New Challenges in Learning Classifier Systems: Mining Rarities and Evolving Fuzzy Models

Albert Orriols i Puig

Grup de Recerca en Sistemes Intel·ligents
Enginyeria i Arquitectura La Salle
Universitat Ramon Llull

November 2008

Supervisor: Dr. Ester Bernadó i Mansilla

Abstract

During the last decade, Michigan-style learning classifier systems (LCSs)—genetic-based machine learning (GBML) methods that combine *apportionment of credit techniques* and *genetic algorithms* (GAs) to evolve a population of *classifiers* online—have been enjoying a renaissance. Together with the formulation of first generation systems, there have been crucial advances in (1) systematic design of new competent LCSs, (2) applications in important domains, and (3) theoretical analyses for design. Despite these successful designs and applications, there still remain difficult challenges that need to be addressed to increase our comprehension of how LCSs behave and to scalably and efficiently solve real-world problems.

The purpose of this thesis is to address two important challenges—shared by the machine learning community—with Michigan-style LCSs: (1) learning from domains that contain rare classes and (2) evolving highly legible models in which human-like reasoning mechanisms are employed. Extracting accurate models from rare classes is critical since the key, unperceptive knowledge usually resides in the rarities, and many traditional learning techniques are not able to model rarity accurately. Besides, these difficulties are increased in online learning, where the learner receives a stream of examples and has to detect rare classes on the fly. Evolving highly legible models is crucial in some domains such as medical diagnosis, in which human experts may be more interested in the explanation of the prediction than in the prediction itself.

The contributions of this thesis take two Michigan-style LCSs as starting point: the *extended classifier system* (XCS) and the *supervised classifier system* (UCS). XCS is taken as the first reference of this work since it is the most influential LCS. UCS is a recent LCS design that has inherited the main components of XCS and has specialized them for supervised learning. As this thesis is especially concerned with classification problems, UCS is also considered in this study. Since UCS is still a young system, for which there are several open issues that need further investigation, its learning architecture is first revised and updated. Moreover, to illustrate the key differences between XCS and UCS, the behavior of both systems is compared on a collection of boundedly difficult problems.

The study of learning from rare classes with LCSs starts with an analytical approach in which the problem is *decomposed* in five critical elements, and *facetwise models* are derived for each element. The analysis is used as a tool for designing configuration guidelines that enable XCS and UCS to solve problems that previously eluded solution. Thereafter, the two LCSs are compared with several highly-influential learners on a collection of real-world problems with rare classes, appearing as the two best techniques of the comparison. Moreover, *re-sampling* the training data set to eliminate the presence of rare classes is demonstrated to benefit, on average, the performance of LCSs.

The challenge of building more legible models and using human-like reasoning mechanisms is addressed with the design of a new LCS for supervised learning that combines the online evaluation capabilities of LCSs, the search robustness over complex spaces of GAs, and the legible knowledge representation and principled reasoning mechanisms of fuzzy logic. The system resulting from this crossbreeding of ideas, referred to as Fuzzy-UCS, is studied in detail and compared with several highly competent learning systems, demonstrating the competitiveness of the new architecture in terms of the accuracy and the interpretability of the evolved models. In addition, the benefits provided by the online architecture are exemplified by extracting accurate classification models from large data sets.

Overall, the advances and key insights provided in this thesis help advance our understanding of how LCSs work and prepare these types of systems to face increasingly difficult problems, which abound in current industrial and scientific applications. Furthermore, experimental results highlight the robustness and competitiveness of LCSs with respect to other machine learning techniques, which encourages their use to face new challenging real-world applications.

Acknowledgments

If belonging to a competitive research group and collaborating, sharing, and enjoying with your group mates is one the best experiences of a PhD lifetime, I consider myself one of the luckiest PhD students, since I have been working in three highly-competitive research groups. For this reason, I am tremendously grateful to all and each one of the people in there. If I tried to write sounding words to express all my gratitude, these would only give a vague idea of what I mean. Thence, following my engineering vocation, I would start simplifying and saying "thank you".

First, I would like to thank Ester Bernadó-Mansilla for accepting me as her first PhD student, for her advice, and for permitting my successive visits to the *Illinois genetic algorithms laboratory* (IlliGAL) and the *soft computing and intelligent information systems* (SCI2S) group. I would like to extend this acknowledgment to all the people in the *grup de recerca en sistemes intel·ligents* (GRSI) and the computer engineering department of *enginyeria i arquitectura la Salle* in general, with which I had interesting discussions, combined with lots of fun, during the last four years.

I am in terrible debt with all the people of the two research groups which I consider as my second homes: the IlliGAL and the SCI2S. My visits to Champaign and Granada were extremely fruitful and defined the largest part of the present document.

I am really glad that Prof. Goldberg accepted my visits to the IlliGAL and considered me one of his students. I still remember when I reached the lab for the first time with a particular problem to solve and left it with a methodology to face new challenging engineering problems. I think that Prof. Goldberg's great ability to see the big picture, to make key points, to formulate difficult, really interesting questions, to express with strength the ideas in a piece of paper, and, to, in five minutes, go far beyond my reach will never stop surprising me. I will never know what I have missed from Prof. Goldberg explanations, but I am extremely glad of all the things I know I got from him.

I also thank all the great people I met in the IlliGAL. I would especially single out Kumara Sastry. I am very lucky to have been able to cope with, as defined by him, his *not-so-nice* explanation skills. I am really grateful to Kumara for all his explanations, for his help, and for making the lab a funny place to work. I also enjoyed working with and learned a lot from Pier Luca Lanzi, who showed me new faces of learning classifier systems that I had never seen before. I am also grateful for the support of Xavier Llorà and Tian Li Yu in my visits.

I am very grateful to Jorge Casillas for insisting on my first visit to the SCI2S group with the original idea of mixing learning classifier systems and fuzzy logic and for all the great support and guidance provided not only during my successive visits, but also during the entire last year of my PhD. I learned a lot from all our long, daily talks and from Jorge's eagerness and passion to face new challenging real-world problems. I would like to extend this acknowledgement to Francisco Herrera for all his valuable advices and for always being ready to help me and to answer my questions. My visits to Granada were not only fruitful research-wise but also life-wise. I really enjoyed our Thursday's home parties and hanging around Granada with the SCI2S members. Especially, I would like to thank my flatmates, Pietro Ducange and Manolo Cobo, for making my stay in Granada so joyful.

The present work is the result of the collaboration with a number of researchers. I would like to thank my coauthors Ester Bernadó-Mansilla, Jorge Casillas, David E. Goldberg, Pier Luca Lanzi, Núria Macià, Francisco J. Martínez-López, Sergio Morales-Ortigosa, Joaquim Rios-

Boutin, Kumara Sastry, and Francesc Teixidó-Navarro. I would also thank Xavier Llorà for really interesting research talks.

Last, but not least, I would like to thank the unconditional support of all my family. I would like to thank my grandparents Josep and Antònia, my parents Albert and Maria Cinta, and my sister Gemma for their motivation to go on with my PhD. Also, I would like to especially thank the person who suffered my busy weekends, my stress when deadlines were approaching, and the distance while I was abroad the most; M. Carme, thanks for being there despite all this.

This research has been financially supported by the *departament d'universitats, recerca i societat de la informació* (DURSI) under a scholarship in the FI research program with reference *2005FI-00252*. I also acknowledge the support provided by DURSI in my visits to the IlliGAL, with two travel grants with references *2006BE-00299* and *2007BE2-00124*. Finally, I would like to acknowledge the *ministerio de educación y ciencia* for its support under the KEEL and the KEEL II projects (with references TIC2002-04036-C05-03 and TIN2005-08386-C05-04), and *Generalitat de Catalunya* for its support under the grant 2005SGR-00302.

Contents

List of Figures **vii**

List of Tables **xi**

List of Algorithms **xvii**

1 Introduction **1**

1.1 Framework: From Holland's Definition to Current LCSs . . . 2
1.2 Two Critical Challenges in LCSs and Machine Learning . . . 5
1.3 Thesis Objectives . . . 7
1.4 Road Map . . . 8

2 Machine Learning with Learning Classifier Systems **11**

2.1 Machine Learning . . . 11
2.1.1 Supervised learning . . . 14
2.1.2 Unsupervised learning . . . 14
2.1.3 Reinforcement Learning . . . 14
2.2 Evolutionary Computation and Genetic Algorithms . . . 15
2.2.1 Biological Principles that Inspire Evolutionary Computation . . . 16
2.2.2 Evolutionary Computation: A Taxonomy . . . 17
2.2.3 Genetic Algorithms . . . 18
2.2.4 Basic Theory of GA . . . 21
2.2.5 Genetic Algorithms in Real-World Applications . . . 24
2.3 Genetic-based Machine Learning and Learning Classifier Systems . . . 25
2.3.1 Michigan-style LCSs . . . 26
2.3.2 Pittsburgh-style LCSs . . . 28
2.3.3 Iterative Rule Genetic-based Machine Learning . . . 29
2.3.4 Genetic Cooperative-Competitive Learning . . . 30

2.3.5 The Organizational Classifier System 30
2.4 Summary 31

3 Description of XCS and UCS **33**
3.1 The XCS Classifier System 34
3.1.1 Knowledge Representation 34
3.1.2 Learning Interaction 35
3.1.3 Classifier Evaluation 36
3.1.4 Classifier Discovery 38
3.1.5 Class Inference in Test Mode 39
3.1.6 Why Does XCS Work? 39
3.2 The UCS Classifier System 41
3.2.1 Knowledge Representation 41
3.2.2 Learning Interaction 42
3.2.3 Classifier Evaluation 43
3.2.4 Classifier Discovery 43
3.2.5 Class Inference in Test Mode 44
3.2.6 Why does UCS work? 45
3.3 Rule Representations for LCSs 45
3.3.1 From the Ternary to the Interval-based Rule Representation in LCSs . . . 46
3.3.2 The Unordered Bound Representation 47
3.4 Summary and Conclusions 50

4 Revisiting UCS: Fitness Sharing and Comparison with XCS **51**
4.1 Fitness Sharing in GAs and LCSs 52
4.2 A New Fitness Sharing Scheme for UCS 53
4.3 Methodology 54
4.4 Analyzing the Fitness Sharing Scheme in UCS 55
4.5 Comparing UCSs with XCS 60
4.6 Lessons Learned from the Analysis 64
4.6.1 Fitness Sharing 64
4.6.2 Explore Regime 65
4.6.3 Accuracy Guidance 65
4.6.4 Population Size 66
4.7 Summary and Conclusions 66

5 Facetwise Analysis of XCS for Domains with Class Imbalances **67**
5.1 The Challenges of Learning from Imbalanced Domains in Machine Learning . . . 68
5.2 The XCS Classifier System in Imbalanced Domains 70
5.2.1 Hypotheses of XCS Difficulties in Learning from Imbalanced Domains . . 70
5.2.2 Empirical Observations of XCS Behavior on Class Imbalances 71
5.3 Facetwise Analysis of XCS in Imbalanced Domains 73
5.3.1 Design Decomposition in GAs 73
5.3.2 Carrying the Design Decomposition from GAs to XCS 73
5.3.3 A Boundedly Difficult Problem for LCSs: The Imbalanced Parity Problem 74
5.3.4 Decomposition of the Class Imbalance Problem in XCS 75
5.4 Estimation of Classifier Parameters 77
5.4.1 Imbalance Bound 77
5.4.2 Does the Widrow-Hoff Rule Provide Accurate Estimates? 78
5.4.3 Obtaining Better Estimates with the Widrow-Hoff Rule 79
5.4.4 Obtaining Better Estimates with Gradient Descent Methods 80
5.5 Supply of Schemas of Starved Niches in Population Initialization 81
5.6 Generation of Classifiers in Starved Niches 82
5.6.1 Assumptions for the Model 83
5.6.2 Genetic Creation of Representatives of Starved Niches 83
5.6.3 Deletion of Representatives of Starved Niches 85
5.6.4 Bounding the Population Size 85
5.6.5 Experimental Validation of the Models 86
5.7 Occurrence-based Reproduction: The Role of θ_{GA} 89
5.7.1 Including θ_{GA} in the Generation Models 90
5.7.2 Experimental Validation 91
5.8 Takeover Time of Accurate Classifiers in Starved Niches 92
5.8.1 Model Assumptions 93
5.8.2 Takeover Time for Proportionate Selection 93
5.8.3 Takeover Time for Tournament Selection 97
5.8.4 Experimental Validation of the Takeover Time Models 101
5.9 Lessons Learned from the Models 104
5.9.1 Patchquilt Integration of the Facetwise Models 104
5.9.2 Solving Problems with Large Imbalance Ratios 105
5.10 Summary and Conclusions 106

6 Carrying over the Facetwise Analysis to UCS 109
6.1 Design Decomposition for UCS . . . 110
6.2 Estimation of Classifier Parameters . . . 111
6.3 Supply of Schemas of Starved Niches in Population Initialization . . . 112
6.4 Generation of Classifiers in Starved Niches . . . 113
6.4.1 Assumptions for the Model . . . 114
6.4.2 Creation and Deletion of Representatives of Starved Niches . . . 114
6.4.3 Bounding the Population Size . . . 115
6.4.4 Experimental Validation of the Models . . . 116
6.5 Occurrence-based Reproduction . . . 119
6.6 Takeover Time of Accurate Classifiers in Starved Niches . . . 120
6.6.1 Conditions for Starved Niches Extinction under Proportionate Selection . 121
6.6.2 Conditions for Starved Niches Extinction under Tournament Selection . . 122
6.7 Reassembling the Theoretical Framework: UCS in Imbalanced Domains . . . 122
6.7.1 Patchquilt Integration: from XCS to UCS . . . 122
6.7.2 Solving Highly Imbalanced Domains with UCS . . . 123
6.8 Summary and Conclusions . . . 125

7 XCS and UCS for Mining Imbalanced Real-World Problems 127
7.1 LCSs in Imbalanced Real-World Problems: What Makes the Difference? . . . 128
7.1.1 XCS and UCS Enhancements to Deal with Continuous Data . . . 128
7.1.2 What Do we Need to Apply the Theory? . . . 130
7.2 Self-Adaptation to Particular Unknown Domains . . . 132
7.2.1 Online Adaptation Algorithms . . . 132
7.2.2 Experiments . . . 134
7.3 LCSs in Imbalanced Real-World Domains . . . 136
7.3.1 Comparison Methodology . . . 136
7.3.2 Results . . . 138
7.4 Re-sampling Techniques . . . 141
7.4.1 Random Over-sampling . . . 142
7.4.2 Under-sampling based on Tomek Links . . . 143
7.4.3 SMOTE . . . 143
7.4.4 cSMOTE . . . 144
7.4.5 What Do Re-sampling Techniques Do? A Case Study . . . 146
7.5 Results on Re-sampled Domains . . . 150

7.5.1 Experimental Methodology 150
7.5.2 Statistical Analysis of the Results 151
7.5.3 Summary 154
7.6 Discussion 155
7.7 Summary and Conclusions 158

8 Fuzzy-UCS: Evolving Fuzzy Rule Sets for Supervised Learning 159
8.1 Why Using Fuzzy Logic in LCSs? 160
8.2 Fuzzy Logics in GBML 162
8.2.1 Fuzzy Logic and Fuzzy Systems 162
8.2.2 Genetic Algorithms in Fuzzy Systems 163
8.2.3 Related Work on Learning Fuzzy-Classifier Systems 165
8.3 Description of Fuzzy-UCS 166
8.3.1 Knowledge Representation 167
8.3.2 Learning Interaction 169
8.3.3 Classifiers Update 170
8.3.4 Classifiers Discovery 171
8.3.5 Fuzzy-UCS in Test Mode 172
8.4 Sensitivity of Fuzzy-UCS to Configuration Parameters 174
8.5 Knowledge Representation and Decision Boundaries 177
8.5.1 Approximate Fuzzy-UCS 179
8.5.2 Decision Boundaries: Study on an Artificial Domain 181
8.5.3 Comparison Between Linguistic and Approximate Representations 185
8.6 Comparison of Fuzzy-UCS to Several Machine Learning Techniques 193
8.6.1 Experimental Methodology 193
8.6.2 Comparison to Fuzzy Rule-Based Classification Systems 194
8.6.3 Comparison with Non-Fuzzy Learners 200
8.7 Fuzzy-UCS for Mining Large Data Sets 208
8.7.1 Data Set Description 209
8.7.2 Results 209
8.8 Summary, Conclusions, and Further Work 211
8.8.1 Summary 211
8.8.2 SWOT Analysis 212

9 Summary, Conclusions, and Further Work 215
9.1 Summary and Conclusions 215

9.2 Lessons from LCSs Design and Application . . . 220
9.3 Further Work . . . 222

A Description of the Artificial Problems **225**
A.1 Parity . . . 225
A.2 Decoder . . . 226
A.3 Position . . . 226
A.4 Multiplexer . . . 227
A.4.1 Imbalanced Multiplexer . . . 228
A.4.2 Multiplexer with Alternating Noise . . . 228

B Statistical Tests **229**
B.1 Statistical Tests for Contrasting Hypotheses . . . 229
B.2 Comparisons of Two Learning Systems . . . 230
B.2.1 The Wilcoxon Signed-Ranks Test . . . 230
B.3 Comparisons of Multiple Classifiers . . . 232
B.3.1 Friedman's Test . . . 233
B.3.2 Post-hoc Nemenyi Test . . . 233
B.3.3 Post-hoc Bonferroni-Dunn Test . . . 235
B.4 Summary . . . 236

C Full Results of the Comparison of the Re-sampling Techniques **237**

D Empirical Analysis of the Sensitivity of Fuzzy-UCS to Configuration Parameters **243**
D.1 Configuration Parameters of Fuzzy-UCS . . . 244
D.2 Experimental Methodology . . . 244
D.3 Fuzzy-UCS's Sensitivity to Configuration Parameters . . . 245
D.3.1 Sensitivity to Rule Initialization . . . 245
D.3.2 Sensitivity to Fitness Pressure . . . 247
D.3.3 Sensitivity to the GA . . . 249
D.3.4 Sensitivity to Deletion . . . 250
D.4 Summary and Conclusions . . . 253

References **255**

Index **279**

List of Figures

2.1 Examples of (a) supervised, (b) unsupervised, and (c) reinforcement learning. . . 13
2.2 Evolution of a GA population. 19
2.3 Simplified schematic of Michigan-style LCSs which the typical process organization. 26
2.4 Simplified schematic of Pittsburgh-style LCSs. 28

3.1 Schematic of the process organization of XCS. 35
3.2 Schematic of the process organization of UCS. 42
3.3 Example of covering in the hyper rectangular representation. 48
3.4 A crossover example. (a) plots the two parents and (b) shows the offspring resulting from two cut points occurring in the middle of each interval. 49
3.5 Example mutation in the hyper rectangle representation. 49

4.1 Proportion of the best action map achieved by UCSns and UCSs in the parity, the position, and the decoder problems. 56
4.2 Proportion of the best action map achieved by (a) UCSns and (b) UCSs in the noisy 20-bit multiplexer with $P_x = \{0.05, 0.10, 0.15\}$. 57
4.3 Proportion of the best action map achieved by (a) UCSns and (b) UCSs in the noisy 20-bit multiplexer with $P_x = \{0.05, 0.10, 0.15\}$ and using $\beta = 0.01$ and $\theta_{GA} = 100$. 59
4.4 Proportion of the best action map achieved by UCSs and XCS in the parity, the position, and the decoder problems. 61
4.5 Proportion of the best action map achieved by (a,c) UCSs and (b,d) XCS in the noisy 20-bit multiplexer with $P_x = \{0.05, 0.10, 0.15\}$ with (a,b) the original configuration and with (c,d) the original configuration but setting $\beta = 0.01$ and $\theta_{GA} = 100$. 62
4.6 Error of XCS's classifiers along the over-general/optimal classifier dimension. The curve depicts how the error of the most over-general classifier `##########:0` evolves as the bits of the classifier are specified, until obtaining the maximally accurate rule `0000000000:0`. 63
4.7 Error of XCS's classifiers along the over-general/optimal classifier dimension. . . 63

5.1 Evolution of (a) the proportion of the optimal population and (b) the product of TP rate and TN rate in the 11-bit multiplexer with imbalance ratios ranging from ir=1 to ir=1024. 72

5.2 Histogram of the error of the most over-general classifier with Widrow-Hoff delta rule at $\beta = 0.2$ and different imbalance ratios. 79

5.3 Histogram of the error of the most over-general classifier with Widrow-Hoff delta rule at $\beta = 0.01$ and different imbalance ratios. 80

5.4 Histogram of the error of the most over-general classifier with gradient descent at $\beta = 0.2$ and different imbalance ratios. 81

5.5 Probability of activating covering on a minority class instance given a certain specificity $\sigma[P]$ and the imbalance ratio ir. The curves have been drawn from equation 5.13 with $\ell = 20$ and different specificities. 83

5.6 Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and the default configuration with (a) Widrow Hoff rule update with adjusted β according to ir and (b) gradient descent parameter update with $\beta = 0.2$. The dots show the empirical results and lines plot linear increases with ir (according to the theory). 87

5.7 Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and different XCS's configurations. The dots show the empirical results and lines plot linear increases with ir (according to the theory). 89

5.8 Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and different XCS's configurations with $\theta_{GA} = n \cdot m \cdot ir$. The points indicate the empirical values of the minimum population size required by XCS. The lines depict the theoretical increase calculated with the previous models, which assumed $\theta_{GA} = 0$. 91

5.9 Takeover time in proportionate selection for (a) m=1, (b) m=2, and (c) m=3 and ρ={0.01,0.10,0.20,0.30,0.40,0.50}. 102

5.10 Takeover time in tournament selection for (a) m=1, (b) m=2, and (c) m=3. 103

5.11 Evolution of (a) the proportion of the optimal population and (b) the product of TP rate and TN rate in the 11-bit multiplexer with imbalance ratios ranging from ir=1 to ir=1024. 106

6.1 Histogram of the error of the most over-general classifier in UCS for $ir = \{1, 10, 100\}$. 112

6.2 Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and the default configuration with (a) tournament selection and (b) roulette wheel selection The dots shows the empirical results and lines plot linear increases with ir (according to the theory). 117

6.3 Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and different UCS's configurations that do not satisfy the initial model assumptions: (a) using 2-point crossover and (b) using the correct set size deletion scheme. The dots shows the empirical results and lines plot linear increases with ir (according to the theory). 118

6.4 Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and different UCS's configurations with $\theta_{GA} = n \cdot m \cdot ir$. The points indicate the empirical values of the minimum population size required by UCS. The lines depict the theoretical increase calculated with the previous models, which assumed $\theta_{GA} = 0$. 119
6.5 Evolution of (a) the proportion of the optimal population and (b) the geometric mean of TP rate and TN rate in the 11-bit multiplexer with ir={1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}. 124

7.1 Example of a domain with two niches (a) and examples of possible representatives of the two niches and over-general classifiers (b) in a two-dimensional problem with continuous attributes 129
7.2 Example of two domains with the same imbalance ratio in the training data set but different niche imbalance ratio. 130
7.3 Example of a domain with oblique boundaries. Several interval-based rules are required to define the class boundary precisely. 132
7.4 Evolution of (a,c) the proportion of the optimal population and (b,d) the geometric mean of TP rate and TN rate of XCS and UCS, respectively, in the 11-bit multiplexer with ir={1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}. 135
7.5 Original domain (a) and domains after applying random over-sampling (b), understampling with Tomek links (c), SMOTE (d), cSMOTE (e). 147
7.6 Models created by C4.5, SMO, IBk, XCS and l'UCS with the original and the re-sampled data sets. 149
7.7 Comparison of the performance obtained by (a) C4.5, (b) SMO, (c) IBk, (d) XCS, and (e) UCS with the different re-sampling techniques. Groups of classifiers that are not significantly different at $\alpha = 0.10$ are connected. 152

8.1 Schematic of GFRBS architecture. 163
8.2 Schematic illustration of Fuzzy-UCS. The run cycle depends on whether the system is under exploration (training) or exploitation (test). 167
8.3 Representation of a fuzzy partition for a variable with (a) three and (b) five triangular-shaped membership functions. 168
8.4 Graphical comparison between (a) linguistic and (b) approximate fuzzy rule sets. 178
8.5 (a) Domain of the tao problem and (b) decision boundaries obtained by UCS. . . 182
8.6 Decision boundaries obtained by linguistic Fuzzy-UCS with weighted average inference and 5 (a), 10 (b), 15 (c) and 20 (d) linguistic terms per variable. 182
8.7 Decision boundaries obtained by linguistic Fuzzy-UCS with action winner inference and (a) 5, (b) 10, (c) 15, (d) and 20 linguistic terms per variable. 183
8.8 Decision boundaries obtained by linguistic Fuzzy-UCS with fittest rules inference and (a) 5, (b) 10, (c) 15, (d) and 20 linguistic terms per variable. 184
8.9 Decision boundaries obtained by approximate Fuzzy-UCS. 184

8.10 Comparison of the training performance of all classifiers against each other with the Nemenyi test. Groups of classifiers that are not significantly different (at $\alpha = 0.10$) are connected. 188

8.11 Comparison of the test performance of all classifiers against each other with the Nemenyi test. Groups of classifiers that are not significantly different (at $\alpha = 0.10$) are connected. 190

8.12 Evolution of the training and test accuracies with approximate Fuzzy-UCS on the *bal* problem. 191

8.13 Comparison of the number of rules evolved by all learners against each other with the Nemenyi test. Groups of classifiers that are not significantly different (at $\alpha = 0.10$) are connected. 191

8.14 Comparisons of one learner against the others with the Bonferroni-Dunn test at a significance level of 0.1. All the learners are compared to three different control groups: (1) Fuzzy-UCS with weighted average inference, (2) Fuzzy-UCS with action winner inference, and (3) Fuzzy-UCS with fittest rules inference. The learners connected are those that perform equivalently to the control learner. . . 196

8.15 Illustration of the significant differences (at $\alpha = 0.05$) of the test accuracy among the fuzzy-methods and Fuzzy-UCS. An edge $L_1 \stackrel{p_{value}}{\rightarrow} L_2$ indicates that the learner L_1 outperforms the learner L_2 with the corresponding p_{value}. To facilitate the visualization, Fuzzy-AdaBoost and Fuzzy MaxLogitBoost, the two most outperformed algorithms, were not included in the graph. 198

8.16 Examples of part of the models evolved by (a) the GP-based methods, i.e., Fuzzy GP, Fuzzy GAP, and Fuzzy SAP; (b) the boosting learners, i.e., Fuzzy AdaBoost, Fuzzy LogitBoost, and Fuzzy MaxLogitBoost; and (c) Fuzzy-UCS for the two-dimensional tao problem. In the fuzzy learners, we used the following five linguistic terms per variable: {XS, S, M, L, XL}. All fuzzy learners use triangular-shaped membership functions. Moreover, GP-based learners also use trapezoid-shaped membership functions. 199

8.17 Illustration of the significant differences (at $\alpha = 0.05$) of the test accuracy among non-fuzzy methods and Fuzzy-UCS. An edge $L_1 \stackrel{p_{value}}{\rightarrow} L_2$ indicates that the learner L_1 outperforms the learner L_2 with the corresponding p_{value}. To facilitate the visualization, ZeroR and SMO with Gaussian kernels, the two most outperformed algorithms, were not included in the graph. 205

8.18 Examples of part of the models evolved by (a) SMO, (b) C4.5, (c) Part, (d) GAssist, (e) UCS, and (f) Fuzzy-UCS for the two-dimensional tao problem. . . . 206

8.19 Evolution of test accuracies and the population size of Fuzzy-UCS with action winner inference in first 50 000 learning iterations of the 1999 KDD Cup data set. 210

B.1 Comparison of the performance of all classifiers against each other with the Nemenyi test. Groups of classifiers that are not significantly different (at $\alpha = 0.10$) are connected. 234

List of Tables

3.1 Example of two-point crossover, in which the two cut points are in the middle of each interval. 48

4.1 Accuracy and fitness of UCSns's classifiers along the generality-specificity dimension, depicted for the parity problem with $\ell = 4$. 58

7.1 Description of the data sets properties. The columns describe the data set identifier (Id.), the original name of the data set (Data set), the number of problem instances (#Ins.), the number of attributes (#At.), the proportion of minority class instances (%Min.), the proportion of majority class instances (%Maj.), and the imbalance ratio (ir). 137

7.2 Comparison of C4.5, SMO, IBk, XCS, and UCS on the 25 real-world problems. Each cells depicts the average value of the product of TP rate and TN rate and the standard deviation. *Avg* gives the performance average of each method over the 25 data sets. The two last rows show the average rank of each learning algorithm (*Rank*) and its position in the ranking (*Pos*). 139

7.3 Comparison of C4.5, SMO, IBk, XCS, and UCS on the 25 real-world problems. For a given problem, the $\bullet$ and $\circ$ symbols indicate that the learning algorithm of the column performed significantly worse/better than another algorithm at 0.95 confidence level (pairwise Wilcoxon signed-ranks test). *Score* counts the number of times that a method performed worse-better, and $Score_{ir>5}$ does the same but only for the highest imbalanced problems ($ir > 5$). 140

7.4 TP rate (TPR) i TN rate (TNR) obtained by C4.5, SMO, IBk, XCS and UCS with the original domain and the re-sampled data sets. 150

7.5 Intra-method ranking for original and re-sampled data sets for C4.5, SMO, IBk, XCS, and UCS. Rows $1st$ to $5th$ indicate the number of times that each re-sampling technique was ranked in the correspondent position. The last column shows the average rank and its standard deviation. 154

8.1 Properties of the data sets. The columns describe: the identifier of the data set (Id.), the name of the data set (dataset), the number of instances (#Ins), the total number of features (#Fea), the number of continuous features (#Cnt), the number of nominal features (#No), the number of classes (#C), the proportion of instances of the minority class (%Min), the proportion of instances of the majority class (%Maj), the proportion of instances with missing values (%MI), and the proportion of features with missing values (%MA). 175

8.2 Configurations used to test the sensitivity of Fuzzy-UCS to configuration parameters. 175

8.3 Comparison of the sensitivity of Fuzzy-UCS to configuration parameters. Each cell shows the average rank of each configuration for a given inference scheme. The best ranked method is in bold. The ⊖ symbol indicates that the corresponding method significantly degrades the results obtained with the best ranked method. 176

8.4 Summary of Fuzzy UCS results with interval-based, approximate and linguistic representation with 5, 10, 15, and 20 linguistic terms per variable in the tao problem. Columns show the training accuracy and the number of rules for action winner and weighted average inference schemes. 185

8.5 Comparison of the training accuracy of linguistic Fuzzy-UCS with weighted average (wavg), action winner (awin), and most numerous and fittest rules inference (nfit), and approximate Fuzzy-UCS on a set of twenty real-world problems. . . . 187

8.6 Pairwise comparisons of the training accuracy achieved by linguistic Fuzzy-UCS with the three types of inference and approximate Fuzzy-UCS. 188

8.7 Comparison of the test accuracy of linguistic Fuzzy-UCS with weighted average (wavg), action winner (awin), and most numerous and fittest rules inference (nfit), and approximate Fuzzy-UCS on a set of twenty real-world problems. 189

8.8 Pairwise comparisons of the test accuracy achieved by linguistic Fuzzy-UCS with the three types of inference and approximate Fuzzy-UCS. 190

8.9 Comparison of the population sizes of linguistic Fuzzy-UCS with weighted average (wavg), action winner (awin), and most numerous and fittest rules inference (nfit), and approximate Fuzzy-UCS on a set of twenty real-world problems. 192

8.10 Pairwise comparisons of the sizes of the rule sets evolved by linguistic Fuzzy-UCS with the three types of inference and approximate Fuzzy-UCS. 192

8.11 Comparison of the test accuracy of Fuzzy-UCS with weighted average (wavg), action winner (awin), and fittest rules (nfit), to Fuzzy GP, Fuzzy GAP, Fuzzy SAP, Fuzzy AdaBoost, Fuzzy LogitBoost, and Fuzzy MaxLogitBoost. 195

8.12 Pairwise comparison of the test accuracy of fuzzy learners Fuzzy GP, Fuzzy GAP, Fuzzy SAP, Fuzzy AdaBoost (ABoost), Fuzzy LogitBoost (LBoost), Fuzzy MaxLogitBoost (MLBoost), and Fuzzy UCS with weighted average inference (wavg), action winner inference (awin), and fittest rules inference (nfit) by means of a Wilcoxon signed-ranks test. 197

8.13 Size of the models evolved by Fuzzy GP, Fuzzy GAP Fuzzy SAP, Fuzzy AdaBoost (ABoost), Fuzzy LogitBoost (LBoost), Fuzzy MaxLogitBoost (MLBoost), and Fuzzy UCS with weighted average inference (wavg), action winner inference (awin), and fittest rules inference (nfit). 201

8.14 Pairwise comparisons of the sizes of the models of Fuzzy GP, Fuzzy GAP, Fuzzy SAP, Fuzzy AdaBoost (ABoost), Fuzzy LogitBoost (LBoost), Fuzzy MaxLogitBoost (MLBoost), and Fuzzy UCS with weighted average inference (wavg), action winner inference (awin), and fittest rules inference (nfit) by means of a Wilcoxon signed-ranks test. 201

8.15 Comparison of the test accuracy of Fuzzy-UCS with weighted average (wavg), action winner (Awin), and fittest rules inference (nfit) to ZeroR (0R), C4.5, IB5, Part, Naïve Bayes (NB), SMO with polynomial kernels of order 3 (SMO_{p3}), SMO with Gaussian kernels (SMO_{rbf}), and GAssist. 203

8.16 Pairwise comparison of the test accuracy of Fuzzy-UCS with weighted average (wavg), action winner (Awin), and fittest rules inference (nfit) to ZeroR (0R), C4.5, IB5, Part, Naïve Bayes (NB), SMO with polynomial kernels of order 3 (SMO_{p3}), SMO with Gaussian kernels (SMO_{rbf}), and GAssist by means of a Wilcoxon signed-ranks test. 204

8.17 Average sizes of the models build by C4.5, Part, GAssist, UCS and Fuzzy-UCS with weighted average (wavg), action winner (awin), and fittest rules inference (nfit). 207

8.18 Properties of the 1999 KDD Cup intrusion detection data set. The columns describe: the identifier of the data set (Id.), the number of instances (#Inst), the total number of features (#Fea), the number of real features (#Re), the number of nominal features (#No), the number of classes (#Cl), the proportion of instances with missing values (%MisInst), and the dispersion of the data set (Disp) computed as #Fea/#Inst. 209

8.19 Test performance and number of rules evolved by Fuzzy-UCS with weighted average (wavg), action winner (awin), and fittest rules (nfit) in the 1999 Kdd Cup intrusion detection data set at different number of learning iterations. 210

8.20 SWOT analysis of Fuzzy-UCS. 212

A.1 Best action map (first and second columns) and complete action map (all columns) of the parity problem with $\ell = k = 4$. 225

A.2 Best action map (first column) and complete action map (all columns) of the decoder problem with $\ell = k = 4$. 226

A.3 Best action map (first column) and complete action map (all columns) of position with ℓ=6. 227

A.4 Best action map (first column) and complete action map (all columns) of the multiplexer problem with $\ell = 6$. 227

B.1 Comparison of the performance of methods M1 and M2 (second and third column). The fourth column provides the performance difference, and the fifth column supplies the rank of the differences. 231

B.2 Comparison of the performance of methods M1, M2, and M3. For each method and data set, the average rank is supplied in parentheses. The last column provides the rank of each learning algorithm for each data set. 232

B.3 Critical values for the two-tailed Nemenyi test. 234

B.4 Critical values for the two-tailed Bonferroni-Dunn test. 235

C.1 Comparison of the performance, measured as the product of TP rate and TN rate, achieved by **C4.5** with the original and re-sampled data sets. For each method and data set, the • and ◦ symbols indicate that the method is statistically inferior/superior than another of the learners according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. *Avg* provides the performance average of each method over the 25 data sets. Rows *Rank* and *Pos* show the average rank of each learning algorithm and its position in the ranking respectively. The last row provides *Inf/Sup*, where *Inf* is the number of times that the learner has been surpassed by another one, and *Sup* is the number of times that the method has outperformed another one. 238

C.2 Comparison of the performance, measured as the product of TP rate and TN rate, achieved by **SMO** with the original and re-sampled data sets. For each method and data set, the • and ◦ symbols indicate that the method is statistically inferior/superior than another of the learners according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. *Avg* provides the performance average of each method over the 25 data sets. Rows *Rank* and *Pos* show the average rank of each learning algorithm and its position in the ranking respectively. The last row provides *Inf/Sup*, where *Inf* is the number of times that the learner has been surpassed by another one, and *Sup* is the number of times that the method has outperformed another one. 239

C.3 Comparison of the performance, measured as the product of TP rate and TN rate, achieved by **IBk** with the original and re-sampled data sets. For each method and data set, the • and ◦ symbols indicate that the method is statistically inferior/superior than another of the learners according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. *Avg* provides the performance average of each method over the 25 datasets. Rows *Rank* and *Pos* show the average rank of each learning algorithm and its position in the ranking respectively. The last row provides *Inf/Sup*, where *Inf* is the number of times that the learner has been surpassed by another one, and *Sup* is the number of times that the method has outperformed another one. 240

C.4 Comparison of the performance, measured as the product of TP rate and TN rate, achieved by **XCS** with the original and re-sampled data sets. For each method and data set, the • and ◦ symbols indicate that the method is statistically inferior/superior than another of the learners according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. *Avg* provides the performance average of each method over the 25 data sets. Rows *Rank* and *Pos* show the average rank of each learning algorithm and its position in the ranking respectively. The last row provides *Inf/Sup*, where *Inf* is the number of times that the learner has been surpassed by another one, and *Sup* is the number of times that the method has outperformed another one. 241

C.5 Comparison of the performance, measured as the product of TP rate and TN rate, achieved by **UCS** with the original and re-sampled data sets. For each method and data set, the • and ◦ symbols indicate that the method is statistically inferior/superior than another of the learners according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. *Avg* provides the performance average of each method over the 25 data sets. Rows *Rank* and *Pos* show the average rank of each learning algorithm and its position in the ranking respectively. The last row provides *Inf/Sup*, where *Inf* is the number of times that the learner has been surpassed by another one, and *Sup* is the number of times that the method has outperformed another one. 242

D.1 Comparison of the test accuracy obtained with the three types of inference and the three configurations which vary $P_{\#}$. *Rank* gives the average rank of each configuration for each one of the three inference schemes. *Pos* shows the absolute position in the ranking. *Frd* reports the p-value obtained with the multiple-comparison Friedman test performed for each inference methodology. 246

D.2 Comparison of the model sizes obtained with the three types of inference and the three configurations which vary $P_{\#}$. *Rank* gives the average rank of each configuration for each one of the three inference schemes. *Pos* shows the absolute position in the ranking. *Frd* reports the p-value obtained with the multiple-comparison Friedman test performed for each inference methodology. 246

D.3 Comparison of the test accuracy obtained with the three types of inference and the three configurations which vary the fitness pressure ν. *Rank* gives the average rank of each configuration for each one of the three inference schemes. *Pos* shows the absolute position in the ranking. *Frd* reports the p-value obtained with the multiple-comparison Friedman test performed for each inference methodology. . 248

D.4 Comparison of the model sizes obtained with the three types of inference and the three configurations which vary the fitness pressure ν. *Rank* gives the average rank of each configuration for each one of the three inference schemes. *Pos* shows the absolute position in the ranking. *Frd* reports the p-value obtained with the multiple-comparison Friedman test performed for each inference methodology. . 248

D.5 Comparison of the test accuracy obtained with the three types of inference and the five configurations varying θ_{GA}, θ_{del}, and θ_{sub}. *Rank* gives the average rank of each configuration for each one of the three inference schemes. *Pos* shows the absolute position in the ranking. *Frd* reports the p-value obtained with the multiple-comparison Friedman test performed for each inference methodology. . 250

D.6 Comparison of the model sizes obtained with the three types of inference and the five configurations varying θ_{GA}, θ_{del}, and θ_{sub}. *Rank* gives the average rank of each configuration for each one of the three inference schemes. *Pos* shows the absolute position in the ranking. *Frd* reports the p-value obtained with the multiple-comparison Friedman test performed for each inference methodology. . 251

D.7 Pairwise comparison of the test accuracy of Fuzzy-UCS obtained with the three types of inference and the five configurations varying θ_{GA}, θ_{del}, and θ_{sub} by means of a Wilcoxon signed-ranks test. 251

D.8 Comparison of the test accuracy obtained with the three types of inference and the two configurations which vary the deletion pressure δ. *Rank* gives the average rank of each configuration for each one of the three inference schemes. *Pos* shows the absolute position in the ranking. *PW* reports the p-value obtained with the pairwise Wilcoxon signed-ranks test performed for each inference methodology. . 252

D.9 Comparison of the model sizes obtained with the three types of inference and the three configurations which vary the deletion pressure δ. *Rank* gives the average rank of each configuration for each one of the three inference schemes. *Pos* shows the absolute position in the ranking. *PW* reports the p-value obtained with the pairwise Wilcoxon signed-ranks test performed for each inference methodology. . 252

List of Algorithms

2.2.1 Pseudo code of a simple GA. 21

7.2.1 Pseudo code for the *online adaptation algorithm* in XCS. 133

7.2.2 Pseudo code for the on-line adaptation of β. 133

7.2.3 Pseudo code for the *online adaptation algorithm* in UCS. 134

7.4.1 Pseudo code for the Tomek Links algorithm. 142

7.4.2 Pseudo code for the SMOTE algorithm. 144

7.4.3 Pseudo code for the cSMOTE algorithm. 145

Chapter 1

Introduction

In the last few decades, there has been increasing interest in *machine learning* (Mitchell, 1997, 2006; Nilson, 2005; Bishop, 2007), a field in artificial intelligence (Feigenbaum and Feldman, 1995; Brooks, 1990; Russell and Norvig, 2002; McCharthy, 2007) that is concerned with the development of machines that can learn from the experience. The appeal of machine learning is based on the idea of having computers that teach themselves to solve new challenging problems instead of programming them with a deterministic behavior. This approach appears to be especially attractive in applications (1) that are too complex for human experts to manually design and implement the solver, or (2) that require the software to improve or refine itself continuously. Therefore, the ultimate aim of machine learning is to solve problems that are too complex for human beings to solve or to give instructions on how to solve them.

Machine learning does not only aim at solving engineering problems, but it is also closely related to different fields such as logic and philosophy, theoretical computer science, statistics, biology, experimental psychology, cognitive science, and communication theory among others (Buchanan, 2005). For example, several machine learning techniques derive from works of psychologists that try to understand animal and human behavior through computational modeling. Similarly, machine learning research and biological learning are related, and research in each area benefits from the other one resulting in a fruitful crossbreeding among the techniques studied in the two areas. Thence, machine learning holds promise not only in empowering computers so that they can solve new challenging problems, but also in providing a framework to study artificial intelligence.

One of the most appealing machine learning techniques is *learning classifier systems* (LCSs), whose theoretical foundation was early outlined by Holland (1962). With the purpose of creating true artificial intelligence itself, Holland (1971, 1976) envisioned LCSs as cognitive systems that received perceptions from their environment and, in response to these perceptions, performed actions in the real world to achieve certain goals; besides, the policy of these programs *evolved* with their interaction with the changing environment, refining the policies with the aim of achieving a maximum reward—which was aligned to the goals of the system—while adapting to the changes found in the environment. Since the first definition by Holland, there has been an augmenting research on LCSs, which has led to design new systems and to solve increasingly complex and challenging problems. Currently, LCSs are competent machine learning techniques that are able to solve hard problems that range in different disciplines. Nevertheless, some chal-

lenges, most of the times shared by the machine learning community, still need to be addressed to scalably and efficiently solve new complex real-world problems.

This thesis is concerned about advancing in the research on LCSs to gain a better understanding of their behavior and to improve them to deal with current problems in science, engineering, and industry. Further, we elaborate the framework of this thesis in more detail, providing some historical remarks on the LCSs's research. Taking the original definition of Holland, we follow the contributions that have led to the current LCSs' architectures. Then, we identify two important challenges in LCSs and machine learning systems alike, which are later articulated in the objectives of this work. Finally, we provide the road map of the entire document.

1.1 Framework: From Holland's Definition to Current LCSs

Holland (1971, 1976) proposed the original idea of LCSs as cognitive systems that infer environmental patterns from experience and associate appropriate response sequences with them. The initial schemas of learning classifier systems already pointed out three key aspects, which have been preserved up to the most recent implementations: (1) a knowledge representation based on *classifiers*—usually implemented as production rules—that enables the system to map sensorial states with actions, (2) an *apportionment of credit algorithm* which shares the credit obtained by the machine among classifiers, (3) an algorithm to *evolve* the knowledge base—typically, a *genetic algorithm* (GA) (Holland, 1975). Since the first definition of LCS, more than 30 years ago, several systems have been designed following Holland's initial definition; also, slightly different angles of the same problem resulted in different types of LCSs, shaping the LCS branches that currently exist. As proceeds, we review some of the most important aspects of these 30 years of history.

The first successful implementation of LCSs was the cognitive system one (CS-1) by Holland and Reitman (1978), which was designed to imitate animal behavior. The goal of the system was to satisfy its needs by means of obtaining a finite number of *resources* that were maintained in different *reservoirs*. CS-1 acted in a stimulus-response way; given each sensorial input, the machine performed an action to the environment, which in turn responded with a reward. The system implemented the three aforementioned key aspects in the following way. CS-1 used simple string rules to code the internal policy. An *epochal apportionment of credit system* was employed to share the resources among rules. This algorithm tracked the utility of rules during an epoch—that is, a certain time in which payoff events were received—and, at the end of the epoch, distributed the payoff according to the value of each rule. Learning was accomplished through a GA. CS-1 was faced to two maze-running tasks in which different units of food and water were placed around the maze. The experimental results showed that CS-1 could evolve an accurate policy to reach the system goals, highlighting that LCSs held promise for machine learning.

Subsequently to the development of CS-1, several authors continued on the design and implementation of new LCSs based on Holland's original ideas. Booker (1982) adopted an LCS with an architecture inherited from CS-1 to study the connections between LCSs and cognitive science. The system was tested on environments where the classifier wandered in a feature space with the aim of avoiding aversive stimuli (poison) and reaching attractive stimuli (food). Contemporaneous with this work, Wilson (1981, 1985a) proposed an LCS for the sensory-motor

coordination on movable video camera, addressed as the EYE-EYE system. Later, Wilson (1985b) simplified the LCSs architecture and applied the system to the modeling of an artificial animal—i.e., the *animat* problem. Thereafter, to investigate further on the system, Wilson (1987) simplified the learning task by designing a non-sequential problem that provided immediate reward at each learning iteration. Approximately together with the development of these works, Goldberg applied LCSs to the learning control of a simulated gas pipeline (Goldberg, 1983, 1985a,b, 1987a,b).

In parallel to Holland's work on CS-1 and its derivations, Smith (1980) took a different approach and developed a new type of LCSs. Regarding learning as an *adaptive search procedure*, Smith raised the focus of operation one notch and developed the learning system 1 (LS-1) (Smith, 1980, 1983, 1984), a system that used a GA to evolve rule sets instead of evolving individual classifiers or rules as in Holland's approach. Therefore, genetic manipulation worked at rule set level instead of at rule level. This permitted sidestepping the apportionment of credit algorithm. That is, as the rules belonged to a set, the need to compute the individual contribution of each rule to the whole model disappeared. This LCS model was furthered in new implementations (Jong et al., 1993; Janikow, 1993). Therefore, CS-1 and LS-1 defined two different ways to perform learning by means of GAs. After some years of development, the algorithms resulting from both approaches were distinguished and addressed with different names: Holland's LCSs were referred to as Michigan-style LCSs, whilst Smith's LCSs were addressed as Pittsburgh-style LCSs. In this thesis, we focus our research on Michigan-style LCS.

Despite these promising designs and first applications on attractive problems, LCSs did not reach a general acceptance in the machine learning community, probably due to their lack of mathematical foundation and their restricted applications. Consequently, after a period of strong research in the late 1970s and early 1980s, the late 1980s were known as the LCSs winter, in which the problems detected in the first LCSs seemed to cloud the whole field. At the end of the 1980s, Wilson and Goldberg (1989) published a critical review of LCSs, identifying the critical factors and problems that hindered the success of the LCSs of that time. These problems were associated with (1) the difficulties of distributing credit among the rules, (2) the inadequacy of the decision-making process and the tendency to produce over-general rules, and (3) the limits of the classifier syntax. Besides, the authors also pointed out the need for theory, such as population size models, to gain a better understanding of how these systems worked.

Some years after the critical review, Wilson (1994) first presented the *zeroth-level classifier system* (ZCS) and, one year after, Wilson (1995) proposed the *extended classifier system* (XCS), heralding the second spring and summer of the LCSs field. XCS came to give answer to most of the key problems that were identified in previous LCSs. XCS introduced a new credit apportionment algorithm—which was adapted from a well-known reinforcement learning technique (Sutton and Barto, 1998), Q-learning (Watkins, 1989)—to solve the difficulties in distributing credit. The tendency of producing a large number of over-general classifiers was corrected by (1) basing the classifier fitness on the accuracy of the prediction instead of the prediction itself and (2) using appropriate *niching techniques* and *fitness-sharing schemes*. Besides, the system architecture was simplified with respect to the initial LCSs' architecture. Already in the original paper, XCS was shown to be able to evolve accurate models for single step tasks—in particular, the multiplexer problem (Wilson, 1987)— and to learn optimal policies in maze-running environments, problems that previously eluded solution.

The first publication of XCS promoted an increasing amount of research in the LCSs area, resulting in the so-called LCSs renaissance. Ad hoc with the implementation of XCS, there have been crucial advances in (1) enhancements of the learning architecture and design of new operators, (2) theoretical analyses for design, and (3) applications in important domains. With respect to the learning architecture, the original scheme of XCS was first refined by Wilson (1998) and later by Kovacs (1999), resulting in the standard XCS scheme currently used. Also, there have been notorious works on inclusion of new knowledge representations (Wilson, 2000, 2001, 2008; Lanzi, 1999a; Lanzi and Perrucci, 1999; Bull and O'Hara, 2002; Butz et al., 2008), analyses and improvements of the credit apportionment algorithm (Butz et al., 2005a; Drugowitsch and Barry, 2008), and enhancements of some genetic operators (Butz et al., 2005b,c). Second, there have been several theoretical analyses that enabled a better understanding of the system (Butz and Pelikan, 2001; Butz et al., 2004b, 2005a, 2007; Drugowitsch and Barry, 2008; Drugowitsch, 2008). Finally, XCS and similar systems have been applied to important applications such as data mining (Bull, 2004; Bull et al., 2008), function approximation (Wilson, 2002b; Butz et al., 2008), reinforcement learning (Lanzi, 1999b, 2002; Lanzi et al., 2005; Butz et al., 2005a), and clustering (Tamee et al., 2006, 2007), demonstrating the competitiveness of LCSs, and XCS in particular, with respect to other machine learning techniques from other paradigms such as decision trees (Quinlan, 1995) or neural networks (Widrow and Lehr, 1990). Besides, Michigan-style LCSs provide a competitive advantage with respect to other machine learning techniques: they evolve the knowledge online from a stream of examples. Therefore, the data can be made available in streams, which is very common in current industrial applications where large volumes of data are generated online (Aggarwal, 2007; Gama and Gaber, 2007).

Along with the application of XCS to important domains, there have been some proposals in which the learning architecture of XCS has been modified for specific types of tasks (Wilson, 2002b; Bernadó-Mansilla and Garrell, 2003; Bull, 2005; Tamee et al., 2006). In the particular case of data classification tasks, Butz et al. (2003) detected that XCS produced a deceptive pressure toward the optimal solution in some specific problems. In order to overcome this problem, Bernadó-Mansilla and Garrell (2003) proposed the *supervised classifier system* (UCS), a system that inherits the main components of XCS, but specializes them for supervised learning—specifically, for classification tasks. The advantages of the new architecture with respect to UCS were analyzed on a set of boundedly difficult problems—problems in which the complexity along different dimensions can be controlled—, illustrating that the system could overcome the detected problems and solve complex applications more efficiently than XCS. Nonetheless, UCS is still young and, as pointed out by Bernadó-Mansilla and Garrell (2003), there are some open issues that still need to be addressed to enhance the system and gain a better comprehension of the implications of the changes introduced to the original XCS.

In summary, starting from Holland's idea of creating true artificial intelligence, during the last decade, research on LCSs has been enjoying a renaissance, which has been mainly promoted by the creation of XCS. Currently, LCSs have reached a mature state and are ready to face new challenging problems in machine learning. Furthermore, Michigan-style LCSs have two main assets that distinguish them from machine learning techniques alike:

1. They have a flexible knowledge representation that can be easily adapted to deal with new types of data.

2. They build the model online, which is crucial to succeed in problems with large volumes

of data or where the data is made available in data streams.

For this reason, the present work focuses on LCSs as promising alternatives for machine learning. Despite the recent improvements and applications proposed in the field of LCSs, there are still important challenges—which are not particular to LCSs, but shared by the machine learning community in general—that need to be addressed to scalably and efficiently solve real-world problems. In the following section, we identify the two key challenges in the machine learning and LCSs community from which we define the objectives of this thesis.

1.2 Two Critical Challenges in LCSs and Machine Learning

Research on machine learning has resulted in the design of several learning techniques, such as LCSs, that can extract accurate models from the experience. Due to the maturity of the area, the machine learning community has started to address new important challenges that appear when applying learning techniques to real-world problems. Among the different research lines, the following two key challenges have received especial attention:

1. Learning from domains that contain rare classes.
2. Building more understandable models and bringing reasoning mechanisms closer to human ones.

A more detailed discussion of why these two items represent a critical challenge not only for LCSs but for machine learning in general is provided as follows.

Learning from domains that contain rare classes. The advances in machine learning have led to the application of learning algorithms to new complex real-world problems—for which humans cannot provide an accurate solution—with the aim of extracting novel, interesting, and useful knowledge. It has been identified that, in these types of problems, the key knowledge usually is hidden in examples that are rare in nature (Chan and Stolfo, 1998; den Bosch et al., 1997; Grzymala-Busse et al., 2000; Kubat et al., 1998). In fact, for this reason it is too complex for human beings to identify this key, hidden knowledge. Empirical studies have shown that traditional machine learning techniques may not be able to extract critical information from these rarities. Therefore, a new field has emerged with the aim of creating new approaches to enhance the extraction of the key knowledge from rare classes. The problem of modeling rare classes has taken several names such as the problem of *mining rarities* (Weiss, 2004), the problem with the *small disjuncts*, (Holte et al., 1989) or the *class-imbalance* problem (Japkowicz and Stephen, 2002). In the three of them, the goal is the same: model patterns or examples that occur infrequently accurately.

While this critical problem has been widely studied in the context of traditional machine learning techniques—which learn from collections of static data—little research has been conducted on online learners and, specifically, on LCSs. Two main reasons explain this lack of analyses in LCS. First, as mentioned in the previous section, the first successful LCS's architecture was designed in 1995, and most of the research conducted during the last decade has been centered on the analysis and improvement of this architecture. This has resulted in mature LCSs

that are now ready to face new challenges. On the other hand, the first studies of the *small disjuncts* problem started in the late 1980s (for example, see (Holte et al., 1989)). Therefore, the problem of rare classes has received much more attention in traditional learning techniques than in LCSs. Second, learning from rare classes poses more complex challenges to LCSs since they have an online architecture that learns from a stream of examples. That is, the online system receives a stream of examples, and it has to learn from the upcoming rarities on the fly. Besides, due to this online architecture, techniques developed for traditional machine learning techniques cannot be applied to LCSs. Note that the study and improvement of LCSs to extract accurate models from rarities that come infrequently in a stream of examples appears to be a crucial task to address, accurately and efficiently, the new problems that are more often presented to machine learning techniques.

Building more understandable models and bringing reasoning mechanisms closer to human ones. Besides extracting accurate models from rare classes, a second important challenge in machine learning is to build learning techniques that represent the knowledge and apply reasoning mechanisms that are similar to the human ones. This point is especially important in classification tasks where human experts may need explanations about the decisions taken by the systems. For example, in medical domains, human experts are sometimes more interested in the explanation that yields a prediction than in the prediction itself (Robnik-Sikonja et al., 2003). As proceeds, we discuss why LCSs may evolve poorly interpretable models and present fuzzy logic as a competent approach to create highly legible models.

