USING GENETIC ALGORITHMS TO OPTIMIZE SOFTWARE QUALITY ESTIMATION MODELS

Danielle Azar

School of Computer Science
McGill University, Montréal

June 2004

A Thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

© Danielle Azar, 2004

Abstract

Assessing software quality is fundamental in the software developing field. Most software quality characteristics cannot be measured before a certain period of use of the software product. However, they can be predicted or estimated based on other measurable quality attributes. Software quality estimation models are built and used extensively for this purpose. Most such models are constructed using statistical or machine learning techniques. However, in this domain it is very hard to obtain data sets on which to train such models; often such data sets are proprietary, and the publicly available data sets are too small, or not representative. Hence, the accuracy of the models often deteriorates significantly when they are used to classify new data.

This thesis explores the use of genetic algorithms for the problem of optimizing existing rule-based software quality estimation models. The main contributions of this work are two evolutionary approaches to this optimization problem. In the first approach, we assume the existence of several models, and we use a genetic algorithm to combine them, and adapt them to a given data set. The second approach optimizes a single model. The core concept of this thesis is to consider existing models that have been constructed on one data set and adapt them to new data. In real applications, this can be seen as adapting already existing software quality estimation models that have been constructed on data extracted from common domain knowledge to context-specific data. Our technique maintains the white-box nature of the models which can be used as guidelines in future software development processes.

Résumé

L'évaluation de la qualité de logiciel s'avère d'une importance fondamentale dans le domaine de développement de logiciel. La plupart des qualités ne peuvent, cependant, pas être mesurées d'avance (à priori), c'est-à-dire avant que le logiciel ne soit utilisé pour une certaine période de temps. Cependant, ces qualités peuvent être estimées à partir de propriétés qui sont mesurables. Dans ce but, sont construits les modèles d'estimation de la qualité de logiciel, qui trouvent un grand usage. La plupart de ces modèles sont construits à partir de techniques statistiques ou d'apprentissage automatique. Toutefois, dans ce domaine, il est difficile d'obtenir des ensembles de données pouvant être utilisés pour produire de tels modèles. En effet, la plupart du temps, ces ensembles de données sont des propriétés privées et ceux qui sont publics sont de tailles très limitées ou non représentatifs. Ceci implique une détérioration assez importante dans la prédiction que les modèles performent sur des données nouvelles.

Cette thèse explore l'utilisation des algorithmes génétiques pour résoudre le problème d'optimisation des modèles, à base de règles, d'estimation de la qualité de logiciel. La contribution principale de cette oeuvre consiste en deux approches évolutionnistes à ce probléme d'optimisation. La première consiste à combiner différents modèles existants, et de les adapter à un ensemble de données. La deuxième consiste à optimiser un seul modèle. L'idée principale de cette thèse est de prendre des modèles qui existent, ayant été déjà construits à partir d'un certain ensemble de données, et de les adapter à d'autres données. En pratique, ceci pourrait être vu comme l'adaptation de modèles, construits à partir des données générales, à des données reliées à un contexte bien spécifique. Notre technique maintient la nature "boîte blanche" des

modèles, ces derniers pouvant être utilisés comme normes dans des procédures futures de développement de logiciel.

Acknowledgements

First, I would like to thank my supervisors Prof. Doina Precup, Prof. Sue Whitesides and Prof. Houari Sahraoui. I thank Prof. Sahraoui for initiating the idea of applying genetic algorithms to optimize software quality estimation models. While my interests initially lied in the field of genetic algorithms and evolutionary computation in general, Houari introduced me to the field of software quality as an area of application for this technique. I thank him for sharing his expertise in this area. I would like to thank Prof. Whitesides for accompanying me on this journey from the beginning until the end. Prof. Whitesides has given me very valuable feedback especially during the writing of this thesis. I also thank her for the financial support that she provided for me during the last six months (and at other times during my studies) so that I could concentrate on writing the thesis only. I thank Prof. Doina Precup for her guidance and supervision. Doina worked closely with me and was a great source of intellectual inspiration. She had immeasurable patience and energy both as a teacher and as a supervisor. Her always-cheerful mood made the task and the worries always smaller. Thank you Doina for putting up with me and managing to send me out on frequent panic trips but always succeeding to bring me back. Thank you also for constantly reminding me that *"GAs are not alive"* and *"chromosomes, in computer science, do not give birth"*.

I would like to extend my gratitude to Prof. Denis Thérien and Prof. Prakash Panangaden for giving me the opportunity to teach in the Computer Science department. Not only this was the major source of my income and allowed me to continue but this, also, was a very enriching experience that I enjoyed and will never forget.

I would also like to extend my appreciation to Vicki Keirl for giving me the opportunity to work as a lab supervisor and a lab consultant and for being a wonderful boss always open to suggestions, always fair.

My appreciation extends to all the administrative staff: Lise Minogue, Lucy St-James, Judy Kenigsberg, Vera Gorni, Liette Chin, Sheryl Morrissey, Danielle Bissonnette and especially Diti Anastasopoulos. As the graduate secretary, Diti was always eager to help with a big smile on her face. I should acknowledge that the administrative staff in the computer science department made my stay at McGill very agreeable.

A very warm thanks goes to system staff: Philippe Ciaravola, Alex Batko, Jason White, Kailesh Mussai and especially Andrew Bogecho and Ron Simpson. Thanks to these guys, I had no worries regarding technical service because I knew for sure that it was more than excellent at all time.

I would also like to thank Prof. Gerald Ratzer for giving me office space in his lab that I invaded during all these years. I also thank him for all the wonderful chats that we had. I really enjoyed being in his lab.

Thanks to Prof. Jörg Kienzle for giving me access to his printer and lab machines.

I would also like to acknowledge my source of funding. The first three years, I received generous funding from CRIM *(Centre de Recherche Informatique de Montréal)*. During the course of my study, I also received some funding from Prof. Whitesides. Thanks to them, I could continue.

I would like to thank Mark Mouadeb and all the people in the Machine Learning lab for putting up with me and my GAs that were constantly running on most of the machines for the past month or so.

I would like to thank Sherif Shaker for implementing the combining GA and Miriam Zia for helping me with the graphs. I also thank them for being wonderful friends and giving me all the space that I needed while I was writing this thesis.

I would also like to thank my friends and office-mates Alexandre Denault and Sadaf Mustafiz for keeping a very friendly atmosphere in the office/lab. I thank

Alexandre for helping me figure out many of the Latex-related enigmas and for asking me every single day for the past 4 months *"Page?..."*. It will feel weird not to hear this question again.

I would like to thank Malvika Rao for answering some Latex-related questions at 3:00 in the morning. I especially thank Malvika for her friendship and for introducing me to delicious desserts.

My deepest gratitude goes to my very good friend and colleague Bohdana Ratitch who helped me and supported me in every way one can imagine. Without Bohdana's help and encouragement, this work would have never been possible. I also thank her for the useful discussions that we had and the brilliant ideas that she shared with me. I have always had a great respect and admiration for Bohdana both as a person and as a researcher.

Finally, my love and thanks go to my family and God. Mom and Dad thank you very much for supporting me and encouraging me. Thank you for giving me all these opportunities in life, for always pulling me up when I went down. My sister Pascale, my very good friend, thank you for all the laughter and joy that you gave me and continue to. Thank you for all those overseas phone calls that you made for the mere purpose of telling me jokes and making me laugh when I needed it the most. Nana, thank you for all your love and affection. Finally, Elie, you have seen me through my three degrees. You never lost faith in me and you always pushed me to pursue my dreams even when it was at your own expense. Regardless of the circumstances, I could always reach you. Thank you my dear...

TABLE OF CONTENTS

Abstract i

Résumé ii

Acknowledgements iv

LIST OF FIGURES xi

LIST OF TABLES xxiii

CHAPTER 1. Introduction 1

1. Software Quality 2
2. Evaluation of Software Quality 3
3. Software Quality Estimation Models 7
4. Problem Statement and Objective 9
5. Contributions 10
6. Statement of Originality 11
7. Thesis Organization 11

CHAPTER 2. Background 13

1. Inductive Learning 14
2. Evaluation Criteria and Definitions 15
3. Previous Work in Building Software Quality Estimation Models 17
4. Rule-Based Classification Models 20
5. Decision Trees 21

6. C4.5 24
7. Previous Work in Building Logical Software Quality Estimation Models 26
8. Previous Work in Optimizing Existing Software Quality Estimation Models 28

CHAPTER 3. Genetic Algorithms 30
1. The Darwinian Theory Behind GAs 31
2. Definitions of Some Biological Terms 32
3. The Genetic Algorithm 33
3.1. Encoding Scheme 34
3.2. Genetic Operators 35
3.3. Selection Techniques 40
3.4. Replacement Policies 43
4. The Pros and Cons of GAs 44
5. Areas of Application 46
5.1. Different Areas of Application 46
5.2. Previous Work in Using GAs to Optimize Rule-Based Classification Models 46

CHAPTER 4. First Approach to Optimization: Deriving a Better Model from a Set of Models 51
1. An Initial Genetic Algorithm 52
1.1. Chromosomes and Fitness Function 53
1.2. The Genetic Operators 54
2. Experiments and Results 56
2.1. Results 59
2.2. Modifications to the Algorithm 62
2.3. Discussion and Summary 63
3. A Refined GA for Combining Rule Sets 64
3.1. Chromosomes and Fitness Function 65
3.2. The Genetic Operators 66

3.3. Trimming 69
4. Experiments and Results 70
4.1. Description of the Data Sets 70
4.2. SETUP I: Single Point Crossover and Roulette Wheel 75
4.3. SETUP II: Double Point Crossover and Roulette Wheel 79
4.4. SETUP III: Single Point Crossover and Rank Selection 83
4.5. SETUP IV: Double Point Crossover and Rank Selection 88
4.6. Discussion and Summary 90
CHAPTER 5. Second Approach: Optimizing a Single Model 95
1. Overview 95
2. The Algorithm 95
2.1. Chromosomes and Fitness Function 96
2.2. The Genetic Operators 97
2.3. Trimming 99
2.4. From Rules to Rule Sets 101
3. Experiments and Results 103
3.1. SETUP I 103
3.2. SETUP II 111
3.3. Comparison of SETUP I and SETUP II 114
3.4. The Effect of Crossover and Mutation Rates 116
4. Discussion and Summary 117
CHAPTER 6. Conclusion 120
1. Summary 120
2. Limitations 122
3. Future Work 122
REFERENCES 124
APPENDICES 137
APPENDIX A 137

A.1. Overview 137
A.2. Representation and Fitness Function 137
A.2.1. Crossover 139
A.2.2. Mutation 140
A.2.3. Postprocessing 142
A.3. Motivations for This Technique 144

LIST OF FIGURES

2.1 Attribute file (left-most), training file and a decision tree constructed from them. The first line in the attribute file indicates the classification labels for the task.The next ones list the attributes and their types. 22

2.2 A decision tree with its branches re-iterated as rules (left-to-right order). 25

3.1 1-Point Crossover: parent chromosomes get cut at one location and offspring interchange the tails. 36

3.2 N-Point Crossover: parent chromosomes get cut at N different locations and offspring inherit alternating segments. N=3 in this figure. 37

3.3 Uniform Crossover: offspring inherit genes from one parent versus another depending on the sequence of random numbers and the threshold value. 37

3.4 Mutation: The third and last genes of the chromosome are mutated. 38

3.5 Floating-Point Mutation: the fifth gene is mutated to a random value from the domain of the gene. 39

3.6 Inversion Mutation: The genes between the second and the fourth ones are inverted 40

3.7 Roulette Wheel Selection. The pie is split among four chromosomes of fitness: 5, 50, 10 and 75. 41

3.8 Rank Selection. The pie is split among chromosomes of fitness: 5, 50, 10 and 75. Chromosomes are ranked (ranks from 1 to 4) and portions of the pie are alloted to them by rank. 42

4.1 This is an example of a rule set constructed by C4.5 and the corresponding chromosome. The chromosome is formed of genes where each, except the last one, points to a rule in the corresponding rule set. The last gene encodes the default classification label of the rule set. 53

4.2 Example of crossover. Rule sets $RS1$ and $RS2$ are cut at gene 3. The two resulting offspring are $RS3$ and $RS4$. The latter is then mutated (mutated gene is in the shaded box). 55

4.3 Initial population of chromosomes built by C4.5. The number on each line indicates the fitness of the chromosome (accuracy of the rule set that the chromosome represents). 61

4.4 Best and worst fitness in each generation that shows an improvement over the previous one. 62

4.5 The modified mutation operator that removes a condition from a rule. Condition $NOP > 0$ is removed. 63

4.6 The modified mutation operator that changes the classification label of a rule. 64

4.7 A rule set and its chromosome representation. Special genes, representing classification labels, are underlined. 65

4.8 Single point crossover where the cut point falls on a rule boundary. 67

4.9 Single point crossover where the cut point falls within a rule. . 67

4.10 Single point crossover where the cut point falls within a rule in parent 1 and on a rule boundary in parent 2. This results in one of the offspring (offspring 1) representing an invalid rule set (missing classification label). 68

4.11 Redundancy in the first rule. The third gene represents a condition that implies the one represented by the first gene. 69

4.12 Redundancy in a rule set. The first and the third rules are the same. 69

4.13 Inconsistency in the second rule (shown in bold). The conditions $NOC > 3$ and $NOC \leq 1$ cannot be true at the same time. . . 70

4.14 MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover. 77

4.15 STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover. 77

4.16 STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover 78

4.17 MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu =$

0.1, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover. 80

4.18 STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover. 80

4.19 STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover. 81

4.20 MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover. 82

4.21 STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover. 82

4.22 STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover. 83

4.23 MAINT-Accuracy (C) and J_index (J) of a random rule set and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover. 84

4.24 STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover. 84

4.25 STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover. 85

4.26 MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover. 86

4.27 STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover. 87

4.28 STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) =$

$C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover. 87

4.29 MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover. 88

4.30 STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover. 88

4.31 STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover 89

4.32 MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover. 90

4.33 STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover. 91

4.34 STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover. 91

4.35 MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover. 92

4.36 STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover. 92

4.37 STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover. 93

4.38 STAB2-Learning curves showing the accuracy on the training set for both the roulette-wheel (left hand side) and the rank-selection (right hand side) techniques. 94

5.1 This is a rule set with three rules and a default classification label. This constitutes a population of three chromosomes. The first and the second one have three genes each and the third one has two. 96

5.2 Crossover where cut points fall in different locations within a pair of chromosomes. Offspring have the same lengths as their parents. 98

5.3 Crossover where cut points fall in different locations within a pair of chromosomes. Offspring have different lengths from their parents. 98

5.4 A chromosome before and after mutation. First and last genes are mutated. 99

5.5 Crossover and mutation that result in a redundant rule (*offspring 1*) and an inconsistent rule (*offspring 2*). The condition in bold is the mutated gene. 99

5.6 A chromosome before and after trimming. Condition $MDS <= 3$ is deleted because $MDS <= 1$ implies $MDS <= 3$. 100

5.7 A rule set before after being pruned. Rule 0 and Rule 1 are deleted because the former has a duplicate in the rule set (Rule 3) and the chromosome representing Rule 1 has a fitness equal to 0. . 102

5.8 STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential. 105

5.9 STAB1-Accuracy of the constant classifier (CF) and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential. 105

5.10 STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$,

$f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential. 106

5.11 MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential. 107

5.12 STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential. 110

5.13 STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential. 111

5.14 MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential. 111

5.15 STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential. 112

5.16 STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential. 113

5.17 MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential. 114

5.18 STAB1- Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential. 115

5.19 STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets.Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$,

selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential. 115

5.20 MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential. 117

5.21 STAB2-Different experiments seeded with the same rule set. Each bar shows the results of one experiment. Each experiment is run 30 times and the average and standard deviation are reported for each. The first bar shows the accuracy of the initial rule set. . 118

5.22 STAB2-Different experiments seeded with the same rule set. Each bar shows the results of one experiment. Each experiment is run 30 times and the average and standard deviation are reported for each. The first bar shows the J_index of the initial rule set. . . 118

A.2.1 An example of a graph and a bipartite graph. 138

A.2.2 A rule set and its bipartite graph representation. Each rule in the rule set is represented with a different type of line in the graph. A node representing a classification label is assigned to each rule. The default classification label of the rule set is saved in a separate field. 139

A.2.3 Crossover. Exchanging colors (line types) between two edges results in the two conditions in bold being swapped. 141

A.2.4 Weight mutation operator. Changing the weight of edge (LCOMB, 22, 1, 1) from 1 to 0 changes the condition $LCOMB > 22$ to $LCOMB \leq 22$ in the first rule. 141

A.2.5 Value mutation operator. Replacing a vertex value with another replaces the condition $NPPM \leq 10$ with $NPPM \leq 5$. 142

A.2.6 Class mutation operator. The vertex representing the mutated classification label is highlighted. 143

LIST OF TABLES

1.1 Three object-oriented metrics and their definition. 7

2.1 Examples for the learning task *Is the class fault-prone?*. 14

4.1 Inheritance Metrics. 58

4.2 Cohesion Metrics. 59

4.3 Coupling Metrics. 60

4.4 Software systems used to build decision trees and rule sets with C4.5. 71

4.5 Software quality metrics used as attributes in **STAB1**. 73

4.6 STAB1- Software systems used to train and test the GA. . . . 73

4.7 Software systems used to build decision trees and rule sets with C4.5. 74

4.8 Software quality metrics used as attributes in **STAB2**. 75

4.9 Experiment SETUP I: Single Point Crossover and Roulette Wheel. 76

4.10 Experiment Setup II: Double Point Crossover and Roulette Wheel 81

4.11 Experiment Setup III: Single Point Crossover and Rank Selection 85

4.12 Experiment Setup IV: Double Point Crossover and Rank Selection 89

4.13 Summary of the results obtained with the GA in the four different setups. For each setup, values are recorded for the accuracy on the

training set (C), the accuracy on the testing set (c)), the J_index on the training set (J) and the J_index on the testing set (j). . 93

5.1 Data sets used in experiments. The first two contain data about the stability of software components (classes) in an object-oriented system and the last contains data about maintainability. 104

5.2 Experiment SETUP I: $f(l) = C(l) * t(l)$. 104

5.3 Experiment SETUP II: $f(l) = C(l) * t(l) + (1 - t(l)) * C(R)$. . 112

CHAPTER 1

Introduction

Assessing software quality is fundamental in the software developing field as it helps reduce costs, time and effort. However, certain important quality characteristics (such as maintainability, reliability, reusability, etc.) cannot be measured before the system is used for a certain period of time. Nonetheless, they can be estimated based on software attributes which can be measured during the design and implementation process (e.g. cohesion, coupling, etc.). Many metrics have been proposed in the literature for this purpose. Software quality estimation models are built to estimate unmeasurable characteristics based on measurable attributes.

Many software quality estimation models have been built and used by companies such as NASA and HP. However, they all suffer from degradation of their predictions when they are applied to new data. This is largely due to the lack of representative samples that can be drawn from available data in the domain of software quality. Unlike other fields where public repositories abound with data, software quality data is usually scarce, because it takes a lot of effort to produce labelled data sets in which a person has annotated the data with the labels corresponding to the unmeasurable characteristics. Moreover, data is often system-specific, and there is a lot of variability in the metrics and labelling produced for different software elements. Also, building a measurement program incurs high costs and many companies are not willing to share the data once it has been collected. Because software quality estimation models are

typically built from data using machine learning or statistical techniques, it becomes hard to build estimation models that maintain their accuracy (or remain close to it) when used to make predictions for new data.

In this thesis, we attempt to solve this problem by proposing a strategy that allows existing software quality estimation models to adapt in order to provide better predictions on new data. The main contribution of this thesis is a genetic algorithm-based strategy to adapt one or more software quality estimation models built on one data set to another set. We propose two approaches:

- Combining and adapting different software quality estimation models to a set of data.
- Adapting a software quality estimation model to a set of data.

In practice, the first approach can be seen as combining the expertise of several software quality estimation models, built from common domain knowledge, and adapting the combined models to context-specific data. The second can be seen as taking an already-existing model, built from common domain knowledge (for example) and adapting it to a specific context. In both approaches, the initial model(s) is (are) already existent and the goal is to transfer the knowledge already acquired into the new adapted model(s).

The rest of this chapter is organized as follows: in Section 1, we define software quality. In Section 2, we describe how software quality is evaluated. Section 3 introduces software quality estimation models. Section 4 describes the problem tackled in this thesis. Section 5 describes the contributions of our work. Section 6 gives the statement of originality of the work, and Section 7 gives the organization of the thesis.

1. Software Quality

Before discussing software quality estimation, perhaps the first question to address is *what exactly is software quality?* Galin gives two definitions for the phrase [Galin, 2004]. The first is the IEEE definition [IEEE, 1991]:

```
Software quality is:
1. The degree to which a system, component, or process meets specified
requirements.
2. The degree to which a system, component, or process meets customer
or user needs or expectations.
```

The second definition is in reference to Pressman [Pressman, 1997]:

```
Software quality is:
Conformance to explicitly stated functional and performance requirements,
explicitly documented development standards, and implicit characteristics
that are expected of all professionally developed software.
```

While these definitions are slightly different, they agree on the point that software quality is not one specific characteristic of a software artifact, but a combination of characteristics.

2. Evaluation of Software Quality

The quality of a software system is evaluated in terms of characteristics such as maintainability, reusability, reliability, etc. Below, we define these terms as this will help us understand more what quality is about.

According to Pressman [Pressman, 1997], **maintainability** is " the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements." In [Krueger, 1992], **reusability** or **software reuse** is defined as " the process of using existing software artifacts instead of building them from scratch. [Ghezzi *et al.*, 2003] adopt the definition of **reliability** as "the probability that the software will operate as expected over a specified time interval." ISO/IEC 9126 offers a framework for the

evaluation of software quality. It defines six product quality characteristics and a suggestion of dividing them into quality subcharacteristics. Namely, these characteristics are: functionality, reliability, usability, efficiency, maintainability and portability. Below, we give the definitions of these characteristics as found in [ISO/IEC9126, 1991].

`functionality: the capability of the software to provide functions which meet stated and implied needs when the software is used under specified conditions.`

`reliability: the capability of the software to maintain its level of performance when used under specified conditions.`

`usability: the capability of the software to be understood, learned, used and liked by the user, when used under specified conditions.`

`efficiency: the capability of the software to provide the required performance, relative to the amount of resources used, under stated conditions.`

`maintainability: the capability of the software to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.`

`portability: the capability of software to be transferred from one environment to another.`

Most of these characteristics cannot be measured before the system is used for a certain period of time. However, there exist some software attributes that can be measured during the software development cycle, and which can be used as indicators

of them. Examples include cohesion and coupling. **Cohesion** is characterized by how closely the local methods are related to the local instance variables of the class in an object-oriented system [Fenton and Pfleeger, 1997]. In other words, it is the degree to which the elements within the same class are linked. **Coupling** refers to the degree of interdependence among the classes of a software system [Briand *et al.*, 1997]. In [Chidamber and Kemerer, 1994], it is defined as "any evidence of a method of one object using methods or instance variables of another object". To illustrate how certain software quality attributes can be indicators of some quality characteristics, we take the example of the reusability of a software component (a class in an object-oriented system, for example). Assessing the reusability of a software product is very important because it helps produce high quality software more quickly [Basili *et al.*, 1996]. It is not possible to directly measure reusability. However, the complexity (measured in terms of the complexity of the underlying algorithm or the complexity of the problem to solve, for example) and volume (amount of computer storage necessary for a uniform binary encoding [Fenton and Pfleeger, 1997]) of a software component can be a good indicator of its reusability. As a matter of fact, some studies have shown that highly reused components tend to have complexity and volume measures lower than those of less reused components [Mao *et al.*, 1998]. Hence, it makes sense to use the complexity of a component or its volume to estimate its reusability. Complexity is a directly measurable quality. Reusability is not. This underlines the importance of metrics. The *IEEE Standard Glossary of Software Engineering Terms* [IEEE, 1993] defines **metric** as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute." According to El-Emam, software product metrics are objective measures of the structure of software artifacts in the sense that repeated measurements of the same (unchanged) software artifact yield the same values [Erdogmus and Tanir, 2002]. In his well-cited book, *Software Engineering: A Practitioner's Approach*, Pressman also stresses the objectivity that measurement gives to the evaluation process [Pressman, 1997]:

```
If you don't measure, judgement can be based only on subjective
evaluation.  With measurement, trends (either good or bad) can be
spotted, better  estimates can be made, and true improvement can be
accomplished over time.
```

Metrics have been proposed and used to measure software elements and development processes. In our discussion, we are interested in the metrics used for software elements.

Several works have proposed metrics to measure the quality of software in object-oriented design. The most popular, to date, remain the CK metrics proposed by Chidamber and Kemerer [Chidamber and Kemerer, 1994]. These were intended to capture cohesion, complexity, coupling and depth of inheritance in object-oriented design. In object-oriented design, classes can be derived from others. In this context, a tree can be drawn where inheriting classes appear as children of the classes from which they inherit. These trees are called **inheritance trees**. [Basili *et al.*, 1996] found that most of the CK metrics were useful for predicting fault-proneness of classes during the design phase (fault-proneness indicates a high maintainability cost). Similarly, [Briand *et al.*, 1999] investigate the usefulness of coupling and cohesion metrics in estimating the fault-proneness of a class. [Demeyer and Ducasse, 1999] show that inheritance and size metrics are good indicators of stability but not reliable for problem detection. In [Briand *et al.*, 1996] and [Briand *et al.*, 1997], the authors define metrics that capture different types of coupling. In [Martin-Albo *et al.*, 2003], an initial version of a classification model for metrics of software components (CQM-Component Quality Model) is proposed. Other metrics were proposed in [Henderson-Sellers, 1991], [Henderson-Sellers, 1996], [Li and Henry, 1993b], [Li and Henry, 1993a], [Coppick and Cheatham, 1992], [Barnes and Swim, 1993] and [Lorenz and Kidd, 1994]. A good survey on metrics and how to measure is [Riguzzi, 1996]. Below, we give a very brief table with the description of a few of these metrics for the mere purpose of de-mistifying what is meant by a software quality metric.

Name	**Description**	**Definition**
DIT	Depth of Inheritance Tree	Length of the longest path from the root to the class in the inheritance tree.
NOC	Number of Children	Number of classes that inherit from a particular class.
NOA	Number of Ancestors	Number of ancestor classes that a particular class has.

TABLE 1.1. Three object-oriented metrics and their definition.

3. Software Quality Estimation Models

Building a software quality estimation model consists of building a relationship between what is directly measurable (complexity, for example) and other qualities that are not directly measurable (reusability, for example)[1]. Many software quality estimation models have been proposed in the literature, and companies have largely adopted them in order to improve their software development process [Sahraoui *et al.*, 2000a]. These take different forms: statistical models, decision trees, rule sets, etc. In all cases, they are used to predict the value of a variable, called **dependent variable**, based on the value of one or more other variables, called **independent variables**. We are interested in rule-based software quality estimation models because of their white-box nature and their ease to understand as we will describe later.

For example, consider the task of estimating the fault-proneness of a class in an object-oriented system. Assume that the task consists of estimating whether a class is fault-prone or not. Let us denote by 0 the fact that a given class is fault-prone and by 1 the fact that it is not. These are called **classification labels**. Suppose, furthermore, that the estimation is based on three metrics: NOC (Number of Classes that inherit from the specified class), NOM (Number of Methods in the class) and DIT (Depth in Inheritance tree). A rule-based estimation model for this task can look like the following:

[1]We are not claiming that this relationship between the two is of the *if-then* type. We are stating that one can be used as an indicator of the other.

$$Rule1: \quad NOM \leq 3 \wedge NOC > 2 \rightarrow 1$$

$$Rule2: \quad NOM > 4 \wedge DIT > 2 \rightarrow 0$$

Default class: 1

Briefly, the estimation model shown above consists of two rules and a default classification label. The first rule estimates that if a class, in an object-oriented system, contains 3 methods or fewer and the number of classes that inherit from it is more than 2 then it is not fault-prone. The second one estimates that if a class contains more than 4 methods, and it is at a depth level more than 2 in the inheritance tree, then it is fault-prone. For all classes that do not satisfy any of the rules, they are estimated to be not fault-prone (default class 1).

Now, consider that this estimation model is used to estimate the fault-proneness of the following three classes:

class 1 has: NOM = 2, NOC = 3 and DIT = 3
class 2 has: NOM = 5, NOC = 1 and DIT = 3
class 3 has: NOM = 4, NOC = 2 and DIT = 1

When presented with the classes, the estimation model starts with the first one, *class 1*, which satisfies the two conditions in *Rule* 1; hence the rule assigns to it its label (1) and the model estimates this class not to be fault-prone. *class 2* is then presented to the estimation model, the class fails to satisfy the first condition of *Rule* 1 ($NOM = 5$ hence, the condition $NOM \leq 3$ is false) so it is immediately passed on to *Rule* 2. The class satisfies both conditions of *Rule* 2 and hence, it is estimated to be fault-prone (label 0). Finally, *class 3* is presented to the model, it fails to satisfy any of the rules and hence the estimation model attributes to it the default label (1) and the class is estimated not to be fault-prone.

As one can see, such estimation models are easy to interpret by human experts. In fact, they allow to better find where to restructure the software to improve its quality. This has earned them a wide use in the field of software quality. In Chapter 2, we give a review of the work that has been done in building software quality estimation models, in general, and rule-based software quality estimation models, in particular. We also cover the work that has been done in optimizing software quality estimation models. In the same chapter, we describe the process of building rule-based quality estimation models (and we describe the machine learning algorithm C4.5 [Quinlan, 1993] used to construct such models). Often, these models leave much room for improvement in their accuracy (percentage of correctly classified classes) especially when the models are used to classify new unseen data (we will focus more on this issue in Chapter 2). Improving this aspect will be a core concern of this thesis.

4. Problem Statement and Objective

The problem tackled in this thesis is the improvement of already existing software quality estimation models in order to use them to classify new, unseen data. The lack of data from which a representative sample can be drawn, in the domain of software quality, makes the process of building a representative software quality estimation model difficult. This is mainly due to the fact that collecting the data is costly in terms of time and money and hence, such data bases become proprietary. As a result, the estimation accuracy of software quality estimation models built from one data set (taken from a specific domain or context) deteriorates as these are used to classify new, unseen data (taken from another domain)[2]. [Sahraoui *et al.*, 2001] show that this is due to the threshold values included in the models being too specific to the learning sample.

Ideally, we want to build models from available data and use them to classify new data. The accuracy of the models, when predicting the new data should not drop significantly. The importance of the reuse of existing models emerges from two

[2]For a detailed explanation of how estimation models are built, the user can refer to Chapter 2, Section 1.

different facts. First, it saves the time needed to collect data from which the models are initially built. Second, it allows the incorporation of the knowledge acquired at the time the models were built into the new models.

The questions that motivate this thesis are the following:

- How can several software quality estimation models be combined in a way that gives one or more better[3] models when used to classify a different data set?
- How can a software quality estimation model (built from one data set) be adapted to classify a new data set with a high accuracy?

Both situations can emerge in practice and in both cases, the core idea is to keep the expertise of the original model(s) acquired at the time they were built. Since the search space is very large and exhaustive search or local search methods are not efficient in such situations, we present a genetic algorithm-based technique that considers two approaches:

- Using a genetic algorithm to combine several rule sets and adapt them to a new data set.
- Using a genetic algorithm to adapt an existing rule set to a new set of data.

In both cases, the objective is to obtain a rule set that is more accurate than the initial rule sets.

5. Contributions

The main contribution of this thesis is a genetic algorithm-based technique to optimize software quality estimation models. The technique is tested and validated on three real data sets. Two of them are used to estimate stability of classes in object-oriented software systems (defined in Chapter 4, Section 4) and one to estimate fault-proneness (defined in Chapter 4, Section 2) in C++ classes.

Our technique uses rule-based predictive models and the end result is of the same form. The advantage of this approach is the white-box nature of the model. The rule

[3]Better in terms of prediction accuracy.

set has a double utility. It can be used to assess the desired software quality as well as to provide practitioners with guidelines to follow at the design stage.

Our technique allows the incorporation of past knowledge learnt from one data set into software quality estimation models that are used on a different data set. This is particularly useful in a field like software quality where data from which representative samples can be drawn is not readily-available. Our approach can also be used to take rule sets built from common domain knowledge and adapt them to specific context data.

Although assessed on data from the object-oriented paradigm, our technique remains general and can be easily adapted to apply to other software systems as well. Also, it is a general technique that is not tied to any specific type of rules or attributes.

6. Statement of Originality

Portions of the results published in this thesis have appeared in [Sahraoui and Azar, 1999], [Azar *et al.*, 2002] and [Bouktif *et al.*, 2004]. In particular, the algorithm and the results in Chapter 4, Section 2 have appeared previously in [Sahraoui and Azar, 1999]. An initial version of the algorithm described in Chapter 5 has appeared in [Azar *et al.*, 2002] and [Bouktif *et al.*, 2004]. These two papers contain results not published in this thesis (the latter contains an improvement of the algorithm). They also compare our approach that adapts a single model to a new data set to an independent work done by Sahraoui, Bouktif and Kégl, which aims at combining different models. The two papers have been co-authored with them.

7. Thesis Organization

The remainder of the thesis is organized as follows. In Chapter 2, we present background material and previous work in building software quality estimation models. In Chapter 3, we give an overview of genetic algorithms. Chapters 4 and 5 contain our main contributions. In Chapter 4, we design and evaluate genetic algorithms that

take several rule sets, combine them and adapt them to a new data set. The genetic algorithm described in Chapter 5 works on a single rule set and adapts it to a new data set. In Chapter 6, we conclude the thesis with final remarks and future work.

CHAPTER 2

Background

In this chapter, we describe previous work done in building and optimizing software quality estimation models. Such models have been used extensively, and most such models are built using machine learning techniques. Some machine learning techniques allow building models that are easy to interpret, such as decisions trees or rule sets. This is in contrast with some statistical techniques, such as regression, which yield models that are harder to inspect. Interpretability is of major importance in software quality models, because we want such models to provide guidelines that are useful in the software development stage. In this thesis, we will focus specifically on rule-based models, which are easy to interpret.

The chapter is organized as follows. In sections 1 and 2, we give an overview of inductive learning and define the measures that are used to evaluate software quality estimation models. In Section 3, we review previous work that has been done in building software quality estimation models, focusing on work that compares different types of models. In Section 4, we describe rule-based models, the kinds of models that we will focus on in this thesis. Sections 5 and 6 describe decision trees and the algorithm C4.5, a state-of-art algorithm for constructing them. We describe the way in which decisions trees can be transformed into rule sets, because such rule sets will be used later in our experiments. In Section 7, we describe previous work using

decision trees and rule-based models for software quality estimation. In Section 8, we describe existing work on optimizing rule-based software quality estimation models.

1. Inductive Learning

In his book *Machine Learning*, Tom Mitchell defines **learning** to include any computer program that improves its performance at some task through experience [Mitchell, 1997]. One paradigm of learning is **inductive learning** which consists of finding a general description of a concept based on a set of examples [Carbonell, 1990]. Examples are usually described by a set of **attributes** and the learning task consists of predicting a **label** based on the values of the attributes for each example.

Consider, for example, the task of predicting whether a class in an object-oriented system is fault-prone or not. Suppose, for the sake of simplicity, that this depends on three attributes of the class namely, the number of classes that inherit from this class, the depth of inheritance of the class and the number of methods in this class. The three metrics that measure these three characteristics are: NOC, DIT and NOM respectively. Table 2.1 shows a set of examples of classes described by the values that they have for these attributes. Each row in the table describes a class in an object-oriented system or a **case**. Each entry in a row, except the last one, is a value that the class has for the corresponding attribute and the last entry indicates the label of the class (whether it is fault-prone or not).

NOC	***DIT***	***NOM***	***fault-prone***
3	2	4	No
4	3	15	Yes
3	3	5	No
5	6	10	Yes

TABLE 2.1. Examples for the learning task *Is the class fault-prone?*.

The learning task consists of learning from these examples to predict under what circumstances a class in an object-oriented system is more likely to be fault-prone.

When the label is discrete (as is the case in the example above), the task is known as a **classification** task; when it is continuous, the problem is called **regression**. Our focus in this thesis is on classification tasks.

Classification models can be built using statistical methods [Hunt, 1975] or machine learning-based methods. In the latter case, models can take many different forms, e.g. decision trees, rule-based models, neural networks, instance-based classifiers, etc. We will focus on rules and decision trees in this thesis because such models are easy to inspect and translate into guidelines for software development. Among the algorithms that construct rules or decision trees are NEWID [Boswell, 1990], CN2 [Clark and Niblett, 1989], C4.5 [Quinlan, 1993] and FOIL [Quinlan, 1990], to name a few. The first three algorithms use an attribute-value-based description language (similar to Table 2.1) to construct classification models. In other words, they allow tests involving comparisons of the individual attributes and their possible values. FOIL is the only system that allows relationships between attributes as well. While the expressive power of FOIL remains higher than the other algorithms, preparing the data for the algorithm is more complex. For more detail on these and other machine-learning algorithms, we refer the reader to [Mitchell, 1997].

2. Evaluation Criteria and Definitions

In most classification tasks, the evaluation criteria depend on the percentage of the examples (cases) that are correctly classified in the set of all examples. Below, we define the evaluation criteria that are commonly used when building software quality estimation models and the criteria that we will use in this thesis.

DEFINITION 2.1. *The **real classification label** of a case is the label that has been recorded during data collection (through an automated process or by human experts).*

DEFINITION 2.2. *The **predicted classification label** of a case is the label predicted by the classification model.*

DEFINITION 2.3. *The **confusion matrix** of a classification model, M, computed on a data set D, is an $n \times n$ table, T, where each number n_{ij} in entry $T(i,j)$ indicates the number of cases in D with a real classification label i and predicted classification label j.*

		predicted label			
		c_1	c_2	$\dots$	c_k
	c_1	n_{11}	n_{12}	$\dots$	n_{1k}
real	c_2	n_{21}	n_{22}	$\dots$	n_{2k}
label	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	c_k	n_{k1}	n_{k2}	$\dots$	n_{kk}

Note that the diagonal elements in this matrix indicate how many examples are classified correctly whereas off-diagonal elements indicate mistakes.

