination Pareto	
	Average	Best	Average	Best	Average	Best
Glass	38.96 ± 13.96	38	464.13 ± 241.12	597	23.12 ± 12.16	43
Ionosphere	49.44 ± 19.95	48	460.50 ± 198.54	500	33.40 ± 19.58	67
Pima	65.64 ± 22.03	84	607.20 ± 258.76	1066	45.20 ± 19.86	58
Segmentation	77.60 ± 30.08	95	564.08 ± 250.15	953	45.60 ± 16.48	51

Table 3.7: Average treesize and standard deviation of the best individual of the 25 runs and the treesize of the best individual of all runs with Fitness Sharing Pareto, with LEF without any size penalty and with Domination Pareto

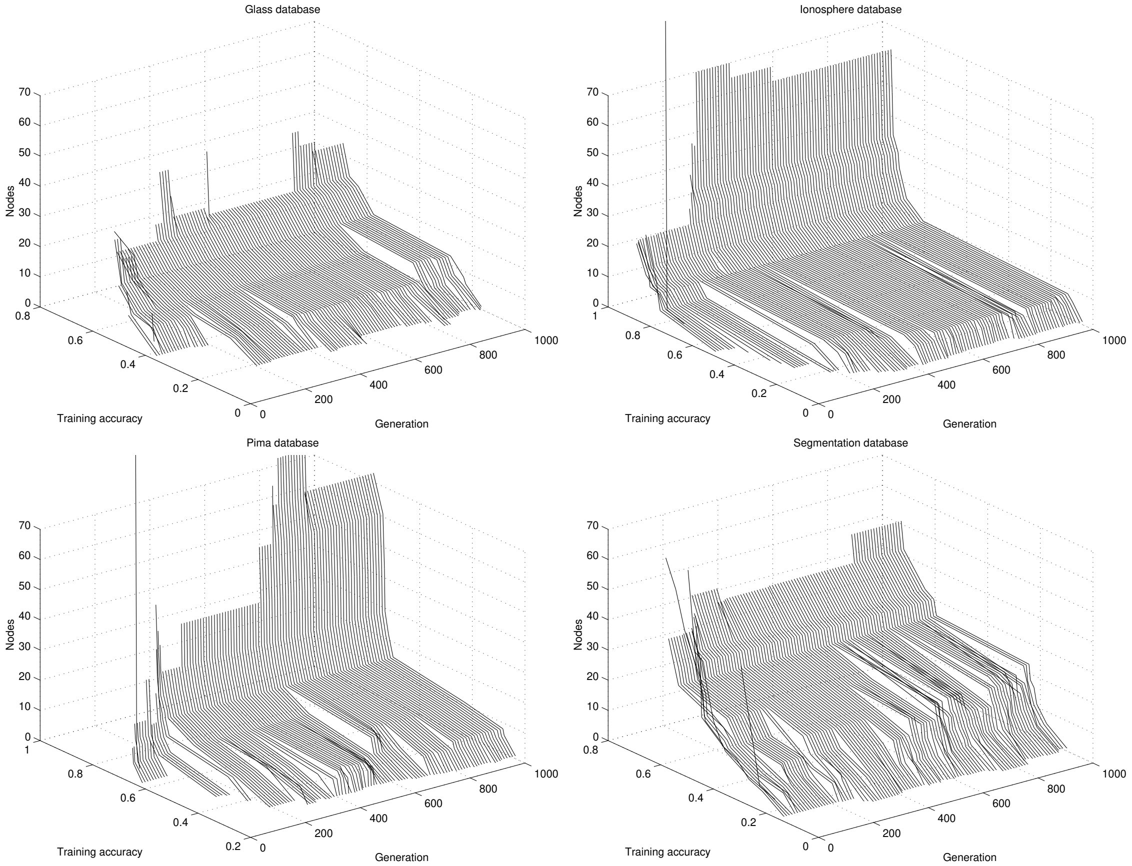

Figure 3.21: Paretofronts over one run with Fitness Sharing of each of the Machine Learning databases (Glass, Ionosphere, Pima and Segmentation).

3.8 Execution speed

In order to find out the relationship between the size of the database and execution time, the following experiment was performed.

The Pima database (both the training and the validation set) was multiplied 1, 2, . . . , 10 times. This way, 10 databases were available that were only different in size. Contents were exactly the same, only each tuple was represented in different amounts in each database.