Michigan-style LCSs evolve models that consist of classifiers—typically rules—which can be individually interpreted by human experts. Nevertheless, it has been detected that Michigan-style LCSs evolve a large number of rules when dealing with problems that have continuous-valued attributes (Bernadó-Mansilla and Ho, 2005; Bacardit and Butz, 2004; Wilson, 2002a; Dixon et al., 2004; Fu et al., 2001), which can be found usually in real-world problems. Besides, the reasoning mechanisms of LCSs may not be natural for human experts. Until recently, few alternatives of new reasoning mechanisms, as well as the first pieces of theory that explain how they work, have been developed for some particular LCSs (Brown et al., 2007). Despite these first promising results, more research needs to be conducted to approach reasoning mechanisms to human reasoning.

Contemporaneous with the recent advances on LCSs, there has been a strong research on fuzzy systems, that is, systems that use fuzzy logic (Zadeh, 1965, 1973) to create highly legible models from environments with uncertainty and imprecision. Essentially, the fuzzy set theory provides a robust reasoning mechanism that approaches human reasoning. Therefore, the combination of fuzzy systems with LCSs appears as an appealing alternative to improve their explicative capabilities. As a consequence, the first attempts to mix both disciplines have been taken (Valenzuela-Rendón, 1991; Nomura et al., 1998; Parodi and Bonelli, 1993; Furuhashi et al., 1994; Velasco, 1998; Ishibuchi et al., 1999b; Casillas et al., 2007); but, so far, no competitive Michigan-style LCSs that creates fuzzy classification models online from streams of examples and uses fuzzy reasoning mechanisms have been designed.

Therefore, the scope of this thesis is to address these two challenges in the context of LCSs. Specifically, we consider XCS and UCS as starting point. We select XCS since it is, by far, the most influential Michigan-style LCS, representing the state of the art in the LCS field. Besides, we incorporate UCS since it was specifically designed for supervised learning, and this work

is especially concerned with classification problems. With these two challenges in mind, the following section explicitly articulates the objectives of this thesis.

1.3 Thesis Objectives

The general goal of the present work is to address the two aforementioned key challenges with LCSs, particularly focusing on XCS and UCS. As stated in the previous sections, UCS is a young promising system derived from XCS that, although having shown to be competitive with respect to other machine learning techniques, still has some open issues that have to be addressed before applying it to new complex problems. Thus, before approaching the two particular challenges defined in the previous section, we first study the UCS classifier system in detail and update its architecture. Furthermore, to understand its differences with XCS, we empirically compare both systems on a set of boundedly difficult problems. Thereafter, we take XCS and the revised version of UCS as a departure point to analyze and improve LCSs to model rare classes accurately. For this reason, we propose to follow an analytical approach to study the LCSs' behavior on problems with rare classes, improve the systems, and apply them to real-world problems with class imbalances. Furthermore, we also propose to include fuzzy logic in an LCS architecture to bring the reasoning mechanisms closer to the human's ones. Specifically, this leads to the definition of the following four objectives:

1. Revise and update UCS and compare it with XCS.
2. Analyze and improve LCSs for mining rarities.
3. Apply LCSs for extracting models from real-world classification problems with rarities.
4. Design and implement an LCS with fuzzy logic reasoning for supervised learning.

As follows, each one of the four objectives is elaborated in detail.

Revise and update UCS and compare it with XCS. Whereas XCS has received an increasing amount of attention during the last decade, resulting in many improvements in the architecture, UCS is still a young system which has received no further modifications since its initial design. Nevertheless, Bernadó-Mansilla and Garrell (2003) detected some critical aspects that needed to be investigated in more detail. The most important one was the lack of *fitness sharing* in the credit apportionment algorithm. That is, differently from almost all the current Michigan-style LCSs, the rules in UCS are evaluated independent of the remaining rules in the population. Bernadó-Mansilla and Garrell (2003) took this approach since the benefits of a credit apportionment algorithm that shared the fitness among individuals were not clearly identified, and further analysis on sharing algorithms was pointed out as an important future work line. Therefore, the first objective of the thesis is to design a fitness-sharing scheme similar to those proposed by GAs and XCS, introduce it to UCS, and analyze the advantages that the new credit apportionment algorithm provides to UCS. We also aim at analyzing the differences between UCS and XCS on supervised learning problems.

Analyze and improve LCSs for mining rarities. In this second objective, we address the challenge of extracting accurate models from domains that contain rare classes with LCSs—in

particular, with XCS and UCS. Specifically, the goal is to study the intrinsic capacities of both LCSs to learn from rare classes, identifying critical factors for the success of the systems. For this purpose, we propose to use *design decomposition* (Goldberg, 2002) to separate the problem of learning from domains with rarities in several critical elements and to derive *facetwise models* for each element. The integration of these models would permit us to draw the domain of applicability of both LCSs and to extract critical bounds on their behavior on imbalanced domains. Moreover, we aim at extracting lessons from the analysis that help improve the systems and enable them to extract accurate models from problems with rare classes that currently elude solution.

Apply LCSs for extracting models from real-world classification problems with rarities. After studying and improving LCSs for mining rarities, we aim at applying both LCSs to a set of real-world classification problems with rare classes. To analyze their performance, we propose to compare the accuracy of the models evolved by the two LCSs with the accuracy of the models created by several highly influential machine learning techniques. As we seek to extract highly accurate models, we also propose to include and analyze the impact of some of the most known *re-sampling techniques* (Japkowicz and Stephen, 2002; Chawla et al., 2002; Batista et al., 2004), that is, pre-processing methods that try to remove rare classes from the original data sets.

Design and implement an LCS with fuzzy logic reasoning for supervised learning. In the last objective of this thesis, we address the second aforementioned challenge and take an inventiveness approach to mix the ideas of the fuzzy systems and the LCSs fields. That is, we purpose to create a hybrid system that mixes the best characteristics of LCSs—as accurate online classifier's evaluators—, GAs—as robust search mechanisms—, and fuzzy logic—as a human-like approach to represent the knowledge and to reason for decision making.

Each one of these objectives gets, at least, a chapter of the present thesis. The overall structure of the document is provided in the following section.

1.4 Road Map

This thesis is organized, in addition to the present chapter, in eight chapters whose content is introduced in what follows.

Chapter 2 starts with a concise introduction to machine learning and to the types of problems that we can find in this discipline, which is followed by an introduction to evolutionary computation. This gives way to the presentation of the current LCS families and to what we currently understand as GBML. That is, while in the present chapter we have provided a brief history of LCSs, in chapter 2 we review the current branches, draw a big picture of different learning methodologies that use evolutionary algorithms, and place LCSs in this picture.

Chapter 3 provides a detailed explanation of both XCS and UCS, which can be used as implementation guidelines. Thence, chapters 2 and 3 give all the background material that is necessary to start with the contributions of this work. Thus, each of the subsequent chapters focuses on one of the objectives of the thesis.

Chapter 4 reviews UCS and updates the system with a new fitness-sharing scheme. Then, UCS with fitness sharing is empirically compared with the original UCS on a set of four artificial problems that have different complexities that are usually present in real-world problems. Therefore, the comparison enables us to highlight the benefits of having a credit apportionment algorithm that shares fitness in UCS. Later, we also introduce XCS in the comparison with the aim of emphasizing the advantages that the modifications introduced by UCS supply in problems with certain characteristics.

Chapter 5 starts with the study of how LCSs can learn from domains that contain rare classes. Although the chapter is focused on XCS, we first take a general approach and intuitively analyze the problems that may arise in a general LCS architecture when learning from rare classes. With this intuition in mind, we decompose the problem and identify five elements that need to be satisfied by any LCS and systems alike to learn, efficiently and scalably, from domains with rare classes. Thence, we create a general framework from learning from class-imbalanced problems without getting tied to any particular LCS architecture. Subsequently, we look at the particular architecture of XCS and derive facetwise models that explain each one of the different elements. During the analysis of each facet, we assume that the remaining facets behave in an ideal manner. The integration of all these models enables us to draw the domain of applicability of XCS, detecting the sweet spot where XCS can efficiently extract accurate models from instances that come infrequently. At the end of the chapter, we show that the lessons extracted from the analysis enable us to solve problems with infrequent classes that previously eluded solution.

Chapter 6 carries over the facetwise analysis to UCS. We start reviewing the general framework proposed in the previous chapter and analyze the components that UCS changes with respect to XCS. We derive new models for these components and plug them into the initial framework, thus, adapting the domain of applicability to UCS. Lastly, we show that UCS has similar learning capabilities to XCS in imbalanced domains.

Chapter 7 moves the theory developed in the previous two chapters to real-world problems, in which the characteristics of the domains are not known. We design two heuristic procedures that gather information from the population evolution and self-adapt XCS and UCS according to the lessons learned from the theory. Then, we test both LCSs on a collection of real-world problems that contain rare classes. To evaluate the performance of both systems, we compare them to three of the most influential machine learning techniques. Later, we introduce pre-processing techniques that try to remove the rare classes by changing the distribution of the training examples and analyze the impact of applying these techniques in combination with each one of the five learners.

Chapter 8 presents Fuzzy-UCS, a hybrid between LCSs, GAs, and fuzzy systems. Fuzzy-UCS is inspired by UCS, but includes a fuzzy representation and usual reasoning mechanisms in fuzzy systems, which approach human reasoning. This chapter performs a large experimentation to show the excellence of Fuzzy-UCS with respect to other machine learning techniques in data classification tasks. We compare Fuzzy-UCS with several top-notch fuzzy learners and show that Fuzzy-UCS outperforms them all. Moreover, we also compare the system to some the most influential non-fuzzy learners. Fuzzy-UCS appears to be, at least, as accurate as the best performer among the tested learners. Besides, we show that the models evolved by Fuzzy-UCS

are clearly more interpretable than those created by UCS. Finally, we finish the chapter by demonstrating the value of Fuzzy-UCS to mine large volumes of data. We use Fuzzy-UCS to evolve classification models from the data provided in the KDD'99 cup intrusion detection data set (Hettich and Bay, 1999). The data set consists of 494 022 instances, 21 classes, and 41 attributes. It is worth noting that most of the learners used in the comparison are not able to learn from such a large data set. The online architecture of LCSs enables Fuzzy-UCS to deal with this large amount of data.

Chapter 9 finishes with the contributions of this thesis by summarizing, providing key conclusions, reviewing the main lessons extracted from this work, and gathering future work lines.

The material presented in the nine chapters is complemented with four appendices. **Appendix A** describes all the boundedly difficult problems used along the experiments of the thesis. **Appendix B** gives details about the statistic tests employed in different chapters to compare results. **Appendix C** supplies the detailed tables of results of the comparison of several machine learning techniques with LCSs on a collection of real-world problems with rare classes performed in chapter 7. Finally, **Appendix D** provides an analysis of the sensitivity of Fuzzy-UCS to its configuration parameters.

Chapter 2

Machine Learning with Learning Classifier Systems

This chapter provides a brief introduction to *learning classifier systems* (LCSs) as one of the most appealing alternatives for *machine learning* (ML). The chapter starts with a concise definition of machine learning, and then, presents a task-oriented taxonomy of ML techniques which divides ML methods—depending on the type of problems that they can solve—in supervised learning, unsupervised learning, and reinforcement learning techniques. Next, *evolutionary computation*—a field of study devoted to the design and implementation of problem solvers inspired by principles of natural evolution and genetics—is briefly introduced. This introduction is followed by a more detailed explanation of *genetic algorithms*, one of the most promising techniques in evolutionary computation, since they guide the discovery process in LCSs. Finally, the current branches or families of algorithms that use GAs for machine learning—usually addressed as *genetic-based machine learning* systems (GBML)—are presented, identifying both Pittsburgh- and Michigan-style LCSs in this taxonomy. For each one of these families, a picture of their process organization is provided, and the main differences among them are discussed.

2.1 Machine Learning

Machine learning is concerned with the design of computer programs that are able to learn from the experience and with the definition of the fundamental laws that govern all learning processes (Mitchell, 1997, 2006; Nilson, 2005; Bishop, 2007). This definition covers a large variety of learning tasks such as (1) the design of goal-oriented agents that learn behavioral policies from their interaction with the real world, (2) the extraction of frequent, interesting patterns from large volumes of plain data generated, for example, from industrial processes, and (3) the modeling of specific domains from examples. As a unified vision of all these learning tasks, Mitchell (2006) considers that a machine *learns* with respect to a certain task T, a performance metric P, and a type of experience E, if the system reliably improves its performance P at task T by following the experience E. Therefore, any application that falls under this definition can be considered as a machine learning method. With this broad definition in mind, this section discusses the relationship of ML with other scientific fields as well as the necessity of having computers that use their own experience to program themselves.

ML is not an isolated discipline, but it is closely related to other fields such as *computer science*, *statistics*, and *psychology* and *neuroscience*. On the one hand, computer science and ML share a common objective; that is, they are concerned about building machines that can solve problems. However, the main difference is that computer science is mainly focused on building deterministic programs, while ML aims at creating programs that learn by themselves. On the other hand, statistics and ML share the common goal of inferring patterns, behaviors, or conclusions from data. Nevertheless, the key difference between both resides in the fact that ML involves additional questions, such as algorithmic scalability, that aim at the creation of machines that can efficiently, accurately, and scalably deal with complex real-world problems. In addition, ML is closely related to the study of human and animal learning in fields such as psychology and neuroscience. For example, several ML techniques derive from works of psychologists that try to understand animal and human behavior through computational modeling. Similarly, ML and biological learning are related, and research in each area benefits the other one, resulting in fruitful crossbreeding of ideas and techniques.

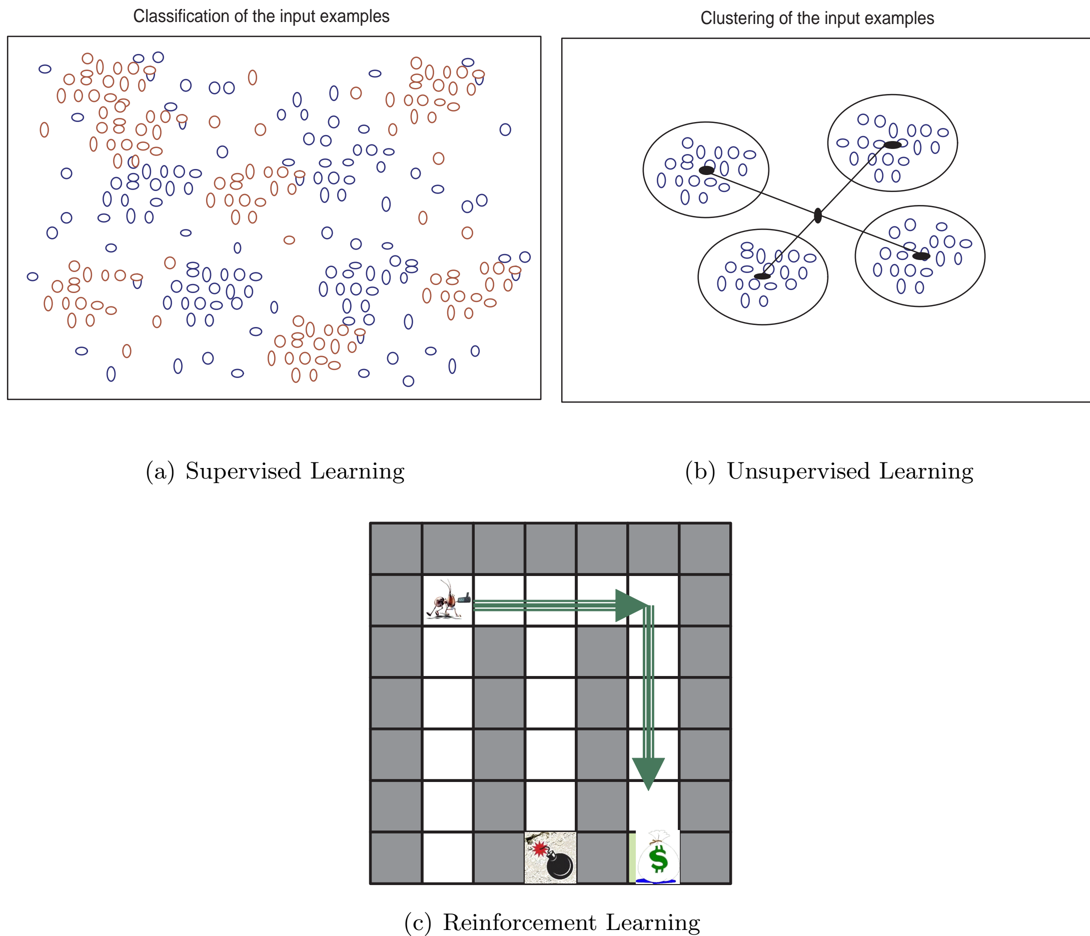
Of course, the idea of having computers teaching themselves is not easy to implement in practice. Therefore, the question that arises is why we should spend efforts in building machines that learn to solve new complex problems instead of relying on humans to code hard-wired solutions for these problems. The need for further research on ML can be explained with two main reasons. From the pure learning point of view, ML can help understand animal and human learning processes, thus providing key insights to psychologist and neuroscientists. From a pure engineering point of view, ML has already provided, and is expected to supply, efficient algorithms to solve new challenging, complex engineering problems whose solution is not known. More specifically, the most important reasons that may lead to the application of ML to solve engineering problems are enumerated as follows.

1. Difficulty of human experts to describe the problem and manually design an algorithm to solve it.
2. Necessity of programs that continuously adapt to changing environments.
3. Necessity of processing overwhelming volumes of data with hidden concepts.

Below, each item is elaborated in more detail.

The application of ML is mandatory when the problem is too complex to manually design and code an algorithm to address it properly. These types of complex problems abound in engineering, having some specific examples in speech recognition (Karat et al., 2003), or computer vision (Jahne et al., 1999). For instance, recognizing faces is a simple task for humans, but manually programming a system to perform this task is too complex. Nevertheless, collecting some examples and training a computer vision program that recognizes these objects—with a certain accuracy—is a more straightforward manner of facing the problem.

Another reason that makes the use of ML necessary is in changing environments. Independent of whether we can provide an initial solution to the problem, the system may need to adapt to changing situations. For example, in speech recognition systems, the program can be provided with an initial speaker-independent voice-recognition system, plus a learning system that adapts to the characteristics of each particular person. Other examples where adaptation is necessary can be found in robot control.



(a) Supervised Learning (b) Unsupervised Learning

(c) Reinforcement Learning

Figure 2.1: Examples of (a) supervised, (b) unsupervised, and (c) reinforcement learning.

The last reason that may lead us to the application of ML is when overwhelming volumes of data need to be processed to extract novel, interesting, and useful knowledge from patterns hidden in these data. Actually, this is a definition of *data mining* (Frawley et al., 1992). In this case, ML can be applied to build programs that use heuristics to extract potentially interesting and novel patterns from the data.

Therefore, ML gathers a large variety of techniques, and several classification criteria can be used to group them in different families. As follows, we present a classic taxonomy of ML techniques that is based on the task to perform. New trends in ML may incorporate new groups or subgroups to this taxonomy; nonetheless, the provided taxonomy gives the three fundamental types of learning: *supervised learning*, *unsupervised learning*, and *reinforcement learning*.

2.1.1 Supervised learning

Supervised learning is the process of extracting a *function* or *model* that maps the relation between a set of descriptive input attributes and one or several output attributes. Depending on the type of output attributes, supervised learning can be further classified as *data classification* or *data regression*. That is, for categorical output attributes (which represent the classes of the examples), the task is addressed as *data classification*; in this case, the goal is to find a model that predicts the class of new instances. Otherwise, for continuous output attributes, the problem is referred to as *data regression*; thence, the goal is to find a function that predicts the output value of new instances. Thus, in general, a supervised learner has to build a function or model that predicts the output value for any valid input object by means of generalizing from the known data.

As follows, we present an application example of supervised learning. Let us imagine that we aim at designing a machine capable of distinguishing poisonous mushrooms from edible mushrooms. Suppose we have been provided with 1 000 examples of poisonous mushrooms and 1 000 examples of edible mushrooms, which form our training data set. Moreover, let us assume that the mushrooms are represented by two characteristics or attributes: the length of the stem and the diameter of the mushroom. These characteristics define the inputs of the classification problem. The problem has a single output attribute that can take two values, which represent whether the mushroom is poisonous or edible. Figure 2.1(a) shows how the different examples are distributed in the feature space (each class is depicted with a different color). By only considering the 2 000 known examples, the given ML technique has to be able to generalize and extract a *classification model*, which will be used to predict the class of new unlabeled examples.

2.1.2 Unsupervised learning

In *unsupervised learning*, the machine receives a set of examples that consist of input attributes, but that have no associated output attributes. Then, the goal of unsupervised machine learning is to build representations from the input that identify novel, interesting knowledge. The resulting representations can be used for decision making, predicting future inputs, grouping similar inputs, or creating prototypes that are fed to other machine learning techniques among others. Two cornerstones of unsupervised learning are *clustering* and *dimensionality reduction*.

Figure 2.1(b) presents an example of clustering. Note that, differently from figure 2.1(a), the training examples have no associated output (all points are depicted with the same color). Without any further information about the data rather than the input attributes, unsupervised learners would group the examples in different clusters according to some proximity criterion (in the example of the figure, the center of each cluster is depicted with a black dot). This type of learning is very common when hidden patterns are searched on large volumes of data. A typical example can be found in the characterization of customer habits from information about their purchases.

2.1.3 Reinforcement Learning

Reinforcement learning lies between supervised and unsupervised learning. In this type of task, an agent interacts with an environment in the following manner: the machine receives

perceptions from the environment—which provide total or partial information about its state—and performs actions to this environment with the aim of achieving a particular, or several, goals. The machine eventually receives positive or negative rewards as consequence of its actions. Therefore, reinforcement learning aims at learning a behavioral policy to maximize a notion of long-term reward—that is, to maximize not the immediate but the total reward received from the environment.

Figure 2.1(c) shows an example of reinforcement learning problem in which an ant or agent aims at reaching its goal—that is, to find the food—as fast as possible without falling in any trap. The agent may receive negative rewards from the interaction with the environment—for example, if it finds a trap—and positive rewards if it reaches a goal. Thence, the aim of the agent is to learn a behavioral policy that maps the best action for each possible sensorial input. Provided that there may be a large number of possible sensorial states and actions for each state, generalization over these sensorial states has become a key aspect in reinforcement learning to scalably solve real-world problems.

In this section we presented a classic taxonomy that identifies three types of learning. Different machine learning techniques have been developed to perform some or several of the aforementioned tasks. One of the most promising approaches to face the problems that range in the three aforementioned families is *learning classifier systems* (Holland, 1971, 1976; Holland and Reitman, 1978). Originally implemented by Holland and Reitman (1978) with the aim of simulating the animal behavior—therefore, falling under the category of reinforcement learning—, current LCSs have been extended to deal with the other two types of learning, that is, supervised learning (Bacardit and Butz, 2004; Bernadó-Mansilla and Garrell, 2003; Fu et al., 2001; Wilson, 2000) and unsupervised learning (Tamee et al., 2006, 2007; Orriols-Puig et al., 2008f). Thence, LCSs represent a general learning architecture that can be used for different tasks ranging from extracting classification models from streams of labeled data to building clusters online, also including reinforcement learning problems. The flexibility of their architecture is one of the most valuable assets of LCSs with respect to other machine learning techniques, which tend to be designed specifically for one of the three types of machine learning.

The remainder of this chapter is focused on LCSs. We first provide a brief introduction to *evolutionary computation*, which is followed by a more detailed explanation of GAs, since they are the core of the discovery component of LCSs. Then, we present the different types of GBML systems, which represent different ways of using GAs for machine learning, and place LCSs in this big picture.

2.2 Evolutionary Computation and Genetic Algorithms

Evolutionary computation (EC) is a field of study devoted to the design, implementation, and analysis of computation techniques that are inspired by the evolution of biological life in the natural world (Jong, 2006). Actually, evolutionary computation does not refer to a single type of algorithm, but to a series of parallel efforts that shared the idea of using an evolutionary process for computer problem solving. In the following sections, we provide a brief introduction to the biological principles that inspire evolutionary computation methods and propose a taxonomy of the different methods that fall under the definition of evolutionary computation; then, we focus our explanation on *genetic algorithms* (Holland, 1971, 1975), one of the most prominent

techniques in the field of evolutionary computation.

2.2.1 Biological Principles that Inspire Evolutionary Computation

At the beginning of the nineteenth century, the first evolutionary theories, which promoted the hypothesis that the species are a result of the natural evolution, started to emerge. Jean Batista Lamark was one of the first researchers that rejected the essentialist thought, which was the theory mainly considered at the time. Essentialism relied on the idea that living forms were unchanging. Lamark proposed some revolutionary theories based on the concept of evolution, which were overlooked by the scientific community at that time. Some decades later, several researchers were inspired by these theories. Among them, there were Wallace and Darwin who independently developed the idea of the mechanism of *natural selection*; this research culminated in the publication of the book *The Origin of Species* by Darwin (1859). From then on, many researchers have adhered to this hypothesis and, currently, the most accepted collection of evolutionary theories is the new-Darwinian paradigm. As proceeds, we provide a brief introduction to the basic concepts of these theories, since evolutionary computation methods are inspired by the evolutionary model proposed by them.

In brief, the evolutionary theory argues that the individuals of a population have a genetic program—i.e., *genotype*—, which defines the genetic constitution of the individual. This genotype, together with the interaction with the environment, forms the *phenotype* of the individual, that is, the observable constitution of the organism. Then, the theory states that life can be accounted for by four physical processes operating on and within populations of species: reproduction, mutation, competition, and selection. That is, individuals of a population:

1. *are reproduced*, transferring the genotype of parents to offspring;
2. *are mutated*; that is, errors in the process of information transfer inevitably occur;
3. *compete*, as a consequence of creating new individuals—over-reproducing the species—in an environment with finite resources; and
4. *are selected*, as an inevitable result of competition due to the existence of finite resources.

Therefore, this results in a cycle where species evolve by means of individual competition for a limited amount of resources. Individuals whose phenotypes are better adapted to the environment are stronger and have higher probability to survive in competition with poorly adapted individuals.

Note that stochastic processes play a key role in the theory of evolution. That is, genetic variation by means of mutation is a chance phenomenon, since errors in information transfer are unpredictable. Also, selection is probabilistic; although the quality of the individual is one of the most important aspects for its survival, there are many external factors that may influence the selection process.

The ideas briefly presented in this section were taken as inspiration by different researchers who identified the evolutionary process as an appealing approach to solve optimization problems. Consequently, several authors started their ways on designing optimization methods that

simulate different aspects of evolution, which, nowadays, have been grouped under the evolutionary computation term. A taxonomy of these different methods is provided in the following section.

2.2.2 Evolutionary Computation: A Taxonomy

In the 1950s, some researchers started to develop the idea of using biological principles to design evolutionary problem solvers. At that time, there were the first attempts to apply these types of computer problems solvers to *automatic programming*—that is, to find a program that calculates input-output functions—(Friedberg, 1958; Friedberg et al., 1959), to numerical optimization problems (Bremermann, 1962), and to the design and analysis of industrial experiments (Box, 1957; Box and Draper, 1969). These early efforts were followed by the establishment, in the middle 1960s, of three main forms of evolutionary computation: *genetic algorithms* (Holland, 1967, 1971, 1975), *evolution strategies* (Rechenberg, 1965, 1973; Schwefel, 1981), and *evolutionary programming* (Fogel, 1962, 1964). Over the next 25 years, these three branches developed quite independently; not until the early 1990s, was the term *evolutionary computation* created to embrace these different technologies, which were considered different "dialects" of biology-inspired problem solvers.

Since then, the strong research on evolutionary computation has resulted in new branches of evolutionary solvers. Two of the most significant of these new approaches are *genetic programming* (Koza, 1989, 1992)—introduced as an extension of genetic algorithms to evolve computer programs—and estimation of distribution algorithms (EDAs) (Pelikan et al., 2000b; Larrañaga and Lozano, 2002; Pelikan et al., 2006)—a new approach that creates probabilistic models to solve optimization problems. In what follows, each one of these families is shortly introduced.

Genetic algorithms (GAs) were originally created by Holland (1967, 1971, 1975) with the initial aim of understanding the underlying principles of adaptive systems, and further propelled by Goldberg, who presented GAs to a broad audience by simply and precisely presenting theory and applications of GAs (Goldberg, 1989a); later, Goldberg (2002) proposed a methodology to design competent GAs. The key idea of Holland's work was to use a combination of competition and innovation to build machines that could adapt to changing environments and could respond to unanticipated events; Holland simulated this process with a simple model of evolution that considered the notions of *survival of the fittest* and *continuous production of offspring*. The first implementations of this model used a binary representation and were based on the interaction of population size, crossover, and mutation. These ideas are still valid in current GAs.

Evolution strategies (ESs) were originally proposed by Bienert, Rechenberg, and Schwefel in 1964. The earliest idea of Bienert et al. did not aim at devising a new optimization method, but at building a robot that performed a series of experiments in a slender three-dimensional body so as to minimize its drag; the minimization method relied on changing one variable at each iteration and testing whether this change produced any improvement. ESs were born from this initial idea plus a random process to decide the variable changes (Rechenberg, 1965). The first versions of ESs used a single solution with continuous attributes that was mutated by means of a binomial distribution. The current ESs incorporate a population of solutions and perform a cycle that is similar to GAs, involving crossover, a normally distributed mutation,

and selection. In addition, ESs incorporate mechanisms for self-adapting the mutation operator to each particular individual.

Evolutionary programming (EP) was originally introduced by Fogel (1962, 1964) with the aim of creating a machine with adaptive behavior that could achieve its goals in a range of environments. For this purpose, Fogel identified that the machine should be able (1) to predict its environment and (2) to take appropriate actions in light of the predicted next state. *Finite-state machines* were found as useful to represent the behavior of the machine. Therefore, the evolutionary programming approach proposed to evolve a set of finite-state machines by using mutation as the primary reproductive operator. This general approach was applied to problems in prediction, identification, and automatic control (Fogel, 1964; Fogel et al., 1966).

Genetic programming (GP), initially proposed by Koza (1989, 1992), is an extension of GAs to evolve *computer programs*. To achieve this, GP usually employs a tree-based representation, whose internal nodes are represented with a set of primitive functions and the leaf nodes consist of terminals—usually variables of the problem. GP is based on the same GA cycle, thus involving crossover, mutation, and selection—which are redefined to let them cope with the new representation—on a finite population. GP have been applied to a large variety of problems, ranging from circuit design to quantum computing, which have result in the discovering of several inventions, which were already patented, and new patentable inventions (Koza et al., 2003).

Estimation of distribution algorithms (EDAs) are optimization methods that were recently derived from the field of GAs with the aim of building probabilistic models instead of coding the solution in populations of individuals (Pelikan et al., 2000b; Larrañaga and Lozano, 2002; Pelikan et al., 2006). Also addressed as *probabilistic model-building GAs*, EDAs replace both crossover and mutation with a probabilistic model—built from a data base that contains individuals from the previous generation—from which the new population is sampled. Although removing these two primary operators in evolutionary computation, EDAs can be considered as evolutionary computation methods since they use selection to choose good subsets of samples.

GAs appear as one of the most appealing alternatives among the five branches of evolutionary computation since they were initially designed with the general purpose of understanding natural adaptive systems and designing robust adaptive artifacts instead of specifically focusing on optimization techniques (Rechenberg, 1965) or intelligent agents (Fogel et al., 1966). This is one of the reasons that explain why GAs, as opposed to ESs or EP, have been selected as the primary discovery approach in GBML systems. Due to their importance, the next section explains how GAs work in more detail.

2.2.3 Genetic Algorithms

Genetic Algorithms (Holland, 1971, 1975; Goldberg, 1989a, 2002) are methods for search, optimization, and machine learning that are inspired by natural principles and biology. The key characteristics that differentiate GAs from other optimization techniques are:

- GAs learn from the objective function without assuming any structure or underlying distribution.

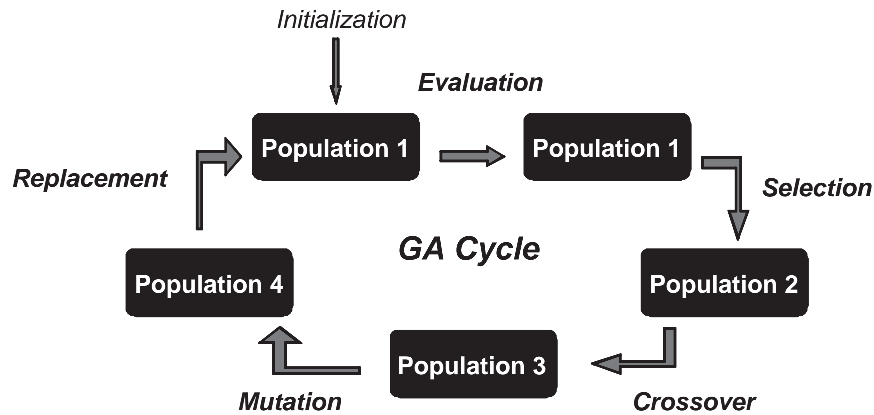


Figure 2.2: Evolution of a GA population.

- GAs search from a population of points that represent candidate solutions, not from a single point.
- GAs code potential solutions instead of directly tuning the decision variables of the problem.
- GAs use random, local operators instead of deterministic, global rules.

As follows, we describe the basic work flow of genetic algorithms, briefly review some existing theory that explains how and why GAs work, and present some of the real-life applications to which GAs have been applied in the fields of engineering, science, and industry.

Description of Genetic Algorithms

GAs evolve a *population* of rules, where each *individual* in the population represents a potential solution to the problem. Analogous to genetics, individuals are represented by *chromosomes*, which encode the decision variables of the optimization problem with a finite-length string. Each of the atomic parts of the chromosome is referred to as *genes*, and the values that the gene can take are addressed as *alleles*. For example, in the traveling salesman problem (Applegate et al., 2006), a chromosome represents a whole route—a sequence of cities—, and a gene represents a city.

To implement natural selection and competition among candidate solutions, GAs incorporate an *evaluation function* that is responsible for assessing the quality of each solution; the quality of each individual is made explicit with a *fitness* value that is given to the individual. Several evaluation functions have been used in GAs, such as mathematical functions provided by human experts or subjective functions where users choose the best solutions from a set of candidates. The design of a fitness function that correctly distinguishes between good solutions and poor solutions is a key point in the success of GAs, since the evolutionary process would push toward the fittest solutions in the population.

The population of individuals is evolved by a continuous process of *selection*, *crossover*, *mutation*, and *replacement* of individuals. Figure 2.2 schematically illustrates the cycle of a GA. Algorithm 2.2.1 complements the explanation with the pseudo code of a simple GA. In the beginning of the run, the population is initialized typically with random individuals—if available, domain-specific knowledge can be incorporated to the initialization process. Then, each individual is evaluated; therefore, each individual has a *fitness* that indicates the quality of the solution. Next, the GA performs a loop where the following for operators are iteratively applied:

- **Selection:** The selection operator chooses the fittest individuals in the population, simulating the survival-of-the-fittest mechanism. So far, several selection schemes have been presented with the common idea of biasing the selection toward the fittest individuals. For example, roulette-wheel selection (Holland, 1975; Goldberg, 1989a) and stochastic universal selection (Baker, 1985; Grefenstette and Baker, 1989) give each individual a selection probability that is proportional to its fitness. Other selection schemes such as tournament selection (Goldberg et al., 1989; Sastry and Goldberg, 2001) and truncation selection (Mühlenbein and Schlierkamp-Voosen, 1993) rank a set of individuals according to their fitness and select the fittest ones.

- **Crossover:** The crossover operator combines the genetic information of two or more parental solutions to create new, possibly better offspring. Recombination plays a key role in GAs, since it should detect important traits of parental solutions and exchange them with the aim of generating better individuals that are not identical to their parents. Several selection operators designed under this goal can be found in (Goldberg, 1989a, 2002; Pelikan et al., 2000a; Pelikan, 2005; Pelikan et al., 2006; Sastry and Goldberg, 2003a).

- **Mutation:** Mutation introduces random errors on the transference of the genetic information from parents to offspring. Thence, this operator acts on single individuals. Although different mutation operators have been designed (Bäck, 1996; Beyer, 1996; Goldberg, 1989a), the commonality among them is that they introduce one or more random changes applied to individual genes. Competent genetic operators that identify important traits of parental solutions and search in the structural neighborhoods of these solutions have been developed Lima et al. (2006); Sastry and Goldberg (2004).

- **Replacement:** After the selected individuals have gone through crossover and mutation, the offspring population replaces the original one. Several replacement schemes could be followed. For example, in a generational GA, all the offspring population may replace the parent population. Other schemes are elitist replacement—the elite individuals of the parent population are copied to the new population—or steady state replacement—the best individuals of the offspring population are copied to the original one, removing classifiers with poor fitness.

The synergy of all these operators pressures toward the evolution and selection of the best solutions, which are recombined yielding new promising offspring. Goldberg (2002) emphasized the idea that, while selection, crossover, and mutation can be shown to be ineffective when applied individually, they might produce a useful result when working together. This was explained with the *fundamental intuition of GAs*, which argues that the combination of the

Algorithm 2.2.1: Pseudo code of a simple GA.

```
   Data:  t is the time stamp and P(t) is the population at time t
1  Algorithm: GA
2  t := 0
3  P(t) := Initialize randomly P(t)
4  P(t) := Evaluate P(t)
5  while not finish do
6  |  t := t+1
7  |  P'(t) := Select individuals from P(t-1)
8  |  P'(t) := Apply crossover to P'(t)
9  |  P'(t) := Apply mutation to P'(t)
10 |  P(t) := P'(t)
11 |  P(t) := Evaluate P'(t)
12 end
```

selection and *crossover* operators introduces a process of *innovation* or *cross-fertilizing*, whereas the combination of *selection* and *mutation* represents the *continuous improvement* or *local* search process.

After outlining a GA procedure and discussing the role of the most important operators, the next section briefly reviews some theory that provides key insights that help explain why GAs work.

2.2.4 Basic Theory of GA

Since the initial definition of GAs, several authors have developed formal theory to explain their behavior. In the following, we first go back to Holland (1975) and introduce the *schema theorem*, which uses the concept of *building block* (BB) to give some insights on how GAs work. Then, we present the work by Goldberg (2002), who adheres to the ideas proposed by the schema theorem and proposes a methodology for designing competent selecto-recombinative GAs.

Intuitive Idea of Why GAs Work: the Schema Theorem

We have just seen that the operation of GAs is based on the exchange of information from parents to offspring. Along the description of GAs, we already pointed out that key operators such as crossover should detect important traits from parents and exchange them properly to create new children. In this section, we further this idea and present the schema theorem, which is concerned about accounting for how the key solutions evolve in a population. We start with the definition of *schema* and then reproduce the *schema theorem* proposed by Holland (1975).

The *schema theorem* is based on the idea of *schema* or *building blocks* (BBs), that is, a template that identifies a subset of individuals. A schema is represented with a string $s = (s_1, s_2, ..., s_\ell)$ where each bit s_i can take a different value of the ternary alphabet {0,1,*} (ℓ is the total number of bits of the schema). Thence, a *schema* represents a subspace $B^n = \{0, 1\}^n$

so that a binary string x belongs to this schema ($x \in B^n$) if

$$x_i \neq s_i \Leftrightarrow s_i = * \quad \forall i = 1, 2, ..., n. \tag{2.1}$$

Thence, for example, provided the schema 1**001, instances 101001 and 111001, among others, belong to this schema.

Before proceeding to the formalization, the following two concepts need to be defined:

- The *order* of the schema h, $o(h)$, is the number of fixed positions in the schema, that is, the number of bits that are 0- or 1-valued. For example, $o(**10*) = 2$.
- The *length* of the schema h, $\delta(h)$ is the distance between the first and the last specific positions. For instance, $\delta(**10**) = 1$.

Provided the definitions above, the schema theorem models how the different schemas evolve along a GA run. For this purpose, it considers the effects of the selection, the crossover, and the mutation operators. Moreover, it assumes fitness-proportionate selection, one point crossover, and gene-wise mutation. Then, the schema theorem demonstrates that the expected number of offspring that belong to schema s at iteration $t+1$, i.e., $E[N_S(P(t+1))|P(t)]$, satisfies that

$$E[N_h(P(t+1))|P(t)] \geq N_h(P(t))\frac{\overline{f}(h,t)}{\overline{f}(t)}\left(1 - \frac{\delta(h)}{\ell - 1}p_c\right)(1 - p_m)^{o(h)}, \tag{2.2}$$

where $N_S(P(t))$ is the number of individuals in the population $P(t)$ that belong to schema h at time t; $\overline{f}(h,t)$ is the average fitness of the individuals that belong to h at time t; and $\overline{f}(t)$ is the average fitness of the population. The effect of fitness-proportionate selection is given by the term $\frac{\overline{f}(h,t)}{\overline{f}(t)}$, which increments the expectation of the number of individuals in the next generation if the average fitness of the individual that belong to schema h is greater than the average fitness of the population. The effect of crossover is reflected in the term $1 - \frac{\delta(h)}{\ell-1}p_c$, which indicates that the probability that a schema survives depends on the length of the schema and the crossover probability. Finally, the effect of mutation is modeled by the term $(1-p_m)^{o(h)}$, which denotes that the probability that the schema is preserved to the next generation is inversely proportional to the mutation probability and exponentially proportional to the number of fixed bits of the schema ($o(h)$).

Thence, the schema theorem demonstrates that the expected number of individuals that belong to schema h at time $t+1$ grows exponentially if the average fitness of the individuals that belong to schema h at time t is greater than the average fitness of the population at time t. Therefore, the effect of reproduction becomes quantitatively clear; that is, reproduction allocates exponentially increasing number of trials to schemas whose fitness is above the average.

Design Decomposition: Goldberg's Approach to Competent GA Design

Although some researchers have strongly criticized or even rejected the schema theorem, Goldberg proposed a framework to design selecto-recombinative GAs based on the initial Holland's notion of building block. Goldberg (2002) suggested thinking of building blocks as a kind of

matter and to ensure (1) that we have an initial stock of them, (2) that good ones grow in the market share, (3) that good decisions are made among them, and (4) that they are exchanged well to solve a large class of difficult problems.

In order to satisfy the four aforementioned points, Goldberg (2002) decomposes the problem of designing competent selecto-recombinative GAs in the following seven aspects:

1. Know that GAs process BBs.
2. Know the BB challengers.
3. Ensure adequate supply of raw BB.
4. Ensure increased market share for superior BBs.
5. Know BB takeover and convergence times.
6. Make decision well among competing BBs.
7. Mix BBs well.

Goldberg (2002) proposed to examine these items by means of facetwise analysis, which suggests analyzing separately each one of these elements, assuming that the other ones behave in an ideal manner. As proceeds, we elaborate on each one of the elements and mention some of the approaches by which GA researchers have studied each element.

The primary idea of this theory is that selecto-recombinative GAs work through a mechanism of *decomposition* and *reassembling*. That is, GAs implicitly decompose the problem and identify sets of well-adapted features, which form a building block. Then, these building blocks have to be correctly processed.

The second key idea in this theory is that complex problems are those problems whose building blocks are difficult to acquire. This could be a result of having large, complex building blocks, having building blocks that are hard to separate, or having a deceptive guidance toward high-order building blocks (Goldberg, 2002).

After identifying the first two key concepts, the next four items of the theory analyze how these building blocks evolve in a *market economy of ideas*. First, we need to ensure that the market is provided with enough stock of BBs. As GA populations are usually initialized randomly, one way to obtain more variability is to use larger populations (Goldberg, 1989b; Goldberg et al., 2001; Holland, 1975).

Having provided the population with an initial stock of BBs, the next two important aspects are (1) that the best BBs should grow and take over a dominant market share of the population, and (2) that this growth should be neither too slow—so, delaying the convergence time—, nor too quick—, thus increasing the risk of falling in a local optimum. Different approaches have been taken to understand time and convergence, which cover the fourth and fifth elements of the design decomposition. Three of the most important approaches are (1) takeover time models, which model the dynamics of the best individual (Bäck, 1994; Cantú-Paz, 1999b; Goldberg and Deb, 2003), (2) selection-intensity models, where the dynamics of the average fitness of the population are modeled (Bäck, 1995; Miller and Goldberg, 1995, 1996; Mühlenbein and Schlierkamp-Voosen, 1993; Thierens and Goldberg, 1994a,b), and (3) high-order cumulant models, where models of

the dynamics of average and high-order cumulants are developed (Blickle and Thiele, 1995, 1996; Cantú-Paz, 1999a).

Yet, just ensuring an initial adequate supply of raw BBs is not enough; in addition, good decisions among competing BBs need to be taken to ensure that the best BBs will grow in the market. It has been acknowledged that as we increase the population size, we increase the likelihood of making the best possible decisions (Jong, 1975; Goldberg et al., 1992; Goldberg and Rudnick, 1991; Harik et al., 1999). Therefore, decision making has been studied from the perspective of population sizing.

The last item of the design decomposition relies on the idea that the correct identification and exchange of BBs is the critical path to innovative success. That is, when designing a competent GA, one of the key challenges that needs to be addressed is how to identify BBs and exchange them effectively. In this regard, facetwise models have been developed to show that fixed-recombination operators, such as uniform crossover, may fail to effectively identify and exchange BBs, resulting in an exponential scaling up of the population size in boundedly difficult problems—that is, problems that, for example, have large sub-solutions that cannot be decomposed in simpler sub-solutions, have several optima, or are affected by noise—(Goldberg et al., 1993; Sastry and Goldberg, 2002, 2003b). In contrast, recombination operators that can automatically identify and exchange BBs efficiently have shown to scale up polynomially with the population size in these boundedly difficult problems (Goldberg, 2002; Pelikan, 2005; Pelikan et al., 2006).

The design decomposition and facetwise analysis has resulted in a better understanding of the underlying processes of GAs, creating a formal framework formed by different pieces of theory. Furthermore, these analyses have been used as a tool for designing *competent GAs*, genetic algorithms that can solve boundedly difficult problems quickly, reliably, and accurately (Goldberg, 2002). The first designs of competent GAs can be found in *messy GA* (Goldberg et al., 1989). Currently, there are several implementations of competent GA such as the *linkage learning genetic algorithm* (Harik, 1997), the *extended compact genetic algorithm* (Harik, 1999; Sastry and Orriols-Puig, 2007), or the *Bayesian optimization algorithm* (Pelikan et al., 1999). The maturity in the GA field has promoted the use of GAs in real-world problems. The next section reviews some of these important applications.

2.2.5 Genetic Algorithms in Real-World Applications

All the success and better understanding of genetic algorithms has led to their application to a large variety of problems in science, engineering, and industry. Therefore, GAs have not been stuck in "toyish" problems but have been applied to complex, previously unsolved, real-world problems. We review some of the most important applications in what follows.

In the scientific field, GAs have been employed in different applications such as the detection of coronary problems (Grefenstette and Fitzpatrick, 1992), the design and interaction in computer games (Jo and Ahn, 2002), and the generation of music (Goksu et al., 2005). But the application of genetic algorithms, differently from other optimization techniques, is not merely limited to a scientific field. GAs have been successfully applied to complex problems in industry, providing novel solutions. For instance, GAs were used to partially design the Japanese bullet train N700; specifically, the shape of the front of the train was optimized by a GA. Another

significant example is the EvoFIT tool[1], a system based on GAs that makes robot pictures, which was used by the Northamtonshire police.

There are also some companies that use GAs as the heart of their applications such as *Optimatics* and *Schema*. *Optimatics* [2] is a world leader provider of innovative and customized optimization solutions to water industry. This company uses GAs as an essential tool for optimization. The results provided by the company highlight that the GA-based optimization has resulted in savings of about 20%-30%, on average, in their projects. *Schema*[3] is a global provider of end-to-end network optimization solutions for transport and mobile networks that uses GAs in their applications. Some of the most significant projects of this company are missile balancing, synthetic aperture radar, optimal container stowage, and frequency allocation for cellular networks.

Therefore, GAs have been used as competent optimization tools in some complex scientific and industrial applications, assisting the creation of commercial products. In addition to these applications in the optimization realm, GAs have also been used as the heart of several machine learning techniques, yielding to a discipline which has been referred to as *genetic-based machine learning*. The next section reviews the main branches of the algorithms that fall under these definitions, which includes both Michigan- and Pittsburgh-style LCSs.

2.3 Genetic-based Machine Learning and Learning Classifier Systems

The application of GAs has not been restricted to optimization problems, but they have also been used as the primary discovery heuristic in machine learning procedures. Since Holland (1962) outlined his theory for adaptive systems, GAs have been used as the main discovery component in Michigan-style LCSs (Holland, 1976; Holland and Reitman, 1978) and Pittsburgh-style LCSs (Smith, 1980, 1983, 1984), which conform the two original branches of GBML. Furthermore, the population-based search, robustness, and knowledge-representation flexibility of GAs, coupled with the recent advances in efficiency and competent GAs (Goldberg, 2002; Pelikan, 2005; Pelikan et al., 2006; Goldberg et al., 2007), has promoted the use of genetic search as the primary discovery heuristic in several machine learning techniques that belong to different learning paradigms that range from neural networks (Kitano, 1990; McInerney and Dhawan, 1993; Liu et al., 2004; Wierstra et al., 2005; Mierswa, 2007) to probabilistic classifiers (del Jesus et al., 2004; Otero and Sánchez, 2006; Yalabik and Fatos, 2007). This has resulted in several new approaches to use GAs in machine learning, which have shown to be highly competitive with respect to traditional non-evolutionary systems (Orriols-Puig et al., 2008d,c).

The purpose of this section is to describe the five main branches of GBML. We start with the description of Michigan- and Pittsburgh-style LCSs. As several particular implementations have been designed for both types of systems, we provide a general schema for each LCS. Then, we present three other forms of GBML that have received a special amount of attention during the last decade: *Iterative rule learning* (IRL) (Venturini, 1993), *genetic cooperative-competitive learning* (GCCL) (Giordana and Neri, 1995; Greene and Smith, 1993), and the

[1] http://www.evofit.co.uk
[2] http://www.optimatics.com
[3] http://www.schema.com

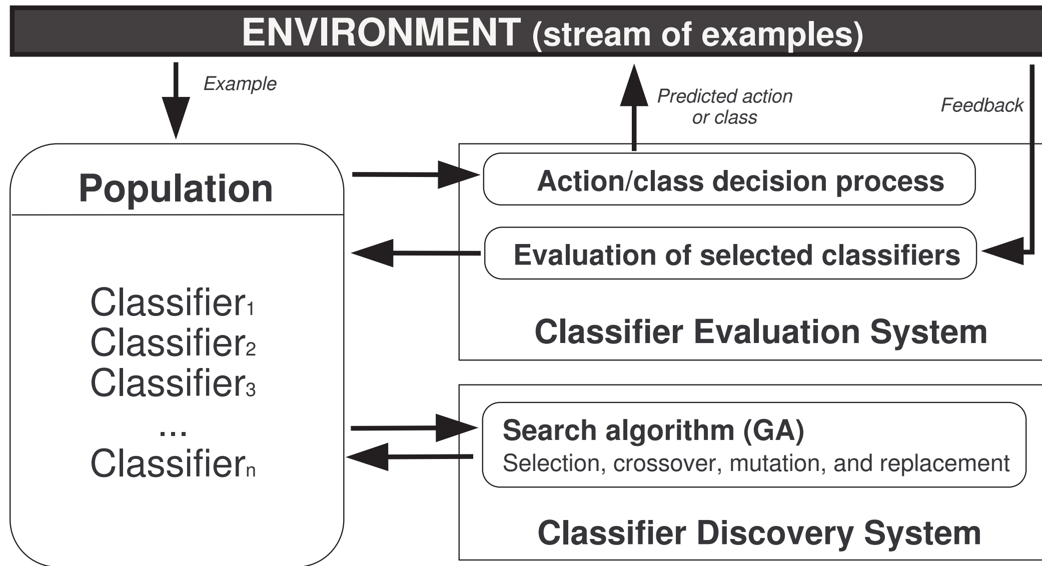


Figure 2.3: Simplified schematic of Michigan-style LCSs which the typical process organization.

organizational classifier system (OCS) (Wilcox, 1995). All these three approaches are combines of Michigan- and Pittsburgh-style LCSs. The IRL approach uses a Michigan-like representation in a Pittsburgh-style LCSs to learn a set of rules incrementally. GCCL systems define a framework where both competition in system niches and cooperation among all rules are performed. OCS distributes classifiers in organizations and takes ideas from the economic study of transaction costs to control the sizes of these organizations. A general schema of the process organization of each branch of GBML is presented as follows.

2.3.1 Michigan-style LCSs

Since the first successful implementation of a Michigan-style LCSs (Holland and Reitman, 1978), research on Michigan-style LCSs has resulted in new systems that have been applied to different types of learning tasks. Therefore, although initially designed to simulate animal behavior—later inspiring the whole field of reinforcement learning (Sutton and Barto, 1998)—, current LCSs can be applied to a large variety of learning tasks such as supervised learning and data mining (Bernadó-Mansilla and Garrell, 2003; Bull, 2004; Bull et al., 2008), function approximation (Wilson, 2002b; Butz et al., 2008), reinforcement learning (Lanzi, 1999b, 2002; Lanzi et al., 2005; Butz et al., 2005a), and clustering (Tamee et al., 2006, 2007). As proceeds, we present a general architecture that highlights the common points among the different implementations.

Figure 2.3 illustrates the common process organization of current Michigan-style LCSs. That is, all Michigan-style LCSs share three key components that distinguish them from other GBML and machine learning techniques:

1. a knowledge representation based on *classifiers*, which maps the inputs with classes or actions,

2. a *classifier evaluation system* which evaluates the population of classifiers online, and

3. a *classifier discovery system* that is triggered with a certain frequency and is responsible for discovering new promising classifiers and adapting the knowledge base to eventual changes in the environment.

It is worth highlighting that the system learns online from an environment, which can represent either the environment in which an agent lives or a set of examples that are made available in a data stream.

The core of the system maintains a population of *classifiers*. Each classifier consists of (a) a structure that maintains an input/output mapping, identifying to which inputs the classifier is applicable and which action should be performed in case of matching, and (b) several parameters that maintain different statistics of each classifier, such as its fitness. The structure that maintains the input/output mapping has usually been implemented with *production rules* (Holland and Reitman, 1978; Wilson, 1994, 1995, 2001; Bernadó-Mansilla and Garrell, 2003); other implementations such as neural networks (Bull and O'Hara, 2002), first-order logic expressions (Mellor, 2005), messy representations (Lanzi, 1999a), LISP s-expressions (Lanzi and Perrucci, 1999), and gene expression programs (Wilson, 2008) have also been used. In any case, note that each classifier covers a restricted set of sensorial inputs; therefore, the solution of a given problem is the whole population.

Michigan-style LCSs update the parameters of these classifiers online by means of interacting with the environment. That is, at each learning iteration, the environment provides a new input example. Then, the system uses a sub-population of classifiers to decide the action or class that should be taken according to the current input. This action is given to the environment, which, in turn, returns a feedback that indicates the quality of the prediction. Then, the evaluation component uses this information to adjust the quality of the classifiers that have participated in the action decision process. Moreover, with a certain frequency, the rule discovery system is triggered, generating new promising classifiers. Usually, a niche-based steady-state GA is employed, which selects a group of classifiers, applies genetic operators to create new ones, and introduces them into the population removing other classifiers if there is no room for the new ones. Other search procedures such as evolution strategies have recently been used to guide the classifier discovery system of LCSs (Morales-Ortigosa et al., 2008a,b).

Several Michigan-style LCSs have been designed since the first implementation of CS-1 by Holland and Reitman (1978), such as the EYE-EYE system (Wilson, 1981, 1985a), the Boole system (Wilson, 1985b, 1987)—which took some inspiration from Goldberg (1983) work on LCSs—, and the NewBoole method (Bonelli and Parodi, 1991). Although these systems were able to solve some specific applications, several drawbacks, mainly associated with the achievement of accurate generalizations, hindered their success. Further research resulted in the design of the *extended classifier system* (XCS) by (Wilson, 1995), supposing a tipping point in the LCSs research. As were its ancestors, XCS was originally devised to solve reinforcement learning tasks. Since then, several new Michigan-style LCSs have been designed based on the XCS's architecture. One of these derived systems can be found in UCS (Bernadó-Mansilla and Garrell, 2003), which inherits the main components of XCS, but specializes the system for supervised learning.