DEFINITION 2.4. *The **accuracy**, $C(M)$, of a classification model M, computed on a data set D, is the percentage of cases in D correctly classified by M.*

$$C(M) = \frac{\sum_{i=1}^{k} n_{ii}}{\sum_{i=1}^{k} \sum_{j=1}^{k} n_{ij}}. \tag{2.1}$$

DEFINITION 2.5. *The **error rate**, $E(M)$, of a classification model M computed on a data set D is the percentage of cases in D incorrectly classified by M.*

$$E(M) = 1 - C(M). \tag{2.2}$$

Most classification algorithms are designed to maximize accuracy. However, in the case in which one classification label appears much more frequently than the others, accuracy is not a good indicator of performance. For instance, suppose that 97 out of 100 cases are of class 1, then, the majority classifier, which assigns to new examples the most frequent label from the training set, will have very high accuracy although it is only doing a good job for one class. This problem is especially important if misclassifications for the less frequent classes are more costly. An alternative measure aimed at capturing such situations is J_index [Youden, 1998].

Definition 2.6. *The **J_index**, $J(M)$, of a classification model M computed on a data set D is the average accuracy of M per class label.*

$$J(M) = \frac{1}{k}\sum_{i=1}^{k}\frac{n_{ii}}{\sum_{j=1}^{k} n_{ij}}. \tag{2.3}$$

3. Previous Work in Building Software Quality Estimation Models

The software engineering literature is rich with work that proposes different techniques for building software quality estimation models. In particular, [Jorgensen, 1995] and [Briand *et al.*, 1993] use machine learning algorithms to build models that estimate corrective maintenance cost. **Corrective maintenance** has to do with the removal of residual errors that are present in the product when it is delivered, as well as errors introduced into the software during its maintenance. Jorgensen compares three approaches: regression, feed-forward neural networks with back-propagation [Mitchell, 1997] and pattern recognition (with the Optimal Set Reduction Method from [Briand *et al.*, 1992]). He considers the task of constructing a predictive model of the cost of corrective maintenance, based on one metric, *LOC* (Line of Code). This metric does not reflect most of the tasks that are involved in maintaining the system, but at the time, the only other metric available was function points, which is not a meaningful predictor on some small maintenance tasks. Pattern recognition proved to be more accurate than neural networks and regression. Also, this technique gives some insight into the maintenance process while models produced by neural networks and regression are black-box: they give the classification label only without allowing the inspection of the concept learned. Inspection is very important as it allows experts to draw guidelines that can be incorporated in future development procedures. These guidelines will help reduce undesired quality characteristics. In [Briand *et al.*, 1993], logistic regression is compared to Optimized Set Reduction (OSR). The work investigates the use of both techniques to build models that can help identify "high risk" components in several Ada systems. Systems were evaluated

according to their correctness (percentage of correct classifications when a component is classified as high risk) and completeness (percentage of high risk components that are classified as such by the model). OSR achieved a higher average correctness and completeness than logistic regression. Similar to our work, there is a strong emphasis on the ability to interpret the obtained classification models (OSR models are easily interpretable whereas logistic regression models lack this characteristic). In the same vein, [De Almeida *et al.*, 1999] analyze five public domain machine learning algorithms: C4.5, C4.5rule, CN2 [Clark and Niblett, 1989], NewID [Boswell, 1990] and FOIL [Quinlan, 1990] . The algorithms are compared on the task of predicting the difficulty to correct faulty Ada programs. The testbed of the experiments is data collected on corrective maintenance activities for the Generalized Support Software reuse asset library located at the Flight Dynamics Division of NASA's Goddard Space Flight Center (GSFC). FOIL showed a higher overall accuracy than the other algorithms. Also, FOIL has better expressive capabilities than the other algorithms since it allows relations between different attributes (as opposed to C4.5, for example, which establishes relations between attributes and values). However, preparing the data for FOIL remains much more complicated than preparing it for C4.5. In [Shepperd and Kadoda, 2001], three prediction techniques are compared namely, stepwise regression, rule induction and case-base reasoning [Mitchell, 1997] using simulated data. The work shows that the results depend on the characteristics of the dataset (size of the training set, number of metrics, type of distribution, etc.). It suggests that such characteristics should be taken into account when choosing a particular technique.

In [Lanubile and Visaggio, 1996], a comparative empirical study was conducted to assess the performance of several techniques for predicting the quality of software components (in this work, a software component refers to a functional abstraction of code such as a procedure, a function or the main program). The techniques considered were discriminant analysis, logistic regression, C4.5, layered neural networks and holographic networks. In addition to these models, discriminant analysis and logistic

regression were combined with principal component analysis. Each model was trained to produce a binary classification of software quality as high-risk or low-risk. Similarly to [Shepperd and Kadoda, 2001], the authors discussed the fact that the predictive power of any model is first of all dependent on the quality of the training data set. The techniques under consideration were compared against each other according to the following performance characteristics: completeness (percentage of high-risk software components that have been actually classified as such by the model), Type 1 misclassification rate (ratio of high-risk software components that were classified as low risk to the total number of components) and Type 2 misclassification rate (ratio of low-risk components classified as high-risk to the total number of components). Finally, verification cost was measured by two criteria: Inspection Cost (percentage of software components classified as high-risk and thus sent for inspection) and Wasted Inspection (percentage of components that were classified as high risk but were indeed low-risk). Among the models that were not combined with principal component analysis, the classification models (obtained with C4.5) and holographic networks achieved the best results in terms of completeness (47.37%) while the former also had a lower rate of Type 2 misclassification rate. These two models also had the lowest Wasted Inspection cost. The performance of other models was around 5% lower in terms of Completeness. The best performance in terms of completeness was achieved by the combination of the principal component analysis and logistic regression[1] followed by the combination of principal component analysis and discriminant analysis (73.68% and 68.42% respectively). However, high completeness of the two combined models was achieved only because these models classified many software components as high-risk resulting in relatively high Wasted Inspection cost and a very high Type 2 misclassification rate.

In [Cohen and Devanbu, 1997], two inductive logic programming (ILP) methods, FOIL and FLIPPER [Cohen, 1995] were used for predicting fault density in C++ classes. Both of them learn function-free Prolog predicate definitions from examples.

[1]Principle component analysis was first applied to reduce the complexity of the feature space from 11 to 3 (11 attributes were grouped into 3 components).

FLIPPER was found to achieve lower error rates than FOIL, especially when there is noise in the data (verified on some artificial datasets). Observed performance differences between the two ILP systems were attributed mainly to differences in pruning strategies (defined in Section 5).

Other techniques have also been used to build software quality estimation models such as Dempster-Shafer belief networks [Guo *et al.*, 2003], unsupervised learning [Zhong *et al.*, 2004], etc. but the majority of the work done in this arena relied on supervised learning.

4. Rule-Based Classification Models

Rule-based classification models are among the easiest to interpret by human experts. For this, they have acquired a wide popularity in the domain of software quality estimation. A **rule-based classification model** or a **rule set** is a list of rules with, sometimes, a default classification label. A **rule** has the form $L \rightarrow R$ where the left hand side (L) is a conjunction of attribute tests (one or more attribute tests that are combined with the logical operator AND) and the right-hand side, R, is a classification label. To illustrate, we repeat the example given in Chapter 1, Section 3, of classifying classes in an object-oriented system as being fault-prone or not. The attributes in this example are: NOC (Number of Classes that inherit from the class), NOM (Number of Methods in the class) and DIT (Depth in the Inheritance tree). The rule set looks as follows:

$$Rule1: \quad NOM \leq 3 \wedge NOC > 2 \rightarrow 1$$
$$Rule2: \quad NOM > 4 \wedge DIT > 2 \rightarrow 0$$

Default class: 1

We say that a rule **fires** on a case or **classifies** a case if the latter satisfies the left hand side of the rule (i.e. it satisfies all the conditions in the left hand side

of the rule). There are two possible schemes that a rule set can follow in order to classify a case. In the **sequential scheme**, the rules in the rule set are considered in a top-down sequential order and the top-most rule whose left hand side matches a case fires. If no rule fires, the rule set classifies the case by the default classification label. In the **voting** scheme, the classification label of all rules that match a case is considered and the case is classified by the label that receives the majority vote from all the rules. Here, also, the default classification of the rule set is applied when no rules in the rule set fire. Variations of the voting scheme allow votes to be weighted. The sequential scheme has the advantage of being faster since not all rules have to be matched with a case. The advantage of the voting scheme is that the order of the rules in the rule set does not matter and there is no bias against rules that appear towards the bottom of the rule set.

There are many algorithms that can be used to construct rule-based classification models. One such approach is to first construct decision trees then convert them into rules. We describe this approach in detail in the next section.

5. Decision Trees

A **decision tree** is a classifier that has the structure of a tree where each internal node is a test involving one attribute and each leaf node is a classification label. A branch in the tree indicates a value for the outcome of the test [2] in the internal node from which the branch emerges. The tree is then used to classify a set of unseen data or a **testing set**. Figure 2.1 shows an attribute file and a training set from which a decision tree is constructed.

When classifying a case, the tree is traversed in a top-down order, starting at the root. At each internal node, the test is evaluated and the search is directed through one path according to the outcome. This proceeds until a leaf node is reached. Then, the classification label at this node is attributed to the case.

[2] The outcomes of a test are all mutually exclusive.

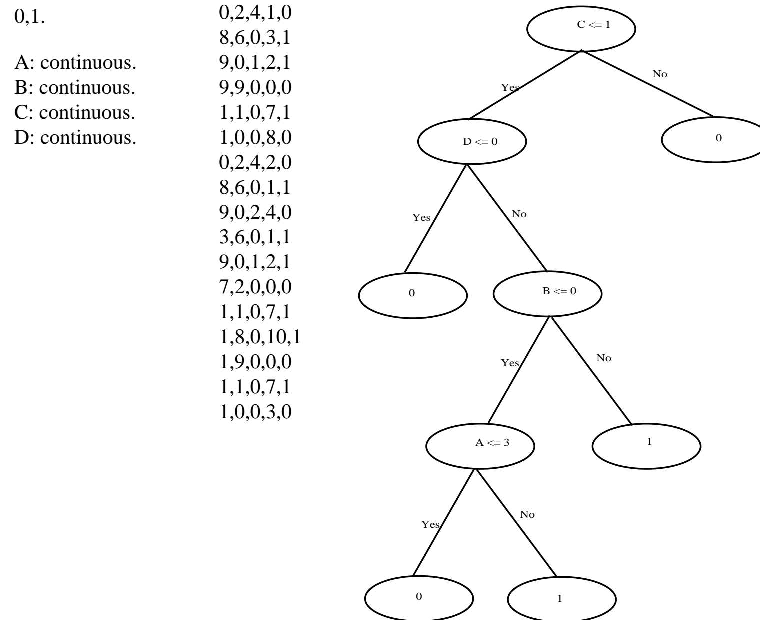

FIGURE 2.1. Attribute file (left-most), training file and a decision tree constructed from them. The first line in the attribute file indicates the classification labels for the task.The next ones list the attributes and their types.

The algorithm for building the tree is based on the divide-and-conquer methodology and is as follows:

```
If    there are no cases in the training set,
      create a leaf node and label it using some other knowledge
      source.
If    all cases in the training set are of the same category,
      create a leaf node and label it with the name of this category.
else  select one attribute,
      perform a test based on this attribute,
      divide the training set into subsets, each associated with one
      possible value of the test outcome.
      repeat the same algorithm above with each subset of the training
      set.
```

The choice of the attribute test is usually done according to the information gain measure which assesses how well a given attribute, by itself, separates the training

examples according to the classification desired. This relies on the measure of entropy. Given a Boolean classification problem and a subset, S, of examples, the **entropy** of S relative to the Boolean classification of the examples is [Mitchell, 1997]:

$$Entropy(S) = -p_{\oplus} log_2 p_{\oplus} - p_{\ominus} log_2 p_{\ominus} \tag{2.4}$$

where $p_{\oplus}$ is the proportion of positive examples in S and $p_{\ominus}$ the proportion of negative examples. 0 log 0 is defined to be equal to 0. If the classification can take on c values (where $c > 2$) and p_i is the proportion of examples in S belonging to class i then the entropy of S is defined as:

$$Entropy(S) = \sum_{i=1}^{c} -p_i log_2 p_i \tag{2.5}$$

The entropy of a subset S measures the average amount of information needed to identify the classification label of a random case drawn from S.

One problem exhibited by most decision trees is overfitting. According to Mitchell, a decision tree is said to **overfit** the training set if some other tree that fits the training set less well actually performs better over the entire distribution of the cases (training and testing sets) [Mitchell, 1997]. Several techniques have been developed in order to prevent overfitting. The most successful one is pruning. **Pruning** of a decision tree consists of replacing one or more subtrees with branches or leaves without compromising the accuracy measured on the testing set.

As we can see from Figure 2.1, decision trees are relatively easy to understand by human experts. However, rule-based classification models remain easier to understand and can be easily derived from decision trees by re-iterating each path from the root to a leaf as a rule[3]. The rule is formed as a conjunction of the attribute tests along the path and the classification label indicated by the leaf. Figure 2.2 shows an example of a tree and its branches re-iterated as rules. The left-most branch is re-iterated at the top of the list and the right-most branch at the bottom.

[3]Some algorithms directly generate such models without going through the intermediate step of generating a decision tree.

The state-of-art algorithm used to construct decision trees is Quinlan's C4.5 [Quinlan, 1993] that we describe in the next section.

6. C4.5

The core of C4.5 is the divide-and-conquer based algorithm described in the previous section. C4.5 can be run with the **windowing technique**. The idea behind this technique is to select a subset of the training cases and build a tree from it. This tree is then used to classify the cases that were not included in the window. Some of these will be misclassified. A selection of the misclassified cases is added to the window and the process is repeated until the tree built from the cases in the window classifies all the cases outside it, or it appears that no more improvement occurs. Multiple runs of C4.5 with the windowing technique will yield multiple decision trees. When it was first introduced in ID3 (the precursor of C4.5), the purpose of the windowing technique was to overcome the small memory size by loading only a subset of the cases in the main memory. However, when C4.5 came to light, memory was not an issue anymore. Windowing was kept because of two major advantages: 1. It has proven to produce trees with good accuracy when used on imbalanced data sets. 2. It gives the possibility to grow several alternative trees from the same data set (since the initial window is a selection of random cases biased towards preserving the distribution of the classes). C4.5 can convert trees to rules. An example is shown in Figure 2.2.

It is possible for C4.5 to perform pruning on the rules generated from the trees. One result is that these rules might end up being non-exclusive or non-exhaustive. i.e. some cases will satisfy more than one rule and others none. To solve the second problem, the algorithm attributes a default classification label to the rule-based model by choosing the label which applies to the most training cases not covered by any rule. Ties are resolved in favor of the class with the highest absolute frequency. C4.5 sorts the rules by their classification label. This has two advantages: 1. the rule set becomes more comprehensible by human interpreters 2. the order of rules for the same class label does not matter anymore.

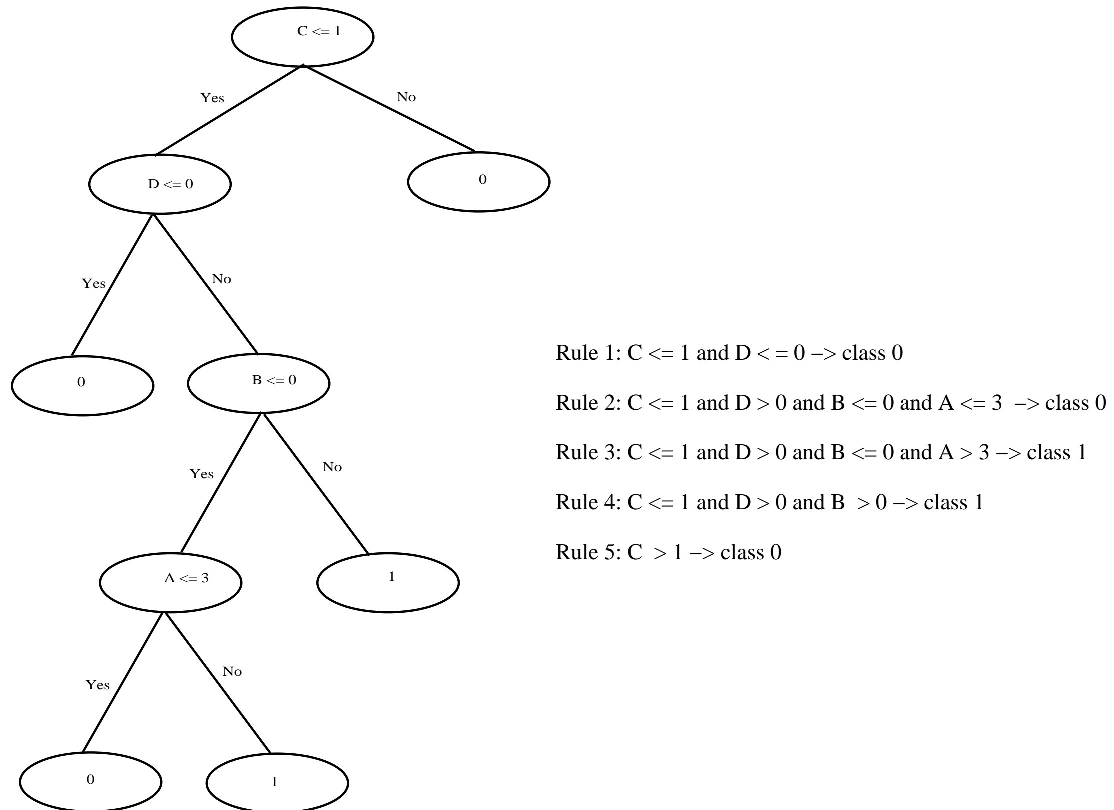

FIGURE 2.2. A decision tree with its branches re-iterated as rules (left-to-right order).

In order for C4.5 to perform a classification task, the latter should satisfy certain requirements that we list below in brief:

- The data must be expressed as a vector of attribute values.
- The labels should be pre-defined and sharply delineated.
- The cases should outnumber the labels by far.

The models generated are logical classification models.

7. Previous Work in Building Logical Software Quality Estimation Models

Selby and Porter have used machine learning to build decision trees as early as 1988 [Porter and Selby, 1988][4]. They performed an experiment involving 16 NASA software systems (with size ranging from 3000 to 112000 source lines of Fortran and a total of more than 4700 modules) on which they measured 74 metrics. These capture development effort, faults, design style and implementation style. Decision trees were constructed using ID3 to predict two kinds of classes: high versus low development effort and high versus low faults. The average accuracy was 79.3% over the 9600 decision trees constructed[5]. Several parameter combinations are tested and the best accuracy obtained is 88.4%. Later on, in [Porter and Selby, 1990], the authors motivate the use of classification techniques in the software development process in general, and especially the use of decision trees. First, classifiers can be easily calibrated to new projects and environments using historical data and are applicable to large-scale systems, as opposed to being limited to small-scale applications. Second, they provide a white-box model from which knowledge can be extracted and fed into the development process.

In [Mao *et al.*, 1998], the authors proposed the use of C4.5 to build software quality estimation models that verify three hypotheses about the impact of inheritance, coupling and complexity, separately, on the reusability of a class in an object-oriented software system. The accuracy of the induced models ranged from 73.8% to 89.3%. Similar to our work, [Mao *et al.*, 1998] proposed the use of estimation models as guidelines for future software development. In the context of estimating the reusability of software components, [Basili *et al.*, 1997] used C4.5 to build models that estimate the cost of rework in a library of reusable software components. Unlike the previous work that has been done for the same purpose, [Basili *et al.*, 1997] concentrated on software

[4]As we have seen earlier, decision trees can easily be transformed into rule-based estimation models.
[5]Nine thousand six hundred decision trees are constructed to test different tree generation parameter combinations

components that have been developed for the purpose of re-use. [Jorgensen, 1995] and [Briand *et al.*, 1993] had attempted the same problem but the former focussed on the comparison of different techniques as mentioned previously, and the latter analyzed components that were developed to satisfy specific application requirements. The work aims at identifying which of the faulty-components are more or less expensive to isolate and re-work (i.e. it is presumed that the faulty-components are already identified). In [De Almeida *et al.*, 1999], C4.5 rule sets were used to predict the average isolation effort and the average effort. An accuracy of 66% was achieved when predicting the isolation effort variable value and 68% when predicting the average effort variable value.

In [Miceli *et al.*, 1999] and [Sahraoui *et al.*, 2000b], software quality estimation models are used to suggest transformations of already existing software. Their technique involves the selection of a set of transformations and a set of metrics. Then, the impact of the transformations on the values of these metrics is studied and a set of new transformation rules is derived. A software quality estimation model is used to propose transformations (different values for metrics) that improve the quality. The transformation is implemented by turning the left-hand side of a rule that suggests a bad quality characteristic to false. For example, a rule like "If the number of children of a class in an object-oriented system is greater than 6 then, the class is fault-prone", suggests to change the number of children of the particular class to a number less than or equal to 6, provided that there is no contradicting rule in the estimation model. However, such a rule does not imply that whenever the number of children of a class in an object-oriented system is less than or equal to 6, the class is not fault-prone. This point is judiciously addressed in [Sahraoui *et al.*, 2000b]. The authors leave to the designer/programmer the option of validating any prescribed transformation.

[Briand *et al.*, 1999] empirically investigate the relationship between most of the existing coupling and cohesion measures, defined at the class level in object-oriented systems, on one hand and fault-proneness on the other. While cohesion did not appear

to be a very good indicator of fault-proneness, coupling proved to be a strong one. This might be due to the inadequate definition of the measures of cohesion.

In [Piattini *et al.*, 2002], two experiments were conducted to validate the usefulness of metrics in predicting the maintainability of existing relational databases. The results were produced using C4.5 and RoC, a bayesian classifier [Ramoni and Sebastiani, 1999]. The experiments were conducted separately (one in CRIM[6] in Canada and one at the University of Catilla-La Mancha in Spain). We believe that conducting the experiments in two geographically different locations is an interesting approach since human experts were involved in the assessment of the maintainability of the databases (in both countries). The accuracy obtained with C4.5 was 94% in the Canadian experiment and 92% in the Spanish experiment. RoC achieved an accuracy of 73.8% in the Canadian experiment and 81.4% in the Spanish experiment. In both experiments, the same subset of two metrics (table size and depth of referential tree) proved to be quite accurate, which proves that these can be good indicators of maintainability of a relational database.

In [Ikonomovski, 1998], C4.5 was used to construct rule sets to predict the fault-proneness of a class (in an object oriented software system) based on inheritance metrics. Two types of models were built. In one, the classification label can take one of two values (fault-prone or not). In the other, it can take one of three values (depending on how many faults the class contains). The generated models show an average prediction accuracy of 83.2% and 75.5% on the training set, for the 2-value and the 3-value data sets, respectively.

8. Previous Work in Optimizing Existing Software Quality Estimation Models

To the best of our knowledge, the only work (other than ours) that tackles the problem of optimizing already existing rule-based software quality estimation models is an independent project that is currently taking place at *Université de Montréal*

[6]Centre de Recherche Informatique de Montréal.

in Quebec, Canada by Salah Bouktif, Houari Sahraoui and Balazs Kégl [Bouktif and Sahraoui, 2002], [Azar *et al.*, 2002], [Bouktif *et al.*, 2004]. While our goal is to explore the use of genetic algorithms on this optimization problem, [Bouktif and Sahraoui, 2002] consider other techniques (such as tabu search and simulated annealing) to optimize the models. Their work differs also from ours in that it does not consider the adaptation of a single rule set to a new data set.

CHAPTER 3

Genetic Algorithms

Our purpose is to optimize and adapt software quality estimation models that take the form of rule sets. The search space for this problem is very large. This makes it impossible to use local search or exhaustive search methods. Genetic algorithms (GAs), on the other hand, have proved to perform well in such large multi-modal spaces, so they will be our choice for approaching the problem.

In Section 1, we introduce the *Darwinian theory of natural selection* [Darwin, 1859] which served as inspiration for GAs. In Section 2, we give some biological background necessary for understanding some of the concepts implemented in GAs. In Section 3, we introduce the details of a GA. In Section 4, we mention the pros and cons of GAs and we refer the interested reader to some works that can be used as a theoretical background in GAs. In Section 5, we list the different areas where GAs have been applied (focussing on the work that has been done using GAs in optimizing classification models in Section 5.2).

This chapter is a brief overview of what genetic algorithms are. For a more thorough introduction, the interested reader is referred to [Holland, 1975], [Goldberg, 1989] and [Mitchell, 1999]. For a more theoretical review, the reader is also referred to [Whitley, 1994].

1. The Darwinian Theory Behind GAs

Genetic Algorithms were introduced in the 70's by John Holland as a general model of adaptation [Holland, 1975], inspired by the *Darwinian theory of natural selection and survival of the fittest* [Darwin, 1859]. According to Darwin, a population of individuals (organisms) exists in an environment. Some individuals have certain traits that make them 'fitter' for the particular environment. These individuals have a higher chance of surviving and passing their 'good' traits on to their progeny as Darwin states in his book *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle For Life*:

`...can we doubt (remembering that many more individuals are born than can possibly survive) that individuals having any advantage, however slight, over others, would have the best chance of surviving and of procreating their kind?...This preservation of favourable variations and the rejection of injurious variations, I call Natural Selection.`
[Darwin, 1909].

Genetic algorithms (GAs) can be used as a simplified, abstracted implementation of this natural process. They have gained great popularity as an optimization techniques. In a GA, a population of **chromosomes** or **individuals** is created. Each chromosome represents a solution to the problem at hand. The 'better' the underlying solution, the fitter the chromosome, and the higher its chance to survive. Chromosomes are selected from the current population based on a fitness value, and new ones are created through the application of some 'operators'. Since the principal of natural selection is applied, it is hoped that through the *recombination* process, good 'traits' dispersed in the population will be combined in one individual that will display a higher fitness.

Before giving a detailed description of the operators that make the creation of new individuals possible, we give, in Section 2, the definitions of some of the terms

that GAs have borrowed from biology and what they mean in the context of computer science.

2. Definitions of Some Biological Terms

In biology, the chromosomes present in every cell of every living organism are formed of deoxyribo nucleic acid (DNA). A chromosome can be viewed, in a simplified way, as a string of **genes** where each gene encodes a trait (for example, the eye color in a human being). In [Falkenauer, 1998], the genes are likened to words formed of the four letters T, C, G and A and the chromosomes to phrases formed of those words. In nature, the information contained in the genetic code is called the **genotype**. The way the genotype manifests itself physically represents the **phenotype** (for example, a flower with yellow petals). This brings us to one of the key steps in designing a genetic algorithm, namely, representing chromosomes. We refer to this as the **encoding scheme**. The chromosome as represented in the GA is called the genotype. The underlying element that it actually represents (in an optimization problem, the solution that the chromosome represents, for example) is called the phenotype. Hence, the encoding scheme can be viewed as a mapping between the phenotype space and the genotype space. In a GA, the set of individuals or chromosomes at a specific time is called a **population**. The process of **evolution** entails several iterations during which new chromosomes are created from existing ones. Each iteration is called a **generation**. Each chromosome displays a certain 'goodness' that measures how well it performs in its environment (i.e. how good a solution it is to the given problem). This degree of goodness is referred to as the **fitness** of the chromosome.

To illustrate these terms, let us consider the famous example from [Goldberg, 1989] of maximizing the values of x^2 for $x \in [0, 31]$. Using a simple 5-bit binary encoding, two randomly generated genotypes are: 11000 and 10011. The phenotypes of these chromosomes are, respectively, the numbers 24 and 19. In the context of this problem, higher values of x^2 imply better or fitter chromosomes. Hence, the fitness of each individual, x is defined as $f(x) = x^2$. The first chromosome (11000) has a

higher fitness than the second one (10011). A population of such individuals can be randomly generated to initialize the GA. Afterwards, new individuals are created by the application of genetic operators, the main ones being *crossover* and *mutation*. In Section 3, we describe these operators in more detail. At this point, we give a brief sketch of what they mean in nature.

Crossover consists of swapping genetic code between two chromosomes. Roughly speaking, in biology, two chromosomes meet, swap parts of their genetic code and drift apart. This results in the creation of two new chromosomes. **Mutation** occurs when genetic code gets perturbed. In nature, duplicating DNA can sometimes result in errors as the genetic information is copied from the parent chromosomes to the offspring. DNA is also prone to damage in day-to-day existence [Falkenauer, 1998]. Both operators are simulated in the core of a genetic algorithm in order to create new individuals from already existing ones as we will see in the next section.

3. The Genetic Algorithm

The method that constitutes the core of the GA consists of the following:

```
1. An initial population of individuals is created. This is usually, but
   not always, done at random. The fitness of each individual is computed
   and a selection probability  is given to each individual. This
   probability is influenced by the fitness of the chromosome.

2. Individuals are selected according to the selection probability that
   they have been assigned. They undergo crossover which consists of
   swapping some of the genetic code (parts of the string that constitutes
   the chromsomes) and produce two offspring. The offspring are copied to
   the new generation.

3. Some of the better  individuals in the current generation can be
```

copied (as they are) to the new generation.

4. A small part of the new generation might be mutated.

5. The process continues until the new generation is complete.

6. Several such generations are created one after the other until a certain stopping criteria is met (a desired fitness has been found or a preset number of generations has been reached).

3.1. Encoding Scheme. The encoding scheme is one of the most crucial steps as it affects heavily how easy it is for the GA to solve the problem of finding a good individual. Encoding also affects some of the genetic operators (in particular, mutation). The example that we showed in the previous section uses a binary representation of the chromosomes. When they were first introduced in [Holland, 1975], GAs used this kind of representation. One advantage of binary encoding is the simple *crossover* and *mutation* operators that come along with it (we will describe this in more detail when we describe the two operators). However, binary encoding is not straightforward to implement in all applications. Often, it is much easier and natural to use integer or decimal numbers. This is known as the **integer/real-valued representation**. Consider, for example, the problem of finding the optimal set of weights for a neural network. One way to solve this problem with GAs is to create a set of possible solutions and map each solution to a string of real values. Each string is a chromosome and each gene in the chromosome encodes one weight in the neural net. Genetic operators are then applied on the strings of numbers to create new ones.

Other representations were created as applications required. One such representation is the **permutation representation** used for the traveling salesman problem (TSP) [Michalewicz, 1996], [Haupt and Haupt, 1998], [Goldberg and Lingle, 1985], [Grefenstette *et al.*, 1985], [Oliver *et al.*, 1987], [Jog *et al.*, 1989], [Whitley *et al.*, 1989],

[Starkweather *et al.*, 1991], [Whitley *et al.*, 1991], [Hamaifar *et al.*, 1993], [Schmitt and Amini, 1998], [Jog *et al.*, 1991] and the job-scheduling problem [Eiben and Smith, 2003]. In the permutation representation, the order of the genes is important. For more detail on this type of representation, the reader is referred to [Mitchell, 1999].

Unfortunately, it is impossible to design one representation that suits all applications. Coming up with a good representation requires a good knowledge of the domain of application [Eiben and Smith, 2003]. Ideally, one would aim for an encoding that allows representing all valid solutions to the desired problem. Often GAs are designed to work in a space that includes all valid solutions but is larger. This is done in order to allow the GA to explore a large search space. Working only with valid solutions is not always possible, and it is often the case that the GA allows the emergence of invalid individuals but penalizes them heavily through the fitness function [Eiben and Smith, 2003].

Another question that is raised when considering the encoding scheme is the length of the chromosomes. Traditionally, GAs dealt with fixed-length chromosomes, where all chromosomes are formed of the same number of genes. This is a big limitation in many practical problems, and it was lifted in some recent approaches [Han *et al.*, 2002]. Here, also, the choice between fixed versus variable length chromosomes is problem-dependent.

3.2. Genetic Operators. In this section, we review the two genetic operators that the GA uses to create new individuals: *crossover* and *mutation*.

3.2.1. *Crossover.* **Crossover**, also referred to as **recombination**, is a major genetic operator (we will use these terms interchangeably).

In biology, *crossover* happens when two chromosomes exchange some of their genes and generate offspring. Each of the resulting offspring inherits traits (pieces of information) from both parents. We can see this in a human baby taking the eye color from the father and the hair color from the mother. The idea behind this operator in GAs is to combine in one individual "good" traits dispersed in the whole population, hence, creating "better" individuals. When chromosomes meet, the exchange of the

genetic material happens with a certain probability. It is possible that chromosomes do not exchange any genetic code and hence the offspring are exact copies of their parents. In GA terms, this is referred to as **asexual reproduction**[1]. Next, we describe the most popular versions of this operator, *single point crossover* and *double point crossover* which we will use in chapters 4 and 5.

Single Point Crossover or **1-Point Crossover**. This is the original version of the operator that Holland introduced in his genetic algorithm [Holland, 1975]. Under this type of crossover, each chromosome in a pair is cut at one location and new chromosomes are formed. The first offspring receives the first part of the first parent along with the second part of the second parent whereas the other offspring receives the first part of the second parent and the second part of the first parent. Figure 3.1 illustrates this operator on a binary representation of chromosomes.

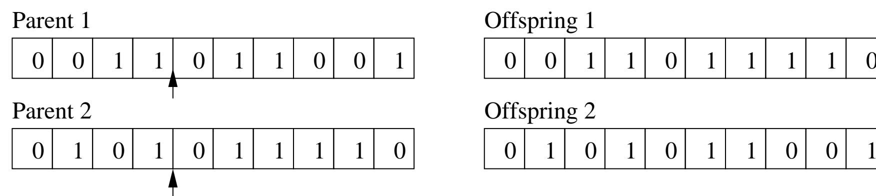

FIGURE 3.1. 1-Point Crossover: parent chromosomes get cut at one location and offspring interchange the tails.

N**-Point Crossover**. Under N**-point crossover**, N random cut points are generated within the parent chromosomes and the offspring inherit from both parents alternating segments defined by the cut points. Figure 3.2 illustrates this operator. **2-point crossover** or **double-point crossover** that we used in our algorithm described in Chapter 4, Section 3 is a special case of *N-point crossover*.

Single-point and N*-point crossover* are only two of the variants of this operator available in the literature. The main appeal of *single-point crossover* is that it is easy

[1]Some work has been done in which more than two parents are allowed to breed. The interested reader is referred to [Eiben and Smith, 2003], [Eiben *et al.*, 1995].

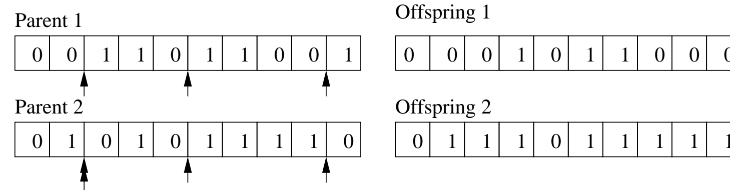

FIGURE 3.2. N-Point Crossover: parent chromosomes get cut at N different locations and offspring inherit alternating segments. N=3 in this figure.

to implement. *N-point crossover* allows bigger changes in the offspring as more parts are exchanged between chromosomes.

Uniform Crossover. Unlike *single-point crossover* and *N-point crossover*, where genes are inherited as contiguous segments, *uniform crossover* treats each gene independently and which offspring inherits which genes is decided randomly. For this, a sequence of l random numbers is generated (l being the size of the chromosome), and for each position, if the respective random value is below a certain threshold, the gene is inherited from parent 1; otherwise, it is inherited from parent 2. The other offspring inherits the remaining genes. Figure 3.3 illustrates **uniform crossover**.

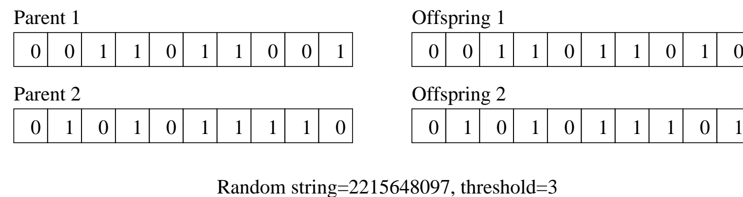

FIGURE 3.3. Uniform Crossover: offspring inherit genes from one parent versus another depending on the sequence of random numbers and the threshold value.

Uniform crossover found its popularity after the work of Syswerda [Syswerda, 1989]. For a more thorough study on this type of crossover, the reader can refer to [Ackley, 1987]. For other types of crossover (arithmetic recombination, partially-mapped crossover, etc.) the reader is referred to [Eiben and Smith, 2003].

3.2.2. *Mutation.* As mentioned earlier, in nature, duplicating DNA can sometimes result in errors while the genetic information is copied from the parents to the offspring. **Mutation** is the operator that simulates this effect. Under this operator,

a randomly chosen gene within a chromosome gets perturbed to a random value from the domain of values for the gene. The motivation for simulating mutation in a GA is to stop the algorithm from being stuck at local optima (as explained later). However, the probability of *mutation* should not be set very high; otherwise, the algorithm will turn into a random search which will slow down the search process. *Mutation* is not a major operator in the sense that the rate at which it happens is much lower than the rate at which *crossover* occurs. In fact, in most experiments, the rate of *mutation* is set to a value between 0.001 and 0.1 (compared to, usually, 50% and above for crossover). In general, genes are mutated with a certain probability independently of each other. It is very common not to apply a uniform mutation probability across a chromosome. Figure 3.4 illustrates the mutation operator in the case of a binary encoding where the third and the last genes are mutated.

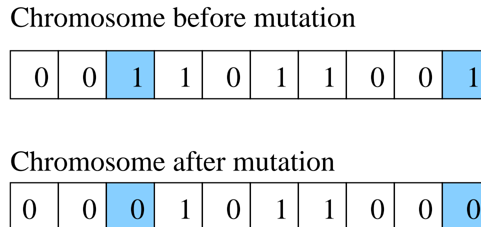

FIGURE 3.4. Mutation: The third and last genes of the chromosome are mutated.

Many researchers do not attribute to *mutation* the same importance that they attribute to *crossover* as they view it as simply a means of avoiding local optima whereas *crossover* is what makes 'good' individuals appear in the population by combining genes from fit chromosomes. A debate is still ongoing in the GA community regarding the importance of *mutation* (in [Rowe and East, 1993], a GA is designed in which *mutation* is not needed). We do believe that *mutation* is a very important operator as it stops the algorithm from being stuck in local optima and it also helps re-introduce information that might be lost during the search process ([Ghanea-Hercock, 2003] and [Holland, 1975]). For example, consider the case of a binary search space. Assume that at one point of the evolution process, all chromosomes have the value 1 in their first gene location. If the optimal solution has a value 0 in the first gene

location, without *mutation*, the GA will never find the optimal solution as it will skip exploring half of the search space.