In figure 3.23 user time (measured with the UNIX-command `time`; it it the amount of time the processor actually spent on the concerning process) is plotted against the size of the database (the number of copies) for 1 run with 1, 20 and 100 generations, both without (left subplot) and with (right subplot) Limited Error Fitness.

Clearly, execution time is approximately linear with size of the database. Deviations may occur due to different loads on the machine at the time each experiment was run. As expected, execution time with LEF is lower than execution time without LEF, especially with more generations, when the error limit is really starting to matter.

Execution time is linear because in each generation, all individuals are evaluated on the whole database. Since evaluation time is linear with respect to database size and no other operation is performed on the database, execution time is also linear.

Deviations from linearness may occur because validation only occurs on the best individual in each generation. If that individual happens to be smaller, less execution time is needed (because there are less nodes to evaluate) and vice versa. Of course, this only applies to LEF, since without LEF,

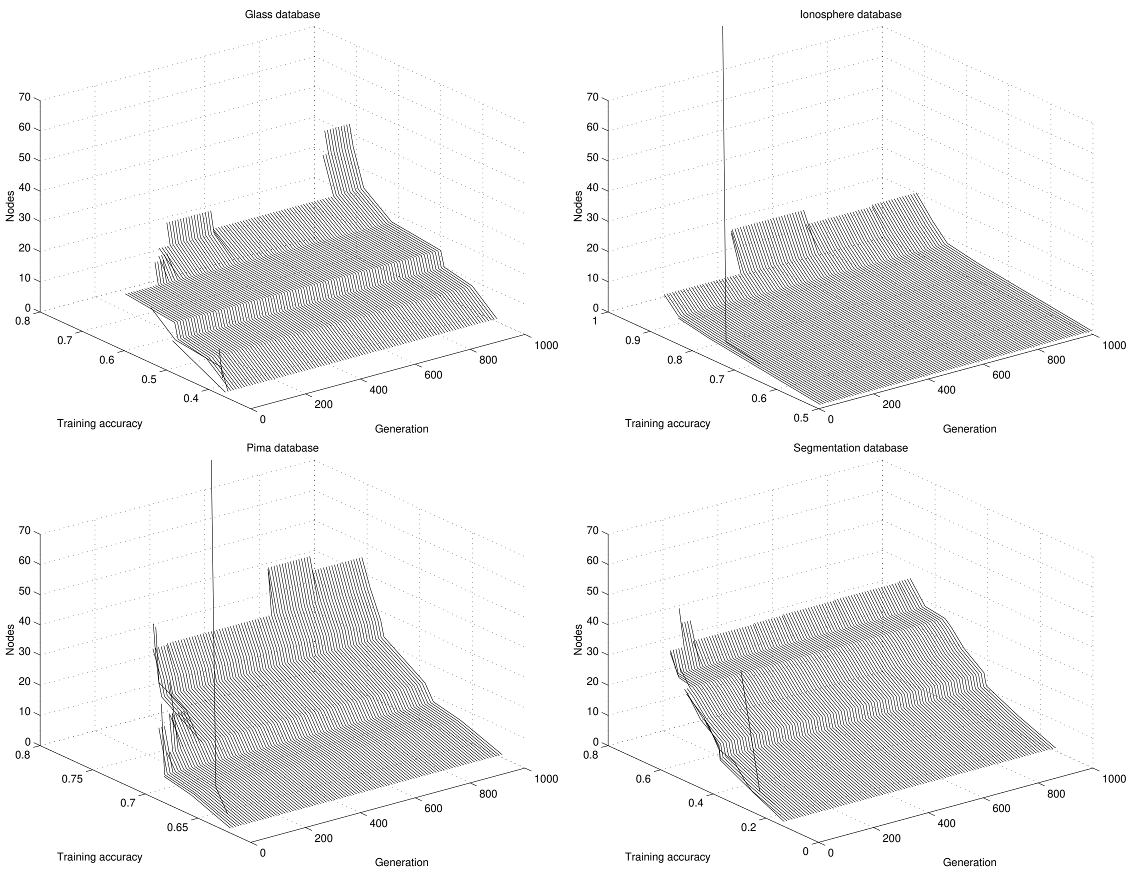

Figure 3.22: Paretofronts over one run with Domination Pareto of each of the Machine Learning databases (Glass, Ionosphere, Pima and Segmentation).