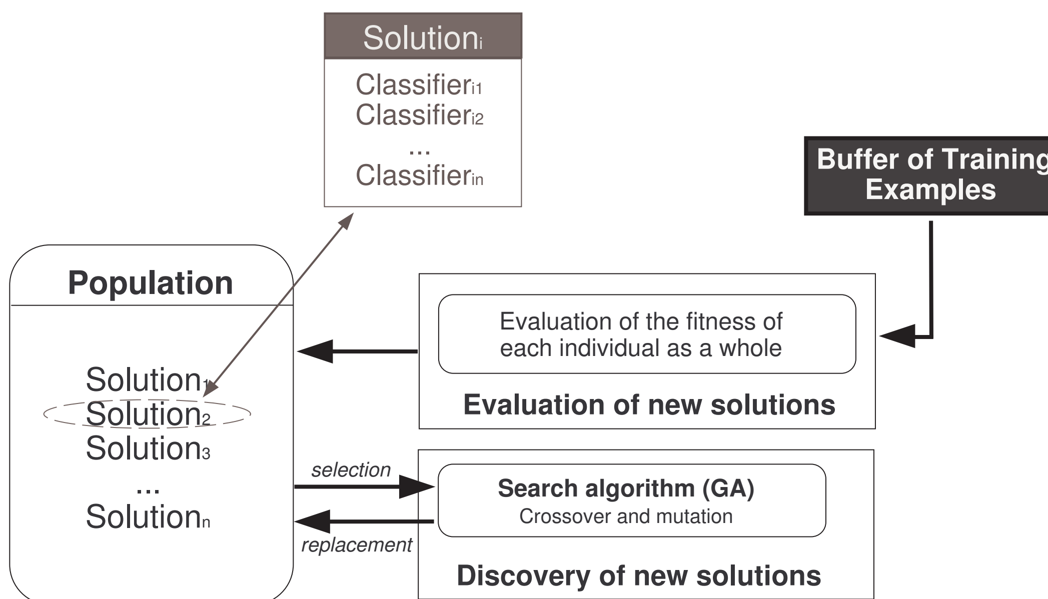


Figure 2.4: Simplified schematic of Pittsburgh-style LCSs.

2.3.2 Pittsburgh-style LCSs

Contemporaneous with the research on Michigan-style LCSs, some authors took another approach and extended GAs to machine learning, resulting in the so-called Pittsburgh-style LCSs. Pittsburgh-style LCSs have three fundamental differences with respect to Michigan-style LCSs: (1) the knowledge representation, (2) the evaluation system, and (3) the application mode of the GA and the definition of the genetic operators. In this section, we examine these differences, describe a general process organization of Pittsburgh-style LCSs, and review some of the most significant implementations in this area.

Figure 2.4 illustrates the process organization of a Pittsburgh-style LCSs, which is directly extended from the typical process organization of a simple GA. In Pittsburgh-style LCSs, individuals are complete solutions to the whole problem; that is, each individual should cover all the feature space, instead of only covering a portion of it as in the Michigan approach. Usually, Pittsburgh-style LCSs represent individuals as a disjunction of rules—which in most cases are made available as a decision list (Rivest, 1987). Nonetheless, other representations such as a set of decision trees (Llorà and Garrell, 2001; Llorà and Wilson, 2004) have also been used. In the remainder of this section, for consistency with the Michigan approach, we use the term *classifier* to refer to each one of fundamental parts of an individual.

Since each individual maintains a set of classifiers, which jointly cover the whole problem, there is no need for evaluating the quality of each of these classifiers on its own. Therefore, differently from the Michigan approach, the classifier apportionment algorithm can be sidestepped; instead, a single measure is enough to evaluate the quality of the whole individual. Different indicators have been used to evaluate the quality of individuals, the prediction accuracy and the generality of the individuals being the most common ones. Thence, immediately after its

creation, each individual is evaluated offline with a set of examples, which either have been provided at the beginning of the run in the form of a static data set or have been collected during the learning process. Note that, under this approach, there is no control about the contribution of each classifier to the performance of the whole individual.

Then, the population is evolved by means of genetic algorithm cycles. That is, at each iteration, the system selects a set of individuals, which are crossed, mutated, and inserted into the population replacing other probably low fit individuals. The crossover and mutation operators are adapted to deal with the representation of the individuals. At the end of the learning process, the best individual in the population is used to predict the output of new test examples.

After the implementation of LS-1 (Smith, 1980, 1983, 1984), the first Pittsburgh-style LCS, there have been some successful developments of Pittsburgh-style LCSs for supervised learning such as GABL (Jong and Spears, 1991) and GIL (Janikow, 1993). In GABL, each individual is encoded with a variable-length set of rules, and each rule follows a fixed-length, binary representation. Rules have no class associated since GABL performs concept learning, that is, it learns only positive or negative examples. The fitness is computed as the squared accuracy function. The system uses the typical genetic operators except for crossover, which is restricted to ensure that the operator selects the same position to cut the variables of two parents. GIL follows a similar scheme but uses rules defined in the VL_1 logic (Michalski et al., 1986) and a fitness function that tries to balance the accuracy-complexity tradeoff of the individuals. In addition, the system is provided with several operators that modify the rules at the semantic level. A more recent implementation of a Pittsburgh-style LCS, which overcomes the scalability problems detected in previous approaches (Freitas, 2002), can be found in GAssist (Bacardit, 2004).

2.3.3 Iterative Rule Genetic-based Machine Learning

Iterative rule learning (IRL) follows a *separate-and-conquer* methodology (Pagallo and Haussler, 1990) to learn a set of rules. The separate-and-conquer methodology proposes to iteratively learn rules that cover a subset of the input instances. That is, the following two steps are iteratively performed: (1) learn a rule that covers part (or all) of the training examples, and (2) remove the covered examples from the training set. This process is repeated until no training examples remain. At the end of the process, the solution is the concatenation of the rules created at each iteration. Notice that this approach incrementally creates new rules and, at the same time, reduces the search space since the covered examples are removed from the training data set. This method has also been referred to as the *covering* strategy (Michalski, 1969).

Thence, IRL defines a general learning architecture in which different learning procedures could be applied to extract the individual rules. Among others, GAs have been used to discover these rules. That is, at each learning iteration, a GA is applied to induce a population of rules. Therefore, the knowledge representation in the GA is the same as in the Michigan approach, but rules compete with all the other rules in the population and are evaluated offline as in the Pittsburgh approach.

The first proposal of IRL in the context of GAs can be found in the SIA system (Venturini, 1993). SIA generates an initial population from generalizations of randomly selected instances,

and a GA is used to evolve these rules. Rules are evaluated according to their complexity and accuracy. The process stops when the best rule remains stable for a certain number of generations. More recent approaches can be found in the HIDER system (Aguilar-Ruiz et al., 2003, 2007) and the NAX method (Llorà et al., 2007) for classification tasks and the HIRElin technique (Teixidó-Navarro et al., 2008) for function approximation tasks. A common characteristic of these three learning algorithms is that they make the rules available as a decision list (Rivest, 1987). GAs for IRL have also been extensively used in in genetic fuzzy systems (Cordón et al., 2001a; González and Pérez, 1999).

2.3.4 Genetic Cooperative-Competitive Learning

Genetic cooperative-competitive learning was initially designed as a synthesis of aspects of both Michigan- and Pittsburgh-style LCSs (Greene and Smith, 1993). This approach combines the offline rule processing of Pittsburgh-style LCSs with the idea of Michigan-style LCSs that the solution is the whole population, and so, that rules need to collaborate to cover all the input space. Below, we provide a general schema of this type of GBML systems in some detail.

GCCL was born with the purpose of explicitly addressing the goal of constructing highly accurate and as-simple-as-possible decision models from a set of examples. To achieve this, GCCL systems approach this problem by assuming that the examples of the training data set correspond to niches in an *ecology*. The exact number of niches is not known, but it is assumed to be less than the total number of examples in the data set; therefore, several examples can be placed in the same ecological niche. Then, the population is considered to be the whole model, which represents all the niches of the ecology, and each individual is a representation of a particular niche. Individuals are coded as single rules, and the examples that are correctly predicted by the individual are assigned to this rule. Then, the objective is to learn the minimum number of niches or individuals that can cover all the input instances accurately.

The first proposal of a GCCL system can be found in COGIN (Greene and Smith, 1993) which was designed after several works on the application of GAs to symbolic induction problems that produced significant systems such as ADAM (Greene, 1987; Greene and Smith, 1987) and GARGLE (Greene, 1992). Later, Giordana and Neri (1995) designed a new GCCL addressed as REGAL, which was based on their previous work on concept learning based on GAs. The main novelty of the system is that it provided a new selection operator that allowed the population to converge, on average, to an equilibrium state.

2.3.5 The Organizational Classifier System

The organizational classifier system takes ideas from both Michigan- and Pittsburgh-style LCSs to debate on appropriately sizing organizations, simulating the economic idea of transaction costs. In what follows, we briefly review the architecture of OCS and discuss the novelties of this approach.

OCS inherits the main ideas of simple classifier systems and focuses on the problem of trying to distinguish rules that lead to optimal decisions from those that lead to suboptimal decisions in order to evolve ideal rule sets. For this purpose, the system distributes the classifiers of the population in different organizations of variable size. These organizations can interact among

themselves. To control the size of the organizations, OCS incorporates ideas from transaction cost theory by using reputation for organizational recruitment and by paying attention to efficient organization sizing. That is, on the one hand, OCS includes a credit-allocation scheme that distributes reputation among classifiers and organizations and a conflict-resolution method that uses rules and organizations reputation to determine the interactions among classifiers and organizations. On the other hand, the system implements an organizational growth component that controls the sizes of the organizations by applying different genetic operators to enlarge or shrink organizations, which preserve the idea that organizations with larger reputation may be larger than organizations with lower reputation.

Despite the novelty of the ideas proposed in the OCS framework, research on OCS systems alike has been scarce during the last decade. Recently, these concepts have been applied by Vallim et al. (2008) to deal with problems of multi-label classification.

In this section, we presented four branches of GBML, which share the use of a GAs for machine learning. Among them, this thesis is focused on Michigan-style LCSs. The most important reasons that led us to research on these types of LCSs is that Michigan-style LCSs

1. Evolve a distributed solution in parallel, applying local search procedures to niches instead of optimizing a set of classifiers globally.
2. Create individual classifiers whose contribution to the whole is determined by the system; therefore, each individual classifier can be regarded as an expert in the region of the feature space that it covers.
3. Learn the model online from a stream of examples. This is not only useful for reinforcement learning problems—where instances come online as the agent finds new sensorial states while moving around its environment—, but also for tackling current industrial and scientific applications in which large volumes of data are generated online, and the learning systems need to extract the key information that resides in the stream of data on the fly.

These three characteristics, together with the increasing application of LCSs to new real-world problems, encouraged us to take this approach in the present work.

2.4 Summary

This chapter provided a brief introduction to ML and to the use of GAs in ML. Starting from a brief description of ML and a classic taxonomy of the different ML tasks, we introduced evolutionary computation methods in general, and GAs in particular, as robust optimization techniques. Then, we explained different types of algorithms that use GAs to evolve their knowledge representation, placing LCSs in this context.

The present work focuses on Michigan-style LCSs, the original approach to use GAs for machine learning. While this chapter has provided a general introduction to these types of systems, the next chapter focuses on the two approaches studied in this thesis: XCS and UCS. We consider XCS since it is, by far, the most influential Michigan-style LCS, which has been widely used to solve different types of problems. Besides, this thesis is also interested in UCS,

an extension of XCS that restricts the learning architecture to supervised learning with the aim of dealing with classification problems more efficiently. In the next chapter, these two LCSs are described in detail.

Chapter 3

Description of XCS and UCS

The design of the *extended classifier system* (XCS) by Wilson (1995) supposed a milestone in the history of learning classifier systems. Wilson proposed XCS after several years of research that yielded important results such as the *boole* system (Wilson, 1985b, 1987) or the most recent *zeroth-level classifier system* (ZCS) (Wilson, 1994). The success of XCS was mainly due to its "simplified" structure which addressed the different challenges of LCSs at that time. XCS avoided the evolution of an excessive number of over-general classifiers by basing fitness on the accuracy of the reward prediction instead of on the prediction itself. Besides, XCS provided intrinsic generalization capabilities due to the combination of a niche-based GA and a population-wise deletion operator.

Since its first proposal in 1995, a lot of research has been conducted on formalizing the algorithmic structure (Butz and Wilson, 2001), enhancing the system with new operators (Wilson, 1998; Kovacs, 1999; Butz et al., 2003), and deriving theory for a better understanding of its underlying processes (Butz and Pelikan, 2001; Butz et al., 2004b, 2005a, 2007; Drugowitsch and Barry, 2008; Drugowitsch, 2008). Besides, new systems have been derived from XCS for specific types of learning tasks. In the context of supervised learning, Bernadó-Mansilla and Garrell (2003) defined the *supervised classifier system* (UCS), an LCS that inherited the process organization from XCS, but was specialized for supervised learning. Since in this thesis we are especially concerned about solving supervised learning tasks, we consider both XCS—as being the general learning architecture—and UCS—as being specialized for these types of tasks.

The purpose of this chapter is to provide a concise description of both XCS and UCS. Section 3.1 introduces the XCS architecture and further details the different components of the system and the process organization; besides, we provide some theory that explains why XCS is able to generalize from a set of examples. Section 3.2 presents UCS, focusing on the modifications introduced with respect to the online architecture of XCS. In both cases, we assume a ternary representation. Section 3.3 reviews some new representations proposed to deal with new types of data more effectively. Finally, section 3.4 summarizes the chapter.

3.1 The XCS Classifier System

XCS (Wilson, 1995, 1998) is a Michigan-style LCSs that evolves a population of *classifiers*—usually, production rules—online by means the interaction with an environment. A steady state genetic algorithm is responsible for evolving these classifiers online. The main differences between XCS and other Michigan style LCSs are (1) that XCS simplifies the architecture of other Michigan-style LCSs—for example, removing the message list—and (2) that XCS computes the classifiers's fitness based on the accuracy of the reward prediction instead of calculating the fitness from the reward prediction itself. Due to the latter aspect, XCS creates a set of maximally general and accurate classifiers that map the problem space completely; that is, classifiers with both low and high expected prediction reward are evolved by the system, creating the so-called *complete action map*.

As follows, we explain the learning architecture of XCS in detail. First, we review the knowledge representation assuming that the classifiers are represented with ternary rules, as done in the first versions of the system. Then, we explain the process organization of the system, which includes the learning interaction, the classifier evaluation system, the discovery component, and the reasoning mechanism to infer the action of a new input instance. Finally, we review theory that explains why XCS is able to generalize and learn a set of maximally general and accurate classifiers. All the explanation assumes that XCS is working in single step tasks. For more details about the architectural changes needed to deal with multiple step problems, the user is referred to (Wilson, 1995, 1998; Butz et al., 2005a).

3.1.1 Knowledge Representation

XCS evolves a distributed knowledge represented by a *population* [P] of *classifiers*, where each classifier contains a rule and a set of parameters that estimate the quality of the rule. Different rule representations have been designed for XCS so far. In general, XCS can evolve any type of rule—or even, other types of representations such as trees or neural networks—provided that the genetic operators are properly redefined. In this section, we consider the ternary rule representation, since this was the representation originally designed with the system. Later, in section 3.3, we present different types of rules representations that have been designed for XCS and UCS in the last few years.

A rule takes the form

$$\textbf{if } \textit{condition} \textbf{ then } \textit{action}. \tag{3.1}$$

That is, it consists of a *condition*, which is formed by a set of variables in disjunctive normal form that specify when the classifier is applicable, and an *action*, which determines the predicted action or class. Each variable of the condition can take a value of the ternary alphabet $\{0, 1, \#\}^\ell$, where ℓ is the number of input variables. The *don't care* symbol '#' allows for rule generalization; that is, '#' indicates that the given variable matches any input value. Therefore, a rule k matches an input example e if for each variable v_i: $v_i^k = e_i \vee v_i^k = \#$.

Besides the rule, a classifier also contains a set of parameters that maintain different statistics of the rules. The most important parameters associated with a classifier are:

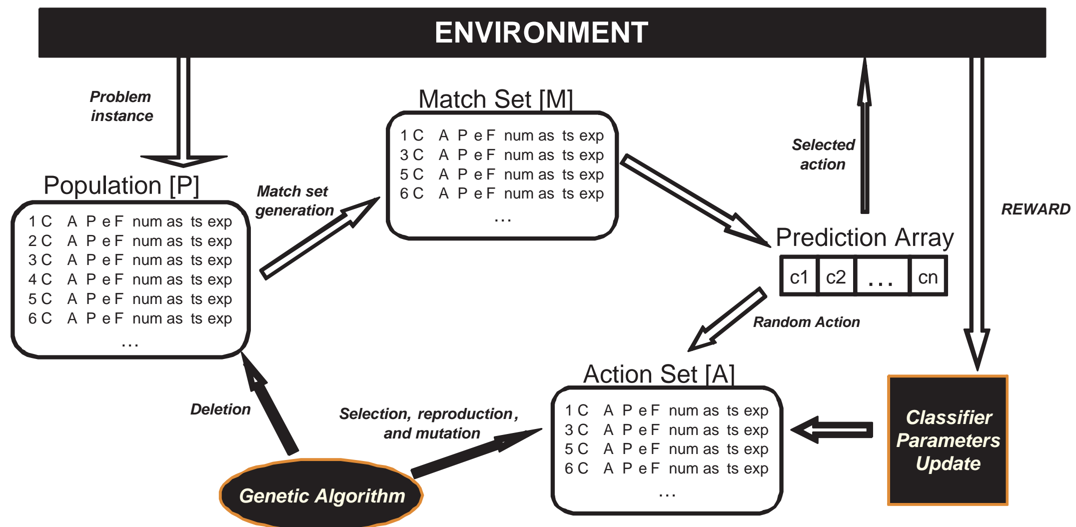


Figure 3.1: Schematic of the process organization of XCS.

1. The payoff prediction p, an estimate of the payoff that the classifier will receive if its condition matches and its action is chosen.
2. The prediction error ϵ, an estimate of the average error between the classifier's prediction and the received payoff; that is, it computes the mean absolute deviation of the prediction error with respect to the received rewards.
3. The fitness F, an estimate of the scaled, relative accuracy[1] of the payoff prediction.
4. The action set size as, an estimate of the size of the action sets in which the classifier has participated (see section 3.1.2).
5. The experience exp, which reckons the number of examples that the classifier has matched during its life.
6. The numerosity n, which indicates the number of copies of the classifier in the population. In this way, identical classifiers can be represented as a single individual in the population, speeding up the runtime since the matching time (as well as the time required for other operations) decreases.

To completely understand the knowledge representation, in the following sections we detail how the different components of XCS interact to evaluate the existing rules and to create new promising classifiers.

3.1.2 Learning Interaction

XCS learns online by interacting with an environment which provides a new training example at each iteration. Figure 3.1 schematically illustrates this process. The system works in two

[1] Relative accuracy is computed with respect to other classifiers in the same action set.

different modes: *exploration* or training and *exploitation* or test. In exploration mode, XCS seeks to evolve a maximally general rule set that minimizes the prediction error of the rules. In exploitation mode, XCS uses the rule set to decide the best action for a new input example. As proceeds, we discuss in more detail how the different components of XCS interact to learn a population of maximally general and accurate classifiers from the interaction with this environment; that is, we focus on the exploration phase. In section 3.1.5, we explain how the evolved knowledge is exploited to predict the action of new inputs.

XCS usually starts the *exploration* phase with an empty population. At each learning iteration, the system is provided with a new instance e. Then, the system builds a *match set* [M] containing all the classifiers in [P] whose conditions match e. If the number of classes represented in [M] is less than the θ_{mna} threshold (θ_{mna} is usually set to the total number of possible classes of the problem), the *covering* operator is triggered, creating as many new classifiers as required to cover θ_{mna} different classes. The condition of the new classifiers created by covering is generalized from e. That is, each variable is set to '#' with probability $P_{\#}$ (where $P_{\#}$ is a configuration parameter); otherwise, the variable takes the corresponding value in e. The class of the new classifier is randomly selected among the classes that are not covered in [M]. The parameters of the new classifiers are set to initial values; typically, $p = 10$, $\epsilon = 0$, and $F = 0.01$. These parameters are initialized with a value close to zero to avoid an excessive influence of young classifiers in the selection and inference procedures; as long as these classifiers participate in action sets, the parameter update procedure adjusts their parameters to their real value. Besides, the numerosity is set to 1, the experience to 0, and the action set size to the size of the match set where the covering has been fired.

Next, the system computes the *system prediction* $P(c_i)$ for each possible class, which estimates the payoff that the system will receive if c_i is selected as output. $P(c_i)$ is calculated as the fitness weighted average of the predictions of the classifiers in [M] that advocate class c_i; that is:

$$P(c_i) = \frac{\sum_{cl.class=c_i \wedge cl \in [M]} cl.p \cdot cl.F}{\sum_{cl.class=c_i \wedge cl \in [M]} cl.F}, \tag{3.2}$$

where $cl.class$, $cl.p$, and $cl.F$ refer to the class, the reward prediction, and the fitness of the classifier respectively. Then, XCS selects one of the classes randomly. Thus, XCS explores the consequences of all classes for each possible input. Notice that other exploration regimes, such as giving each class c_i a selection probability proportional to P_{c_i}, could be applied as well. The chosen class determines the action set [A], which consists of all classifiers advocating that class. The action set works as a *niche* where the parameters update procedure and the genetic algorithm take place. The next subsections explicate these two procedures in detail.

3.1.3 Classifier Evaluation

In training mode, after XCS sends the chosen class to the environment, a reward R is returned. R is maximal if the proposed class is the same as the training example (usually 1000), and minimal (usually zero) otherwise. Then, in single step problems, classifier parameters are updated with respect to the immediate reward in the current action set. As proceeds, we detail this parameter update procedure.

The prediction of each classifier cl is first updated according to the Widrow-Hoff rule (Widrow and Hoff, 1988) as

$$cl.p \leftarrow cl.p + \beta(R - cl.p), \tag{3.3}$$

where β ($0 < \beta \leq 1$) is the learning rate. The learning rate fixes the adaptivity of the parameters to the received rewards. That is, large values of β would produce large corrections in the prediction parameter each time the classifier participates in [A], whilst low values of β will cause small corrections. A typical value for this parameter is $\beta = 0.2$ (Butz and Wilson, 2001). Next, the prediction error $cl.\epsilon$ is computed as

$$cl.\epsilon \leftarrow cl.\epsilon + \beta(|R - cl.p| - cl.\epsilon). \tag{3.4}$$

Then, the fitness is updated as follows. First, the *accuracy* $cl.\kappa$ of each classifier cl in [A] is calculated as

$$cl.\kappa = \begin{cases} \alpha(cl.\epsilon/\epsilon_0)^{-\nu} & cl.\epsilon \geq \epsilon_0; \\ 1 & \text{otherwise.} \end{cases} \tag{3.5}$$

Note that $cl.\kappa$ is an inverse function of the error. The formula uses a power function with exponent ν (ν is a configuration parameter), enabling in this way to tune the pressure toward highly accurate classifiers; besides, when the classifier has a prediction error lower than the configuration parameter ϵ_0, the system considers this classifier maximally accurate. The accuracy $cl.\kappa$ is used to compute the *relative accuracy* $cl.\kappa'$ as

$$cl.\kappa' = \frac{cl.\kappa \cdot cl.n}{\sum_{cl_i \in [A]} cl_i.\kappa \cdot cl_i.n}, \tag{3.6}$$

which reflects the relative accuracy of the classifier with respect to the other classifiers in the same action set. Thence, using this procedure, all the classifiers in the niche share the global resources of that niche. Then, $cl.\kappa'$ is employed to update the fitness as

$$cl.F = cl.F + \beta(cl.\kappa' - cl.F). \tag{3.7}$$

Thus, the fitness is an estimate of the accuracy of the classifier prediction relative to the accuracies of the overlapping classifiers. This provides fitness sharing among the classifiers belonging to the same action set. Finally, the action set size is updated as

$$cl.as = cl.as + \beta(|[A]| - cl.as), \tag{3.8}$$

where $|[A]|$ is the size of the current action set. At the end of this process, the experience of the classifier is incremented.

Classifier's parameters p, ϵ, and as are updated differently in the first iterations of XCS. That is, to let the classifier parameters move to quickly to their real values at the beginning of the classifier life, the *moyenne adpative modifiée* technique (Venturini, 1994) is used. This technique sets the parameters of the classifiers directly to the average value computed with the instances that the classifier has matched. This process is applied while the experience of the classifier is less than $1/\beta$.

Once the parameters of the classifiers in [A] have been evaluated, the GA can be applied to the current niche. The next section explains the genetic search in more detail.

3.1.4 Classifier Discovery

XCS uses a steady-state niche-based *genetic algorithm* (GA) (Goldberg, 1989a) to discover new promising classifiers. The GA is triggered in the current action set if the average time since its last application to the classifiers in [A] is greater than θ_{GA}. Here, we explain the basic mechanisms of the GA.

The GA selects two parents from the current [A] following either proportionate selection (Wilson, 1995) or tournament selection (Butz et al., 2005c) and copies them. Under proportionate selection, each classifier has a probability $p_{sel}(cl)$ to be selected proportional to its fitness; that is,

$$p_{sel}(cl) = \frac{cl.F}{\sum_{cl_i \in [A]} cl_i.F}. \tag{3.9}$$

Under tournament selection, a proportion of the action set, specified with the configuration parameter σ, is selected to participate in the tournament. The classifier with maximum fitness in the tournament is chosen.

The copies undergo crossover with probability χ and mutation with probability μ per allele. Two crossover schemes have been used for XCS: two-point crossover and uniform crossover. Two-point crossover copies the two parents into two offspring, selects two cut points in the offsprings, and swaps all the variables between the two points. Uniform crossover decides, for each variable, from which parent the information is copied. If crossover, is not applied, the offspring are exact copies of the parents. After this, mutation is applied as follows. For each input variable, the mutation operator randomly decides whether the variable needs to be changed. In this case, it randomly chooses a new value for the variable. The class of the rule also undergoes the same process.

The offspring parameters are initialized as follows. If crossover is not applied, the prediction, the error, and the fitness parameters are copied from the selected parent. Otherwise, these parameters are set to the average value between the corresponding parameters in the parents. In both cases, the fitness is decreased to 10% of the parental fitness. Experience and numerosity are initialized to 1.

The resulting offspring are introduced into the population via subsumption (Wilson, 1998). That is, if there exists a sufficiently experienced ($cl.exp > \theta_{sub}$) and accurate ($cl.\epsilon < \epsilon_0$) classifier cl in [A] whose condition is more general than the new offspring, the numerosity of this classifier is increased. Otherwise, the new offspring is introduced into the population. Two classifiers are removed if the population is full. The deletion probability of a classifier is proportional to the action set size estimate of the classifier; moreover, the deletion probability is increased if the classifier cl is experienced enough ($cl.exp > \theta_{del}$) and its fitness $cl.F$ is smaller than a proportion of the average fitness of the population $\overline{F}$ ($cl.F < \delta\overline{F}$) (Kovacs, 1999). That is, the deletion probability p_{del} of a classifier cl is computed as

$$cl.p_{del} = \frac{cl.d}{\sum_{\forall cl_i \in [P]} cl_i.d}, \tag{3.10}$$

where

$$cl.d = \begin{cases} \frac{cl.n \cdot cl.as \cdot F_{[P]}}{cl.F} & \text{if } cl.exp > \theta_{del} \text{ and } cl.F < \delta F_{[P]}; \\ cl.as \cdot cl.n & \text{otherwise,} \end{cases} \tag{3.11}$$

where $F_{[P]}$ is the average fitness of the population. This deletion scheme biases the search toward highly fit classifiers and, at the same time, balances the classifiers' allocation in the different action sets.

Once the learning finishes, the evolved population is used to infer the class of new unlabeled instances. The next section provides the reasoning mechanism used by XCS.

3.1.5 Class Inference in Test Mode

The final solution of XCS consists of a set of rules with minimal error that cover all the problem space. This means that, in classification tasks, the system would evolve two types of rules: (1) rules with high prediction and low error and (2) rules with low prediction—thence, predicting the wrong class—and low error. For this reason, it is said that XCS evolves a *complete action map* (Wilson, 1995; Kovacs and Kerber, 2001). In test or exploitation mode, all the matching classifiers in the population are used to infer the class of a new input instance. The reasoning mechanism works as follows. Firstly, XCS creates [M] with all the matching classifiers; covering is not applied in any case. Then, the prediction array is formed as explained in section 3.1.2, and the most voted class is returned as output. Note that during test, the population is never modified.

In summary, XCS is an online system which represents individuals as classifiers that contain single rules, uses adapted reinforcement learning techniques to evaluate the quality of these classifiers, employs a steady-state niche-based GA to discover new promising rules, and applies a fitness-based voting policy to infer the class of test instances. XCS process organization is based on the activation of classifiers to form match sets and action sets, and on the application of the parameter update procedure and the GA on these action sets or niches. In the next subsection, we explain how this process leads to the evolution of accurate rule sets.

3.1.6 Why Does XCS Work?

After defining the learning process and the reasoning mechanism of XCS, we now intuitively explain the mechanisms that let XCS evolve a set of maximally general and accurate classifiers. For this purpose, we revise the work by Butz et al. (2004b), and explain the five evolutionary pressures identified by the authors that lead the system to evolving a set of optimal classifiers. The explanations are maintained in an abstract level, and the details of the mathematical formulation are not provided. The user is referred to (Butz et al., 2004b) for the details.

Butz et al. (2004b) identified five evolutionary pressures that guide the learning process in XCS:

1. The set pressure.
2. The mutation pressure.

3. The deletion pressure.
4. The subsumption pressure.
5. The fitness pressure.

In what follows, each item is briefly explained.

Set pressure. The *set pressure* is mainly due to the combination of the application of the GA in niches and the deletion in the whole population. This pressure was early explained by Wilson (1995), who formulated the following hypothesis: if two classifiers are equally accurate but have different generalizations, then the most general one will participate in more action sets, having more reproductive opportunities and finally displacing the specific classifier. This hypothesis was later investigated by Kovacs (1997), defining the *optimality hypothesis*, and formalized by Butz et al. (2004b). In brief, this supports the fact that the most general and accurate classifiers will take over their niches, displacing both over-general and most specific, accurate classifiers.

Mutation pressure. Whereas the set pressure moves the population toward generality and accuracy, Butz et al. (2004b) identified that mutation, in its own, causes the population to tend to more specific classifiers. That is, mutation changes the value of a variable, which can take one value from the ternary alphabet {0,1,#}. As two of these values are specific, i.e., {0,1}, and the last one is general, mutation pushes toward a distribution of 66.7%/33.3% of specific/general bits. Obviously, the intensity of this pressure depends on the period of application of the GA and the mutation probability.

Deletion pressure. The population-wise deletion operator removes classifiers depending on their action set size estimate and their fitness. As classifiers that belong to large action sets are given a higher deletion probability, the operator makes pressure towards even distribution of classifiers in the different system niches. Furthermore, the deletion operator also pushes toward removing classifiers with low fitness, driving the search toward the fittest individuals.

Subsumption pressure. The subsumption pressure pushes towards generalization inside the niche. Once several accurate classifiers have been found, subsumption deletion causes the system to prefer the maximally general classifiers over the most specific ones. That is, GA subsumption checks, for each offspring, if there exists any accurate classifier in [A] whose condition includes the offspring's condition; if so, the numerosity of this classifier is incremented. Therefore, subsumption produces an additional pressure toward generalization.

Fitness pressure. Finally, the fitness pressure is present in all the mechanisms of XCS, and influences the four aforementioned pressures as well. In general, fitness pressure pushes the population from over-general to more specific and accurate classifiers. It interacts with the other pressures since selection, mutation, and subsumption depend on classifier's fitness. In summary, the interaction of the five pressures drives the population toward a population of accurate maximally general classifiers.

With the explanation provided in this section we have covered the technical details and have glimpsed the ideas that explicate why XCS works. Notice that, in essence, XCS is a general

architecture—which evaluates rules online and uses a robust search mechanism to discover new promising rules—rather than a specific architecture particularly designed for solving a concrete set of tasks. For this reason, the learning architecture of XCS has been applied—sometimes with few modifications—to solve different types of problems. In the following section, we present one of this system modifications that specializes XCS to deal with data classification tasks more efficiently. The new architecture is addressed as the *supervised classifier system* (UCS).

3.2 The UCS Classifier System

UCS is an accuracy-based LCS that inherits the main components of XCS, but specializes the online learning architecture for classification tasks (Bernadó-Mansilla and Garrell, 2003). The aim of the system is to take advantage of having the class of the training instances, so focusing the exploration process toward classifiers that predict the correct class accurately. Therefore, UCS does not evolve a complete action map—including classifiers with low prediction and error—, but it does create the *best action map*, which consists of a set of maximally general and accurate classifiers that predict the correct class. With this modification, UCS is expected to

1. evolve a solution quicker than XCS, since it only explores the correct class, and
2. require less population to store the solution, as it only needs to maintain the best action map.

Furthermore, UCS adapts the classifier's parameters to supervised learning. As follows, we review the learning mechanism of UCS, especially focusing on the novelties with respect to the XCS architecture. Therefore, we first revisit the knowledge representation, which introduces new parameters to assess the quality of the rules. Then, we analyze the differences in the learning interaction, rule evaluation, rule discovery, and reasoning mechanism to infer the class of new input instances.

3.2.1 Knowledge Representation

As XCS, UCS evolves a population [P] of classifiers, where each classifier contains a rule and a set of parameters. The rule representation is copied from XCS. Therefore, rules consist of a condition that advocates a class. Besides, the classifiers have the following parameters:

1. The accuracy acc, which maintains an average of the proportion of matching examples that have been correctly classified by the rule.
2. The fitness F, which is computed as a function of the accuracy.
3. The correct set size cs, an estimate of the size of the correct sets in which the classifier has participated (see section 3.2.2).
4. The experience exp, which reckons the number of examples that the classifier has matched during his life.
5. The numerosity n, which indicates the number of copies of the classifier in the population.

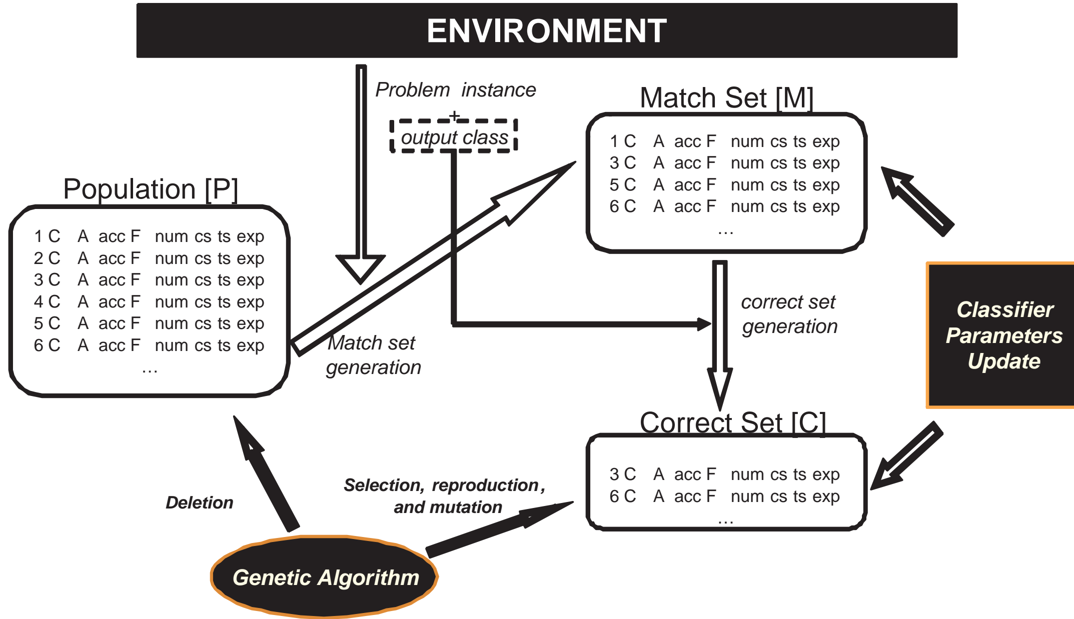


Figure 3.2: Schematic of the process organization of UCS.

Therefore, UCS inherits the experience and the numerosity parameters, redefines the accuracy and the fitness parameters, and introduces the correct set size parameter. Note that one of the key differences with respect to XCS is that, in UCS, fitness is based on accuracy, which is directly computed as the true proportion of correct predictions of the given rule. With the modification of the knowledge representation in mind, the next subsections describes the changes on the learning interaction proposed by UCS to adapt the online learning architecture to supervised learning tasks.

3.2.2 Learning Interaction

Figure 3.2 illustrates the process organization of UCS, which redefines the process organization of XCS according to a supervised learning scheme. The main difference with respect to XCS is that, in UCS, the class of each learning instance is provided by the environment. Therefore, the learning architecture takes advantage of this information to explore only the class of each sampled input example. As proceeds, this procedure is explicated in detail.

As XCS, UCS starts the *exploration phase* with an empty population. At each learning iteration the system receives a new input example e with its class c. Then, the system creates the *match set* [M], which contains all the classifiers in the population [P] whose condition matches e. Next, all the classifiers in [M] that predict the class c form the *correct set* [C]. If [C] is empty, the covering operator is activated, which creates a new classifier whose condition is generalized from e (as in XCS), and whose predicted class is set to c. Hence, covering aims at discovering a single classifier that predicts the sampled input instance correctly; besides, some generalization is added by setting each variable to '#' with probability $P_{\#}$. The parameters of the new classifier

are set to: $exp = 1$, $num = 1$, $cs = 1$, $acc = 1$ and $F = 1$. As fitness and accuracy are estimated from a single instance—and so, the estimate may be poor—UCS does not let young classifiers have a strong participation in the genetic selection and the reasoning mechanism until they do not receive a minimum number of parameter updates.

After this, the parameters of all classifiers in the match set are evaluated, and eventually, the correct set—which defines a niche of classifiers with similar conditions and the same predicted class—receives a genetic event. Note that the parameter update procedure is applied to [M] instead of to [C]. UCS moves this procedure to the match set since the accuracy of all the matching classifiers that predict the input instance wrongly needs to be decreased. Also, notice that the correct set size acts as a niche where the genetic algorithm is applied, following the same idea of XCS. These two procedures are detailed in the following two subsections.

3.2.3 Classifier Evaluation

Each time a classifier participates in a match set, its experience, accuracy, and fitness are updated. Firstly, the experience is increased. Then, the accuracy is computed as the percentage of correct classifications:

$$cl.acc = \frac{number\ of\ correct\ classifications}{cl.exp}. \tag{3.12}$$

Thus, accuracy is a cumulative average of correct classifications over all matches of the classifier. Next, fitness is updated according to the following formula:

$$cl.F_{micro} = (cl.acc)^{\nu}, \tag{3.13}$$

where ν is a constant set by the user that determines the strength pressure toward accurate classifiers (a common value is 10). Thus, differently from XCS, fitness is calculated individually for each micro-classifier, and it is not shared. The fitness of a macro-classifier $cl.F_{macro}$ is obtained by

$$cl.F_{macro} = cl.F_{micro} \cdot cl.n. \tag{3.14}$$

Finally, each time the classifier participates in [C], the correct set size $cl.cs$ is updated. $cl.cs$ is computed as the arithmetic average of the sizes of the correct sets in which the classifier has taken part.

Once the parameters of the classifiers in [M] have been evaluated, the GA can be applied to the current correct set. The following subsections explain this process.

3.2.4 Classifier Discovery

UCS uses a steady-state niche-based GA as the primary search mechanism to discover new promising rules. The GA is applied to [C] following the same procedure as in XCS. Here, we briefly review this process and explain in more detail the rule deletion mechanism, which is slightly modified with respect to XCS's one.

The GA is triggered in the current correct set if the average time since its last application to the classifiers in [C] is greater than θ_{GA}. If so, the GA selects two parents from [C]. The

two selection schemes of XCS are also applicable here: proportionate selection and tournament selection. The only difference is that the fitness of young classifiers ($exp < \theta_{del}$) is divided by θ_{del} to avoid their influence in the selection process. Then, the two parents are copied, creating two new offspring, which are recombined and mutated with probabilities χ and μ respectively. The crossover and mutation operators are directly inherited from XCS (see section 3.1.4). The parameters of the offspring are initialized as follows. The experience and the numerosity are set to 1. The accuracy and the fitness are also set to 1 (these parameters will go quickly to their real values as they participate in successive match sets). Finally, cs is set to the size of the current correct set.

Finally, both offspring are introduced into the population. First, each offspring is checked for subsumption with the classifiers in [C]. The subsumption mechanism is adapted from XCS as follows: if one of the classifiers cl in [C] is sufficiently experienced ($cl.exp > \theta_{sub}$), accurate ($cl.acc > acc_0$) and more general than the offspring, then the offspring is not introduced into the population and the numerosity of the subsumer classifier is increased. If the offspring cannot be subsumed, it is inserted in the population, deleting another classifier if the population is full. The deletion probability p_{del} of a rule cl is calculated as:

$$cl.p_{del} = \frac{cl.d}{\sum_{\forall cl_i \in [P]} cl_i.d}, \tag{3.15}$$

where

$$cl_d = \begin{cases} \frac{cl.cs \cdot cl.n \cdot F_{[P]}}{cl.F_{micro}} & \text{if } cl.exp > \theta_{del} \text{ and } cl.F_{micro} < \delta F_{[P]}; \\ cl.cs \cdot cl.n & \text{otherwise,} \end{cases} \tag{3.16}$$

where δ and θ_{del} are configuration parameters, and $F_{[P]}$ is the average fitness of the population. In this way, deletion will tend to balance resources among the different correct sets, while removing low-fitness classifiers from the population. As fitness is computed from the proportion of correct classifications of a classifier, classifiers that predict wrong classes are not maintained in the population, and so, only the best action map is evolved.

The whole process is iterated during several learning steps in which a new instance is sampled and the processes of match set creation, parameter evaluation, and genetic algorithm application take place. After this, the system results in a population of highly general and accurate classifiers, which are used to infer the class of new input instances. The next subsection provides the reasoning mechanism implemented in UCS.

3.2.5 Class Inference in Test Mode

In test mode, a new input example e is provided, and UCS has to predict the associated class. For this purpose, UCS implements a reasoning mechanism which is similar to the one of XCS. That is, firstly, the match set is created. Then, all classifiers in [M] emit a vote, weighted by their fitness, for the class they predict. The vote of young classifiers (i.e., $exp < \theta_{del}$) is decreased by multiplying its vote by exp/θ_{del} to diminish their influence with respect to more experienced classifiers. The most-voted class is chosen. New inference schemes have been proposed in Brown et al. (2007), showing that the current inference schemes of XCS is one of the best among the compared ones. Under test mode, the population of UCS does not undergo any change; that is, all update and search mechanisms are disabled.

So far, we have detailed the key differences between the architectures of XCS and UCS. In essence, UCS follows the same mechanisms of XCS, but introduces some little modifications to take advantage of knowing the class of the training examples. With this new architecture, UCS is not expected to perform better than XCS, but to be able to evolve a solution spending less computational resources. That is, UCS needs to evolve and maintain a lower number of classifiers—since UCS does not evolve low rewarded rules— and is expected to solve the problem more quickly than XCS, since it only explores the "correct class". Nonetheless, note that the key concepts of XCS, such as the accuracy-based approach, the incremental parameter update procedure, and the steady-state niche-based GA are still present in UCS. In the following subsection, we argue that the same ideas introduced in section 3.1.6 to explain why XCS works are still valid in UCS.

3.2.6 Why does UCS work?

As XCS, UCS is guided by the following five evolutionary pressures:

1. The set pressure.
2. The mutation pressure.
3. The deletion pressure.
4. The subsumption pressure.
5. The fitness pressure.

The main differences with respect to XCS is that, now, the fitness definition differs. Therefore, the system no longer pressures toward obtaining classifiers with low prediction error, but toward highly accurate classifiers. The consequences of this is that, as already discussed, UCS evolves the best action map instead of a complete action map. The other pressures do not need to be redefined as a consequence of the change in the architecture. That is, the set pressure pushes [P] toward the most general classifiers. The mutation pressure pushes toward specificity. The deletion pressure maintains an even distribution of classifiers in the different niches, giving more deletion probability to the classifiers with the lowest fitness with respect to the average fitness of the population. And finally, the subsumption pressure makes UCS prefer the most general classifiers that are still accurate to more specific, accurate classifiers. The overall interaction of these pressures, as in XCS, guides the search toward maximally general and accurate classifiers.

3.3 Rule Representations for LCSs

Thus far, we have described XCS and UCS with a ternary rule representation. During the last few years, several new rule representations have been introduced to XCS and UCS to let the systems deal with real-world problems such as interval-based representations (Wilson, 2000, 2001), hyper elipsoidal representations (Butz et al., 2006, 2008), and convex hulls (Lanzi and Wilson, 2006). Other more general approaches that have been used to codify the classifiers rules are neural networks (Bull and O'Hara, 2002), messy representations (Lanzi, 1999a) LISP

s-expressions (Lanzi and Perrucci, 1999), and gene expression program representations (Wilson, 2008). Besides, fuzzy representations have been designed for some Michigan-style LCSs (e.g., see Valenzuela-Rendón (1991)); fuzzy representations will be presented in detail in chapter 8. In here, we introduce one of the most-used representations to deal with continuous attributes in XCS, that is, the interval-based representation. As follows, we first provide some historical remarks about the different proposals of interval-based rule representation for LCSs, and introduce the one used in the present work.

3.3.1 From the Ternary to the Interval-based Rule Representation in LCSs

Initially designed with a ternary representation, LCSs faced a new challenge when dealing with continuous or quantitative attributes, which are often present in real-world problems. That is, the ternary rule representation was not suitable for directly dealing with continuous data. Data preprocessing techniques could be used to transform the continuous values into discrete values; nonetheless, this could result in an undesirable loss of information. Recently, interval-based rule representations have been designed to to effectively deal with continuous attributes. The most significant of these representations are reviewed as follows.

Wilson (2000) designed one of the first interval-based representation for XCS, addressed as *center-spread representation*. The center-spread representation codifies each rule variable with a pair of values (c_i, s_i) that defines a rectangle with center in c_i and spread s_i. Besides, the representation has to satisfy the constraint that $c_i - s_i/2 \geq max_i$ and $c_i + s_i/2 \leq min_i$, where max_i and min_i are respectively the maximum and the minimum values that the attribute can take.

This representation empirically demonstrated to be able to evolve accurate models in artificial problems with continuous attributes. Nevertheless, the truncation caused by guaranteing the constrain on the maximum and the minimum values might result in an inefficient exploration of the feature space, as later shown by Stone and Bull (2003). To overcome this problem, Wilson (2001) presented another interval-based representation, referred to as *min-max representation* in which each attribute codifies the lower ℓ_i and the upper u_i limit of the interval of values where the attribute is applicable. Although the effects of truncation are not present, this representation still has the problem of invalid intervals eventually caused by the genetic operators. That is, genetic operators may generate intervals where $\ell_i > u_i$, in which the classifier would not match any input instance. This situation could be fixed in several ways by modifying the value of the interval bounds.

A simple approach to deal with this effect was proposed by Stone and Bull (2003), who introduced the *unordered-bound representation*. The unordered-bound representation proposes to use the min-max representation but without prefixing which of the two bounds are the upper bound and the lower bound. That is, the unordered-bound representation codifies each variable as an interval (p_i, q_i); the smaller of these two values is considered as the lower bound of the interval and the larger value is considered the upper bound. This has been, probably, the most used representation for continuous attributes in the last few years.

New efforts in the definition of representations for continuous attributes were made after the presentation of the unordered-bound representation. For example, Dam et al. (2005) argued that the unordered-bound representation produces large changes in the semantics of the intervals

when an operator exchanges the lower bound with the upper bound of an interval. Therefore, this may thwart the correct propagation of the building blocks of the problem. In order to solve this, Dam et al. (2005) proposed to represent an attribute with the pair (m_i, p_i), where m_i is the lower bound of the interval and p_i is a proportion used to compute the length of the interval s_i as

$$s_i = p_i(p_{max} - m_i), \tag{3.17}$$

in which p_{max} is the maximum value that the attribute i can take. Thus, the pair (m_i,p_i) can easily be translated to the lower bound ℓ_i and upper bound u_i of the interval by recognizing that

$$\ell_i = m_i, \tag{3.18}$$

$$u_i = m_i + s_i \tag{3.19}$$

However, the empirical results did not clearly show an improvement with respect to the unordered-bound representation. For this reason, we used the unordered-bound representation in the present work. The next section provides more details about this representation.

3.3.2 The Unordered Bound Representation

We now describe the unordered-bound representation in more detail and explain how the genetic operators were adapted to deal with the new representation. As aforementioned, the new representation codifies each variable with an interval (p_i, q_i), where the minimum value between the two bounds represents the lower bound, and the maximum value represents the upper bound. For example, a classifier whose condition is defined by the two variables <[1,2], [10,8]> matches any input instance whose first variable ranges in [1,2] and its second variable ranges in [8,10]. As proceeds, we explain how the covering operator, the different genetic operators that deal with the representation—i.e., crossover and mutation—, and the subsumption operator are redefined to deal with the new representation. For simplicity, we assume that all the input attributes have been normalized, and so, that their values range in [0,1].

Covering Operator

The covering operator creates a new classifier whose condition is generalized from the input example e. For this purpose, the interval of each variable i of the new classifier is initialized as

$$p_i = e_i - rand(0, r_0) \quad \text{and} \tag{3.20}$$

$$q_i = e_i + rand(0, r_0), \tag{3.21}$$

where r_0 is a configuration parameter ($0 < r_0 \leq 1$), and $rand(0, r_0)$ returns a random number between 0 and r_0. Therefore, this operator creates an interval that includes the value of the corresponding attribute, and r_0 controls the generalization in the initial population (it is equivalent to $P_{\#}$ in the ternary representation). An example of covering is graphically illustrated in figure 3.3.

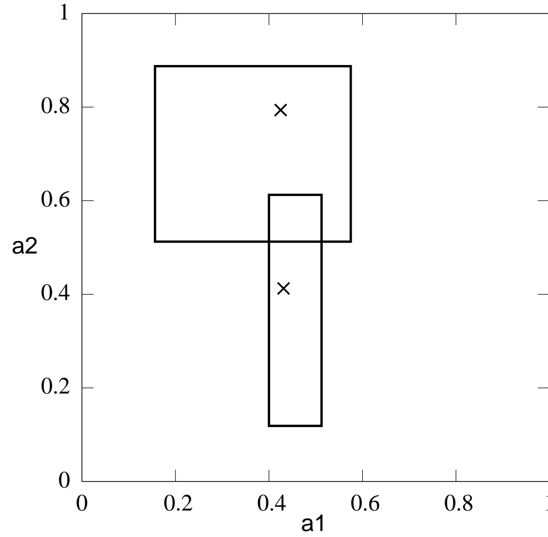


Figure 3.3: Example of covering in the hyper rectangular representation.

Parents		Offspring
$< [0.20, 0.80], [0.45, 0.65] >$	$\Longrightarrow$	$< [0.20, 0.85], [0.25, 0.65] >$
$< [0.60, 0.85], [0.25, 0.75] >$		$< [0.60, 0.80], [0.45, 0.75] >$

Table 3.1: Example of two-point crossover, in which the two cut points are in the middle of each interval.

Crossover Operator

In real-world problems, two-point crossover is usually applied. It randomly selects two cut points, which can occur either between or within an interval predicate. Then, the offspring are created by shuffling the information of both parents. The process is detailed in the example of table 3.1, in which the two parents are crossed, selecting a cut point in the middle of each interval and generating two new offspring. Figure 3.4 visually illustrates this example in the feature space.

Mutation Operator

The mutation operator is applied to each of the bounds of the interval. If it decides to change an interval bound, this is mutated by adding a random value that ranges in $(-m_0, m_0)$, where m_0 is a configuration parameter. Figure 3.5 shows an example of the mutation operator.

Subsumption Operator

To consider a classifier as a candidate subsumer, the same conditions defined in the ternary representation need to be satisfied. That is, the subsumer classifier has to be experienced

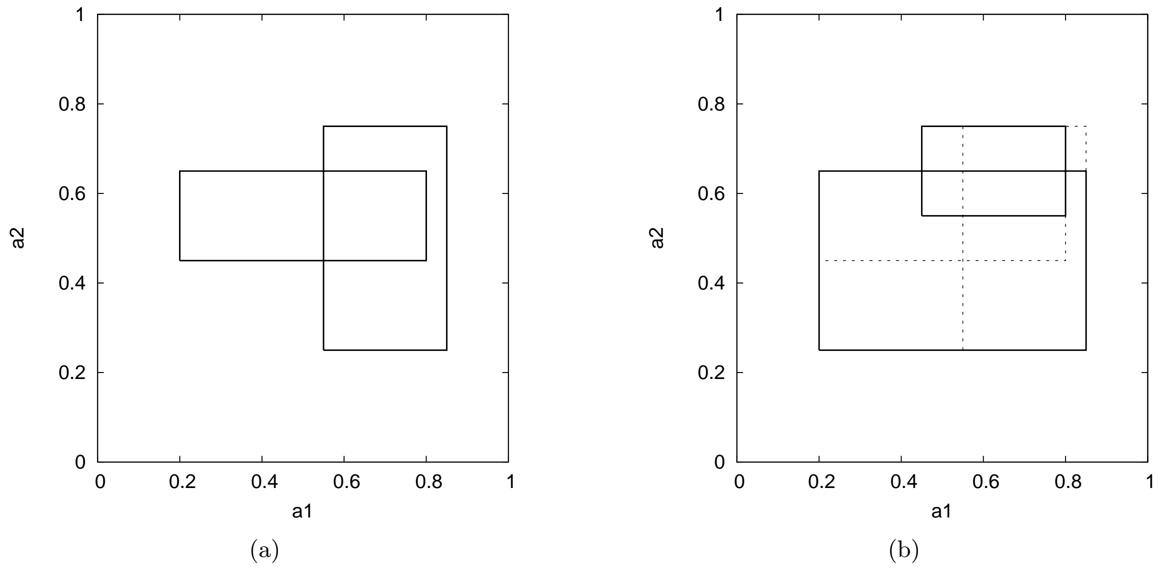


Figure 3.4: A crossover example. (a) plots the two parents and (b) shows the offspring resulting from two cut points occurring in the middle of each interval.

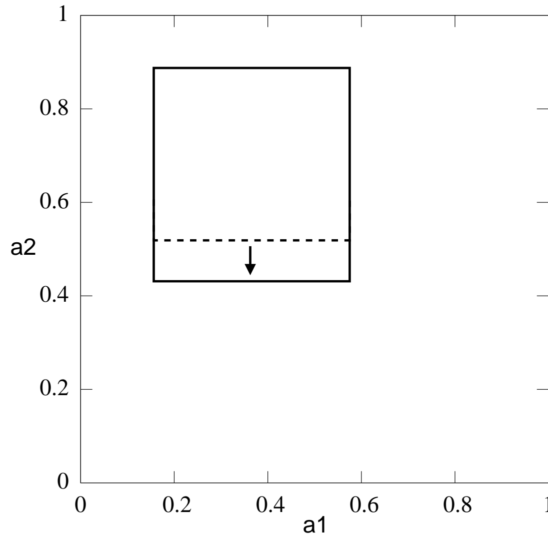


Figure 3.5: Example mutation in the hyper rectangle representation.

enough, accurate, and its condition has to include the condition of the subsumed classifier. For an interval-based representation, the condition of rule cl_1 includes that of rule cl_2 if for each variable i: $cl_1.\ell_i \leq cl_2.\ell_i$ and $cl_1.u_i \leq cl_2.u_i$, where ℓ_i and u_i are respectively the smaller and greater value of the two bounds of the interval.