Unlike *crossover*, there is a great dependence between *mutation* and the encoding scheme. For example, mutating a bit-string consists of just flipping a bit from 1 to 0. As is the case with *crossover*, several *mutation* operators can be found in the literature. We list a few only in order to show the intricate dependence between this operator and the encoding scheme.

Mutation for Real-Valued Representations. This version of the operator is analogous to bit-flipping but operates on chromosomes with floating-point representations. Under this operator, a gene value g_i is replaced with another value g_i' chosen randomly from the domain of g_i and with a uniform probability distribution over the domain (see Figure 3.5).

Chromosome before mutation

1.5	2.5	1.5	3.5	2.5	2.25	1.75	1.5	2.5	1.75

Chromosome after mutation

1.5	1.0	1.5	3.5	2.5	1.25	1.75	1.5	2.5	1.75

FIGURE 3.5. Floating-Point Mutation: the fifth gene is mutated to a random value from the domain of the gene.

Creep Mutation for Integer Representations. To mutate a gene, a small value is added to, or subtracted from its value.

Swap Mutation for Permutation Representations. Under this operator, two positions are randomly picked in a chromosome and their values are swapped. In this case, the mutation of one gene is no longer independent of others.

Inversion Mutation. Similar to *swap mutation for permutation representations*, this mutation links genes to others in the string. This operator generates two random positions within a chromosome and inverts the substring between them (see Figure 3.6).

Chromosome before mutation

0	2	4	1	6	8	2	2	0	4

Chromosome after mutation

0	1	4	2	6	8	2	2	0	4

FIGURE 3.6. Inversion Mutation: The genes between the second and the fourth ones are inverted

Many other *mutation* operators can be found in the literature. We refer the reader to [Mitchell, 1999] and [Eiben and Smith, 2003] for a more detailed discussion.

3.3. Selection Techniques. The core of the Darwinian theory of evolution lies in the idea of fitter individuals getting a higher chance of surviving and producing progeny. Inspired by this theory, in a GA, selecting individuals for mating should be based on their fitness.

Roulette Wheel Selection. When GAs were first introduced, this was the most commonly used selection technique. One can imagine a roulette wheel and each chromosome is given a part of the wheel that is proportionate to its fitness. A marble is thrown and the chromosome corresponding to the piece of the wheel where the marble stops is selected. Roulette wheel selection is a fitness-proportionate selection, i.e., the selection of an individual depends on its absolute fitness value and on the absolute fitness values of other individuals in the population. We can see this as attributing to an individual i a selection probability $s(i)$ proportionate to its fitness $f(i)$ in a population of n individuals (Equation 3.1).

$$s(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)}. \tag{3.1}$$

One drawback of this technique is that when a population exhibits a high variance of fitness, the technique is strongly biased toward individuals that are usually fit compared to the rest of the population. For example, consider a population of 4 individuals having a fitness of 5, 50, 10 and 75. Applying roulette wheel selection will result in the situation shown in Figure 3.7.

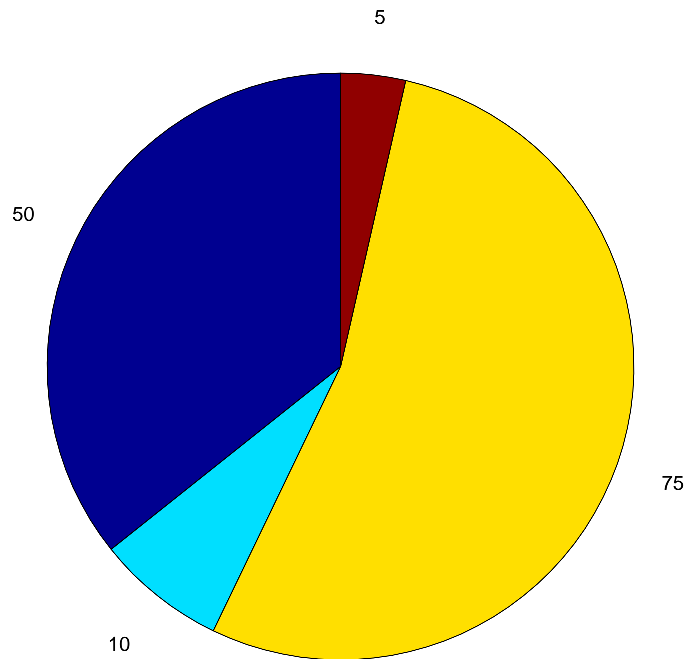

FIGURE 3.7. Roulette Wheel Selection. The pie is split among four chromosomes of fitness: 5, 50, 10 and 75.

It is easy to see that in a similar situation, the chromosomes with fitness 5 and 10 have very little chance of being selected and the fittest chromosomes are almost always the one selected. In real applications, this often translates to a convergence of the GA to a population containing multiple copies of the same (usually suboptimal) individuals, an anomaly commonly referred to as **premature convergence**. An interesting comparison of different selection techniques based on a mathematical description using the fitness distribution can be found in [Blickle and Thiele, 1995].

Rank Selection. One way to overcome the anomaly presented by the roulette wheel selection technique is to assign to individuals ranks and then to select by rank instead

of absolute fitness ([Baker, 1985] and [Grefenstette and Baker, 1989]). Hence, in a population of 4 individuals, the fittest one will be given rank 4, the next one 3 and the least fit one will be given rank 1. Figure 3.8 shows the same four individuals shown in figure 3.7 but the wheel is apportioned relatively to the ranks of the individuals. It is easy to see that they all get a fair chance of being selected while giving priority to the fitter chromosomes. The drawback to rank selection is slowing down the convergence of the GA, because all individuals now get a chance to reproduce. For a more thorough discussion of rank selection, the reader is referred to [Ghanea-Hercock, 2003] and [Eiben and Smith, 2003].

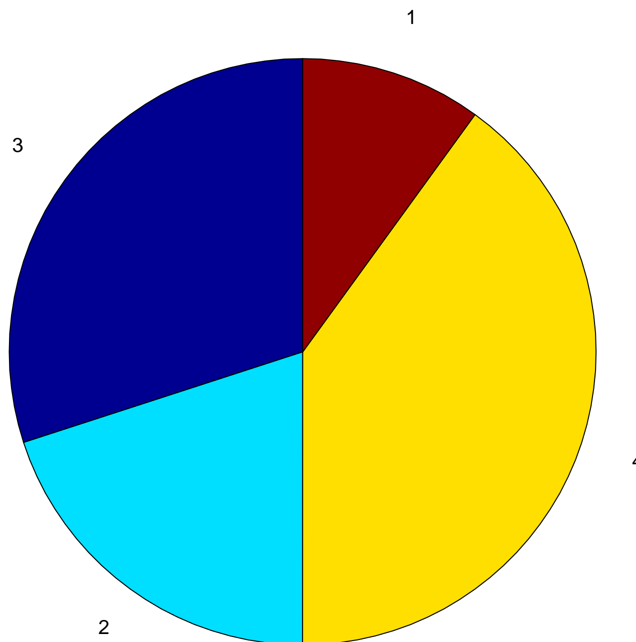

FIGURE 3.8. Rank Selection. The pie is split among chromosomes of fitness: 5, 50, 10 and 75. Chromosomes are ranked (ranks from 1 to 4) and portions of the pie are alloted to them by rank.

Tournament Selection. Roulette wheel selection and rank selection both assume *à priori* knowledge of the fitness of all the individuals in the populations. As we can see from Equation 3.1, the selection probability $s(i)$ of an individual i is computed based on the fitness of all other chromosomes from the same population. Also, in

rank selection, ranks are attributed taking into consideration the fitness of all chromosomes. Tournament selection does not require such a knowledge. Tournament selection consists of selecting k random individuals from the current population and the fittest of them is chosen to be one of the parents in the pair to undergo crossover. The chromosomes are then returned to the population. The process is repeated to choose the next parent. This is particularly useful in situations where the fitness of all the individuals in a population is not available. Consider, for example, a game strategy evaluation. It is computationally inefficient to measure the value of all strategies when this incurs simulating a game with a particular strategy. By setting the tournament size to 2, it is possible to select between two strategies at a time [Eiben and Smith, 2003].

Elitism. Elitism was first introduced by [De Jong, 1975]. It consists of copying the best chromosome(s) to the new population. This ensures that the best found so far is (are) never lost. In many applications, elitism has been found to improve the performance of the GA. However, choosing a large number of chromosomes to be copied to the next generation might result in the premature convergence of the GA to a population formed of copies of the same individual, which usually represents a suboptimal solution.

3.4. Replacement Policies. Traditionally, GAs maintain a population of fixed size throughout the process of evolution. One question that comes to mind is which individuals to replace when creating a new population: replace all individuals and create a fresh new population, or replace only a portion of the older population and allow chromosomes to breed with individuals from the previous generation. These two different tracks have given way to what is known as the **generational GA** and the **steady-state GA**. Under the first replacement policy, the new generation of chromosomes replaces all of the previous generation. Of course, some offspring might be copies of their parents, and also the generations may overlap due to elitism. This is the traditional policy introduced in [Holland, 1975]. A steady-state GA, on the

other hand, consists of replacing only a portion of the population (typically two chromosomes) with the offspring. In this case, the offspring may compete with their parents (this is more like nature where, luckily, the creation of one generation does not efface the previous one). Steady-state GAs have gained popularity in the 80's. This replacement policy has the advantage of allowing an individual to compete as soon as it comes to existence [Beasley *et al.*, 1993a]. For a more thorough study on this topic, the reader can consult [Syswerda, 1989], [G. Rawlins, 1991], [Holland, 1986], [De Jong and Sarma, 1993], [Beasley *et al.*, 1993a] and [Whitley, 1989]. In our problem, we chose to use the generational GA since the search space is big and only a small number of chromosomes are available in each population. Applying the steady-state GA in our case will result in a slow convergence of the GA.

4. The Pros and Cons of GAs

In this section, we discuss the major arguments in favor and against GAs.

Pros. GAs have proven to perform well in many parameter optimization tasks. Holland showed that GAs are good at performing, at the same time, exploration of the search space and exploitation of the knowledge acquired [Holland, 1975]. Many search methods are good at one task or the other. For example, hill climbing is good at exploiting knowledge and weak in exploring the search space whereas random search is good at exploring the search space while exploiting very little, if at all, the knowledge gained throughout the search. GAs combine the two qualities [Beasley *et al.*, 1993a], as the search is always guided by the fitness of the chromosomes (the goodness of the solutions to optimize); as Goldberg puts it beautifully "GAs are random but not directionless" [Goldberg, 1989]. Another argument in favor of genetic algorithms is that no matter which different operators or techniques one might choose at each step of the evolution process, the algorithm is always relatively simple and easy to understand [Whitley, 1994]. Many have praised the implicit parallelism that GAs exhibit. As a matter of fact, many solutions to a problem exist in the same

population and are explored at the same time (in one generation). Tabu search and simulated annealing, on the other hand, work with a single solution. Finally, GAs are biologically inspired and hence appealing for people who are fascinated by mimicking nature or using it as an inspiration.

Cons. The main argument against GAs is that they are computationally expensive, and they do not guarantee an optimal solution. As a matter of fact, in most cases most of the computation time is spent evaluating the fitness function. However, it is quite common to parallelize the evaluation process. Although they do not guarantee an optimal solution, with the appropriate choice of the selection technique, GAs guarantee a non-deterioration of results (by applying elitism, which consists of copying the fittest individual(s) to the next generation).

It is largely agreed upon that if an algorithm is known to work well for a certain problem then a genetic algorithm would not be the proper choice [Whitley, 1994]. However, GAs are worth exploring in areas where other techniques have failed (for example, when exhaustive search is impossible, as is the case in our problem) or in hybrid approaches where they can be combined with another technique. Another concern is the fact that it is often difficult to come up with an encoding of the problem at hand.

No study about genetic algorithms would be complete without an insight into the fundamental theoretical issues that arise in GAs, such as the schema theorem, the building block hypothesis, the effect of epistasis (when the behavior of one gene is affected by others), niching and speciation, deception, etc. Since these are not central to the topic of our thesis, we refer the interested reader to [Whitley, 1994], [Whitley, 1991], [Goldberg, 1989], [Beasley *et al.*, 1993a], [Beasley *et al.*, 1993b] for a comprehensive introduction to these topics.

5. Areas of Application

In this section, we list several areas of application for GAs. We start by giving a general view of the different areas and then we focus on the previous work that, similar to ours, has used GAs in optimizing rule-based classification models.

5.1. Different Areas of Application. GAs are mostly suitable in problems where we can settle for a good solution rather than an optimal one [Goldberg, 1989]. They have been successfully applied to problems such as: finding control policies for manufacturing systems [Porter, 1998], electronic circuit design [Miller *et al.*, 1998], sparse table compression [Driesen, 1994], obstacle avoidance in mobile robotics [Michalewicz, 1996], [Floreano, 1998] and [Potter *et al.*, 1995], mechanical cam shape optimization [Alender and Lampinen, 1998], neural network synthesis [Back, 1998], protein structure prediction [Day *et al.*, 2003], and other classical optimization problems such as activity scheduling, the 0/1 knapsack problem, the personnel scheduling problem [Han *et al.*, 2002], route scheduling problems [Logan and Riccardo, 1996], the traveling salesman problem [Michalewicz, 1996], [Haupt and Haupt, 1998], [Goldberg and Lingle, 1985], [Grefenstette *et al.*, 1985], [Oliver *et al.*, 1987], [Jog *et al.*, 1989], [Whitley *et al.*, 1989], [Starkweather *et al.*, 1991], [Whitley *et al.*, 1991], [Hamaifar *et al.*, 1993], [Schmitt and Amini, 1998], [Jog *et al.*, 1991] and other NP-Complete problems [Falkenauer, 1998]. For an extended but older bibliography on applications of GAs, the reader can consult [Alander, 1995].

GAs have also been used for optimizing rule-based classification models. We give an extensive review of the literature in this domain.

5.2. Previous Work in Using GAs to Optimize Rule-Based Classification Models. GABL (Genetic Batch concept Learner) ([Spears and DeJong, 1993], [Spears and F., 1991], [De Jong and Spears, 1991] and [De Jong *et al.*, 1993]) is one of the most popular systems which used GAs to optimize classification models. The GA at the heart of GABL evolves concepts represented as a disjunction of rules.

The left-hand side of each rule is a conjunction and the right-hand side is a classification label. The rules are presented to the GA as fixed-length chromosomes with a binary representation. Each substring encodes an attribute test and is of length n where n is the number of possible values that the corresponding attribute can have. The work compares GABL to ID5R, a decision tree construction algorithm. Because ID5R learns incrementally as new data becomes available, an incremental version of GABL, called GABIL, was also analyzed. GABIL continually learns and refines the concept classification rules as new data is obtained. The system proved to be competitive with ID5R on complex target concepts. As a matter of fact, the performance of ID5R seemed to suffer as the number of conjuncts and disjuncts increased. The system favored concepts which could be represented with small decision trees. GABIL did not show such bias. In [Spears and F., 1991] GABIL was compared to C4.5 and the rule-induction system NEWGEM [Mozetic, 1985] on two different domains. The authors also included in GABIL the mechanisms in NEWGEM that seemed to be responsible for its superior performance on certain classes of target concepts (for example, dropping conditions). In general, complementing GABIL with one or more such strategies improved significantly the performance of the system. Compared to C4.5 and NEWGEM on two different domains, the modified GABIL performed much better than C4.5 and was competitive with NEWGEM on one of the domains. On the other, its performance was close to that of C4.5 and significantly better than that of NEWGEM (significance at the 90% level).

Corcoran and Sen [Corcoran and Sen, 1994] also used genetic algorithms to optimize rule-based models. In this work, classification rules involve real-valued attributes and classification labels can be either integer or real. Similar to GABIL, the chromosomes have a fixed-length, but each attribute is represented by two values (*min* and *max*) which indicate the valid range of values for this attribute. The GA was tested on the wine classification benchmark problem available at UCI repository of Machine

Learning datasets[2]. Results show that the GA is able to find effective rule sets to accurately classify almost all instances.

The same problem is tackled in [Llorà and Garrell, 1999] where GABL is adapted to real-valued attributes. They implemented GENIFER (GENetic classIFiER). In its original version, GENIFER-BRE (Binary Rule Encoding) the system suffered from two drawbacks: the large variance between results on training sets and results on testing sets and the language being not powerful enough to handle problems with real-valued attributes. Different variants of GENIFER introduced a new concept, namely the expression of the condition part of a rule based on a nearest-neighbour approach. The space is divided into regions each represented with one significant point (associated with one classification label). When a new example is to be classified, it is attributed the classification label of the significant point that is nearest to it. Different ways of computing the distance between two points were considered (Hamming distance, Euclidian distance and cubic distance). Also, other variants of the system took into account the assumption that not all attributes are useful or should be included when computing the nearest neighbour. Compared to neural networks and case-based reasoning, GENIFER performed better than the former and slightly better than the latter.

In [Garrell *et al.*, 1999], Ge-CS (Genetic based Classifier System), another system based on GABL, is presented and compared to CaB-CS (Case Based Classifier System) on data for classifying mammary biopsy images as cancerous or non-cancerous. In Ge-CS, a chromosome is a binary string that encodes a rule. Both systems outperform neural networks. The results obtained with CaB-CS were globally better but the rule sets obtained by Ge-CS were easy to interpret by human experts.

Mianei and Punch [Minaei and Punch, 2003] show that combining classification models gives better results than a single model when classifying students in order to predict their final grade based on other features. A GA is used to search for the best weights for feature vectors. Results show that a combination of multiple classification

[2]http://www.ics.uci.edu/ mlearn/MLRepository.html

models, by itself, improved accuracy significantly. Weighing the attributes and using the GA improves results by at least 10%.

Sen and Knight use a GA to learn appropriate prototypes for classes [Sen and Knight, 1995]. A prototype is defined as a "collection of salient features of a concept". An instance is labelled by the class of the prototype that shares the most features with it. PLEASE is a GA that evolves structures (a structure consists of one or more prototypes). The work compares PLEASE to C4.5 on a set of problems comprising 2 attributes each having a range of [0,1]. Hence, the input space is a square of unit area. Four different problems were formulated by dividing the unit square in different ways. Each division is assigned a label. PLEASE outperforms C4.5 on the testing sets and is consistently able to find a near perfect set of prototypes. Training and testing errors are more consistent in PLEASE than in C4.5. However, C4.5 outperforms PLEASE on the training set (but not significantly). As to the quality of the solutions, PLEASE find solutions with fewer prototypes than those found in C4.5, although, in most cases, the number of prototypes is bigger than the minimal number required.

Less similar to our work but still related is the work of Llorà and Garrell [Llorà and Garrell, 2001b], where an evolutionary algorithm (EA) is used to reduce storage requirements in instance-based learning algorithms. Different evolutionary algorithm techniques other than GAs have been used in similar problems. For example, GALE (Genetic and Artificial Life Environment) is an EA that induces a set of partially-defined instances. A partially-defined instance is an input vector with at least one known value of several input attributes and an output class. In [Llorà and Garrell, 2001b], experiments were performed on 10 different datasets (2 artificial, 2 private and 6 taken from the UCI repository). GALE was compared to four other classification models (IB1, IB2, IB3 and IB4). Results show that GALE slightly improves accuracy and significantly reduces the required storage space. In [Llorà and Garrell, 2001a], experiments were perfomed on 8 datasets chosen from the UCI repository and 2 private datasets. GALE is compared to C4.5 revision 8 (C4.5r8). The orthogonal decision trees evolved by GALE outperform significantly the ones induced by C4.5r8.

The results of GALE on oblique and multivariate decision trees are comparable to those of C4.5r8.

In [Punch *et al.*, 1993], [Pei *et al.*, 1994] and [Pei *et al.*, 1995], GAs were also used to optimize classification systems. In this case, the optimization criterion was based on the number of features included in the classification rules as well as the accuracy of the classifier. As a matter of fact, the GA was used to select features according to how discriminatory they are, keeping into consideration that the accuracy had also to be optimized. The work is based on a previous work by Sieldlecki and Sklansky [Siedlecki and Sklansky, 1989] which uses a GA to select features.

One important difference between our work and the GAs described above is the fact that our GA allows the representation of variable-length chromosomes (as opposed to GABL and GENIFER, for example). As a result, the number and type of attributes in the data set does not affect the speed of the GA. In contrast, in GABL, for example, allowing continuous type attributes increases significantly the size of the chromosomes and hence affects the speed of the search process because the chromosome contains a gene for every possible value of the attribute. Another important difference is the fact that our GA can be used with any type of rules, not only attribute-value based rules although in the results reported in this thesis, we focus on attribute-value based rules.

CHAPTER 4

First Approach to Optimization: Deriving a Better Model from a Set of Models

A major problem in the area of software quality is the scarcity of data from which representative samples can be drawn. Due to this fact, it becomes hard to use machine learning techniques to build software quality estimation models that can be used to classify new/unseen data. As a matter of fact, many machine learning techniques need a lot of data in order to construct reliable classification models; otherwise, these suffer from overfitting. Our approach to the problem consists of combining existing models and adapting them to new, specific domain data. This helps incorporate the expertise of older models into the new models.

This chapter presents a genetic algorithm-based approach that we designed for this purpose. We start by describing our initial, primitive attempt in Section 1. In Section 2, we describe some experiments that we ran with the initial GA. In Section 3, we present a refined algorithm that we designed as a solution to the problems revealed during our initial attempt. In Section 4, we describe experiments conducted with the refined GA. Throughout the discussion in this chapter, we describe the main design issues that we had to address. We focus on the aspects of the algorithm that are specific to our solution.

1. An Initial Genetic Algorithm

As seen in Chapter 3, genetic algorithms evolve populations of individuals (or chromosomes) where each individual can be used to represent a solution to the problem at hand. With this in mind, our first approach to the problem of optimizing software quality estimation models was to consider each model (rule set) as an individual and to design a GA that would evolve populations of such individuals. In this section, we describe the GA that was designed as a first attempt to solving the problem of optimizing rule sets. In order to make it easy for the reader to follow, we start by giving the skeleton of the traditional generic GA (Algorithm 1) and then, in separate sections, we focus on the parts that we designed and implemented in a way specific to our problem.

```
t ← 0 {Initialize time variable t.}
P(t) ← createInitialPopulation();
n ← |P(t)| {Assign to n the population size.}
repeat
  computeFitness(P(t)) {Evaluate the individuals in P(t).}
  elitism(percentage_elit);
  repeat
    i1 ← Select();
    i2 ← Select();
    if (crossoverDecided()==TRUE) then
      crossover(i1,i2) → c1 and c2;
    else
      c1 ← i1;
      c2 ← i2;
    end if
    c1 ←mutate(c1);
    c2 ←mutate(c2);
    copy(P(t + 1), c1);
    copy(P(t + 1), c2);
  until (|P(t + 1)| ≥ n)
  t ← t + 1;
until (stoppingCriteriaMet()==TRUE);
```

Algorithm 1: A Generic GA.

1.1. Chromosomes and Fitness Function. The first design issue that arises when writing a genetic algorithm is the choice of an encoding scheme or, in other terms, what constitutes a chromosome. This is a crucial step in the design of the solution since it influences greatly the difficulty of the search/optimization problem, as well as the genetic operators (such as mutation).

In our case, because the goal is to optimize software quality estimation models (in the form of rule sets), the approach that seems most natural is to consider each model as a chromosome. Since we are only interested in models that take the form of rule sets, we will use the phrases "rule sets" and "software quality estimation models" interchangeably throughout the rest of this chapter and the next. For the sake of simplicity, we start by representing each chromosome as a string of pointers to rules and a default classification label. Figure 4.1 shows an example of a rule set and the chromosome that represents it. The rule set in the figure contains 7 rules (labelled *Rule 1...Rule 7*) and a default classification label (*Default class: 0*). The chromosome is an array of pointers to rules and a classification label. In this array, the value RI_J points to rule J of rule set I and the value $D0$ indicates that the rule set has a default classification equal to 0. Each rule constitutes a gene in the chromosome.

Rule Set 1:
Rule 1: NMO > 1 ^ NMI <= 22 ^ SIX <= 0.222222 –> class 0
Rule 2: NOC > 1 ^ NOD <= 8 –> class 0
Rule 3: DIT > 1 ^ NMA <= 7 –> class 0
Rule 4: NMI > 10 ^ NMI <= 22 –> class 0
Rule 5: CLD <= 0 ^ NMA > 7 ^ SIX > 0.222222 –> class 1
Rule 6: NOC <= 1 ^ NMO <= 0 ^ NMI <= 6 –>class 1
Rule 7: NMI > 22 –> class 1
Default class: 0

Chromosome:
RS1 [R1_1 R1_2 R1_3 R1_4 R1_5 R1_6 R1_7 D0]

FIGURE 4.1. This is an example of a rule set constructed by C4.5 and the corresponding chromosome. The chromosome is formed of genes where each, except the last one, points to a rule in the corresponding rule set. The last gene encodes the default classification label of the rule set.

The GA is seeded with an initial population of chromosomes that represent rule sets. Based on the principles described in Chapter 3, the selection of individuals from which to produce the next generation is based on their fitness as measured by the fitness function. In this problem, since we seek to optimize the accuracy of the rule sets, it is natural to define the fitness of a chromosome to be the accuracy of the rule set that it represents (as defined in Chapter 2, Equation 2.1). Thus, we choose the fitness of a rule set R to be $f(R) = C(R)$ where f is the fitness function and C the accuracy measured on the specific data set.

Now that a chromosome and the fitness function have been defined, we can consider the process of evolution which consists of applying the two genetic operators, crossover and mutation, to the existing population, in order to produce a new population. As previously mentioned, the selection of chromosomes to produce progeny is based on the fitness of the individuals. In our algorithm, chromosomes are selected by the roulette-wheel technique (described in Chapter 3) to undergo crossover. Our choice of this selection technique was based on two considerations. First, this was an initial attempt to explore how well a simple GA will perform on our problem and we wanted the implementation to be as faithful as possible to the classical one presented by Goldberg [Goldberg, 1989]. The second consideration was the fact that the initial rule sets for our application did not exhibit a wide variation in the accuracy. Hence, it was safe to use the roulette-wheel selection technique without biasing the GA strongly towards fitter individuals (an issue described in Chapter 3, Section 3.3).

1.2. The Genetic Operators. For simplicity, we use single-point crossover when two chromosomes are selected to create offspring. For this, a random cut point is defined and the chromosomes are cut at this location. Because chromosomes can be of different lengths, the cut point is always generated within the boundaries of the shorter one. Two offspring are then generated. The first one gets the first part of the first parent (before the cut point) combined with the second part of its second parent (after the cut point). The second offspring is formed by combining the first part of the second parent with the second part of the first parent. From the perspective of

the rule set, this can be seen as combining rules taken from two different rule sets and generating two new rule sets. Crossover happens with a certain probability. If no crossover occurs within a pair, the offspring are exact copies of their parents.

Before copying the offspring to the next generation, the GA mutates them with a certain probability. In an initial effort, we chose to implement a very conservative mutation operator in order not to perturb the rule sets much. In particular, we defined mutation to consist of flipping the default classification label of the rule set that a chromosome encodes to a value chosen randomly from the domain of the last gene (the set of classification labels). From the perspective of the rule set, this is equivalent to keeping the rules intact and modifying the default classification label of the rule set. This will help us evaluate how much impact the default classification label has on the accuracy of the rule set in the context of the specific data set. From a GA perspective, this can be seen as applying a *non-uniform mutation rate* (Chapter 3, Section 3.2.2) across the chromosome where the last gene is mutated with a probability greater than or equal to 0 while all others are mutated with probability 0. Figure 4.2 shows an example of the application of the genetic operators; the chromosomes are cut after the third gene, and one of the offspring is mutated.

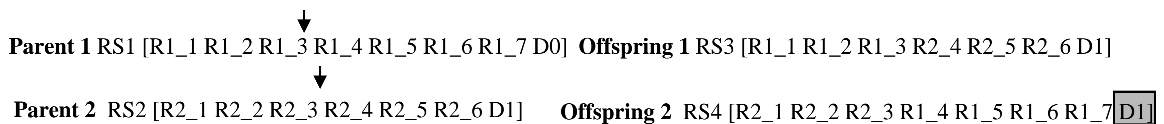

FIGURE 4.2. Example of crossover. Rule sets $RS1$ and $RS2$ are cut at gene 3. The two resulting offspring are $RS3$ and $RS4$. The latter is then mutated (mutated gene is in the shaded box).

Similar to the traditional GA, our algorithm maintains a fixed size population. To achieve this, at each iteration, crossover and mutation are repeated $\lfloor n/2 \rfloor$ times, n being the size of the population. Unlike the traditional GA, our GA performs elitism only when n is odd in order to complete the population[1]. This consists of copying the chromosome with the highest fitness to the next generation. The whole process of

[1]The traditional GA creates $n+1$ chromosomes when n is odd and then deletes the least fit one.

creating a new population is repeated until a pre-specified number of generations is reached. Algorithm 2 shows the pseudocode of a more detailed version of Algorithm 1.

```
t ← 0;
P(t) ← Parse(); {Read initial file of rule sets and create a population of chromo-
somes.}
n ← |P(t)|; {assign to n the population size.}
repeat
   for (i = 0 to n − 1) do
      ruleset[i] ← decode(P(t), i);
      f(i) ← computeAccuracy(ruleset[i]);
      i ← i + 1;
   end for
   if (n%2 ≠ 0) then
      P(t + 1) ← Elitism(1);
   end if
   repeat
      i1 ←rouletteSelect();
      i2 ←rouletteSelect();
      if (crossoverDecide()==TRUE) then
         singlePointCrossover(i1,i2) → c1 and c2;
      else
         c1 ← i1;
         c2 ← i2;
      end if
      c1 ←mutate(c1);
      c2 ←mutate(c2);
      copy(P(t + 1), c1);
      copy(P(t + 1), c2);
   until (|P(t + 1)| ≥ n)
   t ← t + 1;
until (stoppingCriteriaMet()==TRUE);
```

Algorithm 2: A more detailed version of the GA written as an initial attempt.

2. Experiments and Results

In order to assess the performance of this GA, we used the data from [Ikonomovski, 1998] which involves quality estimation models that assess the maintainability of software components in an object-oriented system. In this work, a component is a C++

class. One way to measure maintainability of a component is by assessing its fault-proneness (defined below). Fault-prone components require more maintenance than non-fault-prone ones. Cohesion, coupling and inheritance are object-oriented (OO) design product properties that have been widely used to predict fault-proneness [Basili *et al.*, 1996] and [Demeyer and Ducasse, 1999]. We give a brief definition of these terms before proceeding with our discussion.

According to [Fenton and Pfleeger, 1997], **cohesion** is characterized by "how closely the local methods are related to the local instance variables of the class". In other words, it is the degree to which the elements within the same component are linked. **Coupling** is the amount of linkage between different components of the same software system [Fenton and Pfleeger, 1997] and [Erdogmus and Tanir, 2002]. **Inheritance** is a relationship between two classes in which one class can be seen as a specialization of the other. In tables 4.1, 4.2 and 4.3, we show the metrics that are used to measure these properties.

In his thesis, Ikonomovski built two types of predictive models that establish the relationships between fault-proneness of a software component (OO class) on the one hand and cohesion, coupling and inheritance on the other. In one type of models, he classifies a component as:

- **non-faulty** - the component did not undergo any change of a corrective nature.
- **faulty**- one or more changes had to be done on the component during the development or maintenance phase.

In the other type of models, he classifies a component as:

- **non-faulty**- no change was performed on the component.
- **low-risk**- 1 to 4 changes of a corrective nature were performed on the component.
- **high-risk** - more than four changes of a corrective nature were performed on the component.

We refer to these as the 2-value estimation models and the 3-value estimation models, respectively.

The attributes in the predictive models are metrics that measure cohesion, coupling and inheritance. The predictive models take the form of rule sets constructed by C4.5 using data from an open multiagent system development environment called LALO[2] (developed and maintained at CRIM[3]). At the time when Ikonomovski experimented with LALO, the system contained 83 C++ classes and approximately 57 K source lines of C++ code. These classes translated to 83 instances in the data set that Ikonomovski used with 49 attributes. This data was used to construct 15 different trees with C4.5. Rule sets derived from the trees contain between 4 and 10 rules each for the 2-value estimation models and between 5 and 11 rules each for the 3-value models. The metrics used as attributes in these models are shown in Tables 4.1, 4.2 and 4.3. For a detailed explanation of the hypotheses that link these metrics to the maintainability of a software component, the reader can refer to [Ikonomovski, 1998]. At this point, we simply list them in the tables indicated above as we feel this is enough for the reader to understand our technique.

Metric Name	**Brief Description**
DIT	Depth Of Inheritance Tree
AID	Height Of Inheritance Tree
CLD	Class-to-Leaf Depth
NOC	Number Of Children
NOP	Number Of Parents
NOD	Number Of Descendants
NOA	Number of Ancestors
NMO	Number Of Methods Over(riden-loaded)
NMI	Number of Methods Inherited
NMA	Number of Methods Added
SIX	Specialization Index

TABLE 4.1. Inheritance Metrics.

[2]Langage d'Agents Logiciel Objet.
[3]*Centre de Recherche Informatique de Montréal*, Montreal, Quebec, Canada.

Metric Name	Brief Description
LCOM1	Lack of Cohesion in Methods
LCOM2	Lack of Cohesion in Methods
LCOM3	Lack of Cohesion in Methods
LCOM4	Lack of Cohesion in Methods
LCOM5	Lack of Cohesion in Methods
Coh	A Variation of LCOM5
Co	Connectivity
LCC	Loose Class Cohesion
TCC	Tight Class Cohesion
ICH	Information-flow-based Cohesion

Table 4.2. Cohesion Metrics.

2.1. Results. The results shown in this section have been published in [Sahraoui and Azar, 1999].

We tested our GA on the two fault-proneness data sets but it resulted in no improvement over the rule sets built by C4.5. As a matter of fact, the GA converged rapidly to the best chromosome that already existed in the initial population. In order to understand this behavior, we created an artificial data set by randomly changing attribute values in the real data set. The accuracy of the rule sets obtained with the GA on this data set was higher than that of the rule sets built by C4.5, as shown in the two runs described below. Before looking at the runs, we bring to the reader's attention the fact that we have experimented with several parameter values (crossover probability, mutation rate and location of cut point) and these proved to have very little effect on the best accuracy obtained. Hence, the results shown below can be viewed as typical results of the experiments. We show a single run for each different experiment.

First Run. In the first run, we used 3-value fault-proneness estimation models, a probability of 60% for the crossover, and a probability of 5% for the mutation. The cut point was set to 2 throughout the evolution process (i.e. we always cut the chromosomes after the second gene/rule, all chromosomes having a length strictly

Metric Name	Brief Description
CBO	Coupling Between Object classes
CBO'	Non-inheritance based Coupling
RFC_1	Number of methods invoked by the class
RFC∞	Number of methods that can be invoked by sending a message to the class
MPC	Message Passing Coupling
ICP	Information-flow-based Coupling
IH-ICP	Inheritance ICP
NIH-ICP	Non-Inheritance ICP
DAC	Data Abstraction Coupling
DAC'	Number of classes used as attribute types
IFCAIC	Inverse Friends Class-Attribute Import Coupling
ACAIC	Ancestors Class-Attribute Import Coupling
OCAIC	Others Class-Attribute Import Coupling
FCAEC	Friends Class-Attribute Export Coupling
DCAEC	Descendants Class-Attribute Export Coupling
OCAEC	Others Class-Attribute Export Coupling
IFCMIC	Inverse Friends Class-Method Import Coupling
ACMIC	Ancestors Class-Method Import Coupling
OCMIC	Others Class-Method Import Coupling
FCMEC	Friends Class-Method Export Coupling
DCMEC	Descendants Class-Method Export Coupling
OCMEC	Others Class-Method Export Coupling
IFMMIC	Inverse Friends Method-Method Import Coupling
AMMIC	Ancestors Method-Method Import Coupling
OMMIC	Others Method-Method Import Coupling
FMMEC	Friends Method-Method Export Coupling
DMMEC	Descendants Method-Method Export Coupling
OMMEC	Others Method-Method Export Coupling

Table 4.3. Coupling Metrics.

greater than 2). Figure 4.3 shows the initial population and Figure 4.4 shows the best (solid line) and the worst (dashed line) fitnesses obtained during evolution (only the generations that showed an improvement over the previous one are plotted).

The best rule set constructed by C4.5 had an accuracy slightly around 81% on the training set. At the end of the run, the GA could find a rule set with an accuracy of

RS1 [R1_7 R1_14 R1_5 R1_12 R1_1 R1_4 R1_13 R1_15 D0] 67.4699%
RS2 [R2_2 R2_14 R2_12 R2_7 D1] 74.6988%
RS3 [R3_7 R3_13 R3_5 R3_11 R3_1 R3_4 R3_12 R3_14 D0] 67.4699%
RS4 [R4_7 R4_13 R4_5 R4_11 R4_1 R4_2 R4_12 R4_14 D0] 66.2651%
RS5 [R5_7 R5_13 R5_5 R5_11 R5_1 R5_4 R5_12 R5_14 D0] 67.4699%
RS6 [R6_2 R6_11 R6_13 R6_8 R6_3 R6_12 R6_14 D0] 71.0843%
RS7 [R7_9 R7_7 R7_5 R7_12 R7_1 R7_11 D1] 68.6747%
RS8 [R8_7 R8_13 R8_5 R8_11 R8_1 R8_4 R8_12 R8_14 D0] 67.4699%
RS9 [R9_2 R9_10 R9_12 R9_4 R9_5 D1] 71.0843%
RS10 [R10_2 R10_11 R10_13 R10_9 R10_7 R10_12 R10_14 D0] 72.2892%
RS11 [R11_7 R11_12 R11_1 R11_17 R11_16 R11_11 R11_18 D1] 67.4699%
RS12 [R12_2 R12_12 R12_15 R12_6 R12_5 D1] 73.494%
RS13 [R13_2 R13_7 R13_8 R13_13 R13_4 R13_5 D1] 80.7229%
RS14 [R14_2 R14_11 R14_13 R14_9 R14_7 R14_12 R14_14 D1] 71.0843%
RS15 [R15_5 R15_11 R15_1 R15_7 R15_15 R15_3 R15_14 R15_4 R15_12 R15_16 D1] 71.0843%

FIGURE 4.3. Initial population of chromosomes built by C4.5. The number on each line indicates the fitness of the chromosome (accuracy of the rule set that the chromosome represents).

slightly above 90% on the same set. At the 28th generation, the population was formed of all similar chromosomes hence, not much further progress could be expected.