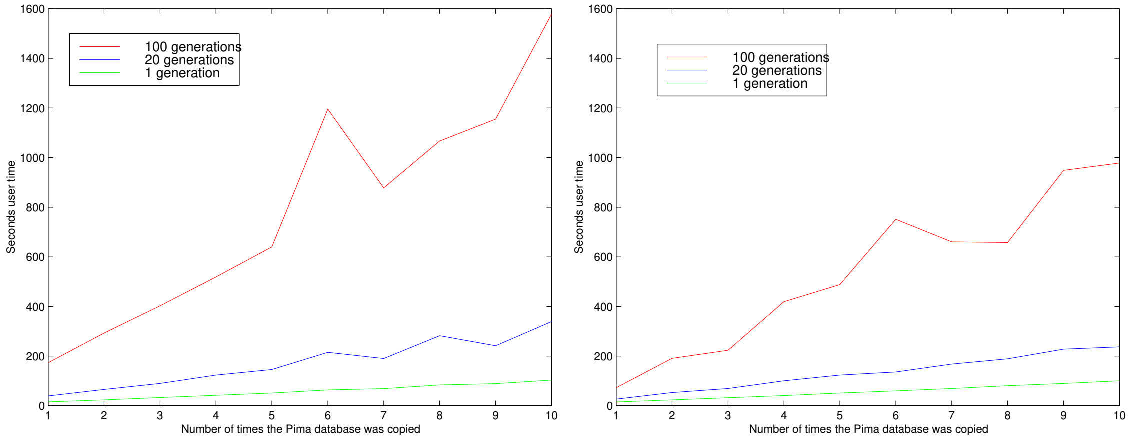

Figure 3.23: Execution time (y-axis) on the Pima database several times copied (number of copies on the x-axis) without (left plot) and with (right plot) Limited Error Fitness

the same evolution occurs with the 10 “different” databases. With LEF, only if the reductions in error limit are also multiplied by the number of times the database was copied, evolution would be the same.

3.9 Effects of noise

In this section, I'll investigate the effects of noise in the training and validation data on the performance of the GP-system.
I recreated the 100 variables, 93 dummies database with different amounts of noise, varying with steps of 5% from 0% to 40%. The same tree was used each time, so the only difference in the datasets was the percentage of noise. The runs were with LEF and Fitness Sharing Pareto scoring.
In figure 3.24 the results of this experiment are given. On the x-axis the percentage of noise is given and on the y-axis the accuracy. The straight black line gives the maximal possible performance when no overtraining occurs, namely the percentage of the data that isn't noise.
Since the amount of noise was only a statistical guidance for the number of switches when creating the datasets, the specific percentages are not exactly 0, 5, 10, 15, ..., but are slightly different for both the training and the validation set. For example, for the validation set, the exact percentages are 0, 6.3, 9.5, 15.4, ... This is which is why the plot looks as such.

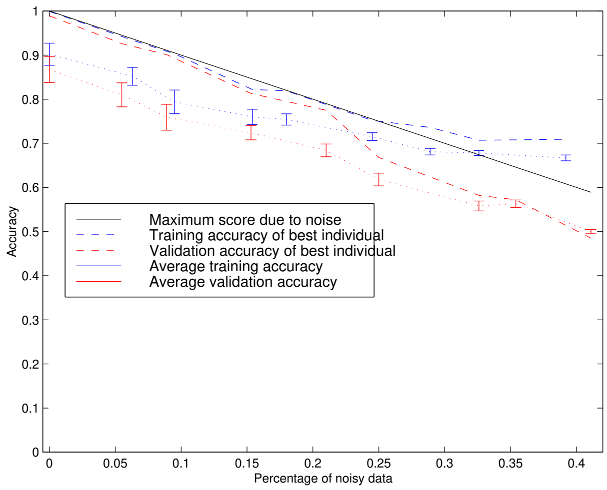

Figure 3.24: Accuracy in presence of noise. The straight line is the maximum score without overtraining (= percentage of non-noise = 1 - noisepercentage), the dashed lines are the training and validation accuracy of the best individual of all 25 runs and the dotted lines are the average training and validation accuracy of the best individuals from each run

When there's little noise (less than 20%), both training accuracy and validation accuracy are very close to the best possible performance. Average performance is about 10% worse than best performance. When more than 20% of the data is noise, overtraining starts to occur. The difference between training accuracy and validation accuracy gets larger and training accuracy gets above the "maximum" possible accuracy (given by $1 - noisepercentage$). As noise increases, so does overtraining.
In figure 3.25 the size of the trees is plotted against the percentage of noise, with 95% confidence intervals for the average treesizes. The trees get significantly larger when there's more noise in the data, another sign of overtraining. Up to 20% noise, a horizontal line can be drawn through the confidence intervals, but after that this is no longer possible. When there's little noise ($\leq 20\%$), the best trees are smaller than average, whereas when there's more noise, they're bigger than average. Again, this is a sign of overtraining.
The peak of the best individual at 35% is just a statistical coincidence. The average treesize increases normally, as can be seen by the confidence intervals.