3.4 Summary and Conclusions

This chapter provided concise descriptions of XCS and UCS, which can be utilized as implementation guidelines. Besides, we explained the evolutionary pressures that guide both systems to evolve a minimal set of maximally general and accurate classifiers. Finally, we presented the interval-based representation used by the two LCSs to deal with continuous attributes.

During the explanation, we highlighted the differences between XCS and UCS, which are due to the specialization of UCS for supervised learning. Basically, the two key aspects that UCS modifies with respect to XCS are:

1. The fitness computation.

2. The exploration scheme.

The fitness computation procedure of UCS introduces two changes with respect to the one in XCS: (1) UCS computes the fitness based on the rule's accuracy on classifying the input instances instead of on the accuracy of the rule's prediction and (2) UCS does not share the fitness among the rules in the same niche. The former modification makes UCS push toward rules that predict all the matching instances correctly; therefore, the system evolves the best action map. Conversely, XCS evolves rules that are accurate in the prediction reward, regardless of whether they predict the correct class; thence, XCS builds the complete action map. Therefore, UCS spends less computational resources with respect to XCS since UCS only evolves and maintains the rules that predict the correct class. On the other hand, UCS does not use a fitness-sharing scheme; instead, it computes the fitness as a power of the raw accuracy. The advantages of not sharing fitness, if neither, are not clear and demand further investigation.

The exploration of UCS was also modified to speed up the convergence in classification problems. That is, as long as the system only needs to evolve the best action map and as the class is made available with each input example, UCS only explores the correct class. Hence, in addition to using less computational resources to store the final population, the system is also expected to obtain the optimal solution more quickly than XCS.

In the next chapter, we further investigate the impact of the architectural changes proposed by UCS. We first carefully study the effect of not sharing fitness in UCS. For this purpose, we design a fitness-sharing scheme, similar to the XCS's one, and incorporate it to UCS. Then, we empirically analyze the advantages of the new fitness computation method. Thereafter, we examine the behavior of UCS with respect to the XCS one in a collection of boundedly difficult problems. Thus, we include XCS in the comparison and analyze how the new fitness computation and the new exploration scheme of UCS affects the learning performance and the convergence to the optimal population.

Chapter 4

Revisiting UCS: Fitness Sharing and Comparison with XCS

The previous chapter explained the mechanisms of XCS and UCS in detail and reviewed the most important differences in the architecture of both systems. In brief, UCS performs a more focused search since its architecture is specialized for supervised learning tasks. The two main differences introduced by UCS affected (1) the fitness computation and (2) the exploration regime. With these changes, UCS was expected to evolve an accurate solution spending less computational resources than XCS[1]. These initial hypotheses were already supported by the experimental results provided by Bernadó-Mansilla and Garrell (2003). In there, the authors compared the original scheme of UCS with XCS on a collection of three *boundedly difficult problems*. The experimental results enabled the authors to emphasize the differences between both architectures and to demonstrate that, in general, UCS could solve these problems more quickly than XCS. Although these were promising results, there were still some open issues that needed to be addressed. The most important aspect in this list is *fitness sharing*. That is, the lack of fitness sharing in UCS was identified as a potential weakness of the system. Nonetheless, no further study about the effect of not sharing fitness has been conducted so far.

The purpose of this chapter is to study whether fitness sharing can provide benefits to UCS and further compare UCS with XCS in an extension of the collection of boundedly difficult problems used by Bernadó-Mansilla and Garrell (2003). To achieve this, we design a fitness-sharing scheme which is inspired by the XCS's one. Then, we test the three systems on four boundedly difficult problems. The experimental results illustrate the benefits of fitness sharing and highlight how the modifications introduced in the learning architecture of UCS affect the system's behavior with respect to the XCS's one in classification problems.

The remainder of this chapter is organized as follows. In section 4.1, we introduce the concept of fitness sharing and motivate its use in UCS by highlighting its importance in the GAs and LCSs realms. Section 4.2 specifies the new fitness-sharing scheme for UCS and section 4.3 draws the analysis methodology. Section 4.4 compares UCS with fitness sharing with the original UCS, showing the benefits of the fitness-sharing scheme. Section 4.6 extends the comparison by analyzing the differences between UCS and XCS. Finally, section 4.6 gathers the key observations

[1]Note that UCS can only be applied to supervised learning, while XCS is a much broader architecture that can be used for both supervised learning and reinforcement learning in general

drawn from the experiments, and section 4.7 summarizes and concludes this chapter. Appendix A supplies a full explanation of all the problems used along the comparison.

4.1 Fitness Sharing in GAs and LCSs

Before proceeding with the description of the fitness-sharing scheme designed for UCS, we first introduce the concept of fitness sharing and how this has been used in both GAs and LCSs. Holland (1975, 1992) initially presented the concept of fitness sharing as a way to accomplish *niching*. In his early work, Holland discussed the concept of limiting the number of individuals that occupy a niche. The underlying idea is that if each niche had associated a particular payoff or objective fitness, and if each individual in this niche were forced to share this payoff with the other individuals in the same niche, then a stable situation, where each niche contains approximately the same number of individuals, would arise. Therefore, niching methods that distribute the payoff among the individuals of the same niche are addressed as *fitness-sharing* methods.

In the GA field, fitness sharing has been used to maintain and evolve diverse solutions in multi-modal optimization problems. Goldberg and Richardson (1987) initially introduced the concept of explicit fitness sharing into GAs to optimize multi-modal functions. The proposed scheme derated the fitness of an individual by an amount related to the number of similar individuals in the population. More specifically, the method computed the *shared fitness* f_s^i of an individual i, whose objective fitness is f^i as

$$f_s^i = \frac{f^i}{\sum_{j=1}^{n} sh(d(i,j))}, \tag{4.1}$$

where sh is a function of the distance d between two solutions. sh returns '1' if the elements are equal, and '0' if they exceed some threshold of dissimilarity, σ_{share}. That is, if the distance between two solutions is greater than σ_{share}, $sh = 0$, indicating that none of the two individuals affects the fitness of the other. One of the most usual sharing functions is:

$$sh(d) = \begin{cases} 1 - \left(\frac{d}{\sigma_{share}}\right)^{\alpha} & \text{if } d < \sigma_{share}, \\ 0 & \text{otherwise;} \end{cases}$$

where α is a configuration parameter that permits regulation of the proximity of the solutions that are considered to be in the same niche. Further analysis investigated other sharing schemes and their capacities to solve multi-modal problems. For example, Deb (1989) empirically demonstrated that GAs with fitness sharing were able to solve a large variety of multi-modal functions. Theoretical models of the effect of different types of sharing can be found in (Mahfoud, 1995) and (Horn, 1997).

Fitness sharing has also been used in LCSs. Smith and Valenzuela-Rendón (1989) modeled a generational GA with infinite population size for LCSs and showed that fitness sharing was necessary to facilitate the coverage in difficult problems. In fact, the majority of new LCS designs present some form of niching or fitness sharing. This is the case of XCS. As seen in the previous chapter, XCS intrinsically performs niching by grouping the classifiers in action

sets. The system also incorporates fitness sharing since the classifier's fitness depends on the classifier's relative accuracy, and the relative accuracy of each classifier is computed with respect to the accuracies of all the other classifiers in the same niche (see section 3.1.3). Some more recent LCSs such as XCSF (Wilson, 2002b; Butz et al., 2008) also adopt the same, or a similar, fitness-sharing scheme.

Although fitness-sharing schemes have been shown to be beneficial to Michigan-style LCSs, UCS was originally designed without a fitness-sharing scheme. Fitness sharing was not incorporated in UCS to keep the initial architecture as simple as possible, which would enable a more detailed analysis of the system. Nonetheless, the combination of the niche-based GA application with the lack of resource sharing in UCS seems to be counterintuitive since both approaches have historically come tied together. Therefore, in this chapter, we provide a fitness-sharing scheme to UCS and empirically analyze its advantages over the original parameter update procedure. The next section introduces fitness-sharing scheme, similar to that of XCS, for UCS. Then, we present the experimental methodology, run UCS with both fitness sharing and the original parameter update procedure, and carefully analyze the results.

4.2 A New Fitness Sharing Scheme for UCS

XCS intrinsically performs niching by grouping similar classifiers in action sets. The niching is considered (1) by the GA, which selects and crosses classifiers from the same niche, and (2) by the parameter update procedure, which shares the fitness among all classifiers in the same niche. As does XCS, UCS performs niching in the first sense because it applies the GA to the niches. Nevertheless, in the original UCS, the resources are not shared among the classifiers in the same niche. Therefore, here, we propose to incorporate the resource sharing in UCS. As follows, a new fitness-sharing scheme for UCS, which is inspired by the one of XCS, is presented, reviewing how all the parameter update procedure works after introducing the new fitness computation. For the sake of clarity, in the remainder of the chapter, UCS without sharing is referred to as UCSns, and UCS with sharing is addressed as UCSs.

The new parameter update procedure works as follows. The experience (exp), the correct set size (cs), and the accuracy (acc) parameters are computed as in UCSns (see section 3.2). However, fitness is shared among all classifiers in [M]. First, a new accuracy $cl.k$ is calculated, which discriminates between accurate and inaccurate classifiers. For classifiers belonging to [M], but not to [C], the accuracy is set to zero; that is, $\forall cl \in ![C] cl.k = 0$. For each classifier cl belonging to [C], $cl.k$ is computed as follows:

$$cl.k = \begin{cases} 1 & \text{if} \quad cl.acc > acc_0; \\ \alpha \left(\frac{cl.acc}{acc_0}\right)^{\nu} & \text{otherwise.} \end{cases}$$

Then, the relative accuracy $cl.k'$ is calculated as

$$cl.k' = \frac{cl.k \cdot cl.n}{\sum_{cl_i \in [M]} cl_i.k \cdot cl_i.num}, \tag{4.2}$$

and the fitness is updated from $cl.k'$:

$$cl.F = cl.F + \beta(cl.k' - cl.F). \tag{4.3}$$

Let us note that, under this scheme, the computed fitness corresponds to the macro-classifier fitness, as the numerosities of the classifiers are involved in computation of the relative accuracy. Whenever the micro-classifier's fitness is needed, we divide this value by the numerosity of the classifier.

This parameter update procedure is incorporated into UCSs, replacing the traditional scheme. In the next sections, we empirically analyze the advantages provided by this new scheme.

4.3 Methodology

The experimental analysis consists of two separate parts. First, we compare UCSns with UCSs, focusing on the advantages provided by the fitness-sharing scheme. Then, XCS is compared with UCS with the aim of highlighting the practical impact caused by the architectural changes introduced by UCS. In brief, UCS introduces (1) a new fitness computation which is based on the classification accuracy instead of on the accuracy of the prediction and (2) a new exploration scheme, which only explores the class of the input examples and maintains the best action map instead of the complete action map (see chapter 3). Therefore, we aim at analyzing how these two modifications influence the learning process in classification problems. The methodology used in both comparisons is detailed in what follows.

To analyze the behavior of the three systems, we took a systematic approach and compared their performance on four artificial problems that gather some complexity factors said to affect the performance of LCSs (Kovacs and Kerber, 2001; Bernadó-Mansilla and Garrell, 2003): a) a binary-class problem, the *parity*; b) a multi-class problem, the *decoder*; c) an imbalanced multi-class problem, the *position* ; and d) a noisy problem, the *multiplexer with alternating noise* . Given a binary input of length ℓ, with k relevant bits ($k \leq \ell$), these problems are defined as follows. The parity problem returns the number of one-valued bits modulo two. The decoder problem gives the decimal value of the input as output. The position problem returns the position of the left-most one-valued bit; if the input does not contain any one-valued bit, it returns zero. The multiplexer problem takes the first $log_2\ell$ bits as the address bits, and the remaining bits as the position bits; then, the output is the value of the position bit referred by the decimal value of the address bits. We added noise to the training instances of the multiplexer problem by flipping the class of the input instances with probability P_x; the test instances are free of noise. For a detailed description of each problem the reader is referred to appendix A. The parity, the decoder, and the position problems were configured with input lengths ranging from $\ell = 3$ to $\ell = 9$. The multiplexer was configured with 20 input bits. These boundedly-difficult problems permit varying the complexity along different dimensions such as input length, size of the optimal population, specificity of the optimal classifiers, number of classes, and class imbalance ratio. For further information, the reader is referred to appendix A. UCSns, UCSs, and XCS were run with each problem.

We configured the systems as follows. We used a standard configuration for XCS: $P_{\#} = 0.6$, $\beta = 0.2$, $\alpha = 0.1$, $\nu = 5$, $\theta_{GA} = 25$, $\chi = 0.8$, $\mu = 0.04$, $\theta_{del} = 20$, $\delta = 0.1$. In UCS, the parameters that are shared with XCS took the same values, except for $\nu = 10$; additionally $acc_0 = 0.999$. In both cases, tournament selection was applied. Subsumption was activated in the genetic algorithm with $\theta_{sub} = 20$. The maximum population size of XCS and UCS was configured depending on the size of the optimal population that the systems were expected to

evolve. As explained in section 3.1, XCS evolves a complete action map [O], which consists of all rules with low error, regardless of whether they have high or low reward prediction. On the other hand, UCS evolves the best action map [B], which includes only highly rewarded rules. As proposed by Bernadó-Mansilla and Garrell (2003), we set population sizes to $N = 25 \cdot |[O]|$ in XCS and to $N = 25 \cdot |[B]|$ in UCS. All the results are averages of 25 runs with different random seeds.

We decided against using the training accuracy to evaluate the performance of the three LCSs since it does not provide enough evidence of effective genetic search, as highlighted by Kovacs and Kerber (2004). Instead of accuracy, the achieved proportion of the optimal action map %[O] was proposed by Kovacs and Kerber (2001) as being a better indicator of the progress of the genetic search. However, UCS and XCS evolve different optimal populations: XCS creates a complete action map, whereas UCS represents a best action map. To allow a fair comparison, we only consider the proportion of best action map %[B] achieved by each system. That is, we only count the proportion of consistently correct rules. To review the best action map of each of the four problems, the reader is referred to appendix A.

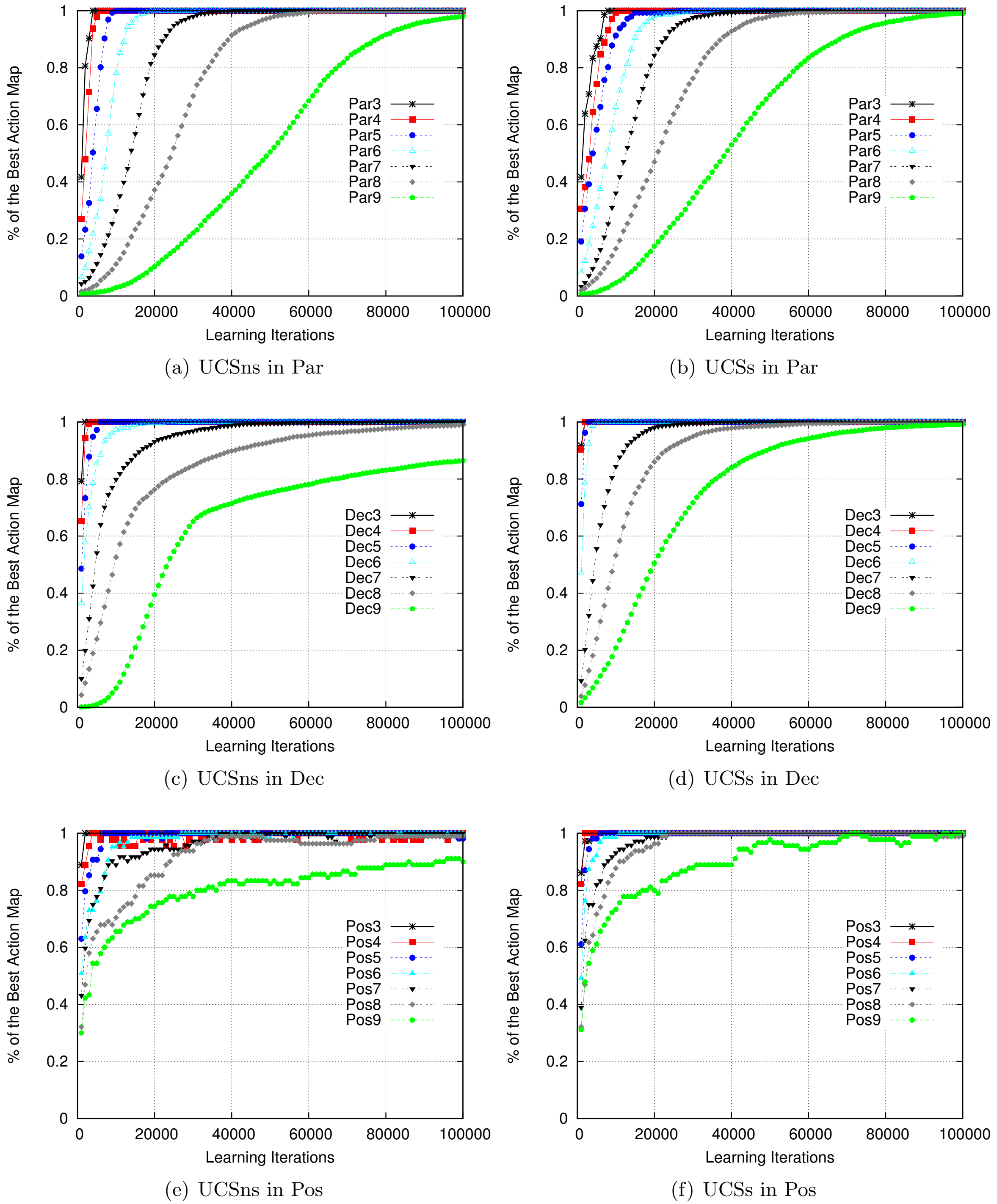
When required, we used statistical tests to compare the convergence curves between pairs of algorithms. As our aim was to compare pairs of learning systems, we used the non-parametric Wilcoxon signed-ranks test (Wilcoxon, 1945), which was fed with the performance measures taken during the learning process. For further details on this statistical test, the reader is referred to appendix B.

Having described the experimental methodology, we are now in position to analyze the results obtained on the four problems. The next section starts with the comparison of UCSns and UCSs. Later, XCS is introduced in the comparison.

4.4 Analyzing the Fitness Sharing Scheme in UCS

Figures 4.1 and 4.2 depict the proportion of the best action map %[B] achieved by UCSns and UCSs in the parity, the decoder, the position, and the 20-bit multiplexer with alternating noise problems. The results show that UCSs could discover the optimal population more quickly than UCSns in the parity, the decoder, and the position problems. These differences were significant in the decoder for $\ell > 4$, in the position for $\ell > 3$, and in the parity with any input length according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. Oppositely, these benefits could not be observed in the multiplexer with the highest levels of noise; that is, for $P_x = \{0.10, 0.15\}$, UCSns significantly outperformed UCSs. As proceeds, we examine these two different behaviors in more detail.

Benefits of fitness sharing. The improvement provided by the fitness-sharing scheme in the parity, the decoder, and the position problem—especially for the largest input lengths—can be explained as follows. Under fitness sharing, the progressive discovery of more accurate classifiers makes the fitness of less accurate, over-general classifiers that participate in the same correct sets decrease quickly. That is, when a better classifier is discovered, it gets a higher proportion of the shared fitness, causing a decrease in the fitness of less accurate classifiers that exist in the same niche. Therefore, fitness-sharing produces (1) a higher pressure toward the deletion of over-general classifiers and (2) a higher selective pressure toward the most accurate



(a) UCSns in Par

(b) UCSs in Par

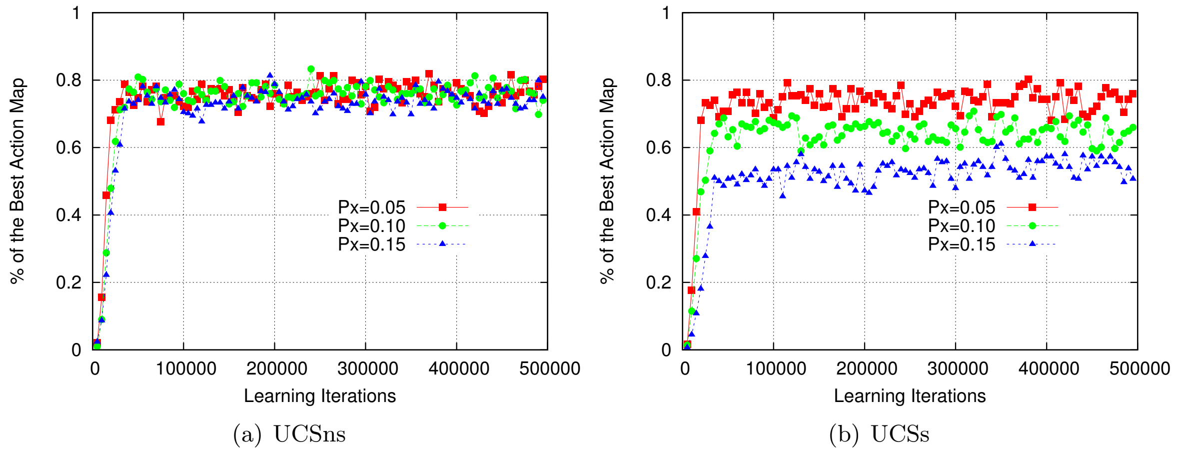
(c) UCSns in Dec

(d) UCSs in Dec

(e) UCSns in Pos

(f) UCSs in Pos

Figure 4.1: Proportion of the best action map achieved by UCSns and UCSs in the parity, the position, and the decoder problems.



(a) UCSns (b) UCSs

Figure 4.2: Proportion of the best action map achieved by (a) UCSns and (b) UCSs in the noisy 20-bit multiplexer with $P_x = \{0.05, 0.10, 0.15\}$.

classifiers in the GA. Without fitness sharing, over-general classifiers maintain the same fitness along the whole learning process.

The effect of fitness sharing was especially beneficial in the decoder and the position problems, while it was more modest in the parity problem. To explain the excellent results in the two former problems, let us first focus on the position problem and then extend the conclusions to the decoder problem. In the position problem, the system has to evolve a best action map that consists of classifiers with different degrees of generality ranging from a classifier in which all bits except one are set to '#' to a classifier with all specific bits (see section A.3). Note that UCS with fitness sharing could solve the position problem for all the tested input lengths, while UCS without fitness sharing was not able to discover the complete action map for $\ell = 9$. A more detailed analysis of the results showed that the final populations of UCSns did not contain the most specific optimal classifiers. Moreover, it was also identified that, even though the most specific optimal classifiers were created in some of the runs, they were lost during the genetic search. This behavior was a result of the combination of the occurrence-based reproduction with the fitness computation without sharing. To illustrate this, let us assume the case that the system discovers the most specific classifier for $\ell = 9$, that is, `000000000:0`. This classifier is activated once every 2^9 sampled instances. Once activated, this classifier may compete with a set of over-general classifiers. If fitness is not shared, the fitness of over-general classifiers remains constant regardless of whether the niche contains a high proportion of over-general classifiers or not. Therefore, as the number of over-general classifiers increases, it is more likely that the selection procedure chooses one of these over-general classifiers. This, coupled with the low activation rate of the most specific optimal classifiers, and so, their low reproductive opportunities, discourages their evolution. In these cases, the fitness-sharing scheme plays a key role to decrease the fitness of competing over-general classifiers, and so, promotes the maintenance of the most specific optimal classifiers.

These conclusions can be extended to the decoder problem. In this problem, all the optimal classifiers are maximally specific, and so, fitness sharing is necessary to promote the selection

Table 4.1: Accuracy and fitness of UCSns's classifiers along the generality-specificity dimension, depicted for the parity problem with $\ell = 4$.

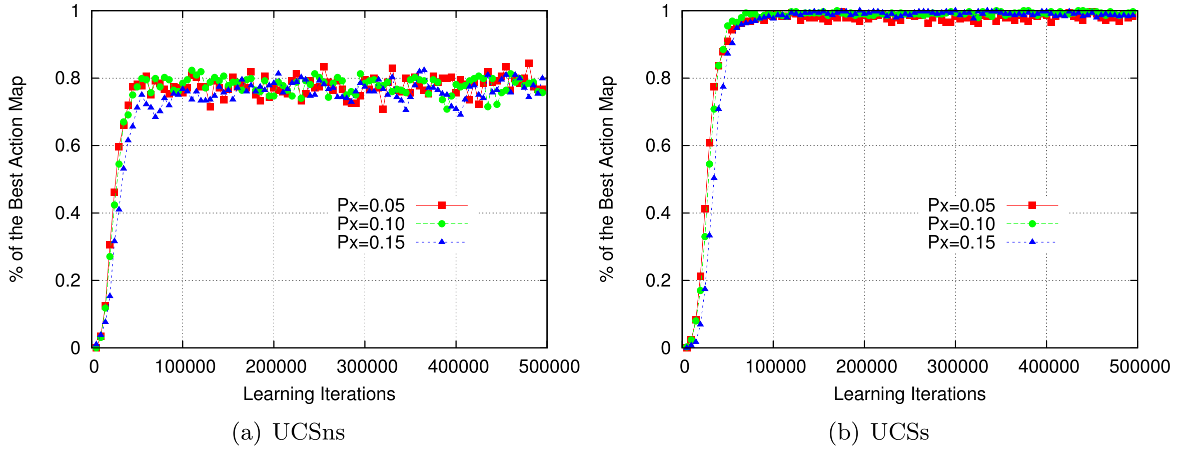
	Condition	Class	Accuracy	Fitness
1	####	0	0.5	0.00097
2	0###	0	0.5	0.00097
3	00##	0	0.5	0.00097
4	000#	0	0.5	0.00097
5	0000	0	1	1

of these specific classifiers. In general, fitness sharing appears to be crucial in problems where some of the optimal classifiers are activated with a low frequency. This is the case in problems that (1) contain class imbalances or (2) permit little generalization.

On the other hand, it is worth noting that the improvement in the parity problem was not as accentuated as the one in the decoder and position problems. This is due to the lack of *fitness guidance* toward optimal classifiers in this particular problem. That is, the best action map of the parity problem consists of classifiers in which all the variables are specific. Therefore, no generalization is allowed in the optimal population (see section A.1 for further details). Nonetheless, at the beginning of the run, the covering operator introduces generalization in the initial population, and UCS has to drive the population from over-generality to optimal classifiers. However, the fitness pressure does not correctly lead to specificity. That is, specifying one bit of an over-general classifier does not increase its accuracy unless all bits are specific. Therefore, although approaching the optimum, all the classifiers in the niche receive the same amount of resources. Only when an optimal classifier is discovered, its accuracy is set to 1, and so, the fitness of all the over-general classifiers that participate in the same niche is decreased. Differently, in the decoder and the position problem, the accuracy guided to the optimal classifiers; for this reason, the improvement in the convergence time provided by fitness sharing was larger in these two problems.

To further illustrate the lack of fitness guidance in the parity problem, table 4.1 shows the evolution that the most over-general classifier needs to go through to become an optimal classifier in UCSns for the problem with four input bits. At each step, one of the *don't care* bits is specified. Note that accuracy, and so, fitness, remain constant during all the process until the optimal classifier is achieved. However, we would expect that the specification of one bit should result in a fitter classifier, since it approaches the optimum classifier (in which all the bits are specific). Therefore, the fitness is not guiding toward optimal solutions. This problem is a type of *needle-in-a-haystack* problem, in which an optimal classifier can only be obtained randomly. For this reason, the improvement provided by fitness sharing, although existing and being statistically significant, is not as strong as the one in the decoder and the position problems.

Fitness sharing in noisy problems. Figure 4.2 shows that UCSns achieved higher performance than UCSs in the multiplexer problem, especially for the highest levels of noise. In particular, UCSns significantly outperformed UCSs for $P_x = \{0.10, 0.15\}$ according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. This behavior cannot be directly attributed to the fitness-sharing



(a) UCSns (b) UCSs

Figure 4.3: Proportion of the best action map achieved by (a) UCSns and (b) UCSs in the noisy 20-bit multiplexer with $P_x = \{0.05, 0.10, 0.15\}$ and using $\beta = 0.01$ and $\theta_{GA} = 100$.

scheme, but to the fact that, with the new parameter computation, UCSs calculates a windowed average of fitness by means of the learning parameter β. In noisy environments, the parameter averages oscillate and cannot stabilize properly, especially with the value of β employed in this configuration (i.e., $\beta = 0.2$). So, high levels of noise require low values of β. As UCSns computes fitness as a power of the accuracy, the parameter values of experienced classifiers remained steady.

To confirm this hypothesis, we repeated the same experiments but decreased β. That is, we configured UCSns with $\beta = 0.01$. In addition, we set $\theta_{GA} = 100$ to have more reliable parameters estimates before the GA triggered. Figure 4.3 shows the proportion of the best action map achieved by UCSns and UCSs. The results show a clear improvement of UCSs with respect to the original configuration. This supports our hypothesis that higher levels of noise require lower β values to allow for better parameter stabilization, coupled with higher θ_{GA} to let the genetic algorithm operate with better estimates. UCSs especially benefits from this, reaching almost 100% of the best action map in few iterations. Note that, for all the noise proportions, UCSs was still able to classify about 99% of the input instances correctly. That is, even when UCSs was trained with environments with 15% of alternating noise—i.e., 15% of the incoming instances are wrongly labeled—UCSs was able to classify nearly all the new free-noise instances correctly. This shows up the robustness provided by the inference scheme of UCS, which infers the class by means of a rule vote policy. The results of UCSns are practically the same as those obtained with the original configuration, as UCSns does not use β in the fitness estimate. Under this configuration, UCSs significantly outperformed UCSns for all the proportions of noise according to a Wilcoxon signed-ranks test at $\alpha = 0.05$.

Overall, this section evidenced the benefits of fitness sharing in the four boundedly difficult problems. This promotes the use of UCS with fitness sharing in the remainder of this thesis. In the next section, we extend this study and compare UCSs with XCS.

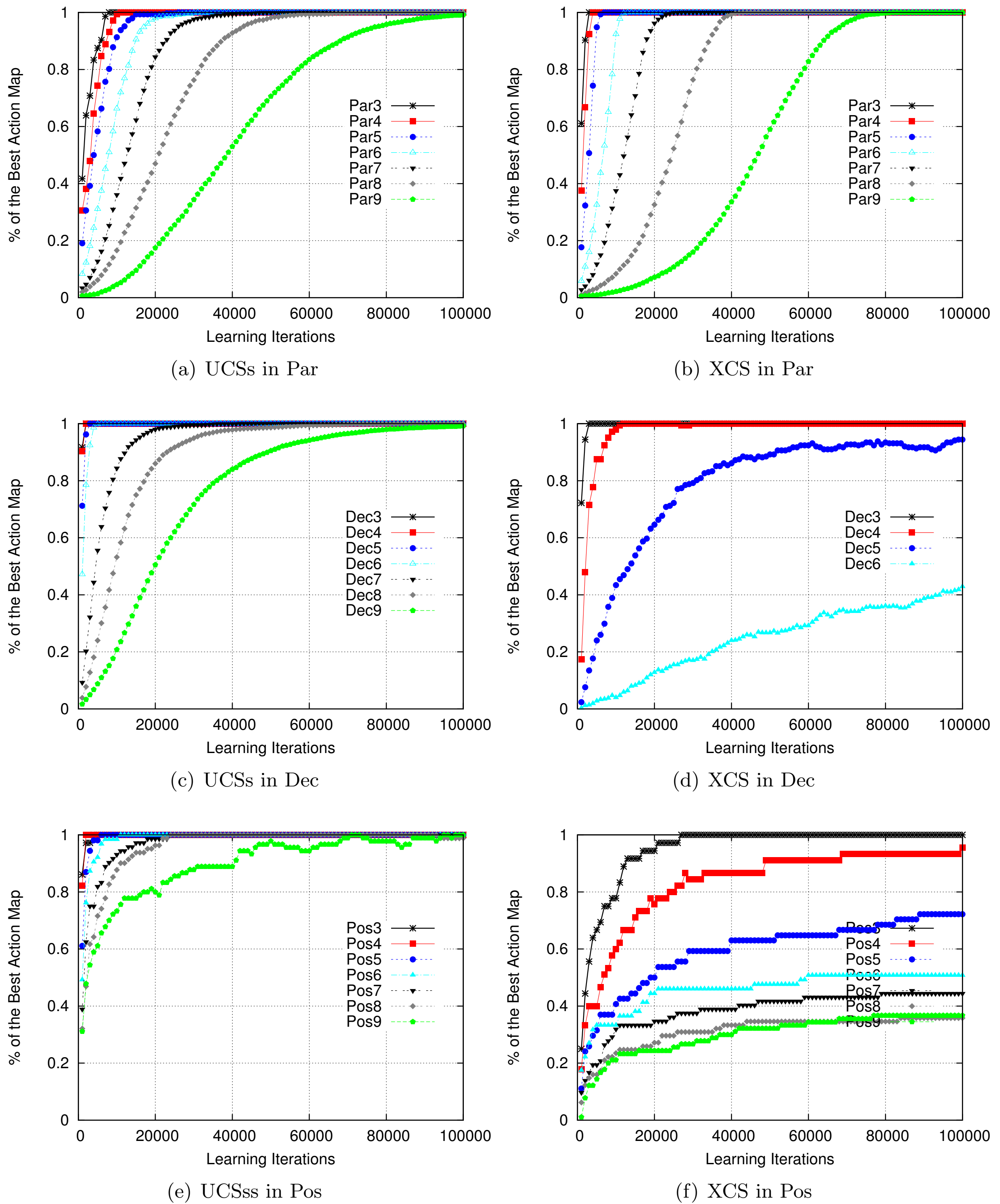
4.5 Comparing UCSs with XCS

This section introduces XCS in the comparison and empirically analyzes the effect of the architectural changes proposed by UCS with the aim of solving classification problems more effectively. Figures 4.4 and 4.5 show the proportion of the best action map achieved by UCSs and XCS on the four boundedly difficult problems. The results of UCSs are the same as those provided in the previous section. In all the problems, except for the parity problem, UCSs was significantly quicker in evolving the best action map than XCS according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. UCS's improvement on these problems is mainly result of (1) the exploration regime and (2) the new accuracy computation, and so, the fitness guidance of UCSs. As follows, these advantages, as well as the slightly poorer results obtained by UCSs in the parity problem, are analyzed in more detail.

Advantages due to the exploration regime. The first aspect that made UCSs evolve the optimal population more quickly than XCS is the exploration regime. That is, there are two reasons that explain, in general, why XCS requires more time than UCS to evolve the optimal population: (1) XCS needs to explore all the possible actions for a given input instead of only the class of the input and (2) XCS has to maintain the complete action map instead of only maintaining the best action map. This behavior is necessary in reinforcement learning problems where the consequences of each action have to be explored and modeled. Notwithstanding, this results in a delay in the convergence curves for the decoder, the position, and the multiplexer problems with respect to those of UCSs. Especially, note the large differences between XCS and UCSs in the decoder and the position problems. Although not as noticeable, UCSs also significantly outperformed XCS in the multiplexer problem with the two tested configurations according to a Wilcoxon signed-ranks test at $\alpha = 0.05$.

To exemplify the disadvantages of exploring the complete action map—as XCS does—, let us consider the decoder problem. The decoder problem is defined by 2^ℓ classes. As XCS explores uniformly each class, only 1 of each 2^ℓ explores will be made on the class of the input instance. The other $2^\ell - 1$ explores will be focused on classifiers that predict wrong classes. Therefore, the system will spend the proportion of $(2^\ell - 1)/2^\ell$ iterations to explore regions of the feature space that do not need to be explored in supervised learning problems. This issue takes on especial importance as the number of classes increases. This example is applicable to the other problems, but is especially important in the decoder and the position problems since the number of classes increases exponentially (for the decoder) and linearly (for the position) with the number of input bits.

On the other hand, the exploration regime seems not to provide UCSs with any advantage in the parity problem. That is, XCS was able to achieve the best action map in an amount of time that was significantly smaller than the time required by UCSs according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. Our hypothesis is that the exploration of consistently incorrect rules may help XCS discover consistently correct rules in this particular problem. That is, the parity problem has the particularity that, for each optimal classifier that predicts the correct class, there exists another one with the same condition but the wrong class in the complete action map. Therefore, mutation can easily generate a highly rewarded optimal classifier while exploring a niche with low rewarded classifiers. For example, if a consistently incorrect classifier such as `0001:0` is evolved, XCS may discover the consistently correct classifier `0001:1` by mutating the



(a) UCSs in Par

(b) XCS in Par

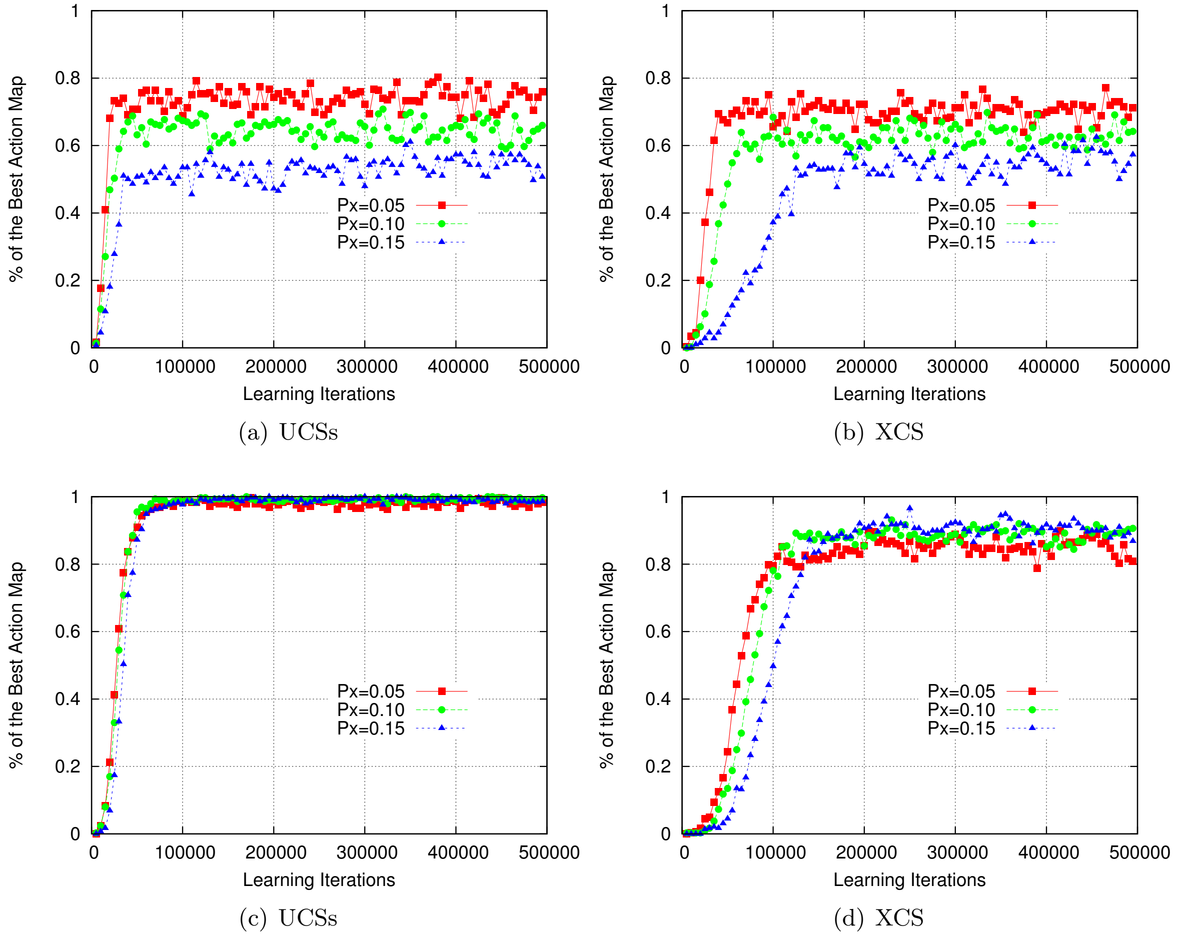
(c) UCSs in Dec

(d) XCS in Dec

(e) UCSss in Pos

(f) XCS in Pos

Figure 4.4: Proportion of the best action map achieved by UCSs and XCS in the parity, the position, and the decoder problems.

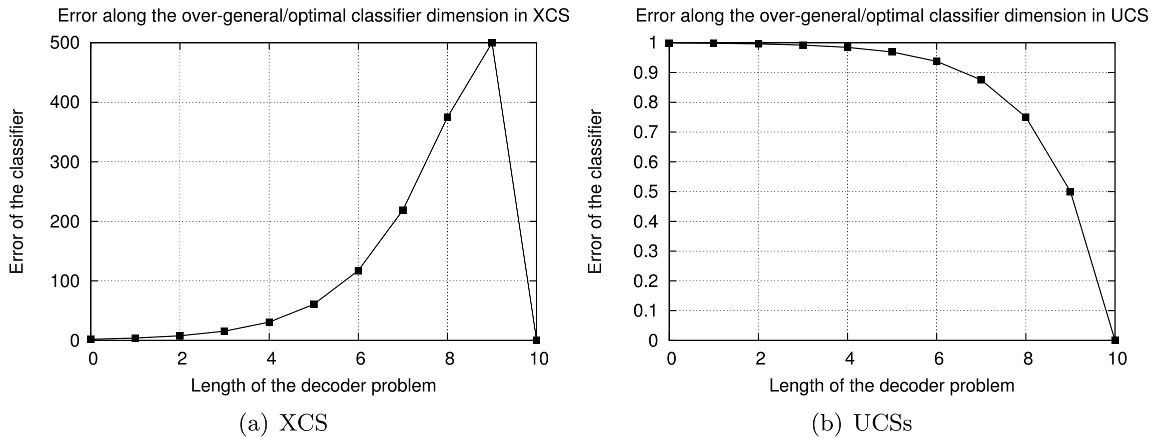


(a) UCSs (b) XCS (c) UCSs (d) XCS

Figure 4.5: Proportion of the best action map achieved by (a,c) UCSs and (b,d) XCS in the noisy 20-bit multiplexer with $P_x = \{0.05, 0.10, 0.15\}$ with (a,b) the original configuration and with (c,d) the original configuration but setting $\beta = 0.01$ and $\theta_{GA} = 100$.

class of the rule. Conversely, UCSs would never maintain an inconsistently correct rule, since the selection operator would never choose a rule with $acc = 0$, and the deletion procedure would eventually remove this rule. As a consequence, UCSs cannot benefit from exploring low rewarded niches. This, coupled together with the fact that XCS is configured with a larger population size, results in that XCS can evolve the optimal population more quickly than UCSs. Nonetheless, note that, still, UCSs uses less computational resources since it is configured with a smaller maximum population size.

Advantages due to the fitness guidance. Although the explore regime explicates why UCSs can converge more quickly than XCS to the best action map, this may not be enough to explain the large differences observed in the decoder and the position problems. Here, we show that, as indicated by Butz et al. (2003), the fitness computation of XCS may provide a deceptive guidance toward the obtention of optimal classifiers in these two problems—and problems with



(a) XCS (b) UCSs

Figure 4.6: Error of XCS's classifiers along the over-general/optimal classifier dimension. The curve depicts how the error of the most over-general classifier `##########:0` evolves as the bits of the classifier are specified, until obtaining the maximally accurate rule `0000000000:0`.

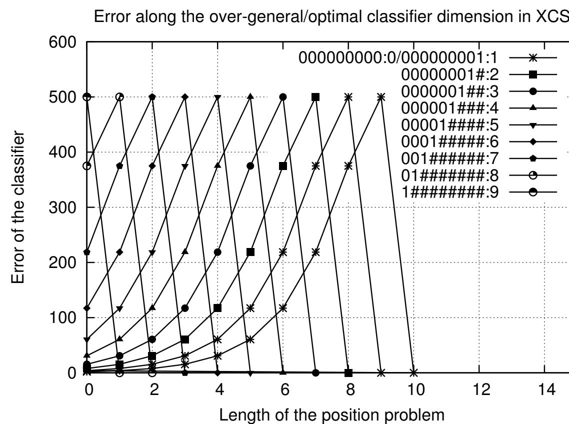


Figure 4.7: Error of XCS's classifiers along the over-general/optimal classifier dimension.

similar characteristics. Furthermore, we show that the fitness computation of UCSs overcomes this problem.

To exemplify the fitness misguidance in XCS in some particular problems, let us first focus on the decoder problem. The best action map of the decoder consists of maximally specific classifiers whose class is the decimal value of the binary input. Then, let us suppose that we have an over-general classifier cl_1 `1###:8`, whose theoretical prediction and prediction error estimates are P=125 and ϵ = 218.75. XCS is expected to drive cl_1 to the classifier `1000:8`. Imagine now that the genetic search generates the classifier cl_2 `10##:8`, whose prediction estimate is P=250 and whose prediction error is ϵ = 375. Note that the error increases as we are approaching the optimal classifier, that is, `1000:8`. Figure 4.6(a) extends this example and shows the evolution of the error from the over-general to the maximally general dimension

for the decoder problem with $\ell = 10$. Note that the error curve has an exponential increase as the classifier moves from the over-general to the maximum general side; the error abruptly decreases to zero when the optimal classifier is reached. Therefore, as long as the classifier moves toward the right direction, it gets smaller fitness, which consequently means fewer genetic opportunities and higher deletion probabilities. Thus, there is a misleading fitness pressure toward optimal classifiers.

This fitness misguidance is also present in the position problem, although the effect is not as important as in the decoder problem. To illustrate this, figure 4.7 shows an example of how the prediction error increases for each one of the optimal classifiers in the position problem with $\ell = 9$. Note that the misleading pressure is larger as the optimal classifier has more specific bits; that is, the Hamming distance from the maximally over-general classifier to the optimal classifier is larger as the optimal classifier is more specific, and so, the deception also increases.

This misleading pressure, which was termed as the *fitness dilemma* by Butz et al. (2003), does not only appear in these two problems, but in any problem whose optimal population contains specific classifiers. UCSs overcomes the fitness dilemma because the accuracy is calculated as the proportion of correct classifications instead of as a function of the accuracy of the prediction estimate. To exemplify this, figure 4.6(b) depicts the evolution of the error (that is, one minus the classifier's accuracy) along the over-general/maximally general dimension for the decoder problem with $\ell = 10$. Note that, in UCSs, the error diminishes as the classifiers approach the optimal one. In this way, UCSs accuracy guidance does not mislead the genetic search. This conclusion is also applicable to the original UCS, that is, UCSns.

Hence, the results provided in this section highlight that, as hypothesized in the previous chapter, UCS can evolve the best action map more quickly than XCS and spending fewer computational resources. Moreover, we also showed that the architecture of UCS does not suffer from the fitness dilemma. In the following section, we summarize all the observations provided in the present and the previous sections, identifying the key conclusions of the comparative analysis.

4.6 Lessons Learned from the Analysis

The empirical study performed in the last two sections provided many useful insights about the advantages of fitness sharing and the benefits of the UCS's architecture with respect to that of XCS in supervised learning problems. The purpose of this section is to gather and summarize all the observations. In particular, we first group the observations about the new fitness-sharing scheme of UCS. Then, we summarize the advantages provided by the explore regime and the accuracy guidance of UCS with respect to those of XCS. Each one of these aspects is elaborated in the following subsections.

4.6.1 Fitness Sharing

Fitness sharing speeded up UCS's convergence in all the tested problems. Especially, we argued that fitness sharing was the key to solve problems with class imbalances, where some optimal classifiers are activated with a lower frequency than other optimal and over-general classifiers. UCSns only yielded better performance in a single configuration of the 20-bit multiplexer with

alternating noise, which corresponded to a parameter setting that was not suited to solve the problem. In any case, this effect cannot be attributed to the presence or absence of fitness sharing, but rather to the way in which fitness is estimated. Recall that UCSns computes fitness as a power of accuracy, while UCSs computes fitness as a weighted windowed average with learning parameter β.

4.6.2 Explore Regime

In the comparison between XCS and UCS, the explore regime of UCS has shown to be a crucial aspect to speed up the convergence time of UCS with respect to the one presented by XCS in classification problems. That is, in general, exploring the best action map instead of the complete action map enables UCS to find the optimal solution more quickly than XCS, while spending fewer computational resources. On the other hand, we empirically illustrated that XCS may benefit from exploring the complete action map in problems, such as the parity problem, where the Hamming distance among the optimal classifiers that predict the wrong class and those that predict the correct class is small. Nevertheless, even in this case, UCS presented similar convergence times while using half of the maximum population size during the learning process, thus, saving computational resources.

Finally, it is worth noting that other exploring mechanisms could be adopted in XCS. That is, XCS uses a pure explore regime, where each available class is uniformly explored for each possible input. Nonetheless, as already pointed out by Wilson (1995) in the original design of the system, other exploration/exploitation schemes could be easily adapted to the system. For example, training in XCS could be based on an exploration regime that gradually changes from pure exploration toward increasing exploitation, similar to schemes such as ϵ-greedy or softmax that can be found in the reinforcement learning literature (Sutton and Barto, 1998). Changing the exploration regime may help XCS converge more quickly to the optimal solution; nonetheless, even with a new exploration/exploitation scheme, XCS would need to explore the different actions and evolve a complete action map, which prevent the system from being as competitive as UCS in solving hard classification tasks.

4.6.3 Accuracy Guidance

The results provided along the experiments showed that XCS may suffer from not only a lack of fitness guidance toward accurate classifiers, but also a deceptive guidance toward the optimal classifiers in some classification domains. This problem, already identified by Butz et al. (2003), was termed the fitness dilemma. Here, we showed that the problem appeared in almost all the tested problems, especially as the number of specific bits of the optimal classifiers increased. More specifically, we showed that XCS strongly suffered from the fitness dilemma in the decoder and, to a lower degree, in the position problems. As these problems gather some characteristics of real-world problems, this seems to indicate that XCS could suffer from the fitness dilemma in complex real-world classification problems. To alleviate the effect of the fitness dilemma in XCS, Butz et al. (2003) proposed a new error computation scheme that was addressed as bilateral accuracy. On the other hand, note that UCS could overcome the fitness dilemma since it computes the accuracy as the proportion of correct classifications.

4.6.4 Population Size

In the tested problems, UCS evolved best action maps with fewer learning iterations. Also, smaller population sizes were used in UCS in all the tested problems. The population evolved by XCS is generally larger, but comparable to that of UCS in terms of readability. In fact, by removing low-rewarded classifiers from XCS's final population, we get a set of rules similar to that of UCS. Thus, the advantage of having smaller populations in UCS is the reduction of computational resources.

4.7 Summary and Conclusions

In this chapter, we set the double objective of (1) revisiting the architecture of UCS by introducing a fitness-sharing scheme and (2) empirically comparing XCS with UCS. With the first objective, we aimed at improving UCS to deal effectively with new challenging problems. The purpose of the second objective was to illustrate empirically how the modifications introduced by UCS to solve classification problems more scalably than XCS actually affected the system's behavior.

We illustrated that the new fitness-sharing scheme enabled UCS to solve the four boundedly difficult problems more quickly. In essence, the fitness-sharing scheme helped UCS eliminate over-general classifiers as long as the first competing optimal classifiers were discovered; therefore, this speeded up the learning process. This was especially important in imbalanced domains such as the position problem, where highly specific, optimal classifiers had to compete with over-general classifiers. In this case, fitness sharing played a key role in decreasing the fitness of over-general classifiers, and so, in preventing these over-general classifiers from taking over a large proportion of the population, removing more specific, optimal classifiers. Thence, these observations promote the use of UCS with the new sharing scheme, instead of the original fitness computation, as a competitive tool for supervised learning, and, in particular, for learning from imbalanced domains. For this reason, we adopt this system in the remainder of our work.

The comparison with XCS allowed for a better understanding of the implications produced by the architectural changes introduced to UCS. In brief, we showed that UCS could solve the four classification problems spending fewer computational resources than XCS. Also, the analysis highlighted that UCS does not suffer from the fitness dilemma detected in XCS. Although the results and conclusions are limited to artificial problems, the experimental test bed contained many complexity factors present in real-world problems: multiple classes, noisy instances, and imbalanced classes, among others. Therefore, this encourages the use of UCS as a competent tool for supervised learning. It is worth noting that XCS is a broader architecture that can be applied to reinforcement learning, in general, and to some other tasks such as function approximation.

After the improvement of UCS and the empirical analysis provided herein, now we are in position to address the second and third objectives of this thesis, which study the performance of the two LCSs on domains that contain rare classes. Therefore, in the three subsequent chapters, we conduct a detailed analysis of the behavior of both LCSs on imbalanced domains and improve the ability of LCSs to extract accurate models from rare classes.

Chapter 5

Facetwise Analysis of XCS for Domains with Class Imbalances

The previous two chapters described XCS and UCS in detail, enhanced UCS with a new fitness-sharing scheme, and empirically compared the performance of these systems. This provided background information on how the two systems work and the key differences between them and led to the choice of using the fitness-sharing scheme instead of the original parameter update procedure in UCS. Although the experimentation empirically showed that both XCS and UCS can effectively solve boundedly difficult problems, there are still some challenges that need to be addressed to solve, scalably and efficiently, real-world problems.

A particular important challenge—shared by traditional machine learning techniques and GBML systems alike—is learning from domains that contain class imbalances. Learning from rare classes is a crucial aspect since the key knowledge usually resides in the minority class, and it has been shown that many traditional learning techniques are not able to extract accurate models from rare classes (Weiss, 2004). Therefore, the machine learning community has recently started to design new approaches that aim at improving the model discovery from rare classes (Chawla et al., 2004). Nonetheless, this aspect has been largely overlooked in online learning architectures. Imbalanced domains still pose more challenges to online learning systems, since the learner receives a stream of examples from which rare classes have to be modeled on the fly. The dearth of examples of rare classes may bias the parameter update procedure, forgetting the feedback provided by these examples, and so, hampering LCSs from evolving accurate classifiers that represent these rarities. In the following three chapters, we address this problem in the context of XCS and UCS, going from an analytic approach to the application of the system to solve real-world imbalanced problems.

This chapter studies the behavior of XCS on imbalanced domains and takes the lessons provided by the analysis to improve the modeling of rare classes. Although the analysis is centered on XCS in this chapter, we methodologically analyze the critical elements that any LCSs should satisfy to extract knowledge effectively from rare classes. Thence, we aim at providing a methodological framework rather than an analysis centered on a particular system. That is, we *decompose* the problem of learning from imbalanced domains in several *critical* elements and derive *facetwise models* (Goldberg, 2002) for each one of them. The integration of these models permits us to detect several crucial conditions that need to be met to ensure

that the system would extract the key knowledge from rare classes; in addition, we also identify *critical bounds* on the system behavior (Orriols-Puig and Bernadó-Mansilla, 2008b). The lessons learned from this analysis result in several configuration recommendations that, when followed, enable XCS to solve highly imbalanced classification problems that previously eluded solution. In the next section, we show that the whole framework can also be applied to UCS.

The remainder of this chapter is organized as follows. Section 5.1 presents the class-imbalance problem, reviews how the machine learning community has faced the problem with offline learning techniques, and places the problem in the context of online learning. Section 5.2 tests XCS on an imbalanced problem, intuitively discusses the complexities that learning from imbalanced domains may pose to the system, and empirically shows limits on the class-imbalance degree that the system can handle. Section 5.3 introduces the *facetwise analysis* methodology and specifies the steps that we follow in the present analysis. In sections 5.4, 5.5, 5.6, 5.7, and 5.8, we derive the different facetwise models. Section 5.9 integrates all the facetwise models, highlights the lessons learned along the analysis, and uses them to solve the 11-bit multiplexer problem (Wilson, 1995) with large degrees of class imbalance, which previously eluded solution. Finally, section 5.10 summarizes and concludes the chapter.

5.1 The Challenges of Learning from Imbalanced Domains in Machine Learning

During the last few decades, the increasing research on machine learning has led to the application of several learning techniques to real-world problems with the aim of extracting novel, interesting, and useful knowledge from these domains (Duda et al., 2000). One of the main characteristics of real-world problems is that some of the sub-concepts or classes may be poorly represented in the training data set due to either the scarcity of these concepts in nature or the cost—or inadequacy of the techniques used—to extract positive samples that represent these concepts. The purpose of this section is to highlight the importance of extracting accurate models from these rare concepts or classes in machine learning tasks. We begin emphasizing the high number of real-world applications in which we have class imbalances. Then, we briefly review how the machine learning community in general, and the GBML field in particular, has approached this problem, accentuating the differences between the challenges that learning from rare classes poses to offline learners and to online systems.

Examples of problems that contain rare classes abound in literature and include identifying fraudulent credit card transactions (Chan and Stolfo, 1998), learning word pronunciation (den Bosch et al., 1997), predicting pre-term births (Grzymala-Busse et al., 2000), detecting oil spills from satellite images (Kubat et al., 1998), and predicting telecommunication equipment failures (Weiss and Hirsh, 1998) among others. In these domains, while regularly occurring patterns can be modeled easily, learners tend to fail to extract accurate models from the rare classes (Japkowicz and Stephen, 2002; Japkowicz and Taeho, 2004; Weiss, 2004). Nonetheless, these rare classes are of primary interest, since they usually contain key knowledge. For this reason, strong research has recently been conducted on designing new approaches, or modifying existing machine learning techniques, with the aim of creating models that represent the rare classes more accurately (Weiss and Provost, 2003; Fawcett, 2008). All these approaches have been designed for offline supervised learning techniques, that is, methods that learn from static data

sets in which the examples are provided at the beginning of the learning process. In online learning, knowledge acquisition from imbalanced domains poses more severe challenges, since systems receive instances and rare classes have to be detected on the fly.

The many different approaches that have been designed to enhance the discovery of useful models of rare classes in offline learning can be grouped in methods working at (1) the learner level or at (2) the sampling level. Learner-level methods usually modify the error calculation of an existing system by either introducing a more appropriate inductive bias (Carvalho and Freitas, 2002) or assigning a misclassification cost per class (Pazzani et al., 1994). Their main drawback is that they are designed for specific learning algorithms, and so, they cannot be adapted to other learning techniques in a straightforward manner. For this reason, sampling-level methods have received much more attention than learner-level approaches. Sampling-level methods, usually known as *re-sampling techniques*, eliminate rare classes by balancing the proportion of examples per class of the training data set. As they are data-preprocessing methods, they can be generally used in any learning architecture. Notwithstanding, these methods can only be applied to static data sets. Many works have shown the benefits of re-sampling the training data sets in some specific imbalanced problems (Chawla et al., 2002; Japkowicz and Stephen, 2002; Batista et al., 2004; García and Herrera, 2008).

While the class-imbalance problem has been extensively analyzed for classical machine learning techniques that learn from static data sets, analyses and new approximations to overcome the problem in online learning are scarce. The typical approaches designed for offline learning to overcome the class-imbalance problem can barely be applied to online learning schemes, since they need to know the distribution of examples in the training data set. That is, in online learning, strategies such as over-sampling the occurrence of instances of the minority class or introducing a higher misclassification cost for minority class instances are impractical solutions. In this case, as we do not have a static data set, we can neither estimate the imbalance ratio to re-balance the training data nor assign a misclassification cost per class to favor the rare classes. Therefore, models for the minority class have to be learned online from a set of samples that come infrequently. This is the case of Michigan-style LCSs.

Although some ad hoc strategies have been proposed to alleviate this problem in particular LCSs (Holmes, 1998; Orriols-Puig and Bernadó-Mansilla, 2005a,b), these strategies cannot be directly extended to other LCSs. In general, these approaches have shown to improve LCSs performance on imbalanced domains. However, it is not clear how they affect the learning processes of LCSs; besides, they do not help increase our understanding of the actual problems that imbalanced domains pose to LCSs. Thence, in this chapter, we propose to start with a systematic analysis that explains the capabilities and limitations of the online learning architecture of LCSs to extract useful information from instances that come very infrequently. Specifically, we first apply the analysis to XCS and, in the next chapter, we carry it over to UCS. Before proceeding with the analysis, the next section provides an intuitive description of the problems that Michigan-style LCSs may face when learning from domains with rare classes. Then, the analysis methodology is introduced and all the study is developed in the subsequent sections.

5.2 The XCS Classifier System in Imbalanced Domains

As mentioned in the previous section, LCSs may have other problems, in addition to those presented by classical machine learning techniques in learning from imbalanced domains, since the model is learned online. With the system description provided in chapter 3 in mind, in this section we first appeal to the intuition to discuss the possible difficulties that XCS may find when learning from class imbalances, and we take advantage of the discussion to provide some notation that will be used in the remainder of this work. Then, before proceeding with a systematic study, we empirically analyze whether XCS is robust to class imbalances; to do this, we test the system on an artificial problem that is unbalanced by progressively removing instances of the minority class and check the maximum amount of under-sampling that the system can accept before failing to discover the concepts of the minority class.

5.2.1 Hypotheses of XCS Difficulties in Learning from Imbalanced Domains

Holland (1975) early defined the term of *schema* as a template that identified a subset of individuals, and used this notion to derive theory that explained how the good solutions are propagated along a GA run. Here, we take the same notion to define the concepts of *problem niche* and *representative* of a niche in LCSs, which will be used to study the effect of class imbalances in XCS. With these definitions, we review the components of the online learning architecture that may malfunction when learning from rare classes.

XCS evolves a distributed set of sub-solutions. In the remainder of this analysis, we use the term *problem niche* (or simply *niche*) to refer to a problem subspace where a maximally general sub-solution applies[1]. Each niche is represented by a *schema* (Holland, 1975), which defines the value of the relevant attributes of the given problem niche, and the class or action of region covered by the schema. The order o of the schema is the number of relevant attributes of the niche. Then, we define that a *representative* of a niche is a classifier whose condition specifies the o relevant attributes of the niche schema correctly and that predicts the class or action of the sub-solution that the niche represents. Conversely, classifiers that do not match any problem schema and cover examples of different classes are referred to as *over-general classifiers*.

For instance, let us suppose that we have a niche represented by the schema 01*0** and whose class is 0. This means that all the examples matching 01*0** belong to class 0. Generalizing any of the o specific bits of the schema will cause the schema to cover instances of other classes, thence, not representing an accurate sub-solution anymore. Any classifier whose condition has the same value for the o specific bits and predicts the niche class is a representative of the niche. For example, classifiers 01#0##:0, 0110##:0, and 010010:0 are representatives of the niche. The maximally general representative of a niche is the classifier that only fixes the o relevant bits of the schema and predicts the action of the sub-solution, i.e., 01#0##:0. An example of an over-general classifier is #1#0##:0, since it generalizes the first fixed bit of the schema.

Therefore, Michigan-style LCSs evolve niches in a distributed manner, and the population consists of classifiers that represent niches and over-general classifiers. Then, a niche—and all the matching classifiers—is activated every time that an instance that matches the niche schema is sampled and the class niche is selected for exploration. In imbalanced domains, instances that

[1] In XCS and UCS terms, an action set and a correct set, respectively, represent a niche.

belong to rare classes are sampled with a lower frequency; thence, niches that match these instances are activated with a lower frequency than the other niches of the system. In the remainder of this analysis, we address the niches that are activated by instances of the majority class as *nourished niches*. Conversely, niches activated by instances of the minority class are referred to as *starved niches*. We also define the *imbalance ratio ir* as the ratio of the number of examples of the majority class to the number of instances of the minority class that are sampled to the system.

Provided these definitions, we now intuitively analyze the possible difficulties that XCS may need to face in imbalanced domains by reviewing how the online architecture works. In the beginning of the learning process, the covering operator initializes the population with a set of classifiers that are generalized from the first sampled input instances. Therefore, the initial population consists mostly of over-general classifiers. Then, the system relies on (1) the parameter update procedure to obtain trusty estimates of classifier parameters online and (2) the evolutionary pressures (see section 3.1.6) to drive the population from a set of over-general classifiers to a set of maximally general and accurate classifiers that represent all niches. Both processes may be biased by the presence of rare classes.

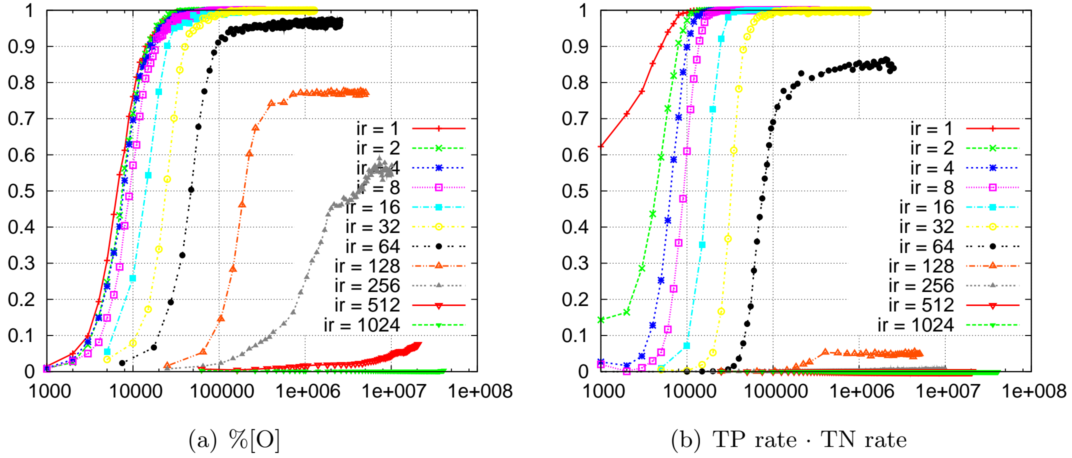
The first peril that appears is that the parameter update procedure provides poor estimates of the parameters of the classifiers that match examples of different classes, that is, over-general classifiers. In XCS, classifier's parameters are estimated by means of a weighted window average. Therefore, the system gives more importance to recently received rewards as opposed to older rewards, with the result that the reward provided by a particular example is forgotten after a certain number of learning iterations. Therefore, in imbalanced domains, it could be that examples of the minority class come so infrequently that the rewards provided by them are forgotten before receiving the next minority class instance, biasing the estimation of the parameters of over-general classifiers.

The evolutionary pressures may also be misled by the scarce sampling of minority class instances. Due to the occurrence-based reproduction of XCS, intuition seems to indicate that the system may suffer to discover representatives of starved niches, since these niches will be infrequently activated with respect to the other niches of the system. Hence, this low number of genetic opportunities combined with the other evolutionary pressures—which promote the most general classifiers—may discourage XCS from evolving representatives of starved niches.

In our study, we take all these hypotheses, systematically analyze the difficulties that XCS may face as the imbalance ratio increases, and provide solutions to let the system learn from highly imbalanced domains. To do this, we define several elements that need to be satisfied and derive models that relate these conditions with the imbalance ratio of the problem. Before proceeding with this analysis, in the next subsection we empirically show the behavior of XCS on an artificially imbalanced domain. The same experimentation is repeated after conducting the facetwise analysis and integrating all the models, emphasizing the importance of the lessons obtained from the analysis.

5.2.2 Empirical Observations of XCS Behavior on Class Imbalances

Here, we show the performance of XCS on the imbalanced multiplexer problem (Orriols-Puig and Bernadó-Mansilla, 2006a), a redefinition of the multiplexer problem where one of the classes is



(a) %[O]

(b) TP rate · TN rate

Figure 5.1: Evolution of (a) the proportion of the optimal population and (b) the product of TP rate and TN rate in the 11-bit multiplexer with imbalance ratios ranging from ir=1 to ir=1024.

progressively unbalanced with respect to the configuration parameter ir. That is, given a certain imbalance ratio ir, the sampling process of the multiplexer problem is modified such that the system receives ir instances of the majority class for each instance of the minority class. Thence, we empirically show the maximum imbalance ratio ir that XCS, using a standard configuration reported in the literature, could solve.

For this purpose, we ran XCS on the imbalanced multiplexer problem with imbalance ratios of $ir = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$. We set XCS with standard values used in the literature for its configuration parameters, that is:

$$\alpha = 0.1,\ \epsilon_0 = 1,\ \nu = 10,\ \chi = 0.8,\ \mu = 0.04,\ \theta_{del} = 20,\ \delta = 0.1,\ \theta_{sub} = ir,\ P_{\#} = 0.6.$$

We used tournament selection, two point crossover with $\chi = 0.8$, and bitwise mutation with $\mu = 0.04$. We applied GA subsumption, setting $\theta_{sub} = ir$ to avoid having poorly evaluated over-general classifiers considered as accurate subsumers. We ran XCS during $40\,000 \cdot ir$ iterations; thus, given a problem, we ensured that the system received the same number of genetic opportunities for all imbalance ratios. Finally, to prevent having young over-general classifiers with poorly estimated parameters in the final population, we introduced $5\,000 \cdot ir$ iterations with the GA switched off at the end of the learning process.

Figure 5.11(a) illustrates the evolution of the proportion of the optimal population %[O] achieved by XCS. That is, XCS was expected to evolve 32 optimal classifiers, each one representing a different niche. In this way, we measured the capacity of XCS to generalize and obtain the best representative of each niche at high imbalance ratios. Figure 5.1(b) depicts the evolution of the product of TN rate—i.e., the proportion of correct classifications of the over-sampled class—and TP rate—i.e., the proportion of correct classifications of the under-sampled class. Note that XCS can evolve all the optimal population and yield 100% of the product of TP rate and TN rate only for $ir \leq 32$. As the imbalance ratio increases, XCS is able to discover a lower proportion of the optimal population; particularly, the classifiers that represent starved niches

are not created and maintained by the system. For $ir \geq 64$, the system cannot discover the knowledge that resides in the minority class.

Therefore, these preliminary experiments show that XCS is robust at moderate imbalance ratios, but that it fails to provide accurate representatives of the minority class for high imbalance ratios. In the next section, we explicate the facetwise methodology used to analyze the possible causes of this failure. We enumerate the different elements that need to be guaranteed to learn accurate models from rare classes; then, models for each one of the elements are elaborated in the subsequent sections.

5.3 Facetwise Analysis of XCS in Imbalanced Domains

In this section, we follow a design decomposition approach to systematically analyze the different sources of difficulty that XCS may find when learning from imbalanced domains. We first briefly introduce the design decomposition methodology adopted by Goldberg (2002) as a design approach to competent genetic algorithms and review how it has been transported to XCS. Then, we follow this approach to decompose the problem of learning from imbalanced domains in XCS.

5.3.1 Design Decomposition in GAs

Goldberg (2002) emphasizes the relevance of *design decomposition* and *facetwise analysis* for advancing in the design and the understanding of complex systems. The design decomposition methodology separates the working process of complex systems into different elements, and each one of these elements is analyzed separately assuming that all the others are behaving in an ideal manner. All these models provide key insights into the working of the complex system, increasing our understanding of the underlying processes of the system. Furthermore, they also can be used as a tool for designing new competent and efficient complex systems that satisfy the requirements identified by the different models. Besides, the individual facetwise models can be combined, identifying the sweet spot where the algorithm actually scales. For further details about the application of this methodology to the design of competent GAs, the reader is referred to section 2.2.4.

As described in the previous section, similarly to GAs, LCSs are complex systems in which several components interact to evolve a set of maximally general and accurate classifiers. Due to this complexity, efforts have recently been made to apply the design decomposition and the facetwise analysis methodology to LCSs. In the next section, the existing work on carrying the design decomposition approach to LCSs domain is briefly reviewed.

5.3.2 Carrying the Design Decomposition from GAs to XCS

The application of facetwise modeling to LCSs, and specifically XCS, has provided key insights into XCS working processes, enabling the solution of more complex problems. As follows, we review these analysis in more detail.

One of the first works toward the definition of a theory based on facetwise analysis for XCS can be found in (Butz et al., 2004b). In this work, the authors studied the learning pressures in

XCS and derived critical bounds on the system convergence. In particular, the authors derived two boundaries beyond which the convergence of XCS could not be guaranteed, which were addressed as the *covering challenge* and the *schema challenge*. Continuing the analysis of the different pressures of XCS, Butz et al. (2003) analyzed the fitness pressure and derived the so-called *reproductive opportunity bound*, which sets the population size required to warrant that a classifier, with a certain specificity, will have reproductive opportunities. Later, Butz et al. (2004a) complemented the previous study by deriving models of the learning time in XCS. The models were simplified by not considering crossover and by assuming a domino-convergence model (Thierens et al., 1998). More recently, Butz et al. (2007) presented a Markov chain analysis of niche support in XCS. The analysis showed that the number of classifiers of a niche followed a binomial distribution, which yielded another population size bound to ensure effective problem sustenance.

However, these facetwise models do not explain all the aspects of XCS. Whereas these models have provided considerable insights and increased our understanding of XCS, they do not fully capture the effect of dealing with problems that contain class imbalances. Hence, in this thesis, we follow a design decomposition methodology to study whether XCS can efficiently deal with rare classes. As follows, we first present an artificial problem that will enable us to identify the different difficulties that we may face when learning from imbalanced domains. Later, we present the problem decomposition.

5.3.3 A Boundedly Difficult Problem for LCSs: The Imbalanced Parity Problem

The first step before proceeding to the facetwise analysis is to design a proper test problem that highlights the difficulties that are to be studied and that serves to validate the developed models. Following this analogy, we designed the imbalanced parity problem, whose description is provided as follows.

The imbalanced parity problem extends the parity problem (Kovacs and Kerber, 2001) by introducing a parameter that permits controlling the complexity along the imbalance dimension. The problem is defined as follows. Given a binary string of length ℓ, where there are k relevant bits ($0 < k \leq \ell$), the output is the number of one-valued bits in the k relevant bits modulo two. This corresponds to the original definition of the parity problem. We introduce the imbalance complexity to this definition by starving the class labeled as '1'. That is, ir denotes the ratio of examples of the majority class to the number of instances of the minority class. For $ir = 1$, the problem has, approximately, the same number of instances per class. For $ir > 1$, there are ir instances of the majority class for each instance of the minority class. Independent of the imbalance ratio, the optimal population for this problem consists of 2^{k+1} classifiers that have specific values for the k relevant attributes and all the remaining attributes are set to '#' (see appendix A.1).

Note that the complexity of the problem can be moved along two dimensions: the building block size k and the imbalance ratio ir. Larger values of k pose more challenges to XCS, since the system needs to discover larger, more complex building blocks whose bits have to be processed together. Moreover, k also defines the number of irrelevant attributes which XCS must generalize to obtain the optimal population. On the other hand, increasing ir implies a progressive under-sampling of instances of the minority class, which may hinder XCS in discovering optimal

classifiers for this class.

Now that we have defined the parity problem and analyzed its possible sources of complexity, the next section introduces the particular decomposition proposed for Michigan-style LCSs in general.

5.3.4 Decomposition of the Class Imbalance Problem in XCS

With the intuitive difficulties of XCS in learning from rare classes discussed in section 5.2.1 and adhering to the design decomposition methodology proposed by Goldberg, we are now in position to articulate the different elements that need to be satisfied to successfully deal with problems that contain class imbalances. Our major concern is to guarantee that representatives of starved niches will win in competition with over-general classifiers even for high imbalance ratios. Thus, we decompose the problem and consider all the facets that are likely to be affected by the dearth of examples of the minority class. Then, we derive facetwise models that model each one of the subproblems. More specifically, we consider the following five subproblems:

1. Estimate the classifier parameters correctly—prediction, error, and fitness of classifiers.
2. Analyze whether representatives of starved niches can be provided in initialization.
3. Ensure the generation and growth of representatives of starved niches.
4. Adjust the GA application rate.
5. Ensure that representatives of starved niches will take over their niches.

Estimate the classifier parameters correctly. The primary factor that needs to be satisfied is that the evaluation procedure obtains accurate estimates of the parameters of all classifiers, and especially of over-general classifiers. In XCS, classifier parameters are updated online according to the reward received at each time step by means of the Widrow-Hoff rule (Widrow and Hoff, 1988). This method makes a temporal windowed average of the received rewards, which gives more importance to the last received rewards as opposed to the older rewards. Consequently, the dearth of sampling of examples that represent one of the classes may cause poor estimations in over-general classifiers, since infrequent negative rewards can be forgotten. This aspect is really important since it may completely mislead the genetic search. That is, if the error of over-general classifiers is underestimated, XCS may promote these over-general classifiers when they are competing with accurate classifiers in the same niche. We investigate the accuracy of the parameter update procedure and provide some alternative parameter evaluation methods in section 5.4.

Analyze whether representatives of starved niches can be provided in initialization. Once ensuring that classifier parameters can be properly evaluated, we have to analyze whether XCS initialization process can supply the initial population with classifiers that contain schemas of starved niches in the beginning of the run (Orriols-Puig et al., 2007c). That is, XCS starts with an empty population. Then, the covering operator is applied in the first iterations of the learning process, providing classifiers whose conditions are generalized from the first instances that are sampled to the system. Covering should initialize the population with several classifiers

that, although not being maximally accurate, contain schemas that represent niches of different classes. Nonetheless, intuition seems to indicate that, for highly imbalanced domains, covering is mainly triggered on instances of the majority class; therefore, covering may fail to provide schemas of niches that represent the minority class. Furthermore, this problem is even more severe as the schema of starved niches gets larger (in the parity problem, this translates to having larger values of k). In section 5.5, we derive models that formally explain this behavior. The remainder of the analysis is derived assuming a covering failure.

Ensure the generation and growth of representatives of starved niches. After the population is initialized, the genetic algorithm drives the search toward more accurate and general classifiers. We already appealed to the intuition that the occurrence-based reproduction of XCS may hamper the discovery of maximally general and accurate classifiers for starved niches. In section 5.6, we derive facetwise models that systematically analyze the effect of class imbalances in the (1) generation and (2) growth of representatives of starved niches. Consequently, we derive bounds on the population size to guarantee that XCS will be able to learn classifiers that belong to starved niches. All the analysis is performed assuming that the GA is applied at each learning iteration.

Adjust the GA application rate. Having ensured the discovery of representatives of starved niches, section 5.7 introduces the frequency of application of the GA into the analysis and theoretically shows that decreasing the frequency of application of the GA results in a counter-balancing effect that may help XCS discover representatives of starved niches.

Ensure that representatives of starved niches will take over their niches. Discovering the first representatives of starved niches is not enough. That is, once discovered, the accurate representatives of starved niches should take over their niches, removing competing over-general, less accurate classifiers. Nonetheless, the effect of the occurrence-based reproduction may produce the opposite effect, resulting in situations where over-general classifiers take over starved niches. In section 5.8, we study the takeover time of the best representative in starved niches, following the methodology used in the genetic algorithms literature for these types of analyses (Goldberg and Deb, 2003). Takeover time expressions are derived for the two most-used selection techniques in XCS: proportionate selection (Wilson, 1995) and tournament selection (Butz et al., 2005c). Moreover, conditions for starved niches extinction, i.e., deletion of the best representatives of starved niches in favor of over-general classifiers, are also derived for both selection schemes.

In the remainder of this chapter, each of these facets is covered in a different section in which the technical argument will be developed more completely. Moreover, all the models are experimentally validated with the *imbalanced parity problem*. Then, in section 5.9, we unify all the models, emphasizing the lessons derived from each one. Specifically, the patchquilt integration of the models results in a domain of applicability of XCS for imbalanced domains, which (1) determines under which conditions and imbalance ratios the system will be able to successfully represent the minority class in the evolved model and (2) provides guidelines of how to set the system for different imbalance ratios. The insights supplied by these models are crucial to increase our understanding of how XCS evolves classifiers that represent starved niches, enabling the solution of highly imbalanced problems that previously eluded solution. We

show, as example, that all the acquired knowledge enables us to appropriately set XCS so that it can solve the multiplexer problem (Wilson, 1995) with large amounts of class imbalances.

5.4 Estimation of Classifier Parameters

In this section, we analyze whether the Widrow-Hoff rule provides accurate estimates of the parameters of over-general classifiers as the imbalance ratio increases. We especially focus on the *prediction error* of over-general classifiers, since it determines the classifier's fitness. If the prediction error is underestimated, over-general classifiers may be considered as accurate classifiers by the system; hence, they will win in competition with more specific but accurate classifiers. Thence, as follows, we first theoretically relate the error of over-general classifiers with the imbalance ratio, deriving a bound beyond which the system will not be able to distinguish between over-general classifiers and accurate representatives. Then, we empirically analyze whether the parameter update procedure of XCS can provide estimates that accurately predict the theoretical bound.

5.4.1 Imbalance Bound

We start the derivation of the maximum imbalance bound by considering that, according to Butz et al. (2003), the prediction p of a classifier can be approximated by

$$p = P_c(cl) \cdot R_{max} + (1 - P_c(cl)) \cdot R_{min}, \tag{5.1}$$

where $P_c(cl)$ is the probability that a classifier predicts the matching input correctly, R_{max} is the maximum reward, and R_{min} the minimum reward given by the environment. Then, the error of a classifier can be approximated as

$$\epsilon = |p - R_{max}| \cdot P_c(cl) + |p - R_{min}| \cdot (1 - Pc(cl)). \tag{5.2}$$

For classification problems, R_{min} is usually 0, so that the prediction of a classifier can be estimated by $p = P_c(cl) \cdot R_{max}$. Substituting p into formula 5.2, we get the following prediction error estimate:

$$\epsilon = 2R_{max} \cdot (P_c(cl) - P_c(cl)^2). \tag{5.3}$$

Now, let us relate $P_c(cl)$ with ir (Orriols-Puig and Bernadó-Mansilla, 2006a, 2008a). In average, over-general classifiers will match ir examples of the majority class for each example of the minority class. Assuming that p is correctly estimated, a classifier would correctly predict the output for the ir instances of the majority class, and would give an erroneous prediction for the example of the minority class. Thus, $P_c(cl)$ can be approximated as

$$P_c(cl) = \frac{ir}{1 + ir}, \tag{5.4}$$

and its error estimate as

$$\epsilon = 2 \cdot R_{max} \frac{ir}{(1 + ir)^2}. \tag{5.5}$$

An over-general classifier will be considered inaccurate as long as

$$\epsilon \geq \epsilon_0. \tag{5.6}$$

Using equation 5.5, we obtain that

$$2 \cdot R_{max} \frac{ir}{(1+ir)^2} \geq \epsilon_0, \tag{5.7}$$

which can be written as

$$-ir^2\epsilon_0 + 2ir(R_{max} - \epsilon_0) - \epsilon_0 \geq 0. \tag{5.8}$$

This represents a parabola where ϵ takes values higher than ϵ_0 for ir ranging between ir_ℓ and ir_u, where $ir_\ell < ir_u$. We are concerned about the maximum imbalance ratio up to which XCS would consider over-general classifiers as inaccurate; that is, ir_u. Solving equation 5.8, and assuming that $\epsilon_0 << R_{max}$, we obtain the following expression:

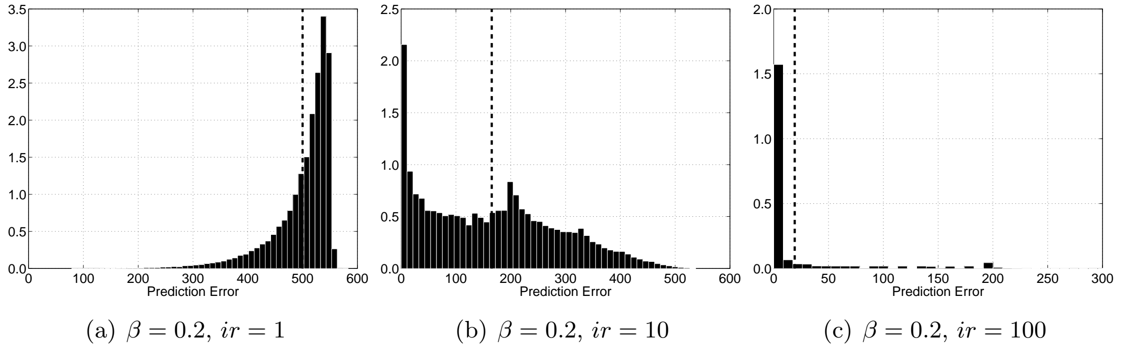
$$ir_u \approx \frac{2R_{max}}{\epsilon_0}. \tag{5.9}$$

That is, the maximum imbalance ratio up to which XCS will be able to detect over-general classifiers grows linearly with R_{max} and decreases linearly with ϵ_0. Substituting $\epsilon_0 = 1$ and $R_{max} = 1000$, the maximum imbalance ratio is: $ir_u \approx 2000$. Nonetheless, the experiments provided in section 5.2.2 illustrated that XCS failed to extract all the knowledge that resides in the minority class for $ir > 32$. As proceeds, we analyze whether this deviation between theory and experiments can be caused by a deviation of the real value of the error with respect to its theoretical estimate.

5.4.2 Does the Widrow-Hoff Rule Provide Accurate Estimates?

Here, we empirically analyze if XCS can obtain reliable estimates as ir increases. For this purpose, we ran the imbalanced parity problem with $\ell = 11, k = 4$, and $ir = \{1, 10, 100\}$. We initialized XCS with the optimal population plus the most over-general classifier predicting class 0 (i.e., `###########:0`) and deactivated the GA. We set $\beta = 0.2$, which is a typical value used in the literature. Figure 5.2 shows a histogram of the error estimate of the most over-general classifier along a complete run. Results are averages over 10 runs. The vertical line shows the theoretical value for each case.

Theoretically, the error of the most over-general classifier should be $\epsilon = \{500, 165.28, 19.60\}$ for imbalance ratios $ir = \{1, 10, 100\}$ respectively. For a completely balanced domain (see figure 5.2(a)), the error oscillates around the theoretical value, i.e., $\epsilon = 500$. For $ir = 10$, the error of the most over-general classifier is around zero with high frequency. For $ir = 100$, the classifier error is zero most of the time. That is, as the classifier receives instances of the majority class, its error keeps on decreasing until becoming approximately zero. At some point, an instance of the minority class is sampled, which causes an increase in the error of the over-general classifier. For $ir = 100$, as 100 instances of the majority class are sampled for each instance of the minority



(a) $\beta = 0.2$, $ir = 1$ (b) $\beta = 0.2$, $ir = 10$ (c) $\beta = 0.2$, $ir = 100$

Figure 5.2: Histogram of the error of the most over-general classifier with Widrow-Hoff delta rule at $\beta = 0.2$ and different imbalance ratios.

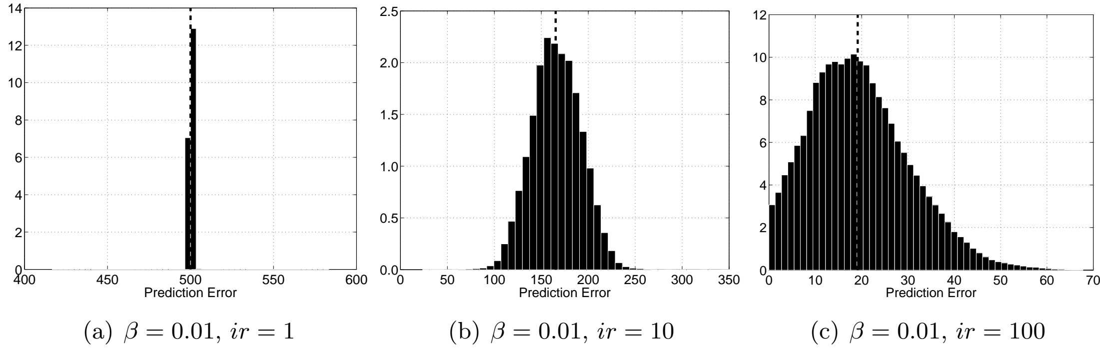
class, the error of the most over-general classifier is underestimated during most of the time. Note that when $\epsilon < \epsilon_0$, XCS considers that the classifier is accurate (a typical value for $\epsilon_0 = 1$). This is the case during large part of the training time for $ir = 10$ and, especially, for $ir = 100$. Besides, as the classifier is the most over-general possible, it will be favored to the detriment of highly accurate but more specific classifiers.

The problem of having non-stable estimates for over-general classifiers does not appear exclusively in highly imbalanced domains. This topic has been studied for multi-step problems with large delayed rewards, and several approaches have been designed to propagate the error effectively along the previous action sets when several steps have to be taken before reaching a reward. Herein, we consider two methods to obtain better parameter estimates. First, we show that the Widrow-Hoff rule can obtain better parameter estimates if β is properly tuned. Then, we adapt one of the most relevant methodologies for parameter evaluation designed for multi-step problems to single step tasks: *gradient descent* (Butz et al., 2005a). The next two sections show that the two methods allow for better estimates in highly imbalanced domains.

5.4.3 Obtaining Better Estimates with the Widrow-Hoff Rule

In the Widrow-Hoff rule, the parameter β determines the proportion of update in the classifier parameters. As the Widrow-Hoff rule works as a temporal windowed average, β also fixes the capacity to forget past rewards. That is, high values of β produce large modifications of classifier parameters every time a new reward is received, forgetting perviously received rewards quickly. Usually, this allows for a faster convergence of the classifier parameters to their real values. However, we have already seen the harmful effect in imbalanced domains.

Here, we show that a simple solution to prevent the oscillation of the parameters of over-general classifiers is to decrease β, considering in this way a longer history of rewards. Lower values of β would cause smaller corrections, and so, less oscillations. Nonetheless, they would also imply slower convergence. For very small values of β, accurate offspring classifiers may lose against over-general parents at the beginning of the run, since their fitness increases slowly. This may impair XCS's ability to discover new accurate and more specific rules. Thence, β should



(a) $\beta = 0.01$, $ir = 1$ (b) $\beta = 0.01$, $ir = 10$ (c) $\beta = 0.01$, $ir = 100$

Figure 5.3: Histogram of the error of the most over-general classifier with Widrow-Hoff delta rule at $\beta = 0.01$ and different imbalance ratios.

be the highest value that prevents over-general classifiers from having zero error.

Figure 5.3 shows a histogram of the error estimate of the most over-general classifier along a complete rule for $\beta = 0.01$. For all the cases, the histograms are centered on the theoretical value of the error. Thus, over-general classifiers have precise estimates of their parameters most of the time. The standard deviation of the histograms increases with the imbalance ratio, since the rewards generated by the minority class examples are less frequent.

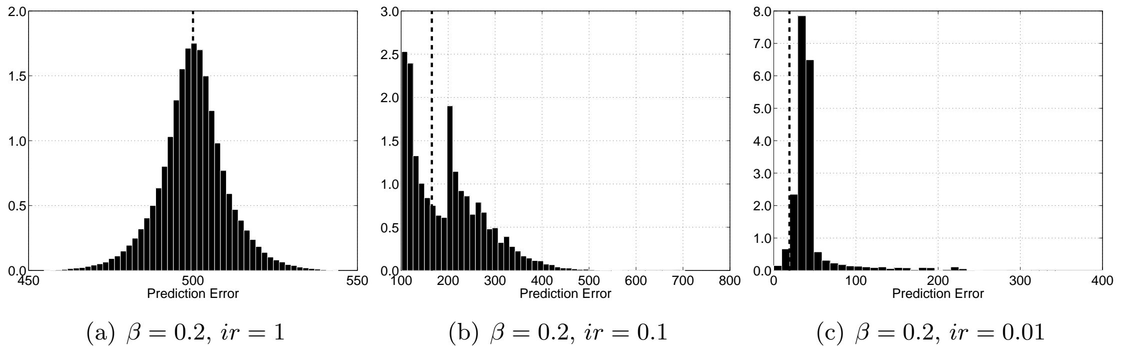
5.4.4 Obtaining Better Estimates with Gradient Descent Methods

Here, we study another approach, originally adapted from a gradient descent methodology, to obtain better parameter estimations. Butz et al. (2005a) introduced a gradient descent method to improve the parameter estimation of XCS in multi-step problems that involve a large number of state-action pairs. The new approach relies on identifying that the gradient term for a classifier is $\frac{F}{\sum_{[A]} F_i}$, in which F_i is the fitness of classifier i of the action set. Thus, classifier prediction is updated as

$$p = p + \beta(R - p)\frac{F}{\sum_{[A]} F_i}, \tag{5.10}$$

where R is the received reward. The rest of the parameters are updated as usual (see section 5.2). Note that this procedure aims at stabilizing classifier prediction. In consequence, the classifier error will also be stabilized since it tracks the prediction error.

The results provided by Butz et al. (2005a) illustrate that the gradient descent technique enables XCS to solve multi-step problems that eluded solution in XCS with Widrow-Hoff rule. Such improvement is explained by noting that the gradient term in the prediction update is actually an adaptive learning rate that prevents parameters from large corrections when the fitness of the updated classifier is much lower with respect to the fitness of the other classifiers in the same action set. Thereupon, gradient descent uses a heuristic procedure to automatically tune β depending on the fitness of each classifier. Then, the difference between gradient descent and decreasing β is that gradient descent automatically uses the aforementioned heuristic to



(a) $\beta = 0.2$, $ir = 1$ (b) $\beta = 0.2$, $ir = 0.1$ (c) $\beta = 0.2$, $ir = 0.01$

Figure 5.4: Histogram of the error of the most over-general classifier with gradient descent at $\beta = 0.2$ and different imbalance ratios.

determine the most suitable value for β instead of requiring the user to tune this parameter.

We repeated the same experiments with the imbalanced parity problem but now used gradient descent with $\beta = 0.2$. Figure 5.4 plots the histograms of the error estimate of the most over-general classifier along a complete run for gradient descent. XCS with gradient descent maintains fairly accurate estimates of the error of the most over-general classifier for all the imbalance ratios, even though a high value of β is used. However, note that these estimates are not as accurate as the ones obtained with Widrow-Hoff rule with a proper configuration of β.

In summary, this section showed that the Widrow-Hoff rule may provide poor estimates of the error of over-general classifiers for high class imbalances. This effect is undesirable since this may cause XCS to consider over-general classifiers as accurate. Two approaches, i.e., decreasing β for Widrow-Hoff rule and gradient descent provided more reliable parameter estimates. Although these methods would result in a slower convergence of XCS, their use is critical to guarantee that XCS will be able to obtain reliable estimates and converge to an optimal population. In the remainder of the analysis, we consider that classifier parameters are accurately estimated by the procedures presented above, and thus, that the genetic search is not misled. With this assumption, the next sections study the generation and growth of representatives in starved niches.

5.5 Supply of Schemas of Starved Niches in Population Initialization

Here, we study whether the covering operator is able to supply classifiers that represent schemas of starved niches for high imbalance ratios. As explained in section 5.2, covering is activated in the first stages of the learning process, creating new classifiers from the first sampled instances. To provide representatives of starved niches, the covering operator has to be triggered on minority class instances. However, as ir increases, fewer minority class instances are sampled. Thus, covering will be mainly activated from majority class instances. Then, most of the classifiers will be generalized from majority class instances, and so, classifiers representing schemas of the

minority class will be scarce. To analyze this effect, here we derive the probability that covering is triggered on the first minority class examples sampled to the system.

For this purpose, we first consider the probability that one instance is covered by, at least, one classifier $P(\text{cover})$. According to Butz et al. (2004b), this probability is

$$P(\text{cover}) = 1 - \left[1 - \left(\frac{2-\sigma[P]}{2}\right)^{\ell}\right]^{N}, \tag{5.11}$$

where ℓ is the input length, N is the population size, and $\sigma[P]$ is the specificity of the population. During the first learning stage of XCS, we can approximate $\sigma[P] \approx 1 - P_{\#}$.

Now, let us relate this probability to the imbalance ratio ir. We consider the worst case where (1) XCS receives ir instances of the other classes before receiving the first instance of the minority class and (2) the covering operator is triggered for each instance supplying n classifiers per instance (where n is the number of classes; thus $\theta_{mna} = n$). In this case, the probability that the population contains, at least, a matching classifier for each class is

$$P(\text{cover}) = 1 - \left[1 - \frac{1}{n}\left(\frac{2-\sigma[P]}{2}\right)^{\ell}\right]^{n \cdot ir}. \tag{5.12}$$

In this equation, we assumed that $N > n \cdot ir$, i.e., that XCS will not delete any classifier during the first ir iterations. This assumption is usually satisfied since covering is only applied for the first input examples, when there is room in the population. It is worth noting that, given a fixed ℓ and $\sigma[P]$, the term in brackets in the right hand of the equation decreases exponentially as the imbalance ratio increases. Thus, the probability of having classifiers in the population that match the first minority class instances tends to one exponentially with the imbalance ratio. Notice that the matching classifiers would have been generated from majority class instances, and so, would not represent schemas of starved niches.

With equation 5.12, we can derive the probability of activating covering having sampled a minority class instance. Provided that the probability of activating covering is $1 - P(cover)$, and recognizing that $(1 - r/n)^n \approx e^{-r}$, we obtain that

$$P(\text{activate cov. on. min.}) = 1 - P(\text{cover}) \approx e^{-ir \cdot e^{-\frac{\ell\sigma[P]}{2}}}, \tag{5.13}$$

which decreases exponentially with the imbalance ratio and, in a higher degree, with the condition length and the initial specificity. Figure 5.5 depicts the equation for $\ell = 20$, $n = 2$, and different initial specificities, showing that the probability of activating covering on the first sampled instance of the minority class decreases exponentially with the imbalance ratio.

Thence, this analysis manifests that the covering operator fails to supply classifiers representing correct schemas of the minority class for moderate and high imbalance ratios. In the next section, we assume a covering failure in providing schemas of starved niches and study whether the genetic search can discover representatives of starved niches.

5.6 Generation of Classifiers in Starved Niches

Assuming a covering failure to provide classifiers that represent schemas of starved niches, we now study how the GA can evolve representative classifiers for these starved niches. As follows,

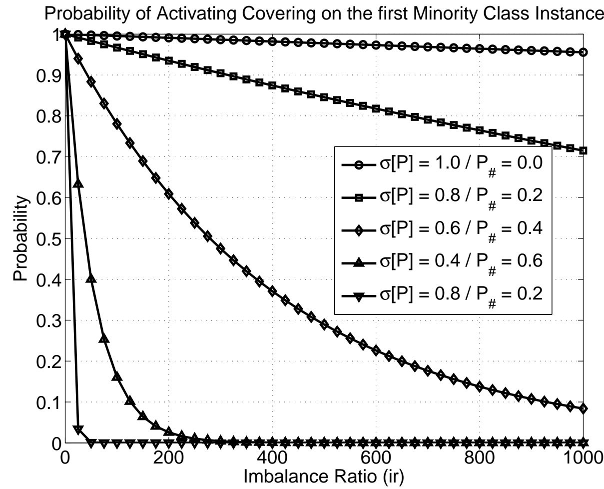


Figure 5.5: Probability of activating covering on a minority class instance given a certain specificity $\sigma[P]$ and the imbalance ratio ir. The curves have been drawn from equation 5.13 with $\ell = 20$ and different specificities.

we first enumerate the assumptions of the models and then analyze the probabilities of creating and maintaining representatives of starved niches. Finally, we use the different models to derive a population size bound to ensure the discovery of starved niches.

5.6.1 Assumptions for the Model

Before proceeding with the theoretical derivations, we first enumerate the assumptions made to develop the models. The analysis is focused on the evolution of starved niches. We assume a simplified scenario model where: (1) we do not consider crossover and contemplate mutation as the main operator for discovery, assuming low probabilities of mutation μ ($\mu < 0.5$) as usual in practice; (2) we assume that the GA is applied at each learning iteration (i.e., $\theta_{GA} = 0$); and (3) we consider random deletion. Subsequently, we relax all these constraints and experimentally analyze the impact of breaking each one of the assumptions. We experimentally examine the effect of introducing two point crossover. Furthermore, we investigate the biases caused by the enhanced deletion technique used currently in XCS (see section 3.1.4). In the next section, we study the effect of θ_{GA} theoretically and empirically.

5.6.2 Genetic Creation of Representatives of Starved Niches

In the first step of the analysis, we derive the time until the creation of the first representatives of starved niches, assuming that covering has not provided any of them. To achieve this, we first derive the probability to obtain the first accurate representative cl_{min} of the starved niche i, which is represented by a schema with order k_m. Thence, we study the probabilities of creating cl_{min} when sampling (1) instances of the minority class and (2) instances of the majority class. Recognizing that the probability of sampling a minority class instance is $1/(1 + ir)$ and the

probability of sampling a majority class instance is $ir/(1+ir)$, we can write that

$$P(cl_{min}) = \frac{1}{1+ir} P(cl_{min}|\text{min. inst}) + \frac{ir}{1+ir} P(cl_{min}|\text{maj. inst}). \tag{5.14}$$

Let us first derive $P(cl_{min}|\text{min. inst})$, that is, the probability of generating a representative of a starved niche when sampling a minority class instance. As we assumed that there are no representatives of starved niches in the population, the match set will only consist of over-general classifiers. Then, the system will choose a class randomly and will explore it, running a genetic event on the selected action set. Regardless of the selected class, and considering that there are only over-general classifiers in [M], a representative of a starved niche can be created if all the k_m bits are correctly set to the values of the niche schema. Here, we consider the worst case and assume that all the k_m bits need to be mutated. Thence, the probability of getting the correct schema is $(\frac{\mu}{2})^{k_m}$. Besides, the class of the rule needs to be set to the class of the niche. If the system selected to explore the minority class (which will be selected with probability $1/n$, where n is the number of classes), we have to ensure that the mutation operator would not change this class (that is, with probability $(1-\mu)$). Otherwise, we have to require that the mutation operator change this class to the niche class (with probability $\mu/(n-1)$). Therefore,

$$P(cl_{min}|\text{min. inst}) = \frac{1}{n}\left(\frac{\mu}{2}\right)^{k_m} \cdot (1-\mu) + \frac{n-1}{n}\left(\frac{\mu}{2}\right)^{k_m} \cdot \frac{\mu}{n-1} = \frac{1}{n}\left(\frac{\mu}{2}\right)^{k_m}. \tag{5.15}$$

The same procedure can be followed to derive the probability of creating cl_{min} when sampling an instance of the majority class, i.e., $P(cl_{min}|\text{maj. inst})$. In this case, the match set will consist of both over-general classifiers and representatives of nourished niches. Again, we consider the worst case and assume that, to create a representative of a starved niche, all the k_m bits of the niche schema need to be mutated. Moreover, if the system chooses to explore the minority class, the class of the classifier must be preserved; otherwise, the class has to be changed to the minority class. This results in exactly the same probability as before, i.e.,

$$P(cl_{min}|\text{maj. inst}) = \frac{1}{n}\left(\frac{\mu}{2}\right)^{k_m}. \tag{5.16}$$

Substituting equations 5.15 and 5.16 into equation 5.14 we obtain that

$$P(cl_{min}) = \frac{1}{n}\left(\frac{\mu}{2}\right)^{k_m}. \tag{5.17}$$

Then, we can derive the time required to discover the first representatives of starved niches $t_{cl_{min}}$ as

$$t_{cl_{min}} = \frac{1}{P_{cl_{min}}} = n\left(\frac{2}{\mu}\right)^{k_m}, \tag{5.18}$$

which depends linearly on the number of classes and exponentially on the order of the schema, but does not depend on the imbalance ratio.

Thus, even though covering fails to provide classifiers representing schemas of the minority class, XCS will be able to generate the first correct classifiers of the minority class independent of the imbalance ratio. In the following, we derive the time until the deletion of these classifiers. With both the generation and deletion time, we calculate the minimum population size to maintain these classifiers and ensure that the best representatives of starved niches will receive, at least, one genetic event.

5.6.3 Deletion of Representatives of Starved Niches

We now provide an approximate time to the extinction of representatives of starved niches. The time to extinction of classifiers mainly depends on the applied deletion procedure. The current deletion scheme of XCS (Kovacs, 1999) gives the classifiers a deletion probability proportional to their action set estimate as. This approach permits balancing the allocation of rules in the different niches in problems for which the frequency of the different niches is similar, i.e., balanced problems. Nonetheless, in highly imbalanced problems, the action set size estimate of accurate classifiers of starved niches may be negatively biased by over-general classifiers. That is, as over-general classifiers participate in the same action sets as accurate classifiers of starved niches, the action set size of these accurate classifiers may be overestimated.

As our model is developed for highly imbalanced domains, we consider the worst case, i.e., that the deletion procedure gives the same probability to each classifier to be deleted. Since two classifiers are deleted at each GA application, we obtain that the deletion probability is $P(\text{delete cl}) = 2/N$, where N is the population size. From this formula, we derive the time until deletion:

$$t(\text{delete cl}) = \frac{N}{2}. \tag{5.19}$$

In the next section, we use both the creation and the extinction time of representatives of starved niches to derive the minimum population size that guarantees the discovery, maintenance, and growth of starved niches.

5.6.4 Bounding the Population Size

The population size is a critical aspect that determines the niches that the system could maintain. In this section, we use the formulas calculated above and derive population size bounds to guarantee (1) that XCS will be able to maintain accurate representatives of starved niches, and (2) that representatives of starved niches will receive genetic events before being removed.

Minimum Population Size to Guarantee Representatives

In the previous section, we theoretically showed that XCS would be able to create representatives of starved niches regardless of the imbalance ratio. To guarantee that these classifiers will be preserved in the niche, we require that, before deleting any representative of a starved niche, another representative for the given niche be generated. Therefore, we use the formulas derived in the previous section and require that the time until deletion be greater than the time until a new representative of a the starved niche is created. That is, we require that

$$t(\text{delete } cl_{min}) > t(cl_{min}). \tag{5.20}$$

Using formulas 5.18 and 5.19, the expression can be rewritten as

$$N > 2n \left(\frac{\mu}{2}\right)^{k_m}. \tag{5.21}$$

which indicates that the population size has to increase linearly with the number of classes and exponentially with the order of the schema to guarantee that representatives will be maintained in starved niches. Note that this formula does not depend on the imbalance ratio. This means that XCS will be able to maintain accurate classifiers in starved niches regardless the imbalance ratio.

Population Size Bound to Guarantee Reproductive Opportunities

To ensure the growth of starved niches, we not only should guarantee that XCS would maintain representatives of starved niches, but that these representatives receive, at least, a genetic opportunity. Otherwise, XCS could be continuously creating and removing classifiers from starved niches, but not searching toward better classifiers. Therefore, here we derive a population size bound to ensure this condition.

In our model, we assume that the selection procedure chooses one of the strongest classifiers in the niche (the effect of different selection schemes will be studied in more detail in the next section). Then, the time required for a classifier of a starved niche to receive a genetic event is inversely proportional to the probability of activation of the niche to which it belongs, i.e.,

$$t(\text{GA } niche_{min}) = n \cdot (1 + ir), \tag{5.22}$$

which depends on the imbalance ratio and the number of classes.

To guarantee that these accurate classifiers of starved niches receive a genetic opportunity before being deleted, we require that $t(\text{delete } niche_{min}) > t(\text{GA } niche_{min})$, from which we derive the population size bound

$$N > 2n \cdot (1 + ir). \tag{5.23}$$

That is, the population size has to increase linearly with the number of classes and the imbalance ratio to warrant that accurate classifiers of starved niches will receive, at least, a genetic event before being deleted.

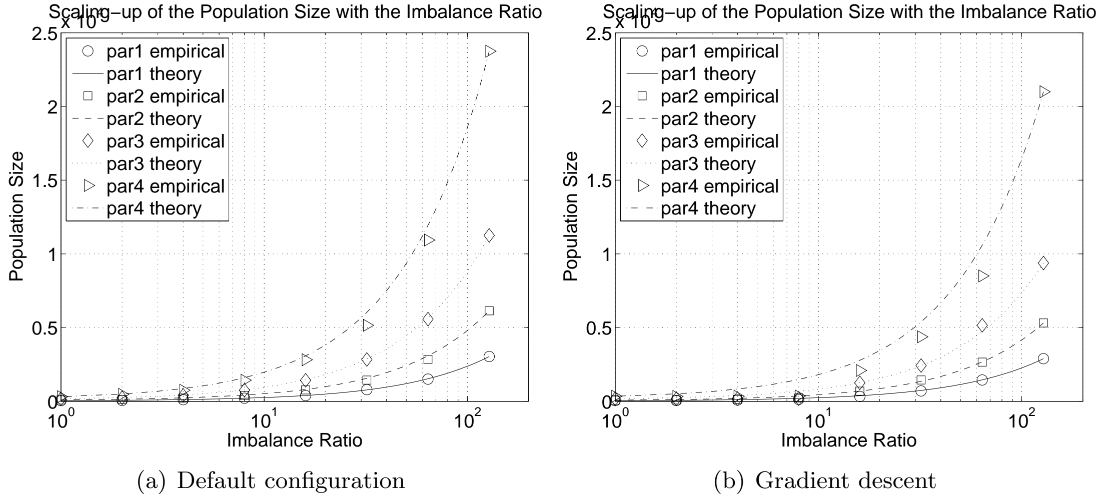
Thereupon, the models derived in this section explained the creation, the maintenance, and the growth of starved niches, showing that XCS is able to maintain representatives of starved niches regardless of the imbalance ratio and that the population size has to increase linearly with the imbalance ratio if we want to ensure that the niche will grow. In the next section, we empirically validate the population size bounds with a set of artificial problems.

5.6.5 Experimental Validation of the Models

In this section, we experimentally evaluate whether the population size bound increases linearly with ir as predicted by the bound in equation 5.23. We first analyze whether the theory fits the experimental results when all the assumptions are made. Later, we study the impact of breaking each one of the assumptions.

Experimental Validation When the Assumptions Are Satisfied

To examine whether the theory approximates accurately the empirical results when all the assumptions are met, we performed the following experiments. We ran XCS on the imbalanced



(a) Default configuration

(b) Gradient descent

Figure 5.6: Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and the default configuration with (a) Widrow Hoff rule update with adjusted β according to ir and (b) gradient descent parameter update with $\beta = 0.2$. The dots show the empirical results and lines plot linear increases with ir (according to the theory).

parity problem with $k = \{1, 2, 3, 4\}$, $\ell = 10$, and $ir = \{1, 2, 4, 8, 16, 32, 64, 128\}$, and we used the bisection procedure to obtain the minimum population size required to solve the problem. That is, for each parity problem and imbalance ratio, we ran XCS with an initial, randomly selected population size. If the run succeeded, we decreased the population size. Otherwise, we increased the population size. This procedure was repeatedly applied until we obtained the minimum population size with which XCS was able to solve the problem. We employed the following procedure to determine if an XCS run was successful. After training, we tested XCS with all the training instances and measured the proportion of correct classifications of instances of the majority class (TN rate) and the proportion of correct classifications of the minority class (TP rate). All these results were averaged over 50 different random seeds. We considered that XCS succeeded if the product of TP rate and TN rate was greater than a certain threshold θ (we set $\theta = 0.95$).

XCS was configured so that all the assumptions of the model were satisfied. Therefore, crossover was deactivated ($\chi = 0$), random deletion was used, and the GA was applied every time a niche was activated (θ_{GA}=0). The other parameters were set as $\alpha = 0.1$, $\epsilon_0 = 1$, $\nu = 10$, $\mu = 0.04$, $\theta_{del} = 20$, $\delta = 0.1$, $\theta_{sub} = ir$, $P_{\#} = 0.6$. We used tournament selection for the GA. We ran XCS during $\{10\,000 \cdot ir, 20\,000 \cdot ir, 40\,000 \cdot ir, 80\,000 \cdot ir\}$ iterations for the parity problem with $k = \{1, 2, 3, 4\}$ respectively; thus, given a problem, we ensured that the system received the same number of genetic opportunities for all imbalance ratios. Finally, to prevent having young over-general classifiers with poorly estimated parameters in the final population, we introduced $5,000 \cdot ir$ iterations with the GA switched off at the end of the learning process. In the remainder of this chapter, this configuration is referred to as the *default configuration*.

The two parameter update procedures proposed in section 5.4 were used: (1) Widrow-Hoff rule and (2) gradient descent. For the former method, we used the following heuristic procedure

to tune β. For each run, we supposed the worst case and assumed that over-general classifiers received 1 instance of the minority class and then ir instances of the majority class. Thus, we set β so that the error calculated for the most over-general classifier was approximately the same as the theoretical error provided by equation 5.9. We followed an iterative approach that incrementally discounted the value of β until a value that yielded error estimates that were close to the theoretical ones was found.

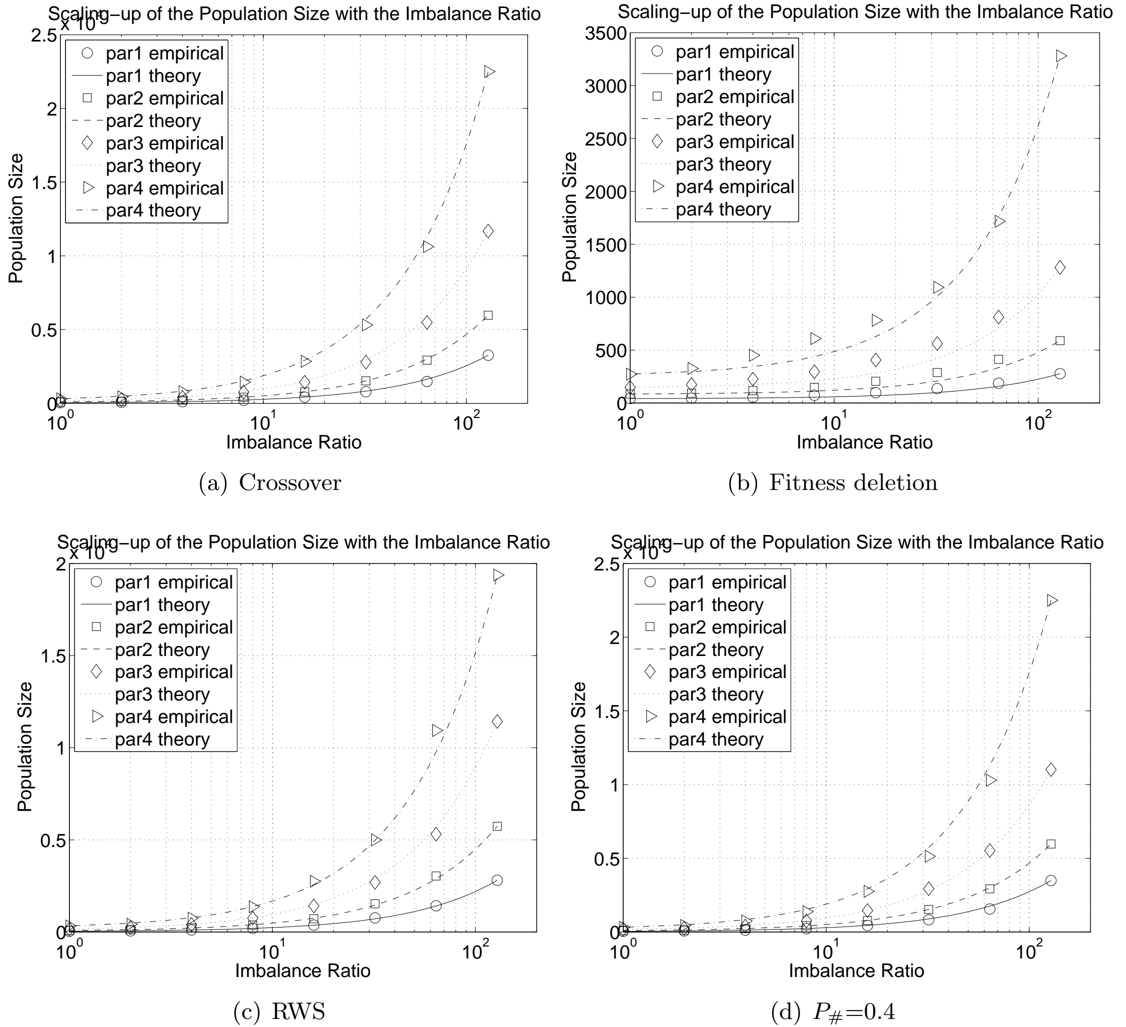
Figure 5.6 shows the minimum population size required to solve the parity with different building block sizes ($k = \{1, 2, 3, 4\}$) and imbalance ratios from $ir = 1$ to $ir = 128$ for Widrow-Hoff rule (figure 5.6(a)) and gradient descent (figure 5.6(b)). For each plot, the points depict the empirical values and the lines show the theoretical bounds. Note that the theory approximates the empirical results accurately for the two parameter update procedures and the different configurations and imbalance ratios. These results also permit establishing a comparison among the two parameter update procedures. The pairwise Wilcoxon statistical test (Wilcoxon, 1945), at $\alpha = 0.05$, indicated that gradient descent needed significantly smallest populations to solve the different configurations of the parity problem.

The results provided herein indicated that the theory nicely predicts the experiments when the assumptions of the models are met. In the next section, we investigate whether the population size bound is still valid when the different assumptions are not satisfied.

Impact of Breaking the Assumptions

Here, we repeated the experiments done in the previous section, but breaking each assumption. That is, we used the default configuration specified in the last section. Widrow-Hoff rule was employed for parameter estimation. Then, we ran XCS with (1) crossover, setting $\chi = 0.8$, and (2) the typical deletion scheme of XCS, configuring $\theta_{del} = 20$ and $\delta = 0.1$. Moreover, we also analyzed (3) the effect of replacing tournament selection with proportionate selection and (4) the impact of increasing the specificity in the initial population by setting $P\# = 0.4$. Figure 5.7 shows the minimum population size required in each configuration.

Several conclusions can be drawn from these results. First of all, it is worth noting that the theory nicely approximates the empirical results for all the experiments, although the initial assumptions were not satisfied. Figure 5.7(a) shows the curves obtained by XCS with crossover, which are equivalent to those evolved with the default configuration (see figure 5.6(a)) according to a Wilcoxon signed-ranks test at a significance level of 0.05. This suggests that the models are still valid although crossover is used in the experimental runs. Figure 5.7(b) plots the curves resulting from running XCS with the typical deletion scheme of XCS. The results clearly evidence the decrease in the population size required to solve the different configurations (the Wilcoxon signed-ranks test confirmed this observation). In any case, note that the the minimum population size still increases linearly with the imbalance ratio, as predicted by the theory. The configuration with proportionate selection (see figure 5.7(c)) yielded equivalent results to those obtained with the default configuration according to a Wilcoxon signed-ranks test at a significance level of 0.05. Finally, figure 5.7(d) illustrates the results obtained when there was a higher specificity in the initial populations. The pairwise analysis supports the hypothesis that a higher initial specificity requires larger population sizes to solve the parity with $k = 1$. For the other parity problems, no statistical differences were found.



(a) Crossover (b) Fitness deletion

(c) RWS (d) $P_{\#}$=0.4

Figure 5.7: Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and different XCS's configurations. The dots show the empirical results and lines plot linear increases with ir (according to the theory).

The overall experimentation conducted in this section showed an agreement between theory and experiments, even when the initial assumptions were not satisfied. Notice that no experiment broke the assumption that the GA is applied at each learning iteration. The effect of varying the frequency of application of the GA is carefully studied in the next section.

5.7 Occurrence-based Reproduction: The Role of θ_{GA}

Throughout all the analysis performed in the last section, we assumed that the different niches of the system receive a genetic opportunity each time they are activated (i.e., θ_{GA}=0). Consequently, nourished niches received more genetic events, and so, generated more offspring. This

section takes in consideration the effect of having $\theta_{GA} > 0$, revisits the models derived in the previous section, and shows that we can use θ_{GA} to re-balance the number of genetic opportunities that both starved and nourished niches receive. Finally, the new models are validated with the imbalanced parity problem.

5.7.1 Including θ_{GA} in the Generation Models

To analyze the impact of varying θ_{GA}, let us consider again the occurrence-based reproduction of both types of niches and calculate the period of application of the GA to the different niches. The frequency of activation of nourished niches ($F_{occ_{maj}}$) and the frequency of activation of starved niches ($F_{occ_{min}}$) are

$$F_{occ_{maj}} = \frac{1}{n \cdot m} \frac{ir}{1+ir} \tag{5.24}$$

and

$$F_{occ_{min}} = \frac{1}{n \cdot m} \frac{1}{1+ir}, \tag{5.25}$$

where m is the number of niches[2]. From these frequencies, we can compute the period of activation of each type of niche as

$$T_{occ_{maj}} = n \cdot m \frac{1+ir}{ir} \tag{5.26}$$

and

$$T_{occ_{min}} = n \cdot m(1+ir). \tag{5.27}$$

Once activated, the niche will receive a genetic event if the time since the last application of the GA on the niche exceeds θ_{GA}. Therefore, the period of application of the GA (T_{GA}) on a niche is

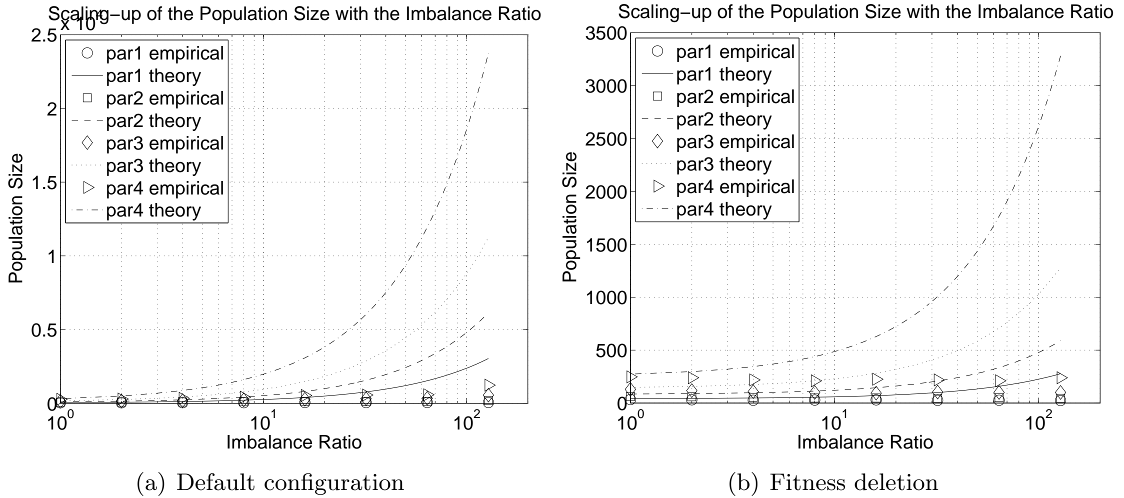
$$T_{GA} = \begin{cases} T_{occ} & \text{if } T_{occ} > \theta_{GA} \\ \theta_{GA} & \text{otherwise.} \end{cases} \tag{5.28}$$

That is, if the period of activation of a niche is greater than θ_{GA}, the classifiers that belong to the niche will receive a genetic event every time the action set is formed; thus, the period of application of the GA equals the period of niche activation. This is the case of the theoretical model, in which we assumed $\theta_{GA} = 0$. On the other hand, if $T_{occ} \leq \theta_{GA}$, T_{GA} is approximately θ_{GA}.

To give all niches the same number of genetic events, T_{GA} should be approximately the same for all the niches. Note that this can be easily satisfied by setting $\theta_{GA} = T^*_{occ}$, where T^*_{occ} is the period of the niche that is activated less frequently, i.e., $T^*_{occ} = T_{occ_{min}}$. Therefore, θ_{GA} should be set as follows:

$$\theta_{GA} \approx n \cdot m \cdot (1+ir). \tag{5.29}$$

[2]We introduce the number of niches in these equations since we are now modeling the occurrence of a specific niche



(a) Default configuration

(b) Fitness deletion

Figure 5.8: Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and different XCS's configurations with $\theta_{GA} = n \cdot m \cdot ir$. The points indicate the empirical values of the minimum population size required by XCS. The lines depict the theoretical increase calculated with the previous models, which assumed $\theta_{GA} = 0$.

Note that if the restriction of equation 5.29 is satisfied, all niches will receive approximately the same number of genetic events; moreover, as deletion is only activated after a GA application, the time of deletion of a classifier (see equation 5.19) would now increase linearly with the imbalance ratio. Therefore, XCS will be able to maintain starved niches without increasing the population size. The next section experimentally validates this assertion.

5.7.2 Experimental Validation

To validate the theory derived in the previous section, we ran the same experiments with the parity problem proposed in section 5.6.5. We configured the system with the default configuration, but we set $\theta_{GA} = n \cdot m \cdot ir$; Widrow Hoff rule was used to update classifier parameters. Figure 5.8(a) shows the minimum population size required to solve the parity problem with different building block sizes ($k = \{1, 2, 3, 4\}$) and imbalance ratios from $ir = 1$ to $ir = 128$. The points depict the empirical values. To analyze the differences introduced by adjusting $\theta_{GA} = n \cdot m \cdot ir$, the lines depict the population size increase predicted by the theoretical model calculated for the same configurations but with $\theta_{GA} = 0$ (see figure 5.6(a)).

The figure shows that, with the default configuration, the population size remained nearly constant for all the tested parity problems and imbalance ratios. This is because the effect of the imbalance ratio was counter-balanced by the increase of the period of application of the GA, as deduced in the previous section. The population size only presented a slight increase for $ir = 128$. This behavior can be easily explained as follows. At such imbalance ratios, the parameter update procedure decreases the value of β to prevent the oscillation of the parameters of over-general classifiers. For very small values of β, accurate offspring classifiers may lose against over-general parents at the beginning of the run, since their fitness increases slowly. As

the deletion procedure is random and these offspring receive a low number parameter updates, they may be removed before their parameters are correctly adjusted to the real value. Therefore, slightly larger populations may be set to let new accurate offspring persist in the population until their parameters are sufficiently updated.

To contrast this hypothesis, we ran the same experiments but used the typical deletion scheme of XCS. Figure 5.8(a) illustrates the minimum population sizes required for each configuration of the parity problem. The experimental results show that the population size remains constant as the imbalance ratio increases, even for the largest imbalance ratios. That is, the typical deletion scheme of XCS protects the young classifiers by giving more deletion probability to over-general, experienced classifiers.

With the present study, we have theoretically and empirically demonstrated that representatives of starved classifiers will be evolved independent of the population size. In the next section, we analyze the takeover time of these classifiers in more detail.

5.8 Takeover Time of Accurate Classifiers in Starved Niches

The study provided so far showed that XCS is able to create accurate classifiers of starved niches, and that these classifiers will receive, at least, a genetic opportunity before being deleted. This facet of the analysis set the population size requirements to guarantee that starved niches are represented. Also, the effect of increasing θ_{GA} was analyzed in detail. However, the conditions required in the previous models are not enough; to ensure full convergence, we have to warrant not only that starved niches will not be extinct but also that accurate classifiers will take over starved niches, removing the majority of over-general classifiers. Therefore, we have to analyze the competition between accurate classifiers of starved niches and over-general classifiers. This analysis is crucial because it permits extracting the upper bound on the admissible imbalance ratio under which XCS will be able to extract the key knowledge that resides in the minority class.

The purpose of this section is to model the takeover time of the best representatives of starved niches and determine the conditions under which starved niches will be extinguished. We first calculate the takeover time of accurate classifiers, which depends on (i) the initial stock of accurate classifiers in the niche and (2) the type of selection used by the GA. In LCSs, two selection procedures have mainly been considered: proportionate selection (Wilson, 1995) and tournament selection (Butz et al., 2005c). In this section, we model the takeover time of the best classifier of a niche for both selection schemes. Although we focus the analysis on the effect of class imbalances, note that the derived models can be used as general models for the two selection schemes. Then, we use the takeover time equations to calculate the extinction conditions of a niche, i.e., the conditions under which all representatives of a given starved niche will be removed from the population due to an overpressure toward generalization. As follows, we present the assumptions made for the analysis, develop the models for each type of selection, and experimentally validate the takeover time and extinction models.

5.8.1 Model Assumptions

Here, we provide the assumptions of the models. We derive takeover time models for problems with an arbitrary number of niches m. The models consider that all the m niches appear with the same frequency. In fact, in section 5.7, we showed that this could be easily achieved by setting θ_{GA} according to ir. Then, we incorporate the effect of the imbalance ratio in the error of the over-general classifier. That is, imagine that we have a two-class problem. For $ir = 1$, the error ϵ_o of the most over-general classifier will be $\epsilon_o \approx 500$, since this classifier will correctly predict half of the instances. As the imbalance ratio increases, ϵ_o decreases as shown in section 5.4. Thence, the imbalance ratio is intrinsically included in the difference between the errors of the over-general and the accurate classifiers. Thus, the takeover time models derived herein compute whether high imbalance ratios may discourage XCS to promote representatives of starved niches in favor of over-general classifiers.

To simplify the mathematical derivation of the models, we make the following assumptions. We consider that XCS has evolved a maximally general and accurate classifier cl_b, with error ϵ_b and numerosity n_b, for each niche of the system (this is ensured by the models provided in the last sections). Moreover, we assume that there is a single over-general classifier n_o, with error ϵ_o and numerosity n_o, which matches all the niches. As cl_b is maximally accurate, $\epsilon_o > \epsilon_b$. The same expression can be written in function of the classifiers accuracy (a inverse function of the error) as $\kappa_b > \kappa_o$. Therefore, our aim is to model the competition between accurate representatives of the different niches and the over-general classifier. For the analysis, we assume random deletion. We also consider that the GA is applied at each learning iteration and that both crossover and mutation are switched off. Therefore, the GA only selects two parents, copies and introduces them into the population, removing two other classifiers. The subsequent sections model the takeover time for proportionate and tournament selection under these assumptions.

5.8.2 Takeover Time for Proportionate Selection

In this section, we first derive the probability of selecting the best representative of a niche under proportionate selection, and then, we use this information to develop equations that model the evolution of the numerosity of this classifier in the niche. Under proportionate or roulette wheel selection (RWS), the selection probability of a classifier i depends on the ratio of its fitness F_i over the fitness of all classifiers in the action set. Without loss of generality, we assume that the classifier's fitness is a simple average of the classifier's relative accuracy. Thus, focusing on a single niche, we compute the fitness of classifiers cl_b and cl_o as

$$F_b = \frac{\kappa_b n_b}{\kappa_b n_b + \kappa_o n_o} = \frac{1}{1 + \rho n_r}$$

and

$$F_o = \frac{\kappa_o n_o}{\kappa_b n_b + \kappa_o n_o} = \frac{\rho n_r}{1 + \rho n_r},$$

where $n_r = n_o/n_b$ and ρ is the ratio between the accuracy of cl_o and the accuracy of cl_b ($\rho = \kappa_o/\kappa_b$). ρ can also be viewed as the fitness separation between cl_o and cl_b. The probability P_s of selecting the best classifier cl_b in the niche is computed as

$$P_{s_{RWS}} = \frac{F_b}{F_b + F_o} = \frac{1}{1 + \rho n_r}.$$

Once selected, each classifier is copied and inserted into the population while one classifier is randomly deleted from the niche with probability $P_{del}(cl_j) = n_j/N$, where N is the population size. With this information, as proceeds we model the evolution of the best classifier in a niche.

Evolution of the Best Classifier

The numerosity of the best classifier cl_b at time $t+1$, $n_{b,t+1}$, given the numerosity of the classifier at time t, $n_{b,t}$, will

- increase in the next generation if the GA is applied to the niche, cl_b is selected by the GA, and another classifier is selected for deletion;
- decrease if (a) the GA is applied to the niche, cl_b is not selected by the GA, but cl_b is selected for deletion or if (b) the GA is not applied to the niche and cl_b is selected for deletion;
- remain the same, in all the other cases.

More formally,

$$n_{b,t+1} = \begin{cases} n_{b,t}+1 & \frac{1}{m}\frac{1}{1+\rho n_{r,t}}\left(1-\frac{n_{b,t}}{N}\right), \\ n_{b,t}-1 & \frac{1}{m}\left(1-\frac{1}{1+\rho n_{r,t}}\right)\frac{n_{b,t}}{N}+\frac{m-1}{m}\frac{n_{b,t}}{N}, \\ n_{b,t} & \text{otherwise.} \end{cases}$$

where m is the number of niches in the problem. Grouping the above equations, we obtain

$$n_{b,t+1} = n_{b,t} + \frac{1}{m}\cdot\frac{1}{1+\rho n_{r,t}}\left(1-\frac{n_{b,t}}{N}\right) - \frac{1}{m}\left(1-\frac{1}{1+\rho n_{r,t}}\right)\frac{n_{b,t}}{N} - \frac{m-1}{m}\frac{n_{b,t}}{N}, \tag{5.30}$$

which can be expressed as

$$n_{b,t+1} = n_{b,t} + \frac{1}{m}\frac{1}{1+\rho n_{r,t}} - \frac{n_{b,t}}{N}. \tag{5.31}$$

This expression can be rewritten in terms of the proportion P_t of classifiers cl_b in the whole population, i.e.,

$$P_t = \frac{n_{b,t}}{N}. \tag{5.32}$$

Considering that the numerosity of the best classifier in each niche is $n_{b,t}$, we write that the numerosity of the over-general classifier $n_{o,t}$ is $n_{o,t} = N - m\cdot n_{b,t}$, and thus, $n_{r,t}$ can be expressed as

$$n_{r,t} = \frac{n_{o,t}}{n_{b,t}} = \frac{1-m\cdot P_t}{P_t}. \tag{5.33}$$

Replacing equations 5.33 and 5.32 into equation 5.31, we obtain

$$P_{t+1} = P_t + \frac{1}{Nm}\frac{P_t}{P_t+\rho(1-mP_t)} - \frac{1}{N}P_t. \tag{5.34}$$

Assuming $P_{t+1} - P_t \approx dp/dt$, we have

$$\frac{dp}{dt} \approx P_{t+1} - P_t = \frac{1}{Nm}\frac{P_t}{P_t + \rho(1 - mP_t)} - \frac{1}{N}P_t = \tag{5.35}$$

$$= \frac{P_t - mP_t^2 - \rho m P_t(1 - mP_t)}{Nm\left[P_t + \rho(1 - mP_t)\right]}. \tag{5.36}$$

That is,

$$\frac{P_t(1 - m\rho) + \rho}{P_t\left[(1 - \rho m) - mP_t(1 - \rho m)\right]}dp = \frac{1}{Nm}dt, \tag{5.37}$$

which can be solved by integrating each side of the equation between the initial proportion P_0 of cl_b and the final proportion P_F of cl_b up to which cl_b has taken over the population. Note that, assuming m balanced niches, cl_b will take over, at most, a proportion $1/m$ of the population.

$$\int_{P_0}^{P_F} \frac{P_t(1 - m\rho) + \rho}{P_t\left[(1 - \rho m) - mP_t(1 - \rho m)\right]}dp = \frac{1}{Nm}\int dt = \frac{t}{Nm}. \tag{5.38}$$

This integral can be solved as follows

$$\frac{t}{Nm} = \int_{P_0}^{P_F} \frac{1}{1 - mP_t}dp + \frac{\rho}{1 - m\rho}\int_{P_0}^{P_F} \frac{1}{P_t[1 - mP_t]} = \tag{5.39}$$

$$= \left[-\frac{1}{m}\ln(1 - mP_t) - \frac{\rho}{1 - m\rho}\ln\left|\frac{1 - mP_t}{P_t}\right|\right]_{P_0}^{P_F} = \tag{5.40}$$

$$= -\frac{1}{m}\ln\left(\frac{1 - mP_0}{1 - mP_F}\right) + \frac{\rho}{1 - m\rho}\ln\left|\frac{P_F(1 - mP_0)}{P_0(1 - mP_F)}\right|. \tag{5.41}$$

Since $P_t < 1/m$, we can rewrite the expression as

$$\frac{t}{Nm} = -\frac{1}{m}\ln\left(\frac{1 - mP_0}{1 - mP_F}\right) + \frac{\rho}{1 - m\rho}\ln\left(\frac{P_F(1 - mP_0)}{P_0(1 - mP_F)}\right), \tag{5.42}$$

from which we derive the takeover time of cl_b in roulette wheel selection

$$t^*_{RWS} = Nm\left[\frac{1}{m}\ln\left(\frac{1 - mP_0}{1 - mP_F}\right) + \frac{\rho}{1 - m\rho}\ln\left(\frac{P_F(1 - mP_0)}{P_0(1 - mP_F)}\right)\right]. \tag{5.43}$$

The takeover time formula depends on (1) the fitness separability ρ and (2) the number of niches m. If ρ increases, the influence of the second logarithm also increases; therefore, as the accuracies of cl_b and cl_o get closer, the takeover time increases.

In this subsection we provided a closed-form solution of the takeover time for proportionate selection. In the next subsection, we derive the conditions under which the best classifier will not take over its niche.

Conditions for the Extinction of Starved Niches under Proportionate Selection

With the formulas derived above, we analyze under which circumstances the best classifier will not be able to take over the population, and then, we relate it to the imbalance ratio. For

this purpose, we take equation 5.35 and analyze under which conditions the increment of the numerosity of the best classifier will be negative; in this case, the best classifier will lose copies in favor of over-general classifiers. We can write this expression as

$$\frac{P_t - mP_t^2 - \rho m P_t(1 - mP_t)}{Nm\left[P_t + \rho(1 - mP_t)\right]} < 0, \tag{5.44}$$

that is,

$$\frac{P_t\,(1 - m\rho)(1 - mP_t)}{Nm\left[P_t(1 - m\rho) + \rho\right]} < 0. \tag{5.45}$$

This condition holds when the numerator is positive and the denominator is negative and viceversa. Thus, we search for values of ρ and m that result in combinations of positive numerator and negative denominator and viceversa. Assuming that $P_t < \frac{1}{m}$, we have the following cases. For $\rho < \frac{1}{m}$, both numerator and denominator take positive values. Therefore, for $\rho < \frac{1}{m}$, the best classifier will always take over the population. For $\rho = \frac{1}{m}$, the expression is 0, indicating that numerosity of the best classifier would remain constant. For $\rho > \frac{1}{m}$, the numerator is always negative. Then, the whole expression will be negative if the denominator is positive, i.e.,

$$Nm\left[P_t(1 - m\rho) + \rho\right] > 0, \tag{5.46}$$

which can be written as

$$P_t < \frac{\rho}{m\rho - 1}. \tag{5.47}$$

Having that $0 < P_t < \frac{1}{m}$ and $\frac{1}{m} < \rho \leq 1$, this expression is always satisfied for $m \geq 2$. Thence, this theoretically demonstrates that for $m \geq 2$ the best representative will not be able to take over the niche if

$$\rho > \frac{1}{m}, \tag{5.48}$$

That indicates that the best classifier will not take over its niche if the ratio of the accuracy of the over-general classifier to the accuracy of the best classifier is greater than $1/m$, that is, the fitness separability between both classifiers is small.

Note that the expression in equation 5.47 can be easily related to the imbalance ratio by identifying that $\rho = \frac{\kappa_o}{\kappa_b}$, where the accuracy of the over-general classifier can be computed from its error, which is expressed in equation 5.9. That is, recognizing that the accuracy of the representative of each niche is one ($\kappa_b = 1$), and that the accuracy of the over-general classifier κ_o is

$$\kappa_o = \alpha \left(\frac{\epsilon_o}{\epsilon_0}\right)^{-\nu}, \tag{5.49}$$

and considering the relation between the imbalance ratio and the error computed in equation 5.9, we obtain that

$$\rho = \alpha \left(\frac{(1 + ir)^2 \epsilon_0}{2 \cdot ir \cdot R_{max}}\right)^{\nu}. \tag{5.50}$$

Replacing equation 5.50 into equation 5.48, we can write that the best classifier will not take over its niche if

$$\alpha \left(\frac{(1+ir)^2 \epsilon_0}{2 \cdot ir \cdot R_{max}} \right)^{\nu} > \frac{1}{m}. \tag{5.51}$$

This expression can be derived as

$$ir^2 + \left(2 - \frac{(m\alpha)^{\nu} 2 R_{max}}{\epsilon_0} \right) ir + 1 > 0. \tag{5.52}$$

Recognizing that

$$1 << \frac{(m\alpha)^{\nu} 2 R_{max}}{\epsilon_0}, \tag{5.53}$$

and making further simplifications, we can get that, under proportionate selection, the best classifier will not take over its niche if

$$ir > \frac{(m\alpha)^{\nu} 2 R_{max}}{\epsilon_0}. \tag{5.54}$$

Note that the maximum accepted ir can be modified by decreasing ϵ_0.

After developing the same models for tournament selection, in section 5.8.4, we experimentally validate the takeover time under proportionate selection, and show that the best classifier cannot take over the niche for $\rho > 1/m$.

5.8.3 Takeover Time for Tournament Selection

To develop takeover time models for tournament selection, we assume that the tournament size s is fixed during all the learning process. That is, in each GA event, tournament selection randomly chooses s classifiers in the action set and selects the one with the highest fitness. As before, we assume that cl_b is the best classifier in the niche and cl_o is the over-general classifier; in terms of tournament selection, this translates into requiring that $f_b > f_o$, where f_i is the fitness of the micro-classifiers associated with cl_i (i.e., $f_i = F_i/n_i$). Given this scenario, the probability of selecting the best classifier is

$$P_{s_{TS}} = \left[1 - \left(1 - \frac{n_o}{n} \right)^s \right], \tag{5.55}$$

where n is the numerosity of the niche, i.e., $n = n_b + n_o$. Thus, the probability of selecting the best classifier is one minus the probability that this classifier does not participate in the tournament. With this information, the next subsections model the evolution of the best classifier, provide some particular expressions of the takeover time for tournament selection, and extract the critical bounds beyond which the best representative will not take over its niche.

Evolution of the Best Classifier

We first model the evolution of the numerosity of the best classifier cl_b at time $t+1$, $n_{b,t+1}$, given the numerosity of the classifier at time t, $n_{b,t}$, which will

- increase if the GA is applied to the niche, cl_b is selected to participate in the tournament, and another classifier in the population is selected for deletion;
- decrease if (1) the GA is applied to the niche, cl_b is not selected to participate in the tournament, but it is selected for deletion; or if (2) the GA is applied to another niche, and cl_b is selected for deletion;
- remain the same otherwise.

More formally,

$$n_{b,t+1} = \begin{cases} n_{b,t}+1 & \frac{1}{m}\left[1-\left(1-\frac{n_{b,t}}{n}\right)^s\right]\left(1-\frac{n_{b,t}}{N}\right), \\ n_{b,t}-1 & \frac{1}{m}\left(1-\frac{n_{b,t}}{n}\right)^s\frac{n_{b,t}}{N} + \frac{m-1}{m}\frac{n_{b,t}}{N}, \\ n_{b,t} & \text{otherwise.} \end{cases}$$

Grouping the above equations we can derive the expected numerosity of cl_b,

$$n_{b,t+1} = n_{b,t} + \frac{1}{m}\left[1-\left(1-\frac{n_{b,t}}{n}\right)^s\right]\left(1-\frac{n_{b,t}}{N}\right) - \frac{1}{m}\left(1-\frac{n_{b,t}}{n}\right)^s\frac{n_{b,t}}{N} - \frac{m-1}{m}\frac{n_{b,t}}{N}, \quad (5.56)$$

from which we obtain

$$n_{b,t+1} = n_{b,t} + \frac{1}{m}\left[1-\left(1-\frac{n_{b,t}}{n}\right)^s\right] - \frac{n_{b,t}}{N}. \quad (5.57)$$

As done for proportionate selection, we rewrite the formula above in function of the proportion P_t of classifiers cl_b in the whole population, i.e.,

$$P_t = \frac{n_{b,t}}{N}. \quad (5.58)$$

From which we can calculate n_o as

$$n_o = N - mn_b = N - mNP_t = N(1-mP_t), \quad (5.59)$$

and n as

$$n = n_b + n_o = NP_t + N(1-mP_t). \quad (5.60)$$

Substituting equations 5.59 and 5.60 into equation 5.57, we obtain

$$NP_{t+1} = NP_t + \frac{1}{m}\left[1-\left(1-\frac{P_t}{1+P_t(1-m)}\right)^s\right] - P_t, \quad (5.61)$$

$$P_{t+1} = P_t + \frac{1}{mN}\left[1-\left(1-\frac{P_t}{1+P_t(1-m)}\right)^s\right] - \frac{1}{N}P_t. \quad (5.62)$$

Assuming $\frac{dp}{dt} \approx P_{t+1} - P_t$, we derive

$$\frac{dp}{dt} \approx P_{t+1} - P_t = \frac{1}{mN}\left[1-\left(1-\frac{P_t}{1+P_t(1-m)}\right)^s - mP_t\right]. \quad (5.63)$$

That is,

$$\frac{1}{mN}dt = \frac{1}{1 - \left(1 - \frac{P_t}{1+P_t(1-m)}\right)^s - mP_t}dp. \tag{5.64}$$

If we integrate each side of the expression, we obtain

$$\frac{1}{mN}dt = \frac{t}{mN} = \int_{P_0}^{P_F} \frac{1}{1 - \left(1 - \frac{P_t}{1+P_t(1-m)}\right)^s - mP_t}dp \tag{5.65}$$

The above integral cannot be solved in general for any value of s and m. Nonetheless, note that this analysis still provides essential information since (1) it permits calculating particular expressions of the takeover time and (2) enables the derivation of the conditions for the extinction of starved niches. With the aim of showing some particular cases of the takeover time in tournament selection, the next section provides (1) a closed-form solution of the integral for problems with a single niche and any selection pressure s and (2) a closed-form solution for problems with two niches ($m = 2$) and tournament size $s = 2$, since $s = 2$ is the lowest pressure that can be applied.

Particular Expressions of the Takeover Time for Tournament Selection

In this section, we use equation 5.65 to derive some particular expressions of the takeover time. Expressions for other tournament sizes and niche sizes can be computed by replacing the corresponding values into equation 5.65.

Number of Niches m=1

Replacing $m = 1$ into equation 5.65 we obtain the following expression

$$t = N\int_{P_0}^{P_F} \frac{1}{1 - (1-P_t)^s - P_t}dp = \tag{5.66}$$

$$= \int_{P_0}^{P_F} \frac{1}{1-P_t}dp + \int_{P_0}^{P_F} \frac{(1-P_t)^{s-2}}{1-(1-P_t)^{s-1}}dp = \tag{5.67}$$

$$= N\left[\ln\left(\frac{1-P_0}{1-P}\right) + \frac{1}{s-1}\ln\left[\frac{1-(1-P)^{s-1}}{1-(1-P_0)^{s-1}}\right]\right]. \tag{5.68}$$

Thence, the takeover time of cl_b for tournament selection is

$$t^*_{TS_{m=1}} = N\left[\ln\left(\frac{1-P_0}{1-P_F}\right) + \frac{1}{s-1}\ln\left[\frac{1-(1-P_F)^{s-1}}{1-(1-P_0)^{s-1}}\right]\right], \tag{5.69}$$

which depends on the initial P_0 and final P_F proportion of classifiers, and it is always positive regardless of the values of P_0 and P_F. Therefore, if the problem contains a single niche—a situation that is nonrealistic in real-world problems—the best classifier always will take over the niche (Orriols-Puig et al., 2007d).

Number of Niches m=2, Tournament Size s=2

Substituting $m = 2$ and $s = 2$ into equation 5.65, we obtain

$$t = 2N \int_{P_0}^{P_F} \frac{1}{1 - \left(1 - \frac{P_t}{1-P_t}\right)^s - 2P_t} dp = \tag{5.70}$$

$$= \int_{P_0}^{P_F} \frac{(1-P_t)^2}{(1-P_t)^2 - (1-2P_t)^2 - 2P_t(1-P_t)^2} dp = \tag{5.71}$$

$$= \int_{P_0}^{P_F} \frac{(1-P_t)^2}{P_t^2(1-2P_t)} dp = \int_{P_0}^{P_F} \frac{1}{P_t^2} dp + \int^{P_F} P_0 \frac{1}{1-2P_t} dp = \tag{5.72}$$

$$= \left[-\frac{1}{P_t} - \frac{1}{2} \ln(1-2P_t) \right]_{P_0}^{P_F} = 2N \left[\frac{1}{P_0} - \frac{1}{P_F} + \frac{1}{2} \ln \frac{1-2P_0}{1-2P_F} \right]. \tag{5.73}$$

Then, the takeover time for tournament selection for $m = 2$ and $s = 2$ is

$$t_{TS_{m=2,s=2}} = 2N \left[\frac{1}{P_0} - \frac{1}{P_F} + \frac{1}{2} \ln \frac{1-2P_0}{1-2P_F} \right], \tag{5.74}$$

which depends linearly on the initial and the final proportion of the best classifier in the population and logarithmically on the difference between them. However, it does not depend on any scale between the fitness of cl_b and cl_o. Moreover, as $P_F > P_o$, the takeover time is always positive; this indicates that the best classifier always will be able to takeover its niche, regardless of the imbalance ratio of the problem. Note that this analysis has been made for the lowest possible selection pressure. Therefore, this conclusion can be extended to any tournament size for problems with two niches.

Finally, we compare the conclusions provided by this analysis with those obtained with proportionate selection. In 5.8.2, we theoretically demonstrated that, with proportionate selection, the best classifier would not be able to take over its niche if $\rho \geq 0.5$ for problems with two niches (see equation 5.48). Thus, tournament selection appears to be more robust in highly imbalanced data sets when the fitness separation between accurate and over-general classifiers is low.

This subsection provided some specific expressions of the takeover time for tournament selection. Note that, although the general closed-form solution could not be extracted, the analysis is still crucial since it enables us to detect the critical bounds of the system behavior when learning from imbalanced domains, which are derived in the next subsection.

Conditions for the Extinction of Starved Niches under Tournament Selection

To derive the conditions for niche extinction for tournament selection, we consider the differential equation obtained during the derivation of the model (see equation 5.57) and require that the increase in the numerosity of the best classifier be negative. That is,

$$\frac{1}{m} \left[1 - \left(1 - \frac{n_{b,t}}{n} \right)^s \right] - \frac{n_{b,t}}{N} < 0, \tag{5.75}$$

which can be rewritten as

$$1 - m\frac{n_{b,t}}{N} < \left(1 - \frac{n_{b,t}}{n} \right)^s. \tag{5.76}$$

This expression depends on the number of niches m, the number of accurate classifiers in the niche $n_{b,t}$, the niche size n, and the population size N. Note that the left-most term decreases linearly with m, whilst the right-most term decreases exponentially with s. Therefore, the condition will be satisfied, i.e., the best classifier will not be able to take over its niche, for low values of s combined with moderate and large number of niches m.

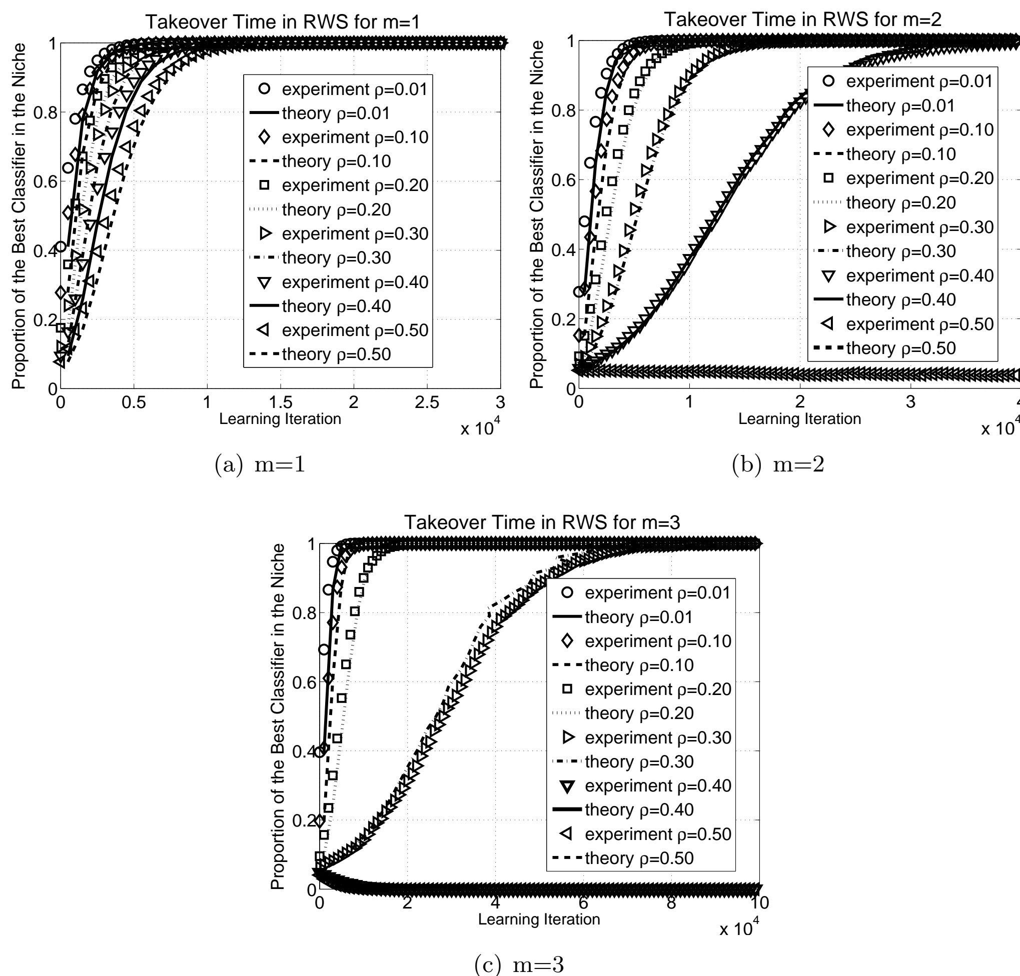
Thence, different from proportionate selection, the imbalance ratio does not appear as a decision variable in the extinction model for tournament selection. This is normal since we assumed that the parameters of classifiers are accurately estimated; thus, the error of cl_o will be always greater than the error of cl_b, and tournament selection will select cl_b when both classifiers compete in the same tournament. Thereupon, the extinction condition is basically guided by the selection pressure s—that is, the number of classifiers that will participate in the tournaments—and the size of the niches, which models the effect of population-based deletion.

5.8.4 Experimental Validation of the Takeover Time Models

Here, we experimentally validate (1) the theoretical models of takeover time and (2) the conditions for the extinction of starved niches for both proportionate and tournament selection. For this purpose, we ran XCS on the parity problem setting the number of niches m of the system. We initialized the population with $P_0 \cdot N$ copies of maximally accurate classifiers (equally distributed in the m niches) and $(1 - m \cdot P_0) \cdot N$ copies of an over-general classifier that appeared in all the niches. The prediction error of the over-general classifier was deterministically calculated as $\epsilon_{ovg} = \epsilon_0 \left(\frac{\rho}{\alpha}\right)^{\nu}$. In our experiments, we set α=0.1 and $\nu = 10$. Note that varying ρ, we are changing the fitness scaling between cl_o and cl_b. This could be equivalently done by maintaining ρ and varying ν, as done by Kharbat et al. (2005).

Figure 5.9 shows the evolution of the proportion of the best classifier in one of the niches for RWS and (a) m=1, (b) m=2, and (c) m=3 number of niches. The empirical data are averages over 50 runs. According to the model, we computed the proportion of the best classifier in the population. Therefore, the average of this proportion would tend to $1/m$, as approximately the same resources would be placed in each niche. In the figure, we plot the proportion of the best classifier in the niche, which ranges from 0 to 1. Figure 5.9 shows a perfect match between the theory and the empirical results. It also shows that, as predicted by the models derived in section 5.8.3, the ratio of the accuracy of the over-general classifier to the accuracy of the best classifier is a crucial aspect. For the problem with two niches, the best classifier could not take over its niche if $\rho \geq 0.5$ (see figure 5.9(b)), as predicted by the niche extinction model provided in equation 5.47. This behavior was also present in the problem with three niches 5.9(c), where neither $\rho = 0.4$ nor $\rho = 0.5$ let the best classifiers take over their niches.

Figure 5.10 shows the evolution of the proportion of the best classifier in the niche for TS and (a) m=1, (b) m=2, and (c) m=3 number of niches. For the problem with one niche, we depict the selection pressures of s={1,2,10}. Figure 5.10(a) shows a perfect match between the theoretical model and the experiments. Tournament selection is not influenced by the ratio of the accuracy of the best classifier to the accuracy of the over-general classifier. That is, we ran experiments with different values of ρ, obtaining equivalent results to those plotted in the figure. Moreover, it is also shown that increasing the tournament size results in faster convergence times. For s>10, the takeover time barely decreases (these curves are not depicted in the figure for clarity).



(a) m=1

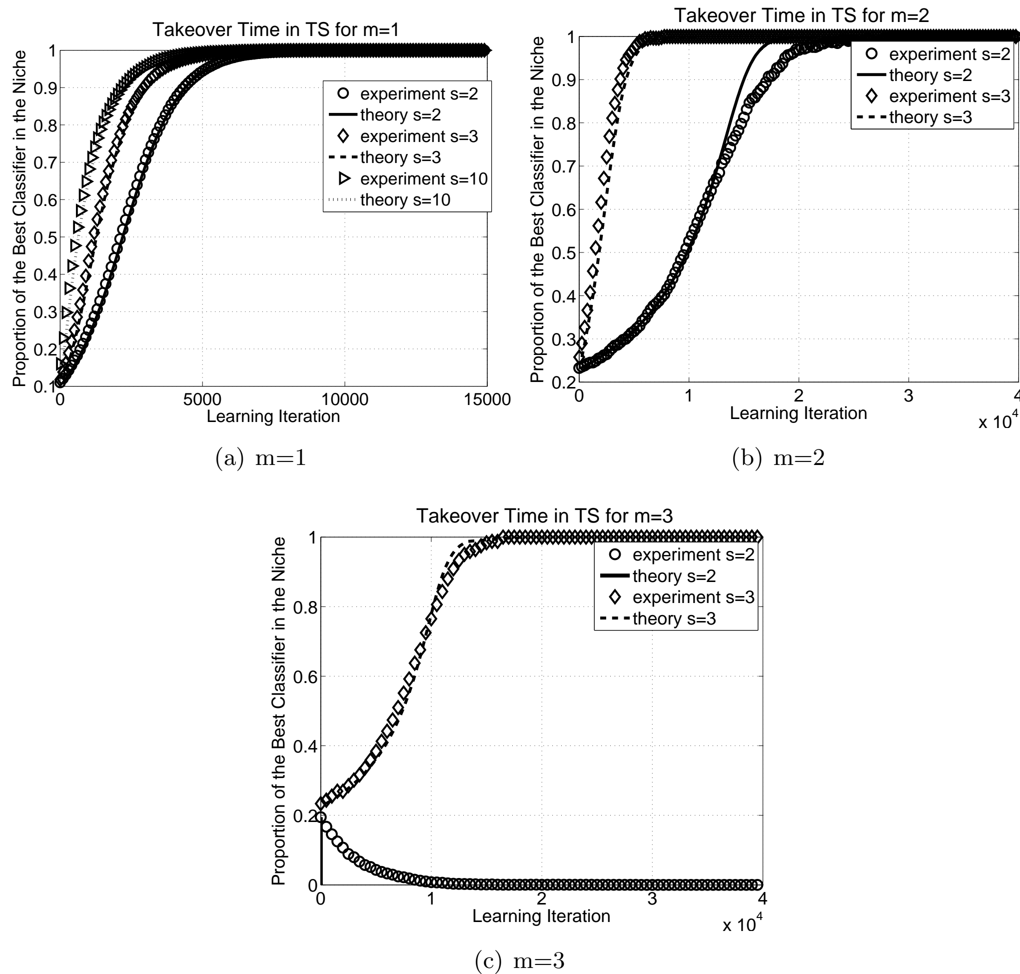
(b) m=2

(c) m=3

Figure 5.9: Takeover time in proportionate selection for (a) m=1, (b) m=2, and (c) m=3 and ρ={0.01,0.10,0.20,0.30,0.40,0.50}.

For the problems with two and three niches, we plot the evolution of the best classifier under tournament selection for s=2 and s=3 (see figures 5.10(b) and 5.10(c)). The theoretical model was calculated for each case by replacing m and s into equation 5.65 and solving the integral. Again, the theory nicely approximates the experimental results. For m=2, the best classifier can take over its niche, even for the lowest possible selection pressure. For m=3, the best classifier can take over its niche only if $s \geq 3$. This experimental evidence agrees with the niche extinction model supplied in equation 5.76. That is, as the number of niches increases, the selection pressure needs to be stronger to let the best classifier emerge, regardless of the initial proportion of this classifier in the population.

The overall analysis also permits comparing the two selection approximations and relating them to the class-imbalance problem. For low values of ρ, that is, when the fitness of the best classifiers is much higher than the fitness of the over-general classifiers, proportionate selection can yield the fastest takeover times. Therefore, proportionate selection appears as the most



(a) m=1

(b) m=2

(c) m=3

Figure 5.10: Takeover time in tournament selection for (a) m=1, (b) m=2, and (c) m=3.

appealing alternative for domains in which there is a high separation among the fitness of accurate and inaccurate rules. As pointed out by Kharbat et al. (2005), in balanced domains, this can be easily done by tuning the fitness pressure ν. Nonetheless, in imbalanced domains, the error of over-general classifiers decreases with the imbalance ratio (see equation 5.9). In these cases, proportionate selection may promote the existence of over-general classifiers. Thence, tournament selection appears to be the most robust selection scheme for imbalanced domains, provided that s is sized properly. A combination of both schemes seems to be an attractive alternative to deal with new real-world problems with unknown characteristics.

With the study of the takeover time and the conditions of extinction of starved niches, we completed the analysis of the different facets proposed in section 5.3.4. Facetwise models have provided key insights and points of view of the problem. The next section integrates all these models, provides configuration guidelines based on the lessons learned from them, and shows that, following these recommendations, XCS is able to solve highly imbalanced problems that previously eluded solution.

5.9 Lessons Learned from the Models

In this section, we use qualitative arguments to integrate the different models and extract lessons from the whole design decomposition and facetwise analysis. Then, we use the derived facetwise analysis as a tool for designing a set of guidelines that should be satisfied to warrant that XCS will be able to extract key knowledge from rare classes. We experimentally show that, if the system is configured according to these recommendations, XCS is able to solve problems with large imbalance ratios that previously eluded solution. More specifically, we show that XCS with a proper configuration solves the imbalanced multiplexer problem with large imbalance ratios, for which we showed that first generation XCS failed to discover the minority class in section 5.2.2.

5.9.1 Patchquilt Integration of the Facetwise Models

Along this chapter, we have studied the five subproblems, identified by the design decomposition, that may impair XCS from learning from rare classes. Now, we integrate the different models in a general framework and highlight the lessons derived from each particular model and, in general, from their interaction. This integration permits us to (1) identify under which cases XCS will not be able to learn from the minority class and (2) establish configuration recommendations to ensure convergence if possible. To achieve this, we revisit the models from the most restrictive one to the less restrictive one, setting the three steps that need to be guaranteed to ensure convergence and pointing out several configuration guidelines.

1. Niche extinction models (see section 5.8) set the conditions on the maximum imbalance ratio admitted by XCS to create key knowledge from the minority class for proportionate and tournament selection (see equations 5.47 and 5.76). If the requirements are met, takeover time models predict the convergence time inside each niche, that is, the number of learning iterations that an accurate, maximally general representative will need to take over its niche. Satisfying the requirements identified by the extinction time models is a necessary but not sufficient condition. That is, these models indicate that, if the identified restrictions are met, representative classifiers will take over their niche. Therefore, we have to guarantee that, at some point, these representatives are fed into the population.

2. Classifier parameters have to be correctly estimated by one of the methods presented in section 5.4. Otherwise, XCS will not be able to distinguish between over-general and accurate classifiers. We experimentally showed that the Widrow-Hoff rule provided very accurate estimates of the parameters if β was properly set. For this reason, we took this approach in our experiments and proposed an heuristic that automatically sets the value of β ensuring that the error estimate of over-general classifiers is close to the theoretical value (see section 5.6.5).

3. If the above two conditions are satisfied, we can ensure convergence by either (1) sizing the population according to the imbalance ratio or (2) setting θ_{GA} depending on the imbalance ratio according to the models evolved in sections 5.5 and 5.6. It is worth noting that the models work independently of whether covering is able to provide the initial population with schemas of starved niches, as identified in section 5.4.

In the next section, we show that, if the recommendations derived from the models are followed, XCS can solve extremely imbalanced data sets.

5.9.2 Solving Problems with Large Imbalance Ratios

In this section, we take again the imbalance multiplexer problem and show that the lessons extracted from the facetwise analysis enable us to properly configure XCS, letting the system learn from imbalance ratios that previously eluded solution. That is, in section 5.2.2 we showed that XCS was not able to extract the key knowledge from rare classes when $ir > 32$. Now, we run the same experiments and illustrate that, with the better understanding acquired along the facetwise analysis, we can set XCS so that it can solve the multiplexer problem with extremely large imbalance ratios; in particular, we solve the problem for $ir = 1024$.

Before proceeding with the analysis of the results, we first explain how the recommendations have been followed to configure XCS. We took the default configuration as a starting point (see section 5.6.5), but we used the typical deletion scheme of XCS, set $N = 1000$ and probability of crossover $\chi = 0.8$, and employed tournament selection with $\sigma = 0.4$ (that is, 40% of the classifiers in the action set participate in the tournament). This also corresponds to the configuration used in section 5.2.2. Besides, the configuration satisfied the conditions required by the different models, that is:

1. As we used tournament selection, we need to satisfy the condition that

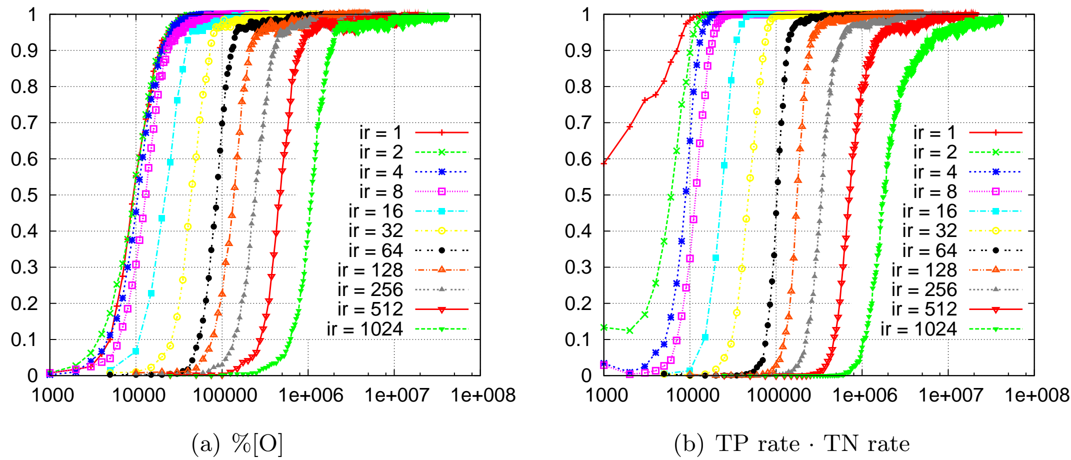
$$1 - m\frac{n_{b,t}}{N} > \left(1 - \frac{n_{b,t}}{n}\right)^s, \tag{5.77}$$

   to ensure that the best classifier will take over its niche (note that we change the inequality with respect to equation 5.76 since, now, we require that the best classifier take over its niche). From this equation, we know that N=1 000 and m=32, but $n_{b,t}$ and n are unknown. We assume the worst case, that is, that we only have a best classifier per niche. Thence, $n_{b,t} = 1$ and $n = 1000 - 32 = 968$. Substituting these parameters into the equation, we obtain that $s \geq 29$; this condition is satisfied since 40% of the n classifiers in each niche are selected to be in the tournament. This condition is also satisfied for larger values of $n_{b,t}$. Therefore, this indicates that, if discovered, representatives of starved niches will take over their niches. It is worth noting that similar results can be achieved with proportionate selection.

2. The classifier parameters are estimated according to the Widrow-Hoff rule, but setting β appropriately so that the error of over-general classifiers does not decrease rapidly to zero. For this purpose, we used the heuristic method mentioned in the previous section.

3. Finally, to guarantee that all the niches receive the same number of genetic opportunities, and so, warrant that representatives of starved niches will be created regardless of the imbalance ratio, we set $\theta_{GA} = n \cdot m \cdot ir$ as proposed in section 5.7.

Figure 5.11 plots the results obtained by XCS in the imbalanced 11-bit multiplexer problem with imbalance ratios ranging from $ir = 1$ to $ir = 1024$. More specifically, figure 5.11(a) illustrates the evolution of the proportion of the optimal population %[O] achieved by XCS. That



(a) %[O]

(b) TP rate · TN rate

Figure 5.11: Evolution of (a) the proportion of the optimal population and (b) the product of TP rate and TN rate in the 11-bit multiplexer with imbalance ratios ranging from ir=1 to ir=1024.

is, XCS was expected to evolve 32 optimal classifiers, each one representing a different niche. In this way, we measured the capacity of XCS to generalize and obtain the best representative of each niche at high imbalance ratios. Moreover, figure 5.11(b) depicts the evolution of the product of TN rate and TP rate. Therefore, in addition to measuring whether XCS evolved the optimal population, this figure visualizes whether the instances of the two classes were correctly predicted by the system. Note that we tested extremely imbalanced problems with imbalance ratios up to $ir = 1024$.

Figure 5.11(a) shows that XCS was able to obtain 100% of the optimal population at any imbalance ratio with the same population size. This indicates that class imbalances did not reduce the generalization capabilities of XCS. Similarly, figure 5.11(b) illustrates that the product of TP rate and TN rate raised to 100% for all the tested imbalance ratios. Notice that before the analysis, XCS could only solve the problem for $ir \leq 32$. Hence, the lessons extracted from the design decomposition served to increase our understanding of the system and solve much more complex problems without introducing new mechanisms to the system.

The whole experimental and theoretical analysis performed herein highlights the importance of, previously to designing new approaches to enhance a system whose behavior is only partially understood, really comprehending the underlying problems of the learning architecture. In this way, better approaches that focus on the actual problems of the system can be designed more effectively. Here, we showed that the increased understanding provided by the facetwise analysis enabled first generation XCS to solve problems that seemed to be initially intractable.

5.10 Summary and Conclusions

In this chapter, we analyzed the behavior of XCS in domains that contain rare classes. As XCS learns a set of distributed solutions online, we investigated whether the system may lose, or may

never discover, some sub-solutions whose representative examples are infrequently sampled to the system. XCS learning is driven by the interaction of several components, which introduces complexity to the derivation of models that explain the behavior of the system in its whole. For this reason, we followed the design decomposition approach to study the effect of class imbalances on different components of XCS. That is, we decomposed the problem of learning from imbalanced domains into five subproblems and derived simpler, tractable models that focused on explaining the behavior of concrete parts of the system assuming that the other elements were functioning in an ideal manner. This enabled us to highlight several aspects—mainly related to classifier evaluation, and creation, maintenance and growth of representatives of starved niches—that were critical to guarantee that XCS extract key knowledge from rare classes. In addition, the patchquilt integration of all these models permitted us to draw the *domain of applicability* of XCS in imbalanced domains. We derived critical bounds on the system behavior, identifying the sweet spot where XCS could scalably and efficiently solve problems with class imbalances. Moreover, the study resulted in several recommendations on the system configuration to deal effectively with rare classes. Finally, we showed that all the insights provided by this analysis served to solve new complex problems with high imbalance ratios which previously eluded solution. As example, we empirically demonstrated that XCS was able to solve the 11-bit multiplexer problem with large degree of class imbalance provided that the system was properly configured according to the guidelines indicated by the models.

The importance of the lessons extracted from the whole analysis goes beyond the application of XCS to imbalanced domains. The analysis sets the conditions required to ensure complete solution sustenance—i.e., discovery, maintenance, and growth of all niches of the system—in problems where some niches are activated with less frequency than other niches. This is a common characteristic in real-world classification problems that contain continuous attributes, in which, although not having large imbalance ratios, there may be small regions of instances of one class for which the system needs to evolve starved niches. A deeper discussion about this aspect is postponed to chapter 7, where LCSs are applied to real-world classification problems.

Finally, let us highlight that, as we applied a design decomposition principle, the provided analysis is not restricted to XCS. In fact, we developed a framework in which several subproblems were analyzed separately and simplified models were provided. In all the derived models, we tried to keep the analysis as simple as possible and used intuitive arguments to patch the pieces together. This supplied high flexibility and power to the theoretical framework, which can be adapted with low cost to model other Michigan-style LCSs or other online learning architectures that are based on a competition-collaboration scheme. For this purpose, models that are affected by the architecture change may be revisited and plugged again into the theoretical framework. Other models may still be valid; for example, takeover time models may still be accurate for most of the Michigan-style LCSs. In the next chapter, we illustrate this, and carry over the design decomposition framework developed for XCS to UCS.

Chapter 6

Carrying over the Facetwise Analysis to UCS

In the previous chapter, we decomposed the problem of learning from imbalanced domains in *five elements* that need to be satisfied by any LCSs to efficiently and scalably extract accurate models from rare classes. Then, we centered on XCS and, with little algebra effort, we developed facetwise models for each one of these elements. Although the models were particularly designed for XCS, we claimed that the framework could be applied to other LCSs for two main reasons. The first reason is due to the simplification and abstraction effort taken when deriving the models and the use of qualitative arguments to patch the pieces together. That is, since the analysis was kept as abstract as possible, considering the general learning architecture and avoiding going into too specific details of XCS, some of the models can be applied to other Michigan-style LCSs. The second reason comes implicitly with the design decomposition methodology; that is, since each facet was analyzed considering that the others behave in an ideal manner, changes that only affect one of the facets could be incorporated by rewriting the models of the corresponding facet and plugging the new model into the general framework. This gives an important advantage of facetwise analysis with respect to global models, in which, probably, a little change in the system architecture may require to rewrite the totality of the model.

In this chapter, we take advantage of the flexibility of the framework provided in the previous chapter and show that, with little changes on some of the facets, the behavior of UCS can be easily modeled using the same ideas employed for XCS. We first recall the design decomposition with the list of elements that should be followed by any LCSs to solve class-imbalanced problems. Then, we review each subproblem for UCS. We show that some of the models developed in the previous section, such as the takeover time models, can be directly applied to UCS with no modifications. In other cases, such as the starved niches generation models, new theoretical derivations need to be done and plugged into the general framework. Therefore, we demonstrate the "*plug and play*" capabilities that design decomposition and facetwise analysis provide.

The remainder of this chapter is structured as follows. Section 6.1 reviews the design decomposition presented in the previous section, which identified five main elements or subproblems, and intuitively analyzes whether UCS can solve the five subproblems. Then, each of these five subproblems gets one of the subsequent sections, regardless of whether the original models provided in the previous chapter are still valid for UCS. Therefore, section 6.2 studies whether

UCS can obtain reliable parameter estimates in imbalanced domains, section 6.3 and section 6.4 model the initialization and generation of classifiers of the minority class, section 6.5 studies the effect of occurrence-based reproduction, and section 6.6 revisits the takeover time models. Note that most of these sections are very concise since they use the models derived in the previous chapters. All these models are integrated in section 6.7. Finally, section 6.8 summarizes and concludes the chapter.

6.1 Design Decomposition for UCS

In the previous chapter, we decomposed the problem of learning from imbalanced domains in LCSs in five elements that need to be guaranteed, i.e.,

1. Estimate the classifier parameters correctly.
2. Analyze whether representatives of starved niches can be provided in initialization.
3. Ensure the generation and growth of representatives of starved niches.
4. Adjust the GA application rate.
5. Ensure that representatives of starved niches will take over their niches.

Here, we follow the same decomposition to study the behavior of UCS in imbalanced domains. As proceeds, we first intuitively discuss the differences between XCS and UCS in these types of problems. Then, each of the subsequent sections analyzes of these elements in detail.

Estimate the classifier parameters correctly. Having accurate estimates of the classifier parameters was identified as one of the most important elements that need to be guaranteed in imbalanced domains. That is, the system relies on these estimates to distinguish between over-general and accurate classifiers; therefore, poor parameter estimates may thwart the competition between over-general and accurate classifiers. XCS used a temporal widowed average to update the classifier parameters. We showed that this may result in poor estimates if the size of the window is not set properly. In UCS, the classifier's accuracy—which is equivalent to the classifier's error in XCS—is computed as a the true average of the number of examples that have been correctly classified over the total number of examples that the classifier has matched. Intuitively, this seems to indicate that the parameters of over-general classifiers would not oscillate as abruptly as in XCS. A further study of the parameter update procedure is conducted in section 6.3.

Analyze whether representatives of starved niches can be provided in initialization. Once the proper evaluation of classifier parameters is ensured, we are concerned about whether UCS is able to provide representative schemas of starved niches in the beginning of the run. For XCS, we showed that the probability that the covering operator supplies the population with schemas of starved niches decreases exponentially with the imbalance ratio. This is because, in XCS, covering is applied to the match set, creating classifiers that can predict any class. On the other hand, UCS applies covering in the correct set as the class of the input example is also provided at each learning iteration. Thence, the covering operator in UCS only generates classifiers

that predict the class of the sampled input instance. In section 6.4, we analyze whether the new covering scheme in UCS is able to provide the initial population with schemas of the under-sampled class.

Ensure the generation and growth of representatives of starved niches. After initializing the population, the GA is responsible for obtaining high accurate classifiers that represent the different niches. As in XCS, intuition seems to indicate that the occurrence-based reproduction of UCS may favor both over-general classifiers and representatives of nourished niches, which may go in detriment of representatives of starved niches. Section 6.4 develops this aspect in detail.

Adjust the GA application rate. As in XCS, varying the application rate of the genetic algorithm influences the genetic opportunities that the different niches receive. Section 6.5 studies the impact of varying the frequency of application of the GA.

Ensure that representatives of starved niches will take over their niches. Finally, we analyze whether the best classifiers will be able to take over their niches. For this purpose, section 6.6 validates the applicability of the takeover time models to UCS and relates these models to the maximum imbalance ratio up to which the representatives of starved niches will be able to take over their niche.

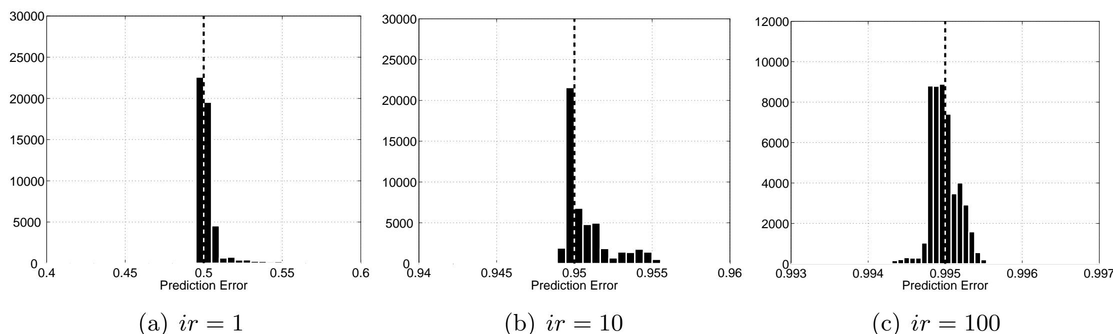
As proceeds, each of the five elements is analyzed in detail and the derived models are validated with the imbalanced parity problem. Section 6.7 unifies all the models, emphasizing the lessons extracted from all them. Finally, as done for XCS, we show that following the recommendations provided by the models, UCS is able to solve the 11-bit multiplexer problem with large imbalance ratios.

6.2 Estimation of Classifier Parameters

The first element of the design decomposition that needs to be satisfied is that the parameters of classifiers be correctly estimated. In XCS, we detected that the original parameter update procedure may provide poor estimates of the parameters of over-general classifiers in problems with large imbalance ratios. However, note the difference between the parameter update procedure in both systems. XCS computes the quality of a rule by means of a fitness based on the error of the prediction of the rule. This error is updated online by a credit apportionment algorithm that performs a windowed average of the last received rewards. Conversely, as UCS is specialized for supervised learning tasks, the fitness of a classifier is based on its classification accuracy, which is computed as the true average of the number of examples correctly classified with respect to the total number of examples matched by the classifier. Then, the fitness is computed from the relative accuracy of each classifier[1]. Therefore, the larger the number of examples matched by the rule, the more accurate the estimation of the classifier's accuracy, and, consequently, the fitness estimate.

To illustrate the differences between the parameter update procedures of XCS and UCS, we ran the same experimentation proposed in section 5.5 but with UCS. Figure 6.1 shows a

[1]In all the experiments conducted along the subsequent chapters, we use UCS with fitness sharing.



(a) $ir = 1$ (b) $ir = 10$ (c) $ir = 100$

Figure 6.1: Histogram of the error of the most over-general classifier in UCS for $ir = \{1, 10, 100\}$.

histogram of the accuracy estimate of the most over-general classifier along a complete run. The vertical line depicts the theoretical value of the accuracy. As expected, the empirical accuracy estimate of the most over-general classifier matches perfectly with the theoretical one most of the time. Note the difference of these results with respect to those provided by XCS. In XCS, the parameters of over-general classifiers oscillated as they received infrequently negative rewards. In UCS, the parameters of over-general classifiers are stabilized as the classifier receives more updates.

In summary, in this section we discussed and empirically showed that the parameters update procedure of UCS enables the system to obtain reliable estimates. Thence, the remainder of the analysis is conducted assuming accurate parameter estimates.

6.3 Supply of Schemas of Starved Niches in Population Initialization

In this section, we analyze the first element that leads to the creation of representatives of the different niches: population initialization. In the previous chapter, we identified that the covering mechanism of XCS impaired the system from initializing the population with correct schemas of starved niches. This was mainly due to the exploration regime of XCS, which, given an unlabeled input example, analyzed the consequences of each possible class. Therefore, we assumed a covering failure in XCS, and derived the models for the remaining elements considering this covering failure.

Differently from XCS, the supervised learning architecture of UCS only applies covering on the correct set. That is, covering only creates classifiers that predict the class of the sampled example. Therefore, covering will be triggered on the first instances of the each one of the classes, including the rare class, regardless of the imbalance ratio of the learning data set. Consequently, we can calculate the probability that a minority class instance is covered by, at least, one classifier

in the population as

$$P(\text{cover}) = 1 - \left[1 - \left(\frac{2-\sigma[P]}{2}\right)^{\ell}\right]^{\frac{N}{ir}}, \tag{6.1}$$

where ℓ is the input length, N is the population size, and $\sigma[P]$ is the specificity of the population. Note that the only difference with respect to the corresponding equation in XCS (see equation 5.11) is that, in UCS, the power of the term in brackets of the right-most expression is N/ir instead of N. This modification is because the number of minority class classifiers provided by covering is directly proportional to the number of instances of the minority class that have been sampled to the system.

Provided that the probability of activating covering is $1 - P(cover)$, and recognizing that $(1 - r/n)^n \approx e^{-r}$, we can derive that the probability of activating covering, having sampled a minority class instance, is

$$P(\text{activate cov. on. min.}) = 1 - P(\text{cover}) \approx e^{-\frac{N}{ir} \cdot e^{-\frac{\ell\sigma[P]}{2}}}, \tag{6.2}$$

which decreases exponentially with the ratio of the population size to the imbalance ratio N/ir and, in a higher degree, with the condition length and the initial specificity. Notice that N/ir decreases linearly with the sampling frequency of the minority class. Therefore, the capabilities of covering to provide accurate schemas of the minority class do not depend directly on the imbalance ratio, but on the initial population specificity.

The analysis performed in this section showed that, differently from XCS, the success of the covering operator in supplying classifiers representing correct schemas of the minority class does not depend on the imbalance ratio. Although these positive results, in the next section we derive the models for creation of new representative classifiers of the minority class under the assumption that the population has not any representative of starved niches, as done for XCS. As we are assuming the most pessimistic situation, we expect that the models predict an upper bound on the time and the population size required by UCS to solve the problem. Note that, although this would not result in a precise model of the population size, it will bound the maximum population size required to solve problems with large imbalance ratios, which still provides critical information about UCS behavior in imbalanced domains.

6.4 Generation of Classifiers in Starved Niches

In this section, we study the conditions that must be satisfied to enable the GA to create representatives of starved niches. Therefore, as done for XCS, we derive models that predict the time until creation and extinction of these representatives. With these models, we write population size bounds to warrant the existence and growth of representatives of starved niches. As proceeds, we first review the assumptions of the model, which are equivalent to those considered for XCS, and then revisit the models for each element.

6.4.1 Assumptions for the Model

The models are developed under the same assumptions considered for XCS, i.e., (i) covering has not provided any representative of starved niches, (ii) mutation is the only operator that guides the genetic search (i.e., we do not consider crossover), (iii) the GA is applied at the end of each learning iteration (i.e., $\theta_{GA} = 0$), and (iv) the system uses random deletion. Note that the first assumption may not be necessarily true in UCS; in fact, the previous section demonstrated that the covering operator could provide the same number of schemas of the minority class regardless of the imbalance ratio. Nonetheless, we consider this assumption and derive an upper bound of the convergence and population size models. After this, we experimentally study the effect of breaking the three last assumptions.

6.4.2 Creation and Deletion of Representatives of Starved Niches

With the assumptions provided above, we are now in position to derive both the time until creation and the time until deletion of representatives of starved niches. For this purpose, we first calculate the probability to obtain the first accurate representative cl_{min} of the starved niche i, which is represented by a schema with length k_m. This probability will be used to compute the creation time.

As proposed in section 5.6.2, we calculate the probabilities of creating cl_{min} when sampling (i) instances of the minority class and (ii) instances of the majority class. Recognizing that the probability of sampling a minority class instance is $1/(1+ir)$ and the probability of sampling a majority class instance is $ir/(1+ir)$, we can write that

$$P(cl_{min}) = \frac{1}{1+ir} P(cl_{min}|\text{min. inst}) + \frac{ir}{1+ir} P(cl_{min}|\text{maj. inst}). \tag{6.3}$$

Let us first derive $P(cl_{min}|\text{min. inst})$. When an instance of the minority class is sampled, a niche containing classifiers that predict the minority class will be activated. As we assumed that there are no representatives of starved niches in the population, the correct set will only consist of over-general classifiers. Thence, to create a representative of a starved niche, all the k_m bits of the schema that represents the niche must be correctly set to the values of the niche schema; here, we consider the worst case, and assume that all the k_m bits need to be mutated. Thence, the probability of getting the correct schema is $(\frac{\mu}{2})^{k_m}$. Besides, the class of the classifier cannot be changed. Therefore,

$$P(cl_{min}|\text{min. inst}) = \left(\frac{\mu}{2}\right)^{k_m} \cdot (1-\mu). \tag{6.4}$$

We follow the same procedure to derive the probability of creating cl_{min} when sampling an instance of the majority class, i.e., $P(cl_{min}|\text{maj. inst})$. When a majority class instance is sampled, a nourished niche containing both representatives and over-general classifiers will be activated. Again, we consider the worst case and assume that we need to change all the k_m bits of the schema, i.e., $(\frac{\mu}{2})^{k_m}$. Furthermore, in this case, the class has to be mutated to the minority class, which will happen with probability $\mu/n-1$, where n is the number of classes. Thence,

$$P(cl_{min}|\text{maj. inst}) = \left(\frac{\mu}{2}\right)^{k_m} \cdot \frac{\mu}{n-1}. \tag{6.5}$$

Substituting equations 6.4 and 6.5 into equation 6.3, we obtain that

$$P(cl_{min}) = \frac{1}{1+ir}\left(\frac{\mu}{2}\right)^{k_m}\left[(1-\mu)+\frac{\mu \cdot ir}{n-1}\right]. \tag{6.6}$$

From this formula, we can derive the time required to discover the first representatives of starved niches $t_{cl_{min}}$ as

$$t(cl_{min}) = (n-1)\left(\frac{2}{\mu}\right)^{k_m}\left[\frac{1+ir}{(1-\mu)(n-1)+\mu \cdot ir}\right], \tag{6.7}$$

which depends on the imbalance ratio ir, the probability of mutation μ, and the length of the schema k_m. For highly imbalanced domains, we can consider that $(1-\mu)(n-1) << \mu \cdot ir$. Thence, $t(cl_{min})$ increases linearly with $\frac{1+ir}{\mu \cdot ir}$, which becomes nearly constant for large values of ir.

After computing the creation time, we now approximate the deletion time of these representatives. As done for XCS, we consider random deletion. Hence, as two classifiers are deleted at each GA application, we obtain that the time until deletion is

$$t(\text{delete cl}) = \frac{N}{2}, \tag{6.8}$$

where N is the population size. In the next section, we use both equations 6.7 and 6.8 to derive population size bounds that guarantee the discovery, maintenance, and growth of representatives of starved niches under the assumption that covering has not provided any of them.

6.4.3 Bounding the Population Size

We now follow the same steps proposed in section 5.6.4 to derive two population size bounds that ensure (i) that XCS will be able to maintain accurate representatives of starved niches and (ii) that these representatives will receive, at least, a genetic opportunity.

The first bound can be derived by requiring that the deletion time of starved niches representatives be larger than their creation time, i.e.,

$$t(\text{delete } cl_{min}) > t(cl_{min}). \tag{6.9}$$

Using formulas 6.7 and 6.8, the expression can be rewritten as

$$N > 2(n-1)\left(\frac{2}{\mu}\right)^{k_m}\left[\frac{1+ir}{(1-\mu)(n-1)+\mu \cdot ir}\right]. \tag{6.10}$$

Again, note that, for large values of ir, the population size increase is guided by the term $\frac{1+ir}{\mu \cdot ir}$, which remains nearly constant. Consequently, at a certain imbalance ratio, the population size needed to discover representatives of the minority class becomes constant.

We now derive the second bound by requiring that the deletion time of representatives of starved niches be greater than the time until these representatives receive a genetic event, that is,

$$t(\text{delete } niche_{min}) > t(\text{GA } niche_{min}). \tag{6.11}$$

As we assumed that $\theta_{GA} = 0$, a starved niche receives a genetic event every time that it is activated. Since the probability of sampling a minority class instance is 1/(1+ir), the time to apply a GA on a starved niche is

$$t(\text{GA } niche_{min}) = (1 + ir). \quad (6.12)$$

Replacing equations 6.8 and 6.12 into equation 6.11, we obtain that

$$N > 2(1 + ir), \quad (6.13)$$

which indicates that the population size has to increase linearly with the imbalance ratio to ensure that representatives of starved niches will receive, at least, a genetic opportunity.

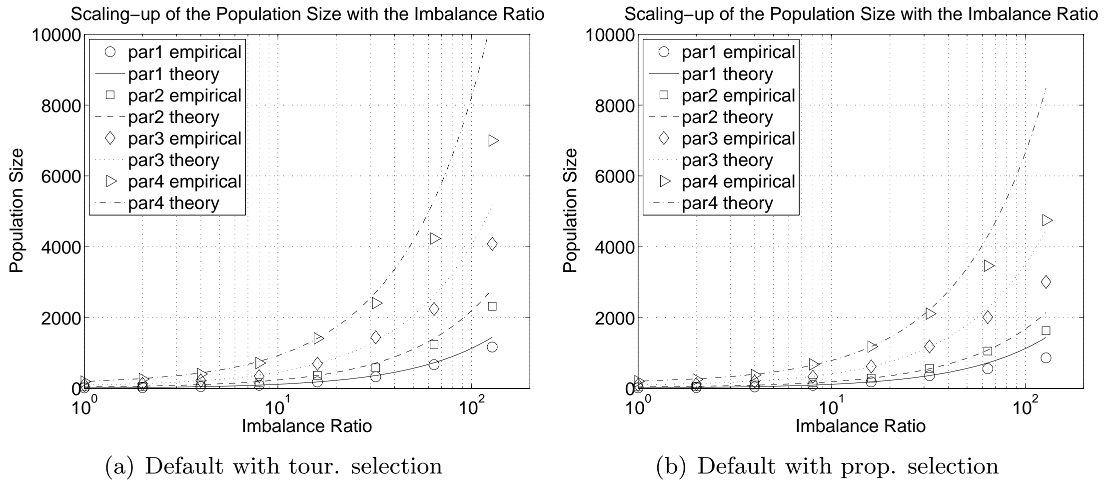
In this section, we derived models that explain the creation and growth of representatives in starved niches. As the models were developed under the assumption of a failure of the covering operator to provide schemas of starved niches—which it is not necessarily the case, as argued in section 6.3—, the models represent an upper bound of the population size required by UCS to solve imbalanced problems. Therefore, the models explain that the population size has to increase, at most, linearly with the imbalance ratio to ensure the discovery and growth of accurate classifiers in starved niches. In the next section, we empirically validate the population size bounds with different configurations of the imbalanced parity problem.

6.4.4 Experimental Validation of the Models

To validate the population size models, we first use a configuration of UCS that satisfies the assumption of the models. Then, we empirically analyze the effect of breaking these assumptions.

Experiments Satisfying the Assumptions

Our first concern is to empirically contrast whether the population size bound derived in equation 6.13 predicts an upper-bound of the population size as the imbalance ratio increases. For this purpose, we performed the same experiments as for XCS (see section 5.6.5). We ran UCS on the imbalanced parity problem with $k = \{1, 2, 3, 4\}$, $\ell = 10$, and $ir = \{1, 2, 4, 8, 16, 32, 64, 128\}$, and we used the bisection procedure to obtain the minimum population size required to solve the problem (see section 5.6.5 for more details about the procedure). The results are averages over 50 runs with different random seeds. UCS was configured so that the initial assumptions were satisfied. Thence, crossover was deactivated ($\chi = 0$), random deletion was used, and the GA was applied every time a niche was activated (θ_{GA}=0). The other parameters were set as $acc_0 = 0.999$, $\nu = 10$, $\mu = 0.04$, $\theta_{del} = 20$, $\delta = 0.1$, $\theta_{sub} = ir$, $P_{\#} = 0.6$, $\beta = 0.2$. We used both proportionate and tournament selection for the GA. We ran UCS during $\{10\,000 \cdot ir, 20\,000 \cdot ir, 40\,000 \cdot ir, 80\,000 \cdot ir\}$ iterations for the parity problem with $k = \{1, 2, 3, 4\}$ respectively; thus, given a problem, we ensured that the system received the same number of genetic opportunities for all imbalance ratios. Finally, to prevent having young over-general classifiers with poorly estimated parameters in the final population, we introduced $5\,000 \cdot ir$ iterations with the GA switched off at the end of the learning process. In the remainder of this analysis, this configuration is referred to as the default configuration.



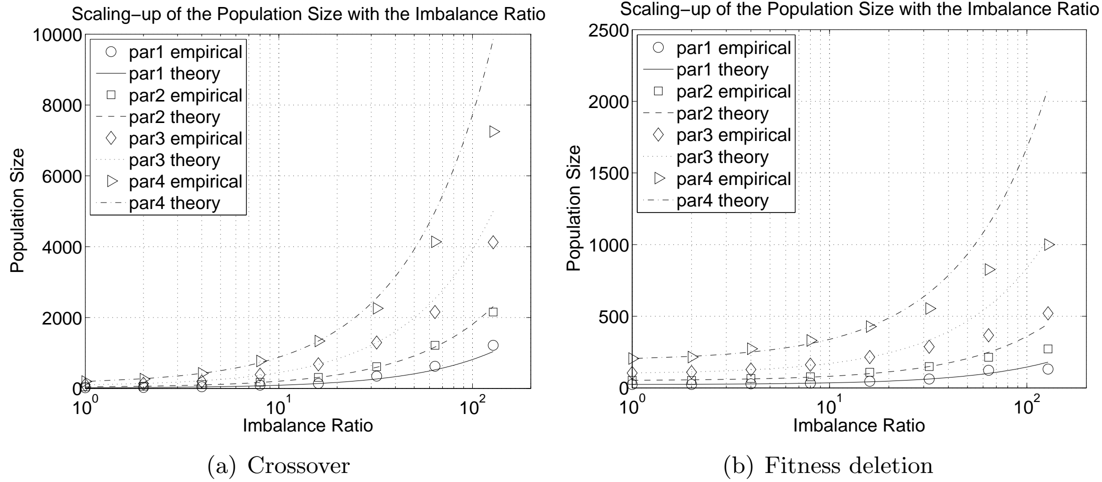
(a) Default with tour. selection

(b) Default with prop. selection

Figure 6.2: Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and the default configuration with (a) tournament selection and (b) roulette wheel selection The dots shows the empirical results and lines plot linear increases with ir (according to the theory).

Figure 6.2 shows the minimum population size required to solve the parity problem with different building block sizes ($k = \{1, 2, 3, 4\}$) and imbalance ratios from $ir = 1$ to $ir = 128$ for (a) tournament and (b) proportionate selection. For each plot, the points depict the empirical values and the lines show the theoretical bound, that is, they draw a linear increase with the imbalance ratio. Two main conclusions can be extracted from these results. Firstly, note that the theory estimates an upper bound of the population size required by UCS to solve the problem, especially as the imbalance ratio increases. This behavior was already announced in the beginning of this section. The theory was developed with the assumption that the covering operator was not able to provide accurate schemas of starved niches. Nonetheless, section 6.3 showed that the initial supply of schemas of starved niches was independent of the imbalance ratio. For this reason, the theory is approximating an upper bound of the required population size. Secondly, UCS with proportionate selection requires smaller population sizes to solve the parity problems than UCS with tournament selection, especially as the imbalance ratio increases. The Wilcoxon signed-ranks test confirmed that this difference was significant at $\alpha = 0.05$. This may be due to the fact that proportionate selection can produce a stronger pressure toward fit classifier than tournament selection in this particular problem. Nevertheless, we leave further discussion about the selection schemes to section 6.6.

In summary, the experimental analysis pointed out that the theory is an accurate upper bound of the population size of UCS for imbalanced domains when the system is configured so that the underlying assumptions of the model are met. In the next section, we investigate whether this population size bound is still valid when the different assumptions are not satisfied.



(a) Crossover (b) Fitness deletion

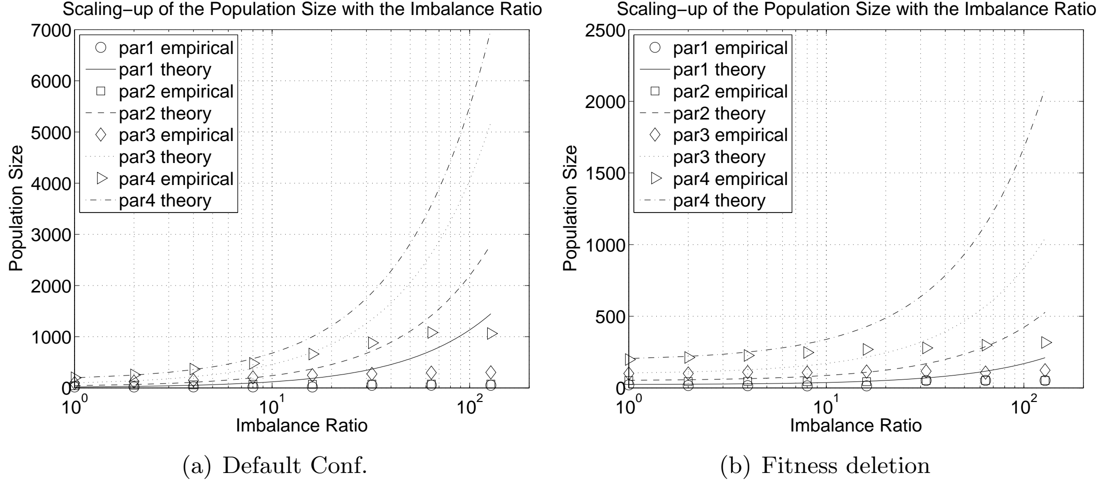
Figure 6.3: Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and different UCS's configurations that do not satisfy the initial model assumptions: (a) using 2-point crossover and (b) using the correct set size deletion scheme. The dots shows the empirical results and lines plot linear increases with ir (according to the theory).

Impact of Breaking the Assumptions

In this section, we experimentally analyze the impact of breaking two of the initial assumptions. That is, we introduce crossover and the typical deletion scheme of UCS. The impact of breaking the third assumption, i.e., varying the frequency of application of the GA, is further studied in the next section. Figure 6.3 provides the results of running UCS on the same configuration of the parity problem used in the previous subsection, but using (a) two-point crossover, with $\chi = 0.8$ and (b) the typical UCS's deletion scheme, setting $\theta_{del} = 20$ and $\delta = 0.1$. In both cases we used tournament selection.

Several conclusions can be drawn from the comparison of these results with those obtained in the previous section. Firstly, notice that, in both cases, the theory still predicts an upper bound of the minimum population size required to solve the problem, although the initial assumptions are not satisfied. The population size required by UCS when using crossover (see figure 6.3(a)) is equivalent to the population size needed by UCS with the default configuration (see figure 6.2(a)) according to a Wilcoxon signed-ranks test at $\alpha = 0.05$. This indicates not only that the models are still valid when using crossover, but also that the population sizes demanded solving the different configurations of the problem are statistically equivalent to the ones required by UCS without crossover. On the other hand, the population sizes needed for UCS with the usual deletion scheme are statistically smaller than those required by UCS with random deletion, since the deletion scheme protects classifiers that belong to starved niches.

The overall study provided along this section showed that the theory approximates the experiments accurately, even though two of the initial assumptions of the model are not satisfied. In the next section, we investigate the effect of breaking the last assumption; that is, we analyze the effect of varying the frequency of application of the GA.



(a) Default Conf.

(b) Fitness deletion

Figure 6.4: Scalability of the population size with the imbalance ratio in the k-parity problem with k={1,2,3,4} and different UCS's configurations with $\theta_{GA} = n \cdot m \cdot ir$. The points indicate the empirical values of the minimum population size required by UCS. The lines depict the theoretical increase calculated with the previous models, which assumed $\theta_{GA} = 0$.

6.5 Occurrence-based Reproduction

The models developed so far have assumed that any niche received a genetic event every time that it was activated, i.e., $\theta_{GA} = 0$. Due to this occurrence-based reproduction, nourished niches receive a larger number of genetic events than starved niches. Besides, the reproductive opportunities of over-general classifiers with respect to the reproductive opportunities of representatives of starved niches increase linearly with the imbalance ratio. To counterbalance this effect in XCS, section 5.7 showed that if θ_{GA} was set according to the imbalance ratio, i.e.,

$$\theta_{GA} \approx n \cdot m \cdot (1 + ir), \tag{6.14}$$

both starved and nourished niches would receive, approximately, the same number of genetic opportunities. The models developed for XCS are still applicable to UCS, since they are based on the occurrence-based reproduction, which is shared in both systems. That is, the key difference between the exploration methodology of both systems is that XCS explores the consequences of all possible actions, while UCS only explores the class of the input instance. Nevertheless, in both cases, niches are only activated when an instance that is matched by the niche schema is sampled. Therefore, in both LCSs, niches that represent instances of the minority class are activated with a lower frequency that niches that represent instances of the majority class. Due to this similarity, we use the models developed for XCS to explain the occurrence-based reproduction in UCS.

To validate that the conclusions extracted from XCS models are still valid for UCS, we ran the same experiments with the parity problem proposed in section 5.7. That is, we ran UCS on the parity problem with $\ell = 10$, $k = \{1, 2, 3, 4\}$, and $ir = \{1, 2, 4, 8, 16, 32, 64, 128\}$. Figure 6.4(a) shows the minimum population with which UCS with the default configuration could

solve the parity problem. In this picture, the empirical values are plotted with points. To analyze the differences introduced by adjusting θ_{GA} according to the theory, the lines depict the population size increase predicted by the theoretical model calculated for the same configurations but with $\theta_{GA} = 0$ (see figure 6.2(a)). As happened with XCS, the empirical results obtained with UCS show that the population size remained nearly constant for all the imbalance ratios. There was only a slightly small increase for $ir > 32$. Figure 6.4(b) provides the same results for UCS with the typical deletion scheme instead of random deletion. The typical deletion scheme protects young classifiers and produces more pressure toward deletion of over-general, inaccurate classifiers. The experimental results show that, with the enhanced deletion scheme, the population size remained constant as the imbalance ratio increased, even for the largest imbalance ratios.

6.6 Takeover Time of Accurate Classifiers in Starved Niches

With the theory and experiments provided so far, we have shown that UCS is able to create representatives of starved niches, and that starved niches receive, at least, a genetic event before removing their representatives. Then, the last element that has to be analyzed according to our design decomposition is whether the best representatives of the different niches will be able to take over their niche when they are in competition with over-general classifiers. Therefore, we model the competition between accurate classifiers and over-general classifiers, especially focusing on the problems caused by rare classes. That is, in imbalance domains, the accuracy of over-general classifiers predicting the majority class may be high since the minority class is under-sampled. This, combined with the occurrence-based reproduction, may promote the existence of over-general classifiers in the population in detriment of accurate representatives of starved niches. In this context, the purpose of this section is two-fold: (i) develop models that predict the takeover time of the best representative of a niche for UCS and (ii) derive the conditions under which the best representative of a starved niche will not be able to take over its niche.

Instead of developing new theory, in this section we consider the same the takeover time models that were derived for XCS in section 5.8. That is, in the takeover time models developed for XCS, we considered a system that evolved a distributed set of niches where each niche contained a representative that was maximally accurate, which we addressed as cl_b; besides, there was an over-general classifier that matched all niches, which was referred to as cl_o. The quality of these classifiers was denoted by a the accuracy parameter k associated with each classifier, i.e., κ_b and κ_o. Notice that UCS exactly follows the same schema. Therefore, the takeover time models can be directly applied to UCS. The only difference between XCS and UCS is that the accuracy of each classifier is computed differently. For this reason, the conditions under which the best classifier will not be able to take over its niche may vary in both systems. The next section computes these conditions.

6.6.1 Conditions for Starved Niches Extinction under Proportionate Selection

To compute the conditions of niche extinction under proportionate selection, we depart from equation 5.47, that is,

$$P_t < \frac{\rho}{m\rho - 1}, \tag{6.15}$$

to derive under which conditions the best classifier will not take over its niche. m is the number of niches and ρ is the ratio of the accuracy of the over-general classifier to the accuracy of the best representative of the niche, i.e., $\rho = \frac{\kappa_o}{\kappa_b}$. As demonstrated in section 5.47, this inequality only holds for $m > 2$ and

$$1/m < \rho \leq 1. \tag{6.16}$$

In UCS, κ is computed from the raw accuracy acc of the classifier. In imbalanced domains, the most over-general classifier that predicts the majority class will receive ir examples of the majority class for each example of the minority class. Therefore, the raw accuracy of the most over-general classifier is

$$acc_o = \frac{ir}{1 + ir}, \tag{6.17}$$

where $acc_o < acc_0$; that is, the accuracy of the over-general classifier is less than the threshold beyond which UCS considers that a classifier is accurate. The accuracy of the best representative is $acc_b = 1$. From this, we can compute κ_b and κ_o using equation 4.2 of the fitness-sharing scheme as

$$\kappa_o = \alpha \left(\frac{acc_0}{acc_o} \right)^{\nu}, \tag{6.18}$$

and

$$\kappa_b = 1. \tag{6.19}$$

Replacing equations 6.17, 6.18 and 6.19 into equation 6.16 we obtain that, for proportionate selection, the best classifier will not be able to take over its niche if

$$\frac{1}{m} < \alpha \left(\frac{acc_0 \cdot (ir + 1)}{ir} \right)^{\nu} < 1 \tag{6.20}$$

where $\frac{ir}{1+ir} < acc_0$; besides, provided that $0 < \alpha < 1$ (usually 0.1) and that $\nu \geq 1$, the right-most inequality is typically satisfied. The left-most inequality can be expressed as

$$\frac{1 + ir}{ir} > \frac{1}{acc_0} \left(\frac{1}{m\alpha} \right)^{\frac{1}{\nu}}. \tag{6.21}$$

Recognizing that the left-most term is the inverse of the raw accuracy of the over-general classifier cl_o, we can derive that the best classifier will not be able to take over its niche if

$$acc_o < acc_0 (\alpha m)^{\frac{1}{\nu}}. \tag{6.22}$$

Note that the right-most expression depends on the threshold acc_0, but especially in α and the number of niches m. As the number of niches depends on the problem, the user can tune the imbalance acceptance of UCS by adjusting the α parameter. That is, lower values of α produce a decrease in the right-most expression. Given the accuracy of the most over-general classifier, which depends directly on ir, we can tune α so that the equation is not satisfied, and thus, the best classifier takes over its niche.

6.6.2 Conditions for Starved Niches Extinction under Tournament Selection

We now perform the same analysis for tournament selection. Tournament selection randomly chooses a set of classifiers from the population and selects the one with highest fitness. As the fitness of the best classifier is greater than the fitness of the over-general classifier, if the best classifier participates in a tournament, it will be selected. For proportionate selection, the condition for the extinction of starved niches depended on ρ; for this reason, the condition varied from the one computed for XCS. For tournament selection, the condition for the extinction of starved niches only depends on the selection pressure s, the number of niches m, the size of the niche n, the size of the population N, and the number of the representatives in the niche n_b. For this reason, the same condition derived for XCS is still valid for UCS. Thus, the best representative will not be able to take over its niche if

$$1 - m\frac{n_{b,t}}{N} < \left(1 - \frac{n_{b,t}}{n}\right)^s. \tag{6.23}$$

where s is the tournament size, N is the population size, m is the number of niches, n is the number of classifiers in the niche, and $n_{b,t}$ is the numerosity of the best classifier.

In this section, we argued why the takeover time models derived for XCS are still valid for UCS, and have used the takeover time equations to develop the conditions for the extinction of starved niches. In the following section, we put all the pieces together and provide recommendations for UCS configuration in imbalanced domains. Finally, we show that, following the guidelines, UCS, as XCS, is able to solve the 11-bit multiplexer problem with large imbalance ratios.

6.7 Reassembling the Theoretical Framework: UCS in Imbalanced Domains

With the different models and qualitative arguments provided along this chapter, this section follows the same steps as done for XCS to unify all the different models, analyze the interaction among them, and give guidelines on how the system should be configured to guarantee the discovery of the minority class. Then, we show that applying the lessons learned from the theoretical study enables UCS to solve problems with large imbalance ratios.

6.7.1 Patchquilt Integration: from XCS to UCS

In this section, we consider the theoretical framework derived for XCS and study how the new models of UCS can be plugged into the framework. We review the models in the same order

as proposed in section 5.9, that is, from the most restrictive one to the less restrictive one, and compare the differences with respect to the models derived from XCS.

1. The takeover time models set the maximum imbalance ratio beyond which UCS will not be able to discover the minority class. The takeover time models derived for XCS are still valid, and the specific conditions under which representatives of starved niches will be deleted from the population for proportionate and tournament selection have been calculated in equations 6.22 and 6.23 respectively. Satisfying the requirements identified by these models is a necessary but not sufficient condition.

2. The parameter update procedure in UCS is fairly robust, providing accurate approximations of the real values of over-general parameters. Thence, differently from XCS, in UCS it is not necessary to tune the parameter update procedure according to the imbalance ratio.

3. Once the takeover time requirements are met, we have to ensure that accurate representatives of starved niches will be feeded into the population. For this purpose, we can either (i) increase the population size linearly with ir–at maximum—or (ii) set θ_{GA} according to ir. Note that the main difference with respect to XCS is that, in UCS, the population size model provides an upper bound instead of predicting the actual increase.

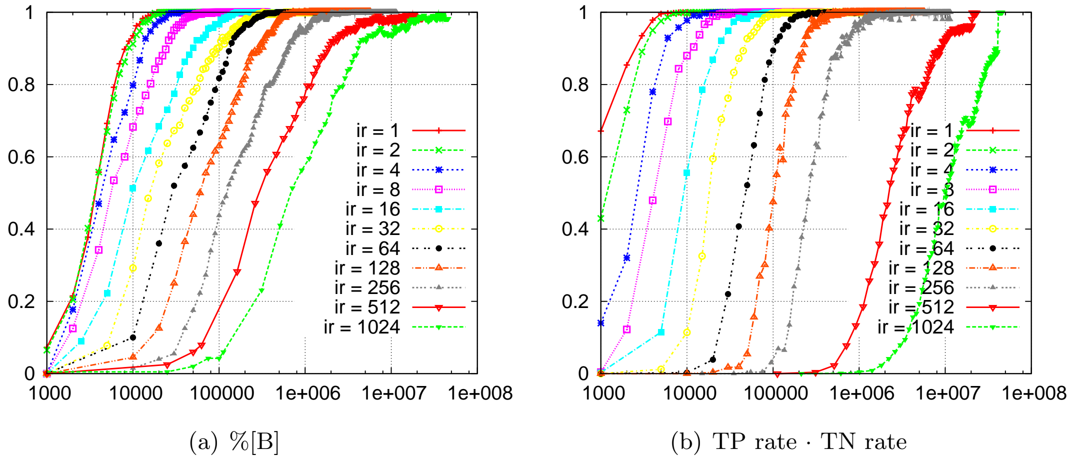
UCS appears to be slightly more robust to class imbalances than XCS since the parameter update procedure is not as sensitive as the XCS's one and, as experimentally shown, the population size increases slightly slower than XCS's one. In the next section, we show that, if the recommendations derived from the models are followed, UCS can solve extremely imbalanced data sets.

6.7.2 Solving Highly Imbalanced Domains with UCS

Having revised the information provided by the different methods and established the framework of UCS's learning from class imbalances, we use this information to tune UCS so that it can solve highly imbalanced domains. For this purpose, we ran UCS on the imbalance 11-bit multiplexer problem with $ir = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$ (see appendix A.4.1 for a description of the problem). We used the default configuration provided in this section with proportionate selection and the following exceptions: (i) crossover was activated with $\chi = 0.8$, (ii) the typical deletion scheme of UCS was employed and (iii) θ_{GA} was set to $n \cdot m \cdot ir$. We set a population size of N=1,000, and we used tournament selection. Note that, with this configuration, the requirements of the three items enumerated in the previous section are satisfied:

1. As we used tournament selection, we need to satisfy the condition of equation 6.23. In the previous chapter, we already showed that the proposed configuration satisfied this condition (notice that the condition imposed by tournament selection is equal in both XCS and UCS).

2. Parameters are correctly estimated by the update procedure, especially as the experience of the classifier increases.



(a) %[B]

(b) TP rate · TN rate

Figure 6.5: Evolution of (a) the proportion of the optimal population and (b) the geometric mean of TP rate and TN rate in the 11-bit multiplexer with ir={1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}.

3. As we set $\theta_{GA} = n \cdot m \cdot ir$, we ensure that starved and nourished niches will have, approximately, the same number of genetic opportunities. Since we take this approach, we maintain the same population size for all the runs.

Figure 6.5 plots (a) the evolution of the proportion of the optimal population size (%[O]) achieved by XCS and (b) the evolution of the geometric mean of TN rate and TP rate in the 11-bit imbalanced multiplexer problems from $ir = 1$ to $ir = 1024$. Note that, for $ir = 1024$, the system only received an instance of the minority class every each 1024 instances of the majority class. The results show that, even under these large imbalance ratios, UCS is able to extract accurate knowledge from the under-sampled class. Figure 6.5(a) shows that UCS is able to obtain 100% of the optimal population for all the runs. Besides, figure 6.5(b) indicates that the system achieves 100% performance measured as the geometric mean of TP rate and TN rate. Notice that, for $ir = 1024$, the performance reaches 100% after activating the condensation runs, which explains that the performance curve increases abruptly from 94% to 100% in the last few iterations. This is because the frequent activation of nourished niches and over-general classifiers of the majority class leads to the creation of some over-general classifiers of the majority class with poorly estimated parameters. As these over-general classifiers match a negative example very infrequently, the system needs several learning iterations to adjust their parameters, realize that they are not accuracy, and remove them from the system. When condensation is activated, as crossover and mutation are deactivated, these classifiers are correctly evaluated and removed from the population. In any case, these results evidence that UCS is able to classify correctly all the input instances, regardless of whether they belong to the minority class or not.

6.8 Summary and Conclusions

In this section, we carried over the facetwise analysis of XCS to UCS. Following the design decomposition provided for LCSs in general, we examined the behavior of UCS in imbalanced domains. Similar conclusions than those extracted for XCS were reached for UCS. That is, the models showed that, to ensure the growth and takeover of representatives of starved niches, either (1) the population size needs to increase linearly with the imbalance ratio or (2) the frequency of application of the GA has to decrease linearly with the imbalance ratio. Besides, two key differences with respect to XCS were found. Firstly, the parameter update procedure of UCS was shown to provide accurate estimates of classifier parameters without requiring any especial configuration. Secondly, theory indicated that the covering operator is able to supply the initial population with schemas of the minority class regardless of the imbalance ratio. Consequently, the population size bounds derived subsequently predict an upper bound, instead of an exact bound, of the scalability of the population size with the imbalance ratio.

Finally, let us point out two important conclusions. The first conclusion is related to the analysis methodology, that is, the design decomposition and facetwise analysis principle. Note that, at the beginning of the previous chapter, we decomposed the complex problem of learning from imbalanced domains in five critical *elements* that need to be satisfied to efficiently deal with rare classes. Then, for each one of the elements, we developed low cost models that explained the corresponding facet, assuming that the others behave in an ideal manner. This approach has two key advantages with respect to creating complex models that try to capture all the interactions in the components of the whole system. The first advantage is that the algebra effort is reduced with respect to that required in global models since each element is analyzed separately, and the interactions with other elements are not considered. At first glance, one may think that this approach also results in models that can explain less than global models which include complex interactions among different elements. Nonetheless, as shown along the two previous chapters, this may not be the case. That is, facetwise models permit focusing on the actual problems of each element, some of which could be hidden in more complex models. Then, the patchquilt integration enables to draw a domain of competence of the systems, indicating the sweet spot in which the system actually scales. The second advantage is that design decomposition enables us to easily transport models from one system to another. In the present chapter, we used parts of the theory developed for XCS, merged this theory together with new models particularly developed for UCS, and put the pieces together, obtaining a framework that explains how UCS behaves in imbalanced domains.

The second conclusion is about the excellence of UCS—and XCS as well—in imbalanced domains. The experiments provided in this section culminated the whole study of the behavior of both LCSs in domains that contain rare, under-sampled classes. In summary, we showed that, as XCS, UCS is a competitive machine learning technique able to deal with large imbalance ratios. We showed this competitiveness in a set of artificial problems which were defined with binary attributes. In the next section, we move to real-world problems which contain continuous values. We will discuss how the theory adapts to these cases and will test both LCSs on a collection of real-world imbalanced classification problems.

Chapter 7

XCS and UCS for Mining Imbalanced Real-World Problems

In the previous two chapters, we have carefully analyzed the behavior of XCS and UCS in domains that contain class imbalances. We decomposed the problem of learning from imbalanced domains in several elements or subproblems and derived facetwise models for each element. This resulted in a better understanding of how the two LCSs work and in the definition of several guidelines or recommendations that need to be satisfied to warrant that the two LCSs are able to learn from rare classes. All these models and recommendations depended on the imbalance ratio ir. Throughout all the theory development, we assumed that the imbalance ratio of the training data set was equivalent to the ratio of the frequency of activation of nourished to the one of starved niches. The artificial problems used to contrast the models met this assumption. Nevertheless, the number of niches and their frequency of activation is not known in real-world problems. Therefore, there is a gap between the theory and its application to effectively solve real-world problems.

The purpose of this chapter is three fold. Firstly, we aim at connecting the dots between theory and application in imbalanced real-world domains. We study in more detail the structure of real-world problems and provide some heuristic procedures, which are based on the information gathered during the online evolution of the two LCSs, to estimate the imbalance ratio between niches; this estimate is used to self-adapt the parameters of the two LCSs according to the recommendations derived from the theory. We show the effectiveness of these procedures in the imbalanced 11-bit multiplexer problem. The second objective is to confirm that both LCSs are really valuable machine learning techniques for supervised learning, and especially, for extracting classification models from imbalanced domains. For this purpose, we compare the performance of XCS and UCS with the one achieved by three of the most influential machine learning techniques (Wu et al., 2007). The third objective is to incorporate re-sampling methods into the comparison, since these types of techniques have been identified—and widely used in the machine learning community—as one of the best alternatives to improve the accuracy of different learning methods in imbalanced domains. For this reason, we include some of the most-used re-sampling techniques into the comparison and empirically analyze how the different learners are influenced by these re-sampling techniques.

The remainder of this chapter is structured as follows. Section 7.1 points out new character-

istics that can be found in real-world problems and how the theory can be adapted to these new characteristics. With the new identified challenges, section 7.2 proposes a heuristic method to self-adapt the configuration parameters of XCS and UCS that are sensible to class imbalances and shows that, with this heuristic procedure, both XCS and UCS can solve the imbalanced 11-bit multiplexer problem although they are not properly configured in the beginning of the run. Section 7.3 compares XCS and UCS with three highly-competent learners, showing the competitiveness of the two LCSs. The study of learning from imbalanced domains is complemented with the introduction of re-sampling techniques. That is, section 7.4 presents some of the most-used re-sampling techniques and illustrates how they work in a case study. These techniques are introduced in the comparison of the five learners in section 7.5. Section 7.6 discusses the overall results and points out some future work lines. Finally, section 7.7 summarizes and concludes this chapter.

7.1 LCSs in Imbalanced Real-World Problems: What Makes the Difference?

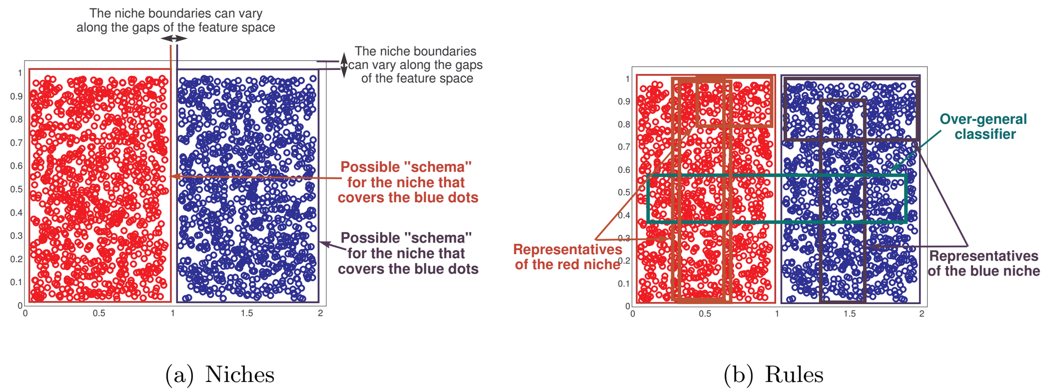
In this section, we study how the theory—which has been validated with artificial problems with known characteristics— can be applied to imbalanced real-world problems whose characteristics are unknown and can barely be estimated. As proceeds, we deal with two aspects that need to be solved to adapt the theory to real-world problems. Firstly, as a reminder of the concepts presented in chapter 3, we briefly reintroduce a rule representation for XCS and UCS that is able to deal with data that contain continuous attributes; then, we revise the concept of niche under this new representation. Lastly, we discuss which information is lacking in real-world problems to apply the theory.

7.1.1 XCS and UCS Enhancements to Deal with Continuous Data

Thus far, all the artificial problems used in the previous chapters were defined with binary strings. To solve these problems, we used the original rule representation defined by Wilson (1995), in which each variable of a rule takes a value of the ternary alphabet {0,1,#} (see chapter 3). Nonetheless, real-world problems have new types of attributes such as continuous attributes and ordered nominal attributes. To cope with these new types of data, the system was provided with an interval-based representation in which each variable of a rule is coded with an interval which determines the range of values that the corresponding input attribute can take (Wilson, 2001; Stone and Bull, 2003). Therefore, a rule is a conjunction of feasible intervals, and a new example e matches the a rule if each attribute e_i is included in the interval of the corresponding variable of the rule. For more information about this rule representation the reader is referred to chapter 3.

The introduction of this new representation makes the definition of problem niche and representative of a niche a little fuzzy. According to the definitions given in chapter 5, a niche is a subproblem where a maximally general sub-solution applies. This niche is defined by a schema, and a representative of a niche is any classifier whose condition specifies all the relevant bits of the schema.

These ideas still apply—not rigorously, but intuitively—to real-world problems. In the



(a) Niches

(b) Rules

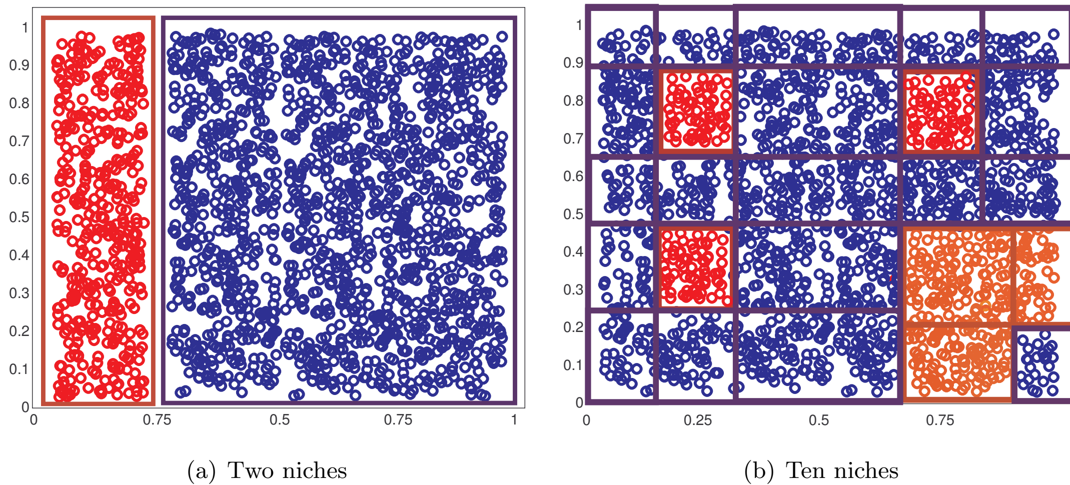
Figure 7.1: Example of a domain with two niches (a) and examples of possible representatives of the two niches and over-general classifiers (b) in a two-dimensional problem with continuous attributes

interval-based representation, a niche can be defined as a hyper rectangle in the feature space in which a maximally general and accurate solution applies. This hyper rectangle expresses the schema of the niche. Consequently, a representative of this niche is any classifier whose condition draws a hyper rectangle included in the hyper rectangle of the niche. Moreover, an over-general classifier is a classifier that defines a hyper rectangle that covers examples of different classes.

To further explain the consequences of this redefinition, figure 7.1(a) shows a very simple two-class domain where there are two niches, and figure 7.1(b) illustrates some examples of representative classifiers of each niche and over-general classifiers. This simple example shows two important aspects that must be highlighted since they make the difference with the definitions of niche and representatives given in the previous chapters. Firstly, notice that there may be several hyper rectangles that represent the niche. That is, by slightly varying one of the sides of any of the hyper rectangles in figure 7.1(a), with the condition that all the instances of the corresponding class are covered and that no instance of another class is matched, we obtain another hyper rectangle that can represent the niche as well. Therefore, the definition of niche schema is not deterministic. Secondly, there may be different accurate representatives of the niches whose condition is partially overlapped. This aspect is not exclusive of continuous-valued problems; in binary problems, we could find some overlapped representatives. Nonetheless, in continuous-valued problems, as the interval-based representation can define any possible hyper rectangle, the number of potential overlapping representatives increases abruptly. In general, it can exist an infinite number of representatives, equally general[1], that are highly overlapped. For example, in figure 7.1(b) there are two representatives of the red niche that are equally general and highly overlapped.

Although these differences, the ideas derived from the facetwise analysis are still valid in this new scenario. That is, the same mechanisms of the evolutionary learners apply: the population is initialized by the covering operator, and the evolutionary pressures drive the search toward obtaining representatives of different niches. These best representatives should be able

[1]In the interval-based representation, the generality can be computed as the volume of the hyper rectangle defined by the condition.



(a) Two niches

(b) Ten niches

Figure 7.2: Example of two domains with the same imbalance ratio in the training data set but different niche imbalance ratio.

to take over their niches and remove competing over-general, less accurate classifiers. The main difference with the binary case is that, due to the fuzziness inherent in the definition of niche schema, there may be several highly overlapped representatives of the niches which will share the niche resources instead of having a single representative per niche. Therefore, an effort needs to be made to connect this new situation to the theory. The implications of this difference are analyzed in more detail in the next subsection.

7.1.2 What Do we Need to Apply the Theory?

Having redefined the concepts of niche and representative in imbalanced domains, now we are in position to examine which information, handy in the artificial problems used in the two previous chapters, is not available in real-world data sets. That is, the different models developed in the previous chapters were translated into a set of recommendations that suggested to configure different parameters of XCS and UCS depending on the imbalance ratio ir. We assumed that the class-imbalance ratio of the training data set reflected the ratio of the frequency of activation of nourished niches to the frequency of activation of starved niches, which we refer to as the *niche imbalance ratio* in the rest of this chapter. This condition was satisfied in the tested artificial problems, since the number of starved niches was equal to the number of nourished niches, all nourished niches had the same activation frequency, and all starved niches had the same activation frequency which, in turn, was smaller than that of nourished niches.

Nonetheless, this is not the case in real-world domains since, the formation of niches depends not only on the class-imbalance ratio but also on the distribution of the training examples in the feature space. To illustrate this, figure 7.2 shows two domains with the same imbalance ratio. Notice that, in these examples, the niche imbalance ratio is not directly determined by the class-imbalance ratio of the training data set. That is, the domain in figure 7.2(a) contains one niche of the minority class, and the domain in figure 7.2(b) consists of five niches of the

minority class—two of them overlapped—with a lower number of instances per niche. Therefore, the domain in figure 7.2(b) is more difficult to learn than the domain in figure 7.2(a), since both LCSs have to discover a larger number of smaller niches. Besides, notice that, in figure 7.2(b), there exist niches with different frequencies of activation[2], and that, in this case, the most starved niche corresponds to a niche of the majority class—that is, the bottom right most niche.

Two important conclusions can be extracted from this elemental example. Firstly, that the imbalance ratio of the training data set may provide a misleading information about the real niche imbalance; therefore, we need to develop new procedures to obtain more accurate estimates of the niche imbalance ratio. Secondly, that the *niche imbalance problem* may be present in the majority of real-world domains, and that this is related to (1) the *knowledge representation* and (2) the *geometrical distribution* of the training examples in the feature space.

To further illustrate this last point, let us suppose that XCS or UCS are used to extract an interval-based rule set from the domain represented in figure 7.3, which is a completely balanced data set with oblique boundaries. The same figure shows some of the possible niches and representatives of the blue class. The two LCSs may be expected to have no problems to learn this domain since the training data set is completely balanced. Nevertheless, note that the interval-based representation needs to evolve some representatives whose conditions define small hyper rectangles to approximate the class boundary accurately; these representatives belong to *starved niches*. On the other hand, the interval-based representation also enables the existence of representatives with larger conditions—which belong to *nourished niches*—that match examples that are far away from the class boundary. Notice that, in this particular example, this problem is due to the combination of geometrical complexity and expressiveness—or shape—of the rule representation, but not to the class-imbalance ratio. If we had conditions that defined triangles in the solution space, this domain could be predicted with only two rules.

In fact, a similar problem has been addressed by the machine learning community, in the context of offline learning, under the label of the problem with *small disjuncts* (Holte et al., 1989). That is, a disjunct is the analogous definition of niche in an offline system, and the problem of small disjuncts refers to the problem of extracting accurate models of infrequent or starved niches. As discussed in chapter 5, approaches designed to deal with this problem in offline learning can be barely carried over to online learning since, in the latter one, instances are made available in data streams, and so, no information about the class distribution is known a priori. For sake of notation, in the remainder of this chapter, we will indistinctively use the two terms to refer to the described problem.

In summary, the problem of learning from imbalanced domains has been broadened due to the presence of continuous attributes; note that, now, the effects of class imbalances can be present in any real-world problem. Thence, we reformulate the problem as follows. As in the binary case, we are concerned about the competition among starved niches, nourished niches, and over-general classifiers. Notwithstanding, the imbalance ratio gives now little information about the distribution of niches around the feature space. In this context, the general purpose would be to estimate the number of niches of the system and the frequency of all these niches to get an accurate estimate of the niche imbalance ratio and tune the configuration of the two LCSs based on this estimate. In fact, if we could be able to perform this complex task, we would

[2]The frequency of activation is directly related to the number of instances that are included in the hyper rectangle defined by the niche.

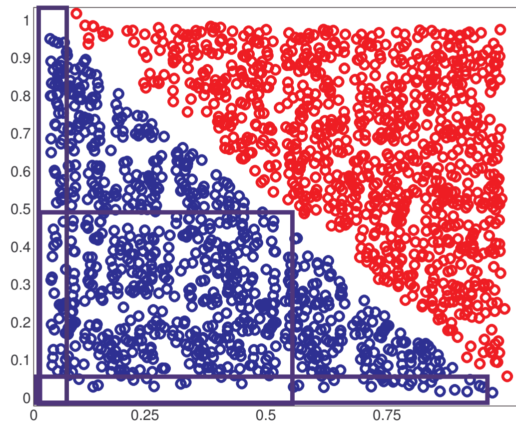


Figure 7.3: Example of a domain with oblique boundaries. Several interval-based rules are required to define the class boundary precisely.

have solved the learning problem itself, removing the necessity of applying a machine learning technique. Here, we relax the goal of obtaining the information of all niches and define that the niche imbalance ratio equals to the ratio between the frequency of the most nourished niche and the frequency of the most starved niche that lay together in the solution space. Thus, we only need to estimate the frequency of two niches of our problem to obtain an estimate that represents the upper bound of the niche imbalance ratio. Even though this simplification is done, the problem of estimating the niche frequency is not trivial. In the following section, we propose a mechanism to estimate the niche imbalance ratio.

7.2 Self-Adaptation to Particular Unknown Domains

This section presents a heuristic approach to determine the frequency of application of starved and nourished niches and self-configure XCS and UCS based on this information. This approach enables us to properly estimate the niche imbalance ratio and so to self-configure both LCSs according to this information. Before applying XCS and UCS to real-world problems, we show that this self-configuration procedure enables both XCS and UCS to solve the imbalanced 11-bit multiplexer problem with large imbalance ratios without being previously configured according to the imbalance ratio.

7.2.1 Online Adaptation Algorithms

To estimate the niche imbalance ratio ir_n and self-adapt the LCSs based on this estimate, we propose to use the information that intrinsically resides in over-general classifiers. Over-general classifiers cover several niches that are close in the feature space. By computing the number of examples covered per class of an over-general classifier, we can estimate the imbalance ratio between these niches. Note that this strategy permits not only detecting the presence of starved niches, but also calculating an estimate of the imbalance ratio between these starved niches and

Algorithm 7.2.1: Pseudo code for the *online adaptation algorithm* in XCS.

```
1 Algorithm: OnlineAdaptationXCS ( cl is classifier )
  Data: cl is a classifier after updating its parameters.
  Result: Modify β and θ_GA if necessary.
2 if cl is overgeneral then
3 |   ir_n := exp_maj(cl) / exp_min(cl)
4 |   if ( ir_n < 2R_max/ε_0 ∧ exp_cl > θ_ir ∧ num_cl > avg(num)_[P] ) then
5 |   |   Adapt β and Adapt θ_GA based on ir_n
6 |   end
7 end
```

Algorithm 7.2.2: Pseudo code for the on-line adaptation of β.

```
1 Algorithm: Adapt β ( ir_n is double )
  Data: ir_n is the niche imbalance ratio
  ζ is a discount factor (0 < ζ < 1)
  Result: New value of β.
2 ε_th = 2 · R_max · ir/(1+ir)^2
3 Obtain ε_ir_β with the current value of β
4 while ε_ir_β < p_th do
5 |   β = β · ζ
6 |   Obtain ε_ir_β with the current value of β
7 end
```

their neighbors.

We first present the algorithm for online self-adaptation of XCS and later translate this algorithm to the particular case of UCS. Algorithm 7.2.1 provides the pseudo code of the main procedure, and algorithm 7.2.2 supplies the code for the subroutine that specifically tunes the β parameter for the Widrow-Hoff rule. The algorithm works as follows. Algorithm 7.2.1 is applied to each classifier after updating its parameters. It first checks whether the classifier is over-general or not. For this purpose, we extended the parameters of a classifier to compute the experience per class. Then, a classifier is over-general if it is experienced in more than one class. Next, the imbalance ratio between the niches in which this classifier participates is estimated as follows. We select the class with maximum experience $exp_{maj}(cl)$ and the class with minimum experience $exp_{min}(cl)$ (we require that $exp_{min}(cl) > 0$), and return the ratio of these two values as an estimate of the niche imbalance ratio. Then, we use this information to self-adapt the configuration of XCS only if (i) the niche imbalance ratio is less than the maximum imbalance ratio identified in equation 5.9, (ii) the classifier is experienced enough ($exp(cl) > \theta_{ir}$, where θ_{ir} is a configuration parameter), and (iii) the classifier is strong in the population, i.e., if its numerosity is greater than the average numerosity of the classifiers in the population.

If we are using the Widrow-Hoff rule, we first adapt β. Algorithm 7.2.2 provides the implementation details of this procedure. The goal of the algorithm is to adjust the value of β

Algorithm 7.2.3: Pseudo code for the *online adaptation algorithm* in UCS.

```
1 Algorithm: OnlineAdaptationUCS ( cl is classifier )
  Data: cl is a classifier after updating its parameters.
  Result: Modify θ_GA if necessary.
2 if cl is overgeneral then
3     ir_n := exp_maj(cl) / exp_min(cl)
4     if ( ir_n < acc_0 ∧ exp > θ_ir ∧ num_cl > avg(num)_[P] ) then
5         Adapt β and Adapt θ_GA based on ir_n
6     end
7 end
```

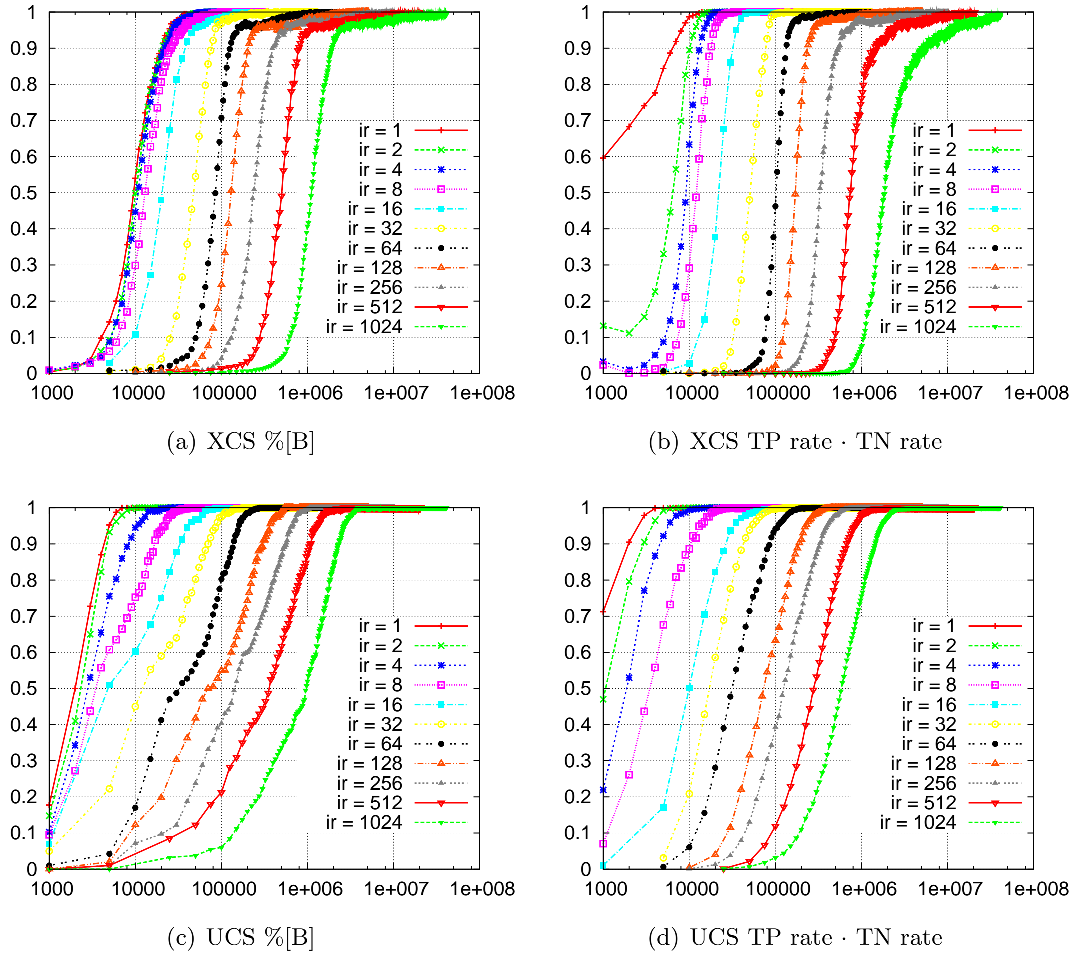
so that the estimate error of a classifier approaches its theoretical value, which is computed in equation 5.5. Thus, the procedure first computes this theoretical value, ϵ_{th}. Then, it calculates the real value of the error for the given β. To do this, the algorithm assumes the worst case: that an instance of the minority class is sampled, and then, ir_{cl} instances of the majority class are received. If the real value of the error is lower than the theoretical one, β is decreased according to a discount factor ζ. This process is repeated until a β for which the theoretical and the real value of the error are approximately the same is found. Finally, the algorithm adapts θ_{GA} by setting $\theta_{GA} = ir$.

In algorithm 7.2.3, this procedure is extended to UCS. The algorithm works similarly to the one designed for XCS with two main differences. The first difference is that, in UCS, the update parameter procedure does not need to be adapted. The second difference is that the condition to update θ_{GA} depends on whether $ir_n > acc_0$. The remaining part of the algorithm is the same. In the next section, we show that these self-adaptation mechanisms enable XCS and UCS to self-configure according to the estimated niche imbalance ratio and solve the imbalanced 11-bit multiplexer with large imbalance ratios.

7.2.2 Experiments

In this section, we empirically analyze whether the two heuristic procedures can provide accurate estimates for XCS and UCS. For this purpose, we ran XCS and UCS on the imbalanced 11-bit multiplexer problem with $ir = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$, that is, the same experiments with the imbalanced multiplexer problem performed in chapters 5 and 6. We used the same configuration proposed in these two chapters for XCS and UCS, but with the following exceptions. We set $\theta_{GA} = 0$ for both systems and fixed $\beta = 0.2$ for XCS. That is, we did not configured the systems according to the imbalance ratio and the recommendations derived from the theory. Therefore, we expected the heuristic procedures to discover the niche imbalance ratio of each problem and to use it to self-adapt both systems. All the results provided as follows are averages over 25 runs with different random seeds.

Figure 7.4 plots the results obtained by XCS and UCS in the imbalanced 11-bit multiplexer problem with imbalance ratios ranging from $ir = 1$ to $ir = 1024$. More specifically, figures 7.4(a) and 7.4(c) plot the proportion of the optimal population achieved by XCS and UCS, and figures 7.4(b) and 7.4(d) depict the evolution of the performance, measured as the product of TN rate



(a) XCS %[B]

(b) XCS TP rate · TN rate

(c) UCS %[B]

(d) UCS TP rate · TN rate

Figure 7.4: Evolution of (a,c) the proportion of the optimal population and (b,d) the geometric mean of TP rate and TN rate of XCS and UCS, respectively, in the 11-bit multiplexer with ir={1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}.

and TP rate, of XCS and UCS respectively. These results can be compared with those obtained by XCS and UCS when they were properly configured according to the imbalance ratio and the recommendations derived from the theory (see figures 5.11 and 6.5 respectively).

The results show that both XCS and UCS could achieve 100% optimal population and 100% performance for all the imbalance ratios. Therefore, this confirmed that the heuristic procedure was able to tune the configuration of both LCSs correctly. Furthermore, these results also permitted establishing a comparison of the performance of XCS and UCS. UCS achieved 100% of the optimal population slightly quicker than XCS, especially in the larger imbalance ratios. Note that, although XCS's curves were stepper at the beginning of the run, the system suffered more than UCS to discover the last optimal classifiers in the population. These behavior was also observed in the performance curves for high imbalance ratios. Nonetheless, it is worth noticing that both approaches could completely learn all the optimal population and classify all

the input instances correctly with similar training time. This indicates that the application of these systems in real-world imbalanced domains with unknown characteristics holds promise. In the next section, we deal with real-world problems and analyze the capabilities of both online learning architectures.

7.3 LCSs in Imbalanced Real-World Domains

After analyzing the two LCSs on artificial problems, we now apply both systems to a collection of real-world imbalanced problems and examine their behavior. The understanding of LCSs behavior—and the behavior of any learner in general—on real-world problems is really complicated since these problems may have different sources of complexity which can hardly be identified; the interaction of all these complexities may limit the maximum performance that a given learner can achieve (Ho and Basu, 2002). Notice the difference between real-world problems and the artificial problems that have been used through all the previous study, where we could control the different sources of complexity.

Thence, we need to take another approach to evaluate the competence of XCS and UCS in real-world problems. That is, information about the optimal population is no longer available, and providing the training or test accuracy of the two learners may be not enough to conclude whether the two systems are competitive for mining real-world domains. To measure the competence of both LCSs on imbalanced real-world domains, we compare the performance of XCS and UCS with three of the most influential machine learning systems (Wu et al., 2007). Therefore, the aim of this section is to analyze whether XCS and UCS are competitive with these highly recognized learning methods. It is worth noticing that XCS and UCS perform online learning—i.e., they process data streams—, whereas the three order methods learn offline. Thus, XCS and UCS provide an added value with respect the three other techniques. As proceeds, we first present the methodology, and then, we compare XCS and UCS with the other learners.

7.3.1 Comparison Methodology

Before proceeding with the analysis of the experimental results, we first describe the test problems used in the comparison, and the details about the metrics used to evaluate the learners and the statistical tests employed to aid the process of conclusion extraction.

We used a collection of 25 real-world problems with different characteristics and imbalance ratios, which were constructed as follows. We selected the following twelve problems: *balance-scale*, *bupa*, *glass*, *heart disease*, *pima indian diabetes*, *tao*, *thyroid disease*, *waveform*, *Wisconsin breast-cancer database*, *Wisconsin diagnostic breast cancer*, *wine recognition data*, and *Wisconsin prognostic breast cancer*. All the real-world problems were obtained from the UCI repository (Asuncion and Newman, 2007), except for *tao*, which was selected from a local repository (Bernadó-Mansilla et al., 2002). To force higher imbalance ratios and increase the test bed, we discriminated each class against all the other classes in each data set, considering each discrimination as a new problem. Thus, n two-class problems were created from a problem with n classes ($n > 2$), resulting in a test bed that consisted of 25 two-class real-world problems. Table 7.1 gathers the most relevant features of the problems. Note that the imbalance ratio between niches ir_n can be much higher than the imbalance ratio of the learning data set reported in the

Table 7.1: Description of the data sets properties. The columns describe the data set identifier (Id.), the original name of the data set (Data set), the number of problem instances (#Ins.), the number of attributes (#At.), the proportion of minority class instances (%Min.), the proportion of majority class instances (%Maj.), and the imbalance ratio (ir).

Id.	Data set	#Ins.	#At.	%Min.	%Maj.	ir
bald1	balance-scale disc. 1	625	4	7.84%	92.16%	11.76
bald2	balance-scale disc. 2	625	4	46.08%	53.92%	1.17
bald3	balance-scale disc. 3	625	4	46.08%	53.92%	1.17
bpa	bupa	345	6	42.03%	57.97%	1.38
glsd1	glass disc. 1	214	9	4.21%	95.79%	22.75
glsd2	glass disc. 2	214	9	6.07%	93.93%	15.47
glsd3	glass disc. 3	214	9	7.94%	92.06%	11.59
glsd4	glass disc. 4	214	9	13.55%	86.45%	6.38
glsd5	glass disc. 5	214	9	32.71%	67.29%	2.06
glsd6	glass disc. 6	214	9	35.51%	64.49%	1.82
h-s	heart-disease	270	13	44.44%	55.56%	1.25
pim	pima-inidan	768	8	34.90%	65.10%	1.87
tao	tao-grid	1888	2	50.00%	50.00%	1.00
thyd1	thyroid disc. 1	215	5	13.95%	86.05%	6.17
thyd2	thyroid disc. 2	215	5	16.28%	83.72%	5.14
thyd3	thyroid disc. 3	215	5	30.23%	69.77%	2.31
wavd1	waveform disc. 1	5000	40	33.06%	66.94%	2.02
wavd2	waveform disc. 2	5000	40	33.84%	66.16%	1.96
wavd3	waveform disc. 3	5000	40	33.10%	66.90%	2.02
wbcd	Wis. breast cancer	699	9	34.48%	65.52%	1.90
wdbc	Wis. diag. breast cancer	569	30	37.26%	62.74%	1.68
wined1	wine disc. 1	178	13	26.97%	73.03%	2.71
wined2	wine disc. 2	178	13	33.15%	66.85%	2.02
wined3	wine disc. 3	178	13	39.89%	60.11%	1.51
wpbc	wine disc. 4	198	33	23.74%	76.26%	3.21

table.

The performance was measured with the product of TP rate and TN rate. Ten-fold cross validation (Dietterich, 1998) was used to estimate the product of TP rate and TN rate. The results obtained with the different techniques were statistically compared with the following procedure. We first used the multiple-comparison Friedman's test (Friedman, 1937, 1940) to test the null hypothesis that all the learning methods performed the same on average. If the null hypothesis was rejected, the Nemenyi test (Nemenyi, 1963) was employed to identify groups of learners with statistically equivalent results. Moreover, as we were interested in analyzing the differences in particular problems, the performance of each pair of learning algorithms on each problem was compared using the Wilcoxon signed-ranks test (Wilcoxon, 1945). We acknowledge in advance that pairwise comparisons increment the risk of rejecting null hypotheses that are actually true.

In our experiments, we assume this risk with the aim of providing further information about the excellence of each learning algorithm in particular problems. For more information about the used tests, the reader is referred to appendix B.

Both LCSs were compared with three of the most competent learners: C4.5 (Quinlan, 1995), SMO (Platt, 1998), and IBk (Aha et al., 1991). C4.5 is a decision tree derived from the ID3 algorithm (Quinlan, 1979). SMO is a support vector machine (Vapnik, 1995) that implements the *Sequential Minimal Optimization* algorithm. IBk is a nearest neighbor algorithm. All these machine learning methods were run using WEKA (Witten and Frank, 2005), and the recommended default configuration was used. We selected the model for SMO as follows. We ran SMO with polynomial kernels of order 1, 5, and 10, and with Gaussian kernels. Then, we ranked the results obtained with the four configurations and chose the model that maximized the average rank: SMO with lineal kernels. In this way, we avoided using particular configurations for each problem. We followed the same process with IBk, which was ran for $k = \{1, 3, 5, 7\}$; here, we provide the results with k=5. XCS and UCS were configured as previously specified, except for N=6400, and the two parameters that refer to the interval-based representation, i.e., r_0=0.6, and m_0=0.1. Finally, we did not introduce asymmetric cost functions in any system, although the majority of them permitted it. In this way, we aimed at analyzing the intrinsic capabilities of each method to deal with class imbalances.

7.3.2 Results

After defining the experimental methodology, we now analyze the results obtained with the different learning methods. Table 7.2 summarizes the performance of the different learners on the 25 data sets. The last three rows provide the average accuracy, the average rank, and the position of each learner in the ranking. The ranks were calculated as follows. For each data set, we ranked the learning algorithms according to their performance; the learner with the highest accuracy held the first position, whilst the learner with the lowest accuracy held the last position of the ranking. If a group of learners had the same performance, we assigned the average rank of the group to each one of the learners in the group.

The results provided in this table allowed for two types of analyses. Firstly, the results indicated which problems were more complex, in general, for all the learning systems. All learners presented poor performance in the problems *bald1*, *bpa*, *glsd1*, *glsd3*, *pim*, and *wpbc*. Examining the measure of performance, we observed that all the learners had a low TP rate, which indicated that the minority class was not well defined in these problems. Most of these data sets were highly imbalanced; so, the imbalance ratio turned up to be an important factor that hindered the performance of the tested learners. Nonetheless, the problems *bpa* and *pim* were almost balanced, so there might be other complexity factors affecting the learning performance such as small disjuncts.

Moreover, the experimental results also allowed for a statistical comparison of the performance of the different learners. Firstly, let us note that XCS and UCS were the two best ranked techniques. That means that the two LCSs were among the best performers in most of the problems. To analyze whether this improvement was statistically significant, we used multiple-comparison tests to check the null hypothesis that all the learners performed the same on average. The Friedman multiple-comparison test did not permit rejecting the null hypothesis with p = 0.2519. Consequently, post-hoc tests could not be applied since no significant

Table 7.2: Comparison of C4.5, SMO, IBk, XCS, and UCS on the 25 real-world problems. Each cells depicts the average value of the product of TP rate and TN rate and the standard deviation. *Avg* gives the performance average of each method over the 25 data sets. The two last rows show the average rank of each learning algorithm (*Rank*) and its position in the ranking (*Pos*).

	C4.5	**SMO**	**IB5**	**XCS**	**UCS**
bald1	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
bald2	69.30 ± 6.83	84.03 ± 7.30	81.16 ± 5.54	71.14 ± 5.02	69.75 ± 8.19
bald3	71.20 ± 6.04	85.81 ± 8.40	82.11 ± 8.67	69.98 ± 7.23	73.61 ± 6.66
bpa	33.08 ± 14.09	0.00 ± 0.00	32.40 ± 9.44	47.58 ± 10.92	47.59 ± 11.22
glsd1	79.50 ± 42.16	0.00 ± 0.00	69.32 ± 48.30	20.00 ± 42.16	59.00 ± 50.87
glsd2	34.50 ± 47.43	15.00 ± 33.75	24.13 ± 35.36	59.00 ± 45.02	74.00 ± 41.89
glsd3	28.97 ± 42.16	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	19.49 ± 25.17
glsd4	73.55 ± 32.63	80.03 ± 24.33	77.07 ± 24.98	80.03 ± 24.33	83.54 ± 19.53
glsd5	66.52 ± 16.77	9.50 ± 9.42	62.26 ± 21.14	68.67 ± 18.71	65.63 ± 21.46
glsd6	52.54 ± 15.13	0.00 ± 0.00	61.74 ± 18.23	60.53 ± 11.21	57.06 ± 14.20
h-s	63.33 ± 13.29	68.83 ± 8.87	64.40 ± 14.65	59.89 ± 15.59	55.00 ± 13.61
pim	43.87 ± 13.27	48.31 ± 5.60	46.91 ± 4.84	45.85 ± 6.37	47.82 ± 6.60
tao	90.98 ± 2.14	70.59 ± 6.45	94.25 ± 2.10	82.89 ± 5.42	78.81 ± 7.18
thyd1	87.61 ± 16.10	76.67 ± 22.50	76.67 ± 22.50	78.36 ± 22.01	92.25 ± 13.66
thyd2	93.24 ± 12.45	54.17 ± 24.92	77.90 ± 21.40	82.50 ± 24.98	93.06 ± 12.09
thyd3	87.65 ± 10.34	33.81 ± 21.35	81.12 ± 16.16	89.84 ± 11.75	88.08 ± 14.89
wavd1	67.79 ± 4.06	78.68 ± 4.27	72.28 ± 3.97	80.44 ± 2.97	76.33 ± 2.10
wavd2	62.54 ± 3.89	72.30 ± 2.71	67.49 ± 1.75	73.48 ± 2.88	71.49 ± 3.83
wavd3	68.60 ± 2.38	79.57 ± 2.04	74.14 ± 2.86	81.01 ± 3.99	76.60 ± 4.14
wbcd	89.12 ± 3.42	92.70 ± 5.32	92.72 ± 5.36	92.31 ± 5.50	94.06 ± 4.23
wdbc	88.79 ± 5.09	94.28 ± 3.28	93.47 ± 3.64	90.27 ± 4.61	89.68 ± 5.61
wined1	85.15 ± 16.63	98.46 ± 3.24	94.98 ± 8.29	99.23 ± 2.43	99.23 ± 2.43
wined2	91.81 ± 8.05	97.50 ± 5.62	97.50 ± 4.03	99.17 ± 2.64	91.88 ± 10.02
wined3	87.62 ± 11.70	97.14 ± 6.02	87.94 ± 12.53	93.38 ± 7.15	85.33 ± 9.55
wpbc	33.55 ± 12.87	9.37 ± 16.98	28.98 ± 16.49	20.33 ± 16.38	17.17 ± 21.63
Avg	66.03	53.87	65.64	65.83	68.26
Rank	3.46	3.14	3.08	2.52	2.80
Pos	5	4	3	1	2

differences among the learners were found (Demšar, 2006). This conclusion is not surprising since compared XCS and UCS with three of the most competent machine learning techniques. Nonetheless, note that these results highlight the robustness of XCS and UCS. That is, XCS and UCS were not only as competitive as three of the most competent machine learning techniques in the used test bed, but they also were the best ranked methods of the comparison.

To extend the statistical analysis to each particular problem, we applied statistical pairwise comparisons according to a Wilcoxon signed-ranks test at 0.95 confidence level. Table 7.3 shows the results. The • and ∘ symbols denote a significant degradation/improvement of the given learning algorithm with respect to another in a particular data set. The overall degradation/improvement comparison (see the row labeled *Score*) permitted ranking the quality of the five learners. Under this criterion, XCS appeared as the most robust method with a ratio of

Table 7.3: Comparison of C4.5, SMO, IBk, XCS, and UCS on the 25 real-world problems. For a given problem, the • and ◦ symbols indicate that the learning algorithm of the column performed significantly worse/better than another algorithm at 0.95 confidence level (pairwise Wilcoxon signed-ranks test). *Score* counts the number of times that a method performed worse-better, and $Score_{ir>5}$ does the same but only for the highest imbalanced problems ($ir > 5$).

	C4.5	**SMO**	**IBk**	**XCS**	**UCS**
bald1					
bald2	••	◦ ◦ ◦	◦ ◦ ◦	••	••
bald3	••	◦ ◦ ◦	◦ ◦ ◦	••	••
bpa	• • ◦	• • ••	• • ◦	◦ ◦ ◦	◦ ◦ ◦
glsd1	◦◦	• • •	◦	•	◦
glsd2		••	•	◦	◦◦
glsd3					
glsd4					
glsd5	◦	• • ••	◦	◦	◦
glsd6	◦	• • ••	◦	◦	◦
h-s		◦	◦		••
pim					
tao	• ◦ ◦◦	• • ••	◦ ◦ ◦◦	• • ◦◦	• • •◦
thyd1					
thyd2	◦	• • ••	◦	◦	◦
thyd3	◦	• • ••	◦	◦	◦
wavd1	• • ••	◦◦	• • •◦	◦ ◦ ◦	• ◦ ◦
wavd2	• • ••	◦◦	• • •◦	◦◦	◦◦
wavd3	• • ••	◦◦	• • ◦	◦ ◦ ◦	•◦
wbcd	• • •	◦	◦		◦
wdbc	•	◦◦	◦	•	•
wined1	• • •	◦		◦	◦
wined2					
wined3		◦		◦	••
wpbc					
Score	26-10	29-18	11-22	8-20	14-18
$\mathbf{Score}_{ir>5}$	0-3	9-0	1-2	1-2	0-4

degradation/improvement of 8/20, followed closely by IBk and UCS. Both LCSs presented the poorest results with respect to the other learners in the *bald*2, *bald*3, and *tao* problems, which have a low imbalance ratio. In (Bernadó-Mansilla and Ho, 2005), the hyper rectangle codification used by XCS and UCS was shown to be inappropriate when the boundary between classes in the learning data set was curved. This is the case of the *tao* problem (Bernadó-Mansilla et al., 2002). We hypothesize that *bald*2 and *bald*3 are also characterized by curved boundaries, which would explain the degradation in performance of both LCSs. This hypothesis is also supported by the results obtained with IBk, which improved XCS and UCS in the three aforementioned problems. IBk is not affected by curved boundaries since it decides the output as the majority class of the k nearest neighbors.

The two last methods in the ranking were C4.5 and SMO. The surprisingly poor rank of C4.5 was mainly caused by the results obtained in the problems *wavd*1, *wavd*2, and *wavd*3,

in which C4.5 was outperformed by all the other learners. These results were not correlated with the imbalance ratio, so there may be other types of complexity that made C4.5 perform poorly in these problems. Finally, SMO was the last ranked method. It showed a tendency to over-generalize toward the majority class in problems with moderate and high class imbalances such as *glsd*1, *glsd*3, and *glsd*6, in which the TP rate was zero. The same behavior was shown in problems with low imbalance ratios such as the *bpa* problem, which we identified as a difficult problem may be due to the tendency of the learners to create small disjuncts to create accurate models of these problems. However, we can also find significant improvements with respect to other learners in the problems: *bald*2, *bald*3 and *wdbc*. Thus, these results indicate that SMO performs competitively in a restricted set of problems, but it is affected by some complexities among which we may find the imbalance ratio.

Finally, let us compare the learners in terms of imbalance robustness. To do this, we consider the data sets with the highest imbalance ratio: *glsd1*, *glsd2*, *bald1*, *glsd3*, *glsd4*, *thyd1*, and *thyd2*, which have imbalance ratios ranging from ir=5 to ir=23. In these problems, UCS appeared to be the best learner, with a degradation/improvement ratio of 0/4, followed closely by C4.5. These results agree with several papers which indicate that C4.5 can deal with high amounts of class imbalance (Japkowicz and Stephen, 2002; Batista et al., 2004). IBk and XCS were the two next methods in the ranking. IBk might suffer from *small disjuncts*, since minority class regions are surrounded by many instances of the majority class, concentrating a high amount of the test error around the small disjuncts. XCS also turned up to be more sensitive to class imbalances than UCS and C4.5. Lastly, SMO performed poorly in the most imbalanced data sets. As mentioned above, we tried other orders of polynomial kernels, as well as a Gaussian kernel, but no significant improvement was found.

In this section, we have shown the competitiveness of both XCS and UCS with respect to three of the most influential machine learning techniques in imbalanced data sets. Throughout all the comparison, we have considered the intrinsic capabilities of the learners to deal with rare classes without introducing further mechanisms to promote the discovery of the knowledge that resides in the minority class. In the following sections, we consider these types of mechanisms. As the comparison contains learning algorithms with different characteristics, we use re-sampling techniques since they are independent of the final learner. As proceeds, we first explain the re-sampling methods considered in the analysis, and further study whether they improve the accuracy of the models of the minority class when they are combined with the five learning methods used in this section.

7.4 Re-sampling Techniques

Re-sampling techniques have become one of the most used approaches to boost the capabilities of machine learning techniques to discover the knowledge that resides in rare classes. These types of methods are pre-processing methods that re-balance the proportion of examples of the minority class in the training data set by either over-sampling the minority class or under-sampling the majority class. During the last few years, several approaches have been developed in this field. Herein, we adopt three of the most famous algorithms: random over-sampling (Ling and Li, 1998), under-sampling based on Tomek links (Batista et al., 2004), and synthetic minority over-sampling technique (SMOTE) (Chawla et al., 2002). We selected these three

Algorithm 7.4.1: Pseudo code for the Tomek Links algorithm.

```
1  Algorithm: TomekLinks ( d is Dataset )
   Data: d is the training data set
   Result: Collection of Tomek links represented as pairs of examples
2  var
3  |  setTomek is PairsExamples
4  |  exMin, exMaj, ex is TrainingExample
5  |  dst is double
6  end
7  forall  example of the majority class  exMaj  in  d do
8  |  forall  example of the minority class  exMin  in  d do
9  |  |  dst = dist(exMin, exMaj)
10 |  |  if ¬ ∃ex ∈ d|dist(ex, exMin) < dst ∨ dist(ex, exMaj) < dist then
11 |  |  |  cjtTomek := addLink (cjtTomek,<exMin,exMaj>)
12 |  |  end
13 |  end
14 end
```

algorithms since they have been empirically shown to the some of the most competitive re-sampling techniques (Chawla et al., 2002; Batista et al., 2004). Moreover, we introduce a modified version of SMOTE that incorporates a data cleaning process, which we address as *cluster SMOTE* (cSMOTE) (Orriols-Puig and Bernadó-Mansilla, 2008b). As proceeds, each one of these approaches is explained in detail, and their behavior is illustrated in a case study; in the next section, the performance of each one of the re-sampling techniques, in combination with the five learners, is empirically examined.

7.4.1 Random Over-sampling

The first re-sampling technique considered in the comparison is random over-sampling. This is a very simple approach that proposes to over-sample the rare classes in the training data set so as to match the size of the majority class. Although the simplicity of this approach, several authors have demonstrated that it improves the performance of highly-known learners in pattern recognition tasks. Japkowicz and Stephen (2000) showed that random over-sampling, combined with a multi-layer perceptron classifier (Rumelhart et al., 1986), was a very effective method to deal with class imbalances. Later, Japkowicz and Stephen (2002) extended this conclusion to the C5.0 decision tree. Moreover, Batista et al. (2004) experimentally showed that random over-sampling resulted in one of the best improvements in comparison with eleven more sophisticated re-sampling techniques. Due to these excellent results, we included random over-sampling in our experiments.

7.4.2 Under-sampling based on Tomek Links

Under-sampling with Tomek links (Batista et al., 2004) is an under-sampling technique that uses the concept of *Tomek link* (Tomek, 1976) to remove examples of the majority class from the training data set. As follows, we first present the concept of Tomek link and then explain how the re-sampling technique works.

The goal of the Tomek links procedure is to find pairs of examples with different class, but that are geometrically close in the feature space. Algorithm 7.4.1 provides the pseudo code of the algorithm that finds all the Tomek links. That is, a *tomek link* is a pair of examples $< E_i, E_j >$ that belong to different classes and for which there does not exist any other example E_k so that $dist(E_i, E_k) < dist(E_i, E_j)$ or $dist(E_j, E_k) < dist(E_i, E_j)$, where $dist$ is a function that computes the distance between two examples.

Therefore, the examples that do not form any Tomek link are not in the decision boundary, and thus, the information that they provide is not as interesting as the information that resides in the examples that form Tomek links. Batista et al. (2004) used this idea and proposed an under-sampling technique that removed examples of the majority class that did not belong to any Tomek link. This technique showed to provide competitive results, especially when combined with other over-sampling methodologies. For this reason, we incorporate this re-sampling algorithm in our analysis.

7.4.3 SMOTE

The *synthetic minority over-sampling technique* (SMOTE), originally designed by Chawla et al. (2002), is one of the most influential re-sampling techniques. SMOTE is an over-sampling technique that creates new minority class instances by means of performing different operations on the minority class instances of the training data set. Therefore, the application of this technique results in a new data set where the presence of the minority class is increased by the creation of new "synthetic" examples of the minority class. As follows, the details of the algorithm are given.

Algorithm 7.4.1 provides the pseudo code for the SMOTE algorithm, which works as follows. For each example of the minority class e_i, the procedure searches for the k nearest neighbors of e_i that also belong to the minority class. Then, it creates N examples of the minority class along the line segments joining any of the k minority class nearest neighbors (the value of N depends on the desired degree of over-sampling). To achieve this, for each new example of the minority class that has to be generated, the algorithm randomly selects one of the k nearest neighbors e_r and creates a new instance in which each attribute is a randomly generated on the segment that joins e_i and e_r.

Chawla et al. (2002) empirically demonstrated the competitiveness of SMOTE with respect to other re-sampling techniques. Subsequent to this publication, several authors proposed new approaches, based on SMOTE, to generate synthetic data. For example, Chawla et al. (2003) combined SMOTE with a boosting technique to improve the detection of rare classes. Later, Han et al. (2005) designed a new approach which mainly used the SMOTE algorithm, but trying to re-sample only those instances that lay closely to the decision boundary. All these new approaches supposed little modifications of the initial idea of creating synthetic data of the

Algorithm 7.4.2: Pseudo code for the SMOTE algorithm.

```
1  Algorithm: SMOTE ( d is Dataset, N is integer, k is integer ) return (dOut is
   Dataset)
   Data: N is the proportion of over-sampling
         k is the number of neighbors considered to create new instances
   Result: dOut is the new re-sampled data set
2  var
3  |  numNewNeigh, numMin, i is integer
4  |  currEx, newEx, selectedNeigh is Example
5  |  att is Attribute
6  |  dOut, neighbors is Dataset
7  end
8  numMin := number of instances of the minority class in dOut
9  i := 0
10 dout := d
11 while i < numMin do
12 |  currEx := get the ith example of the minority class
13 |  neighbors := get the k nearest neighbors of the minority class closer to currEx
14 |  numNewNeigh := N
15 |  while numNewNeigh > 0 do
16 |  |  selectedNeigh := randomly get a neighbor from neighbors
   |  |  /* create a new example of the minority class                  */
17 |  |  forall attribute att do
18 |  |  |  newEx[att] := curEx[att] + (currEx[att] - selectedNeigh[att])· rand(0,1)
19 |  |  end
20 |  |  numNewNeigh := numNewNeigh - 1
21 |  |  dout := addExample(dout, newEx)
22 |  end
23 |  i := i + 1
24 end
25 return dout
```

minority class. In the next section, we propose another modification of the SMOTE algorithm that combines these ideas with a data cleaning phase.

7.4.4 cSMOTE

We now introduce *cluster SMOTE* (cSMOTE), a re-sampling technique based on SMOTE. cSMOTE introduces two main modifications to the SMOTE algorithm, which consist in:

- including a phase to clean instances that are considered noise; and
- disabling the creation of new minority class instances beyond the boundaries of a virtual cluster calculated for each minority class instance.

Algorithm 7.4.3: Pseudo code for the cSMOTE algorithm.

```
1  Algorithm: cSMOTE ( d is Dataset, k is integer ) return (dOut is Dataset)
   Data: k is the number of neighbors considered to create new instances
   Result: dOut is the new re-sampled data set.
2  var
3  |  numFavor, numAgainst is integer
4  |  currEx, newEx, selectedNeigh is Example
5  |  att is Attribute
6  |  dOut,neighbors is Dataset
7  end
8  dOut = empty data set
9  forall example currEx en d do
10 |  neighbors := et the k nearest neighbors closer to currEx
11 |  numFavor := number of neighbors of the same class as currEx
12 |  numAgainst := number of neighbors of differnet class than currEx
13 |  if numFavor = 0 then
14 |  |  Do not insert currEx into dOut
15 |  else
16 |  |  if currEx belongs to the majority class then
17 |  |  |  dOut := addExample(dOut, currEx)
18 |  |  else
19 |  |  |  dOut := addExample(dOut, currEx)
20 |  |  |  while numAgainst > 0 do
21 |  |  |  |  selectedNeigh := select a neighbor from the same class as currEx
   |  |  |  |  /* create a new example of the minority class                */
22 |  |  |  |  forall attribute att do
23 |  |  |  |  |  newEx[att] := curEx[att] + (currEx[att] - selectedNeigh[att])· rand(0,1)
24 |  |  |  |  end
25 |  |  |  |  dOut := addExample(dOut, newEx)
26 |  |  |  |  numAgainst := numAgainst - 1
27 |  |  |  end
28 |  |  end
29 |  end
30 end
31 return dOut
```

As follows, we explain each one of these two modifications in detail.

Algorithm 7.4.3 provides the pseudo code for cSMOTE. The algorithm is guided by a main loop over all the examples of the data set, independent of whether they belong to the minority class or to the majority class. For each example e_i, the algorithm selects the k nearest neighbors regardless of their class; besides, it counts the number of these k neighbors that belong to the same class as e_i (*numFavor*) and the number of them that belong to another class (*numAgainst*). Depending on these variables and the selected example e_i, the next steps are taken:

1. If, given e_i, there does not exist any other example among its k nearest neighbors that belongs to the same class, e_i is considered as a noisy instance and it is not included in the final data set.

2. If e_i belongs to the majority class and, at least, there exists another example of the majority class among its k nearest neighbors, e_i is copied to the final data set.

3. If e_i belongs to the minority class, we create as many new examples as the number of nearest neighbors of the majority class that e_i has (i.e., new *numAgainst* instances of the minority class are created). The underlying idea of this re-sampling strategy is the fact that, in general, the number of nearest neighbors of another class would be higher as the minority class instance approaches the class boundary. Thence, this technique aims at introducing more examples of the minority class around the class boundary, but not increasing the number of them in regions of the feature space that are far from this boundary.

The generation of new examples of the minority class is also modified with respect to that in SMOTE. cSMOTE does not allow the creation of examples of the minority class beyond the virtual cluster to which the original example belongs. This virtual cluster is defined as follows. The center of the cluster is determined by the example E_i. Then, the cluster is defined by a sphere with radius equal to the distance to the example E_j, where E_j is the farthest neighbor of E_i for which it does not exist any other instance E_k of another class such as $dist(E_i, E_k) < dist(E_i, E_j)$. Thus, as the new instances are created in the segment defined between two instances of the minority class that belong to this cluster, they would never go beyond the virtual cluster. The behavior of cSMOTE and of the other three re-sampling techniques as well is exemplified in the following subsection.

7.4.5 What Do Re-sampling Techniques Do? A Case Study

Before proceeding with the comparison of the four re-sampling techniques—combined with each one of the five learning methods—on the collection of real-world problems, we first illustrate how these techniques work on a two-dimensional artificial problem. Thence, the purpose of this section is not to extract general conclusions about the re-sampling techniques behavior, but to intuitively explain how they work. As follows, we first introduce the artificial domain used in the case study and illustrate how the four re-sampling techniques modify this domain; then, we depict the knowledge created by the five different learners on this problem.

Artificial Problem Used in the Case Study

To illustrate the behavior of the different re-sampling techniques, we designed the two-dimensional artificial problem shown in figure 7.5(a). The problem consists of four concepts of the minority class (red dots) that have a circular shape and are distributed around the feature space. Moreover, the two concepts of the minority class placed in the top of the feature space contain a sub-concept of the majority class inside them, drawing the shape of a “doughnut”. Therefore, this data set shows that some of the small disjuncts belong to the majority class, highlighting

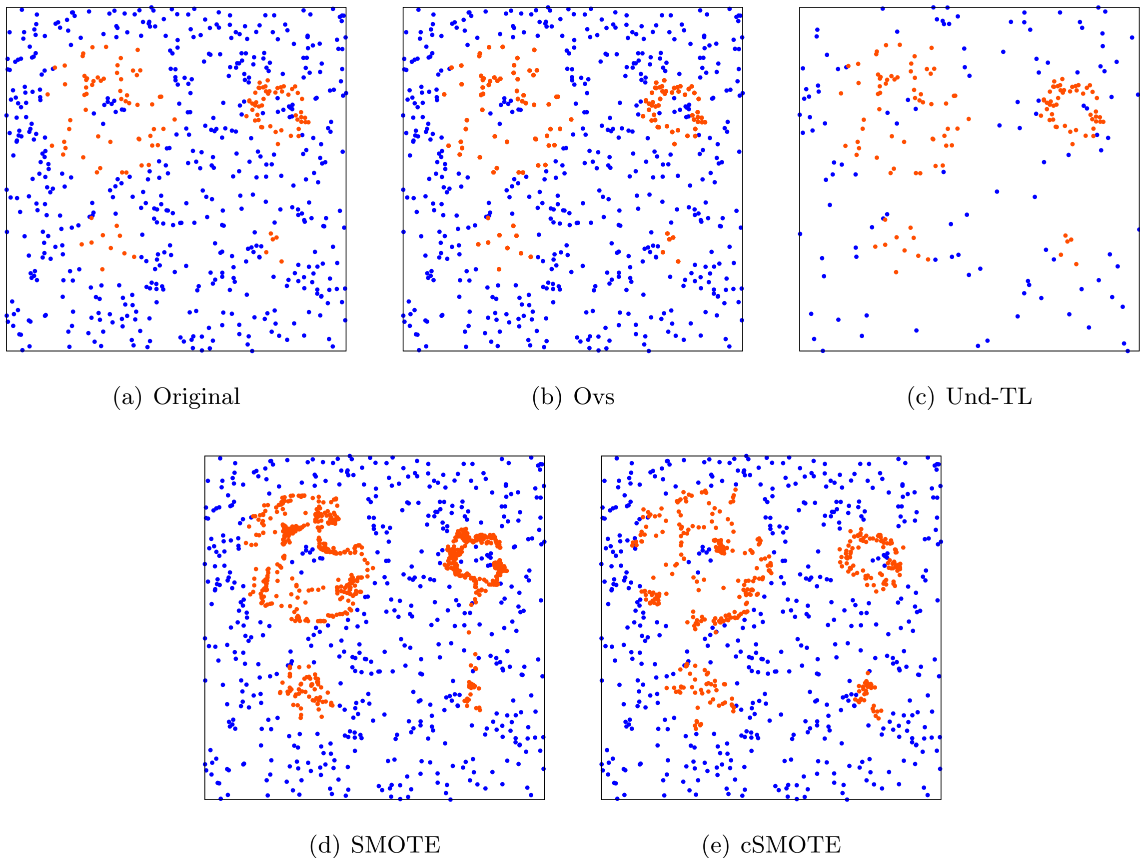


Figure 7.5: Original domain (a) and domains after applying random over-sampling (b), under-stampling with Tomek links (c), SMOTE (d), cSMOTE (e).

that there is not always a direct mapping between the imbalance ratio of the training data set and the imbalance ratio among the sub-solutions or clusters in the solution space.

Figures 7.5(b), 7.5(c), 7.5(d), and 7.5(e) show the modified data set after applying random over-sampling, under-sampling with Tomek links, SMOTE[3], and cSMOTE[4] respectively. Random over-sampling replicates some of the training instances until the data set contains the same number of instances of both classes; for this reason, the resulting domain is apparently equal to the original domain. Under-sampling based on Tomek links removes instances of the majority class that are not close to the class boundary. Note that the final data set contains a considerable lower number of examples. This may be beneficial in terms of run time of the final learner, especially when the original data set consists of a large number of instances; nonetheless, in domains with few instances, it may result in a problem of sparsity. SMOTE creates new instances of the minority class by interpolation. A potential problem of this technique is that, depending on the size of the small disjunct and the chosen k, SMOTE can generate noisy instances if one

[3]SMOTE was configured with k=N=5

[4]cSMOTE was configured with N=10

of the nearest neighbors selected to create a new example belongs to another disjunct. cSMOTE returns a domain that is similar to the one generated by SMOTE. The main difference is that the minority class instances generated by cSMOTE are closer to the class boundary. In the next section, we show the models evolved by the different learners on the original and the re-sampled data sets.

Models Built by the Learners

Figure 7.6 illustrates the domain learned by the five learners on the original and the re-sampled data sets. These results are complemented by table 7.4, which provides the TP rate and TN rate, measured on the training data set, of each learner and domain. The same configurations of the five learners used in the previous sections were employed for these experiments. The domains are depicted by exhaustively testing all the feature space. That is, we generated a test problem with ten million instances distributed uniformly around the feature space, and we used each learner to predict the class of each instance; then, we depicted a point—with a different color depending on the predicted class—in the solution subspace.

Several observations can be drawn from these results. We first analyze the behavior of the different learners on the original data set. In this case, note that all the learners, except for SMO, could discover totally or partially all the sub-concepts of the minority class. IBk (see figure 7.6(k)) is the learner that resulted in the model that is probably closer to the model that a human expert would define from the set of points depicted in Figure 7.5(a). On the other hand, C4.5, XCS, and UCS discovered concepts whose shape resembled rectangles. This is due to the knowledge representation employed by each learner. That is, C4.5, XCS, and UCS (see figures 7.6(a), 7.6(p), and 7.6(u)) used a knowledge representation based on hyper rectangles to discriminate between classes; for this reason, they had more difficulties to approximate curved boundaries. Despite this, note that these learners, and especially UCS, provided an accurate approximation of the class boundary for the given problem. On the other hand, SMO (see figure 7.6(f)) used a linear kernel which failed to discriminate between classes. Note that SMO predicted that any instance in the input space belonged to the minority class. We tried polynomial kernels with higher degree, but significantly better results were not found for this particular artificial problem. The first row of table 7.4 complements these visual results by reporting the TP rate and the TN rate, measured on the training instances, achieved by the five learners. The table shows that UCS predicted all the training instances correctly; IBk, C4.5, and XCS also yielded accurate results.

Let us now examine the results obtained with the re-sampling techniques. The figures show that, in general, all the learners, except for SMO, benefited from re-sampling the training data set. XCS especially benefited from random over-sampling and SMOTE and, to a lower extend, from cSMOTE. However, note that random over-sampling and under-sampling based on Tomek links result in some uncovered regions in the feature space (white regions in the figures). UCS seems to give the best results with cSMOTE. It is worth noting that UCS evolves a maximally accurate model with the original data set.

In general, SMOTE and random over-sampling were the most effective re-sampling techniques. But this general behavior needs to be analyzed carefully. For example, when C4.5 was trained with the domains under-sampled with Tomek links, and re-sampled with SMOTE, or cSMOTE, it was not able to identify the first majority class sub-concept. This behavior can