Second Run. In this run, we used also 3-value fault-proneness estimation models, a probability of 60% for crossover, and a probability of 5% for mutation. However, during crossover, we defined the cut point randomly for each pair of chromosomes (as opposed to having a fixed cut point throughout the evolution process). This choice was made to introduce more possible combinations among rules. We ran the GA through 100 generations. The fittest chromosome found had a fitness around 85.5% and was found at the 89th generation (compared to 90% found much earlier, at the 7th generation, in the previous run). We suspect that the difference is due to the size of the search space in both runs. In the first one, the chromosomes were cut at the same location during crossover. In the second one, they were cut at different locations which implies more possibilities of cut points and hence more possible combinations of rules (and hence, more rule sets). Enlarging the search space usually delays finding a good solution and at the same time can create more local optima, which we suspect was the case in the second run.

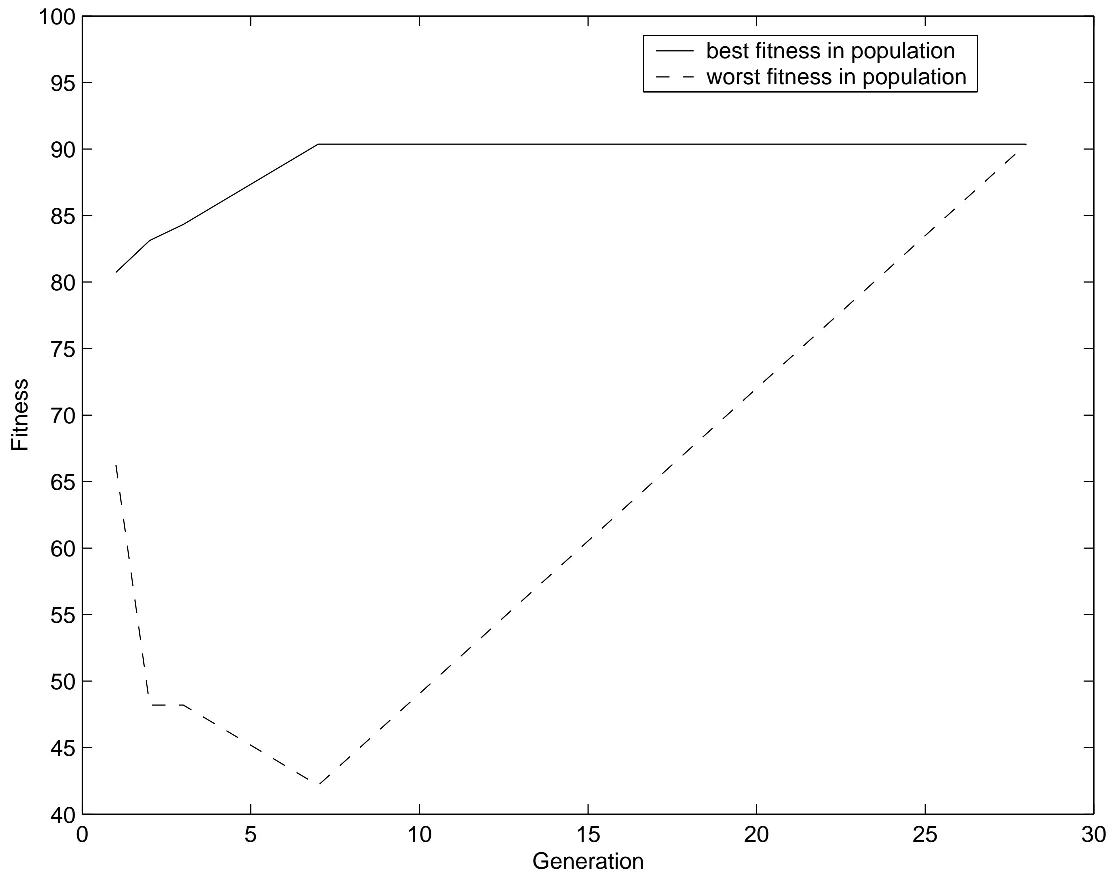

FIGURE 4.4. Best and worst fitness in each generation that shows an improvement over the previous one.

2.2. Modifications to the Algorithm. The previous experiments (with real data) indicated a premature convergence of the GA to a population of similar individuals representing the best rule set already existing in the initial population. This behavior was consistent across a wide range of parameter settings. In order to prevent premature convergence, we experimented with different elitism and mutation operators, as follows:

Elitism. We replaced the elitism operator with the random selection of a chromosome to be copied to the next generation. Still, the GA converged to the best rule set

already existing in the initial population when tested on the real data set. However, improvement was always obtained with the artificial data set.

Mutation. As one can see, the mutation operator does not allow the creation of new rules. It only changes the default classification label of the rule set. We defined two variants of this operator, both of which modified the rules. One mutation operator consists of deleting a condition in one of the rules in the rule set represented by the chromosome. Figure 4.5 shows an example.

Rule before mutation

NOC <= 1 ^ NOP > 0 ^ NMO <= 1 ^ NMI <= 10 ^ NMA <= 19 –> class 1

Rule after mutation

NOC <= 1 ^ NMO <= 1 ^ NMI <= 10 ^ NMA <= 19 –> class 1

FIGURE 4.5. The modified mutation operator that removes a condition from a rule. Condition $NOP > 0$ is removed.

Another mutation operator consists of changing the classification of a rule. Figure 4.6 shows such an example where the chromosome is mutated by changing the classification label of a rule from 1 to 0. In all these attempts, improvement was obtained with artificial data only.

2.3. Discussion and Summary. Although our first GA did not bring any improvement over the initial rule sets on the maintainability data set, it did improve the accuracy of rule sets on artificial data. We suspect that the lack of improvement on real data is due to two things. First, the rule sets constructed by C4.5 already have high accuracy, and given the small amount of data available, we may be witnessing a ceiling effect. As a matter of fact, in the case of the real data set, the rule sets have an average accuracy rate of 75.5% whereas, on the artificial data, the rule sets have an average accuracy rate of 70.5%. Second, the granularity of the encoding we use is too coarse and does not allow much variation. The crossover operator creates new combinations of the existing rules, but does not create any new rules. The variants of mutation that we defined either change the classification label of a rule or delete

Rule before mutation
NOC > 1 ^ NOP > 0 ^ NMO <= 1 ^ NMI <= 10 ^ NMA <= 19 –> class 1

Rule after mutation
NOC > 1 ^ NOP > 0 ^ NMO <= 1 ^ NMI <= 10 ^ NMA <= 19 –> class 0

FIGURE 4.6. The modified mutation operator that changes the classification label of a rule.

a condition but do not create any new conditions. This limited effect of the genetic operators on the rule sets, and especially the fact that no new conditions are created, might be causing the GA to converge to the best rule set that already exists in the population. An attempt to address this problem is presented in Section 3.

Despite the disappointing results that we obtained on real data, this initial attempt to use GAs to optimize software quality estimation models (in the form of rule sets) opened a new line of work. Below we present a refined GA that avoids the problem highlighted above by allowing crossover and mutation to work at a much finer level.

3. A Refined GA for Combining Rule Sets

The technique described in Section 1 suffers from a major drawback namely, the granularity of the genetic operators can operate. More precisely, since each gene in a chromosome represents a rule (or the default classification label[4]), the cut points during crossover can only fall between rules (as opposed to within rules). This leads to the crossover allowing only a recombination of already existing rules. Mutation also has very little effect on rules - it modifies the default classification label of the rule set, the classification label of the rule, or drops a condition in the rule. None of this proved to be enough to improve on the accuracy of the initial rule sets. In this section, we describe a GA that can be seen as a refinement to the previous one. Like the previous GA, it considers each rule set as a chromosome, and a population is formed of a number, n, of rule sets. But here, we drop the restriction of a rule being

[4]The default classification label can be thought of as a rule with an empty left-hand side i.e any case satisfies its left-hand side.

a gene. Instead, the rule set is represented as an array of conditions and classification labels. With this new, finer granularity, the genetic operators can now operate on the level of conditions as well as classification labels. This allows the introduction of more variety in the rule sets generated because changes can now occur at the level of the conditions as well. In the remainder of this section, we focus on the differences between this GA and the previous one.

3.1. Chromosomes and Fitness Function. Like in the previous approach, the initial population is formed of chromosomes that represent rule sets. However, the genes are at a finer level: Each gene represents either a condition or a class label. We call the genes that represent class labels **special genes**. There are $m+1$ special genes in a chromosome that represents a rule set of m rules (one for the classification label of each rule plus the default classification label of the rule set). Figure 4.7 shows an example of a rule set and its representation as a chromosome. We define the fitness of a chromosome to be equal to the accuracy of the rule set ($f(R) = C(R)$) since this is the measure we are trying to optimize.

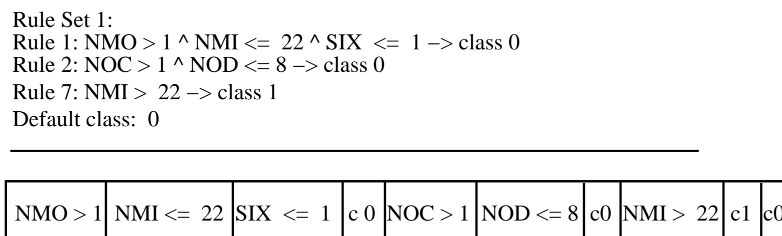

FIGURE 4.7. A rule set and its chromosome representation. Special genes, representing classification labels, are underlined.

In order to pass from one generation to the next, we implemented the two traditional selection techniques, roulette wheel and rank selection. In Section 4 we discuss how each influences the behavior of the GA. Chromosomes are selected to undergo crossover and mutation. The GA also copies chromosomes by elitism in order to conserve the best found during the process of evolution. In order to have more control on the number of rule chromosomes to preserve during the process of evolution, we

designed elitism to copy $x\%$ of the chromosomes in the current population to the next (as opposed to only one). The value of x is decided at the onset of the experiments.

3.2. The Genetic Operators. The two main genetic operators that the GA uses to create new chromosomes are crossover and mutation.

3.2.1. *Crossover.* Because the definition of genes is different in this refined GA, during crossover, a cut point can fall between rules or within a rule. The latter case allows conditions to be taken from different rules and recombined. The former one allows rules to be exchanged between rule sets.

We have experimented with two crossover operators: single-point crossover and double-point crossover. When single-point crossover occurs, a cut point is generated randomly in each of the parent chromosomes. This can fall within a rule or on a rule boundary (between rules or between a rule and the default classification label of the rule set). In order to ensure the validity of the chromosomes resulting from crossover, we imposed the following restrictions on where a cut point can fall. Let us designate by $p1$ and $p2$ the two chromosomes undergoing crossover (the parent chromosomes) and by $i1$ and $i2$ the cut points in $p1$ and $p2$, respectively. If $i1$ falls within a rule in $p1$, $i2$ should fall within a rule in $p2$. If $i1$ falls on a rule boundary in $p1$, $i2$ should also fall on a rule boundary in $p2$. For this, the algorithm keeps indices of the locations in a chromosome that delimit rules and those that lie inside rules. These indices designate either **within rule cut points** or **boundary cut points**. When the GA generates a cut point, it can decide from which set of indices to pick its second cut point by checking the type of the first one. The cut point is not allowed to fall on a chromosome boundary (i.e. before the first cell in the string or after the last one).

Figures 4.8 and 4.9 show two examples of crossover, where the cut points fall within rules and on rule boundaries, respectively. Figure 4.10 shows an example of an invalid crossover, where $i1$ falls within a rule in $p1$ and $i2$ falls on a rule boundary in $p2$. As we can see from the figure, *offspring 1* represents an invalid rule set (it contains a rule with no classification label).

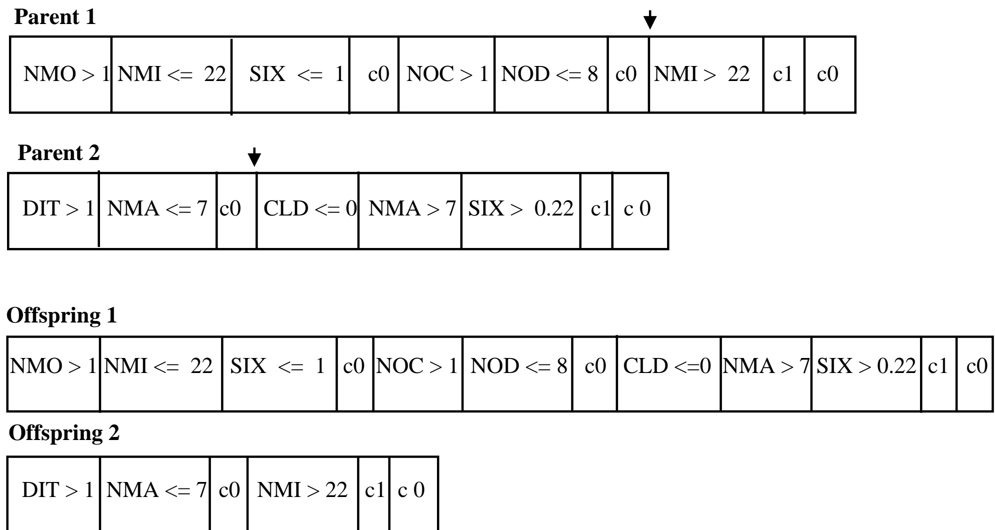

FIGURE 4.8. Single point crossover where the cut point falls on a rule boundary.

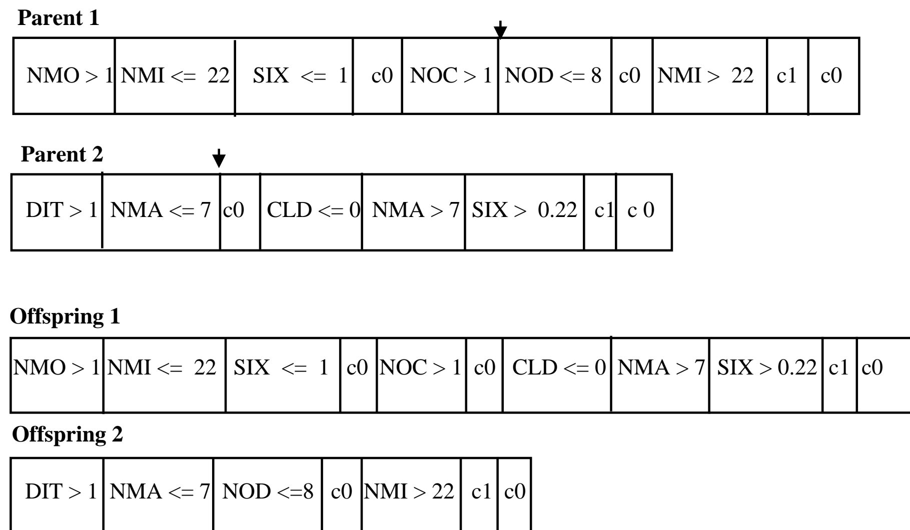

FIGURE 4.9. Single point crossover where the cut point falls within a rule.

Since the length of the chromosomes was sufficiently large, we found it interesting to implement and test double-point crossover, as well. As we can recall from Chapter 3, when double-point crossover occurs, two cut points are generated within each of the parent chromosomes and the offspring take alternating segments of both parents. Traditionally, both parents are cut at the same locations. In our case, we

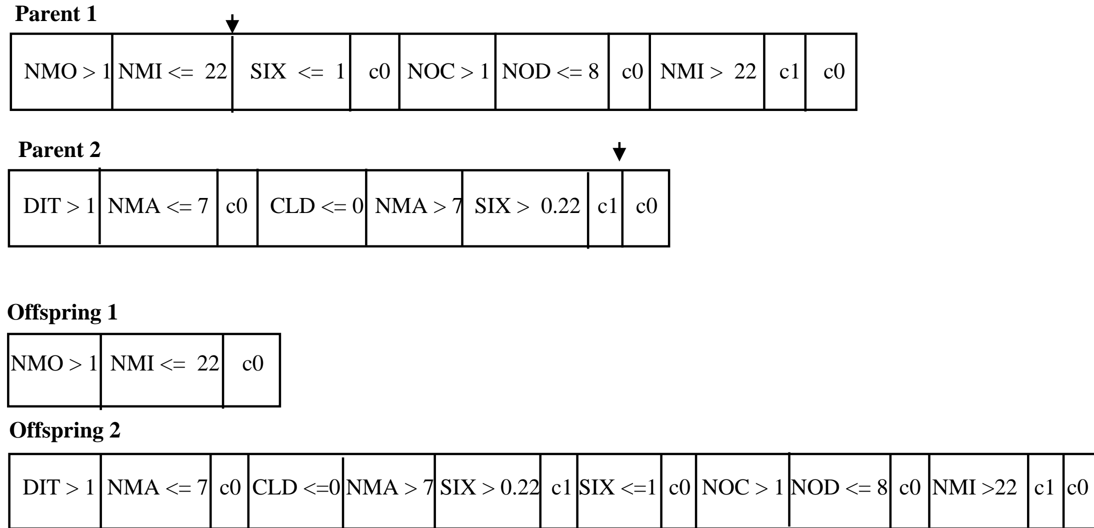

FIGURE 4.10. Single point crossover where the cut point falls within a rule in parent 1 and on a rule boundary in parent 2. This results in one of the offspring (offspring 1) representing an invalid rule set (missing classification label).

allow for cut points to fall at different locations in each parent. This allows for more variety in the combination of conditions and allows rules of sizes different from their parents to be formed. Here also, we ensure the validity of the offspring by imposing restrictions on where the cut points can fall with respect to each other. To make the implementation simple, if the first cut point within the first parent falls within a rule, the same should happen for the first cut point within the second parent. If it falls on a rule boundary, the same is also imposed on the cut point in the second parent. The same rules are applied to the second cut point. Similarly to the previous algorithm, the cut points cannot fall on the boundaries of the chromosomes. Of course, since crossover happens only with a certain probability, it is possible that offspring end up being duplicates of their parents.

3.2.2. *Mutation.* In this refined GA, a gene represents either a condition or a classification label. In the case where a special gene is mutated, the value is changed to another one chosen randomly from the domain of the classification labels. For all other genes which represent conditions of rules, mutation consists of changing the value to which the attribute is compared to another value picked randomly from the

set of cutpoints for this attribute. A **cutpoint** for an attribute, not to be confused with the crossover cut point is the median of the two values of the attribute where the classification label changes. Unlike the previous technique, this allows the creation of new conditions. We choose a value from the set of cutpoints of an attribute because these are more likely to change the set of examples that the rule classifies.

3.3. Trimming. One can easily see that crossover and mutation, as defined in this GA, might result in redundancy and inconsistency. **Redundancy** exists in a rule r when a condition c_i implies or is the same as another condition c_j in r. **Redundancy** exists in a rule set R when R contains two or more rules that have the same conditions (the order of the conditions is not important) and the same classification label. Figure 4.11 shows an example of redundancy that occurs in the first rule in the rule set. Condition $NMO > 7$ implies condition $NMO > 1$. Figure 4.12 shows an example of redundancy in a rule set where the first and the third rules are the same (the two conditions are: $NMO > 1 \wedge NMI \leq 22 \wedge SIX \leq 1$ c 0 and $NMI \leq 22 \wedge SIX \leq 1 \wedge NMO > 1$ c 0).

NMO > 1	NMI <= 22	NMO > 7	c0	NOC > 3	NOD <= 8	NOC <= 1	c0	NMI <= 22	SIX <=1	NMO >1	c0	1

FIGURE 4.11. Redundancy in the first rule. The third gene represents a condition that implies the one represented by the first gene.

NMO > 1	NMI <= 22	SIX <= 1	c0	NOC > 1	NOD <= 8	c0	NMI <= 22	SIX <=1	NMO >1	c0	1

FIGURE 4.12. Redundancy in a rule set. The first and the third rules are the same.

The algorithm eliminates redundancy inside the rules by combining conditions c_1, ..., c_n if one implies the others. In particular, if condition c_i implies condition c_j, the latter is deleted. It eliminates redundancy inside the rule sets by eliminating duplicate rules and keeping only one instance of each such rule.

We define **inconsistency** inside a rule r to occur whenever r has two or more conditions that cannot be true at the same time. Figure 4.13 shows an example of a rule that has the two conditions $NOC > 3$ and $NOC \leq 1$, which cannot be true at the same time.

NMO > 1	NMI <= 22	SIX <= 1	c0	**NOC > 3**	**NOD <= 8**	**NOC <= 1**	**c0**	NMI <= 22	SIX <=1	NMO >1	c0	1

FIGURE 4.13. Inconsistency in the second rule (shown in bold). The conditions $NOC > 3$ and $NOC \leq 1$ cannot be true at the same time.

Inconsistent rules are allowed to remain in the rule sets as long as the evolution is ongoing. Some conditions, although "fatal" in some combinations, might turn out to form good rules when combined with other conditions. Inconsistent rules are eliminated only when the last population is created (there is no need to keep these conditions anymore, and they are disturbing to human experts). Rules are sorted by the classification label inside the rule set. This is inspired from C4.5 and has two advantages: the order of the rules for one classification label does not matter anymore and the rule set is easier to interpret by human experts [Quinlan, 1993]. It is possible for the rule set to be inconsistent (for example, a rule set can have the two rules $NOC \leq 22 \rightarrow class\ 0$ and $NOC \leq 22 \rightarrow class\ 1$. Such rules are kept inside the rule sets as the sequential classification process allows one of them to fire only (this does not lead to any inconsistent classification).

4. Experiments and Results

4.1. Description of the Data Sets. Most of the experiments were conducted using data sets in which the goal is to predict the stability of a software component, based on several metrics. For the purpose of this thesis, a **software component** is a class in an object-oriented software system. During its operation time, a software component undergoes several changes due to error detection, environment changes, etc. It is important for a software component to remain stable when

these changes occur. A component is said to be **stable** if its public interface remains valid between different versions; otherwise, the component is said to be **unstable**. We used 2 data sets, **STAB1** and **STAB2**, prepared by Salah Bouktif at Université de Montréal, Canada.

(i) **STAB1**: This data set was generated using the 11 software systems listed in Table 4.4 and the 19 metrics shown in Table 4.5. These metrics correspond to four groups: cohesion, coupling, inheritance and size complexity and the Stress metric. Fifteen subsets (of size 1, 2, 3, and 4) were created by combining these groups of metrics (the Stress metric was always included). The combination was based on the relationship desired among the different quality characteristics. For example, for the goal of constructing rule sets that estimate stability of a component based on its complexity and coupling, only the metrics that measure these two characteristics are included.

Software Systems
Bean browser
Ejbvoyager
Free
Javamapper
Jchempaint
Jedit
Jetty
Jigsaw
Jlex
Lmjs
Voji

TABLE 4.4. Software systems used to build decision trees and rule sets with C4.5.

The 15 subsets of metric groups were used with the 11 software systems to extract 165 (11 X 15) data sets used to build decision tree classifiers with C4.5. Constant classifiers (classifiers that have a single classification label) and classifiers with a training error of more than 10% were eliminated. Then, 40 decision trees were selected randomly from the retained ones.

These decision trees were converted into rule sets using C4.5. Our GA evolves these rule sets using the four software systems shown in Table 4.6. From these, a data set of 2920 cases was extracted (Table 4.6). Ten-fold cross validation was performed to assess the GA. The data set was randomly split into 10 folds of roughly equal size. Nine out of the ten folds were randomly chosen and combined to form the learning (or training) set for the GA and the remaining fold constitutes the testing set. Throughout our discussion, we refer to one split as a **(training, testing)-pair**. Ten such pairs are possible. One run of the GA consists of learning and testing on all 10 (training,testing)-pairs.

(ii) **STAB2**: The previous data set is imbalanced (one classification label appears much more frequently than the other). As a matter of fact, 2481 of the 2920 cases are stable and the remaining ones are unstable. In order to test our algorithm on a balanced data set, we used **STAB2** which also involves stability. Twenty two software metrics were extracted as shown in Table 4.8. Nine of the 11 software systems shown in table 4.4 are used to build the experts with C4.5. These are the ones shown in table 4.7. The remaining two, namely, `Jedit` and `Jetty`, were used to train and test the GA. For this, 15 subsets of groups of metrics were created by combining 1, 2, 3, or 4 groups in all possible ways. These subsets were used with the 9 chosen software systems (Table 4.7) to create 135 (9X15) data sets. C4.5 was used on each of these data sets to construct a classification model in the form of a rule set. The models with an error rate greater than 10% and constant classifiers were eliminated and 23 were retained. 10-fold cross-validation is used like before to run the GA.

(iii) **MAINT**: This is the data set described in Section 2. It estimates fault-proneness (as an indicator of maintainability) based on cohesion, coupling and inheritance. We consider the 2-value classification models only. To prepare the data for the GA and in order to simulate the same setup that

Name	Description
	Cohesion metrics
LCOM	lack of cohesion methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
	Coupling metrics
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
	Inheritance metrics
NOC	number of children
NOP	number of parents
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
	Size complexity metrics
NOM	number of methods
WMC	weighted methods per class
WMCLOC	LOC weighted methods per class
MCC	McCabe's complexity weighted meth. per cl.
NPPM	number of public and protected meth. in a cl.
NPA	number of public attributes
	The stress metric
STRESS	stress applied to the class

TABLE 4.5. Software quality metrics used as attributes in **STAB1**.

JDK version[5]	Number of classes
jdk1.0.2	187
jdk1.1.6	583
jdk1.2.004	2337
jdk1.3.0	2737

TABLE 4.6. STAB1- Software systems used to train and test the GA.

we had with **STAB1** and **STAB2**, we divided the data sets into two parts of roughly equal size: one was used to create decision trees with C4.5 and

the other was used to train and test the GA (using 10-fold cross validation technique).

Software Systems
Bean browser
Ejbvoyager
Free
Javamapper
Jchempaint
Jigsaw
Jlex
Lmjs
Voji

TABLE 4.7. Software systems used to build decision trees and rule sets with C4.5.

Next, we describe the different experiments that we ran to assess the performance of our GA. We group these experiments by parameter setup. In order to account for the element of randomness in the GA, each experiment was repeated 30 times and the average over the 30 runs was reported. During each run, 10-fold cross-validation is used whereby the training set is divided into 10 subsets, the GA is trained on 9 and tested on the remaining one. For each setup, we describe the parameter values and then we give the results obtained on the three data sets described above. In Section 4.6, we summarize all the results in Table 4.13.

When assessing our GA, we give four measurements: the accuracy on the training set, the accuracy on the testing set, the J_index on the training set and the J_index on the testing set (all of them defined in Chapter 2, Section 2). For each experiment, we show the best rule set in the initial population and the one in the last population. In the context of these experiments, the **best** rule set is the one with the highest accuracy measured on the training set. We are measuring the J_index as well as the accuracy because the data set in **STAB1** is imbalanced (one classification label appears much more frequently than the others). For this, we find it important to measure the average accuracy per class label in the case of **STAB1** as well as the

Name	Description
Cohesion metrics	
LCOM	lack of cohesion methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
Coupling metrics	
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
CUBF	number of classes used by a member function
Inheritance metrics	
NOC	number of children
NOP	number of parents
NON	number of nested classes
NOCONT	number of containing classes
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
Size complexity metrics	
NOM	number of methods
WMC	weighted methods per class
WMCLOC	LOC weighted methods per class
MCC	McCabe's complexity weighted methods per class
DEPCC	operation access metric
NPPM	number of public and protected methods in a class
NPA	number of public attributes

TABLE 4.8. Software quality metrics used as attributes in **STAB2**.

overall accuracy. However, we train our GA to optimize the accuracy of the rule set rather than its J_index. Hence, an improvement is guaranteed on the training accuracy, but not necessarily on the J_index. We decided to use the accuracy in our fitness function rather than the J_index because most of the work that has been done in building software quality estimation models uses the accuracy rather than the J_index, as a measure of performance.

4.2. SETUP I: Single Point Crossover and Roulette Wheel. We start with a setup similar to the previous GA in which roulette wheel and single point

Parameter	Value	Parameter	Value
crossover probability	0.9	**Generations**	300
mutation rate	0.1	**Selection technique**	Roulette Wheel
Elitism	10%	**Crossover type**	Single point

TABLE 4.9. Experiment SETUP I: Single Point Crossover and Roulette Wheel.

crossover were used. These two choices of parameters make the GA as similar as possible to the classical one described in [Goldberg, 1989].

The experiments in this setup had: crossover probability ($\aleph$) of 0.9, mutation rate (μ) of 0.1 (applied uniformly across a chromosome). The percentage of chromosomes copied by elitism (ε) was set to 10% and the number of generations (G) was set to 300. Roulette wheel selection technique was used as well as single-point crossover. Table 4.9 summarizes these parameters.

On **MAINT** (Figure 4.14), the GA achieved an accuracy of 85% on the training set and about 54% on the testing (compared to 72% and 51%, respectively, for C4.5 rule set) and a J_index of 81% on the training set and 47% on the testing set (compared to 65% and 45% for C4.5). We believe the big difference between the accuracy on the training set and the accuracy on the testing (and between the J_index on the training set and the J_index on the testing) is due to the small size of the data set.

On **STAB1** (Figure 4.15), almost no improvement was made by the GA as it found a rule set with an accuracy equal to 86% on the training set and 85.5% on the testing set (the best rule set constructed by C4.5 had an accuracy of 85% on both sets). However, the GA improved the J_index and it reached 60.5% on the training set and 59% on the testing set compared to 51% for C4.5 on both sets. It is worth recalling that **STAB1** is imbalanced. According to [Elomaa, 1994], when the data set is imbalanced, it is important to compare the performance of learning algorithms to the **majority classifier**, which classifies all the cases to be of the majority category. In the case of **STAB1**, the majority classifier had an accuracy of 84.95 % on the training set 84.7 % on the testing set and a J_index of 50% on both sets. Hence, the GA outperformed it.

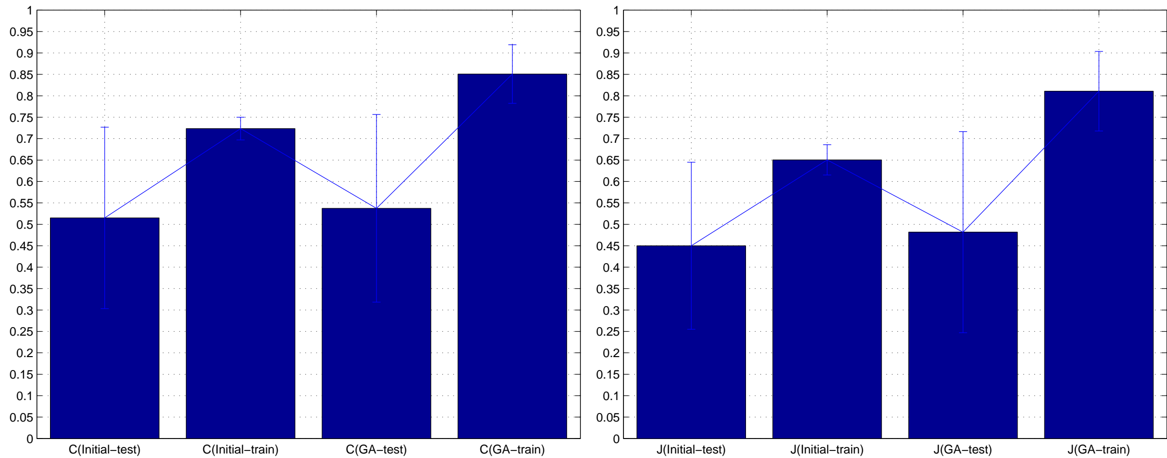

FIGURE 4.14. MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover.

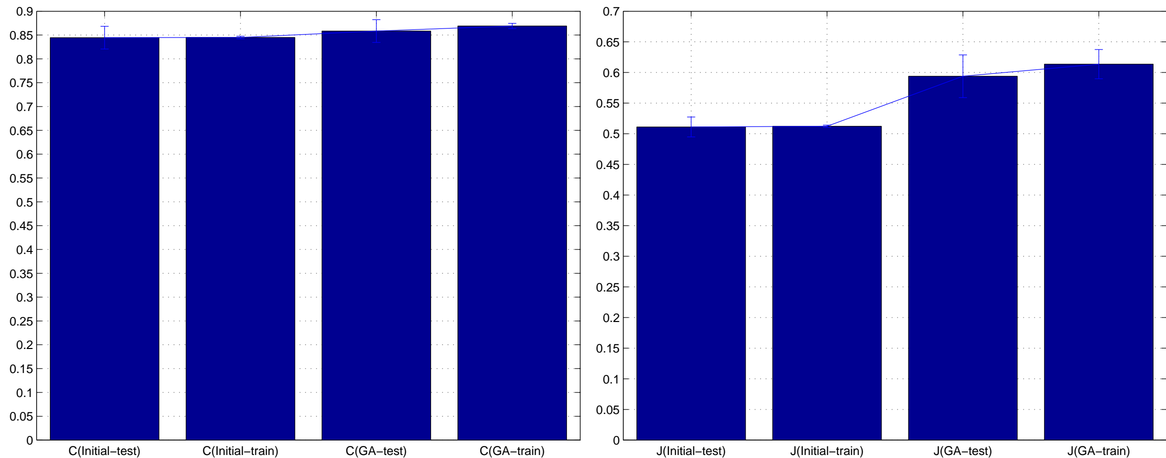

FIGURE 4.15. STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover.

STAB2 is a balanced data size that is bigger in size than **MAINT**. Figure 4.16 shows the results on this data set. The GA could achieve an accuracy of 74.5% on the training set and 70% on the testing set and a J_index of 65% on the training set and

60.5% on the testing set. The best rule set constructed by C4.5 had an accuracy of 68% and a J_index of 58% on both sets and hence, our GA outperforms it. **STAB2** is a balanced data set and improving the accuracy on a balanced data set is harder than improving it on an imbalanced one according to existing literature, e.g. [Elomaa, 1994].

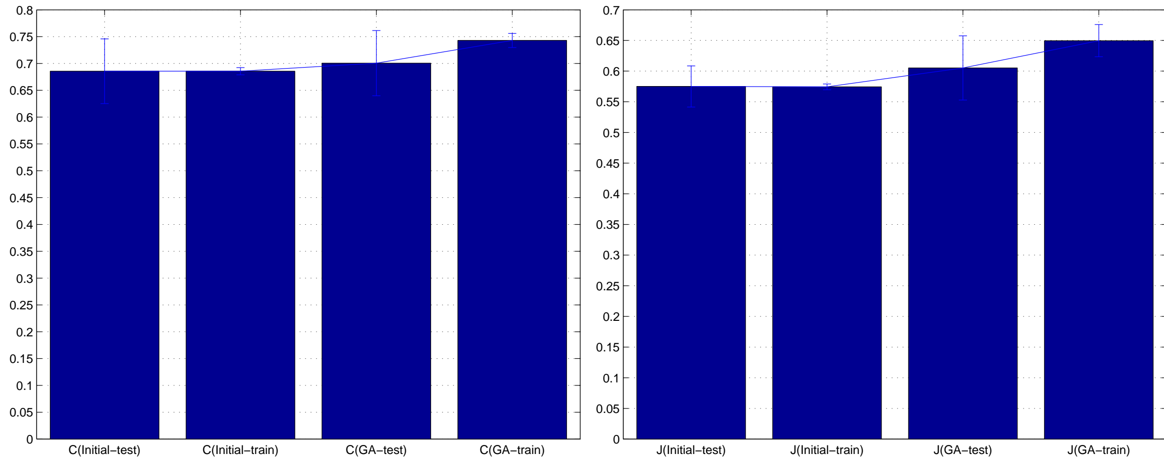

FIGURE 4.16. STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover

.

In order to assess the performance of our GA with any rule sets (not only those constructed by C4.5), we built random ones. For this, attributes were picked randomly from the set of attributes that describe the data. Conditions were created by combining each randomly chosen attribute with a relational operator from the set $\{>, \leq\}$[6] and a cutpoint also chosen at random from the set of cutpoints for this attribute. The number of rules included in these rule sets varied from 5 to 70 and the number of conditions included in the rules varied between 3 and 30. We repeated the experiments using the random rule sets as an initial population for the GA, instead of C4.5 rule sets. The same set of random rule sets was used across all experiments.

[6]The algorithm is not restricted to these two operators. We chose these two, in the experiments, in order to be consistent with the syntax of the rule sets created by C4.5.

Figures 4.17, 4.18 and 4.19 show the results. It is worth pointing out that the randomly generated rule set chosen as the best in the initial population happened to have a high accuracy (higher than C4.5). It is not representative of the rest of the rule sets in the same population which had a much lower accuracy. In the case of **MAINT**, the GA was able to attain an accuracy of 94% on the training set and 63% on the testing (compared to 70% on both sets for the initial random rule set). The improvement was also noticeable in the case of the J_index computed on the training set. The GA achieved a J_index of 91% on the training set and 55% on the testing set whereas the best rule set in the initial population had a J_index of 61% on the training set and 60% on the testing set.

On **STAB1**, almost no improvement was noticed in the accuracy but the J_index achieved by the GA was about 56% on both sets (compared to 50% for the best rule set in the initial population and 50% for the majority classifier).

One important detail to point out is the shape of the rule sets that are found in the last population when the GA is seeded with random rule sets in the case of **STAB1**. Most of the time, these rule sets are formed of one or two attributes only (most of the time, one attribute). This happened most of the time in the case of **STAB1** but only very rarely in the case of **STAB2** and **MAINT**. In our opinion, this is due to the fact that the data is imbalanced and one attribute or two attributes are enough to split it into positive and negative examples.

On **STAB2**, the best accuracy achieved by the GA was 73% on the training set and 69% on the testing set (compared to about 65% for the best rule set in the initial population on both the testing and the training sets). The GA also improved the J_index as it scored 65% on the training set and 60% on the testing set (compared to 50% on both sets for the best in the initial population of rule sets).