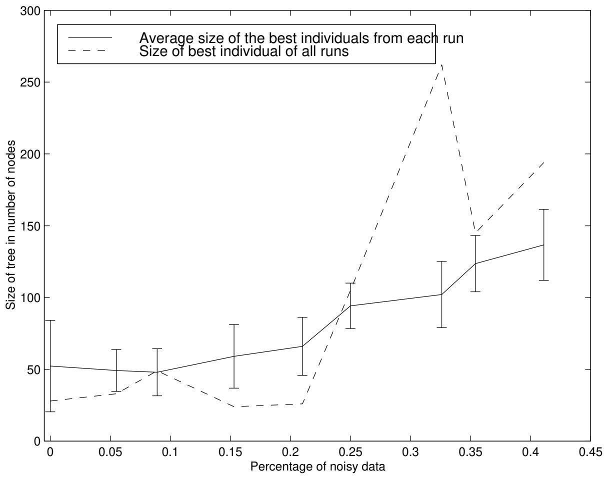

Figure 3.25: Sizes of trees in presence of noise. The straight line is the average treesize of the best individual of each of the 25 runs with 95% confidence interval. The dashed line is the treesize of the best individual of all runs

Chapter 4

Conclusions

Using GP to find linear classifiers works reasonably well, but it takes quite a long time to find these solutions, especially with larger databases. Many generations are needed before good individuals are found, often near 1000. Even after that, improvements are still being made over time. Obviously, the longer tha GP is allowed to run, the more thorougly it will have searched through the space of all possible trees and the better the chance of finding the best tree. Eventually, it will come close to a complete random search.

4.1 Bloat

In order to reduce bloat, several methods were tried: penalties on size and two ways of Pareto scoring, one with Fitness Sharing and one based on domination.
The pressure on creating small trees shouldn't be larger than the pressure on making few errors, but a small pressure is better than no pressure.
There's no consistent difference between a penalty on depth and a penalty on the number of nodes. This may have something to do with the optimal tree being sparse of dense: a dense tree may be easier to find with a depth penalty and a sparse tree with a nodes penalty.
All methods succeed in producing much smaller trees than the system without any way to avoid bloat. The penalties cause the trees to become smallest, but the Pareto systems create trees that aren't much larger. Fitness Sharing Pareto creates larger trees than Domination Pareto.

4.2 Accuracy

The GP creates more accurate trees when using Limited Error Fitness than when not using it. Using Fitness Sharing Pareto scoring on the individuals works better in terms of accuracy than using a penalty on size.
Domination Pareto scoring works slightly less good than Fitness Sharing Pareto scoring.
Both types of Pareto scoring cause the population to become much more diverse in size and errors than when no Pareto scoring is involved. The Fitness Sharing Pareto system seems to be quite able to find both small and accurate trees.

4.3 Performance

Execution speed is linear with respect to database size (mutatis mutandis of course).
The GP is quite resistant to amounts of noise of up to 20%. When more than 20% of the data is noise however, overtraining starts to occur. The more noise there is, the more overtraining occurs and the larger the trees become.

Bibliography

[Blickle and Thiele, 1994] Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik (MPI-I-94-241).

[Gathercole and Ross, 1997] Chris Gathercole and Peter Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Goldberg, 1989] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., January 1989.

[Greeff and Aldrich, 1997] D. J. Greeff and C. Aldrich. Evolution of empirical models for metallurgical process systems. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, page 138, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[McPhee and Miller, 1995] Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Montana, 1995] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[Nordin and Banzhaf, 1995] Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Qureshi, 1997] A. Qureshi. Gpsys 1.1, June 1997. http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html.

[Soule *et al.*, 1996] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Soule, 1998] Terence Soule. *Code Growth in Genetic Programming.* PhD thesis, University of Idaho, Moscow, Idaho, USA, 15 May 1998.