4.3. SETUP II: Double Point Crossover and Roulette Wheel. In order to see how double-point crossover affects the performance of the GA, we set up experiments to run with the following parameters: crossover probability ($\aleph$) was 0.9, mutation rate (μ) was 0.1 (applied uniformly throughout a chromosome). The

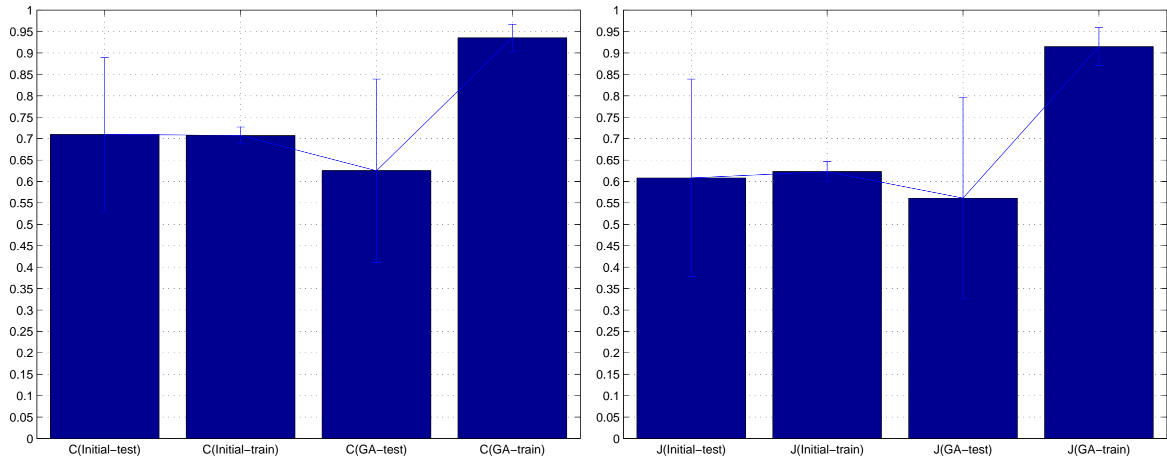

FIGURE 4.17. MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover.

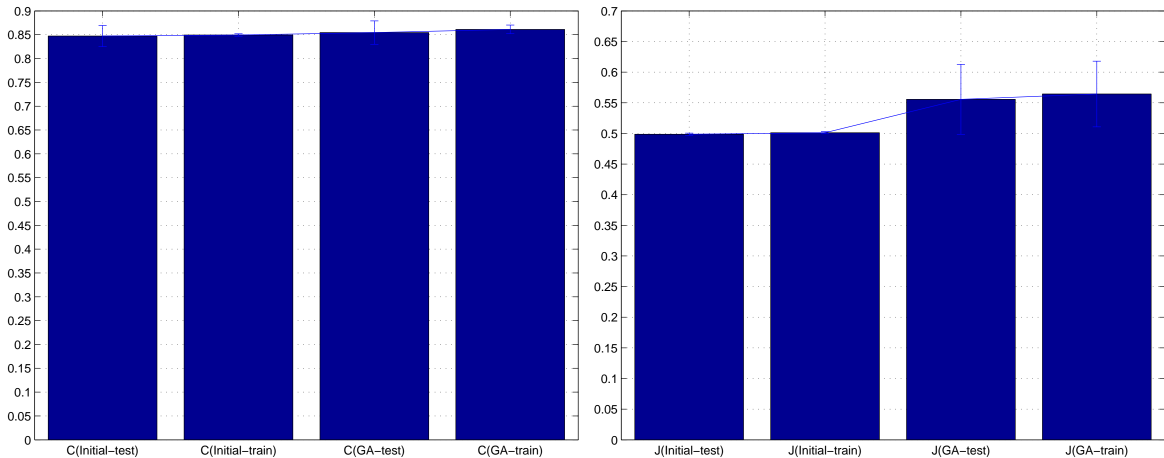

FIGURE 4.18. STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover.

percentage of chromosomes copied by elitism (ε) was set to 10 and the number of generations (G) was set to 300, roulette wheel selection technique was used as well as double-point crossover. Table 4.10 summarizes these parameters.

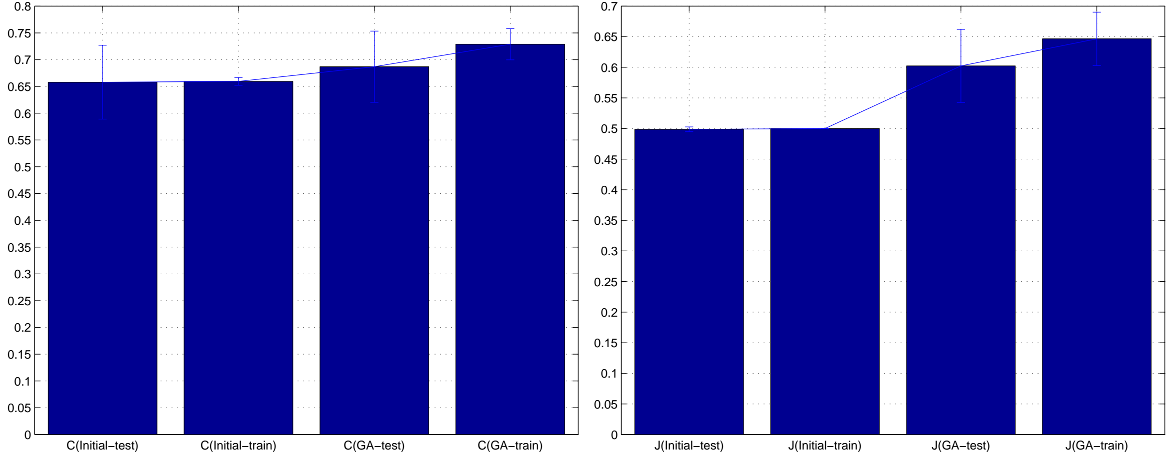

FIGURE 4.19. STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=single-point crossover.

Parameter	**Value**	**Parameter**	**Value**
crossover probability	0.9	**Generations**	300
mutation rate	0.1	**Selection technique**	Roulette Wheel
Elitism	10%	**Crossover type**	Double point

TABLE 4.10. Experiment Setup II: Double Point Crossover and Roulette Wheel

Figures 4.20, 4.21 and 4.22 show the results on all three data sets. In the case of the maintainability data set, the accuracy achieved by the GA is the same as in the previous setup (85% on the training set and 54% on the testing set) and the J_index is only slightly different. The rule set obtained by the GA has a J_index of 82% on the training set and 49% on the testing set (compared to 81% and 47%, respectively, in the last setup). Hence, the results are very similar in both setups.

On **STAB1** (Figure 4.21), the GA could reach a J_index of 60% on the training set and 57% on the testing set (compared to 60.5% and 59%, respectively, in the previous setup). The accuracy is the same as in the previous setup. On **STAB2** (Figure 4.22), the results were very close to those obtained in the previous setup.

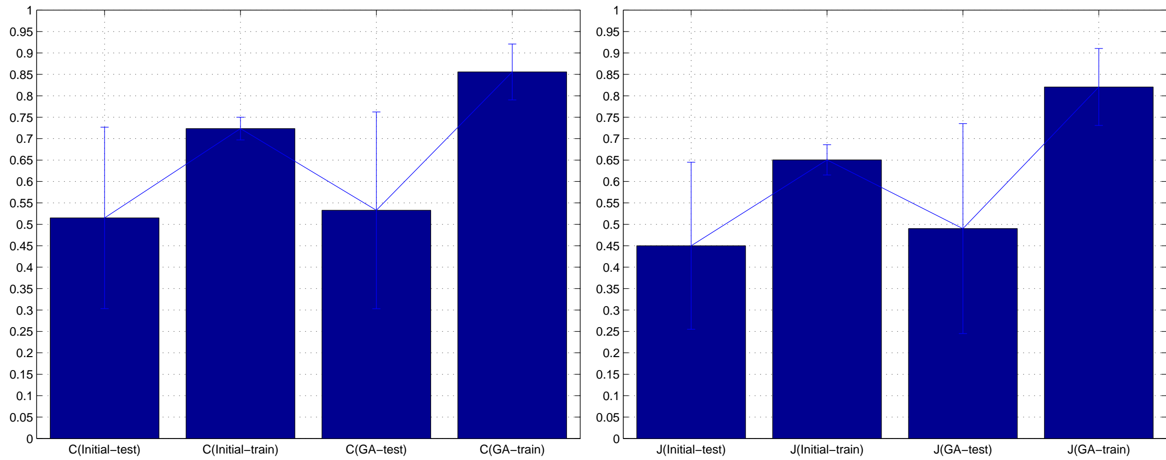

FIGURE 4.20. MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover.

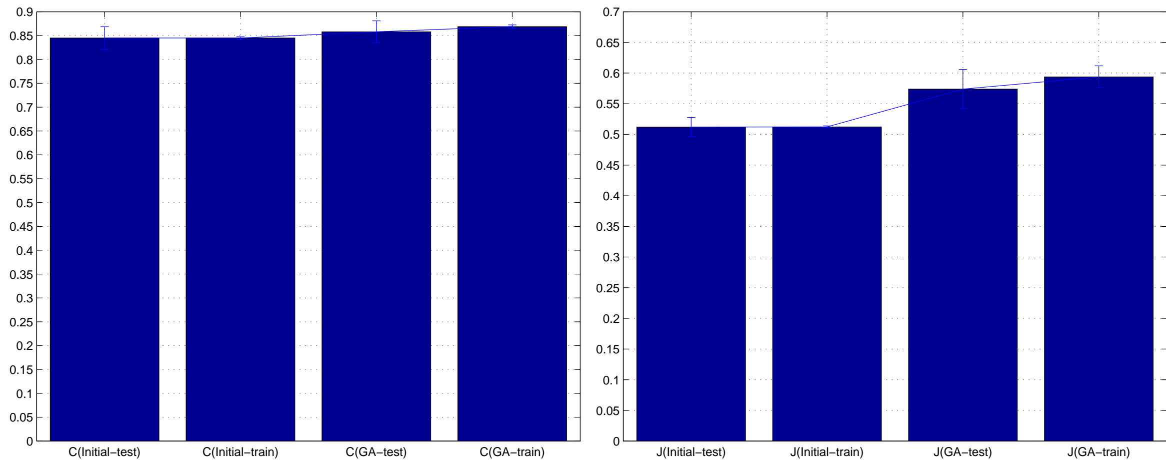

FIGURE 4.21. STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover.

Since the GA was designed to optimize the accuracy on the training set, and the accuracy obtained in this setup was very similar to the one obtained in the previous

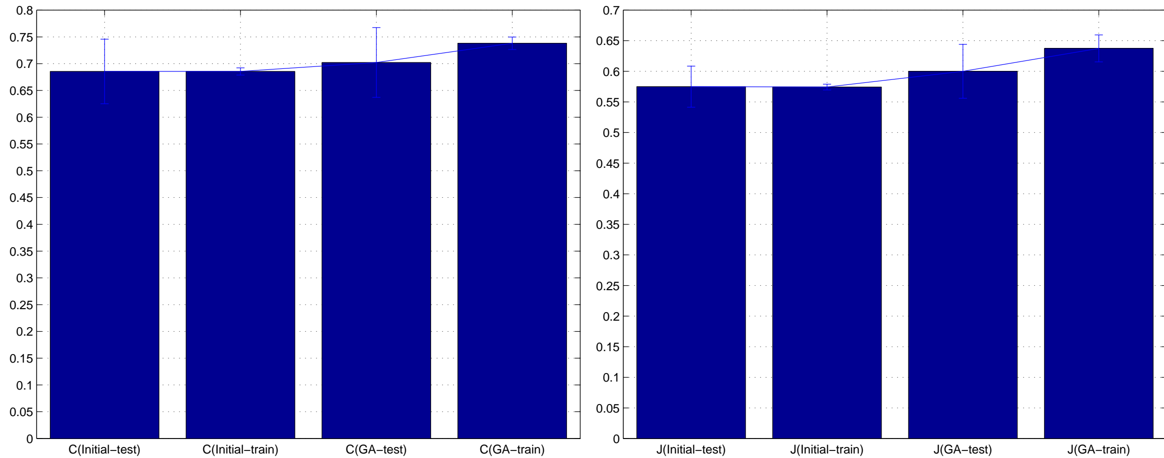

FIGURE 4.22. STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover.

setup, we can conclude that double-point crossover did not affect the GA behavior (when combined with the roulette wheel selection technique).

We repeated the same experiments seeding the GA with a population of random rule sets (Figures 4.23, 4.24 and 4.25). The only difference was noticed in the case of the maintainability data, on which the GA achieved an accuracy of 91% on the training set and 61% on the testing set (compared to 94% on the training set and 63% on the testing set in the previous setup). It outperformed C4.5 rule set which had an accuracy of 73% on the training set and 51% on the testing set.

4.4. SETUP III: Single Point Crossover and Rank Selection. As already discussed in Chapter 3, roulette wheel suffers from a drawback when the variance of the fitness is high in a population: lower fitness chromosomes are not given a fair chance to survive and produce progeny as most of them die out early in the process of evolution. In the context of our work, rule sets are not given a chance to survive to the next generation because of their low accuracy. However, as one can imagine, some conditions might be 'fatal' when they are combined with others in a rule. This can deteriorate significantly the accuracy of the rule and hence the rule

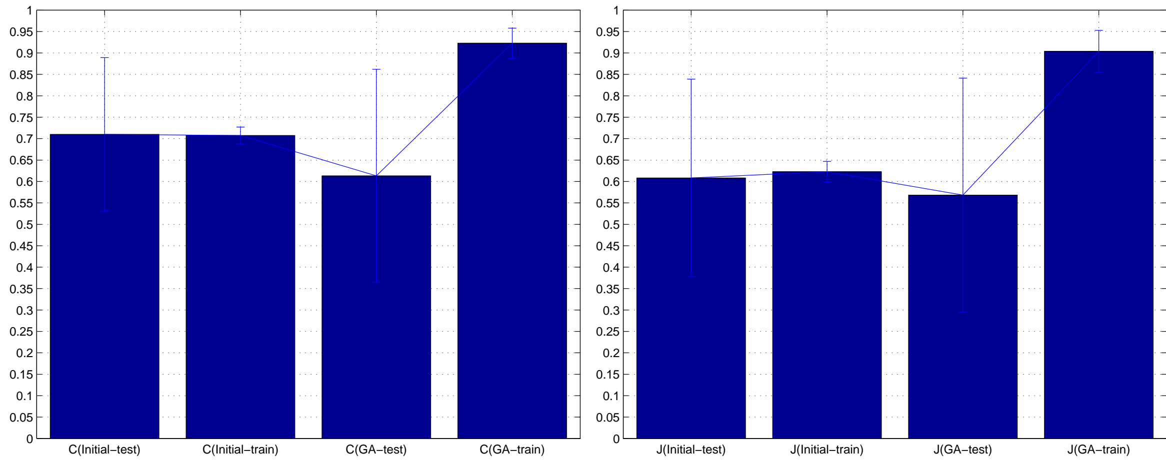

FIGURE 4.23. MAINT-Accuracy (C) and J_index (J) of a random rule set and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover.

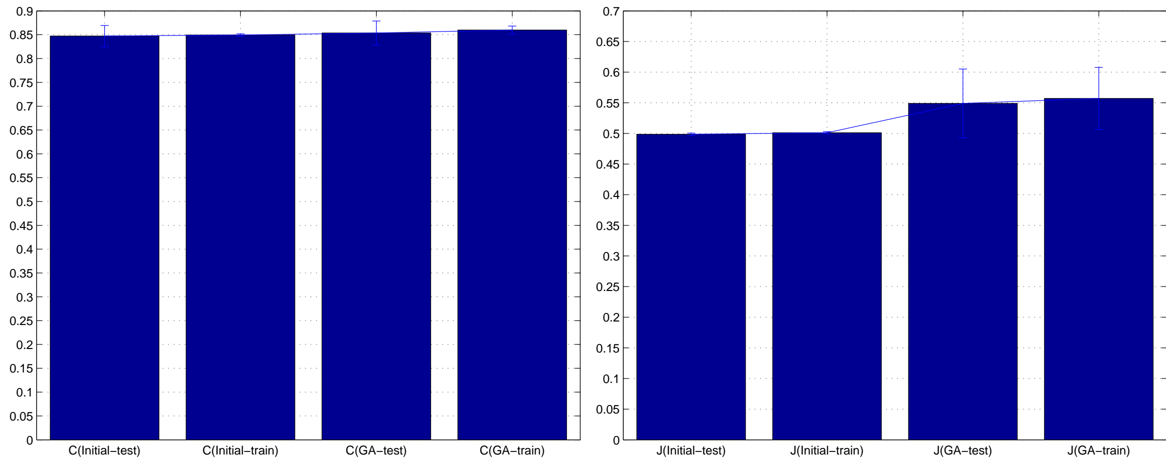

FIGURE 4.24. STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover.

set that contains it (if the coverage of the rule is significant). As a result, the chromosome that represents the rule set is given a low fitness and might not be selected to reproduce. However, these same conditions might turn out 'good' when they are

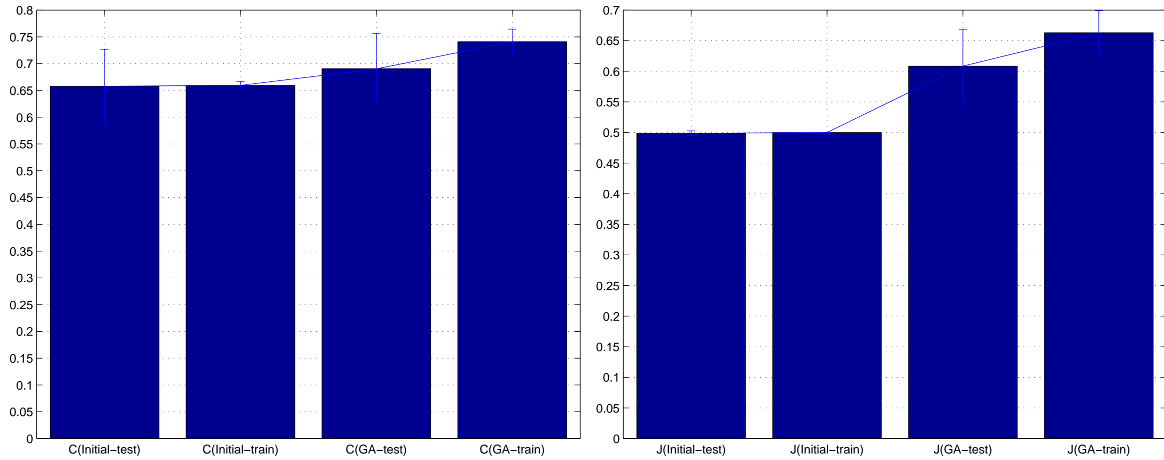

FIGURE 4.25. STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=roulette wheel and crossover=double-point crossover.

Parameter	Value	Parameter	Value
crossover probability	0.9	**Generations**	300
mutation rate	0.1	**Selection technique**	Rank Selection
Elitism	10%	**Crossover type**	Single point

TABLE 4.11. Experiment Setup III: Single Point Crossover and Rank Selection

combined with different ones forming different rules. In order to allow such rules (and rule sets) to survive, we repeated the experiments described above and replaced the roulette wheel selection technique with rank selection. Hence, this setup was similar to SETUP I except for the use of rank selection. Table 4.11 summarizes the parameters.

Figure 4.26 shows the accuracy and the J_index on **MAINT**. The GA achieved an accuracy of 87% on the training set and 59% on the testing set and a J_index of 85% on the training set and 54% on the testing set. The rule set constructed by C4.5 had an accuracy of 72% on the training set and 51% on the testing set and a J_index of 65% on the training set and 45% on the testing set. Hence, the GA outperformed C4.5 in both the accuracy and the J_index although it was not trained to optimize

the J_index. Compared to the GA in SETUP I (single point crossover and roulette wheel selection), performance was slightly better. This might be due to the fact that more conditions were kept in the pool with the rank selection technique. Since the data set was small in size, these conditions might be important in achieving a better accuracy.

On **STAB1** (Figure 4.27), the results were very similar to the previous two setups and the behavior of the GA is also very similar (converging to a classifier with one or two attributes only).

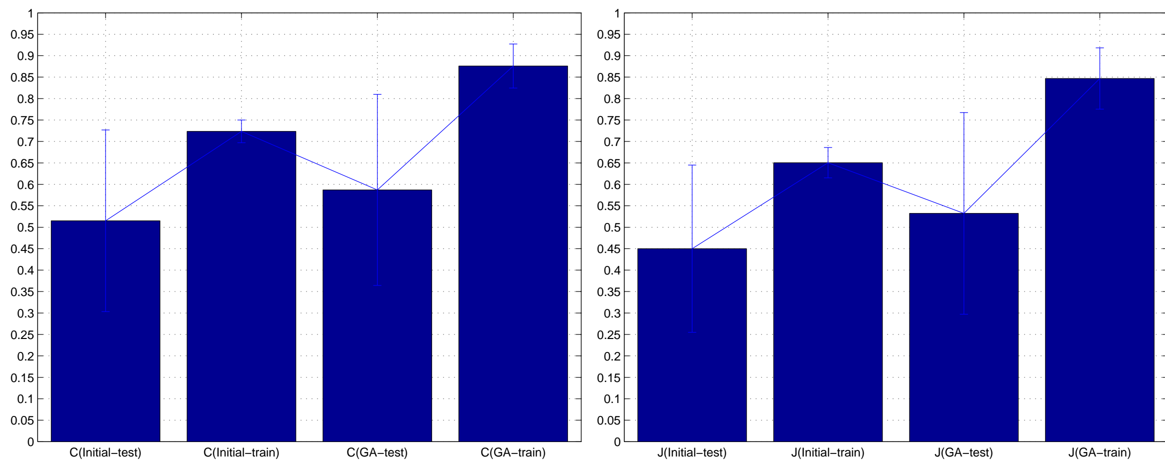

FIGURE 4.26. MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover.

On **STAB2** (Figure 4.28), the GA achieved an accuracy of 74% on the training set and 69% on the testing set. The J_index was 64% on the training set and 60% on the testing set.

We repeated the experiments seeding the GA with random rule sets. Figure 4.29 shows the results on **MAINT**. These were lower than the results obtained with SETUP I but better than the results obtained with the same setup (SETUP III) seeding the GA with C4.5.

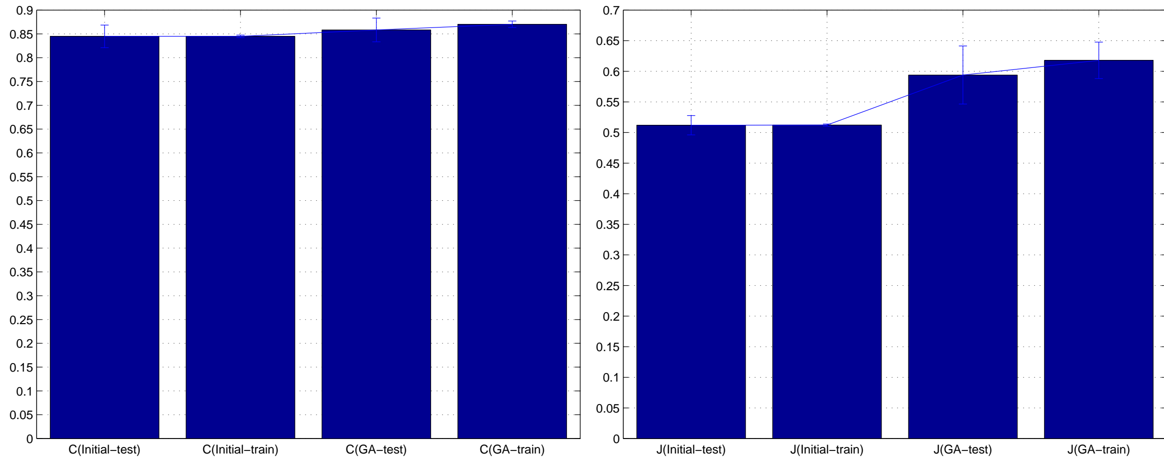

FIGURE 4.27. STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover.

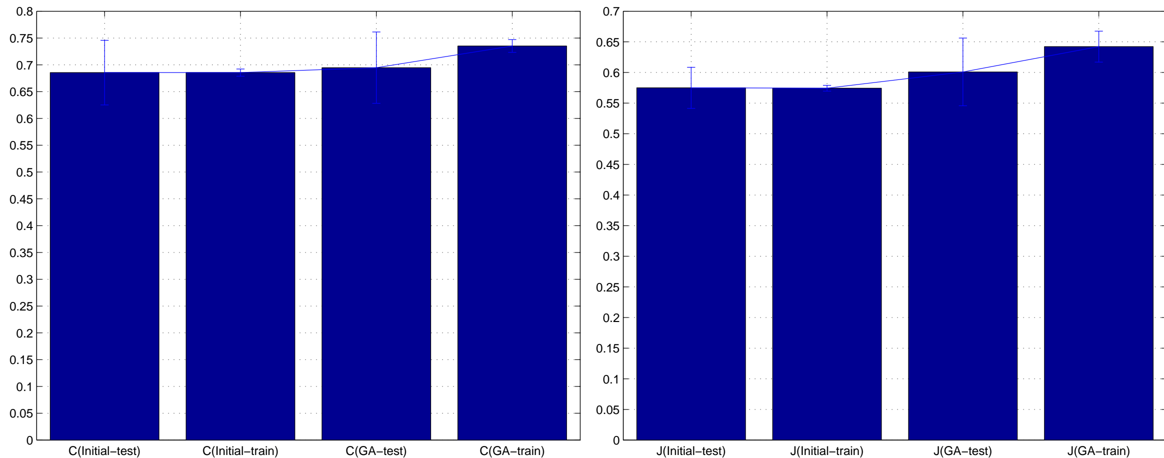

FIGURE 4.28. STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover.

On **STAB1**, the GA always behaves the same (Figure 4.30). On **STAB2** (Figure 4.31), the results were similar to those obtained in SETUP I. Rank selection seemed to delay the convergence of the GA when the number of attributes increased.

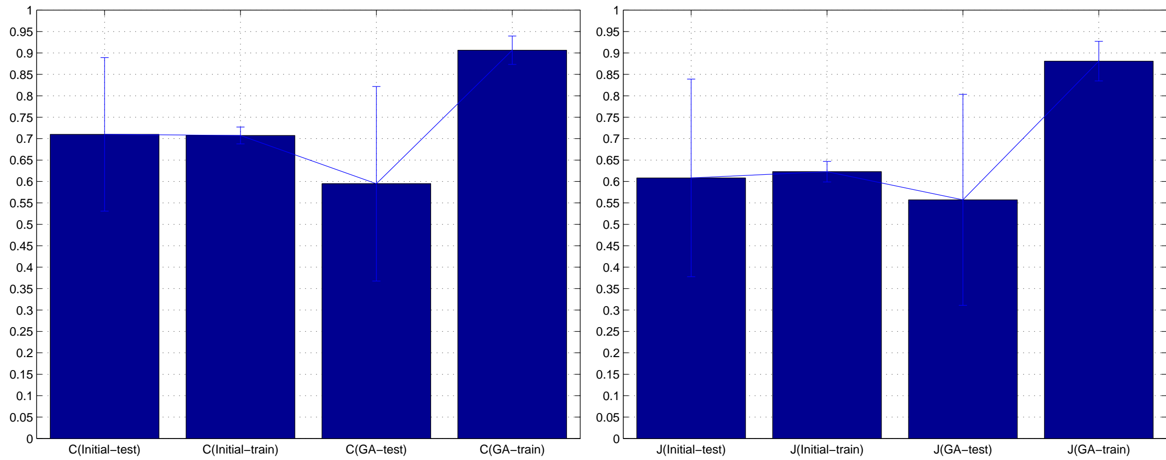

FIGURE 4.29. MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover.

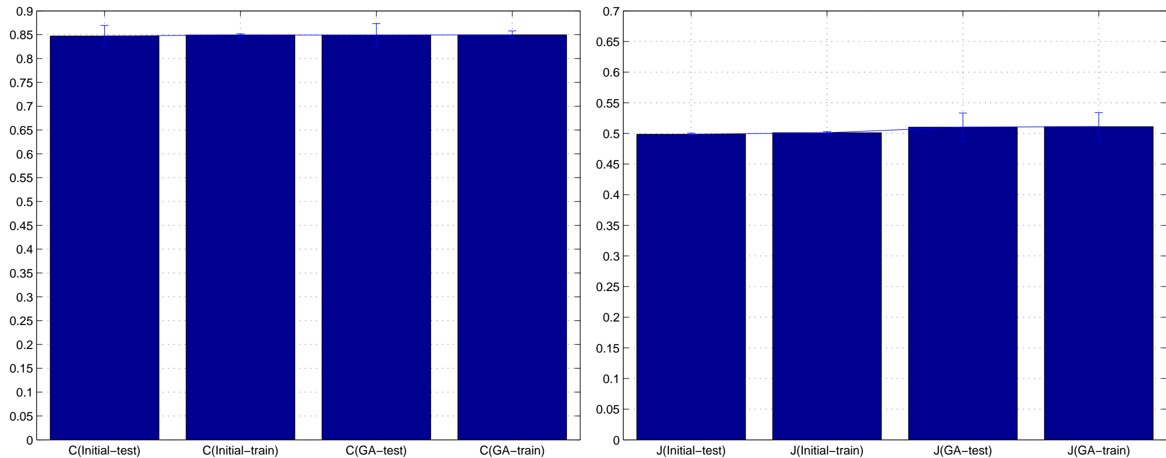

FIGURE 4.30. STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover.

4.5. SETUP IV: Double Point Crossover and Rank Selection. Also, to assess how the GA would perform with rank selection combined with double-point crossover, we set up the following experiments to run with the following parameters:

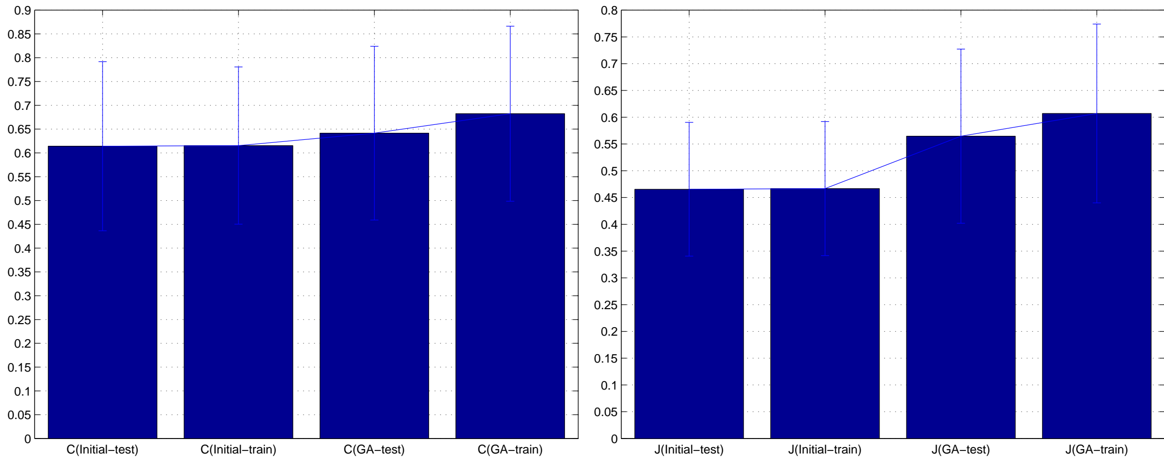

FIGURE 4.31. STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=single-point crossover .

Parameter	Value	Parameter	Value
Crossover probability	0.9	**Generations**	300
Mutation rate	0.1	**Selection technique**	Rank Selection
Elitism	10%	**Crossover type**	Double point

TABLE 4.12. Experiment Setup IV: Double Point Crossover and Rank Selection

crossover probability ($\aleph$) was 0.9, mutation rate (μ) was 0.1 (applied uniformly across a chromosome), the fitness of a chromosome was equal to the accuracy of the rule set that it represents ($f(R) = C(R)$). The percentage of chromosomes copied by elitism (ε) was set to 10. The number of generations (G) was set to 300, rank selection technique was used as well as double-point crossover. Table 4.12 summarizes these parameters.

Figures 4.32, 4.33, 4.34 show the results for all experiments run with this setup when the GA was seeded with C4.5 rule sets. On **MAINT**, the results were better than those obtained with roulette-wheel and double-point on both the training and the testing sets. Again, this might be due to the fact that more rules are surviving to the next generation and hence more combinations of conditions are possible. Compared

to SETUP III, it gave very similar results on the training set (slightly lower on the testing set) showing that the choice of double-point or single-point crossover did not affect the behavior of the GA (compared to single-point).

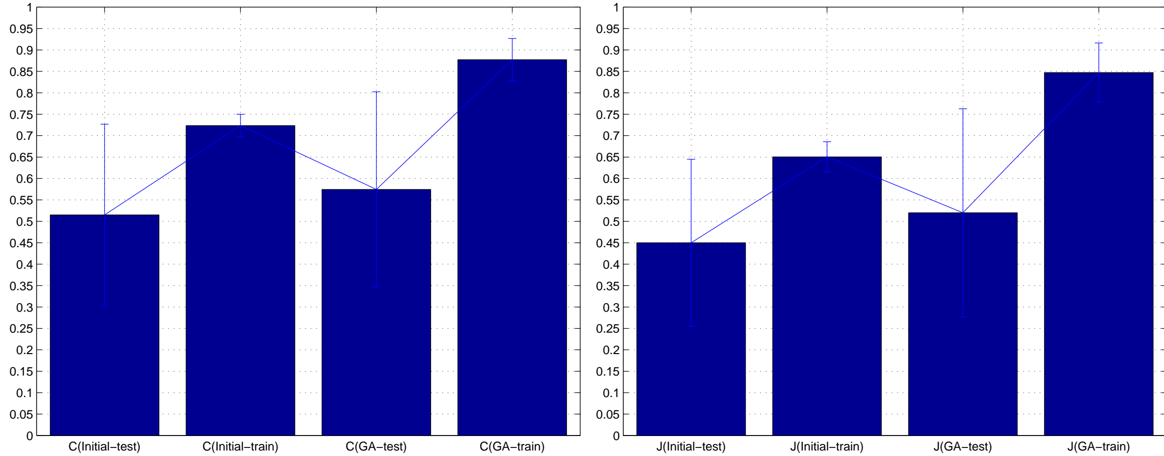

FIGURE 4.32. MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover.

STAB1 leads the GA to converge to the majority classifier (in most of the runs). On **STAB2**, the results are similar to those obtained with roulette wheel (SETUP II).

Figures 4.35, 4.36, 4.37 show the results of the experiments when the GA was seeded with random rule sets. Differences from SETUP II can be seen on **MAINT** (accuracy of 65% on the testing set compared to 61% in SETUP II and J_index equal to 60% compared to 56% in SETUP II).

4.6. Discussion and Summary. In Table 4.13, we summarize the results obtained with the GA with each of the three data sets **MAINT**, **STAB1** and **STAB2**. The first three rows show the results of the GA when seeded with C4.5 rule sets. The last three rows show the results of the GA when seeded with random rule sets. For each experiment, we record the accuracy measured on the training and the testing set and the J_index measured on both sets.

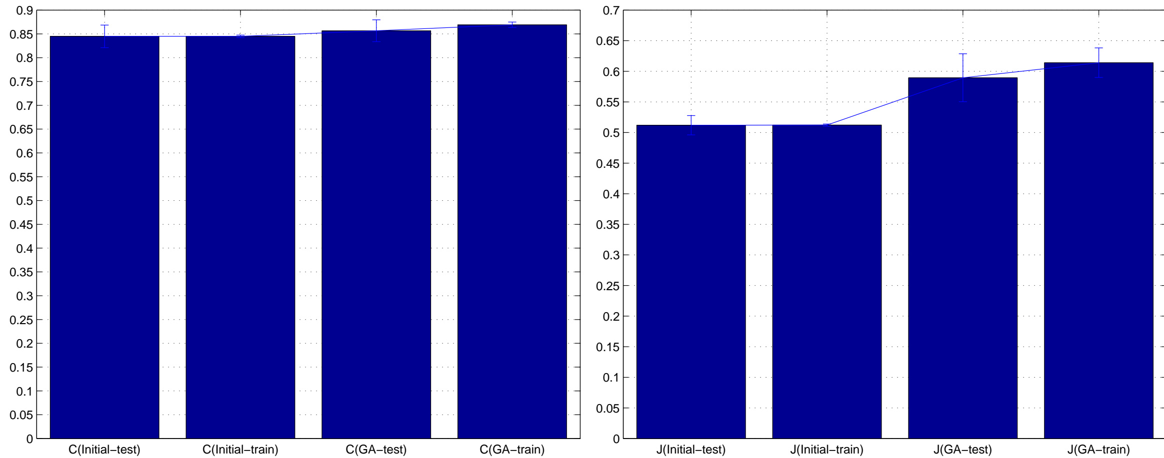

FIGURE 4.33. STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover.

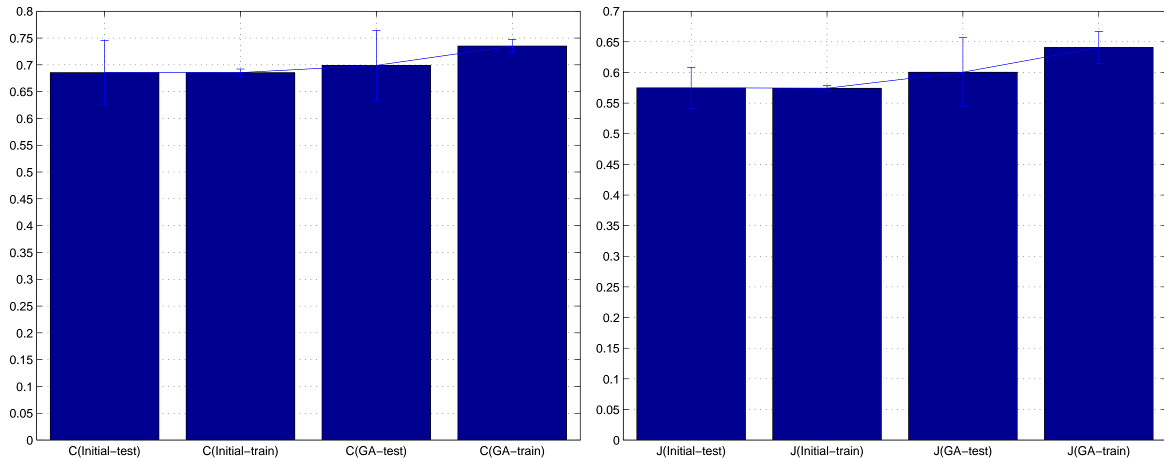

FIGURE 4.34. STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule sets on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover.

In summary, in the case of **MAINT**, in all four experimental setups, the GA resulted in an improvement in the accuracy measured on the training set while the previous GA (described in Section 1) was always converging to the best rule set

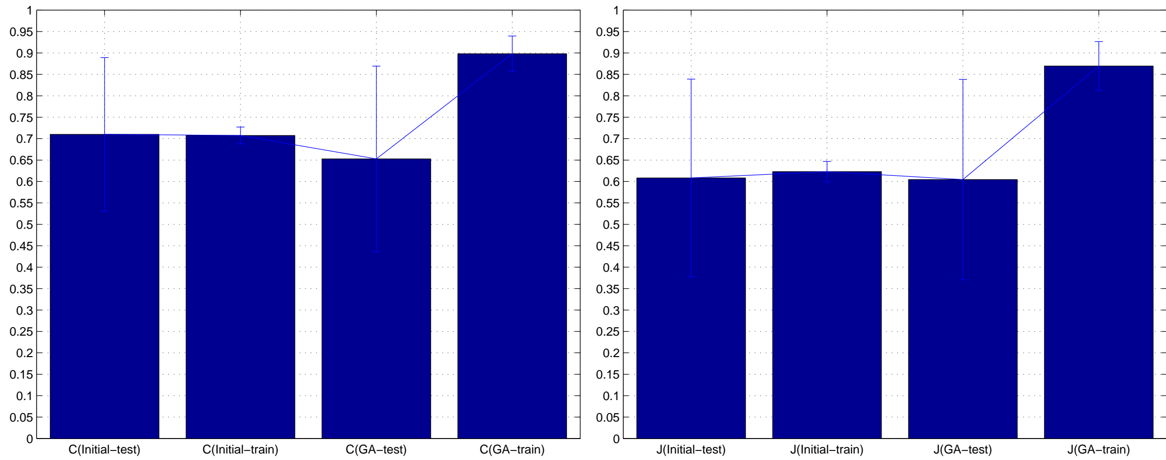

FIGURE 4.35. MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover.

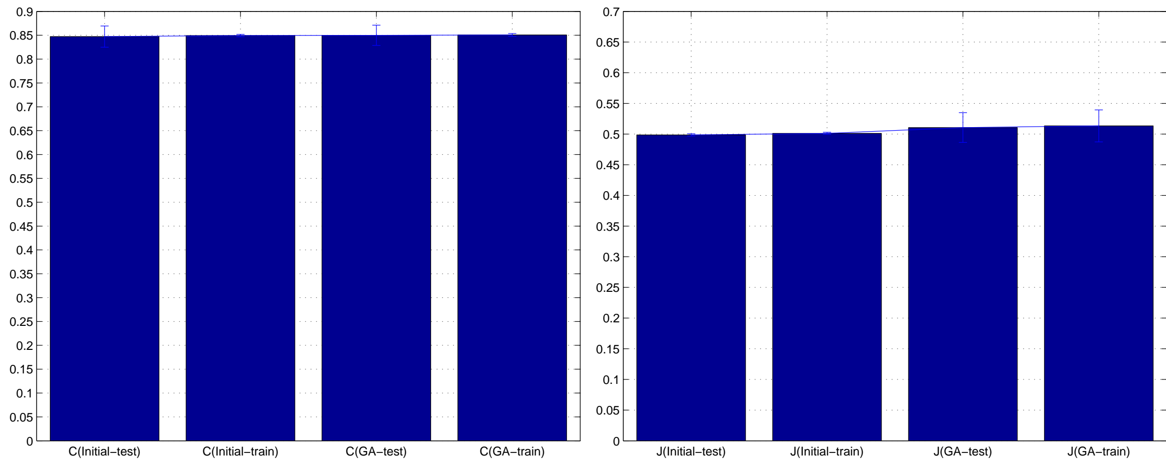

FIGURE 4.36. STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover.

already in the initial population. This shows that the granularity at which the operators were allowed to perform in the previous GA was indeed a problem. This also

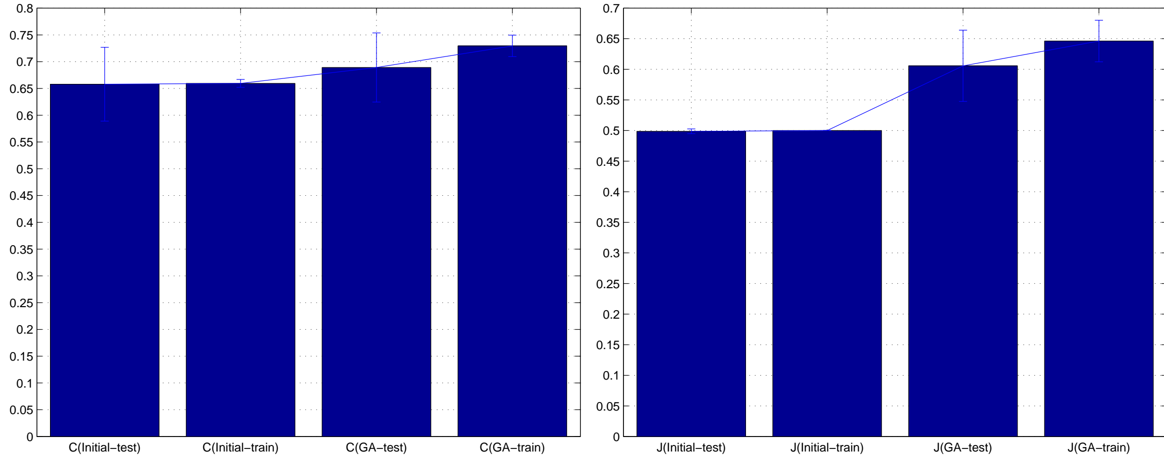

FIGURE 4.37. STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(R) = C(R)$, $\varepsilon = 10$, $G = 300$, selection technique=rank selection and crossover=double-point crossover.

	SETUP I				**SETUP II**				**SETUP III**				**SETUP IV**			
	c	**C**	**j**	**J**	**c**	**C**	**j**	**J**	**c**	**C**	**j**	**J**	**c**	**C**	**j**	**J**
MAINT	54	85	53	80.5	54	85	49	82	59	87	54	85	57	87	52	85
STAB1	85.5	86	59	60.5	85.5	86	57	59	85.5	86	59	61	85.5	86	59	62
STAB2	70	74.5	60.5	65	70	74	60	64	69	74	60	64	70	74	60	64
MAINT-R	61	94	55	91	61	92	56	90	60	90	55	88	65	90	60	86
STAB1-R	85	85.5	55.5	56	85.5	86	55	55.5	85	85	50.5	50.5	85	85	50.5	51
STAB2-R	69	73	60	65	69	74	60	66	64	68	56	61	69	74	60	65

TABLE 4.13. Summary of the results obtained with the GA in the four different setups. For each setup, values are recorded for the accuracy on the training set (C), the accuracy on the testing set (c)), the J_index on the training set (J) and the J_index on the testing set (j).

underlines the impact that the representation of a solution can have on the performance of the GA. Moreover, except for the case of the maintainability data with the roulette wheel selection technique, the GA could always improve on the accuracy on the testing set of the best rule set constructed by C4.5. We suspect that such an improvement did not happen on the maintainability data because of its small size.

On **STAB1**, the GA outperformed C4.5 and the majority classifier with respect to the J_index.

In the case of **STAB2**, the improvement was not as big as in the case of **MAINT** but was consistent in both the accuracy and the J_index not only on the training set but on the testing set, also.

The experiments presented in this section prove that the GA performs well on a balanced data set and at least as well as the majority classifier on an imbalanced data set. In Figure 4.38, we show an example of the learning curves for both SETUP I and SETUP III. The curves plot the best accuracy on the training set at every generation. As the figure shows, the convergence of the GA with the roulette-wheel is faster than with the rank selection. As a matter of fact, the GA finds a rule set with an accuracy of 73% around the 110th generation when roulette-wheel is used and around the 160th generation when rank selection is used. At the end of the experiment, the accuracy of the best rule set on the training is 74.5% in the first case and 73.5% in the second. These results are taken from experiments run when seeding the GA with C4.5 rule sets.

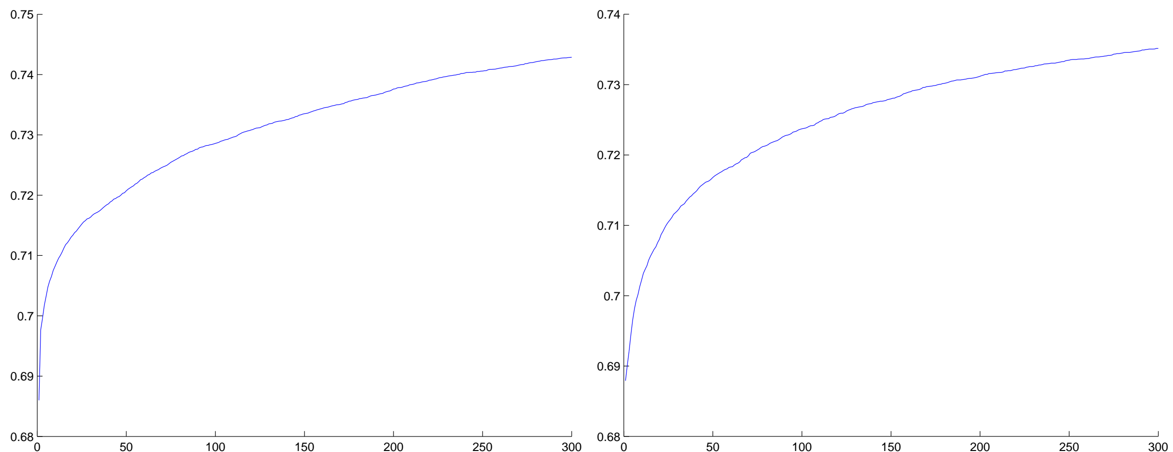

FIGURE 4.38. STAB2-Learning curves showing the accuracy on the training set for both the roulette-wheel (left hand side) and the rank-selection (right hand side) techniques.

CHAPTER 5

Second Approach: Optimizing a Single Model

1. Overview

In Chapter 4, we proposed a genetic algorithm to combine and adapt several software quality estimation models that take the form of rule sets to a new data set in order to obtain one or more models of the same form, with a better prediction accuracy. One question that comes to mind when considering software quality estimation models is how much can we do with one quality estimation model alone? In other words, suppose we have a software quality estimation model that has been constructed from a particular set of data, can we adapt it to a new set of data and use it to estimate the desired software quality without incurring a decrease in the rule set prediction accuracy? This is the question that motivates the work presented in this chapter.

2. The Algorithm

The genetic algorithm that we designed for this approach starts with a single quality estimation model that takes the form of a rule set. For this GA, a rule set represents a population of chromosomes, each rule being a chromosome. Each condition in a rule, as well as the classification label, are genes in the chromosomes.

Similar to the refined GA described in Chapter 4, Section 3, the genetic operators that we designed for this GA operate on the level of conditions and classification labels. Below, we describe the different design issues that were involved in this algorithm. Again, we focus on what we designed in a way specific to our problem and different from the two GAs presented in the previous chapter. We refer the reader to the appropriate literature when traditional methods were used.

2.1. Chromosomes and Fitness Function. Figure 5.1 shows a rule set that has three rules and a default classification label. It constitutes a population of three chromosomes. The default classification label is not included in the population but will be re-instated into the rule set at a later stage (explained later). The first and second chromosomes in this population have three genes, each - two conditions and one classification label. The third chromosome has two genes - one condition and a classification label.

```
A <= 10 ^ B <= 7 –> class 0
A > 3 ^ C <= 2  –> class  1
B <= 4 –>  class 0
Default class: 0
```

FIGURE 5.1. This is a rule set with three rules and a default classification label. This constitutes a population of three chromosomes. The first and the second one have three genes each and the third one has two.

It is important to point out that traditionally, chromosomes represent solutions to the optimization problem that the GA is tackling. As such, the 'goodness' of the solution is usually directly linked with the fitness function (for example, in the previous approach to the problem, we defined the fitness of the chromosomes to be equal to the accuracy of the underlying rule set). In this approach, this is not the case. Here, a chromosome represents a rule but a rule is not a solution (a rule set is). For this, we need to define what a 'good' rule is in the context of the rule set of which it is part. We define a **good** rule as one that has a high accuracy and, at the same time, classifies a large number of cases[1]. Since each chromosome in the population

[1] A rule that is highly accurate but barely matches any of the cases is not very useful.

encodes a rule, the fitness of a chromosome should reflect both the accuracy of the rule represented by the chromosome and its **coverage**, that is, the fraction of cases in the data set classified by the rule. With this in mind, we define two different fitness functions, f_1 and f_2, as shown in equations (5.1) and (5.2). In the equations, $C(l)$ is the accuracy of rule l, $t(l)$ is its coverage and $C(R)$ is the accuracy of rule set R containing rule l (all terms are formally defined in Chapter 2, Section 2).

$$f_1(l) = C(l) * t(l) \tag{5.1}$$

$$f_2(l) = C(l) * t(l) + (1 - t(l)) * C(R) \tag{5.2}$$

Note that both equations take into account the accuracy and the coverage of the rule because, as previously explained, this is what defines what a good rule is. Equation (5.2) takes into account the accuracy of the rule set as well. With this fitness function, rules with a coverage or accuracy of 0 get a higher chance to survive and produce progeny than with f_1. As we mentioned earlier, some conditions might be “fatal” in one context but perform very well when combined with other conditions. In Section 3, we compare the empirical behavior of these functions.

2.2. The Genetic Operators. To create a new population from the current one, chromosomes are selected either by roulette-wheel or by rank selection. New chromosomes are created by applying elitism, crossover and mutation. In this GA, we implemented these operators to work in a way very similar to the operators described in Chapter 4, Section 3. We highlight the small differences only.

We chose to do single-point crossover and this for two main reasons: the simplicity of the operator and the small size of the chromosomes in the initial population when the latter was derived from rule sets created by C4.5. As a matter of fact, the rules that C4.5 creates are small in size (average of 3 conditions per rule). This does not leave much room for double-point crossover which would have a greater effect when the chromosomes are longer. In this algorithm also, we allowed chromosomes

within a pair to be cut at different locations to allow a wider variety in the length of the chromosomes and more possible recombinations. In this GA, crossover implies a recombination of conditions in the rules that constitute the rule set[2]. Figures 5.2 and 5.3 show two examples of crossover. In the first one, the offspring have similar length as their parents. In the second one, the offspring have different lengths from their parents. The cut points are indicated with small arrows in the figures.

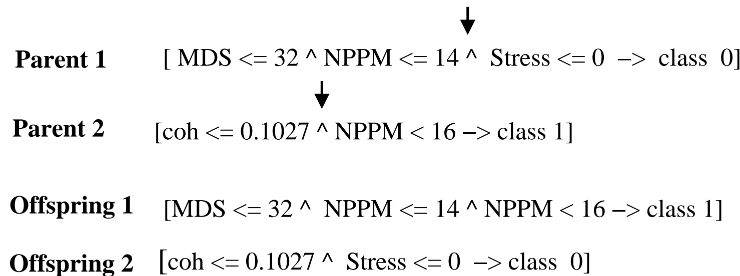

FIGURE 5.2. Crossover where cut points fall in different locations within a pair of chromosomes. Offspring have the same lengths as their parents.

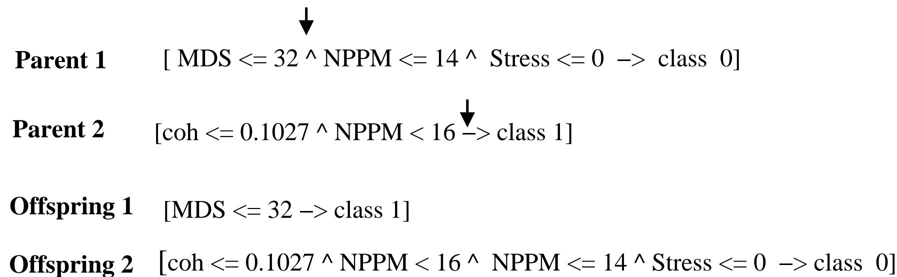

FIGURE 5.3. Crossover where cut points fall in different locations within a pair of chromosomes. Offspring have different lengths from their parents.

We designed the mutation operator to be similar to the one in the refined combining GA. Hence, mutation involves either the change of a class label or the value to

[2]In the refined GA, described in Chapter 4, Sectionsec:CombiningAdaptingRefined, crossover implied a recombination of conditions in rules and of rules in rule sets.

which an attribute is compared in a condition[3]. For example, mutating the first and the last genes in *offspring 2* in Figure 5.2 results in the rule shown in Figure 5.4.

Offspring before mutation [coh <= 0.1027 ^ Stress <= 0 –> class 0]

Offspring after mutation [coh <= 0.75 ^ Stress <= 0 –> class 1]

FIGURE 5.4. A chromosome before and after mutation. First and last genes are mutated.

2.3. Trimming. Here also, one can easily imagine that crossover and mutation, as we defined them above, might result in redundant and/or inconsistent rules (defined in Chapter 4, Section 3.3). In this case, redundancy happens inside a rule only. Figure 5.5 shows an example where crossover and mutation result in one inconsistent and one redundant rule. As a matter of fact, *offspring 1* in the figure, represents a rule where condition $MDS \leq 1$ implies condition $MDS \leq 3$ hence the rule is redundant. *Offspring 2* represents an inconsistent rule because it contains the conditions $NOM \leq 13$ and $NOM > 25$ which cannot be true at the same time.

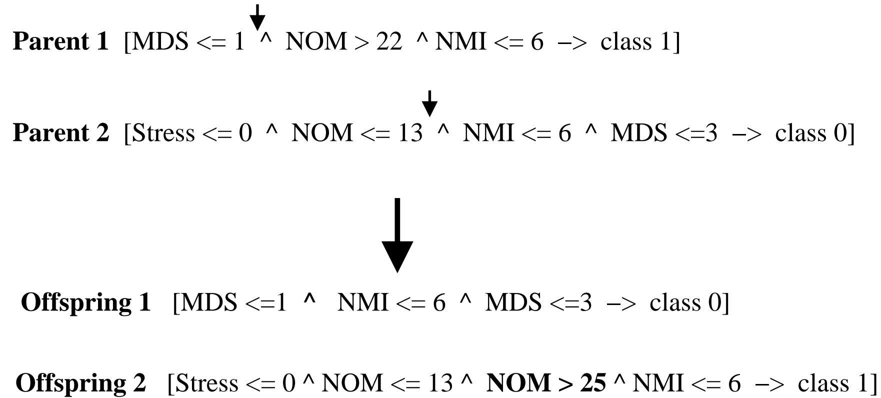

FIGURE 5.5. Crossover and mutation that result in a redundant rule (*offspring 1*) and an inconsistent rule (*offspring 2*). The condition in bold is the mutated gene.

[3]Unlike in the refined GA, mutation is not applied to the default classification label of the rule set because in this GA, a rule is a chromosome as opposed to a rule set being one in the refined GA.

A postprocessing step happens at two levels in the algorithm: between iterations, to get rid of redundancy, and at the end, when the last rule set is generated, to eliminate inconsistent rules. In order to eliminate redundancy, the offspring are **trimmed** before being thrown into the next generation. Trimming consists of getting rid of all redundant conditions in a rule. As such, conditions in the same rule are compared, and if a condition c_i implies a condition c_j, the latter is dropped. Figure 5.6 shows *offspring 1* in Figure 5.5 trimmed.

Offspring 1 before trimming [MDS <=1 ^ NMI <= 6 ^ MDS <=3 –> class 0]

Offspring 1 after trimming [MDS <=1 ^ NMI <= 6 –> class 0]

FIGURE 5.6. A chromosome before and after trimming. Condition $MDS <= 3$ is deleted because $MDS <= 1$ implies $MDS <= 3$.

The other postprocessing phase is after the last population is generated. At this point, the algorithm deletes all chromosomes with fitness equal to 0. Let us see what this means in the case of each of the fitness functions, f_1 and f_2 (defined in Equations 5.1 and 5.2, respectively) and which chromosomes get deleted by this process.

$f(l) = f_1(l) = C(l) * t(l)$:

$f(l) = 0 \Rightarrow C(l) = 0$ or $t(l) = 0$, i.e. the rule constantly misclassifies all the cases that it covers or does not classify any of the cases. With this fitness function, chromosomes representing inconsistent rules also get a fitness equal to 0 (their accuracy is 0).

$f(l) = f_2(l) = C(l) * t(l) + (1 - t(l)) * C(R)$:

Recall that $C(l)$, $t(l)$ and $C(R)$ are all greater than or equal to 0.

$f(l) = 0 \Rightarrow$

(i) $C(l) * t(l) = 0$, which is the same as the previous case; **and**

(ii) $(1 - t(l)) * C(R) = 0 \Rightarrow$

(a) $1 - t(l) = 0 \Rightarrow t(l) = 1$ $t(l) = 1$ and $C(l) = 0$ (from (i)); in this case, l is a rule that misclassifies all the cases in the data set.

or

(b) $C(R) = 0$; in this case the rule does not bring any 'useful' knowledge to R.

As we can see from above, chromosomes with fitness equal to 0 represent rules that are not needed for the classification process. Ideally, chromosomes with a fitness equal to 0 would die during the process of evolution since the selection procedure rarely allows them to be picked to survive or mate with other chromosomes. However, in some rare cases, preliminary experiments showed that such chromosomes could survive until the last generation. We did not design the GA to eliminate chromosomes with fitness 0 as soon as they appear because we do not want to impoverish the genetic pool. In the end, the algorithm prunes the last population and eliminates all chromosomes with fitness 0.

One phenomenon that we noticed when experimenting with our GA is overcrowding: early in the evolutionary process, the population consists of similar chromosomes of a relatively short size (involving one or two conditions only). We refer to this phenomenon as **overcrowding**. This does not leave much room for improvement. The algorithm deals with overcrowding by eliminating all duplicates at the end of each iteration. Figure 5.7 shows a rule set before and after pruning. As a result, and unlike what is most commonly the case in GAs, at iteration t, population $P(t + 1)$ might have size $n' \neq n$, where n is the size of $P(t)$.

2.4. From Rules to Rule Sets. At the end of each iteration, a rule set is formed out of the rules. For this, we designed the GA to complement the rules with a default classification label. This is chosen to be the majority classification label in the data set.

It is important to point out that the accuracy of the rule set is dependent on the order of the rules in it. Consider, for example, the following two rule sets that are

used to estimate whether a software component is fault-prone or not according to the metrics NOM and NOC:

$$
\begin{array}{l}
RuleSet\ 1 \\
R2 : NOM \leq 3 \wedge NOC \leq 2 \rightarrow class\ 1 \\
R3 : NOM \leq 4 \rightarrow class\ 0 \\
Default\ class :\ 0
\end{array}
$$

and

$$
\begin{array}{l}
RuleSet\ 2 \\
R3 : NOM \leq 4 \rightarrow class\ 0 \\
R2 : NOM \leq 3 \wedge NOC \leq 2 \rightarrow class\ 1 \\
Default\ class :\ 0
\end{array}
$$

They consist of the same rules. However, if we use them to classify a software component where $NOM = 2$ and $NOC = 1$, by applying the sequential classification procedure (where the top-most rule that fires on a case gives it its label) this component will be classified as fault-prone (class 1) by $Rule\ Set$ 1 and non fault-prone

Before Pruning

Rule 0
NOM <= 17.0 ^ Stress <=0.184 –> class 1

Rule 1
NOM <= 17.0 ^ Stress > 2.578 –> class 1

Rule 2
NOM <= 17.0 ^ Stress <= 0.184–> class 1

Rule 3
NPPM <= 2 –> class 0

Default class: 1

After Pruning

Rule 1
NOM <= 17.0 ^ Stress > 2.578 –> class 1

Rule 2
NOM <= 17.0 ^ Stress <= 0.184 –> class 1

Rule 3
NPPM <= 2 –> class0

Default class: 1

FIGURE 5.7. A rule set before after being pruned. Rule 0 and Rule 1 are deleted because the former has a duplicate in the rule set (Rule 3) and the chromosome representing Rule 1 has a fitness equal to 0.

(class 0) by *Rule Set* 2. The GA does not take into consideration this ordering and chromosomes generated first (copied by elitism or created by crossover, for example) are inserted at the top of the new generation. There are two ways to account for the importance of the order of rules. The first one is the voting scheme (described in Chapter 2, Section 4). This technique is computationally expensive since all rules in the rule set get to classify a case. The second way consists of sorting the chromosomes in a population and applying the sequential classification procedure. This technique gives priority to rules that appear first in the rule set and is computationally more efficient than the previous one. This leads to the question of the criteria by which the rules should be sorted. One choice would be to sort them by their fitnesses. However, this cannot be done with f_2 because in this case, the fitness depends on the accuracy of the rule set and the accuracy of the rule set depends on the order of the rules hence, their fitness. Another choice would be to sort them by the classification label (similar to C4.5). This has two advantages: the order of rules for one classification label does not matter, and the rule sets are easier to interpret by human experts [Quinlan, 1993]. In the experiments shown later in this chapter, we chose to sort the rules by their classification label in order to be consistent when we compare both fitness functions.

3. Experiments and Results

We tested our algorithm on the three different data sets, **STAB1**, **STAB2** and **MAINT**, described in Chapter 4, Section 4.1. We give a brief summary of them in Table 3. For each experiment, we start by describing the setup then we show and interpret results obtained with the three data sets. Here also, each experiment was repeated 30 times and the average over the 30 runs was reported for each. During each run, 10-fold cross-validation is used whereby the training set is divided into 10 subsets, the GA is trained on 9 and tested on the remaining one.

3.1. SETUP I. The experiments described here had the following parameters: Crossover probability ($\aleph$) was 0.9, mutation rate (μ) was 0.1 (applied uniformly

Set	Size	Data Distribution	Software Quality Assessed
STAB1	2920	imbalanced	stability
STAB2	690	balanced	stability
MAINT	83	balanced	maintainability

Table 5.1. Data sets used in experiments. The first two contain data about the stability of software components (classes) in an object-oriented system and the last contains data about maintainability.

Parameter	Value	Parameter	Value
crossover probability	0.9	**Generations**	50
mutation rate	0.1	**Selection technique**	Roulette Wheel
Elitism	0	$f(l)$	$C(l) * t(l)$

Table 5.2. Experiment SETUP I: $f(l) = C(l) * t(l)$.

throughout a chromosome), the fitness value was $f(l) = C(l) * t(l)$ (Equation 5.1), the elitism flag was de-activated ($\varepsilon = 0$), the number of generations (G) was set to 50 and roulette-wheel selection technique was used. When the algorithm evaluated the rule set, it complemented it with the majority class as the default classification label and the sequential classification procedure (with sorting the rules by the classification label) was applied. Table 5.2 summarizes these parameters.

3.1.1. *STAB1.* We ran 40 experiments seeding the GA with a different rule set in **STAB1**, each time. Figure 5.8 shows the average accuracy and J_index, respectively, of all the experiments[4]. As we can see, the GA could achieve an average accuracy of 85% on both the training and the testing sets whereas the initial rule set (constructed by C4.5) had an average accuracy of about 73% on both sets. However, it is important to recall that the data set in STAB1 is imbalanced. Hence, we compare the results to the majority classifier. As we can see in Figure 5.9, the GA gives results very close to the majority classifier. As a matter of fact, most of the times, the rule set that the GA converged to was, itself, the majority classifier.

[4]Each experiment was repeated 30 times.

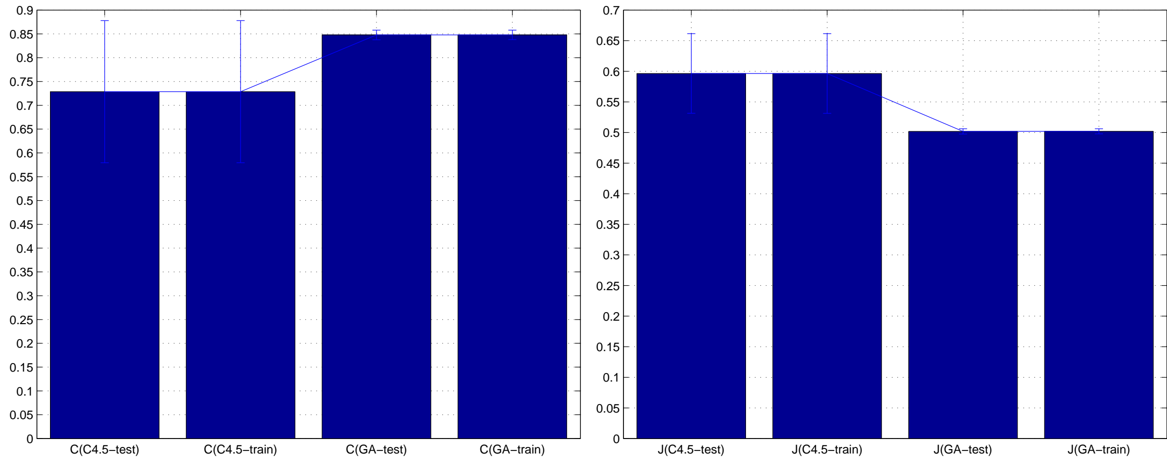

FIGURE 5.8. STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential.

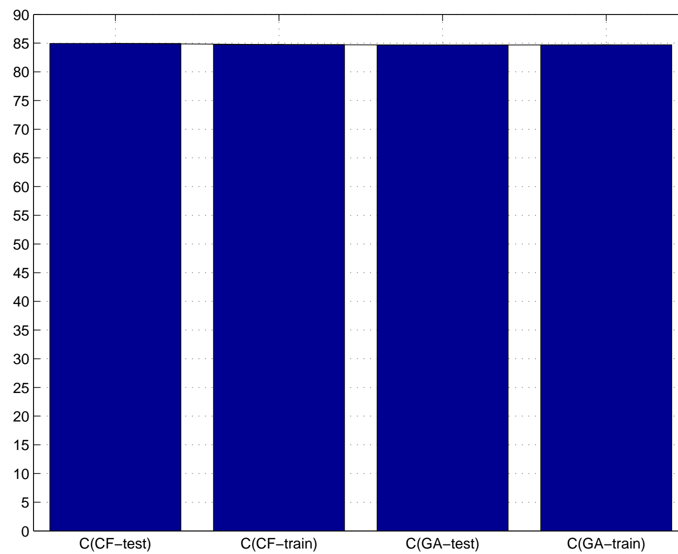

FIGURE 5.9. STAB1-Accuracy of the constant classifier (CF) and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential.

Figure 5.9 shows that the GA achieves better accuracy than C4.5 although the latter achieves a better J_index (not suprisingly, since the GA was trained to optimize the accuracy and not the J_index).

3.1.2. *STAB2.* The same experiments were repeated with **STAB2** to demonstrate the behavior of the GA on a balanced data set. Figure 5.10 shows the accuracy and J_index of the rule set generated by C4.5 and that generated by the GA on both the testing and the training sets. The figure shows that the rule set generated by the GA has an accuracy of 66% on the training set and 56% on the testing set whereas C4.5 rule set had an accuracy of 62% on the training set and 55% on the testing set. A small improvement in the J_index can also be seen on both sets.

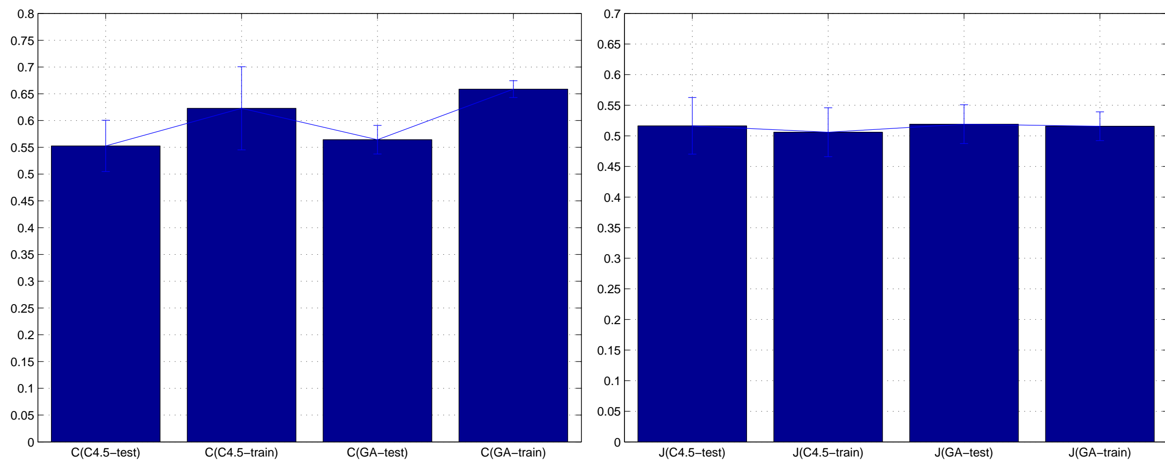

FIGURE 5.10. STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential.

3.1.3. *MAINT.* The same experiments were repeated on the data set **MAINT**. Results are shown in Figure 5.11. On this data set, also, the GA was able to improve the accuracy of the rule set on the training set only (68% versus 66% for C4.5). The J_index of the rule set obtained by the GA is lower than that of the rule set constructed by C4.5. Such a deterioration is not surprising since the J_index is the average per classification label and has a meaning for rule sets but not for rules (a rule can predict one classification label only). The fitness function does not take into consideration the average classification per class label.

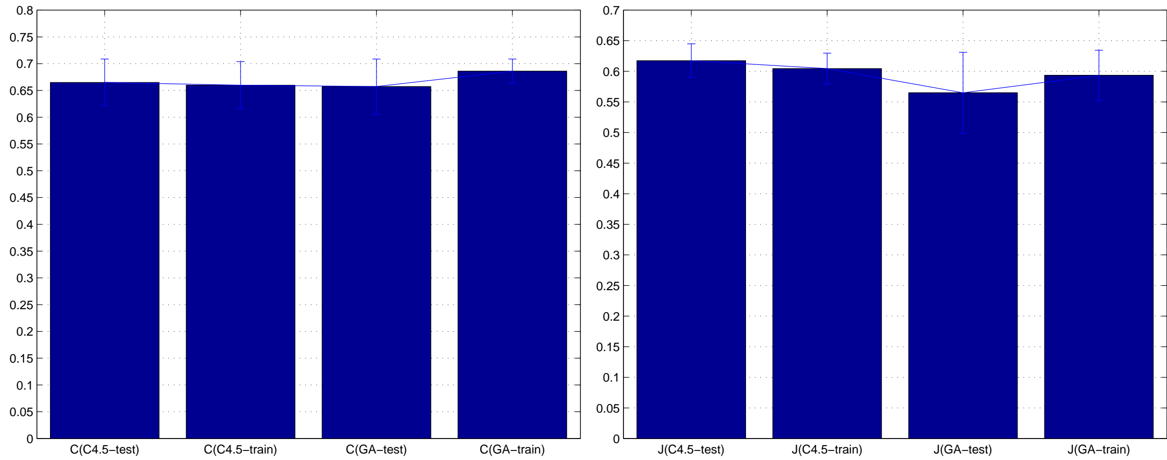

FIGURE 5.11. MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential.

3.1.4. *A Note on Results.* The experiments described above demonstrated the behavior of the GA on three data sets: a small one and two larger ones - one imbalanced and the other balanced. Results proved that our algorithm could improve the initial rule sets constructed by C4.5 on the three data sets. However, most of the time, the rule set was formed of one rule that included one attribute only. This rule has a higher accuracy than the longer rules in the initial rule set. On the one hand, this is appealing from a complexity point of view: The GA reduced the number of attributes in a rule set without compromising its accuracy, i.e., the end result was a rule set that is easier to interpret by human experts than C4.5 rule sets. On the other hand, the size of the rule set hinders the evolution beyond the best found accuracy (in case a better one is possible). When the rule set includes one attribute only, there is not enough room for improvement anymore. We suspect that this is due to the small size of the rule sets with which the GA was seeded. One can imagine a typical scenario where this can happen. Consider a population of three chromosomes encoding the three rules shown below:

$$R1: \quad NOM \leq 2 \rightarrow class\ 0 \qquad f = 88.81$$
$$R2: \quad NOC > 1 \wedge COM \leq 2 \rightarrow class\ 1 \qquad f = 91.95$$
$$R3: \quad NOC > 3 \rightarrow class\ 0 \qquad f = 90.95$$

Suppose that the two chromosomes selected for crossover are those encoding the last two rules, $R2$ and $R3$ and then $R3$ is selected by elitism to be copied to the next population. This results in the following three rules:

$$R4: \quad NOC > 1 \wedge NOC > 3 \rightarrow class\ 0$$
$$R5: \quad COM \leq 2 \rightarrow class\ 1$$
$$R6: \quad NOC > 3 \rightarrow class\ 0$$

At this point, $R4$ is trimmed and the three rules are now:

$$R4: \quad NOC > 3 \rightarrow class\ 0$$
$$R5: \quad COM \leq 2 \rightarrow class\ 1$$
$$R6: \quad NOC > 3 \rightarrow class\ 0$$

We can already see that, at this point, we already lost the attribute NOM, and NOC starts to overcrowd the population. The algorithm will prune such a population the only two chromosomes remaining are $R5$ and $R6$ as shown below:

$$R5: \quad COM \leq 2 \rightarrow class\ 1$$
$$R6: \quad NOC > 3 \rightarrow class\ 0$$

It is possible for $R6$ to breed with itself[5] and to be selected again to be copied to the next generation. This results in the following population formed of chromosomes that encode the same rule[6].

[5] It is always possible for a chromosome to breed with itself but as the size of the population shrinks, the possibility increases.

[6] Even if $R6$ does not breed with itself, the attribute pool is too small and no more combinations are possible.

$R7: \quad NOC > 3 \rightarrow class\ 0$

$R8: \quad NOC > 3 \rightarrow class\ 0$

$R9: \quad NOC > 3 \rightarrow calss\ 0$

Again, the algorithm eliminates all duplicates and the population becomes:

$R10: \quad NOC > 3 \rightarrow class\ 0$

One way to prevent the attribute pool from shrinking in size is to include the number of the attributes in the fitness function. By doing this, we can design the GA to penalize short chromosomes. However, this favors rules with more conditions and leads the GA into a search for longer rules, which defeats the purpose of simplicity in building quality estimation models with simple, easy to interpret rule sets. Another possibility is to generate rule sets from the entire set of attributes and use them to seed it. This also helps us test our GA on rule sets other than those built by C4.5. We explain this in more detail below.

3.1.5. *Randomly Generated Rule Sets.* The experiments described above assess the behavior of our GA when seeded with a rule set constructed by C4.5. These are rule sets that have already learned from common domain data and hence, have a prediction accuracy that is relatively high. In order to assess the behavior of the GA with any rule set, we generated random ones and we used them to seed the GA[7]. Figures 5.12, 5.13 and 5.14 show the results for the three data sets, **STAB1**, **STAB2** and **MAINT**.

The GA improved the accuracy significantly on **STAB1** by reaching one of 85% on the training set as well as the testing set. The initial rule set had an accuracy of 15% on both sets. The GA achieved an accuracy of 85%. No deterioration of the J_index was perceived. On **STAB2**, the GA could obtain a rule set with accuracy equal to 65% on both sets where the initial rule set had an accuracy of 53% hence, resulting in an improvement not only on the training set but on the testing set as

[7]For a description of how the rule sets were generated, the reader can refer to Chapter 4, Section 4.2.

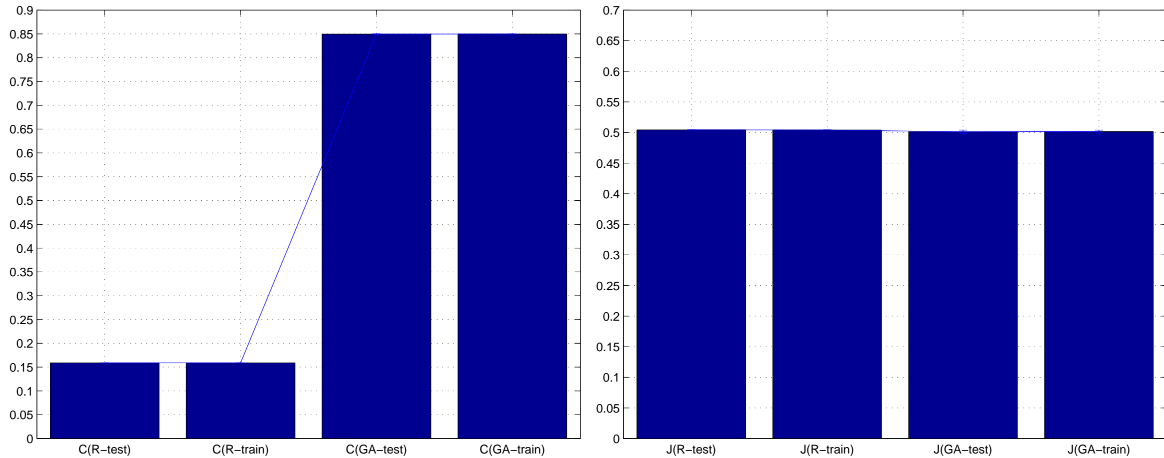

FIGURE 5.12. STAB1-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential.

well. The J_index also improved on both sets and the GA achieved 52% while the random rule set had a J_index around 44%. The accuracy obtained by the GA is higher than that of C4.5 on the same data set (55% on the testing set and 62% on the training set). On **MAINT**, the GA reached an accuracy of 65% on the testing set and slightly more on the training set, hence improving on the accuracy of the initial rule set (50%) and reaching the accuracy of C4.5 on **MAINT**. The J_index, however, deteriorated and reached 50% whereas the initial random rule set had one equal to 50% on the testing set and 60% on the training set.

It is also important to compare the results obtained with the GA seeded with random rule sets to those constructed by C4.5. On **STAB1**, the GA almost always converged to the majority classifier (whether it was seeded with a random rule set or one constructed by C4.5) and hence outperformed C4.5 with respect to the accuracy but scores lower than it with respect to the J_index. On **STAB2**, the GA outperformed C4.5.

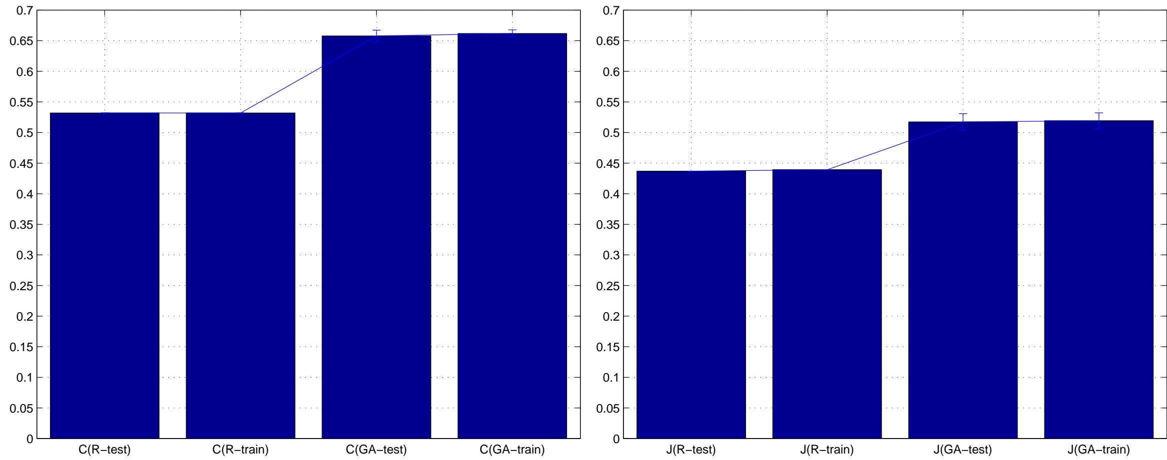

FIGURE 5.13. STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential.

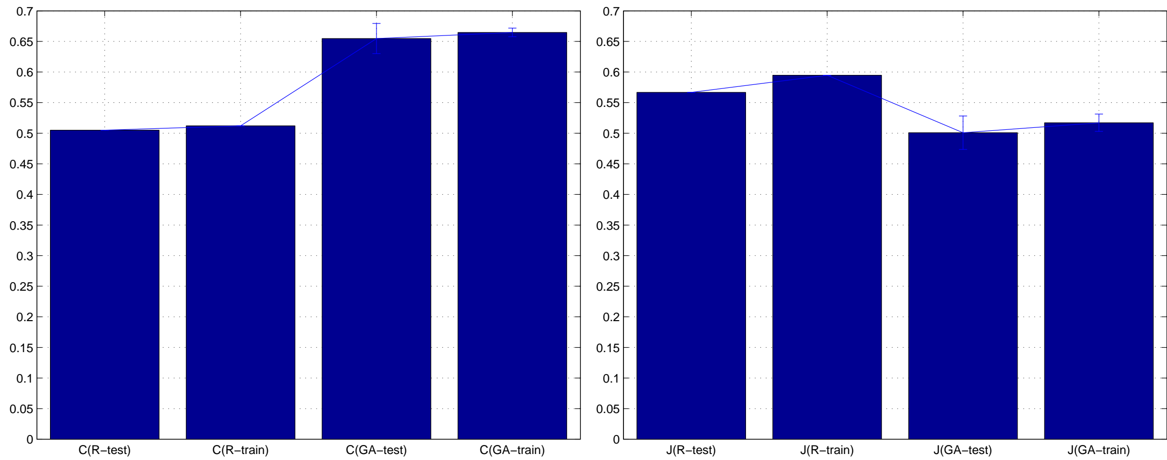

FIGURE 5.14. MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label and classification procedure is sequential.

3.2. SETUP II. In order to compare both fitness functions f_1 and f_2 (Equations 5.1 and 5.2 respectively), we set up another set of experiments that differ from

Parameter	**Value**	**Parameter**	**Value**
crossover probability	0.9	**Generations**	50
mutation rate	0.1	**Selection technique**	Roulette Wheel
Elitism	0	$f(l)$	$C(l) * t(l) + (1 - t(l)) * C(R)$

TABLE 5.3. Experiment SETUP II: $f(l) = C(l) * t(l) + (1 - t(l)) * C(R)$.

the previous one in the fitness function only. In this set of experiments, the fitness function shown is $f_2(l) = C(l) * t(l) + (1 - t(l)) * C(R)$ (Equation 5.2). Everything else is the same as in **SETUP 1**. Table 5.3 summarizes this setup.

3.2.1. *STAB1.* Figure 5.15 shows the accuracy and the J_index of C4.5 and the GA on the testing and the training sets. The GA achieved an accuracy of 85% on both the training set and the testing set whereas C4.5 achieved an accuracy of 72%. It also achieved a J_index of 52% on both sets slightly better than the J_index of the majority classifier (J_index of C4.5 was close to 60% on both sets).

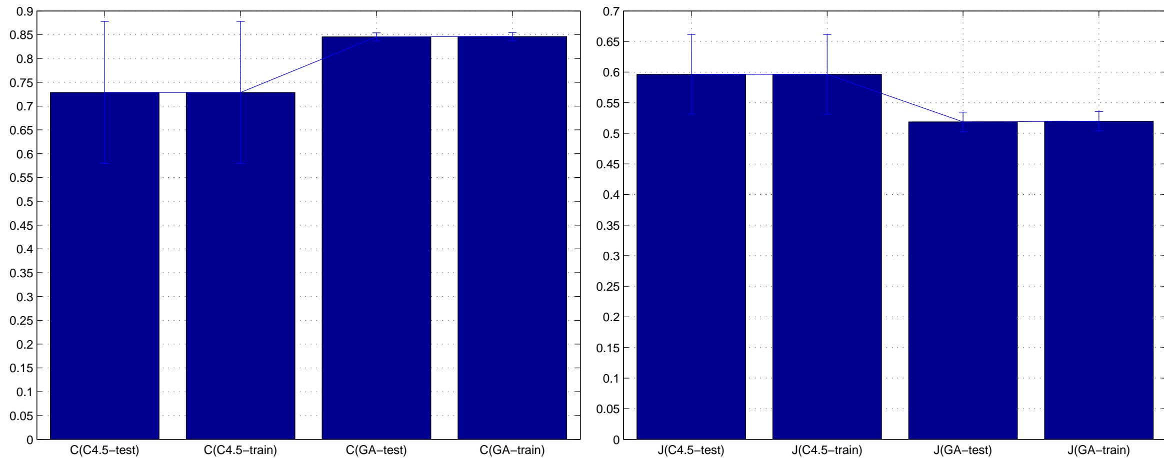

FIGURE 5.15. STAB1-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential.

3.2.2. *STAB2.* Figure 5.16 shows the accuracy and the J_index of C4.5 and the GA on the testing and the training sets in the case of the balanced data set **STAB2**.

Here, also, the GA improved the accuracy on both sets (slightly only on the testing set) and a small improvement can be seen in the J_index. The results are similar to those obtained with the previous setup.

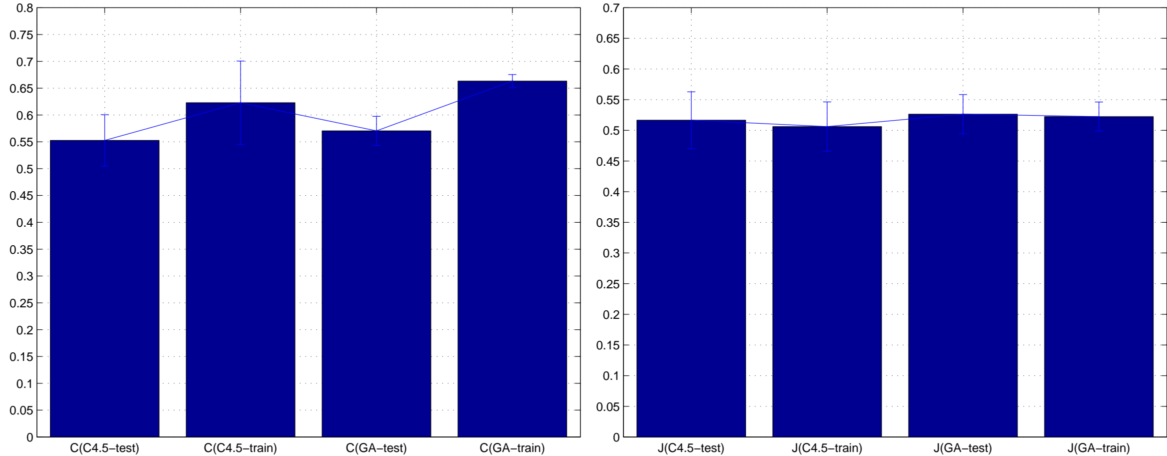

FIGURE 5.16. STAB2-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential.

3.2.3. *MAINT.* Figure 5.17 shows the results of the GA on the maintainability data set. Compared to the previous setup, SETUP II achieved similar results. Hence, accounting for the accuracy of the rule set in the fitness function did not affect the performance of the GA when seeded with a rule set constructed by C4.5.

3.2.4. *Randomly Generated Rule Sets.* In the same setup, we performed experiments seeding the GA with a random rule set. These were the same rule sets that we tested the algorithm with in the previous setup (Section 3.1.5). Figures 5.18 show the results on **STAB1** and Figure 5.19 shows the results of these experiments on **STAB2**. Again, on **STAB1**, the GA converged to the majority classifier and scored the same accuracy and J_index as it did in the previous setup.

On **STAB2**, the GA improved the accuracy on the training set and the testing set and especially the J_index. The improvement was slightly better than the one

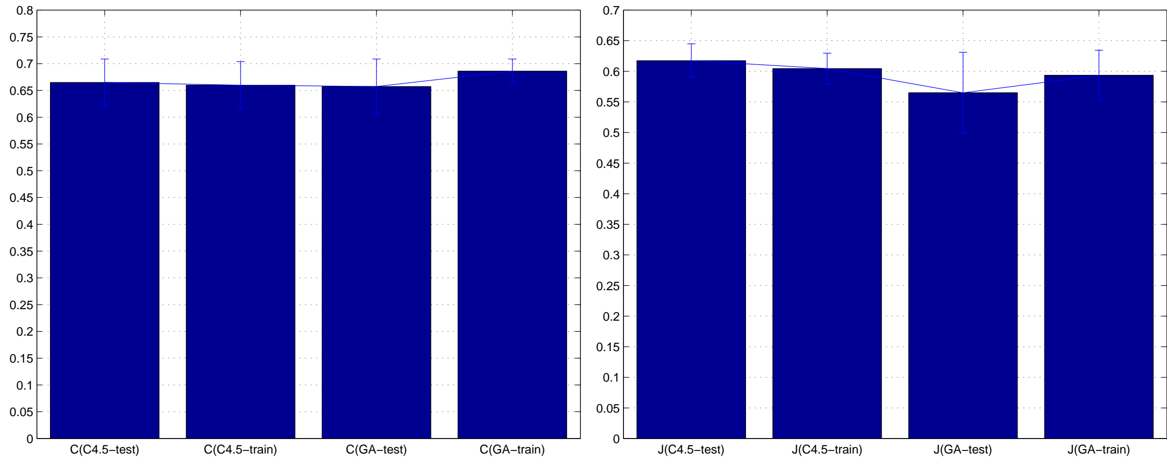

FIGURE 5.17. MAINT-Accuracy (C) and J_index (J) of C4.5 and GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential.

obtained in SETUP I. It is important to point out that the final rule sets obtained with the GA have an accuracy higher than the rule sets that C4.5 constructed on the same data set. Hence, the GA was able to adapt a randomly generated rule set to **STAB2** and achieve an accuracy (68% on training, 67% on testing) and a J_index (about 55% on both sets) higher than the rule set that C4.5 constructed (accuracy on training equal to 62%, accuracy on testing equal to 55%, J_index on training and testing = 52%).

Figure 5.20 shows the results obtained on **MAINT**. In this setup also the GA improved the accuracy and the J_index. The rule set obtained had an accuracy of 72% on the training set and 65% on the testing set (compared to 51% and 50%, respectively, in the case of the initial rule set) and a J_index of 63% on the training set and 56% on the testing set (compared to 60% and 56%, respectively, for the initial rule set).

3.3. Comparison of SETUP I and SETUP II. When seeded with a rule set constructed by C4.5, the GA performed similarly in both setups regardless of the

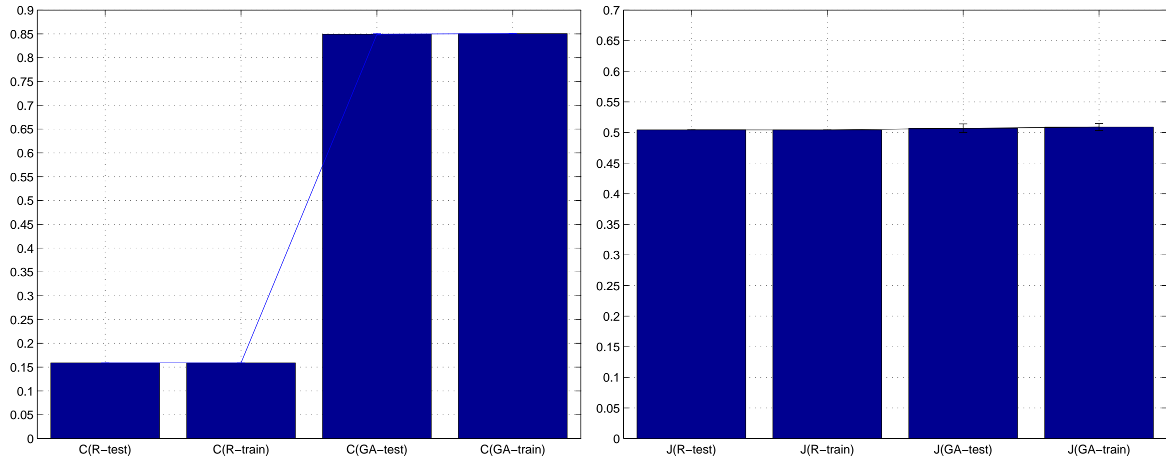

FIGURE 5.18. STAB1- Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential.

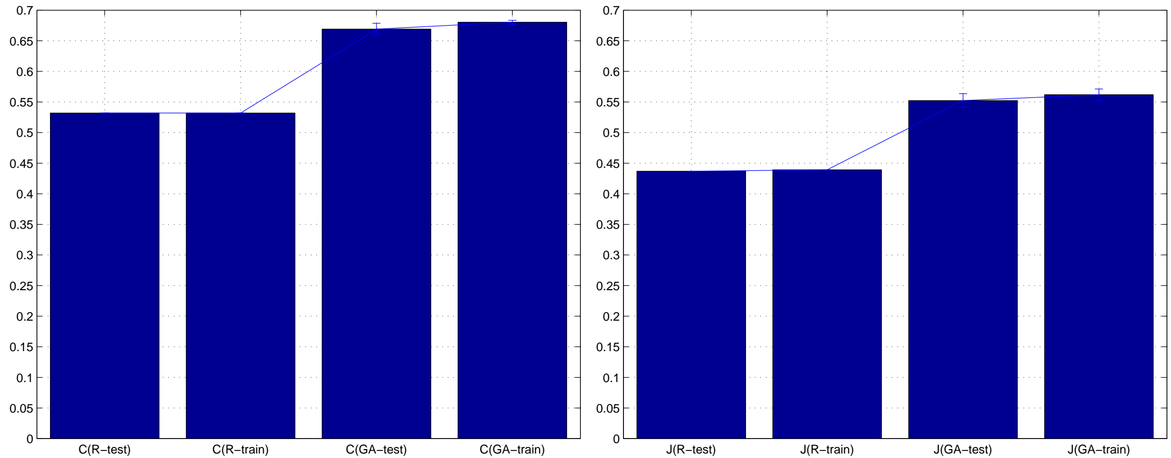

FIGURE 5.19. STAB2-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets.Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential.

data set that we used. However, when the GA was seeded with a random rule set, it gave a higher accuracy and a higher J_index in SETUP II than it did in SETUP I on

the balanced data sets **STAB2** and **MAINT**. In our opinion, the reason is two-fold: the size of the rule set and the fitness function. The rule sets constructed by C4.5 are small in size (small number of attributes and small number of rules). During the evolutionary process, some rules get a very low fitness and eventually die (are never picked to produce progeny). This depletes the attribute pool and the GA converges to a rule set with one or two attributes only (in most of the cases, 1 attribute). Not much improvement ca be expected beyond this point. However, when we generate random rule sets, we pick attributes from the whole set describing the data set. Including all (or most of) the attributes in the initial rule set gives the GA more possibilities for constructing rules (and conditions). This prevents its premature convergence to a small rule set and allows the process of evolution to proceed longer.

Why does the GA perform better with f_2 than with f_1 when seeded with a random rule set? The reason is that f_2 includes the accuracy of the rule set in the fitness of a chromosome. Hence, a chromosome representing a rule with an accuracy or a coverage of 0 will not be given a fitness of 0 and has a higher chance to survive (in the case of f_1 such a chromosome will have a fitness equal to 0). As already mentioned, some conditions, although "fatal" when they appear in a combination might turn out good when combined with other conditions. f_2 gives a chance for such conditions to survive and be combined with other conditions throughout the process of evolution. f_2 does not have the same influence when used with C4.5 rule sets because the number of conditions is already very small. It is important to point out that when seeded with random rule sets, our GA still constructs a rule set with fewer attributes than the original one and in most cases, the number of attributes does not exceed 3. However, such a convergence happens at a later stage in the evolutionary process than it does when the GA is seeded with C4.5 rule sets.

3.4. The Effect of Crossover and Mutation Rates. In order to show the effect of different crossover and mutation rates on the performance of the GA, we chose one rule set constructed by C4.5 and we seeded the GA with it. We ran several experiments. They differ from each other in the crossover and/or the mutation

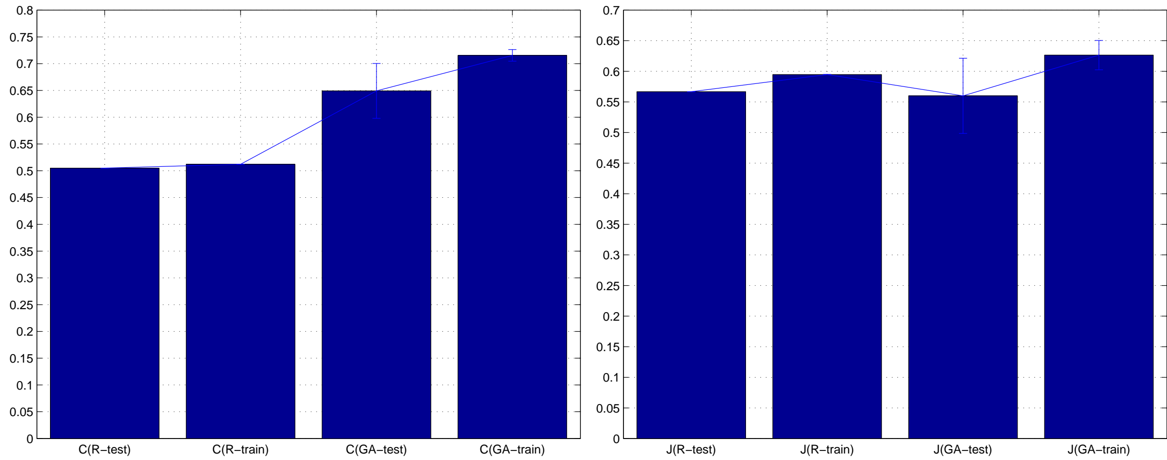

FIGURE 5.20. MAINT-Accuracy (C) and J_index (J) of a random rule set and the GA generated rule set on both the testing and the training sets. Experiments were run with the following parameters: $\aleph = 0.9$, $\mu = 0.1$, $f(l) = c(l) * t(l) + (1 - t(l)) * C(R)$, $\varepsilon = 0$, $G = 50$, selection technique is roulette wheel, sorting of rules in the rule set is by classification label classification procedure is sequential.

rate. In all experiments, the number of generations was set to 1000 and elitism was performed. The fitness function used was f_1 (Equation 5.1). Each experiment was repeated 30 times and the average over the 30 runs was reported. Figures 5.21 and 5.22 show the results (accuracy and J_index) on both the training and the testing sets. Each bar in the graph corresponds to one experiment (average over 30 runs). The first bar in each figure indicates the accuracy (or the J_index) of the initial rule set used to seed all the experiments in this figure. The figures show that changing crossover and mutation rates affect somehow the results (not much the training accuracy). Usually, crossover and mutation rates are set after several experiments are run and the best set of parameters for the particular problem is chosen.

4. Discussion and Summary

The GA outperforms C4.5 in both the complexity and the accuracy of the rule sets that it builds especially on **STAB1** and **STAB2**. As a matter of fact, not only does the rule set obtained with our GA have a higher accuracy (on both the testing

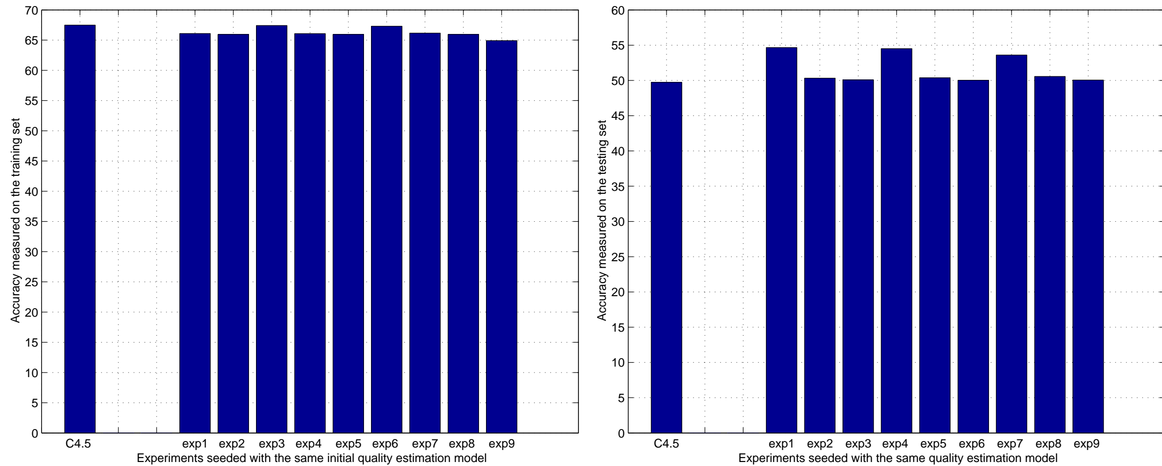

FIGURE 5.21. STAB2-Different experiments seeded with the same rule set. Each bar shows the results of one experiment. Each experiment is run 30 times and the average and standard deviation are reported for each. The first bar shows the accuracy of the initial rule set.

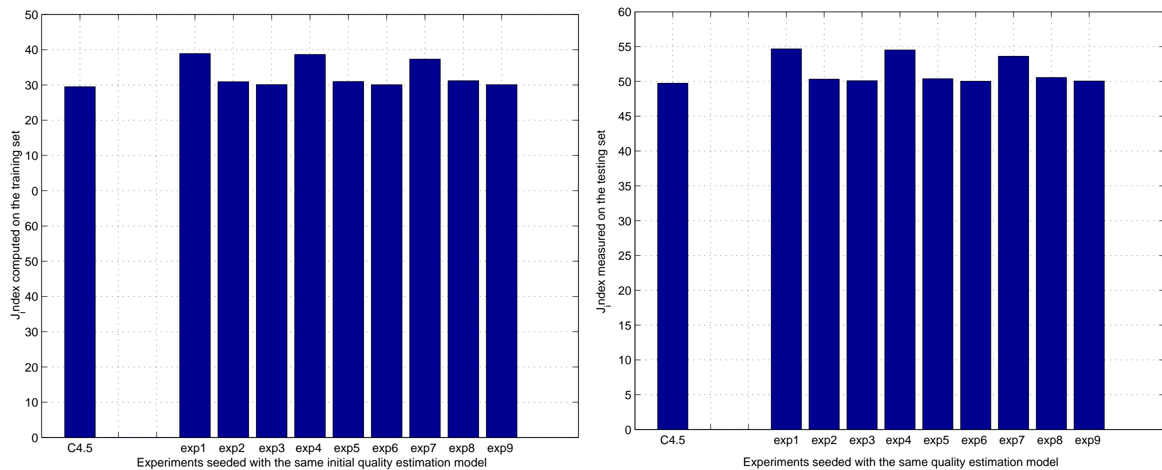

FIGURE 5.22. STAB2-Different experiments seeded with the same rule set. Each bar shows the results of one experiment. Each experiment is run 30 times and the average and standard deviation are reported for each. The first bar shows the J_index of the initial rule set.

and the training sets) than the one built by C4.5 but also this rule set has fewer attributes and conditions. Hence, it is much easier to understand by human experts. Feature reduction is a commonly studied problem in the field of classification systems and a well-desired feature in building software quality estimation models. Another

important feature of the GA is the speed with which it finds the "best" rule set. This is obtained early during the evolution process (in most cases, within the first 10 to 15 generations when the seed is a C4.5 rule set, at a later iteration (35 and later) when the seed is a random rule set). This is interesting since in most applications, GAs suffer from the speed in discovering a "good" solution which is not the case with ours.

CHAPTER 6

Conclusion

1. Summary

This thesis has proposed two adaptive approaches to optimize existing software quality estimation models. The first one starts with several rule sets and searches for a combination of rules, derived from them, that makes better rule sets with respect to a new data set. In real applications, this can be seen as taking rule sets built from common domain knowledge and finding a combination of their "expertise" that results in a rule set with a higher prediction accuracy when used on context specific data. The second approach adapts a single rule set, built from a specific data set, to a new one. In real applications, this can be seen as taking one rule set built from common domain knowledge and adapting it to context specific knowledge. In both approaches, the knowledge learnt at the time the initial rule sets were built is incorporated in the final one. To the best of our knowledge, ours is the first work that aims at optimizing the performance of already-existing software quality estimation models on new, unseen data.

Both approaches that we have presented are derived from the genetic algorithm methodology. Our experience with this technique proved that the choice of the representation of the models in the GA affected to a great extent the performance of the algorithm. As a matter of fact, it is widely recognized that setting up a problem to be solved by GAs is not straightforward.

We have conducted experiments with three different data sets. Two involved the stability of classes in an object-oriented system and one involved maintainability of C++ classes. One of the two stability data sets was imbalanced. The maintainability data set had a smaller size than the other two. In the first approach (combining multiple estimation models) the GA outperformed C4.5 by achieving higher accuracy and J_index on all three data sets. In the second approach (adapting a single model), the GA achieved a better J_index (on both, the training and the testing sets) than the majority classifier on the imbalanced data set. It achieved higher accuracy and J_index than C4.5 on the balanced stability data set. On the maintainability data set, it achieved a better accuracy on the training set only (due to the small size of the data set).

We have also experimented with randomly generated rule sets and our technique outperformed random guess on the two larger data sets. It also outperformed the majority classifier on the imbalanced data set.

The end result of our technique consists of rule sets. These provide two separate utilities: an estimation of a quality characteristic and guidelines that can help software engineers attain the predicted quality characteristic.

Our technique guarantees no damage to results. In all cases, the GA finds a software quality estimation model that is at least as accurate as the initial one. In most cases, the rule set found by the GA is different from the initial one and hence, can be used as an additional guideline during the development phase of a software product. In the experiments that we presented, the adaptive GA found rule sets that not only are more accurate but also have fewer attributes in the rules. Rule sets with fewer attributes are easier to interpret by human experts. We must admit, however, that the reduction in the complexity of the rule set was a "by-product" of the GA that we were not aiming at achieving when we designed the algorithm. Nonetheless, in our design, we made sure not to guide the GA, in its search, to more complex rule sets.

Our technique is general and not tied to any specific type of rules or type of attributes (i.e. the GA can perform well with continuous and discrete attributes and with rules other than attribute-value rules). Although it has been validated with object-oriented metrics and data, our technique should also be applicable to other types of software systems since it is independent of the metrics or the type of data.

2. Limitations

The running time of the GA is polynomial in the size of the data set, the number of rule sets in each population, the size of rule sets (number of rules per rule set), the size of rules (number of conditions per rule) and the number of generations. For example, the algorithm implementing the combining approach executes in approximately 23 minutes in a run over 300 generations on STAB2, with a population of 23 rule sets, a total of 90 rules or 181 conditions. When we benchmarked the algorithm, we found that most of the time is spent evaluating the fitness function (which is usually the case in GAs). The execution time is not too bad regarding the benefits that an optimized model can bring. Nonetheless, one way to optimize this running time would be by parallelizing the evaluation of the fitness of the chromosomes in one population.

The results obtained with the maintainability data set indicate that a set with a decent size is still needed in the adaptive approach that considers a single model.

Another limitation of this technique is that neither of the approaches that we presented considers the gain ratio associated with each attribute (defined in Chapter 2, Page 5) whereby the algorithm chooses the attribute test that seems most promising in terms of dividing the training set. We think that incorporating such knowledge into the GA will improve the results significantly.

3. Future Work

There are a few paths that derive from this work and are worth exploring. One path that we have already embarked on is the design of a genetic algorithm that is based on a different representation of the rule sets. More precisely, we have started

laying the ground for a GA that represents rule sets as graphs. Genetic operators (*crossover* and *mutation*) are defined to be simple graph operations (changing edge weights, drawing edges between vertices, etc.). This new design differs from the ones we have seen in this thesis in two major ways: 1. simplicity of implementation and 2. strong effect of *mutation* and *crossover* on the rules and hence, the rule sets. Its strength, as we expect it, lies in the capability of the GA to add attributes to rule sets during the process of adaptation. This is important as it is often the case in the field of software quality for new metrics to be proposed. Incorporating them into the adapted models would be highly desirable. An overview of this new design can be found in Appendix A. The algorithm presented in the appendix considers the approach of adapting a single rule set to a data set. It can be extended to consider combining several rule sets.

Another work that could be extended from this thesis is building a hybrid-approach in which a genetic algorithm will combine and mutate rule sets while another type of algorithms (simulated annealing, hill-climbing, or even another GA) optimizes the rules.

Finally, a third direction would be the use of the GA to create rule sets from scratch by deriving them directly from the data set instead of optimizing existing models. This direction can be considered orthogonal to the current one since it does not allow the incorporation of already acquired knowledge into the new models - a characteristic of core interest in our current work.

REFERENCES

[Ackley, 1987] D. Ackley. *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers, 1987.

[Alander, 1995] Jarmo T. Alander. Indexed bibliography of genetic algorithms in power engineering. Technical Report 94-1POWER, University of Vaasa, Department of Information Technology and Production Economics, Finland, 1995.

[Alender and Lampinen, 1998] J.T. Alender and J. Lampinen. Cam shape optimization by genetic algorithms. In D. Quagliaverlla, J. Periaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolutionary Strategies in Engineering and Computer Science. Recent Advances and Industrial Applications*. 1998.

[Azar *et al.*, 2002] D. Azar, D. Precup, S. Boutkif, B. Kégl, and H. Sahraoui. Combining and adapting software quality predictive models by genetic algorithms. In *The Seventeenth IEEE International Conference on Automated Software Engineering*, 2002.

[Back, 1998] T. Back. Evolutionary algorithms: Applications at the informatik center darmunt. In D. Quagliaverlla, J. Periaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolutionary Strategies in Engineering and Computer Science. Recent Advances and Industrial Applications*. 1998.

[Baker, 1985] J.E. Baker. Adaptive selection methods for genetic algorithms. In Grefenstette, editor, *First International Conference on Genetic Algorithms and Their Applications*, Erlbaum, 1985.

[Barnes and Swim, 1993] G. Michael Barnes and Bradley R. Swim. Inheriting software metrics. *JOOP*, 6(7):27–34, Nov./Dec. 1993.

[Basili *et al.*, 1996] V. Basili, L. Briand, and W. Melo. How reuse influences productivity in object-oriented systems. In *Communications of the ACM*, volume 30, pages 104–114, 1996.

[Basili *et al.*, 1997] V. Basili, K. Condon, K. El Emam, R.B. Hendrick, and W.L. Melo. Characterizing and modeling the cost of rework in a library of reusable software components. In *Proceedings of the 19th International Conference on Software Engeineering*, pages 282–291, Boston, MA, May 1997.

[Beasley *et al.*, 1993a] D. Beasley, D. Bull, and R. Martin. An overview of genetic algorithms: Part 1, Fundamentals, 1993.

[Beasley *et al.*, 1993b] D. Beasley, D. Bull, and R. Martin. An overview of genetic algorithms: Part 2, Fundamentals, 1993.

[Blickle and Thiele, 1995] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in genetic algorithms. Technical Report 11, Computer Engineering and Communication Networks Lab (TIK). Swiss Federal institute of Technology(ETH), Zurich, Switzerland, June 1995.

[Boswell, 1990] R. Boswell. Manual for NewID. The Turing Institute, January 1990.

[Bouktif and Sahraoui, 2002] B. Bouktif, S. Kégl and H. Sahraoui. Combining software quality predictive models: An evolutionary approach. In *IEEE International Conference on Software Maintenance*, Montreal, 2002.

[Bouktif *et al.*, 2004] S. Bouktif, D. Azar, H. Sahraoui, B. Kégl, and D. Precup. Improving rule set-based software quality prediction: A genetic algorithm-based approach. *Journal of Object Technology*, 3(4), April 2004.

[Briand *et al.*, 1992] L.C. Briand, V.R. Basili, and W.M. Thomas. A pattern recognition approach for sfotware engineering analysis. In *IEEE Transactions on Software Engineering*, volume 18, Nov. 1992.

[Briand *et al.*, 1993] L. C. Briand, W.M. Thomas, and C. J. Hetmanski. Modeling and managing risk early in software development. In *Proceedings of the 15th International Conference on Software Engineering*, pages 55–65, 1993.

[Briand *et al.*, 1996] L.C. Briand, P. Devanbu, and W.L. Melo. Defining and validating design coupling measures in object-oriented systems. Technical Report ISERN-96-08, Fraunhofer Institute of Experimental Software Engineering, Germany, 1996.

[Briand *et al.*, 1997] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th International Conference on Software Engineering*, 1997.

[Briand *et al.*, 1999] L. Briand, J. Wüst, S. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: an industrial case study. In *Proceedings of the Twenty First IEEE International Conference on Software Engineering ICSE'99*, Los Angeles, USA, 1999.

[Carbonell, 1990] J. Carbonell, editor. *Machine Learning: Paradigms and Methods*. MIT Press, 1990.

[Chidamber and Kemerer, 1994] S. Chidamber and C.F. Kemerer. A metrics suite for object-oriented design. In *IEEE Transactions on Software Engineering*, volume 20, pages 476–493. June 1994.

[Clark and Niblett, 1989] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[Cohen and Devanbu, 1997] W. W. Cohen and P. Devanbu. A comparative study of inductive logic programming methods for software fault prediction. In *Fourteenth International Conference on Machine Learning*, pages 66–74. Morgan Kaufmann, 1997.

[Cohen, 1995] W. Cohen. Learning to classify english text with ILP methods. In *Advances in ILP*. IOS Press, 1995.

[Coppick and Cheatham, 1992] J. C. Coppick and T. J. Cheatham. Software metrics for object-oriented systems. In *CSC '92 Proceedings*, pages 317–322, 1992.

[Corcoran and Sen, 1994] A. L. Corcoran and S. Sen. Using real-valued genetic algorithms to evolve rule sets for classification. In *IEEE Conference on Evolutionary Computation*, pages 120–124, Orlando, Florida, June 1994.

[Darwin, 1859] C. Darwin. *The Origin of Species*. John Murray, 1859.

[Darwin, 1909] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle For Life*. London Cassell, 1909.

[Day *et al.*, 2003] R.O. Day, G. B. Lamont, and R. Pachter. Protein structure prediction by applying an evolutioanry algorithm, 2003.

[De Almeida *et al.*, 1999] M. A. De Almeida, H. Lounis, and W. Melo. An investigation on the use of machine learned models for estimating software correctability. *International Journal of Software Engineering and Knowledge Engineering, Spcial Issue on Knowledge Discovery from Empirical Software Engineering Data*, October 1999.

[De Jong and Sarma, 1993] K. A. De Jong and J. Sarma. Generation gaps revisited. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, 1993.

[De Jong and Spears, 1991] K. A. De Jong and W. M. Spears. Learning concept classification rules using genetic algorithms. In *Proceedings of the 12th Internationl Joint conference on Artificial Intelligence*, pages 651–656, Sydney, Australia, 1991.

[De Jong *et al.*, 1993] K. A. De Jong, W. M. Spears, and D. F. Gordon. Using genetic algorithms for concept learning. *Machine Learning*, 13(2/3):161–188, 1993.

[De Jong, 1975] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, Ann Arbor, 1975.

[Demeyer and Ducasse, 1999] S. Demeyer and S. Ducasse. Metrics, do they really help? In *Proceedings of LMO*, 1999.

[Driesen, 1994] K. Driesen. Compressing sparse tables using a genetic algorithm. In *Proceedings of the GRONICS'94 Student Conference*, Gorningen, the Netherlands, February 1994.

[Eiben and Smith, 2003] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computation*. Springer-Verlag, Berlin Heidelberg, 2003.

[Eiben *et al.*, 1995] A.E. Eiben, C.H.M. van Kemenade, and J.N. Kok. Orgy in the computer: Multi-parent reproduction in genetic algorithms. In F. Morán, A. Moreno, J.J. Merelo, and P. Chacón, editors, *Advances in Artificial Life. Third International Conference on Artificial Life*, volume 929 of Lecture Notes on Artificial Intelligence, pages 934–945, Berlin, Heidelberg, NY, 1995. Springer.

[Elomaa, 1994] T. Elomaa. In defense of C4.5: Notes on learning one-level decision trees. In *Machine Learning: Proceedings of the 11th International Conference*, pages 62–69. Morgan Kaufmann, 1994.

[Erdogmus and Tanir, 2002] H. Erdogmus and O. Tanir, editors. *Advances in Software Engineering - Comprehension, Evaluation, and Evolution*. Springer-Verlag, New York, 2002.

[Falkenauer, 1998] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. Wiley, Chichester, England; New York, 1998.

[Fenton and Pfleeger, 1997] N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, Boston, USA, second edition, 1997.

[Floreano, 1998] D. Floreano. Evolutionary mobile robotics. In D. Quagliaverlla, J. Periaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolutionary Strategies in Engineering and Computer Science. Recent Advances and Industrial Applications*. 1998.

[G. Rawlins, 1991] ed. G. Rawlins, editor. *A Study of Reproduction in Generational and Steady-State GAs*. Morgan Kaufmann, 1991.

[Galin, 2004] D. Galin. *Software Quality Assurance - From Theory to Implementation*. Addison Wesley, 2004.

[Garrell *et al.*, 1999] J. M. Garrell, E. Golobardes, and E. Bernadò. Automatic diagnosis with genetic algorithms and case-base reasoning. *In AIENG Journal*, 13:367–372, 1999. Elsevier Science.

[Ghanea-Hercock, 2003] Robert Ghanea-Hercock. *Applied Evolutionary Algorithms in Java*. Springer-Verlag, New York, 2003.

[Ghezzi *et al.*, 2003] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Pearson Education, Inc., 2003. 2nd Edition.

[Goldberg and Lingle, 1985] D. Goldberg and R. Lingle. Alleles, loci and the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms*, Mahwah, N.J., 1985. Lawrence Eribaum Associates.

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

[Grefenstette and Baker, 1989] J. Grefenstette and J.E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In *Proceedinf of the Thrid Interantinal Conference on Genetic Algorithms*, pages 20–27, San Mateo, CA, 1989. Morgan Kaufmann.

[Grefenstette *et al.*, 1985] J. J. Grefenstette, R. Gopal, R. Rosmaita, and D. Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms*, Mahwah, N.J., 1985. Lawrence Eribaum Associates.

[Guo *et al.*, 2003] L. Guo, B. Cukic, and H. Singh. Predicting fault prone models by the dempster-shafer belief networks. In *Eighteenth IEEE Conference on Automated Software Engineering*, pages 249–252, Montreal, Quebec, Canada, October 2003.

[Hamaifar *et al.*, 1993] L. Hamaifar, C. Guan, and G. Liepins. A new approach to the traveling salesman problem by genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, Los Altos, CA, 1993. Morgan Kaufmann Pulbishers.

[Han *et al.*, 2002] L. Han, G. Kendall, and P. Cowling. An adaptive length chromosome hyperheuristic genetic algorithm scheduling problem, 2002.

[Haupt and Haupt, 1998] S. E. Haupt and R.L. Haupt. *Practical Genetic Algorithms*. John Wiley and Sons, 1998.

[Henderson-Sellers, 1991] B. Henderson-Sellers. Some metrics for object-oriented software engineering. In *TOOLS Proceedings*, pages 131–139, 1991.

[Henderson-Sellers, 1996] Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[Holland, 1975] J.H. Holland. *Adaptation in Natural and Artificial Systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, Ann Arbor, MI, 1975.

[Holland, 1986] J.H. Holland. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R.S. Michalski, J.G. Carbonell, and eds. T. M. Mitchell, editors, *Machine Learning II*. Morgan Kaufmann, 1986.

[Hunt, 1975] E.B. Hunt. *Artificial Intelligence*. Academic Press, New York, 1975.

[IEEE, 1991] IEEE. IEEE std 610.12-1990 - ieee standard glossary of software engineering terminology. In *IEEE Software Engineering Standards Collection*. The Institute of Electrical and Electronics Engineers, New York, 1991. Corrected Edition, February 1991.

[IEEE, 1993] IEEE. IEEE std 610.12-1990. In *IEEE Software Engineering Standards Collection*, pages 47–48. The Institute of Electrical and Electronics Engineers, 1993.

[Ikonomovski, 1998] S. Ikonomovski. Detection of faulty components in object-oriented systems using design metrics and a machine learning algorithm. Master's thesis, McGill University, Montreal, Canada, 1998.

[ISO/IEC9126, 1991] ISO/IEC9126. Information technology-software product evaluation-quality characteristics and guidelines for their use, 1991.

[Jog *et al.*, 1989] P. Jog, J. Y. Suh, and D. V. Gucht. The effect of population size, heuristic crossover and local improvement on a genetic algorithm for the traveling salesman problem. In *Proceedings of the Third International Conference on Genetic Algorithms*, Los Altos, CA, 1989. Morgan Kaufmann Publishers.

[Jog *et al.*, 1991] P. Jog, J. Y. Suh, and D. V. Gucht. Parallel genetic algorithms applied to the traveling salesman problem. *In SIAM J. Optimization*, 1:515–529, 1991.

[Jorgensen, 1995] M. Jorgensen. Experience with the accuracy of software maintenance task effort prediction models. In *IEEE TSE*, volume 21, pages 674–681. 1995.

[Krueger, 1992] C.W. Krueger. Software reuse. In *ACM Computing Surveys*, volume 24, pages 131–183. 1992.

[Lanubile and Visaggio, 1996] F. Lanubile and G. Visaggio. Evaluating predictive quality models derived from software measures: Lessons learned. Technical Report ISERN-96-03, University of Bari, Italy, 1996.

[Li and Henry, 1993a] W. Li and S. Henry. Maintenance metrics for the object-oriented paradigm. In *Proceedings of the First International Software Metrics Symposium*, pages 52–60, Baltimore, Maryland, 1993.

[Li and Henry, 1993b] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2), 1993.

[Llorà and Garrell, 1999] X. Llorà and J. M. Garrell. GENIFER: A nearest neighbour based classifier system using ga. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO99)*. Morgan Kaufmann, 1999.

[Llorà and Garrell, 2001a] X. Llorà and J. M. Garrell. Evolution of decision trees. In *Proceedings of the Fourth Catalan Conference on Artificial Intelligence (CCIA 2001)*. ACIA Press, 2001.

[Llorà and Garrell, 2001b] X. Llorà and J. M. Garrell. Inducing partially-defined instances with evolutionary algorithms. In *Proceedings of the 18th International Conference on Machine Learning (ICML 2001)*. Morgran Kauffman, 2001.

[Logan and Riccardo, 1996] B. Logan and P. Riccardo. Route planning with GA. In *First Online Workshop on Soft Computing*. August 19-30, 1996.

[Lorenz and Kidd, 1994] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Approach*. Prentic-Hall, 1994.

[Mao *et al.*, 1998] Y. Mao, H.A. Sahraoui, and H. Lounis. Reusability hypothesis verification using machine learning techniques: A case study. In *IEEE Automated Software Engineering Conference*, 1998.

[Martin-Albo *et al.*, 2003] J. Martin-Albo, M.F. Bertoa, C. Calero, A. Vallecillo, and A. Chechich. Cqm: A software component metric classification model. In *Seventh ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2003)*. July 2003.

[Miceli *et al.*, 1999] T. Miceli, H. Sahraoui, and R. Godin. A metric based technique for design flaws detection and correction. In *IEEE Automated Software Engineering Conference*, 1999.

[Michalewicz, 1996] Z. Michalewicz. *Genetic Algorithms+Data Structures=Evolution Programs*. Springer Verlag, 1996.

[Miller *et al.*, 1998] J.F. Miller, Thomson P., and T. Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetics circuits: A cast study. In D. Quagliaverlla, J. Periaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolutionary Strategies in Engineering and Computer Science. Recent Advances and Industrial Applications*. 1998.

[Minaei and Punch, 2003] B. Minaei and Punch. Using genetic algorithms for data mining optimization in an educational web-based system. In *GECCO2003*, pages 2252–2263, 2003.

[Mitchell, 1997] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[Mitchell, 1999] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press Commercial, Cambridge, Mass., 1999.

[Mozetic, 1985] I. Mozetic. NEWGEM: Program for learning from examples, program documentation and user's guide. Technical Report Report Number UIUCDCS-F-85-949, University of Illinois, 1985.

[Oliver *et al.*, 1987] I. M. Oliver, D. J. Smith, and J. R. Holland. A study of permutation crossover operators on the traveling salesman prolbem. In *Proceedings of the Third International Conference on Genetic Algorithms*, London, 1987. Lawrence Eribaum Associates.

[Pei *et al.*, 1994] M. Pei, E. D. Goodman, W. F. Punch, and Lai C. Intelligent classification and identification of biology data patterns using genetic algorithms. In *The First Symposium on Life Sciences and Biotechnology for Chinese Scientists Overseas and At Home*. Beijing, China, August 1994.

[Pei *et al.*, 1995] M. Pei, E. D. Goodman, W. F. Punch, and Y. Ding. Genetic algorithms for classificaiton and feature exctractoin. 1995 Annual Meeting, Classification Society of North America, June 1995.

[Piattini *et al.*, 2002] M. Piattini, C. Calero, H. A. Sahraoui, and H. Lounis. An empirical studey with object-relational databases metrics. In *The Seventh European Conference on Software Quality*, 2002.

[Porter and Selby, 1988] A. Porter and R. Selby. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *Software Engineering*, 14(12):1743–1757, 1988.

[Porter and Selby, 1990] A. Porter and R. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, 1990.

[Porter, 1998] B. Porter. Evolutionary synthesis of control policies for manufacturing systems. In D. Quagliaverlla, J. Periaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolutionary Strategies in Engineering and Computer Science. Recent Advances and Industrial Applications.* 1998.

[Potter *et al.*, 1995] M.A. Potter, K. De Jong, and J. Grefenstette. *A Coevolutionary Approach to Learning Sequential Decision Rules*, 1995.

[Pressman, 1997] R.S. Pressman. *Software Engineering: A Practical Approach*. McGraw-Hill, fourth edition, 1997.

[Punch *et al.*, 1993] W. F. Punch, E. D. Goodman, M. Pei, L. Chia-Shun, P. Hovland, and R. Enbody. Further research on feature selection and classification using genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 557–564, Urbana-Champaign Ill., July 1993.

[Quinlan, 1990] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, August 1990.

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

[Ramoni and Sebastiani, 1999] M. Ramoni and P. Sebastiani. Bayesian methods for intelligent data analysis. In M. Berthold and D. J. Hand, editors, *An Introduction to Intelligent Data Analysis*. Springer, New York, 1999.

[Riguzzi, 1996] F. Riguzzi. A survey of software metrics. Technical Report 96-010, Universita degli Studi di Bologna, July 1996.

[Rowe and East, 1993] J. Rowe and I. East. Direct replacement: A GA without mutation which avoids deception. In *Lecture Notes in AI. Progress in Evolutionary Computation*, volume 956, 1993.

[Sahraoui and Azar, 1999] H. Sahraoui and D. Azar. Quality estimation models optimization using genetic algorithms: Case of maintainability. In *European Software Measurement Conference*, 1999.

[Sahraoui *et al.*, 2000a] H. Sahraoui, M. Boukadoum, and H. Lounis. Using fuzzy threshold values for predicting class libraries interface evolution. In *The Fourth International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2000.

[Sahraoui *et al.*, 2000b] H. A. Sahraoui, R. Godin, and T. Miceli. Can metrics help bridging the gap between the improvement of OO design quality and its automation? In *Proceedings of the International Conference on Software Maintenance (ICSM2000)*, 2000.

[Sahraoui *et al.*, 2001] H.A. Sahraoui, M. Boukadoum, and H. Lounis. Building quality estimation models with fuzzy threshold values. In Edition Hermès Sciences, editor, *L'Objet*, volume 17, pages 535–554, 2001.

[Schmitt and Amini, 1998] L. J. Schmitt and M. N. Amini. Performance characteristics of alternative genetic algorithmic approaches to the traveling salesman problem using path representation: An empirical study. *European Journal of Operational Research*, 108:551–570, 1998.

[Sen and Knight, 1995] Sandip Sen and Leslie Knight. A genetic prototype learner. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 725–731, Montreal, Canada, August 1995.

[Shepperd and Kadoda, 2001] M. Shepperd and G. Kadoda. Using simulation to evaluate prediction techniques. In *IEEE Seventh International Software Metrics Symposium*. London, England, April 2001.

[Siedlecki and Sklansky, 1989] W. Siedlecki and J. Sklansky. A note on genetic algorithms for large-scale feature selection. In *Pattern Recognition Letters*, volume 10, pages 335–347. 1989.

[Spears and DeJong, 1993] W.M. Spears and K.A. DeJong. Using genetic algorithms for supervised concept learning. In *Machine Learning*, volume 13, pages 161–188, 1993.

[Spears and F., 1991] William M. Spears and Gordon D. F. Adaptive strategy selection for concept learning. In *Proceedings of the First International Workshop on Multistrategey Learning*, pages 231–246, 1991.

[Starkweather *et al.*, 1991] T. Starkweather, D. Whitley, C. Whitley, and K. Mathial. A comparison of genetic sequencing operators. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, Los Altos, CA, 1991. Morgan Kaufmann Pulbishers.

[Syswerda, 1989] G. Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan-Kaufmann, 1989.

[West, 1996] Douglas B. West. *Introduction to Graph Theory*. Upper Saddle River, NJ : Prentice Hall, Prentice Hall, 1996.

[Whitley *et al.*, 1989] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Proceedings of the Third International Conference on Genetic Algorithms*, Los Altos, CA, 1989. Morgan Kaufmann Publishers.

[Whitley *et al.*, 1991] C. Whitley, T. Starkweather, and D. Shaner. The traveling salesman and sequence scheduling quality solutions using genetic edge recombination. In Van Nostrand Reinhold, editor, *Handbook of Genetic Algorithms*. New York, 1991.

[Whitley, 1989] D. Whitley. The GENITOR algorithm and selective pressure. In *Proceedings fo the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan-Kaufmann, 1989.

[Whitley, 1991] D. Whitley. Fundamental principles of deception in genetic search. In *Foundation of Genetic Algorithms*, pages 221–241. Morgan Kaufmann, San Mateo, CA, 1991.

[Whitley, 1994] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[Youden, 1998] W.J. Youden. How to evaluate accuracy. In *Materials Research and Standards, ASTM*. 1998.

[Zhong *et al.*, 2004] S. Zhong, T. Khoshgoftaar, and N. Seliya. Unsupervised learning for expert-based software quality estimation. In *The Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE 2004)*, Tampa, Florida, USA, March 2004.

APPENDICES

APPENDIX A

In this appendix, we describe a genetic algorithm based on a graph representation of the chromosomes. As we describe the technique, we will see that its strengths lie in three major points:

(i) **Implementation simplicity.** The graph representation leads to a very simple implementation of *crossover* and *mutation*.

(ii) **Ease to incorporate new attributes.** A rule set built with a specific set of attributes can be adapted to a data set with more attributes.

(iii) **More variation in the conditions making the rules.** The mutation operator changes not only the values involved in the conditions but also the attributes and the operators.

A.1. Overview

We consider the approach of optimizing a single model. Similar to the GA presented in Chapter 5, a rule set represents a population of chromosomes. The major difference compared to the previous approach is the representation of a chromosome. Here, a rule set is a graph and a rule in the rule set is a subgraph.

A.2. Representation and Fitness Function

A rule set is represented as a **bipartite graph**.

Definition A.2.1. *An **independent set** in a graph is a set of pairwise nonadjacent vertices [West, 1996].*

Definition A.2.2. *A graph G is **bipartite** if the set of vertices $V(G)$ is the union of two disjoint (possibly empty) independent sets called **bipartite sets** [West, 1996].*

Figure A.2.1 shows an example of a graph and a bipartite graph.

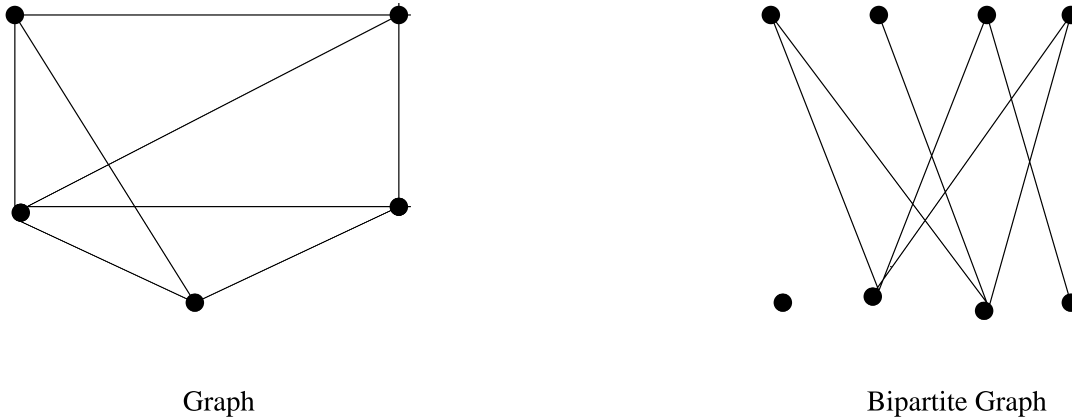

Figure A.2.1. An example of a graph and a bipartite graph.

Our representation consists of mapping a rule set to a bipartite graph. One independent set, A, is formed of vertices that represent attributes a_i $(1 \leq i \leq n)$ where a_i is one attribute in the data set from which the rule set is built. All the attributes are represented by vertices. The other independent set, V, is formed of vertices that represent values, v_j $(1 \leq j \leq m)$ where v_j is a value in a condition. We draw an edge, E, between two vertices, $a_i \in A$ and $v_j \in V$, if and only if a condition including the attribute a_i and the value v_j is found in the rule set. Each condition in the rule set is represented by exactly one edge in the graph. Hence, if a value appears more than once in the rule set, it will appear more than once in a value vertex in the graph. We call the vertex a_i the **attribute-vertex** of edge E and v_j its **value-vertex**. We assign to each edge in the graph a weight $w \in \{0, 1\}$. w is 0 if the condition $a_i \leq v_j$ is in the rule, and 1 if the condition $a_i > v_j$ is found in the rule. We

assign to all the edges representing one rule a unique color that identifies this rule. We use the notation $E(a_i, v_j, w, c)$ to designate the edge connecting vertices a_i and v_j such that the edge is assigned a weight w and a color c. Hence, each subgraph representing a rule has a unique color. We complement each rule subgraph with a **class vertex** c_k which represents the classification label of the rule that the subgraph represents. To attach the vertex c_k to the subgraph, we draw an edge between c_k and exactly one of the attribute vertices, a_i, in the subgraph. We give to this edge the color that identifies the subgraph. We assign to all edges that connect a class vertex to an attribute vertex a weight of -1 (this is equivalent to assigning no weight to the edge). We call such edges **special edges**. One can easily see that the graph remains bipartite. Figure A.2.2 shows a rule set and its graph representation.

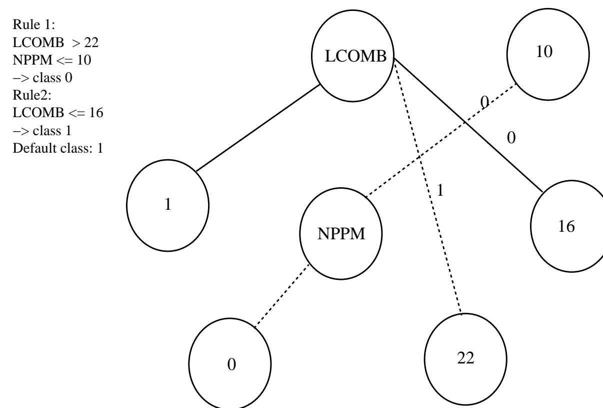

FIGURE A.2.2. A rule set and its bipartite graph representation. Each rule in the rule set is represented with a different type of line in the graph. A node representing a classification label is assigned to each rule. The default classification label of the rule set is saved in a separate field.

Similar to the algorithm described in Chapter 5, a chromosome is evaluated in terms of the accuracy and the coverage of the rule that it represents using fitness functions (5.1) and (5.2).

A.2.1. Crossover. In this GA, each rule is a chromosome represented as a subgraph. Genes are edges in the subgraph. *Crossover* consists of exchanging colors between two edges (of a different color and different from the special edges) in the

graph. This is equivalent to exchanging conditions between two rules in the rule set. More formally, if we consider the following two chromosomes:

Chromosome 1 $(a_{11}, v_{11}, w_{11}, l_1), (a_{12}, v_{12}, w_{12}, l_1), (a_{13}, v_{13}, w_{13}, l_1), (a_{11}, c_1, -1, l_1)$

Chromosome 2 $(a_{21}, v_{21}, w_{21}, l_2), (a_{22}, v_{22}, w_{22}, l_2), (a_{23}, v_{23}, w_{23}, l_2), (a_{21}, c_2, -1, l_2)$

One way of performing crossover between them is to exchange the color between the first edge in Chromosome 1 and the third in Chromosome 2. This will give the following two offspring:

Offspring 1 $(a_{23}, v_{23}, w_{23}, l_2), (a_{12}, v_{12}, w_{12}, l_1), (a_{13}, v_{13}, w_{13}, l_1), (a_{11}, c_1, -1, l_1)$

Offspring 2 $(a_{21}, v_{21}, w_{21}, l_2), (a_{22}, v_{22}, w_{22}, l_2), (a_{11}, v_{11}, w_{11}, l_1), (a_{21}, c_2, -1, l_2)$

Figure A.2.3 shows a more specific example on a graph representation. In this figure, we designate by 0 the 'less than or equal operator' and by 1 the 'greater than' operator, exchanging colors[1] between the two edges, $(LCOMB, 16, 0, 0)$ and $(LCOMB, 22, 1, 1)$ results in the conditions $LCOMB \leq 16$ and $LCOMB > 22$ being exchanged between two rules.

A.2.2. Mutation. The mutation operator that we define for this representation changes not only the conditions to which attributes are compared but the relations as well. We define three different types of mutation:

(i) *Weight mutation* changes the weight assigned to an edge $E = (a_i, v_j, w, l)$. This is equivalent to changing the relational operator in a condition. Figure A.2.4 shows an example.

(ii) *Edge insertion* adds an edge to the graph by connecting an attribute vertex a_i to value vertex v_j where v_j is a value chosen randomly from the set of cutpoints for attribute a_i. The weight and the color of the edge are chosen randomly. This is equivalent to adding a condition to a rule.

[1]Different line types indicate different colors in the figure.

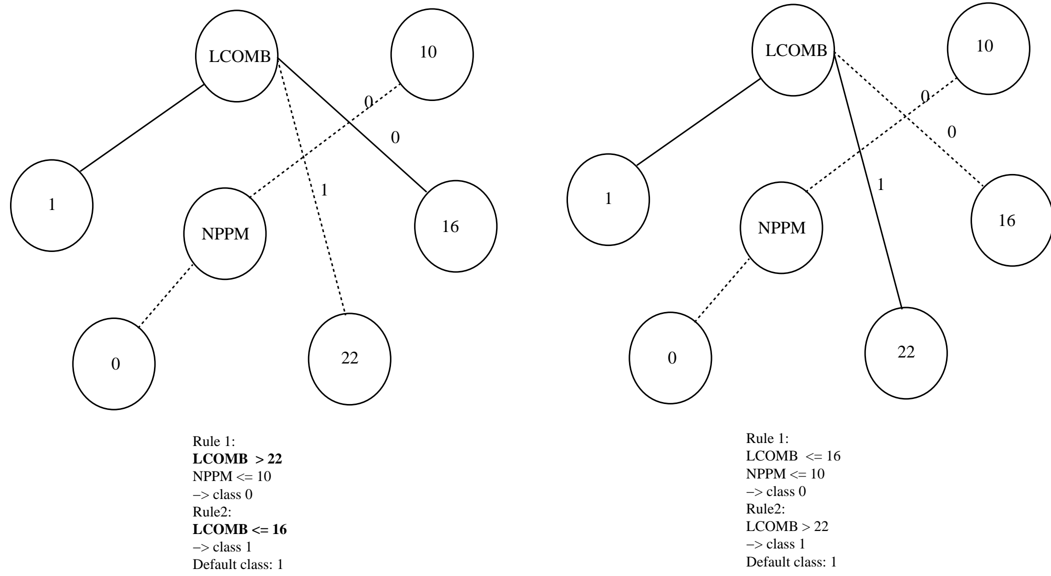

FIGURE A.2.3. Crossover. Exchanging colors (line types) between two edges results in the two conditions in bold being swapped.

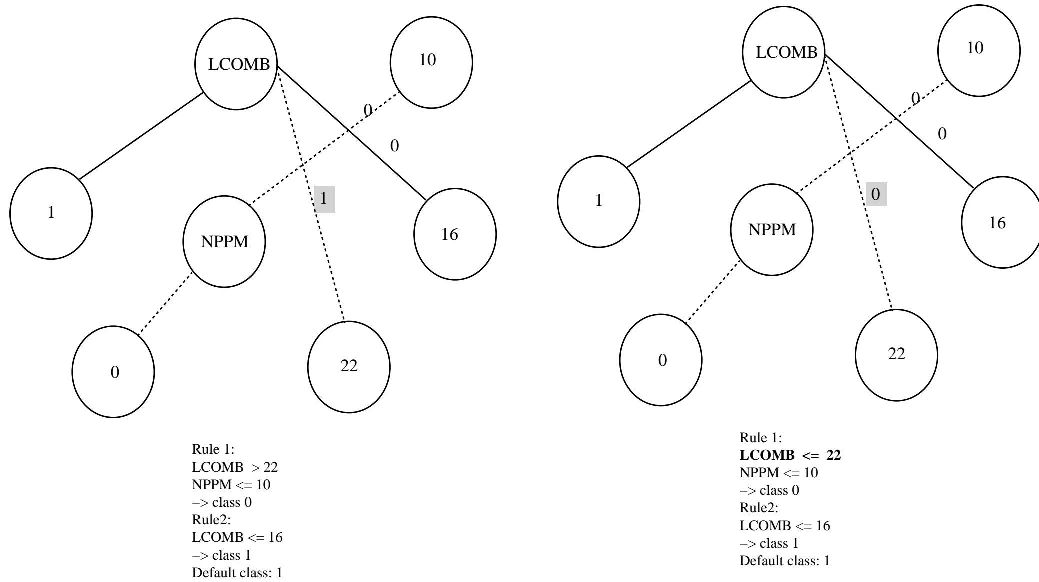

FIGURE A.2.4. Weight mutation operator. Changing the weight of edge (LCOMB, 22, 1, 1) from 1 to 0 changes the condition $LCOMB > 22$ to $LCOMB \leq 22$ in the first rule.

(iii) *Value mutation* operates on the level of the value vertices. It changes the value associated with such a vertex to an adjacent one on the cutpoint list.

For this, we associate with each attribute a_i an ordered set of cutpoint values, $V = \{v_1, v_2, ...v_n\}$ where $v_i < v_{i+1}, \forall i < n$. If we designate by $adj(v_i)$ the set of values adjacent to v_i then, $\forall v_i \in V$, where $i \neq 1$ and $i \neq n$, $adj(v_i) = \{v_{i-1}, v_{i+1}\}$, $adj(v_1) = \{v_2\}$ and $adj(v_n) = \{v_{n-1}\}$. When value mutation is applied to (a_i, v_j, w, l), this will change into (a_i, v_k, w, l) where $v_k \in adj(v_j)$. This is a conservative approach that does not result in big changes. A value in a condition is slightly modified to see how much it can affect the rule set as a whole. Figure A.2.5 shows an example.

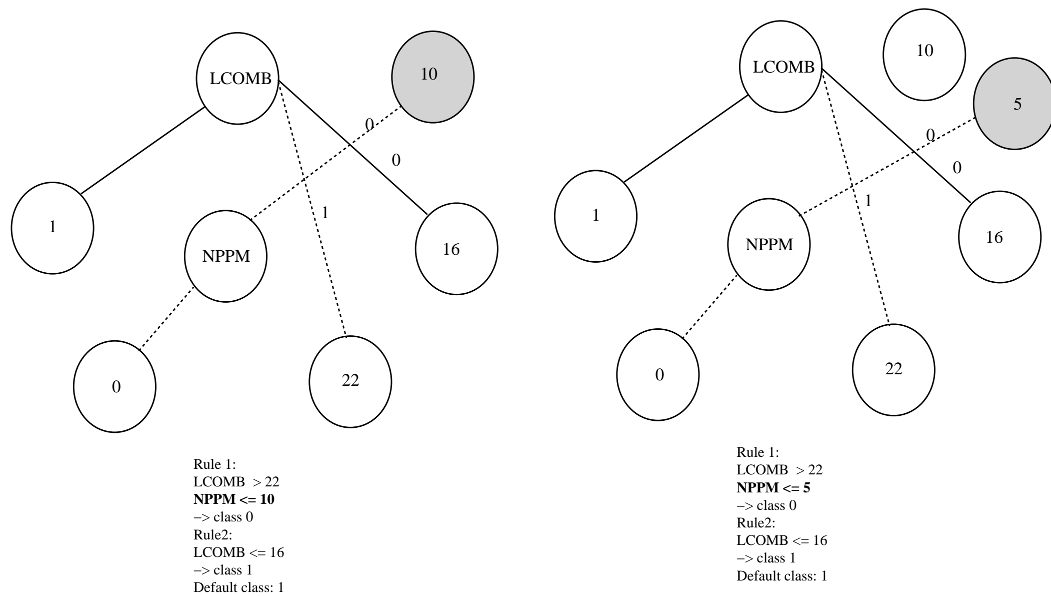

FIGURE A.2.5. Value mutation operator. Replacing a vertex value with another replaces the condition $NPPM \leq 10$ with $NPPM \leq 5$.

(iv) *Class mutation* is similar to *value mutation* but is applied to *special edges* only. The value associated with the class vertex is changed to a different one chosen randomly from the set of classification labels. Figure A.2.6 shows an example where the classification label of *Rule*2 is changed from 1 to 0.

A.2.3. Postprocessing. We can see that the way we defined *crossover* and *mutation* above results in syntactically correct rules all the time. However, it is possible that the graph represents rules that contain redundancy and inconsistency

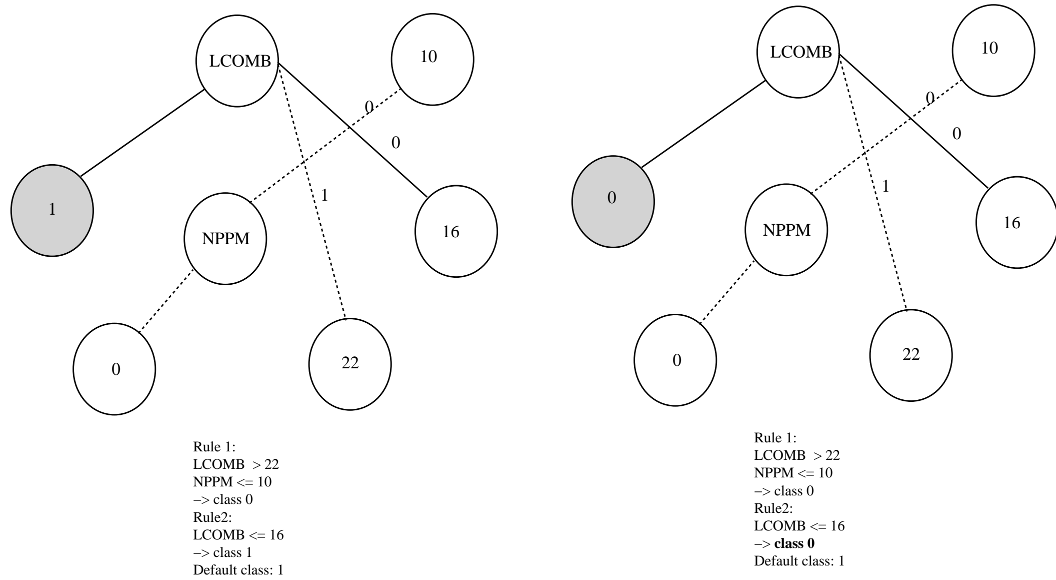

FIGURE A.2.6. Class mutation operator. The vertex representing the mutated classification label is highlighted.

(see definitions in Section 2.3). Solving these two problems is very simple with this representation. We say that edge (a_i, v_i, w_i, l_i) is **greater than** edge (a_j, v_j, w_j, l_j) if and only if the value that v_i represents is greater than the value that v_j represents. For each vertex a_i, for each color c, we sort in an increasing order all edges incident with a_i with a weight of 1 and we sort in a decreasing order all edges with a weight of 0. These represent all the conditions involving the attribute represented by a_i in a rule. Among the sorted edges with a weight of 1, we keep the one connecting a_i to the vertex with the highest value. Among those with a weight of 0, we keep the one connecting a_i to the vertex with the lowest value. All edges incident at a class vertex c are kept. We delete all the others. This eliminates redundancy by removing a condition l_i if it is subsumed by at least one other condition l_j in the same rule. It is also possible that the rules contain contradicting conditions. These are kept until the end of the evolutionary cycle. They are removed from the last rule set in case they survive.

A.3. Motivations for This Technique

(i) **Simple crossover and mutation operators.** Representing the rule set as a graph leads to a simple implementation of crossover and mutation. As a matter of fact, crossover consists of just a change in the edge color. Mutation consists of a modification in the value vertices, the weights of edges or the addition of new edges. These are operators very simple to implement as opposed to the crossover operator (especially double point crossover) for the GA described in Chapter 4.

(ii) **Strength of the mutation operator.**
The mutation operator used previously in this thesis, produced a change only in the value to which an attribute is a compared in a condition and the class labels. In this implementation, mutation leads to a change in the value (*value mutation*), the relational operator (*weight mutation*), and the classification label (*class mutation*). Moreover, *edge insertion mutation* creates new conditions, from scratch, by including edges between an attribute vertex and a value vertex. The adaptive GA that we presented in Chapter 5 resulted in a depletion of the condition pool at a certain point during the evolution process. No further improvement was possible due to this behavior. The algorithm that we present in this appendix overcomes this problem by keeping the disconnected attribute vertices in the graph, making it possible to draw an edge between an attribute vertex and a disconnected value vertex. Hence, if an attribute disappears from a population, it is still possible for it to re-emerge in future populations.

(iii) **Ease of incorporating new attributes.**
With this GA, it is possible to take a rule set built on one data set with a set of attributes and adapt it to a different set of data with more attributes. The new attributes are included in the graph as disconnected vertices which can become connected by the *edge insertion* mutation. This is especially

important in software engineering where new metrics are constantly proposed and older metrics can become deprecated.

Extending this technique to combine several rule sets should not be too difficult.

Document Log:

Manuscript Version 0

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX — 8 October 2004

Danielle Azar

McGill University, 3480 University St., Montréal (Québec) H3A 2A7, Canada, *Tel.* : (514) 398-7071

E-mail address: dazar@cs.mcgill.ca, danielle.azar@mail.mcgill.ca

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX