Data Classification using Genetic Programming

by

Emmanuel Dufourq

Submitted in fulfilment of the academic
requirements for the degree of
Master of Science in the
School of Mathematics, Statistics, and Computer Science,
University of KwaZulu-Natal,
Pietermaritzburg

February 2015

As the candidate's supervisor I have/have not approved this thesis/dissertation for submission.

Name: Professor Nelishia Pillay

____

Signature

____

Date

PREFACE

The experimental work described in this dissertation was carried out in the School of Mathematics, Statistics, and Computer Science, University of KwaZulu-Natal, Pietermartizburg, from February 2013 to February 2015, under the supervision of Professor Nelishia Pillay.

These studies represent original work by the author and have not otherwise been submitted in any form for any degree or diploma to any tertiary institution. Where use has been made of the work of others it is duly acknowledged in the text.

Supervisor: Professor Nelishia Pillay	*Candidate:* Emmanuel Dufourq
______________________________	______________________________
Signature	*Signature*

DECLARATION 1 - PLAGIARISM

I, Emmanuel Dufourq (student number: 208517550) declare that

1. The research reported in this thesis, except where otherwise indicated, is my original research.
2. This thesis has not been submitted for any degree or examination at any other university.
3. This thesis does not contain other persons' data, pictures, graphs or other information, unless specifically acknowledged as being sourced from other persons.
4. This thesis does not contain other persons' writing, unless specifically acknowledged as being sourced from other researchers. Where other written sources have been quoted, then: a. Their words have been re-written but the general information attributed to them has been referenced b. Where their exact words have been used, then their writing has been placed in italics and inside quotation marks, and referenced.
5. This thesis does not contain text, graphics or tables copied and pasted from the Internet, unless specifically acknowledged, and the source being detailed in the thesis and in the References sections.

Candidate: Emmanuel Dufourq

Signature

DECLARATION 2 - PUBLICATIONS

DETAILS OF CONTRIBUTION TO PUBLICATIONS that form part and/or include research presented in this thesis

- Publication 1: E. Dufourq and N. Pillay, "Incorporating Adaptive Discretization into Genetic Programming for Data Classification," *in proceedings of the 2013 World Congress on Information and Communication Technologies (WICT 2013)*, pp. 127-133, 2013
- Publication 2: E. Dufourq and N. Pillay, "A Comparison of Genetic Programming Representations for Binary Data Classification," *in proceedings of the 2013 World Congress on Information and Communication Technologies (WICT 2013)*, pp. 134-140, 2013
- Publication 3: E. Dufourq and N. Pillay, "A Preliminary Study on the Reuse of Subtrees Within Decision Trees in a Genetic Programming Context for Data Classification," *in proceedings of the 2013 World Congress on Information and Communication Technologies (WICT 2013)*, pp. 287-292, 2013
- Publication 4: E. Dufourq and N. Pillay, "Hybridizing Evolutionary Algorithms for Creating Classifier Ensembles," *in proceedings of the 2014 Sixth World Congress on Nature and Biologically Inspired Computing (NaBIC 2014)*, pp. 84-90, 2014

Supervisor: Professor Nelishia Pillay	*Candidate:* Emmanuel Dufourq
______________________	______________________
Signature	*Signature*

Abstract

Genetic programming (GP), a field of artificial intelligence, is an evolutionary algorithm which evolves a population of trees which represent programs. These programs are used to solve problems. This dissertation investigates the use of genetic programming for data classification. In machine learning, data classification is the process of allocating a class label to an instance of data. A classifier is created in order to perform these allocations. Several studies have investigated the use of GP to solve data classification problems. These studies have shown that GP is able to create classifiers with high classification accuracies. However, there are certain aspects which have not previously been investigated.

Five areas were investigated in this dissertation. The first was an investigation into how discretisation could be incorporated into a GP algorithm. An adaptive discretisation algorithm was proposed, and outperformed certain existing methods. The second was a comparison of GP representations for binary data classification. The findings indicated that from the representations examined (arithmetic trees, decision trees, and logical trees), the decision trees performed the best. The third was to investigate the use of the encapsulation genetic operator and its effect on data classification. The findings revealed that an improvement in both training and test results was achieved when encapsulation was incorporated. The fourth was an investigative analysis of several hybridisations of a GP algorithm with a genetic algorithm in order to evolve a population of ensembles. Four methods were proposed and these methods outperformed certain existing GP and ensemble methods. Finally, the fifth area was to investigate an ensemble construction method for classification. In this approach GP evolved a single ensemble. The proposed method resulted in an improvement in training and test accuracy when compared to the standard GP algorithm.

The methods proposed in this dissertation were tested on publicly available data sets, and the results were statistically tested in order to determine the effectiveness of the proposed approaches.

Acknowledgements

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

I would like to thank the Centre for High Performance Computing for granting access to their resources.

I would also like to thank my supervisor, Professor Nelishia Pillay, for her guidance, as well as my family and friends who have encouraged and supported me, especially my parents. I would like to thank the technical staff from the School of Mathematics, Statistics and Computer Science for their support and for enabling me to perform my simulations.

Contents

PREFACE i

DECLARATION 1 - PLAGIARISM ii

DECLARATION 2 - PUBLICATIONS iii

Abstract iv

Acknowledgements v

Contents vi

List of Figures xii

List of Tables xv

List of Algorithms xx

1 Introduction 1

1.1 Purpose of the Study 1
1.2 Objectives 1
1.3 Contributions 3
1.4 Dissertation Layout 4

2 Genetic Programming 6

2.1 Introduction 6
2.2 Introduction to Genetic Programming 6
2.3 Overview of the Generational GP Algorithm 7
2.4 Terminal Set 8
2.5 Function Set 9
2.6 Tree Based GP 9

2.7 Initial Population Generation 10
2.7.1 Full method 11
2.7.2 Grow method 12
2.7.3 Ramped half and half 12
2.8 Fitness 14
2.8.1 Fitness cases 14
2.8.2 Fitness functions 15
2.9 Selection Methods 16
2.9.1 Fitness proportionate selection 17
2.9.2 Tournament selection 18
2.10 Genetic Operators 19
2.10.1 Reproduction 20
2.10.2 Mutation 20
2.10.3 Crossover 20
2.11 Termination 21
2.12 Strongly-Typed GP 22
2.13 GP Control Models 22
2.14 Modularisation 23
2.14.1 Encapsulation 24
2.14.2 Compression 25
2.15 GP and Bloat 26
2.16 Strengths and Weaknesses of GP 26
2.16.1 Strengths 26
2.16.2 Weaknesses 26
2.17 Conclusion 27

3 Data Classification 29
3.1 Introduction 29
3.2 Introduction to Data Classification 29
3.3 Definitions 30
3.3.1 Instance 30
3.3.2 Attribute 31
3.3.3 Class 32
3.3.4 Data set 32
3.3.5 Class balance 33
3.3.6 Classifier 33
3.4 Performance Measures 34
3.4.1 Confusion matrix 34
3.4.2 Sensitivity and specificity 35

3.4.3 Receiver operating characteristics 36
3.5 Evaluating Classifiers 37
3.5.1 Train/test split 38
3.5.2 K-fold cross-validation 38
3.5.3 Leave-one-out 39
3.5.4 Bootstrapping 39
3.6 Previous Work on Data Classification 40
3.6.1 K-nearest neighbour 40
3.6.2 Decision trees 41
3.6.3 Artificial neural networks 43
3.6.4 Naïve bayes 43
3.6.5 Evolutionary algorithms 44
3.7 Active Research Areas in Data Classification 46
3.7.1 Feature selection 46
3.7.2 Missing values 47
3.7.2.1 Discarding missing values 48
3.7.2.2 Imputation 48
3.7.2.3 Missing values and decision trees 48
3.7.3 Ensemble classifiers 49
3.7.4 Discretisation 51
3.8 Software 53
3.9 Conclusion 54

4 GP and Data Classification 56
4.1 Introduction 56
4.2 GP and Decision Trees 56
4.2.1 Advantages and disadvantages of GP decision trees 60
4.2.2 Summary of the findings 61
4.3 GP and Arithmetic Trees 62
4.3.1 Binary classification 62
4.3.2 Multiclass classification 68
4.3.3 Advantages and disadvantages of GP arithmetic trees 74
4.3.4 Summary of the findings 76
4.4 GP and Logical Trees 77
4.4.1 Advantages and disadvantages of GP logical trees 80
4.4.2 Summary of the findings 81
4.5 GP and Other Representations 81
4.6 GP and Ensemble Classifiers 83
4.6.1 Strengths and weaknesses of GP ensembles 88

4.6.2 Summary of the findings 88
4.7 Strengths and Weaknesses of Applying GP to Data Classification . 89
4.7.1 Strengths 89
4.7.2 Weaknesses 90
4.8 Conclusion 91
4.8.1 GP for data classification 91
4.8.2 GP representations for data classification 92
4.8.3 GP discretisation for data classification 94
4.8.4 GP encapsulation for data classification 94
4.8.5 GP ensembles for data classification 94

5 Methodology 96
5.1 Introduction 96
5.2 Addressing the objectives 96
5.3 Statistical testing 98
5.4 Data Sets 99
5.4.1 Characteristics of data sets for data classification problems . 99
5.4.2 Binary data sets 100
5.4.3 Multiclass data sets 106
5.4.4 Rationale behind the selected data sets 111
5.5 GP System 113
5.6 Performance Measures 114
5.7 Technical Specifications 114
5.8 Conclusion 114

6 Adaptive Discretisation for GP 116
6.1 Introduction 116
6.2 Proposed Discretisation Methods for GP 116
6.2.1 Equal Width Intervals (EWI) 118
6.2.2 GP Evolved Intervals (GPEI) 119
6.3 Experimental Setup 123
6.3.1 Data sets 124
6.3.2 GP parameters 125
6.4 Conclusion 125

7 GP Representations for Binary Classification 126
7.1 Introduction 126
7.2 GP Representations for Binary Classification 126
7.2.1 Arithmetic trees 126
7.2.2 Decision trees 127

7.2.3 Logical trees 128
7.3 Experimental Setup 130
7.3.1 Data sets 131
7.3.2 GP parameters 131
7.4 Conclusion 132

8 GP Encapsulation for Data Classification 133
8.1 Introduction 133
8.2 Incorporating Encapsulation into GP for Data Classification 133
8.2.1 Decision trees and encapsulation 133
8.2.2 Maintaining the most called subtrees 136
8.3 Experimental Setup 138
8.3.1 Data sets 139
8.3.2 GP parameters 139
8.4 Conclusion 140

9 Hybridising Evolutionary Algorithms 141
9.1 Introduction 141
9.2 Proposed Hybridisation of GP and GA 141
9.2.1 GA encoding 143
9.2.2 GA run after the last GP generation (GA-at-end) 144
9.2.3 GA run after each GP generation (GA-after-each-gen) 145
9.2.4 GA with hill climbing (GA-with-HC) 146
9.2.5 Steady state GA (SSGA-GP) 146
9.3 Experimental Setup 150
9.3.1 Data sets 150
9.3.2 GP and GA parameters 150
9.4 Conclusion 151

10 GP Ensemble Construction 152
10.1 Introduction 152
10.2 Proposed Ensemble Construction 152
10.2.1 Selecting a tree to add to the ensemble 153
10.2.2 Ensemble evaluation 154
10.2.3 Evaluating the GP trees using weights 155
10.2.4 Updating the weights 158
10.3 Experimental Setup 160
10.3.1 GP parameters 161
10.3.2 Data sets 161
10.4 Conclusion 162

11 Results and Discussion **163**
11.1 Introduction 163
11.2 GP Discretisation 163
11.3 GP Representations for Binary Classification 168
11.4 GP Encapsulation 173
11.5 Hybridisation of GA and GP 176
11.6 GP Ensemble Construction 185
11.7 Conclusion 191

12 Conclusions and Future Work **192**
12.1 Objective 1 - GP Discretisation 192
12.2 Objective 2 - GP Representations for Binary Classification 193
12.3 Objective 3 - GP Encapsulation 194
12.4 Objective 4 - Hybridising GA and GP 194
12.5 Objective 5 - GP Ensemble Construction 195
12.6 Conclusion 195

Bibliography **197**

Appendices **212**

A User Manual **213**
A.1 Program Requirements 213
A.2 Starting the Program 214
A.3 Selecting an Experiment to Run 214
A.4 Starting an Experiment 215
A.4.1 Selecting a data set 215
A.4.2 Executing the experiment 216

List of Figures

2.1 Illustrating nodes with different arity. 9
2.2 GP tree. 10
2.3 Illustrating the depth of each node within a tree. 11
2.4 A tree created using the full method (left), and a tree created using the grow method (right). 12
2.5 Illustrating an initial population created using the ramped half and half method. 13
2.6 Mutation operator, adapted from [1]. 20
2.7 Crossover operator [2]. 21
2.8 If-Then-Else function. 22
2.9 Encapsulation operator, adapted from [3]. 24
2.10 Compression operator, adapted from [4]. 25

3.1 The classification process. 30
3.2 A sample data set. The weather data set, adapted from [5]. 31
3.3 An example of a ROC graph, adapted from [6]. 37
3.4 10-fold cross-validation, adapted from [7]. 39
3.5 Example of a neuron. 43
3.6 Example of an ensemble. 49

4.1 Axis parallel and oblique decision trees, along with graphs illustrating how the data is partitioned in the two representations. The graphs do not represent the partitioning of the data for the corresponding trees. 57
4.2 *CheckCondition2Vars*, function for an oblique decision tree. 59
4.3 Oblique tree used in the study of Shali *et al.* [8]. 60
4.4 Axis parallel decision tree. 61
4.5 An arithmetic tree. 62

4.6 Illustrating how to map the output of a GP tree onto two classes using a threshold value. In this figure, the threshold is 0.5. 63
4.7 An example of a tree created using class enumeration. 70
4.8 Representing a categorical attribute as a numerical one. 75
4.9 The *IN* function proposed by De Falco *et al.* [9]. 79

5.1 Difference in the number of attributes and instances in the binary data sets. 111
5.2 Difference in the number of attributes and instances in the multiclass data sets. 112
5.3 Difference in the number of classes in the multiclass data sets. . . . 112

6.1 Example of a GP tree using a decision tree representation. 117
6.2 Intervals created using EWI. 118
6.3 Intervals created using GPEI. 120
6.4 Illustrating the alter interval GO. The algorithm selected attribute 3 (highlighted in grey) for modification. 123
6.5 Illustrating the alter interval GO. The intervals for attribute 3 were altered which resulted in three new intervals. 123

7.1 Arithmetic tree representation for GP. 127
7.2 Decision tree representation for GP. 128
7.3 Logical tree representation for GP. 128
7.4 The between GP operator for logical tree representations. 129
7.5 Creating the *OUT* function by preceding the between with a *NOT* operator. 130

8.1 Pruning trees and adding encapsulated terminals at the leaves. . . . 135
8.2 Evaluating a tree with an encapsulated terminal. 135

9.1 Illustrating an ensemble. 144
9.2 Example of a chromosome created using *SSGA-GP*. The genes correspond to GP trees which have been added from different GP generations. 147

10.1 Ensemble with corresponding trees at each index. 153

11.1 Comparing GPEI and EWI in terms of training and test accuracy (%). 166
11.2 Illustrating the average training accuracy (%) for the different representations. 170
11.3 Illustrating the average test accuracy (%) for the different representations. 172

11.4 Comparison between the average test results for the ensembles and standard GP. 188

A.1 Main menu. 213
A.2 GP Arithmetic Representation menu. 215
A.3 Popup message which appears at the end of the run. 216

List of Tables

2.1 Fitness case for the even-3-parity problem. 15
2.2 Illustrating fitness proportionate selection. 17

3.1 Class output from figure 3.2, and the output for the simplistic classifier. 34
3.2 Confusion matrix, extracted from [10]. 35
3.3 Illustrating accuracy paradox - classifier 1 confusion matrix. 35
3.4 Illustrating accuracy paradox - classifier 2 confusion matrix. 35

5.1 *Pima Indians* data set characteristics. 100
5.2 *Sonar* data set characteristics. 101
5.3 *WDBC* data set characteristics. 101
5.4 *Parkinsons* data set characteristics. 102
5.5 *Mammographic* data set characteristics. 102
5.6 *Ionosphere* data set characteristics. 103
5.7 *Spectf* data set characteristics. 103
5.8 *Climate* data set characteristics. 104
5.9 *Fertility* data set characteristics. 104
5.10 *Monk2* data set characteristics. 105
5.11 *TTT* data set characteristics. 105
5.12 *Balance* data set characteristics. 106
5.13 *Car* data set characteristics. 106
5.14 *Glass* data set characteristics. 107
5.15 *Ecoli* data set characteristics. 107
5.16 *Zoo* data set characteristics. 108
5.17 *Iris* data set characteristics. 108
5.18 *Wine* data set characteristics. 109
5.19 *Yeast* data set characteristics. 109

5.20 *Vehicle* data set characteristics. 110
5.21 *Soybean* data set characteristics. 110

6.1 Sample data for an attribute. 119
6.2 Experiments conducted and their different combination of parameters. 124
6.3 Selected data sets for the adaptive discretisation experiments. . . . 124
6.4 GP parameters. 125

7.1 Characteristics of the five experiments comparing different GP representations for binary data classification. 130
7.2 Selected binary data sets for the GP representation experiments. . . 131
7.3 GP Parameters for comparison of different GP representations for data classification. 131

8.1 Data sets used for GP encapsulation experiments. 140
8.2 GP parameters used. 140

9.1 Selected data sets for the hybridisation experiments. 150
9.2 GP and GA parameters for the hybridisation experiments. 151

10.1 Possible functions for $g(x_i)$. 156
10.2 Different weight values and their corresponding value for $g(x_i)$. . . . 157
10.3 Illustrating how the weights are updated. Let the correct class for some instance of data be "B". 160
10.4 GP parameters used. 161
10.5 Data sets used for ensemble construction experiments. 162

11.1 Experiment IDs for the discretisation methods. 164
11.2 Training accuracy (%) results for the different GP discretisation methods. For each data set, the best result is highlighted in bold, and was statistically tested with every other result. 164
11.3 Test accuracy (%) results for the different GP discretisation methods. For each data set, the best result is highlighted in bold, and was statistically tested with every other result. 165
11.4 Average size (number of nodes) of the best GP individuals for each method. The smallest size for each data set is highlighted in bold. . 167
11.5 Comparison between GPEI with arity 2 and other state-of-the-art discretisation methods. 168

11.6 Training accuracy (%) results for the different representations. For each data set, the best result is highlighted in bold. A "**" indicates that the best result for the data set is statistically significant when compared to that result. A "†" indicates a statistically insignificant result when compared to the best result. 169
11.7 Test accuracy (%) results for the different representations. For each data set, the best result is highlighted in bold. A "**" indicates that the best result for the data set is statistically significant when compared to that result. A "†" indicates a statistically insignificant result when compared to the best result. 170
11.8 Variance amongst the different methods for each data set. 172
11.9 Average size (number of nodes) of the different representations. The smallest result for each data set is highlighted in bold. 173
11.10 Training accuracy (%) results for standard GP and the two proposed encapsulation methods. For each data set, the best result is highlighted in bold, and the results for the encapsulation methods were statistically tested with the results obtained by standard GP. A "**" indicates that the result is statistically significant when compared to the result obtained by standard GP without encapsulation. A "†" indicates a stastically insignificant result compared to standard GP without encapsulation. 174
11.11 Test accuracy (%) results for standard GP and GP with the two proposed encapsulation methods. For each data set, the best result is highlighted in bold, and the results for the encapsulation methods were statistically tested with the results obtained by standard GP. A "**" indicates that the result is statistically significant when compared to the result obtained by standard GP without encapsulation. A "†" indicates a statically insignificant result compared to standard GP without encapsulation. 175
11.12 Comparison between the number of encapsulated terminals which were present in the best GP individuals for the two encapsulation methods. All the results obtained by encapsulation with maintained list were statistically significant compared to when the list was not used, this is denoted by "**". 176
11.13 Average training classification accuracy (%) for *GA-at-end*. For each data set, the best result is highlighted in bold, and the best ensemble result is statistically tested against the standard GP result. 177

11.14 Average test classification accuracy (%) for *GA-at-end*. For each data set, the best result is highlighted in bold, and the best ensemble result is statistically tested against the standard GP result. 178
11.15 Average training classification accuracy (%). The best result for each data set is highlighted in bold. The result for each ensemble method was statistically compared to the result obtained by standard GP. Between each pairwise comparison, a "**" denotes that the higher result is statistically significant. A "†" denotes statistical insignificance. . 180
11.16 Average test classification accuracy (%). The best result for each data set is highlighted in bold. The result for each ensemble method was statistically compared to the result obtained by standard GP. Between each pairwise comparison, a "**" denotes that the higher result is statistically significant. A "†" denotes statistical insignificance. . . . 180
11.17 Comparison of the average size of the ensembles of *GA-after-each-gen* and *GA-with-HC*. For each data set, the larger result was statistically compared to the other result. A "**" indicates that the result is statistical larger than the other method. A "†" denotes a statistically insignificant result. 182
11.18 Training accuracy (%) comparison between *GA-at-end* and other methods found in literature. 183
11.19 Test accuracy (%) comparison between *GA-at-end* and other methods found in literature. 183
11.20 Training accuracy (%) comparison between the proposed ensemble methods and other methods found in literature. 184
11.21 Test accuracy (%) comparison between the proposed ensemble methods and other methods found in literature. 184
11.22 Training accuracy (%) results for the different ensembles and standard GP. The best result for each data set is highlighted in bold. For each data set, *ensemble5* was statistically compared to standard GP 200 generations, *ensemble* 7 to standard GP 280 generations, and *ensemble9* to standard GP 360 generations. 186
11.23 Test accuracy (%) results for the different ensembles and standard GP. The best result for each data set is highlighted in bold. For each data set, *ensemble5* was statistically compared to standard GP 200 generations, *ensemble* 7 to standard GP 280 generations, and *ensemble9* to standard GP 360 generations. 187
11.24 Training accuracy (%) comparison between the proposed ensemble construction method and other methods found in literature. 189

11.25 Test accuracy (%) comparison between the proposed ensemble construction method and other methods found in literature. 189
11.26 Comparison between the hybrid ensemble methods with the ensemble construction methods. The best training and test result for each data set is highlighted in bold. For each training and test set, a "**" denotes that the best result is statistically significant when compared to the other result, and a "†" denotes statistical insignificance. . . . 190

List of Algorithms

2.1 Generational GP algorithm. 7
2.2 Pseudocode for tournament selection. 18
2.3 Steady state GP. 23

6.1 Pseudocode for creating an attribute node using GPEI. 121
6.2 Alter interval genetic operator. 122

8.1 Pseudocode for encapsulation in the context of data classification. . . 134
8.2 Pseudocode of proposed GP algorithm with encapsulation. 136
8.3 Pseudocode for initialising the maintained list. 137
8.4 Pseudocode for updating the maintained list. 138
8.5 Pseudocode which selective encapsulation uses for adding terminals to the GP trees. 139

9.1 Pseudocode of genetic algorithm for ensemble representation. 142
9.2 Pseudocode for GA mutation. 142
9.3 Pseudocode for GA one point crossover. 143
9.4 Pseudocode for *GA-at-end*. 145
9.5 Pseudocode for *GA-after-each-gen*. 146
9.6 Modified mutation GA operator. 148
9.7 Pseudocode for *SSGA-GP*. 149

10.1 Pseudocode for adding a tree to the ensemble. 154
10.2 Pseudocode for evaluating a GP tree. 158
10.3 Pseudocode for updating the weights. 159

Chapter 1

Introduction

1.1 Purpose of the Study

Data classification techniques deal with creating classifiers which allocate a label to data. These techniques use the existing data in order to produce these classifiers, and once created, the classifiers are applied to new unseen data. An application area which illustrates the purpose of data mining is the allocation of credit scores to individuals. A credit score represents a numerical value which denotes the risk associated to lending finance to an individual. Assume that a large database containing relevant information about consumers and their credit behaviour is maintained, and that through the use of data classification, a classifier is created. Provided a suitable classifier is created, this could assist credit providers in assessing the risk involved in providing financial support to new customers.

Various techniques have been applied to data classification, including statistical methods such as Bayesian and regression methods, and evolutionary algorithms such as genetic algorithms and genetic programming algorithms. Genetic programming is inspired by nature. It has often been used to solve data classification problems and has been successful in producing good classifiers [11]. Despite the large number of studies which have addressed data classification by using genetic programming, it is apparent from the literature that there are still certain areas of research which have not been explored. These areas represent the rationale behind this dissertation and are listed as the objectives in the following section.

1.2 Objectives

The primary objective of this dissertation is to develop, and evaluate the performance of genetic programming in the domain of data classification, and to investigate certain areas of research which have not been previously addressed. The

objective of this dissertation is not to propose algorithms which will outperform all the existing methods found in the literature, but instead, the objective is to conduct an investigation on how several proposed genetic programming methods perform on data classification problems. Tied in with the primary objective previously stated, this dissertation will conduct a thorough analysis of the related literature on genetic programming and data classification. Five objectives were formulated for this dissertation and are listed below:

- Objective 1: Incorporating discretisation into genetic programming.

  To determine and compare the performance of several proposed methods which incorporate discretisation into a genetic programming algorithm. The proposed methods will be applied to data sets in which the attributes are made up of real values.

- Objective 2: Genetic programming representations for binary data classification.

  To determine the primary representations for genetic programming and data classification, and to compare their performance in the context of binary data classification.

- Objective 3: Creating an encapsulation genetic operator for data classification.

  To incorporate, investigate and evaluate the encapsulation genetic operator in the context of data classification. The objective is to determine whether the performance of a genetic programming algorithm is impacted as a result of incorporating this genetic operator.

- Objective 4: Hybridising evolutionary algorithms for classifier ensembles.

  To propose, implement, and hybridise a genetic algorithm with a genetic programming algorithm and conduct an analysis on variations of this hybridisation in the domain of data classification.

- Objective 5: Creating a genetic programming ensemble construction method.

  To propose and investigate an ensemble construction method which will create genetic programming ensembles. The objective is to determine how a single ensemble can be constructed using genetic programming.

1.3 Contributions

This dissertation makes the following contributions:

1. Discretisation is required when using decision trees and continuous data. There has been no previous work which has incorporated discretisation into the genetic programming algorithm. The findings of this study reveal that discretisation can be successfully incorporated and that the proposed methods achieve good results when compared to existing discretisation methods.

2. The choice of representation is an important decision to make when implementing a genetic program. There has been no previous work comparing the three major representations for binary classification. The findings of this study show that decision trees provided the best overall accuracy; however, any of the three representations can be used in order to achieve good results.

3. Functions are often used when writing a computer program. The encapsulation genetic operator is used to achieve this purpose in the context of genetic programming. There has not been any previous attempt to investigate the encapsulation genetic operator for data classification problems. It was concluded from the results that the encapsulation genetic operator can yield an improvement in accuracy.

4. Based on a thorough investigation of the literature, it was found that ensemble methods produce classification models which obtain better results than non-ensemble approaches. This dissertation proposed four ways for combining a genetic algorithm and genetic programming algorithm in order to improve the accuracy of the classifiers. The proposed methods outperformed the standard genetic programming approach. On certain data sets the proposed methods outperformed other state-of-the-art ensemble approaches.

5. Contribution 4 proposed methods for evolving a population of ensembles. This dissertation made another contribution to ensemble methods by proposing a genetic programming ensemble approach which focuses on creating a single classifier. Weights were associated to the instances of data in order to allocate a level of difficulty to the instances. The findings revealed that the proposed method outperformed the standard genetic programming method. The proposed method further provides an alternative approach to existing boosting algorithms.

1.4 Dissertation Layout

This section provides a summary of the chapters in this dissertation.

Chapter 2 - Genetic Programming

This chapter provides detailed descriptions about genetic programming, and sets the foundation for this dissertation. Each process within the algorithm is thoroughly described and analysed. This chapter also discusses the strengths and weaknesses of genetic programming.

Chapter 3 - Data Classification

This chapter introduces the concept of data classification and describes the relevant terminology. Previous work on data classification is discussed and details regarding active research areas are provided.

Chapter 4 - Genetic Programming and Data Classification

Discussions regarding previous work which have used genetic programming to solve data classification problems are presented in this chapter. The strengths and weaknesses of applying genetic programming to data classification are discussed.

Chapter 5 - Methodology

This chapter describes how the investigation on genetic programming and data classification will be performed, and how each objective will be met. The data sets are described in this chapter.

Chapter 6 - Adaptive Discretisation for Genetic Programming

This chapter presents several algorithms which have been proposed in order to incorporate discretisation into the genetic programming algorithm.

Chapter 7 - Genetic Programming Representations for Binary Data Classification

This chapter proposes an investigation on genetic programming representations for binary data classification.

Chapter 8 - Genetic Programming Encapsulation for Data Classification

Chapter 8 describes how a proposed encapsulation genetic operator can be used in the context of data classification.

Chapter 9 - Hybridising Evolutionary Algorithms

A description of several algorithms for hybridising a genetic algorithm with a genetic programming algorithm is discussed in this chapter. This chapter focuses on evolving a population of ensembles.

Chapter 10 - Genetic Programming Ensemble Construction

This chapter presents an algorithm for evolving a single ensemble using genetic programming.

Chapter 11 - Results and Discussion

Chapter 11 presents and discusses the results obtained by the investigations proposed in chapters 6 to 10. The proposed methods are compared to standard genetic programming and to other existing methods.

Chapter 12 - Conclusions and Future Work

Finally, this chapter provides a summary of the findings from the research presented in this dissertation, and discusses how the objectives presented in chapter 1 have been met. This chapter also describes future work which will be investigated.

Chapter 2

Genetic Programming

2.1 Introduction

This chapter introduces genetic programming and provides details and an analysis on the different aspects of the algorithm.

Section 2.2 introduces genetic programming, this is followed by an overview of the generational genetic programming algorithm in section 2.3. When using genetic programming to solve a problem, a representation has to be chosen. Each representation has its own function and terminal set, these two concepts are discussed in sections 2.4 and 2.5 respectively. The tree based representation is discussed in section 2.6. Three initial population generation methods are described in section 2.7, followed by a discussion on fitness in section 2.8. Two parent selection methods are discussed in section 2.9; this is followed by section 2.10 which describes three commonly used operators to generate offspring. Genetic programming is executed until a certain condition is met, and this is discussed in section 2.11. The concept of strongly-typed genetic programming is discussed in section 2.12. Control models are highlighted in section 2.13. The concept of code reuse and modularisation is discussed in section 2.14. Genetic programs suffer from bloat, which is described in section 2.15. Like other evolutionary algorithms, genetic programming has its own strengths and weaknesses; these are highlighted in section 2.16. Finally, section 2.17 concludes this chapter and summarises the critical aspects of the genetic programming algorithm.

2.2 Introduction to Genetic Programming

Genetic Programming (GP) was first introduced by Koza in 1992 [3] and deals with evolving computer programs using biologically inspired methods. Each program is represented as an individual in a population, and each individual competes for

resources and survival, similar to the analogy of natural species competing for resources such as food. Only the fittest or near fittest individuals survive, and they give birth to new offspring in the hope that these offspring will be able to survive. The process of giving birth to offspring is similar to the concept of genetic operators (GO) which will be further discussed in section 2.10.

GP is stochastic and random in nature, and thus it cannot be guaranteed that a solution to a problem will be found [1], nevertheless, GP has proved successful in numerous application domains and has been used to create programs which are better than those written by human programmers [2].

2.3 Overview of the Generational GP Algorithm

The generational GP algorithm [2] is presented in algorithm 2.1. The first step is to randomly create the initial population of GP individuals. The size of the population is a user defined parameter. The algorithm proceeds into a loop, and each iteration of this loop represents a *generation*. The initial population is evaluated by examining each individual in order to determine if a solution to the problem has been found. If a solution is found, then the algorithm can terminate and output the GP individual which solves the problem. However, if no solution exists within the initial population, the algorithm continues. Four actions are performed during each generation: evaluate the current population, apply the selection methods, apply the genetic operators, and finally update the population.

Algorithm 2.1: Generational GP algorithm.

```
1  begin
2      Randomly create the initial population.
3      repeat
4          Evaluate the population.
5          Apply the selection methods and obtain parents.
6          Apply the genetic operators to the parents, and create offspring.
7          Replace the current population with the offspring.
8      until a solution to the problem has been found, or a termination criteria
       is met;
9  end
10 return The best individual from the population
```

The selection methods are applied to obtain parents for the genetic operators. Once the parents have been obtained, the genetic operators are applied and the new individuals - referred to as the offspring - replace the current population. In each generation of the generational GP algorithm, the entire population is replaced by the offspring. The new population is then evaluated and the process is repeated

until one of the termination criteria are met.

Typically, the termination criteria is met when the maximum number of generations is reached, or once a solution to the problem has been found [3]. The algorithm which has just been described is the *generational control model.* This shall be further discussed in section 2.13. The following sections describe fundamental areas which are related to the GP algorithm.

The even-3-parity problem will be used to assist with the description of certain processes of the GP algorithm. This is a boolean problem (which has 3 input variables; x, y, and z) in which the task is to create a solution that can correctly output whether a given string contains an even or an odd number of true values. The value "true" is represented by a 1, and the value "false" is represented by a 0. Consider the string 000, there are no 1s, thus since there are an even number of 1s, the output is 1. Now consider the string "010", in this case there is an odd number of 1s, thus the output is 0.

2.4 Terminal Set

GP individuals represent candidate solutions to a problem. The problem will typically have a number of input variables which can be used to solve the problem. In GP, the *terminal set* [1, 2] is made up of all the variables which are used to solve the problem. The terminal set can also contain constants; these for example can be integers, strings, characters, or floating point values which have some significance to the problem domain.

Ephemeral random constants are also added to the terminal set. Whenever an ephemeral random constant is added to the leaf node of an individual during the GP execution, its value is randomly generated within a specified range, and this value remains unchanged during the entire evolutionary process [1,3]. For instance, let a be an ephemeral random constant with a range of integer values $[-1, 1]$. Thus, at any point during the evolutionary process, if a is selected from the terminal set, a random integer value in the range of $[-1, 1]$ will be chosen for it, and the value will remain fixed for the duration of the GP execution. Thus, it is possible that when terminal a is selected numerous times, different values are generated. The type and range of the values for an ephemeral random constant are problem dependent.

Finally, any function with an *arity* of zero is included. Arity is the number of arguments which a function has [2]; figure 2.1 illustrates different arity values. For example, consider a *random()* function which generates random integers. Since such a function has an arity of zero, it would be included in the terminal set. These terminals are found at the leaf nodes in tree structures. In the case of the even-3-parity problem, the terminal set will be made up of the three input variables, thus

the set will be $\{x, y, z\}$.

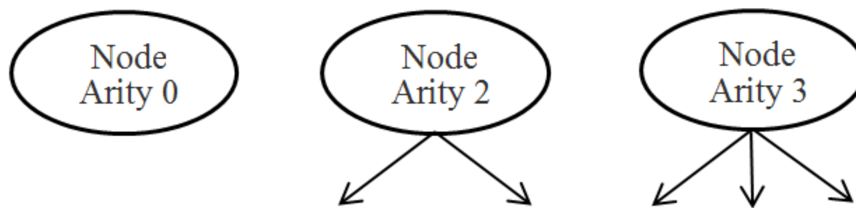


Figure 2.1: Illustrating nodes with different arity.

2.5 Function Set

The *function set* contains all the functions specific to the problem domain which are available to the GP algorithm [2]. Examples of such functions are arithmetic functions, logical operators, loop statements and conditional statements. The function set should be made up of functions which would be useful to the domain. In the case of the even-3-parity problem, since this is a boolean problem, a possible function set would include logical operators, such as {AND, OR, XOR}.

Below is a list of commonly used function sets extracted from [1, 3].

- Arithmetic functions: $\{+, -, \times, /\}$
- Mathematical functions: $\{sin, cos, tan, exp, square\}$
- Logical operators: $\{AND, OR, NOT\}$
- Conditional operators: $\{If - Then - Else\}$

2.6 Tree Based GP

As mentioned in section 2.3, the first step in the GP algorithm is to create the initial population which is made up of GP individuals. These individuals - which are made up of elements from the terminal and function set - need to be represented in some manner. This section will discuss the tree based GP representation.

Trees, also known as syntax trees, are the most commonly used representation for GP, and are made up of one or several nodes. The top most node is referred to as the root, and the bottom most nodes are the leaves. Leaves are usually represented by elements of the terminal set. Tree nodes which are non-leaves are represented by elements from the function set. Trees are commonly output from a pre-order traversal to improve readability; however, they are typically evaluated in an infix order to ensure the correct evaluation of the mathematical or logical expressions

which are represented by the trees. Figure 2.2 illustrates a GP based tree which corresponds to the mathematical expression $max(x + x, x + 3 \times y)$.

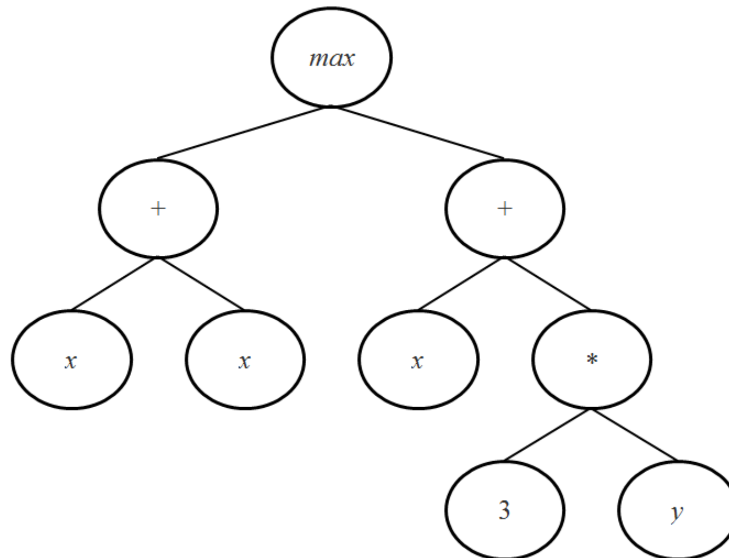


Figure 2.2: GP tree, extracted from [1].

2.7 Initial Population Generation

Once a suitable representation has been decided upon, an initial population can be created. The initial population is referred to as *generation zero*. Several methods exist for the creation of the initial population. Three common methods for creating the initial population are the full method, the grow method, and finally the ramped half and half method. Additional initial population generation methods for GP trees were examined by Luke and Panait [12].

In order to maintain genetic diversity no duplicates should be created when initialising the population; this is done in order to represent as much of the program space as possible. Koza describes duplicate individuals in generation zero as "*unproductive deadwood*" [3].

If only a small portion of the program space is being represented then the GP algorithm may converge prematurely to a local optimum. However, if a sufficient amount of the program space is represented then there is a greater chance of converging to the global optimum. Incidentally, if the program space being represented is too large, then this can hinder GP's ability to converge towards the global optimum.

Before the initial population generation is described, the term depth needs to be defined. The *depth* of a node is the distance from the root node to that particular node. The root node has a depth of 1. The *maximum depth* of a tree is the distance

- in terms of the nodes - from the root of a tree to the bottom-most leaf. From figure 2.3, the tree has a maximum depth of 4. When creating the initial population for tree structures, a maximum depth must be specified in order to limit the size of the trees when they are created.

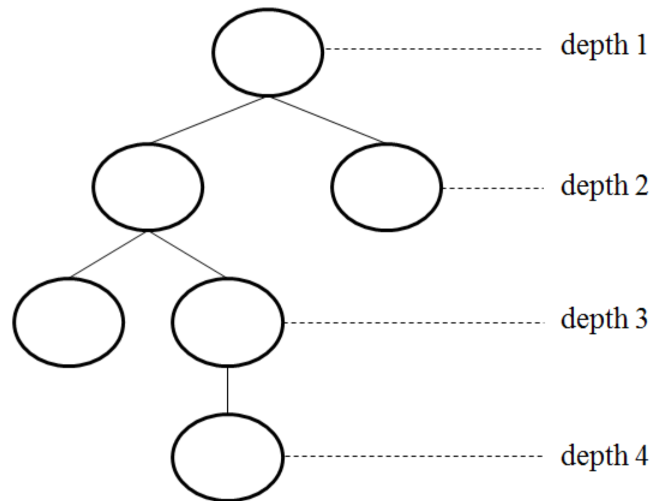


Figure 2.3: Illustrating the depth of each node within a tree.

2.7.1 Full method

Each individual created by the *full method* has the maximum possible size, in the sense that the distance from the root node to each leaf is equal to the maximum tree depth [3]. When creating a tree using the full method, provided that the maximum depth has not been reached, an element from the function set is always selected. The leaves are made up of elements of the terminal set. By using the full method, the distance between all of the leaves and the root is equal to the maximum depth, and consequently, all the trees in the population have the same depth. In figure 2.4, the tree on the left illustrates the result of applying the full method when creating a tree.

Depending on the arity of the functions, the trees might not all have the same number of nodes. For instance, if the full method is used to create a tree using functions of arity 2, then this particular tree will have less nodes than a tree created using functions of arity 3. Regardless of the arity of the functions, all the leaves within all of the trees will be at the same depth.

A consequence of the full method is that a large quantity of the trees in the initial population will have a similar structure. Consequently, the initial population is less diverse due to the similarity in tree structures. This lack of diversity can result in the GP algorithm searching a restricted area of the program space, and

thus hindering the performance of the algorithm.

2.7.2 Grow method

The *grow method* creates trees of different shapes and sizes [3]. When creating a tree using this method, at each depth an element from the terminal set or from the function set can be randomly selected. However, at the maximum depth, only an element from the terminal set can be selected. In figure 2.4, the tree on the right illustrates a tree which was created using the grow method.

This method benefits from the fact that trees of different sizes are created which will result in greater diversity as opposed to the full method. Although the grow method results in greater diversity, the method also suffers from the randomness involved in creating the trees, which is highlighted by Poli *et al.* [1]. Consider the 6-multiplexer problem; this problem has 6 input variables. In order to solve this problem using a tree representation, all of the variables should be used. It is possible, due to the randomness involved when creating the nodes, that the trees created using the grow method are not sufficiently large enough to make use of all the variables.

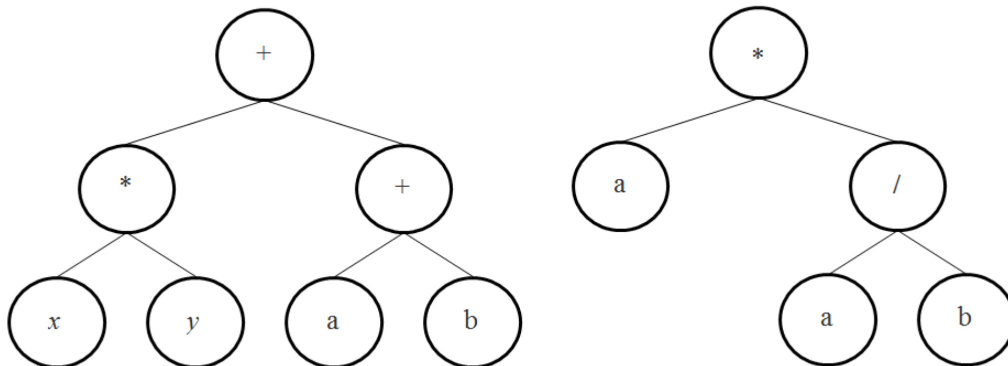


Figure 2.4: A tree created using the full method (left), and a tree created using the grow method (right).

2.7.3 Ramped half and half

The *ramped half and half* method creates half of the initial population using the grow method, and the other half using the full method [1]. Let *size* denote the population size, and let d denote the maximum depth. Thus at each depth, a total of $\frac{size}{d-1}$ individuals are to be created. Of these $\frac{size}{d-1}$ individuals, half of those must be created using the grow method, and the other half using the full method. This method of initial population generation has been proven successful and is commonly used [1]. The reason behind this is that genetic diversity is maintained since trees

of different shapes and sizes are created [3].

For example consider a population size of 6 and a maximum depth of 4 which is illustrated in figure 2.5. Half of the population must be created using the grow method, and the other half using the full method. Thus, since $size = 6$ and $d = 4$ a total of 2 individuals are to be created at each depth. Thus, the individuals are created as follows:

- At depth 2: one tree using the grow method, one tree using the full method.
- At depth 3: one tree using the grow method, one tree using the full method.
- At depth 4: one tree using the grow method, one tree using the full method.

From the figure, the trees on the left represent those created using the grow method, and the trees on the right were created using the full method. At a depth of 4, the tree created using the grow method is much smaller than the one created using the full method. This is because after the root was set to a function node, the next created node was randomly assigned to an element of the terminal set.

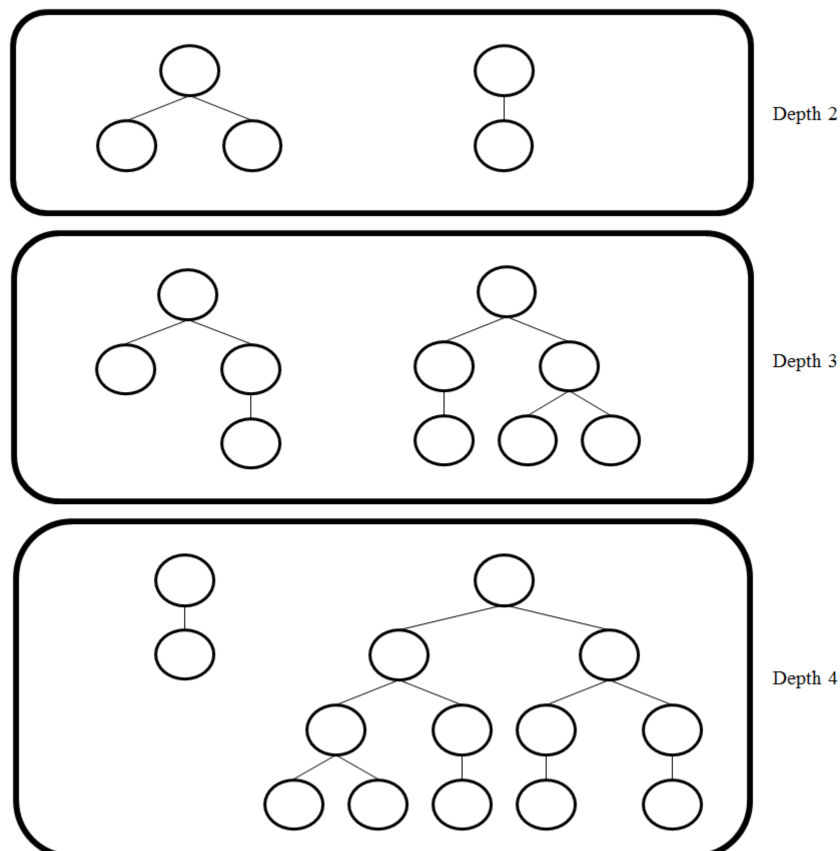


Figure 2.5: Illustrating an initial population created using the ramped half and half method.

2.8 Fitness

In this section the concept of fitness is introduced. This is an essential part of the GP algorithm and other evolutionary algorithms. The idea of fitness and fitness functions is used to drive the algorithm towards the global optima. These are usually problem dependent formulations and there are a vast amount of approaches present in literature which deal with the concept of fitness in different ways. In this section the fundamental ideas are presented.

2.8.1 Fitness cases

The ultimate goal of an evolutionary algorithm, including the goal of GP is to find a solution to a problem. In order to evaluate an individual in the population, *fitness cases* are required. The fitness cases represent input and output pairs to the problem domain [2]. For instance, assume GP is used to evolve a mathematical function given some values for the input variables, and their corresponding output value. In this case the input and output pairs are known however, the actual mathematical expression is unknown, and the goal of a researcher would be to create an expression which models the data. In this scenario the input variables and their output value represent the fitness cases.

Banzhaf *et al.* state that a "*machine learning system goes through the training set [or fitness cases] and attempts to learn from the examples*" [2]. Hence GP must evolve a program which is able to map the input variables onto the output variables of those examples specified as the fitness cases.

The fitness cases must represent a sufficient amount of the problem domain since GP will evolve solutions which meet as many of these fitness cases as possible. Thus, if the problem domain contains an infinite number of cases, a sufficient amount of these cases must be included, as well as those of particular interest to the problem. For example, consider generating fitness cases in which GP is being used to create a solution to the factorial problem. For input values greater than 1, the factorial of the input is computed in the same manner. Two particular cases which have to be included in the fitness cases are when the input is 0 or 1, as these both generate an output of 1.

Table 2.1 illustrates an example of fitness cases for the even-3-parity problem. In this situation the input variables are x, y, and z, and the output variable is also illustrated. Based on the fitness cases in the example, the goal of GP is to evolve an individual which will correctly map the three input variables onto the output variable for each fitness case.

x	y	z	output
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 2.1: Fitness case for the even-3-parity problem.

2.8.2 Fitness functions

As previously stated, the fitness cases are made up of input/output pairs, however, a numerical measure is required in order to determine how good an individual actually is. In other words, a measure is required to determine how close/far an individual is from a perfect solution, and this measure can be used to compare one individual to another. *Fitness functions* are used to obtain a *fitness* value for each GP individual. Banzhaf *et al.* define fitness as a *"measure used by GP during simulated evolution of how well a program has learned to predict the output(s) from the input(s)"* [2]. Additionally, the fitness value is used to determine which areas of the program space are useful in solving the problem [1]. The raw fitness denotes the fitness of GP individuals with respect to the problem domain [3]. Thus the raw fitness for one problem may be different to the raw fitness in another.

The *number of hits* is the number of fitness cases for which an individual produces the same output as the fitness cases. In certain cases the raw fitness is equal to the number of hits. From this formulation, the greater the number of hits, the better an individual is.

Since in certain problem domains, a lower raw fitness may represent a better individual, and in other domains, a higher raw fitness represents a better individual; an alternative measure is to use the standardised fitness. A lower numerical value denotes a better individual when the *standardised fitness* is applied [3]. One way to formulate the standardised fitness is to use the raw fitness in the following way: if the problem is a minimisation problem (in which case a lower raw fitness represents a better individual), then the standardised fitness is equal to the raw fitness. However if the problem is a maximisation problem, then a higher raw fitness represents a better individual, and thus the standardised fitness can be formulated as $raw_{max} - raw_{fitness}$, where raw_{max} denotes the maximum possible raw fitness value for the problem [3].

Fitness functions can have either have a single objective, or multiple objectives.

Fitness functions play an important role in guiding the GP algorithm towards the global optimal solution. The function needs to be correctly defined in such a way so that individuals representing weaker areas of the program space are not confused to be stronger areas and vice versa. Additionally, the fitness function must be able to determine near-solutions as well as solutions to the problem. Examples of fitness functions for solving different problems are listed below:

- In a symbolic regression problem, a suitable fitness function would be the cumulative absolute value in difference between the correct output and the output of the individual across the fitness cases [3].
- In the case of data mining, one could define a fitness function which measures the percentage of instances which GP is able to correctly recognise from the fitness cases [13].
- In solving the artificial ant problem, a fitness function could be one that determines the number of pieces of food which has been picked up [3].
- In solving the 11 multiplexer problem, the raw fitness was defined as the number of hits [3].
- In the game Ms. Pac-Man, the fitness function favoured higher scores by adding the scores for the pills, power pills and ghosts eaten [14].

Multi-objective fitness functions are made up of two or more different objectives. A GP algorithm which implements this approach is referred to as a *multi-objective genetic program* (MOGP). A MOGP algorithm deals with simultaneously finding optimal solutions in order to meet as many of the objectives as possible [1].

A simple example of a multi-objective fitness function for a maximisation problem is one which computes the number of hits, as well as the size of an individual. For instance, $fitness = raw_{fitness} - tree_{size}$, where $tree_{size}$ represents the number of nodes within a tree. Thus, GP would favour those trees with a higher number of hits which simultaneously represent smaller trees.

2.9 Selection Methods

Selection methods allow GP to select the individuals - known as parents - which are used to create offspring. There are numerous selection methods; however, this section reviews two common methods, the fitness proportionate and the tournament selection methods.

2.9.1 Fitness proportionate selection

Fitness proportionate selection [3] is more computationally expensive when compared to tournament selection as it requires additional calculations to be performed for each individual in the population. Firstly the *adjusted fitness* has to be calculated. Once obtained, a *normalised fitness* has to be calculated. The reasons for doing so is that it results in fitness values between 0 and 1, and the sum of all the fitness values adds to 1. The normalised fitness is then multiplied by the number of individuals in the population to determine the number of occurrences of each individual in a *mating pool*. An individual is randomly selected from the mating pool and is returned to be a parent.

- Adjusted fitness:

$$a(i,t) = \frac{1}{1+s(i,t)}$$

where $s(i,t)$ corresponds to the standardised fitness. The standardised fitness transforms the raw fitness of an individual in such a way that a lower value represents a fitter individual. In the situation where a lower value represents a fitter individual, *i.e.* a minimisation problem, then $s(i,t) = r(i,t)$ where $r(i,t)$ is the raw fitness. Otherwise, if the problem is a maximisation problem, then $s(i,t) = r_{max} - r(i,t)$, where r_{max} is the maximum possible fitness for the problem.

- Normalised fitness:

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)}$$

where M corresponds to the population size.

Individual	Standardised Fitness	Adjusted Fitness	Normalised Fitness	Number of Occurrences in Mating Pool
Tree 1	30	0.03	0.09	$0.34 \approx 0$
Tree 2	15	0.06	0.17	$0.69 \approx 1$
Tree 3	5	0.17	0.49	$1.94 \approx 2$
Tree 4	10	0.09	0.26	$1.03 \approx 1$
Total		0.35	1.00	4

Table 2.2: Illustrating fitness proportionate selection.

Table 2.2 illustrates how the fitness proportionate selection method is applied. The first step is to compute the adjusted fitness (column 3), and then to compute the normalised fitness (column 4). The number of occurrences (column 5) that each tree will appear in the mating pool is determined by multiplying the normalised

fitness with the population size, in this case the population size is 4. The number of occurrences are rounded up, and from the example, the mating pool is {tree 2, tree 3, tree 3, tree 4}. Parents are then randomly selected from the mating pool.

A disadvantage of fitness proportionate selection is that an individual with a high normalised fitness will appear several times in the mating pool, and consequently there is a greater probability that it will be selected several times. Furthermore, individuals with a low normalised fitness will never be selected. By selecting the same individual to be the parent, this will result in a loss of diversity due to the fact that the same parents will be used repeatedly. This could further lead to premature convergence.

2.9.2 Tournament selection

Tournament selection [2] is dependent on the tournament size. A subset of individuals is created by randomly selecting individuals from the population; the size of this set is equal to the tournament size. The fitness of each individual in the subset is calculated and the fittest individual is returned. This fittest individual which is returned represents the parent which will then be used to create offspring. The pseudocode for tournament selection is presented in algorithm 2.2.

Algorithm 2.2: Pseudocode for tournament selection.

```
   input  : tournament_size
   output: A GP individual
1  begin
2      for i ← 1 to tournament_size do
3          random_individual ← randomly select an individual from the GP
           population.
4          if i = 1 then
5              best_individual ← random_individual
6          end
7          else
8              Compare best_individual and random_individual and store the one
               with the higher fitness to best_individual.
9          end
10     end
11     return best_individual
12 end
```

The tournament size dictates the *selection pressure* [2]. If the tournament size is small then there is a small amount of selection pressure. The opposite can be said for a large tournament size whereby it results in a greater amount of selection pressure.

A large tournament size leads to an *elitist* GP algorithm which could converge prematurely to a local optimum. Tournament selection is a commonly used selection method. In comparison to the fitness proportionate method, tournament selection benefits from the fact that a selection pressure can be used. If the tournament size is selected correctly, then tournament selection will maintain diversity and additionally drive the GP algorithm towards a solution. Additionally, tournament selection does not require the extra computations involved in calculating the adjusted fitness and the normalised fitness.

2.10 Genetic Operators

Genetic operators (GOs) represent the search operations which are used by the GP algorithm. Koza [3] mentions that if two programs are capable of solving a certain problem to a certain extent, then there are some useful parts within those two programs that contribute to the program's performance. Thus Koza states that by recombining random parts of the parents the resulting programs may be even better at solving the problem [3].

GOs are used to combine, alter or duplicate the genetic material from the parents to obtain offspring. Typically, the initial population will contain individuals which are not able to solve the problem at hand. GOs are thus applied to these individuals in the hope that they will drive the population towards a solution. Thus the GOs are used to transform the population [2]. These offspring are of different shapes and sizes when compared to their parents. The parents are obtained by a selection method as described in section 2.9.

Each genetic operator can be categorised as being a local or a global search operator. A *global search* operator allows an evolutionary algorithm to explore different areas of the program space. On the contrary, a *local search* operator is one which makes use of exploitation to examine the surrounding areas of the program space in which the evolutionary search is currently situated. GP makes use of the GOs to traverse the program space using exploration and exploitation. The application rate of the GOs will affect the evolutionary process. If a large amount of global search is used, then the GP algorithm will jump to random areas of the program space and may never have an opportunity to converge towards the global optimum. Conversely, if a large amount of local search is applied, then the GP algorithm may end up being stuck in a local optimum and not have an opportunity to explore other areas of the program space.

The three most common GOs are the crossover, mutation, and reproduction operators. Koza [3] presents additional operators (permutation, editing, and decimation) which can be applied to GP; however, these will not be examined in further

detail.

2.10.1 Reproduction

The *reproduction* operator copies a parent across to the next generation by simply duplicating the individual and making no alterations to it [1–3]. The reproduction operator is a local operator since it makes no alterations to the parent being copied across.

2.10.2 Mutation

The *mutation* operator [1–3] creates an offspring by mutating a single parent as follows. A mutation point, p, is randomly selected in the parent, and the subtree rooted at the point is removed. A new randomly generated subtree is inserted at point p. Mutation can cause trees to grow rapidly and thus *pruning* is used to ensure that individuals do not grow beyond a certain size. Pruning is achieved by replacing any function node at the maximum tree depth with a terminal node. Mutation is a global operator due to the fact that random subtrees are created at the mutation points which can result in a significant difference between the offspring and the parent. Consequently, the mutation operator does not promote convergence. Figure 2.6 illustrates the mutation operator.

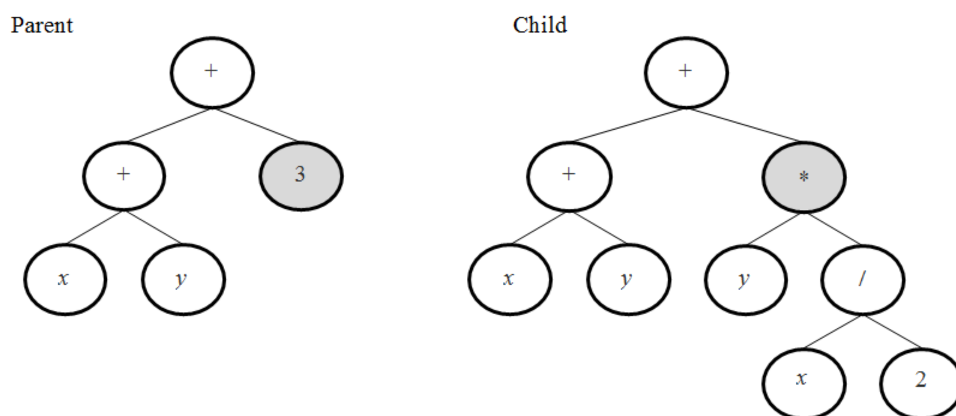


Figure 2.6: Mutation operator, adapted from [1].

2.10.3 Crossover

The *crossover* operator creates two new offspring which are formed by taking parts (genetic material) from two parents [3] [2]. The operator selects two parents from the population based on a selection method. A crossover point is then randomly selected in both trees, say point p_1 and p_2, from tree t_1 and t_2 respectively. The

crossover then happens as follows: the subtree rooted at p_1 is removed from t_1 and inserted into the position p_2 in t_2. The same logic applies to the point p_2; the subtree root at the point is removed from t_2 and inserted into the place of p_1 in t_1. Figure 2.7 illustrates the crossover operator. Crossover promotes convergence and is a local search operator.

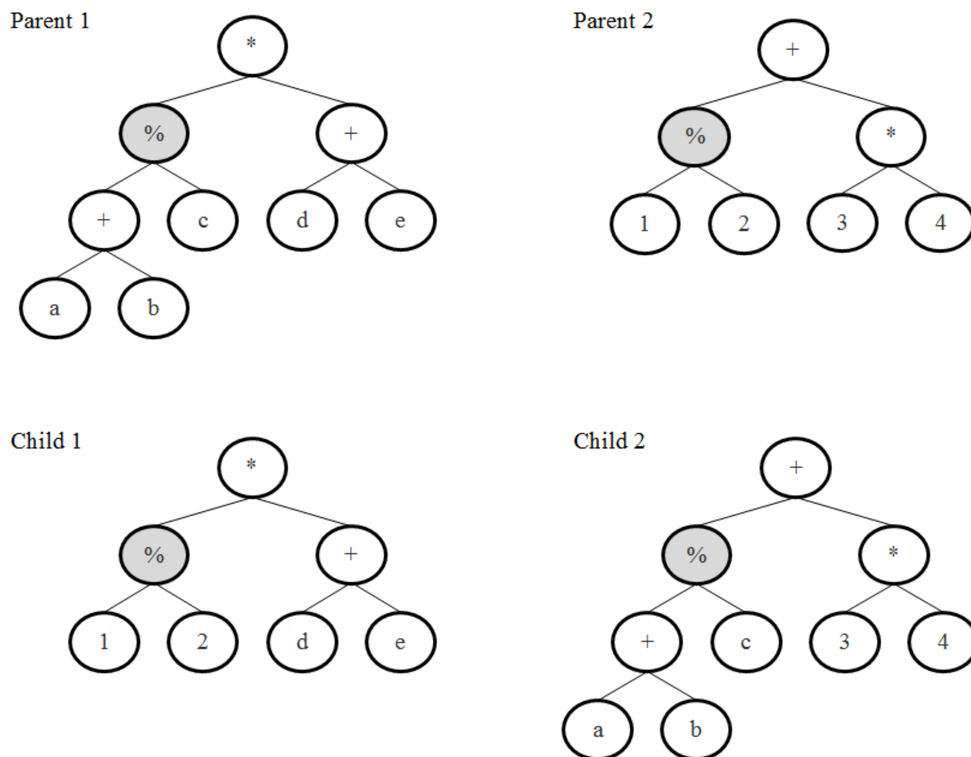


Figure 2.7: Crossover operator [2].

2.11 Termination

Koza states that the GP "*paradigm parallels nature in that it is a never-ending process*" [3]. This obviously is not feasible, hence the GP algorithm should terminate once a *success predicate* is met. The success predicate can be defined in different ways however the most common one is to find a solution which has a hits ratio of 100%, i.e. a perfect solution to the problem. The success predicate can be problem dependent [1] and thus defined differently from one problem to another. However in certain problem domains, aiming for a perfect solution is not reasonable, hence once a near-solution is found the GP algorithm can terminate.

2.12 Strongly-Typed GP

Strongly-typed genetic programming [15] enforces constraints on the nodes within a representation. A type is allocated to the terminals and functions, and consequently, this guarantees that syntactically correct trees are evolved. Strongly-typed GP will ensure that during the initial population generation and application of the GOs, the types of the functions and terminals are respected and not violated. Additionally, strongly-typed GP reduces the size of the program space by limiting the different combinations of functions and terminals [15].

The *If-Then-Else* function is one which requires strongly-typed GP. This function is illustrated in figure 2.8, and each of the three arguments can be assigned a type. The *If-Then-Else* function has an evaluation (the *If* part) and two consequences (the *Then* and *Else* parts) which are executed respectively based on the evaluation. Assume a boolean type is assigned to the *If* part as illustrated in figure 2.8; as a result of strongly typed GP, the *If* part should always return a boolean type and this enforcement cannot be violated. Thus variables x and y have to be boolean types.

When applying GOs to strongly-typed GP, it must be ensured that the assigned types are not violated. Thus assume in figure 2.8 that the terminal x is selected as a mutation point, it must be ensured that the new subtree at this point will return a boolean value. If the assigned types are violated, then the trees would be invalid.

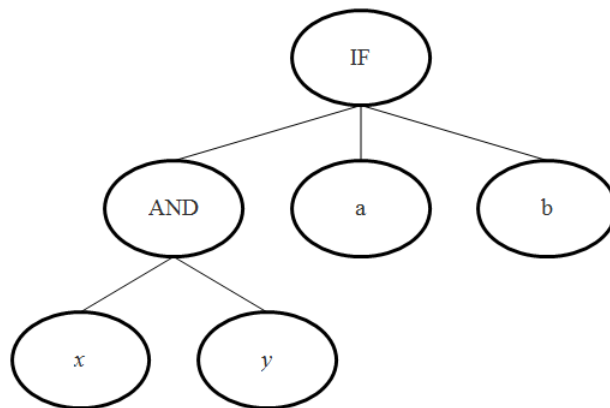


Figure 2.8: If-Then-Else function.

2.13 GP Control Models

There are two major control models which can be used when implementing a GP algorithm, a generational control model and a steady-state control model [2]. The models control the population and how they evolve. The generational control model

was presented in section 2.3. In this model, at each generation, the entire current population is replaced with a new one which is obtained by applying the GOs.

In a *steady-state control model*, there are no explicit generations [2]. Algorithm 2.3 illustrates a steady-steady GP algorithm [2]. A fixed population size is preserved throughout the execution of the algorithm. The algorithm iterates several times by randomly selecting a subset of individuals from the population and replacing those individuals with the offspring. These iterations are referred to as generations. A *generational equivalent* is when the subset of individuals is equal to the population size [16].

Individuals with a low fitness are replaced by the offspring, and in comparison to a generational GP model, the entire population is not replaced with a new population. This replacement is referred to as the *replacement strategy*. The replacement strategy is essentially an inverse selection method, such as an inverse fitness proportionate selection, or inverse tournament selection. In an inverse selection method, the worse individuals are returned.

Algorithm 2.3: Steady state GP.

```
1  begin
2      Randomly create the initial population.
3      repeat
4          Randomly select individuals from the population to take part in a
           tournament.
5          Evaluate those individuals.
6          Obtain the winner(s).
7          Apply the genetic operators to the winner(s).
8          Replace the losers in the tournament with the offspring created.
9      until a termination criteria is met;
10 end
11 return The best individual from the population
```

2.14 Modularisation

Several operators have been designed in such a way that they can preserve parts of the GP individual. These preserved parts can then be further reused within other individuals. The goal in this approach is to capture good *building blocks* so that the GOs cannot alter these building blocks. Two of the methods described in this section are GOs which achieve modularisation.

2.14.1 Encapsulation

Koza [3] defined the *encapsulation operator* as one which selects a subtree from a tree and gives it a name. This named subtree (*encapsulated function)* can be referenced by other individuals in future generations. A single parent is obtained using a selection method, then a random function node is selected from the parent. Since a function node does not have an arity of zero it means that the function node has a subtree. This subtree is removed from the parent and is given a name, say E_0, and is stored in memory. The deleted subtree in the parent is replaced with the encapsulated function E_0. These encapsulated functions are added to the terminal set and have an arity of zero. The next time the encapsulation operator is executed the new encapsulated function will be named E_1, and similarly each encapsulated function will be named in a numerically ordered manner. When a tree is evaluated, if an encapsulated function is present, this subtree represented by the encapsulated function is called from memory, thus the encapsulation operator does not affect the evaluation of a tree.

Koza states that the encapsulated functions are "*no longer subject to the potentially disruptive effect of crossover*" [3], and thus this operator is useful in preserving good building blocks. However from the original definition of the encapsulation operator made by Koza, there is no mention on how the good building blocks are selected. The building blocks are randomly selected and hence there is no guarantee that the GP algorithm will preserve good building blocks. Figure 2.9 illustrates the encapsulation operator. The dashed lines represent the subtree which is encapsulated when the multiplication function is selected. The tree on the right is the result of the encapsulation operator whereby the original subtree is replaced with the terminal E_0.

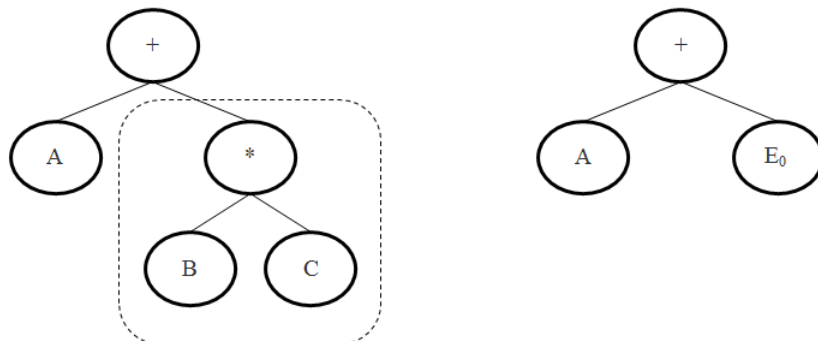


Figure 2.9: Encapsulation operator, adapted from [3].

2.14.2 Compression

Modular acquisition was also investigated by Angeline and Pollack [4]. A *compression operator* was developed which compresses a part of a tree up to a certain depth. A random subtree is selected up to a randomly selected depth d, and this selected portion of the tree is removed from the original tree and replaced with a new function. If the nodes at depth d have an arity of zero, then this operation is identical to the encapsulation operator [17]. However, it may occur that there are additional functions or terminals which are at a lower depth than the selected portion, in this case, all of the extra branches that are below the selected portion are used as arguments to the new function.

Each created compressed function is allocated a name, say C_i, and these functions are added to the function set. Angeline and Pollack point out that through the use of compression, the functions created are made up of a "*higher level of abstraction from the components which comprise it*" [4]. In a similar fashion to encapsulation, the compression operator protects the subtrees from the destructive operators.

Figure 2.10 illustrates the compression operator. In this instance, the dashed lines represent the subtree which was randomly selected with a cut-off depth of 2 from the *or* function. The tree on the right illustrates the result of the compression. Since in this case there were additional branches extending from the cut-off depth, these elements have been added as arguments to the newly created function. The new function has an arity of 3.

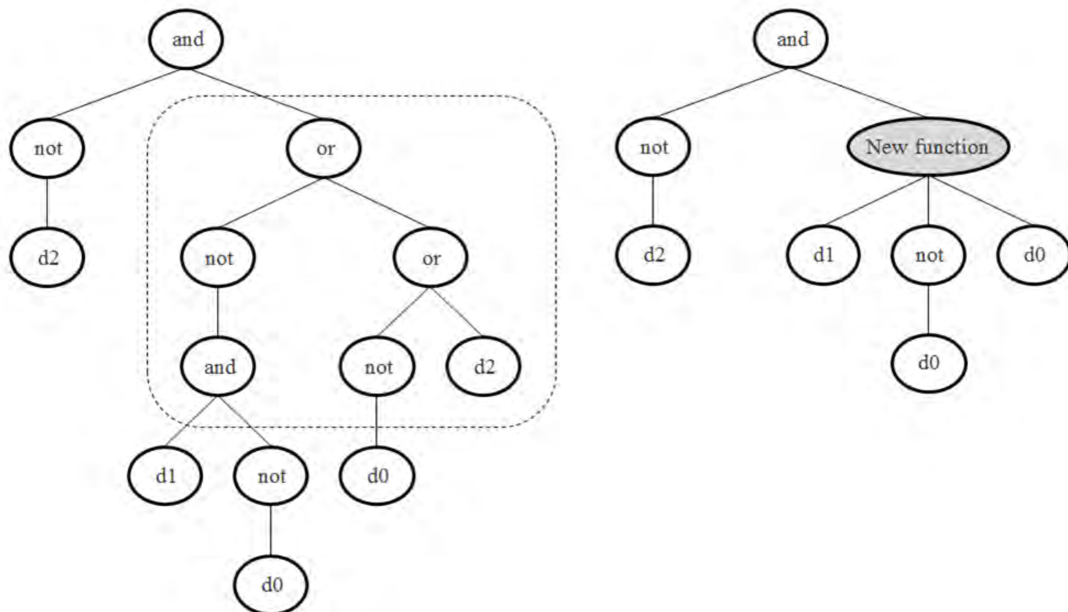


Figure 2.10: Compression operator, adapted from [4].

2.15 GP and Bloat

Before introducing the concept of bloat, introns should be discussed. *Introns* are defined as redundant code in GP individuals which have no overall affect towards its fitness [1]. An intron does not directly affect the survivability of the GP individual [2]. Examples of introns in prefix notation are, *(AND 1 1)*, *(MOVE-UP MOVE-DOWN)*, *(OR 0 0)*. *Bloat* occurs when the individuals in a GP population grow uncontrollably to their maximum tree depth. The exponential growth of introns leads to bloat [2]. Bloat will most certainly occur if no measure is installed to prevent it. Introns clearly have an effect on bloat however it can be argued that the exponential growth of introns acts as global protection [2]. The crossover operator can have a destructive effect if it removes good parts from a GP individual. The destructive effect is reduced if the crossover operator removes an intron from a tree instead of a good building block. As the GP algorithm iterates, the individuals get fitter, and eventually the GP algorithm will find it difficult to further improve individuals. Once bloat occurs, the GP algorithm will struggle to improve the fitness of the current best individual found so far. Thus, it can be argued that introns have a positive effect on individuals, however, they consequently lead to bloat nonetheless. Luke and Panait [18] discussed several methods for controlling bloat.

2.16 Strengths and Weaknesses of GP

2.16.1 Strengths

- Since GP relies on randomness and uses a random seed on each execution of the algorithm, a different solution can be obtained on each run.

- GP evolves programs which in turn resemble computer programs. Thus the GP solutions are often easily understood and executable.

- Little prior knowledge of the problem domain is required. Once a set of fitness cases has been obtained, and suitable functions and terminals have been formulated, GP is then able to evolve solutions.

- Several areas of the GP algorithm can be parallelised in order to speed up the computational time.

2.16.2 Weaknesses

- GP suffers from the large amount of parameters which need to be optimised for different problem domains.

- GP can suffer from large run times, especially if there are a large number of fitness cases which can slow down the evaluation of the population.
- Premature convergence is one of the major issues with GP. The lack of genetic diversity and destructive effects of the crossover operator can result in the GP algorithm getting stuck in a local optima.
- There is no guarantee that GP will find the global optimum solution to a given problem due to the random nature of the GP algorithm.

There are ways of addressing the some of shortcomings of GP. When dealing with a large number of fitness cases, the evaluation phase of the GP algorithm tends to be a bottleneck, and thus a possible solution is to parallelise the evaluation phase. Harding and Banzhaf [19] make use of a graphics processing unit to speed up the evaluation. A successful investigation by Ciesielski and Mawhinney [20] showed that a similarity replacement - where similar individuals in the population were replaced with a new random individual - resulted in a greater number of solutions; thus dealing with the issue of premature convergence. When running a GP algorithm, if solutions to the problem are not found, then experimenting with the GP parameters can improve the algorithm's ability to find a solution.

2.17 Conclusion

This chapter provided a description of the GP algorithm. The generational GP algorithm was presented and each process of the algorithm was described. When implementing GP, one has to first consider which representation is most suitable for the problem. This chapter described the tree based GP representation. Once a representation has been chosen, the initial population has to be created. Three initial population generation methods were described, with the ramped half and half being the most commonly used method. The choice of the method will affect the structure of the trees and can impact the diversity amongst the population. When the initial population has been created, the individuals have to be evaluated. The evaluation is carried out using a fitness function and fitness cases. The fitness cases need to be selected in such a way as to sufficiently represent the problem domain. The fitness function will vary depending on the problem domain. Parents are required in order to create offspring. Two selection methods were discussed, and the tournament selection method allows one to have flexibility over the selection pressure. In order for GP to explore the program space, GOs are applied to the parents to create offspring. Three GOs were discussed and their effects on the evolutionary process were highlighted. A combination of crossover and mutation can permit GP to make use of exploration and exploitation to traverse the program

space. In GP, modularisation can be achieved using encapsulation or compression; these two GOs were described in this chapter. Two well-known control models were described. Although GP suffers from several disadvantages, it has been used in numerous studies and has been proven successful in solving a vast number of problems [1, 2].

Chapter 3

Data Classification

3.1 Introduction

This chapter first introduces data classification in section 3.2, and then provides clarification of terminology pertaining to data classification in section 3.3. Details on performance measures and how the data is used to evaluate classifiers are presented in sections 3.4 and 3.5 respectively. Section 3.6 describes other methods which have previously been applied to data classification. A discussion on active research areas in data classification is presented in section 3.7. Existing software which can be used for data classification is highlighted in section 3.8. Finally, section 3.9 concludes this chapter.

3.2 Introduction to Data Classification

In machine learning, data classification is a technique whereby a classifier is created in order to allocate a class label to an instance of data [10]. The data is made up of several attributes (for example *blood type*) and each attribute has corresponding values (for example *A+*, *AB-*, *O+*). Alpaydin [21] describes a credit scoring example in which the data includes information about customers (for instance, income and profession) and, for this data, the objective is to create a classifier which is able to correctly classify the customers within the data as either low-risk or high-risk customers. In supervised learning, a classifier is built from known instances and their corresponding class, and in turn a classifier should be able to predict the class for unseen instances.

A number of problems can be formulated as classification problems. Applications of classification include:

- credit scoring [22]

- image recognition [23]
- medical diagnosis [24]
- handwritten character recognition [25]

Figure 3.1 outlines the overall classification process. Initially, the data has to be collected in some manner; this can be achieved in a number of ways, such as conducting a survey or recording values from a device. The data is then combined to create a data set. Pre-processing steps such as feature selection is then performed if necessary. A training set is created from the data set, and then a classifier is developed. Once the classifier has been developed, it is evaluated on a test set. This chapter deals with providing an overview of the processes - from the creation of the training set, to the evaluation of the classifiers.

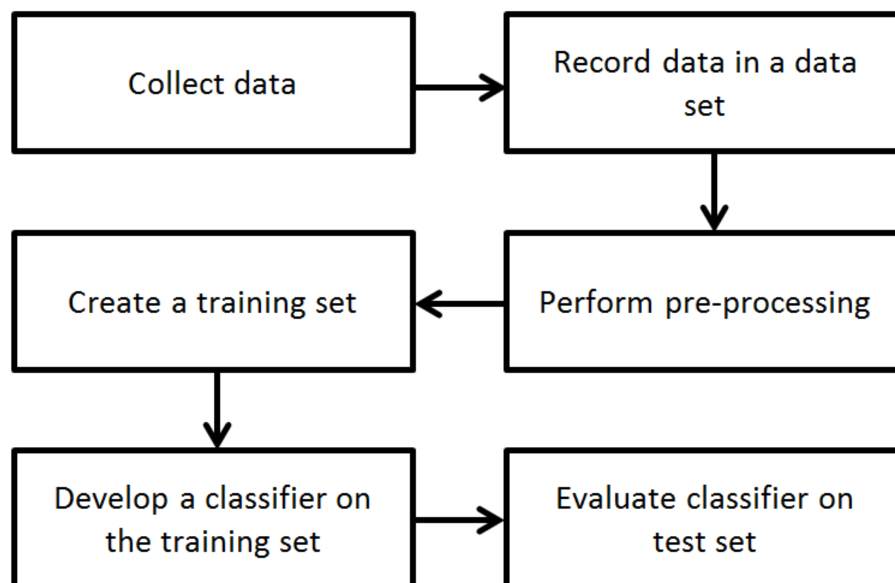


Figure 3.1: The classification process.

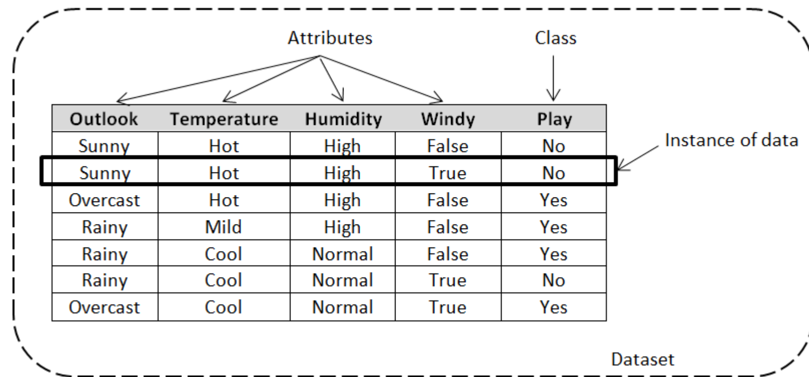
3.3 Definitions

The following section presents several definitions of terms which are commonly used in data classification. A sample data set is presented in figure 3.2 in order to assist with the explanations.

3.3.1 Instance

An *instance* represents an entity described by one or more attributes [26]. Instances are typically visually represented as rows within a data set, and an example of an

instance is presented by the bold rectangle in figure 3.2. For this particular instance shown in the figure, the value for outlook is "Sunny", the value for temperature is "Hot", the value for humidity is "High", and the value for windy is "True", and finally, the output for this instance, being whether or not to play, is "No". An instance contains a value for each attribute, and a value for the class.



Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes

Figure 3.2: A sample data set. The weather data set, adapted from [5].

3.3.2 Attribute

An *attribute*, or feature, can be described as a variable which represents a particular characteristic of a data set [10, 26]. An attribute can be either numerical or categorical. Numerical attributes can be categorised as being either discrete or continuous. Discrete values usually consist of integers, whereas continuous attributes consist of real valued numbers. Categorical attributes, on the other hand, are made up of a finite set of values which *"puts objects into categories, e.g. the name or colour of an object"* [10]. A categorical attribute could be made up of integer values; however, they should have no mathematical significance [10]; these integer values should merely be labels.

For instance, *height*, *age*, or *gender*, are possible examples of attributes, and in turn these attributes will have values of a different type; continuous values, integer values, and categorical values respectively. Since the *height* attribute is continuous, and assuming the values were measured to two decimal places, then typical values could include 150.33, 185.65, or 191.22. Since *gender* is a categorical attribute, there are two possible values for this attribute; male or female. The attributes are visually represented as columns within a data set; as seen in figure 3.2. This data set has four attributes, namely, *Outlook*, *Temperature*, *Humidity*, and *Windy*.

3.3.3 Class

A *class* is the target output for each instance of data. In data classification, the goal of a machine learning algorithm is to predict the class through the use of the attributes. The class should be thought of as an output value, whereby a machine learning algorithm uses all the attributes as input. From the sample data set in figure 3.2, the classes are “Yes” and “No”.

An instance of data typically contains a single class; however, this may not always be the case. When more than one class is present, these instances are referred to as *multi-labelled instances* [5]. In this case, an instance can belong to more than one class.

3.3.4 Data set

A *data set* refers to the complete set of instances of data [10]. A data set consists of a number of instances of data which are in turn built up of several attributes. Each instance of data has a class value and thus a data set has several classes. A binary data set is one with only two classes. In a classification task, when a binary data set is used, the machine learning task is then referred to as *binary classification*. When a data set consists of more than one class, the machine learning task is referred to as *multiclass classification*. In figure 3.2 the data set is represented by the dotted lines, i.e. the entire table of data is the data set. For this data set, the task is thus to determine whether or not to play some game based on the weather conditions. There are two classes, *no* and *yes*.

Data sets play a vital role in the development of data classification algorithms and also in the comparison between different algorithms. There are several repositories available for researchers to use in order to test their algorithms. One of the most commonly used one is the UCI machine learning repository [27]. To this date, the repository contains over 200 data sets for classification.

It is common practice that when a new algorithm is developed, researchers will compare the performance of the algorithm on several data sets in the UCI machine learning repository [10]. However, if a new classifier performs well on certain data sets, this does not necessarily imply that the algorithm is good [10] and thus a researcher should take care as to test the performance over several data sets.

When a class within a data set is known, applying data mining techniques to such a data set is known as *supervised learning*. Conversely, applying data mining techniques to a data set without a known class is referred to as *unsupervised learning*.

3.3.5 Class balance

The *class balance* of a data set refers to the number of instances within each class and the ratio between them [28]. If there are approximately an equal number of instances in each class, then the data set is said to be well-balanced. An imbalanced data set is one whereby one or more classes have a number of instances significantly smaller than the number of instances in the other classes. Assume, for instance, that for particular a data set, 90% of the instances specify *class A* as the output, and 10% specify *class B*, then this data set is considered to be imbalanced. Imbalanced data sets increase the complexity of the data classification process due to the fact that more data is available for one class, and thus learning from such a data set will result in a bias towards the majority class [29].

From figure 3.2, in the class column, one can see that there are 3 *no* labels (approximately 43% of the total data), and 4 *yes* labels (approximately 57% of the total data), thus it can be said that this data set is well-balanced.

3.3.6 Classifier

A classifier is an algorithm that can discriminate between several classes [26]. Classifiers are created in the hopes that they are able to correctly allocate class labels to as many instances of data as possible. Let $\mathbf{x}$ consist of a vector representing the value of each attribute for an instance of data, thus $\mathbf{x}_i = \{x_{i,0}, x_{i,1}, ..., x_{i,n}\}$ where n represents the total number of attributes within a data set, and $x_{i,j}$ represents the value of instance i for attribute j. Let w_i correspond to the class value for the ith instance of data.

The goal of a machine learning algorithm is to create some function ϕ whereby $\phi(\mathbf{x_i}) = w_i$ [26]. The function ϕ can be represented in various ways, for example through the use of mathematical mappings, decisions, or logical rules. An example of a simplistic classifier which is based on the data set in figure 3.2 would be:

IF outlook = rainy THEN ***don't play***

ELSE ***play***

The output from the simplistic classifier in comparison to the data set is illustrated in table 3.1. An instance is correctly classified when the class value matches the output of the classifier. The simplistic classifier is able to correctly classify 3 instances, the third, sixth, and the seventh. The ultimate goal is thus to create a classifier which is able to correctly classify all of the instances of data within the data set.

Instance	**Class - Play**	**Simplistic classifier output**
1	No	Yes
2	No	Yes
3	Yes	Yes
4	Yes	No
5	Yes	No
6	No	No
7	Yes	Yes

Table 3.1: Class output from figure 3.2, and the output for the simplistic classifier.

3.4 Performance Measures

Performance measures are used to interpret the quality of a classifier. In this section several performance measures will be described.

3.4.1 Confusion matrix

The confusion matrix "*describes how well a classifier can recognise different classes*" [7] and provides a way of obtaining a breakdown of the performance of the classifier. This breakdown clearly illustrates how many instances of data from a particular class, say x, were classified as being class x and how many of them were classified as another class [10]. The confusion matrix can be used to determine the performance of classifiers for both binary and multiclass classification problems. Table 3.2 illustrates the layout of a confusion matrix for a binary classification problem.

There are four terms which are used when using a confusion matrix, these terms are defined below [7, 10]. In the following definitions the two classes are labelled *positive* and *negative*.

- *True Positive* (TP): these are instances from the positive class that have been correctly classified as being positive.
- *True Negative* (TN): these are instances from the negative class that have been correctly classified as being negative.
- *False Negative* (FN): these are instances from the positive class that have been incorrectly classified as being negative.
- *False Positive* (FP): these are instances from the negative class that have been incorrectly classified as being positive.

Correct classification	Classified as	
	Positive	*Negative*
Positive	True Positive (TP)	False Negative (FN)
Negative	False Positive (FP)	True Negative (TN)

Table 3.2: Confusion matrix, extracted from [10].

A confusion matrix can be generalised to analyse the performance of classifiers which are created for multiclass problems. In this case, if there are c classes, a c by c table is created, and index (i, j) represents the number of instances from class i that have been classified as class j [7].

From the terms defined above, it is possible to define additional performance measures. The first is the *accuracy* measure, and is defined as the the following percentage: the total number of correctly classified instances divided by the total number the instances in the data set [7,10]. For a binary classification problem the accuracy is defined as [7]:

$$Accuracy(classifier) = \frac{TP+TN}{TP+TN+FP+FN}$$

It is also common practice to report on the *error rate*, which is defined as [7]:

$$Error(classifier) = 1 - Accuracy(classifier)$$

3.4.2 Sensitivity and specificity

The accuracy measure described above can suffer from a flaw which can have a significant impact based on the nature of the classification task. For instance, consider creating a classifier for determining whether or not individuals have a medical issue or not.

	Classified positive	**Classified negative**
Positive instances	10	20
Negative instances	20	50

Table 3.3: Illustrating accuracy paradox - classifier 1 confusion matrix.

	Classified positive	**Classified negative**
Positive instances	0	30
Negative instances	0	70

Table 3.4: Illustrating accuracy paradox - classifier 2 confusion matrix.

Classifier 1, illustrated in table 3.3, results in an accuracy of 60% whereas classifier 2, illustrated in table 3.4, has an accuracy of 70%. Thus one would assume that classifier 2 is more promising than the other. However, classifier 2 is of no use in the

medical domain, as it does not correctly classify any of the cases pertaining to the medical issue since none of the individuals who had the medical issue were classified as true positive. This problem is known as the accuracy paradox. In order to deal with this, two additional performance measures are presented, the sensitivity and specificity measures.

The sensitivity represents the percentage of positive instances that were correctly classified as being positive [7, 10] and is defined as,

$$Sensitivity(classifier) = \frac{TP}{TP+FN}$$

The specificity represents the percentage of negative instances that were correctly classified as being negative [7, 10] and is defined as,

$$Specificity(classifier) = \frac{TN}{FP+TN}$$

3.4.3 Receiver operating characteristics

Receiver Operating Characteristics (ROC) [6] graphs are commonly used when creating a classifier for binary classification. A ROC graph plots the FP rates on the x axis and the TP rates on the y axis. An example of a ROC graph is illustrated in figure 3.3. When a classifier is applied to a test set, the corresponding TP and FP rates can be obtained and represented as one point on the ROC graph. Points from multiple different classifiers that have been evaluated on the same test set may be plotted on the same ROC graph, providing a way to visually compare their performance. In an ideal situation, a classifier will generate a point at $(1.0, 0.0)$, denoted as point "x" in the figure, which represents a TP rate of 100% and a FP rate of 0% implying that all the instances were correctly classified. The worst situation is represented by the point $(0.0, 1.0)$ whereby the TP rate is 0% and the FP rate is 100% and consequently all the instances have been incorrectly classified. The line $y = x$ is represented by random guessing, and any point on the line represents a classifier which guesses the positive class $x\%$ of the time. Any point below this line, such as point "p", represents a classifier which is worse than random guessing [6, 10, 26]. Thus generally speaking, a point which is located to the top-left of another point represents a better classifier [10]. Typically a single point in a ROC space is not used in literature when comparing performance; instead a ROC curve is constructed. A complete description on how to create ROC curves is presented by Fawcett [6].

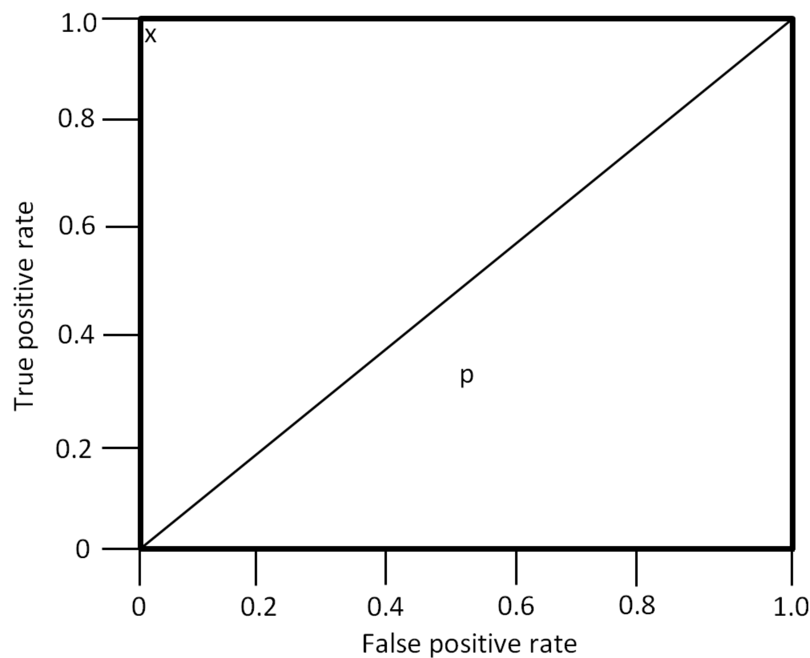


Figure 3.3: An example of a ROC graph, adapted from [6].

3.5 Evaluating Classifiers

The predictive accuracy of a classifier can be measured by determining how well the classifier is able to predict the class valuesh describes several applications of GP which for instances of data which were not used during the training of the algorithm [10]. Evaluating the performance of a classifier based solely on the data which was used for training will result in an optimistic performance; such an evaluation is not a good indicator of the true performance [5, 7]. In order to correctly assess the performance of a classifier, two sets of data have to be created from the original data set, a *training set* and a *test set*. In terms of the accuracy measure, the training accuracy is the percentage of correctly classified instances in the training set divided by the total number of instances in the training set, and the test accuracy is the percentage of correctly classified instances in the test set divided by total the number of instances in the test set [10].

The training set is used by the machine learning algorithm to create the classifier [5]. The test set is then used to evaluate how good the prediction of the classifier actually is [26]. The training and test set must not contain overlapping instances; additionally, the test set should not be used whilst creating the classifier [5]. When the training and test set are combined together, they should form the original data set. There are several methods for creating the training and test sets and these methods are described in subsections that follow.

3.5.1 Train/test split

In this approach (also referred to as the *holdout method*), the data set is simply split into two sets, a training set and a test set. The classifier is created using the training set and then the test set is used to predict the accuracy of the classifier. Typically, the training set consists of 2/3 of the data and the remaining 1/3 is allocated to the test set [7, 10, 26]. Since the splitting of the data is performed randomly, it is thus possible that one of the two sets is not sufficiently representative of the problem [5]. For instance, consider a highly imbalanced data set, it is possible that the training set does not contain a single instance of the minority class. Consequently, an algorithm trained on this data set will perform poorly when attempting to classify instances from the minority class. In order to overcome this, stratification can be applied. This ensures that each of the two sets contain an equal number of instances from each class.

3.5.2 K-fold cross-validation

The *k-fold cross-validation method* begins by splitting the data set into k-folds of equal (approximately equal) size, and running the algorithm k times using each fold as a test set exactly once and the remaining *(k-1)* folds as the training set [7,10,26]. Each fold should be disjoint from each other [7] and if the total number of instances in the data set is not completely divisible by k then the last folds should contain fewer instances [10].

Figure 3.4 illustrates the 10-fold cross-validation method. Each square represents a fold and there are a total of 10 folds. The black square represents the test set, whereas the grey squares collectively represent the training set. The folds used for testing are selected in sequential order thus allowing each fold to be used once for testing whilst the remaining folds are used for training. When applying this method, the value of k is commonly selected to be 10 as this has been shown to yield the best evaluation estimate [5, 10, 26]. Witten *et al.* [5] point out that performing the 10-fold cross-validation method once is insufficient, and that multiple runs would lead to a more reliable performance estimate. In other words, the process of randomly creating the folds should be repeated several times, and in each turn the 10-fold cross-validation method should be applied.

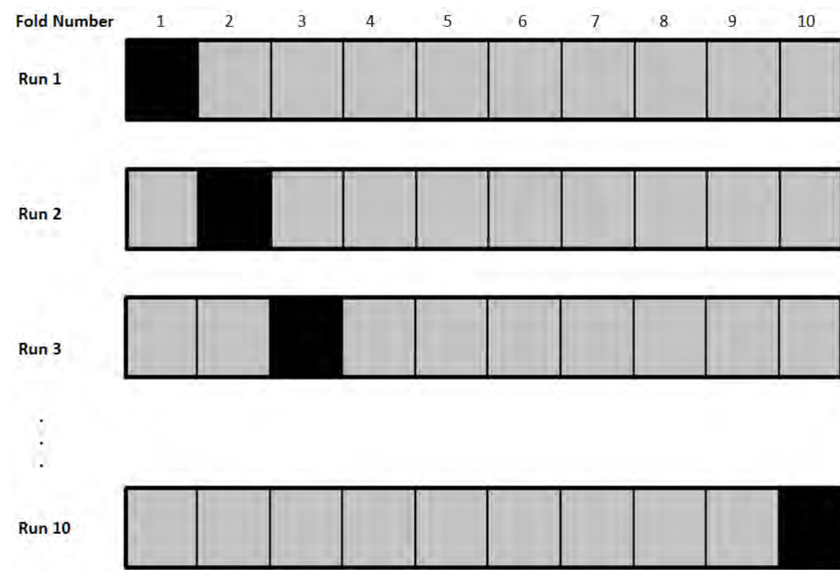


Figure 3.4: 10-fold cross-validation, adapted from [7].

3.5.3 Leave-one-out

The *leave-one-out* method (also known as *N-fold cross-validation*, or *jack-knifing*), is similar to the k-fold cross-validation method, however, in this case the value of k is equal to the number of instances in the data set and hence each instance is used once for testing [5, 7, 10, 26]. The name leave-one-out arises from the fact that a fold is created for each individual instance of data, and for each run of the algorithm, only one instance of data (the one which is left out) is used for testing, while the remaining N-1 instances are used for training. Bramer [10] points out that this method is not suitable for large data sets due to the amount of computational time required to execute all the runs. Bramer, however, further mentions that this method is more suitable for very small data sets because it allows as much data as possible to be used for training.

3.5.4 Bootstrapping

The *bootstrapping* method makes use of random sampling with replacement [5,7]. In this approach, the training set is created by randomly sampling (with replacement) the instances from the original data set. Due to the fact that the sampling is done with replacement, duplicate instances may occur in the training set. The sampling is continued until the number of elements in the training set is equal to the number of instances in the original data set. Once the training set has been created, the test set is made up of all the instances from the original data set which were not added to the training set. For example, let a data set have the following instances

represented by numbers, {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. Suppose a training set is constructed as follows, {2, 2, 2, 5, 6, 6, 6, 7, 10}, then the test set is {1, 3, 4, 8, 9}.

3.6 Previous Work on Data Classification

The sections above described various aspects of data classification and defined crucial terminology pertaining to the domain of classification. This section will describe existing techniques which are commonly used when dealing with data classification tasks.

3.6.1 K-nearest neighbour

The *K-nearest neighbour* (k-NN) [30, 31] algorithm is a simplistic machine learning algorithm which can be used to solve data classification problems. The value of k is a user defined parameter. Each instance is represented as a point in a multi-dimensional space. The dimensionality of this space is defined to be the same as the number of attributes within the data set. A new instance of data, say instance x, is allocated a class by placing it into the space, and finding k points which are closest to the point x. The closest points are usually found using the Eucledean distance formula.

Assume k is 3, and let points $\{a,\ b,\ c\}$ be the closest instances to point x. Since each point already in the space has a class; the class for point x is computed by the mode of the classes from points a, b, and c. Thus to summarise, a new instance is added to the space, then its k nearest neighbouring points are found, and the mode of the classes is computed based on the these points. Finally, the mode is allocated to the new instance.

This algorithm is simple to implement and fine tune; however, a shortcoming of k-NN is that a large amount of computation will take place if the number of attributes is large. k-NN is also hindered by imbalanced data sets. Points in the minority class will typically be surrounded by points from the majority class. Assume a new point is added for which the correct classification is the minority class; there is a greater probability that the neighbours are from the majority class.

In order to overcome some of the limitations of the k-NN, several variations to the original method have been investigated. Voulgaris and Magoulas [32] proposed five variations of the k-NN algorithm which do not simply consider the neighbours. The proposed methods evaluate the neighbours using properties of the data set. V-kNN attempts to optimise the value of k for each instance prior to testing, and then uses these values when classifying new instances. CB-kNN was proposed in order to overcome the issue of class balance; W-kNN makes use of an index in order to weight the attributes based on their usefulness. DB-kNN makes use of

an additional property defined as the structural density, this property takes into account the number of neighbouring points and the volume of the neighbourhood. DB-kNN evaluates the quality of the neighbours as a combination of CB-kNN and W-kNN. The proposed methods outperformed the standard k-NN algorithm.

Bao *et al.* [33] propose a modification of the k-NN algorithm by combining multiple k-NN classifiers in order to improve the accuracy of the algorithm; this proposed approach was called KNN-DC. Each classifier uses a different distance function, and computes k neighbours for each function. All of the k neighbours are combined, and a majority vote is used to determine the class for the test instances. KNN-DC was tested using 3 to 6 different functions on 21 data sets from the UCI repository, and the findings indicated that, when all 6 functions were used, KNN-DC was the best performing method.

Another modification to the k-NN algorithm is presented by Parvin *et al.* [34] whereby the validity of each instance in the training set is computed. The validity for an instance i is computed as the number of neighbours which have the same class as i, divided by the value of k. In order to determine the class for instances in the test set, a weight is associated with the neighbours. The weight takes into consideration the distance between each test point and the neighbours, as well as the validity of the neighbours. The modified approach was tested on 9 data sets and generally outperformed the standard k-NN algorithm.

k-NN is often criticised for the large computational effort when evaluating the distance between all training instances and each test instance. Suguna and Thanushkodi [35] proposed GKNN which makes use of a genetic algorithm to overcome the computational requirement. In this approach, a fixed number of training instances are selected from the training set, and the distance between the training instances and test instances is computed as a fitness value. The genetic algorithm performs optimisation and the instances are classified as a result. This method benefits from the fact that fewer distance computations are performed since not all of the training instances are used for the distance computation.

3.6.2 Decision trees

Decision trees are a commonly used method to generate models which represent classifiers [21]. Decision trees are made up of a tree based structure. The top most node within a decision tree is called the *root* node. The bottom most nodes are the *leaves* and represent the classes. Each node within the decision tree which is not a leaf node represents an attribute, and *branches* connect the attributes together. A decision tree is traversed in a top-down manner from the root, until a leaf node is reached. During the traversal, each attribute is evaluated and the corresponding branch is traversed. In order to classify an instance from the test set, the root node

is first evaluated. Each attribute is evaluated based on the corresponding value from the test instance, and from the evaluation, the corresponding branch is traversed. The test instance is classified based on the value of the leaf. Decision trees represent rules and relationships which are easy to understand [36].

Kotsiantis [37] states in a review that there are two major procedures that must be followed when making use of decision trees for classification. The first procedure is one which deals with the construction of the decision tree. This procedure starts off with an empty tree, and an attribute is added one at a time. Whenever an attribute is added it essentially splits the training data into several parts based on the attribute. Thus, the goal is to select the attribute which will be best at discerning between the instances of each class. The attributes are added in turn until finally a leaf is added which represents a class. The second procedure is that of classification which represents the decision tree traversal from the root to a leaf node.

Two commonly used algorithms for creating decision trees are ID3 [38] and C4.5 [31, 39]. ID3 is a greedy search algorithm which makes use of a statistical property, known as information gain, in order to decide on which attribute to select during the execution of the ID3 algorithm. At first, the information gain for each attribute is computed and the one with the highest information gain is selected as the root. This process is repeated for each branch. Once an attribute has been selected and added to the tree, this choice is never altered, and thus ID3 is susceptible to converging toward a local optimum solution. C4.5 benefits from four improvements over ID3 [31]. Firstly, C4.5 is able to handle missing values. Secondly, C4.5 is able to create decision trees for both continuous and categorical attributes. Thirdly, it is able to handle attributes with different weights, and finally C4.5 attempts to reduce the size of the decision tree by applying prunning.

A review on creating decision trees using evolutionary algorithms is presented by Barros *et al.* [40]. The authors point out that evolutionary algorithms benefit from the fact that they can escape from local optima when evolving decision trees. Additionally Barros *et al.* state that evolutionary algorithms have the advantage of being able to easily deal with multi-objective functions. Using more than one objective function can impact the evolution of the decision trees. For instance, an evolutionary algorithm can attempt to minimise the tree size and maximise the accuracy. Or alternatively, minimise the tree size and maximise the sensitivity of the decision tree. Algorithms like ID3 or C4.5 do not naturally handle multi-objective functions. Barros *et al.* however discuss that evolving decision trees is computationally expensive and leads to long training times. The authors further mention that a disadvantage of evolutionary algorithms is the large number of parameters which need to be fine-tuned.

3.6.3 Artificial neural networks

Artificial neural networks (ANN) were inspired by the human brain and represent a network of neurons combined together in order to learn from data [10, 41–43]. Essentially, ANN deals with the computation of the output from all of the neurons within the network. A neuron can be described as an "*information-processing unit*" [44] which has several input variables along with a weight associated with each input variable. The neuron performs a function on the input values and produces an output. Figure 3.5 illustrates an example of a neuron. Haykin [44] describes a neuron as having three major components, the inputs and their corresponding weights, an adder which is responsible for adding all the inputs and their weights, and finally an activation function which outputs a value within a finite range, usually between 0 and 1. The most common activation function is a sigmoid function.

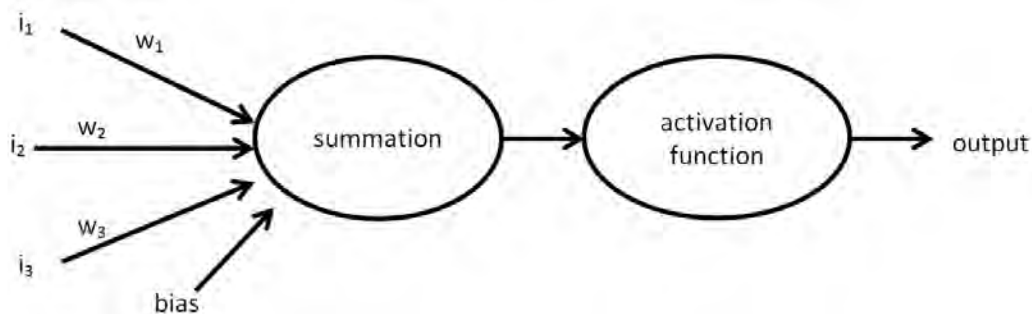


Figure 3.5: Example of a neuron.

An ANN can be made up a single or multiple layers. In a multilayer feed forward network, the input variables are connected to a hidden layer. The hidden layer itself can have additional layers. In a fully connected network, each neuron within one particular layer is connected to each neuron within the next layer. A comparison of neural networks for binary classification is presented in [45].

3.6.4 Naïve bayes

Naïve Bayes classifiers are probabilistic methods based on Bayes theorem. These classifiers have performed well [10,41,42]. This method relies on the assumption that for a given classification prediction, the value for a particular attribute is unrelated to the values of the other attributes, i.e. each attribute contributes independently to the probability of a particular classification [10]. Naïve Bayes classifiers combines the prior probability with the conditional probabilities gained from the training instances in order to compute the posterior probability that a given test instance belongs to a particular class [10]. Naïve Bayes classifiers are hindered by the fact that they can only be applied to categorical attributes, thus continuous attributes

have to be converted into categorical attributes in order to use them. A detailed example is provided by Bramer [10].

For class c_i the posterior probability is computed as,

$$P(c_i) \times \prod_{j=1}^{n} P(a_j = v_j | class = c_i)$$

where $P(c_i)$ denotes the prior probability, a_j denotes the attribute for $j = 1, 2, \ldots, n$, and v_j denotes the attribute values for the test instances for $j = 1, 2, \ldots, n$.

Bramer [10] mentions that for a particular class, if the number of instances having a particular attribute value is small, then this will result in a poor estimate in the posterior probability.

3.6.5 Evolutionary algorithms

There are additional methods which can be applied to the task of data classification other than the techniques described in the sections above. Surveys on techniques for data classification include [36, 46–48]. Although there are a wide variety of algorithms which can be applied to data classification, there is no single method which obtains the best results across any given data set. Each method has its own strengths and weaknesses, and members of the machine learning community are constantly enhancing the current state-of-the-art methods by improving on the existing limitations and weaknesses.

Evolutionary algorithms (EAs) [49] represent algorithms which are population based, stochastic in nature, and make use of operators to alter the individuals within the population. EAs iterate several times, and during each iteration, operators are applied to the population to create offspring, which in turn are evaluated using some fitness function, and finally individuals are selected to be placed in the population for the next iteration. Surveys on EAs and their application to classification is presented by Freitas [50, 51].

Freitas points out two advantages for using EAs, the first being that EAs are robust search methods which can perform a global search within a solution space, and the second being that EAs are flexible algorithms. The flexibility arises from the fact that there are several aspects of the algorithm for which one is free to incorporate their own implementation. The representations used for an individual, the fitness function and evaluation of the individuals, and the genetic operators are areas of EAs which can be adapted to the problem domain, thus providing flexibility to the researcher. On the other hand, Freitas mentions that EAs are computationally expensive methods which can result in long execution times; a long execution time can further be affected by the size of the data set. Consider a data set with thousands

of instances, and a population size of 1000 individuals. If each individual within the population is evaluated on each of the instances, it is clear that this will hinder the execution time significantly. It is however possible to overcome this by parallelising the implementation.

Genetic algorithms (GAs) [52] have also been applied to the task of data classification. The most significant difference between a GP algorithm and a GA, is that a GP algorithm searches a program space, whereas a GA searches a solution space. The two EAs therefore implement different representations. A GP algorithm typically evolves a population of parse trees which represent programs, whereas a GA evolves a population of chromosomes. Freitas [50, 51] discusses how GAs can be used to solve data classification problems. There are two principle ways in order to represent a GA chromosome for classification, namely the Pittsburgh and the Michigan-style approach. In the Pittsburgh approach, one individual represents a set of rules, and in the Michigan approach an individual represents a single rule. A rule is typically in the form "if condition then class", whereby the condition can represent a combination of several logical operators such as "*AND*" or "*OR*". One approach for encoding the conditions is as follows. Assume a data set has categorical attributes, and let one of the attributes be named X having these possible values {x, y, z}, then it is possible to represent this condition using three bits respectively for each of those values, where a value of "1" denotes that the attribute value is used, and "0" denotes that the attribute value is not used. Thus, a condition of "101" would represent the rule, *X = "x" OR X = "z"*.

Freitas [51] mentions three ways of assigning the class to the rules. The first approach is to encode the class into the chromosome and to allow the algorithm to evolve the best choice for the class for each particular chromosome. The second approach is to run the entire algorithm x number of times, where x represents the total number of classes. Each class is picked in turn, and during each repetition, that class is assigned to every rule. Thus, the best rule for each class is evolved during a repetition. The third approach is to select the class for the rules in a deterministic manner, for instance to pick the class which will result in the highest fitness for a particular individual.

Freitas mentions that "*the power of GP is still underexplored*" [50] and that "*an important research direction is to better exploit the power of GP*" [50]. Freitas' work was published in 2008 and many studies have addressed GP in the context of data classification since then; however, GP and data classification still remains an active area of research as will be noted in the following chapter.

3.7 Active Research Areas in Data Classification

The primary task involved in data classification is to create a classifier which can accurately allocate a class to instances of data. Typically, a better method is one which achieves the highest possible number of correct classifications. However, there are additional subproblems which form part of active research areas in data classification. This section will review studies which have addressed these subproblems.

3.7.1 Feature selection

A data set may not always contain attributes which are useful for building classifiers. Attributes can be described as *relevant, irrelevant, or redundant. Relevant features* are those which are useful to a classifier in predicting class values. *Irrelevant features* consist of features which can be removed from the data set and will cause no effect to the construction of a classifier [53] as they do no contribute to the accuracy of a classifier [26]. Redundant features are those which can be removed from the data set as there exists another feature which can be used in its place. When a data set contains a significant amount of irrelevant and redundant attributes the performance of a classifier may in turn be hindered, and thus feature selection methods are applied prior to the construction of a classifier [5]. Feature selection, or attribute selection, is defined "*as a process of finding a subset of features, from the original set of features*" [26]. Once a feature selection method has been applied, the expectation is that a classifier built upon the reduced data set will perform better [5].

There are three main categories of feature selection methods, *filter*, *wrapper*, and *embedded* methods. Filter methods are those which are applied prior to the construction of the classifier. Once the features have been obtained they serve as input for the construction of the classifier [54]. Features are selected without taking into consideration the performance of these on the classification task [5, 26, 54]. Filter methods are not as computationally expensive as wrapper methods [26]. Wrapper methods use a classification algorithm to determine how good the selected features are by measuring their performance [53]. In this approach several subsets of features are created and these subsets are evaluated by constructing a classifier. Search methods are used in order to obtain the optimal subset [53]. Embedded methods combine the search for an optimal subset of features and the construction of the classifier into a single process [53] and are computationally less expensive than wrapper methods [54]. Saeys *et al.* [54] present a taxonomy for the three methods, and Liu and Yu [55] present a generalisation of the three algorithms.

There exists a vast quantity of literature available on this topic. For the remainder of this section several methods will be reviewed to present an overview of feature selection methods and their application to data classification.

Pujari and Gupta [56] make use of the Pearson Chi-square measure to determine useful features. The Ionosphere data set was used and a list of features that were used is presented. A total of 20 attributes were considered useful and 14 attributes were considered less useful and were omitted. An ensemble model which in turn used CART, CHAID and QUEST (decision tree algorithms) was implemented. The results show that when the useful attributes are used there was an improvement in accuracy in certain methods.

GAs have also been applied to the feature selection problem. The work of Yang and Honavar [57] used GAs to encode chromosomes having a length equal to the number of attributes. The genes consisted of 0's and 1's and if a 1 is encoded then that attribute was selected. The selected attributes were then used as input for a neural network. The neural networks were evaluated using a fitness function which took into account the accuracy and a cost associated with classifying each attribute present. A comparison was made between using the accuracy only as opposed to using both the accuracy and a cost; the latter obtained better results. There was a reduction of approximately 50% in the number of attributes and the accuracy was better than in the case when all attributes were used.

Jacob and Ramani [58] implemented several feature extraction techniques. The *breast tissue* data set was used and there was a reduction from 9 attributes to 3 when the ReliefF feature selection algorithm was used.

Relief [59] is a commonly used filter feature selection algorithm. This method is used to obtain relevant features from binary classification problems. Given a sample of instances, a random instance is selected, say x, and two additional instances are obtained. The first is called the near hit, a randomly selected instance (within a certain Euclidean distance) from the same class as instance x, and the second is called the near miss, which is a randomly selected instance from the opposite class to instance x. A feature weight vector is computed based on the instance x, the near hit, and near miss. This feature weight vector is updated in order to determine those features which are relevant or not based on a threshold value. Relief is executed in linear time thus making it a rapid filter method. ReliefF [60] is an extension of Relief and is able to handle multiclass problems as well as noisy and incomplete data [53]. The original Relief algorithm finds a single random near hit and near miss, whereas ReliefF benefits from selecting several near instances from each class and thus can handle noisy data by averaging those values, and additionally cater for multiclass problems.

3.7.2 Missing values

Missing data in data sets is problematic, and a decision needs to be made as to how this will be addressed. There can be several reasons for data to be missing, such

as malfunctioning measuring equipment, or a respondent refusing to answer certain questions in a survey, or possibly even due to the fact that a questionnaire was too long and a respondent chose to stop midway [5] [61]. Typically missing values can be handled in two ways: to discard them, or to impute values in the place of the missing values.

3.7.2.1 Discarding missing values

A simplistic approach to handling missing data is to discard those instances or features from the data set [10, 26, 62]. This approach has consequences as it could affect the reliability of the results [10] and should only be applied when the amount of missing data is relatively small in compared to the size of the data set [10,26,62,63], and when the missing data is *"missing completely at random"* [62, 64]. Discarding instances containing missing values is not considered to be a recommended approach [10].

3.7.2.2 Imputation

Imputation methods are used to compute a substitute value for the missing data [61]. There are several approaches and these will be briefly discussed here. The first approach is to simply obtain the mean value for each attribute that contains a missing value, and to replace the missing values with the mean [10, 26]. If the attribute is numerical, then the statistical mean is computed, however if the attribute is nominal then the statistical mode is calculated. Cios *et al.* [26] describe another approach named *hot deck imputation.* In this method the most similar instance to the one with a missing value is found and used to replace the missing values.

Imputation methods can be categorised as being either a single or a multiple imputation method. Additional single imputation methods are discussed in [61]. McKnight *et al.* [61] point out that single imputation methods tend to be problematic and introduce bias. The alternative is to use multiple imputation methods which generate more than one imputation value.

3.7.2.3 Missing values and decision trees

Statistical approaches represent one way of dealing with missing values, however there are alternative methods which can be applied when using machine learning algorithms for classification problems. When dealing with decision trees, the traversal is based upon the value of the attribute, and thus when a value is missing the question becomes - which branch to choose?

Twala *et al.* [65] developed *missingness incorporated in attributes* (MIA). One of three choices is made when creating a split within a branch. For some cut-off

point P and an attribute value x, whereby a decision as to which branch to select is currently being decided for x, one of the following options can be applied:

- Follow left branch if $x \leq P$ or x is a missing value, otherwise follow right branch
- Follow right branch if $x > P$ or x is a missing value, otherwise follow left branch
- Follow left branch if x is missing, otherwise follow right branch

This method can also be applied to nominal values whereby the comparison is now to check if $x \in P$ or $x \notin P$. Twala [66] compared several techniques for dealing with missing values when using decision trees.

3.7.3 Ensemble classifiers

An ensemble is defined as a set of classifiers which are used to classify data by combining their individual decisions to produce a final classification decision for each instance of data [67, 68]. Figure 3.6 illustrates an example of an ensemble classifier. Several independent classifiers are created from the training data, and their output is combined in order to allocate a class to the instances of the test data. Instead of creating a single classifier which is responsible for classifying an entire data set on its own; an ensemble benefits from the fact the several classifiers work together in order to solve the classification problem.

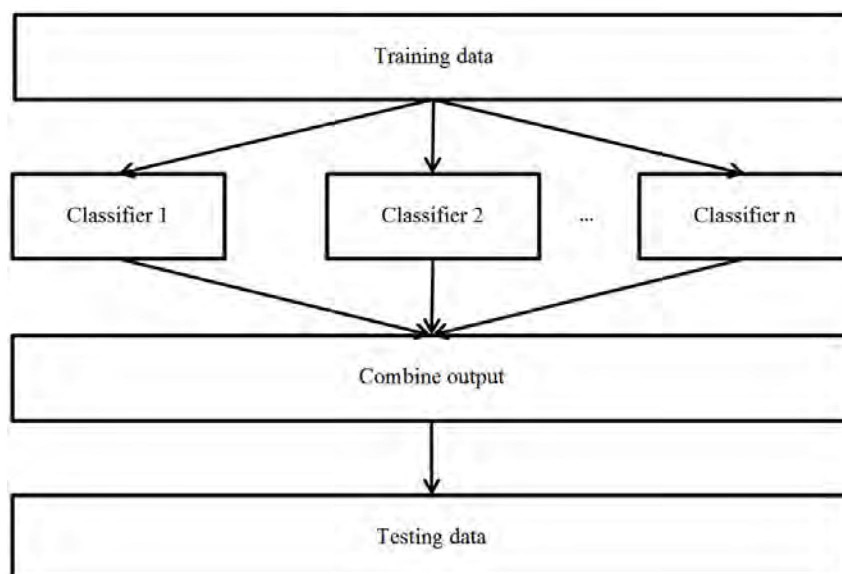


Figure 3.6: Example of an ensemble.

The most common ensemble methods are bagging and boosting. Bootstrap aggregating (bagging) was introduced by Breiman [69]. This method starts by creating multiple new training sets by resampling the original training data with replacement. Thus, the new training sets can have duplicate instances of data, and additionally, there can be several instances in common amongst the sets. Breiman's implementation of bagging proceeded as follows. Breiman split each data set into a training set which consisted of 90% of the data, and the remaining 10% was placed into the test set. The training set was then used to create 50 new training sets by randomly selecting instances - with replacement - and adding them to each new training set. For each of the 50 created training sets, a classifier was constructed using 10-fold cross-validation. Each classifier was then applied to the test set to predict the class values for each instance. Each of the 50 classifiers placed their vote, and the class with the highest relative majority became the class for that specific test instance; this was performed for each instance in the test set. The entire process of evolving the 50 classifiers and evaluating them on the test set was repeated 100 times, and the results indicate that bagging reduces the misclassification rate.

Furthermore, Breiman investigated the effect of the size of the new training sets. In the experiments described above, the new training sets were the same size as the original training set. Thus, if the original training set had 100 instances, 50 new training sets were created having 100 instances each. When new training sets were created consisting of double the size of the original training set there was no improvement in the results. Additionally, the quantity of new training sets that were needed to be created was examined, and the results show that there was no significant reduction in misclassification rate when more than 25 sets were used.

Boosting algorithms [70, 71] allocate weights to the instances in the data set, and use those weights to train the classifiers. Initially all the weights have an equal value, and they are modified as the learning algorithm iterates. During each iteration a classifier is built. The construction of the classifier is influenced by the weights. The concept behind this algorithm is to allocate the weights based on the difficulty the classifiers have in allocating the correct class label to an instance of data. If an instance of data is correctly classified, it receives a lower weight, on the other hand, those instances which are difficult to classify receive a higher weight. Thus, during each iteration the weights influence the construction of the next classifier; by creating one which will be biased towards those instances which were harder to classify by the previous classifier. The performance of each classifier will affect how the next classifier is constructed based on the influence of the weights. Boosting is typically used to create an ensemble of weak classifiers; although, this may not always be the case.

Freund and Schapire proposed AdaBoost [31, 70, 72] which is a commonly used

boosting algorithm for creating ensembles. Each instance is allocated a weight of $\frac{1}{n}$ where n represents the number of training instances. The algorithm iterates for several rounds, and during each round, a classifier is trained based on the current weights of the instances. The error is computed between the correct output and the output obtained by the classifier, and it is then used to update the weights in such a way that they are normalised. An instance which is misclassified receives a greater weight. Once all the weights have been updated, the next round is executed and a new classifier is created. Liu *et al.* [73] proposed BoostGA which evolves a population of rule conditions using a genetic algorithm. BoostGA is based on AdaBoost and is run over several rounds. During each round a classifier is created and the weights are updated. The proposed method was tested on two data sets and the BoostGA improved the classification accuracy.

3.7.4 Discretisation

Certain classification algorithms, such as ID3, cannot directly handle continuous valued attributes; instead they can only handle categorical data [74]. In order for these algorithms to work with data sets containing numerical attributes, these attributes need to be transformed in such a way that the algorithm is presented with discrete and finite intervals. Discretisation is the process of transforming a continuous attribute into a discrete one with a finite number of intervals [74, 75]. Discretisation is essentially needed when dealing with decision trees since each node in the tree should branch out based on the data. It is not feasible to create a branch for each continuous value as this may result in nodes having hundreds of branches, and consequently creating a massive search space for the learning algorithm.

An interval is created by a cut-off point. Liu *et al.* [75] define a cut-off point, as a single real value within the continuous values for an attribute which separates the values into two intervals. Let x be a cut-off point, then if the minimum value for an attribute is a and the maximum value is b applying x to the attribute will result in $[a, x)$ and $[x, b]$. The order in which the open interval is allocated is not an issue, thus instead the algorithm could split the data into $[a, x]$ and $(x, b]$. An algorithm can further split those two intervals into additional intervals, and so on.

A recent survey by Garcia *et al.* [74] provides a complete and thorough taxonomy of discretisation methods. A comparison network of discretisation methods is presented, and the methods which are most compared in literature are equal width, equal frequency, chiMerge [76], MDLP, and ID3 [77]. A comparison the performance of 30 discretisation methods with 6 classifiers on 40 data sets from the UCI machine learning repository was performed. The authors state that the best performing methods are the chiMerge, MDLP, distance, chi2 [78], and modified chi2 [79]. Chi2 is an enhancement to the original chiMerge algorithm. Garcia *et al.* state that a

conclusion as to which discretisation method is the best cannot be made. The research of Garcia *et al.* serves as a baseline for comparison when new methods are created and can also serve as a starting point for researchers wanting to implement discretisation in their machine learning algorithms. Garcia *et al.* [74] present an overview of the main characteristics of a discretiser which is briefly listed here:

- Static discretisation is one which is done before the learning algorithm is applied. It is independent from the creation of the classifier. On the other hand, dynamic discretisation takes place while the classifier is being created.
- Univariate discretisation considers a single attribute at a time to create the intervals, whilst multivariate discretisation takes all the attributes into consideration.
- When the class labels are not taken into consideration the method is referred to as unsupervised. A supervised discretisation algorithm is one which takes the class label into consideration.
- A discretiser which is based on splitting selects the best cut-off points amongst all the possible ones to create intervals. On the other hand merging consists of removing cut-off points and consequently merge two adjacent intervals.

Several researchers have investigated the use of entropy based discretisation in order to obtain the cut off-points for the intervals. The entropy is used to evaluate the cut off-points and is applied recursively on each attribute until some stopping criteria is met. Hacibeyoglu *et al.* [80] implemented an entropy based discretisation method using k-NN, Naïve Bayes, C4.5 and CN2 classification algorithms. Candidate cut-off points were selected and used to split the data for a particular attribute into two sets. A formula is then used to determine which cut-off point is most suitable. This formula computes the entropy of these two sets by taking into consideration the instances of the attribute for each class. Once a suitable cut off-point is obtained this procedure is applied recursively on each attribute until some stopping criteria is met. The classification algorithms were tested on 6 data sets using all the original attributes and compared to the discretized attributes. On each data set, the four classification algorithms were applied, and on 19 of the 24 cases there was an improvement when using the discretized attributes. A maximum improvement of 26% was observed when using the k-NN algorithm on the Statlog (Heart) data set.

In the survey of Kotsiantis and Kanellopoulos [81] the term *Adaptive Discretisation Intervals* (ADI) is presented. ADIs alter the intervals during the evolutionary process, and was first introduced by Bacardit and Garrell [82] where a GA was used. In the approach of Kotsiantis and Kanellopoulos, a chromosome was made up of

micro-intervals. A micro-interval represented an interval. Two operations were performed, merging and splitting. Merging consisted of combining two micro-intervals whereas splitting consisted of randomly selecting a micro-interval to create two new intervals. Additionally a multi-adaptive approach was examined where an attribute can contain a different number of micro-intervals thus allowing the evolutionary process to select the most suitable number of micro-intervals for each attribute. The proposed method was tested on 8 data sets. The adaptive approaches obtained higher classification accuracies on 6 data sets when compared to a simple uniform width interval approach and to the method of Fayyad and Irani [83]. Bacardit and Garell further improved the research in [84] where uniform and non-uniform width intervals are allowed. This new approach ADI2, outperformed the original approach.

Anguilar-Ruiz *et al.* [85] present a comparison between three evolutionary algorithms (HINDER, ECL, GAssist) which evolve the intervals along with classification rules. A total of 6 discretisation methods were applied to the three evolutionary algorithms, including ID3 [77], Fayyad and Irani [83], USD, and a random discretiser. The random discretiser selects a random subset of midpoints within the values of an attribute. It obtained the lowest average accuracy and also resulted in the most number of cut off points. The best results were obtained using ID3, Fayyad and Irani's approach, and USD. The research of Bacardit and Garell [82,84] and Anguilar-Ruiz *et al.* [85] both present adaptive discretisation techniques applied to evolutionary algorithms and serve as a comparative study for future research in adaptive discretisation.

3.8 Software

There are two open source computer programs which are of particular interest in the domain of data mining and classification, these are Weka and KEEL. These programs allow researchers to make use of a wide number of data mining algorithms. These algorithms include classification, clustering, feature selection and pre-processing algorithms. A list of available software for data mining tasks is presented in [86].

Witten *et al.* defined Weka (Waikato Environment for Knowledge Analysis) as "*a collection of state-of-the-art machine learning algorithms and data pre-processing tools*" [5] developed at the University of Waikato. Weka [87] contains a large number of classification algorithms. Amongst the available types of algorithms Weka contains Bayes classifiers, trees, classification rules, functions and other machine learning algorithms. Weka is commonly used by the machine learning community and references to Weka are often found in literature when comparing algorithms. When a new method is developed, the results obtained by this new method can be compared to those obtained by an algorithm in Weka. This comparison can serve

as an initial estimate of the quality and performance of the new algorithm.

KEEL [86, 88] is an open source software and, similarly to Weka, is a software written in Java to allow researchers to perform data mining operations. This software has a number of built in evolutionary algorithms. Additionally in a similar manner to Weka, KEEL is able to perform pre-processing tasks such a feature selection.

There exists a number of additional computer programs for data mining which have been developed in a variety of programming languages other than Java, such as scikit-learn [89] which was developed in Python.

3.9 Conclusion

This chapter describes and covers the main areas of data classification. In supervised learning, a vast number of real world problems can be formulated as classification problems, such as determining which candidate will win a presidential election based on data collected through surveys, and by mining other sources of information. Classification problems can be formulated as being either binary or multiclass problems. Classifiers are created from the data sets. Each data set is made up of numerous instances, which have attributes and a class. Creating a classifier using an imbalanced data set is significantly more challenging since a classifier will be biased towards the majority class. The performance of classifiers can be determined using different performance measures, with accuracy being the most commonly used measure. There are several ways of separating data sets for training and testing. These methods vary both in complexity and computational effort required to evaluate the classifiers. The train/test split is computationally inexpensive as a single training set is created. However, this method can be affected by the way the data is randomly split. In order to overcome this, the stratified holdout can be performed. The k-fold cross-validation and the leave-one-out methods are significantly more computationally expensive in comparison to the train/split method; however, the 10-fold cross-validation is frequently used, as the performance across multiple runs are averaged thus decreasing the variation in the results. When dealing with data sets which contain a smaller number of instances, the leave-one-out approach is recommended in order to train the classifier on as many instances as possible.

Data classification has been investigated using a variety of techniques. In this chapter several of the existing methods have been presented. Common methods for data classification include K-nearest neighbour, decision trees, artificial neural networks, Naïve Bayes and genetic algorithms. Data classification has been an active area of research over the years and is still being investigated.

Active areas of research include feature selection, missing values, ensembles and discretisation. Feature selection is often performed as a pre-processing step and deals

with determining which of the attributes are most useful for the classification task. Data sets which contain missing values present a challenge in creating accurate classifiers. A simple way of handling such data sets is to compute the average value for each attribute, and to replace each missing value with the corresponding attribute average. Creating ensembles is another active area of research. Researchers that have used ensembles have shown that these classifiers often outperform single classifier methods.

When a data set is made up continuous valued attributes, discretisation is required to transform the continuous data into intervals when creating a decision tree. The intervals are most commonly created using statistical measures. Studies have shown that intervals can be created during the training phase. In terms of evolutionary algorithms, discretisation has only been used with GAs. Finally, this chapter introduced common software packages for data classification.

Chapter 4

Genetic Programming and Data Classification

4.1 Introduction

The previous two chapters focused on applying GP and data classification individually. This chapter reviews previous studies on GP which were applied to data classification problems. A GP representation has to be chosen before using GP to solve any type of problem. Three common GP representations have been used in previous data classification studies. Each of these representations use different function and terminal sets, and are applied to the task of data classification in different ways. Sections 4.2, 4.3, and 4.4 describe how GP has been used to evolve trees using the three major representations for data classification. This is followed by section 4.5 which describes other GP representations which have been used in the literature. In the previous chapter, ensembles were described as methods which usually produced classifiers with greater accuracy compared to non-ensemble methods. Studies which have used GP to evolve classifier ensembles are described in section 4.6. The applicability of GP to data classification presents certain advantages and disadvantages, these are discussed in section 4.7. Finally, this chapter is concluded in section 4.8.

4.2 GP and Decision Trees

Recent studies in GP which have made use of decision trees generally categorise these trees as being either axis parallel or oblique decision trees [40]. In the following explanations, let x and y represent two attributes in some data set. Figure 4.1 illustrates the two types of decision trees. Axis parallel decision trees have a single attribute at each non-leaf node within the decision tree. Thus, during the evaluation

of each of these trees, only a single attribute is tested at a time and a decision as to which branch to follow is made. Axis parallel decision trees are the most commonly used decision tree representation. Let two classes be represented using $+$ and $-$, thus in figure 4.1, the two graphs below the decision trees illustrate how data is partitioned in the case of axis parallel and oblique decision trees; these graphs however do not represent the corresponding trees in the figure. In the case of axis parallel decision trees, the data is partitioned into spaces which are parallel to the axes. For oblique trees on the other hand, the data is partitioned into spaces according to the linear combination of the attributes.

Oblique decision trees, also referred to as linear multivariate trees, can have more than one linear attribute at each non-leaf node [90]. For instance, a non-leaf node may be made up of the decision $x + y < 1$ as illustrated in figure 4.1. For such decisions, if the linear expression is evaluated to true, then the left branch is followed, and if the expression is evaluated to false then the right branch can be followed.

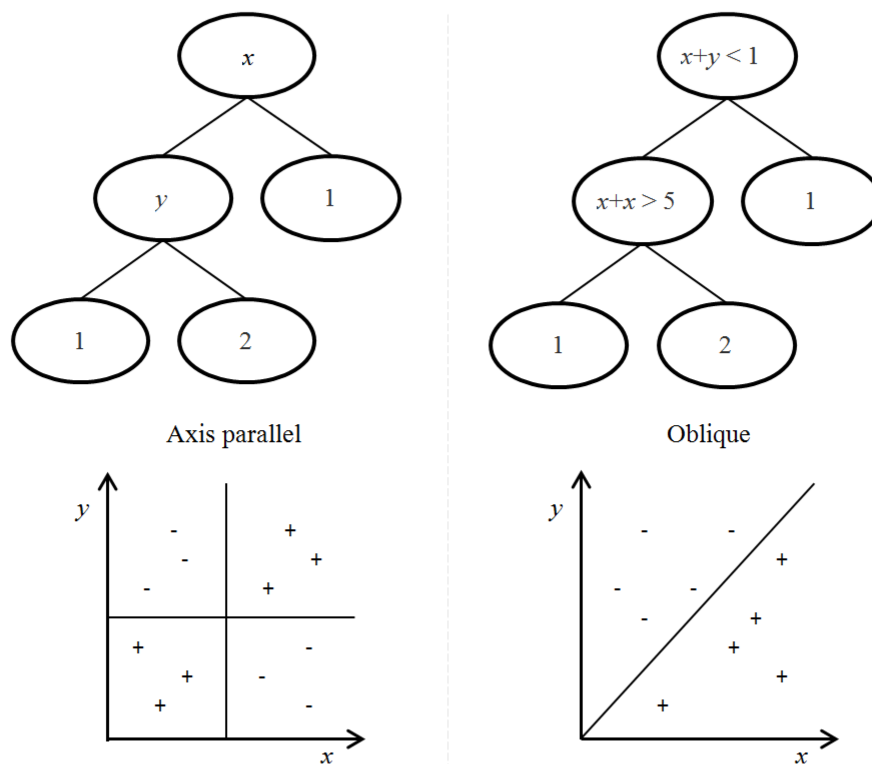


Figure 4.1: Axis parallel and oblique decision trees, along with graphs illustrating how the data is partitioned in the two representations. The graphs do not represent the partitioning of the data for the corresponding trees.

Tur and Guvenir [91] made use of GP to evolve a population of 100 axis parallel decision trees. The fitness function considered both the size in terms of the number

of nodes, and the accuracy of the classifier. Furthermore, the fitness function made use of weights to select whether the size or accuracy had a greater impact on the fitness of a tree. The method was applied to a single binary classification problem. In order to validate the proposed method, the data set was split with 69% of the instances in the training set, and 31% of the instances in the test set. In this study, the attributes formed the function set, and the classes formed the terminal set.

From the studies examined in the review by Espejo *et al.* [92], it is apparent that researchers commonly use a fitness function which takes into consideration both the accuracy and size of the decision trees. In the study conducted by Shirasaka and Zhao [93], GP was used to evolve 200 axis parallel decision trees over 500 generations. The size of the decision trees was not used in the fitness function; it was, however, used when comparing individuals during selection. If two trees had the same fitness, then the smallest tree was considered the better individual. The purpose of this research was to emperically determine the performance of GP decision trees on a character recognition data set, and the findings revealed that the decision trees obtained good results.

Koza [94] shows how GP can be used to evolve 300 axis parallel decision trees when solving the Saturday Morning problem presented by Quinlan [38]; this problem is a binary classification problem. As in other work, such as the study by Khoshgoftaar and Seliya [95], the function set and terminal set was composed of the attributes and classes respectively. For each non-leaf node in the population, a single attribute is compared to a constant using the $<$ operator. In this study, 1000 axis parallel decision trees are evolved over 200 generations. Additional parameters include a crossover rate of 60%, a mutation rate of 30%, and a reproduction rate of 10%. The fitness proportionate selection method was used. There was no attempt at optimising the GP parameters. The proposed GP approach was tested on a single binary data set, and in order to validate the model, the training data contained 66% of the total instances, and the test set contained 34%. A similar approach of representing the non-leaf nodes was proposed by Wang *et al.* [96]. GP was used to evolve 900 axis parallel decision trees over 50 generations. Each non-leaf node had an arity of 2, and was in the format $attribute \leq constant$.

Estrada *et al.* [97] use GP to evolve a population of 1000 axis parallel trees over 60 generations. Once again, the attributes formed the function set, and the classes formed the terminal set. The fitness function considered both the accuracy and the size of the GP individuals in terms of the number of nodes. Tournament selection with a size of 7 was used. A 90% mutation rate was used, along with a 10% reproduction rate. The GP classifiers were validated using 10-fold cross-validation on data which was generated by the authors.

Oblique decision trees were used by Bot and Langdon [98]. In order to ex-

press the linear combination of attributes, three functions were used in the function set, namely *CheckCondition1Var*, *CheckCondition2Vars*, and *CheckCondition3Vars*. *CheckCondition2Vars* is illustrated in figure 4.2. The first two pairs of nodes denote the coefficients and attributes, c_1 denotes the coefficient for attribute x_1, and c_2 denotes the coefficient for attribute x_2. The node *val* denotes the value to which the linear combination is being compared to in the expression $c_1x_1 + c_2x_2 \leq val$. Finally, r_1 and r_2 denote which branch to follow next based on whether the expression has been evaluated to true or false. A steady state GP was evolved over 1000 generations with a population size of 200 individuals. Tournament selection with a size 7 was used. The evolved trees were validated using 10-fold cross-validation and tested on four data sets from the UCI repository.

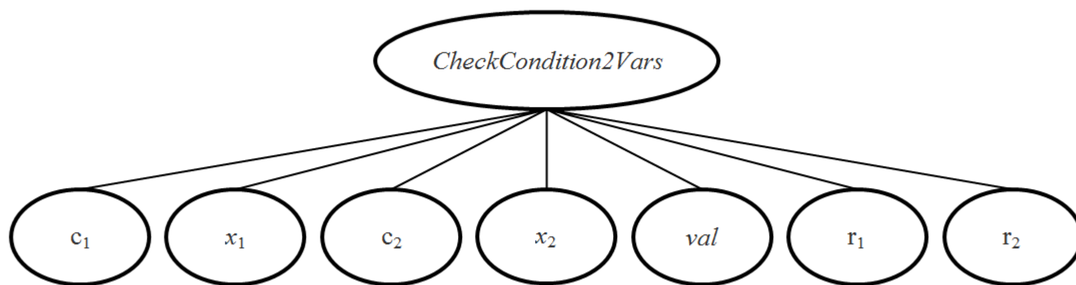


Figure 4.2: *CheckCondition2Vars*, function for an oblique decision tree.

Shali *et al.* [8] used GP to evolve 200 oblique decision trees over 100 generations. In comparison to the study by Bot and Langdon, Shali *et al.* did not encode a new type of node to express linear combinations. Instead, mathematical functions $\{+, -, \times, /, ln, \surd\}$, two logical functions NAND and NOT, and the relational operator $\leq$, were used in order to represent the linear expressions. These functions and the relational operator formed the function set. Each node within the decision tree represents a linear expression, with two branches extending from each node. The left branch is followed if the expression is evaluated to *false,* and the right branch is followed if the expression is evaluated to *true.* An example of a linear expression is given by $X + 2\sqrt{Y} \leq W$, where *X, Y,* and *W* are numerical attributes for some classification problem; figure 4.3 illustrates the expression.

The classes formed the terminal set. The evolved classifiers were validated using 5-fold cross-validation on 19 data sets from the UCI repository. Tournament selection with a size of 15 individuals was used. The fitness function made use of the gain ratio measure and took the size of the trees into consideration. Crossover was applied with a probability of 0.65, and mutation with a probability of 0.2.

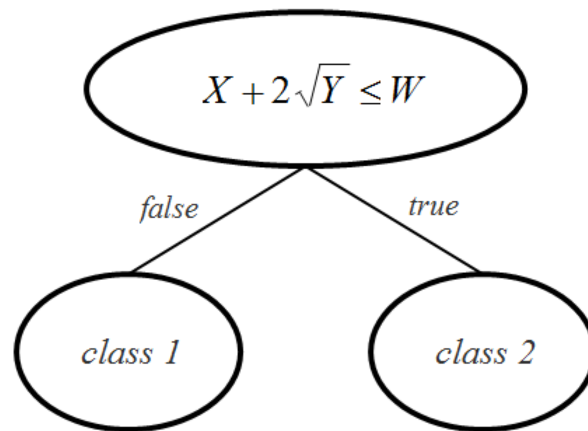


Figure 4.3: Oblique tree used in the study of Shali *et al.* [8].

4.2.1 Advantages and disadvantages of GP decision trees

The primary advantage of decision trees when evolving classifiers using GP is the simplicity involved in understanding the evolved classifiers. At each node within the tree, a decision needs to be made as to which branch to follow based on the evaluation of that node. Thus, if a path from the root node to a leaf node is made, it is possible to evolve a sequence of basic "rules". For instance, consider the decision tree in figure 4.4. It is possible to create the following four rules:

- If X is true, AND Y is true, then class 1
- If X is true, AND Y is false, then class 2
- If X is false, AND Z is true, then class 3
- If X is false, AND Z is false, then class 1

From these four rules it is possible to combine the first and the last as they both output the same class. This can assist in discovering rules for complicated data sets whereby the classifier itself can provide some knowledge from the rules. Oblique decision trees are not as easily comprehensible as axis parallel decision trees, due to the fact that the linear combination of attributes makes it more challenging to interpret the classifier.

Another advantage of decision trees is that only a single path from the root to a leaf node is required. Since not every node within the decision tree has to be evaluated, this can result in rapid evaluations. The amount of time required to evaluate a decision tree is dependent on its depth; shorter trees are evaluated more quickly as there is a shorter path from the root to a leaf node, and the converse holds for decision trees with a large depth.

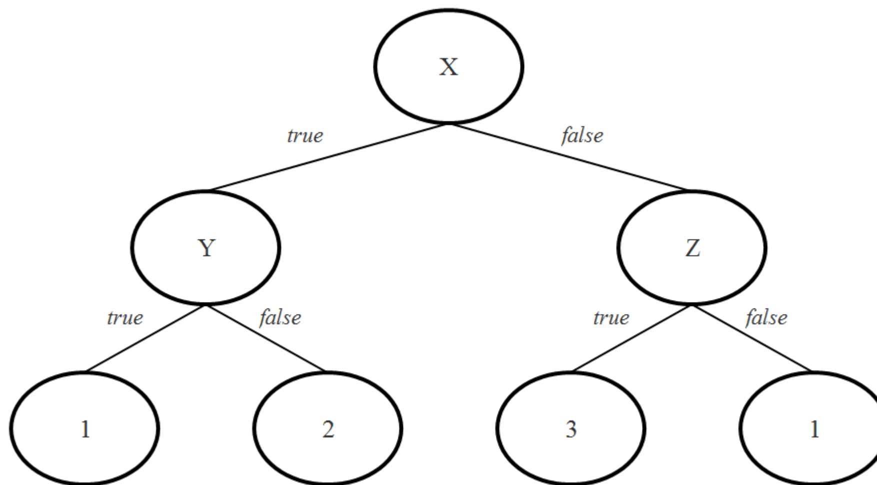


Figure 4.4: Axis parallel decision tree.

Discretisation is required in order to deal with continuous data when a decision tree is evolved [74]. This additional step represents a disadvantage as it results in further complexity which the other representations do not suffer from. Decision trees can handle categorical attributes naturally and are often used when data sets contain only categorical data. Decision trees suffer from the problem of overfitting, whereby the trees can grow very large and become too specific to the training data during the evolutionary process. As a consequence, the trees will generally perform well on the training data and poorly on the testing data.

4.2.2 Summary of the findings

When using GP to evolve decision trees, the axis parallel decision tree representation is commonly used. This representation is simple to implement and offers easy interpretability. From the work presented in this section, the function set is commonly composed of the attributes within the data sets, and the terminal set is composed of the classes. In terms of the GP parameters, selection methods, and genetic operators, there is no consistency between the previous studies. For instance, the population size varies from 100 to 1000. In terms of the initial population generation method, only the work of Bot and Langdon [98] mentioned the use of the ramped half and half method. In the other studies there were no details about the initial population generation method. There was no discussion as to how the GP parameters were obtained. Certain studies made use of the train/split validation, others used 5- and 10-fold cross-validation, and in other studies there is no mention as to how the classifiers were validated. There is no consistency in the number of data sets used, and in certain studies the data sets were not publicly available

ones. The primary aspect to be extracted from the previous studies is that decision trees are easily interpreted, and consequently they are the favoured representation as researchers are easily able to understand the classification models evolved.

4.3 GP and Arithmetic Trees

This section will review those studies which make use of both GP and arithmetic trees, also referred to as discriminant functions, to evolve classifiers for classification problems. Arithmetic trees represent mathematical expressions which can discriminate between classes. Figure 4.5 illustrates an example of an arithmetic tree. When evolving decision trees using GP, there is no need to change the GP algorithm in order to use either binary or multiclass data sets. When using arithmetic trees however, changes to the GP algorithm are required when switching between the use of binary and multiclass data sets, and thus the two cases are reviewed separately.

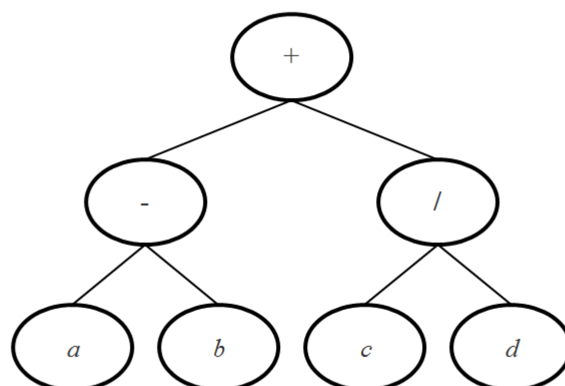


Figure 4.5: An arithmetic tree.

4.3.1 Binary classification

Etemadi *et al.* [99] use GP to evolve arithmetic trees for a binary classification problem in terms of bankruptcy prediction. A 0/1 rounding threshold with a value of 0.5 was applied in the following manner. For a particular instance of data, if a tree outputs a value greater or equal to the rounding threshold, then that instance of data is classified as a bankrupt firm, otherwise the instance is classified as a non-bankrupt firm. The researchers do not mention how the rounding threshold value of 0.5 was obtained. Figure 4.6 illustrates the 0/1 rounding approach. Different threshold values may be used, and it is also possible to use a different range of values other than 0 and 1. For instance, it is possible to use a range of $-\infty$ to ∞ with a threshold value of 0.

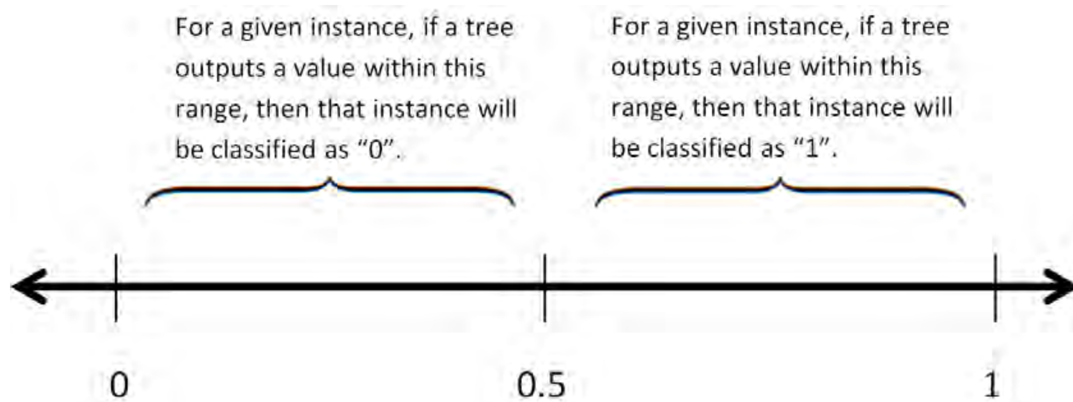


Figure 4.6: Illustrating how to map the output of a GP tree onto two classes using a threshold value. In this figure, the threshold is 0.5.

Additionally, in the study of Etemadi *et al.* the function set that was used was $\{+, -, \times, \hat{}, NOT, LT\}$. The NOT operator has an arity of one and is applied to an attribute. It returns the result obtained by subtracting the attribute from 1. The LT operator has an arity of 2 and is applied to two variables. It returns a value of 1 if the first attribute is smaller than the second attribute, and if not, 0 is returned. The attributes and constants formed the terminal set. The data set was split with 72% of the instances in the training set, and 28% in the test set. Crossover was applied with a probability of 0.6 and mutation with a probability of 0.06. No further details regarding other GP parameters were given. The proposed GP approach was compared to a multiple discriminant analysis model developed by the authors; GP obtained better accuracy.

Gray *et al.* [100] evolved GP trees in order to classify brain tumours. GP was tested on a single binary data set, and a threshold of 0 separated the output of the trees. For a given tree, a positive output represented the non-meningioma class, and a negative output represented the meningioma class. The function set used was $\{+, -, \times, /, tan, myAND, myOR, myNOT\}$, where the logical operators return either 0 or 1 based on their logical evaluation. The researchers point out that the tan function was not present in the best individual and that only three attributes were present. This indicates that GP is indirectly able to perform the task of feature selection and to make use of the most relevant attributes. The fitness proportionate selection method was used, and the population was generated using the ramped half and half method. The terminal set was composed of the attributes. Two experiments were performed on a single data set which is not publicly available. In the first experiment, the entire data set was used as the training set, while the second experiment used 87% of the data for training, and the remaining 13% for testing. Seven hundred trees were evolved over 41 generations in the first experiment, and

200 were evolved over 20 generations in the second. There is no mention as to why the population size and the number of generations were varied. The standardised fitness was used as a measure of fitness, and was defined as the total number of instances minus the total number of correctly classified instances.

Bhowan *et al.* [101] evolved 500 GP trees over 50 generations in order to solve classification problems with unbalanced data sets. A threshold value of 0 was chosen to distinguish between the two classes. The function set used was $\{+, -, \times, /, if\}$. Attributes and random constants formed the terminal set. The *if* function takes three arguments which represent three branches in the node. The first argument is evaluated, and the second branch is followed if the argument is evaluated to a negative value, and the third branch is followed otherwise. Crossover, mutation and elitism [102] had application rates of 60%, 35% and 5% respectively. Tournament selection with a size of 7 was used. The proposed approach was tested on six publicly available data sets from the UCI repository, with 50% of the data used for testing.

Hennessy *et al.* [103] investigate the use of GP to evolve 2000 trees over 50 generations for a binary classification task of determining whether or not a solvent is present in a mixture of solvents in Raman spectra. The data set used had 1024 attributes and only 24 instances. The training set contained 58% of the instances of data, and the remaining 24% was used for testing. The function set $\{+, -\}$ was used. Hennessy *et al.* state that other functions could have been used; however the two selected functions were sufficient in order to achieve high accuracy. The attributes formed the terminal set. A threshold value of 0 was applied in order to map the output of an individual to either *presence* or *absence* of a solvent. The fitness function chosen takes two aspects into consideration. The first is the classification accuracy, and if all the training instances are correctly classified, the second aspect is examined. This second aspect is a measure of certainty, and is determined by finding the minimum absolute value of the output on all the training instances for a given individual. The authors do not compare the performance of the proposed approach without the measure of certainty, thus it is unclear as to whether or not the measure of certainty impacted the overall performance of the GP algorithm.

The work by Li and Ciesielski [104] shows that by modifying the function and terminal set, GP can be applied to different problem domains. In this study, GP is used to evolve 100 trees over 2000 generations in order to distinguish between squares and circles within an image classification context. This dissertation does not deal with image processing. Image classification is a data classification problem in the sense that each pixel in the image can be converted into numerical data. Li and Ciesielski investigate the use of loops. A loop takes three arguments, where the first two correspond to positions within an image, and the third corresponds to a function which will be applied to the data between the two positions. Two loop func-

tions were researched, namely the *PlusMethod* and the *MinusMethod* function. For example, if the loop is executed with positions 10 and 15 on a particular image using the *PlusMethod* function, then the result will be a numerical value representing the sum of pixels between those two positions. The function set was $\{+, -, ForLoop\}$, where + and − denote mathematical addition and subtraction. The terminal set used was $\{RandDouble, RandPosition, PlusMethod, MinusMethod\}$ where *RandDouble* generates a random double between 0 and 100, and *RandPosition* generations a random integer between 0 and 255. Mutation had an application rate of 28%, with crossover and elitism having rates of 70% and 2% respectively. The initial population was generated using ramped half and half with a maximum tree depth of 7. The fitness proportionate selection method was used. A threshold value of 0 was applied, whereby a positive output from a GP individual represented a square, and a negative output represented a circle. The images did not come from a benchmark data set, but were instead generated.

GP has been applied to the classification task of determining if two proteins interact or not in [105]. Garcia *et al.* evolved 1000 trees over 50 generations. The maximum tree depth was set to 17 and tournament selection of size 7 was used. Accuracy was used as the fitness function. Crossover, mutation, and reproduction had an application rates of 50%, 40% and 10% respectively. Preliminary runs were performed in order to optimise the parameters. A threshold value of 0.5 was applied when comparing two proteins using the following formulation:

if(function) ≥ *0.5* then the two proteins interact functionally,

else the proteins do not interact

whereby the parameter *"function"* defined in the *if* statement represents a GP tree. The function set was $\{+, -, \times, /, \geq\}$, and the terminal set was composed of the the attributes, along with an ephemeral random constant (ranging from $[0, 1]$). The data set was split evenly with 50% of the data used for training and 50% for testing.

Agnelli *et al.* [106] use GP to evolve 5000 trees over 50 steady state generations in order to classify segments from 102 scanned documents. Segments of images and text were extracted from these documents resulting in a total of 821 instances of data. The aim was to distinguish between textual and a graphic segments. The trees were trained to allocate a positive value for image segments and a negative value for text segments, thus the threshold was set to zero. The GP approach was tested on a single data set and obtained high classification accuracy. The initial population was generated using the ramped half and half method, with a maximum tree depth of 17. Tournament selection with a size of 7 was used. Crossover had an application

rate of 90%, and mutation 10%. The function set was $\{+, -, \times, /, 2^x, if\}$, unary minus, and an ephemeral random constant in the range of $[0, 11]$. The attributes formed the terminal set.

A population of 4000 GP trees were evolved over 100 generations by Topon and Iba [107]. In this study GP was applied to a binary classification problem in order to distinguish between systemic sclerosis and normal biopsies. Arithmetic trees were evolved using the function set $\{+, -, \times, /, \hat{}, \surd\}$. The attributes formed the terminal set. Each individual represented a rule in the form of *if* (expression ≥ 0) *then* systemic sclerosis, *else* normal. The initial population was created using the ramped half and half method, with a maximum depth of 7. The probability of applying crossover was 0.9, mutation 0.1, and reproduction 0.1. The fitness function took into consideration the correlation between the tree's output and the correct output. Greedy over-selection [3] was used when selecting parents for the crossover operator. When several individuals are considered as a parent using greedy over-selections, the fittest individuals have a greater chance of being selected over the other individuals. The algorithm was tested on a single data set having 27 instances; 81% of the instances were used for training and 19% for testing.

Arcanjo *et al.* [108] evolved 100 GP trees over 50 generations. In this proposed method a sigmoid function maps the output from a tree onto a range of $(0, 1)$. A threshold value of 0.5 was chosen so as to discriminate between the two classes by determining if the result of the sigmoid function is less or greater than the threshold. The threshold value was determined through experimentation. The function set was $\{+, -, \times, /\}$. The attributes and an ephemeral random constant (ranging between -9 to 9) were used as the terminal set. The initial population was created using the full and grow method with a maximum depth of 5. Tournament selection with a size of 3 was used. Crossover had an application rate of 85%, and mutation 5%; elitism was also used. The GP approach was tested on 8 data sets taken from the UCI repository.

Zhang and Wong [109] investigate the use of online simplification. Simplification is used in order to reduce the complexity of the classifier by reducing the number of nodes. This allows the classifier to be interpreted more easily and therefore allows for faster processing. It can be applied after or during the evolutionary process. The simplification process was achieved through simplification rules which were defined prior to the evolutionary process. An example of such a rule is to reduce *"a – 0"* to *"a"*. Each tree is travered recursively with the simplification being applied to each node. Further investigation included the frequency at which the simplification should be applied. The function set used was $\{+, -, \times, /, if\}$, and the attributes along with ephemeral random constants formed the terminal set. Five hundred trees were evolved over 50 generations using GP. The initial population was created using

the ramped half and half method, with a maximum tree depth of 6. Crossover had an application rate of 60%, mutation 30%, and reproduction 10%. The fitness proportionate selection method was used. Accuracy was used as the fitness function. Two data sets from the UCI repository were used; namely, WDBC and spectf. The 10-fold cross-validation was applied to validate the classifiers. Due to the random nature of GP, a total of 50 independent runs were performed. The proposed method was compared to a GP approach without simplification, neural networks, Naïve Bayes, decision trees, nearest neighbour, and the nearest centroid classifier. The results show that the proposed GP method with online simplification obtained a higher classification accuracy than the other methods. Furthermore, the proposed method showed a reduction in the total number of nodes. Finally, the researchers point out that simplification should not be applied at every generation, but with intervals ranging from 2-5 generations.

Several methods for creating threshold values for binary classification were explored by Fitzgerald *et al.* [110]. The traditional threshold approach was compared to eight other proposed threshold approaches. The proposed methods allow GP to decide upon the threshold value instead of setting a fixed threshold prior to the evolutionary process. Amongst the eight methods, the Optimised Individual Class Boundaries (OICB) performed well in terms of achieving a high accuracy. OICB uses a boundary search algorithm which attempts to find the best boundary by partitioning the output values and exploring different threshold values until the most suitable ones are found. Each individual can choose its polarity based on its misclassification error. For the following explanation, assume that there are two classes, positive and negative, for a binary data set.

In binary decomposition, tress output a value greater or smaller than the threshold, and this output is then mapped to a class. In a typical situation, this mapping is determined in advance. Fitzgerald *et al.* define this as the polarity. Typically when a threshold of zero is used, a tree that outputs a negative value will have its output correspond to the negative class, and a tree which outputs a positive value will correspond to the positive class. This is defined in advance and remains unchanged. In terms of the tree output, the polarity is defined in accordance to whether the instances of the positive class are above or below the threshold. A negative polarity is where instances from the negative class are situated below the boundary value. Fitzgerald *et al.* proposed OICB+, where, in this approach, an individual can alter its polarity so as to obtain a smaller misclassification error. The proposed methods used steady state GP with a population size of 500 trees evolved over 60 generations. The function set was given by $\{+, -, \times, /\}$, and the attributes formed the terminal set. Crossover had an application rate of 80%, and mutation 20%. Tournament selection with a size of 5 was used. The initial population was created

using the ramped half and half method. The traditional static threshold approach obtained the highest training accuracy on four out of the six data sets from the UCI repository. However, on the test set, the traditional threshold approach was outperformed by the eight proposed boundary methods. OICB+ obtained statistically significant results that outperformed the traditional threshold approach. OICB+ offers additional flexibility to the overall algorithm in comparison to the threshold approach and stands out as a novel approach for using arithmetic trees for binary classification.

4.3.2 Multiclass classification

Muni *et al.* [111] evolve multitree GP individuals in which each individual contains a tree that represents an expression for each class. Thus, for a data set with three classes, each GP individual will have three trees, one for each of the classes. Each tree within an individual is designed to output a positive value when an instance of data belongs to that particular class, and a negative value when an instance does not belong to that class. Thus, it can be thought of as a positive output is meant to represent a "belong to" signal. A tree representing class i within an individual should output the "belong to" signal when evaluating instances of data from class i. One of three cases can arise when evaluating a GP individual on an instance of data x. The first case is when a single tree (say from class i) outputs a positive value (output $\geq$ 0), and the trees representing the remaining classes output a negative value. The output of the GP individual for the instance of data x is thus class i. Two additional cases may arise namely, where more than one tree outputs a positive value, or no tree outputs a positive value. The researchers modified a heuristic proposed by Kishore *et al.* [112] to allocate a class label to instances which have a conflict. The function set was $\{+, -, \times, /\}$. The attributes and an ephemeral random constant (ranging from 0 to 10) formed the terminal set. The GP approach was tested on five data sets, and the parameters were not consistent on the data sets. The population size varied from 300 to 700, the number of generations varied from 10 to 520, and the maximum tree depth varied from 10 to 13. For four data sets, the results were validated using 10-fold cross-validation. The train/test split was utlised on one data set in order to compare the results obtained by GP to another study which made use of a train/test split. Parents were selected using tournament selection with a size of 7. The probability of applying crossover, mutation and reproduction were 0.75, 0.15 and 0.1 respectively.

One significant advantage of the approach proposed by Muni *et al.* is that a single GP run is required to evolve a solution for a multiclass problem. Another commonly used approach when using GP to evolve solutions for multiclass problems is to use binary decomposition; this is explained as follows. For a multiclass problem

having c classes, a total of c GP runs are performed. Each run is simplified to a binary classification problem. Let a binary problem have two classes a and b, and let some data set have 3 classes {1, 2, 3} for which binary decomposition is applied. The GP runs will be performed as follows:

- In the first GP run, $a = \{1\}$and $b = \{2, 3\}$.
- In the second GP run, $a = \{2\}$and $b = \{1, 3\}$.
- In the first GP run, $a = \{3\}$and $b = \{1, 2\}$.

Formally, let the classes for a multiclass problem be $\{c_1, c_2, \ldots, c_n\}$ where n represents the number of classes. Then for each GP run i, one class corresponds to c_i, and the other class corresponds to the set $\{c_j \mid i \neq j, j = 1, 2, \ldots n\}$.

Several GP representations were investigated by Loveard and Ciesielski [113] for binary and multiclass classification problems. Five representations were proposed, namely, binary decomposition, static range selection, dynamic range selection, class enumeration, and evidence accumulation. In the context of this study, the ranges represent a set of values which are used to map the tree output to a class. The ranges are separated using threshold values.

For static range selection the values of the ranges were intuitively chosen, for example in the *Thyroid* data set, the ranges were chosen to be [-∞, -1), [-1, 1), and [1, ∞]. The authors mention that there is no optimal way of deciding upon the ranges. The function set for this representation was $\{+, -, \times, /, if, \leq, \geq, =, between\}$, and the attributes along with a constant represented the terminal set.

The dynamic range selection allows the class ranges for each individual in the population to be dynamically determined. A range of $[-250, 250]$ was used as a maximum bound. Thus for example, a 3 class problem could have ranges [-250,-50), [-50, 100) and [100, 250]. A subset from the training set is used to determine the ranges for an individual. The same function and terminal set was used as for the static range selection.

The class enumeration representation makes use of the same function and terminal set as in the previous representations; however, class enumeration introduces a new element to the function and terminal set. A new *if* statement is added to the function set. This *if* statement has an arity of 3. The first argument represents a boolean type and uses inequality operators to compare attributes and constants. The second or third argument is executed based on the output of the first argument. The second and third argument returns a class type. This class type is the new element which is added into the terminal set, and is used to represent one of the classes available to the data set. Strongly-typed GP was used in such a way that the first argument of the *if* statement can only be of type boolean, and that the second

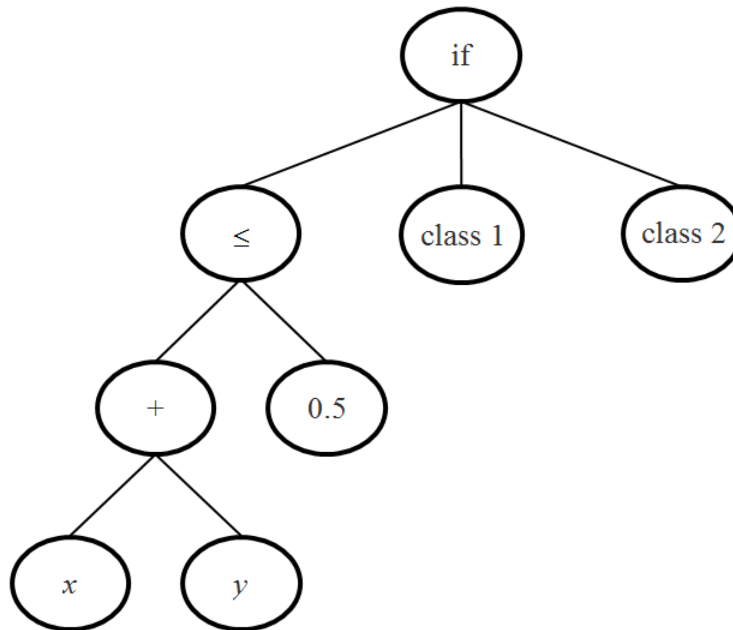


Figure 4.7: An example of a tree created using class enumeration.

and third argument can either be the *if* statement or a class. Figure 4.7 illustrates an example of how the *if* statement is used.

The evidence accumulation representation involves a certainty vector which is created for each individual in the population. The certainty vector stores a numerical value for each class represented by a position in the vector, thus position 1 in the vector corresponds to class 1, position 2 corresponds to class 2, and so on. A tree can add or subtract a value to any position in the vector by using a new terminal, *AddToClass[x](y)*, which adds the value y (y ranges from -1 to 1) to the class in position x. In addition to the *AddToClass* function, the root node of every individual was a *BLOCK* node, this node has an arity between 2 and 4. An individual is evaluated by sequentially traversing each branch from the *BLOCK* node. This representation does not output a value however the class with the highest certainty value is the output class. The population size used varied from 500 to 1000, and the trees were evolved over 50 generations. Elitism and crossover had an application rate of 10% and 90% respectively. The fitness proportionate selection method was used. The maximum initial population depth was 6, and maximum depth during evolution was set to 17. On four of the data sets having less than 1000 instances, the 10-fold cross-validation method was used to evaluate the classifiers, the train/split method was used for the two larger data sets. The five representations were tested on six data sets from the UCI repository, namely, *WBC*, *Bupa*, *Pima*, *Pixel*, *Thyroid*, and *Vehicle*. The binary decomposition and the dynamic range selection approaches

obtained the lowest mean error rate.

Zhang and Ciesielski [114] use GP to detect objects in images. 20 features were extracted from each image in the data which represented the attributes for the data set. Those attributes and a constant formed the terminal set. The function set that was utilised was $\{+, -, \times, /\}$. Each tree outputted a real value which was then mapped onto one of the classes. In order to create a set of boundaries for each class, a threshold T was utilised. The boundaries were computed as follows, $[(i-1) \times T, i \times T]$ for $i = 1, 2, \ldots n$ where n denotes the number of classes. Thus, the boundary for the first class was $[0, T]$, and the boundary for the second class was $[T, 2T]$. The value of T was set to 100, and there are no details as to how this value was chosen. GP was tested on three data sets, of which one was generated, and the other two were obtained through photographs. The GP parameters were not consistent for the three data sets. The population size varied from 100 to 500, the maximum tree depth varied from 8 to 20, and the maximum number of generations varied from 100 to 250. The GOs were crossover, mutation, and elitism, and they had different application rates for each data set. The rationale behind these different parameters was not provided. The proposed GP algorithm obtained better results than a neural network which was applied to the same data sets.

Smart and Zhang [115] evolved arithmetic trees to compare static and dynamic range methods for image classification. Four statistical measures were obtained from the images and represented the terminal set. The function set that was utilised was $\{+, -, \times, /, if\}$. Two dynamic range selection methods were created in order to map the output of a tree onto some class. The first was the centred dynamic range selection (CDRS) method which calculates the centre of each class. The centre of the class is computed by evaluating each tree in the population on the training data. The boundaries which separate each class are obtained by calculating the midpoint between the centre of each class. Once the boundaries are calculated, each individual in the population is evaluated again based on the new boundaries. The second method was the slotted dynamic range selection, this is similar to the dynamic range selection represented in [113] except the slots varied from [-25, 25] with slots at each 0.5 step, thus resulting in 100 slots. The population size varied from 300 to 500, and the maximum tree size varied from 5 to 6. No rationale behind the variation in parameters was provided. The maximum number of generations was set to 50. Crossover, mutation, and reproduction had an application rate of 50%, 30% and 20% respectively. The proposed methods were tested on five image data sets having 3, 4 and 5 classes. These data sets were generated by the authors. The results show that the dynamic methods obtained a higher classification accuracy, furthermore the CDRS method obtained the best results. A similar comparison between static and dynamic range selection was studied by Song *et al.* [116]. The results of this

study once again shows that dynamic methods obtain a higher classification accuracy than static range methods. Song *et al.* applied the proposed GP method to texture classification.

Multiclass image classification is achieved in a single GP run by the proposed method of Smart and Zhang [117]. Similar to binary decomposition, several binary sub-problems are constructed based on the total number of classes for a particular data set. Each combination of binary sub-problems was considered in a single GP run, and a fitness function based on the distribution distance of each pair was used. The distribution distance is calculated through the mean and standard deviation of the outputs of each program for a particular class. The distribution distance measures the separability between two classes. A large value represents a good separability between two classes. The individuals in the population are examined to determine which class it separates the best. For each generation, all of the binary pair sub-problems are considered and the individual which can best separate the pair is compared to the current "expert". If an individual is able to separate the pair better than the expert, it will replace the current expert. Furthermore, once an expert obtains a distribution distance value lower than a user specified threshold, that particular binary problem is then considered to be solved. Consequently, individuals in the population no longer have to solve that specific binary problem. Thus, a program is only required to be able to distinguish between a pair of classes in order to obtain a high fitness. A probability function is used to combine all the experts together before applying it a the test set. GP evolved 500 trees over 40 generations. The initial population was created using the ramped half and half method. The maximum tree depth was set to 7. The function set was $\{+, -, \times, /, if\}$, and the attributes, along with a constant formed the terminal set. Crossover, mutation, and reproduction had application rates of 60%, 30% and 10% respectively.

Jabeen and Baig [118] proposed a two stage learning approach using GP to evolve a population of 600 trees. A population of trees is evolved in a binary decomposition manner for each class. For each class in a data set, a GP run is executed, and the final population for each class is stored. Given an instance of data x, and a tree belonging to class i, if the tree outputs a positive value, then instance x belongs to class i. A negative output implies that a particular instance does not belong to the class which the tree represents. When more than one tree belonging to different classes, output a positive value, then there is a conflict, i.e. an instance of data belongs to more than one class. In the first stage of the proposed method, the initial population is created using ramped half and half. The function set was $\{+, -, \times, /\}$, and the terminal set was formed using the attributes and an ephemeral random constant. The fitness function made use of the true positives and true negatives. When two individuals had the same accuracy, the smaller tree was selected. Crossover, mutation, and

reproduction were used as GOs and had application rates of 50%, 25%, and 25% respectively.

The second stage of the proposed method evolves a population of chromosomes. The chromosomes were structured as follows. For a n class classification problem, the size of the chromosome corresponds to n. Referring to the final populations in stage one, position 1 in the chromosome corresponds to a tree from class 1, position 2 in the chromosome corresponds to a tree from class 2, and so on. The chromosomes were initialised as follows. One tree from each of the populations in stage 1 is chosen using tournament selection. Thus, for position 1 in the chromosome, tournament selection is applied to the population representing class 1, and the chosen tree is added to position 1. This is repeated for each class. The population of chromosomes is evolved over 50 generations. In stage 2, the fitness function considered both the accuracy and the number of conflicts, thus creating a preference for chromosomes which have a high accuracy and a small number of conflicts. The proposed method was tested on 5 publicly available data sets (*Iris*, *Wine*, *Vehicle Silhouettes*, *Glass*, and *Yeast*) and achieved better results when compared to a binary decomposition method. The rationale behind the two stages was to reduce the amount of conflicts which occurs from the standard binary decomposition approach.

Silva and Tseng [119] used 500 GP trees in order to classify seafloor habitats. In this study 7 attributes were used on a 5 class problem (*algae*, *australis*, *sand*, *sinuosa*, and *reef*). The initial population was created using the ramped half and half method with a maximum depth of 5. There was no limit on the growth of the trees during the GP evolution. The function set was $\{+, -, \times, /\}$. The attributes formed the terminal set. Crossover, mutation, and reproduction had probabilities 0.5, 0.5, and 0.1 respectively. Two methods were investigated. The first approach was a pairwise separation between the classes. In the first run, GP was used to separate *sand* and *algae*, *sand* and *austrailis*, *sand* and *sinuosa*, and finally, *sand* and *reef*. Then in the second run, reef was separated from the remaining classes as done in the first run. This was repeated for each class. The pairwise separations could be performed in any order.

The second approach performed a similar separation to binary decomposition. For each class i, the aim was to find a tree which would separate class i from the remaining classes, and then to ignore class i in the next run. In the first instance, a tree had to be evolved to separate *sand* from the remaining classes. In the second run, a tree was evolved to separate *reef* from the other classes, except *sand*. This process was repeated for each class. Since this approach excludes the class previously used, the order in which the classes are chosen will impact the final classification model. In binary decomposition, the order in which the multiclass problem is converted into a set of binary problems has no impact on the final model, because in

each case one class is compared to all the other classes, and this is repeated for every class. There was no discussion as to the order in which the separations were performed; however, the authors mention that *sand* was chosen first since it was easily separated from the other classes in the first approach.

One drawback of binary decomposition is that for a particular instance of data, several trees can output a positive value for that instance, and as a consequence, the instance will be classified as a number of different classes. Chien *et al.* [120] investigate an approach in order to minimise the number of conflicts. In this study, GP is used to evolve arithmetic trees using the function set $\{+, -, \times, /\}$. In this case, a n class problem is run n times using the proposed methods to create n functions, one for each class. The functions are then combined together in a decision tree to create the final classification model. The researchers point out that the functions cannot be added to the decision tree in any order. By adding functions in any order, a situation can arise where a high level function (one closest to the root) has a high misclassification rate, and consequently, that function will output a positive value for many instances which do not belong to that particular class which is represented by that function. This situation can be avoided if another function with a low misclassification rate is placed at the root. The precision and recall measures are used in order to determine the order of the functions. The functions were first sorted according to their precision, and should a tie occur, according to their recall. A function with a high precision rate has a lower misclassification rate. For a function f and class i, the precision and recall are defined as follows:

$$precision(f) = \frac{number\ of\ instances\ which\ are\ correctly\ classified\ by\ f\ and\ that\ belong\ to\ class\ i}{number\ of\ instances\ classified\ by\ f}$$

$$recall(f) = \frac{number\ of\ instances\ which\ are\ correctly\ classified\ by\ f\ and\ that\ belong\ to\ class\ i}{number\ of\ instances\ in\ class\ i}$$

The proposed GP approach evolved 1000 trees over 10000 generations. Five publicly available data sets taken from the UCI repository were used for experimentation, namely, *Iris*, *WBC*, *BUPA*, *Vehicle Silhouettes*, and *Pima Indians*.

4.3.3 Advantages and disadvantages of GP arithmetic trees

When using GP to evolve arithmetic trees the function set is made up of mathematical functions, and thus arithmetic trees are able to naturally create classifiers for data sets having numerical attributes. One significant difference with decision and arithmetic trees, is that with the latter discretisation is not needed when dealing with numerical attributes. However, if GP is used to evolve arithmetic trees for data sets containing categorical attributes, then a problem is encountered as mathematical functions cannot be applied to categorical data. Two ways of addressing this issue were proposed by Lovelard and Ciesielski [121]. Both of these proposed

methods added new terminals which would allow GP to use mathematical functions on categorical attributes. Based on the proposed approach, the function set remains unchanged, and terminals to cater for categorical data are added to the terminal set using one of two methods. The first method allocates an integer to each value that a categorical attribute can take. For example, the categorical attribute *gender* could be converted to a terminal with the following property, *male=0*, and *female=1*. Once this conversion takes place, the *gender* terminal can now be used with mathematical functions. Thus, the expression *gender + 1* would represent a valid GP tree and is illustrated in figure 4.8. For a given instance of data, when the *gender* terminal is encountered, the corresponding value is returned based on the conversion. Thus, if for a particular instance the value of *gender* is "female", then the output of the tree will be 2.

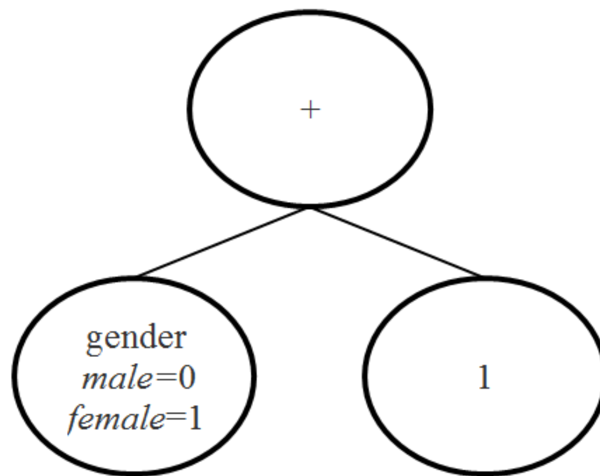


Figure 4.8: Representing a categorical attribute as a numerical one.

The second approach converted the categorical attribute into binary terminals. For example, the following two terminals could be created, *male = 1*, *not male = 0*; and, *female = 1*, *not female = 0*. Thus for each categorical attribute, a binary terminal is created for each possible value.

A significant difference in the evaluation of an arithmetic tree and a decision tree is that when a decision tree is evaluated only a single path from the root to a leaf node is traversed. When evaluating an arithmetic tree, every single node within the tree has to be traversed in order to evaluate the entire expression. Thus, evaluating arithmetic trees which have a large depth will result in slower execution times. The second major drawback of arithmetic trees is that additional complexity is encountered when dealing with multiclass classification problems. It is simpler to evolve decision trees for multiclass classification problems, the only difference be-

tween a binary classification problem to a multiclass one, is that additional terminals are added to the terminal set when using decision trees for multiclass classification problems.

4.3.4 Summary of the findings

Based on the studies reviewed, when evolving arithmetic trees the attributes are used as the terminal set. There are differences in the function set amongst the various studies, however typically the set $\{+, -, \times, /\}$ is commonly used. The *if* statement is used in 33.33% of the studies; however, there has been no study comparing the performance of the evolved classifiers when this function is present and absent. This summary describes the findings for binary and multiclass classification.

In the case of binary classification, evolving GP trees is simpler as a threshold value can discriminate between the two classes. This simple approach is easy to implement and has been successfully used in the various studies. Typically a threshold of 0 has been used. Based on the studies reviewed, the population size varied from 100 to 5000 trees, and no conclusion as to what the optimal population size should be set to can be made. The number of generations varied from 41 to 2000. The ramped half and half method was used in six studies, and there was no mention of the initial population generation method in the remainder of the studies. Crossover, mutation, and reproduction were used, with crossover having the highest application rate, followed by mutation and then reproduction. Both the fitness proportionate and tournament selection methods were equally used. Tournament selection was used with sizes varying from 5 to 7. The number of data sets used in the studies varied from 1 to 8. In certain cases the data sets were generated by the authors, and in other cases the data sets are not made publicly available. In such cases it is difficult to compare the performance of new methods to those studies. The train/test split performance measure was most commonly used throughout the studies reviewed, and the 10-fold cross-validation method was used in one study. In certain cases no detail was provided as to which approach was used to measure the performance of the proposed algorithms. The 10-fold cross-validation method is generally the most commonly used method across studies using GP for data classification, however this method is also more time consuming than the train/test split approach. It is unclear as to why most of the studies used the train/test split.

In the case of multiclass classification, evolving GP trees using an arithmetic tree representation involves more complexity than for binary data sets. Either a binary decomposition approach is used, or a method allowing a single tree to perform multiclass classification is proposed. In the case of binary decomposition, n GP runs are required for a n class problem. In the case where n is large, this can result in large computation times, especially if in addition the data set has a large number

of instances. Furthermore, clashes arise when using binary decomposition whereby numerous trees can output a positive value indicating that a particular instance belongs to numerous classes. This has been addressed using statistical measures [120] and evolutionary algorithms [118]. Regardless of the extra number of runs required, and the additional complexity involved in conflict resolution, this approach has been used several times and remains a simple strategy in order to perform multiclass classification. In the case where binary decomposition is not used, static or dynamic range selection has been employed. The static range selection is the simplest approach; however there is no way of determining the optimal thresholds for each class. Dynamic range selection methods have led to better performance, however the implementation of this approach involves additional complexity. There was a variation in GP parameters across the studies dealing with multiclass classification. The population size varied from 100 to 1000, and the number of generations varied from 10 to 1000. There is little to no detail regarding the parent selection methods used. Similar to studies addressing binary classification, crossover, mutation, and reproduction were the most used GOs with crossover having the highest application rate, followed by mutation, and then reproduction. The 10-fold cross-validation and train/test split were both used as performance measures, however, in certain studies there was no mention as to which method was used. There was no consistency in the number of data sets used for testing, these varied between 3 and 5. In certain cases the data sets were generated, and in others they were obtained from the UCI repository.

4.4 GP and Logical Trees

This section reviews studies which have used GP to evolve logical trees. Logical trees are composed of boolean operators, and they output a value of true or false.

Kuo *et al.* [122] evolved logical trees using GP. The function set that was used was $\{AND, OR, NOT, >, \geq, <, \leq, If-Then, If-Then-Then-Else\}$. The terminal set was made up of the values that each attribute could take, for example, an attribute having values $\{A, B, C\}$ would result in those three values being added to the terminal set. Furthermore, the classes were added to the terminal set, and could only appear in the second or third branch of the *if* statements. Two genetic operators were developed to resolve the issues of redundancy and subsumption. The eliminator operator was proposed in order to identify duplicate rules within any given tree (redundancy), if duplicates were found, then the deepest duplicated rule within the tree was removed. The merge operator attempted to remove rules that subsume another by examining if two rules had the same classes, and if the attributes of one was a subset of the other (subsumption). The fitness function computed the

accuracy and complexity of each tree. For a particular tree t, the complexity of t was defined as the ratio between the number of nodes within t, and the average number of nodes within all the trees in the the initial population. The proposed GP algorithm was evolved over 500 generations and applied to a binary classification problem. The data set was split with 70% of the data being allocated to the training set, and 30% to the test set. Crossover, mutation, elimination, and merge had probabilities of 0.9, 0.02, 0.8, and 0.8 respectively. The proposed GP approach outperformed C5.0 and a standard GP algorithm. Standard GP did not make use of the eliminator and merge operators, and thus indicating the usefulness of those two operators.

De Falco *et al.* [9] evolved 2000 trees over 30 generations. A binary decomposition approach was used. The function set utilised was $\{AND, OR, <, \leq, >, \geq, IN, OUT\}$. The *IN* and OUT functions take 3 arguments each and are used to represent internal and external intervals respectively; figure 4.9 illustrates the *IN* function. The *IN* function outputs true if the value of the attribute is within the range of the lower and upper bound, and outputs false otherwise. The attributes formed the terminal set. Since a binary decomposition approach was used, a method was proposed in order to resolve conflicts between the classifiers. Each tree corresponds to a logical rule which is represented by a boundary in Euclidean space. To illustrate this, consider a two dimensional space, thus two trees represent two rectangular boundaries within the space. The surroundings of each boundary are referred to as a frontier. Thus in order to resolve conflicts, the distance between the points in Euclidean space is computed, however there are no further details regarding this proposed method. The fitness function took the accuracy, tree depth, and number of nodes into consideration. The initial population was created using the ramped half and half method and a maximum initial depth of 5. The maximum tree depth allowed during evolution was set to 7. Crossover, mutation, and reproduction had application rates of 80%, 10% and 10% respectively. Tournament selection with a size of 50 was used. The proposed GP algorithm was evaluated on six data sets from the UCI repository, namely, *WBC*, *Diabetes*, *Heart Disease*, *Horse Colic*, *Glass*, and *Australian Credit Approval*. For each data set, 75% of the data was allocated to the training set, and the remaining 25% allocated to the test set. Thirty GP runs were performed for each data set using different seeds.

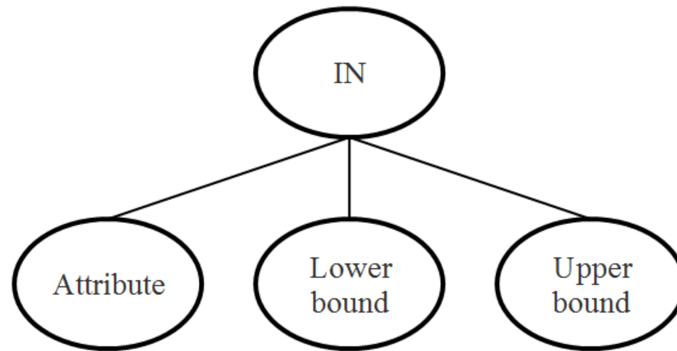


Figure 4.9: The *IN* function proposed by De Falco *et al.* [9].

In the two previous studies, the terminal set was made up of attributes. In the study of Eggermont *et al.* [123], the terminal set consisted of atoms. Each atom had an attribute, one inequality $\{>, <\}$, and a constant between 0 and 1. For example, an atom cam be represented as $X > 0.4$, where X is an attribute in some data set. The authors proposed subatomic mutation, this operator selects a node within a tree, and if that node is an atom, then either the variable or the constant is modified. For this study the function set was $\{AND, OR, NAND, NOR\}$. A population size of 2000 was initialised using ramped half and half with a maximum tree depth of 5. Crossover and subatomic mutation had probabilities of 0.9 and 0.1 respectively. The proposed GP approach was tested on 4 binary data sets, namely, *Australian Credit, German Credit, Heart Disease,* and *Pima*. The k-fold cross-validation method was used with values of k varying from 9 to 12. No rationale behind the variation in the values of k was provided.

A binary decomposition approach is also investigated by Tan *et al.* [124] whereby n GP runs are performed for an n class problem. The logical trees form rules in the following format: *if antecedents then class*. In this study, the antecedents are made up of functions and terminals. The function set was $\{AND, NOT\}$, and the terminal set represented all the values for each attribute. Unlike the other studies, Tan *et al.* enforced a constraint which ensured that for a given tree, once a value has been selected for a particular attribute, another value for that same attribute cannot be selected. For example, assume some attribute i has values $\{1, 2, 3\}$, and the value of *"1"* is used within a particular tree, the constraint ensures that another value for attribute i cannot be selected. This was enforced to reduce redundancy. The population size varied from 10 to 100, and the number of generations from 10 to 50. The initial population was generated using the ramped half and half method. Crossover, mutation, reproduction, and elitism were applied with probabilities 0.9, 0.1, 0.1, and 0.05 respectively. The GP approach was tested on 9 binary and mul-

ticlass data sets from the UCI repository, namely, *Weather*, *Contact-Lenses*, *Zoo*, *Breast Cancer*, *Monk1,2,3*, *Mushroom*, and *Nursery*. 66% of the instances was allocated to training, and 33% to testing. Thirty GP runs were performed on each data set.

GP was applied to a medical data set consisting of 165 attributes and 12 classes for chest pain diagnosis in the study of Bojarczuk *et al.* [125]. Binary decomposition was also used, thus GP was run 12 times in order to create a classifier for each class. The rules were encoded in the form, *if antecedent then class*, and the antecedent is represented by a GP tree. The function set was composed of $\{AND, OR, NOT\}$, and the attributes represent the terminal set. Five hundred GP individuals were evolved over 50 generations. The initial population was generated using the ramped half and half method with an initial maximum depth of 10. The maximum tree depth during evolution was set to 17. The fitness function made use of the sensitivity and specificity (refer to chapter 3, section 3.4.2), and also the number of nodes within a tree. Parents were selected using the fitness proportionate selection method. Crossover and reproduction were applied with probabilities of 0.9 and 0.1 respectively. The data set was split into a training set containing 65% of the data, and 35% was allocated to the test set. Ten GP runs were performed for each class; the proposed GP algorithm outperformed the C5.0 algorithm.

4.4.1 Advantages and disadvantages of GP logical trees

Evolving logical trees using GP has the benefit of being able to handle both numerical and categorical data fairly simply in comparison to decision trees or arithmetic trees. With decision trees, discretisation is required in order to use numerical attributes, and in order to use categorical attributes when evolving arithmetic trees, one has to convert those categorical attributes into numerical ones. When using the right function set it is possible to cater for either categorical or numerical attributes. A typical function set to handle numerical attributes is $\{AND, OR, NOT, <, >\}$, whereas for categorical data a suitable function set is $\{AND, OR, NOT, =\}$. Through the use of the equal operator, it is possible to make use of categorical attributes.

An additional significant advantage of evolving logical trees is the fact that the trees are highly interpretable to a researcher. This is further accentuated when simple logical functions are used, such as $\{AND, OR, NOT\}$. Including functions such as $\{NAND, NOR, XOR\}$ will increase the complexity in terms of interpretability since those functions are not as trivially understood compared to the simpler ones. Interpreting logical trees is easier than interpreting arithmetic trees.

Logical trees and arithmetic trees share a common disadvantage. Every single node within the tree for both representations has to be traversed in order to output the classification result. This can affect the GP run time especially if large trees

are used, and additionally, if a large data set is used. Similarly to arithmetic trees, additional complexity is encountered when evolving logical trees for multiclass classification problems. Typically, binary decomposition can be used to solve this issue, however this means running the GP algorithm numerous times depending on the number of classes.

4.4.2 Summary of the findings

GP has been used to evolve logical trees in several studies. The logical functions $\{AND, OR, NOT\}$ have been used as the function set, and the terminal set was composed of the attributes. Similarly to evolving arithmetic trees, it is simpler to evolve logical trees for binary classification problems due to the fact that no conflict resolution method has to be put in place. When dealing with multiclass classification problems, the binary decomposition approach has been used [124, 125] successfully. It is however possible to make use of the conflict resolution approaches which were proposed for evolving arithmetic trees as discussed in the previous section. The GP parameters varied from one study to another. The population size varied from 500 to 2000, and the number of generations from 30 to 500. In four studies, the ramped half and half method was used to create the initial population with a maximum depth varying from 5 to 10; this method has been used in most of the reviewed studies. This study shall thus make use of it to generate the initial population. The maximum allowed depth during the evolutionary process varied from 7 to 17. Since the studies reviewed in this section made use of generational GP, this study will also make use of it. Similar to studies using GP and the two previously discussed representations, the accuracy and complexity of the trees was used as the fitness function. This study will take into account the accuracy and complexity of the trees when evolving logical trees. Both the tournament and fitness proportionate selection methods have been used for evolving logical trees. However, the tournament selection method has generally been used more often when solving data classification problems and thus will be used in this study. Crossover and mutation were the most commonly used GOs with crossover having the highest application rate; this study will also make use of a similar setting.

4.5 GP and Other Representations

The three most common GP representations for data classification were reviewed in sections 4.2, 4.3, and 4.4, however there are other studies which investigated alternative representations. Eggermont *et al.* [126] proposed an atomic representation which is slightly different to the one described in their previous study [123] which was reviewed in section 4.4. Each atom is defined in terms of an attribute, an opera-

tor, and a value. The atoms are combined into a tree structure. Three variations of the atomic representation were investigated. The differences in these representations were in the way the numerical attributes were dealt with.

In the first GP representation, each possible value for an attribute was assigned to an atom using the $<$ operator. For an attribute having 100 possible values $\{0, 1, \ldots, 99\}$ there would be 100 atoms, i.e. $attribute < 0$, $attribute < 1$ and so on. This representation suffers from the fact that the search space is extremely large.

The second GP representation used the *gain* and *gain_ratio* measures in order to obtained thresholds. The thresholds were obtained in order to reduce the number of atoms used. In this representation atoms formed partitions in the form of $attribute < threshold_1$, $attribute \in [threshold_1, threshold_2)$, and $attribute \geq threshold_2$. The first atom to partition an attribute used the $<$ operator, the last atom used the $\geq$ operator, and the atoms in between used two threshold values. A maximum of 5 partitions were allowed per attribute, and thus, this reprentation significantly reduces the search space since a given atom within a tree can have a maximum of 5 branches.

The third GP representation made use of clustering, more specifically the K-means algorithm. Clustering was performed on the values of a single attribute at a time. The number of atoms corresponded to the number of clusters. Each atom is in the form $attribute \in [min_i, max_i]$, where min_i and max_i are the upper and lower bounds for cluster i.

For each representation, a population of 100 trees were evolved over 99 generations. The ramped half and half method was used to create the initial population. The fitness function used the misclassification rate, and the number of nodes as a tie breaker. The selection method was tournament selection with a size of 5. Ten GP runs were performed on each data set for the proposed representations. The representations were tested on 5 data sets from the UCI repository, namely, *Australian Credit*, *German Credit*, *Pima Indians*, *Heart*, *Ionosphere*, and *Iris*. The performance of the proposed representations were compared to C4.5, a simple GP algorithm, and two evolutionary algorithms. The proposed representations outperformed the standard GP representation in certain cases.

Jabeen and Baig [127] presented a novel approach for dealing with mixed attributes (data sets having both categorical and numerical attributes) for binary classification. The proposed method consisted of a two layer approach, an outer layer and an inner layer. For the outer layer, the function set was $\{AND, OR, NOT\}$, and the terminals of the outer layer corresponded to the inner layer trees. Two types of inner layers were used, a logical and an arithmetic one. The function sets for the logical and arithmetic layers were $\{AND, OR, NOT\}$ and $\{+, -, \times, /\}$ respectively. For the logical layer, the terminals were composed of the categorical attributes along

with an "=" or "≠" sign, for example *gender='female'*. The arithmetic inner layer used the numerical attributes as terminals. The initial population of 600 trees was created using the ramped half and half method, and evolved over 250 generations. Crossover, mutation and reproduction had application rates of 50%, 25%, and 25% respectively. The proposed method was tested on 5 data sets from the UCI repository, namely, *Japanese Credit*, *Australian Credit*, *German Credit*, *Heart*, and *Hepititus*. Five executions of the 10-fold cross-validation method were performed on each data set.

The two studies described in this section propose representations which are not similar to the common ones. Furthermore, these two methods are able to handle data sets having both categorical and numerical attributes.

4.6 GP and Ensemble Classifiers

This section reviews studies which have used GP to evolve ensembles. Ensembles were described in chapter 3, section 3.7.3. Adaboost has been applied to data classification using GP by Idris *et al.* [128]. In this study, GP and Adaboosting are combined for a binary classification problem, churn prediction. In this implementation the GP individuals are evolved in such a way as to focus on the instances which were misclassified by the previous base classifier. For each class a number of GP classifiers were evolved. Initially the weights were set to a fixed value. The first base classifier was evolved and the weights were updated. The second base classifier had the task of focusing on those instances which the previous base classifier was not able to correctly classify. For each class a weighted score was calculated and the class with the highest score was the resulting class.

Thomason and Soule [129] proposed two ensemble methods, Orthogonal Evolution of Teams (*OET1*) and *OET2*. Steady-state GP was used to evolve ensembles made up of arithmetic trees. For both of proposed methods, each ensemble had a fixed size N, and each individual in the ensemble assumed a position i where $i = 1, 2, \ldots, N$. In *OET1* parent selection is performed on the individuals, and the replacement is done on the teams. An empty ensemble e_1 is created, and individuals are added into e_1 as follows. Individual i is added to e_1 by performing tournament selection on all of the individuals in position i for all of the ensembles. A second empty ensemble e_2 is created in a similar way as e_1. Crossover and mutation are applied to e_1 and e_2, and the fittest is chosen for the next generation by replacing the weakest offspring in the population of ensembles.

In *OET2,* selection was performed on the teams and replacement on the individuals. Tournament selection was applied to the population of ensembles and two parents were selected. Two offspring were created using crossover and mutation. In

a similar way to *OET1*, for each position i within the ensembles, the weaker individuals at position i in the ensemble population were replaced by those at position i in the offspring.

An ensemble was evaluated as follows. Each individual in an ensemble outputs a real value number which is mapped to a class. If a tree outputs a value outside the range of the available classes, then it is classified as "*I don't know*" (IDK). The fitness of each individual takes into consideration the number of correctly and incorrectly classified instances as well as the number of IDK instances. Each individual in an ensemble casts a vote which is weighted between 0 and 1 according to its fitness. The individual with the highest weight is used as the classification result for the ensemble.

Zhang and Bhattacharyya [130] proposed an ensemble approach where subsets from large data sets were created in order to reduce the amount of memory required to evolve classifiers, and also to speed up the evolutionary process. Ten subsets were randomly created (with replacement) from a data set, and three types of classifiers were created for each subset, namely, a GP classifier, a decision tree, and a logistic regression model. For each type, the corresponding classifiers were combined into an ensemble, thus for each type, an ensemble of 10 individuals was created. The classification output of each ensemble was computed by obtaining the output of each individual in the ensemble and applying a majority vote. The effect of the number of subsets was investigated, and the findings revealed that the classification accuracy improved as more subsets were used; however, this change was less noticeable beyond 10 subsets.

A similar approach to reduce the amount of memory required to evolve classifiers for large data sets was investigated by Folino *et al.* [131]. The proposed method, *BagCGPC*, makes uses of bags similar to bagging (defined in chapter 3). In this study a data set is partitioned into smaller subsets and GP subpopulations are trained on the subsets. A classifier is evolved for each subset and majority voting based on the output of each classifier is used to classify the test cases. Each subpopulation is evolved on a processor and the experiments are run in parallel. The proposed method was applied to 5 data sets from the UCI machine learning database and obtained a small error rate on the *Cens* data set which contains a relatively large number of instances. The results were not as significant on the other four data sets which contained fewer instances. Folino *et al.* [132] further investigated the effect of the size of the ensemble and the results reveal that smaller ensembles perform better than larger ones.

Iba [133] proposed two GP methods, *BagGP* and *BoostGP* based on bagging and boosting respectively. For each method, ten subpopulations were created and used to solve several machine learning problems. The instances were classified by

determining the class having the highest number of votes from the individuals in the ensembles. The findings reveal that when compared to a standard GP approach, *BagGP* performed better in terms of mean square errors. The results also reveal that the proposed methods evolved smaller trees in terms of the number of nodes than in the standard GP approach. The major difference between this work and Folino *et al. [131]* is that Iba did not implement parallel programming.

Augusto *et al.* [134] implemented a coevolutionary multi-population approach based on bagging and boosting. The populations were evolved in parallel and implemented a semi-island model approach whereby the populations apply genetic operators amongst themselves independently. Furthermore, based on a defined migration rate, genetic material between the individuals from different populations take place. The proposed method, Coevolutionary Multi-population Genetic Programming (*CMGP*) is similar to boosting in the sense that hard instances of data apply pressure to the classifiers. This was achieved through coevolution competition where a subpopulation of classifiers competed with a subpopulation of data instances. An individual from a subpopulation of classifiers was selected, and an instance of data from a subpopulation of instances was selected. If the classifier was able to correctly classify the instance, then the individual's fitness was increased and the weight of the instance was decreased. However if the instance was incorrectly classified, then the weight associated with that instance was increased and the fitness of the individual was decreased. In this manner, the harder instances were selected more often in a similar way to boosting. Additionally, each individual was assigned a confidence measure to allow a fitter individual to have a more significant vote when the ensemble was evaluated. *CMGP* was experimented with several ensemble sizes varying from 1 to 31. Each ensemble corresponded to a population and evolved a classifier. The proposed methods were tested on 7 data sets from the UCI repository, and the results show that for certain ensemble sizes, *CMGP* obtained lower error rates when compared to bagging and boosting. The findings also reveal that on 6 data sets, a lower error was obtained for boosting when a larger ensemble was employed, a similar observation was made for bagging on 4 data sets.

Paul *et al.* [135] used GP to create ensembles made up of arithmetic trees. In the study binary and multiclass problems were dealt with separately. For binary classification problems, a single threshold value of 0 was used to differentiate between the output of a tree, and x rules were created by running the GP x times. A majority vote was then applied to determine the class for test cases. For multiclass classification problems, x rules were created for each class by running GP x number of times. Thus if a classification problem had three classes and a value of $x = 2$, then a total of 6 rules were created, i.e. two for each class. The test cases were evaluated as follows. Each rule belonging to class C_i casted a vote, if a positive

output was obtained then the total votes for C_i was incremented by one, conversely, a negative output incremented the total votes against C_i. The ratio of positive to negative votes for each class was then calculated, and the one corresponding to the highest ratio was selected as the predicted class. An investigation on the optimal value of x was performed. The proposed method was tested on one binary and one multiclass gene expression data set. The results show that the highest accuracy was obtained when x was set to the number of instances in the training set. The proposed GP method outperformed a kNN classifier. Paul and Iba [107] also observed that the best accuracy was obtained when the number of GP individuals in an ensemble was equal to the number of training instances. In this study there were 22 training instances. GP was used to evolve ensembles made up of arithmetic trees, and additional experiments were performed for which ensembles were made up of logical trees. Both representations obtained 100% classification accuracy when the training data was equal to the entire data set.

Pappa and Freitas [136] use a two stage approach for creating ensembles. The first stage uses GP to evolve a population of rule induction algorithms. Then the last population is used in the second stage to create rules that are combined into an ensemble. Two methods for combining the votes of individuals in the ensembles were investigated. The first, *Rules-Ens Maj*, was based on a majority vote. The second, *Rules-Ens Fit*, was based on a fitness weighed voting approach; the weights were determined by the fitness of each individual in an ensemble. Furthermore, two approaches for creating the ensembles were examined. In the first approach, the entire last population from the first stage was used as candidates to create an ensemble. In the second approach, ensembles could only be created using a subset of individuals from the last population. Two methods for creating the subset were investigated, and individuals were selected based on their fitness. The first method selected the best 10 individuals, and the second selected the best 5 and worst 5. The latter was proposed as it naturally promotes greater diversity by combining the best 5 and worst 5 individuals in the population. The methods were tested on 5 data sets from the UCI repository, and the ensembles were compared to the best single rule induction algorithm in the population. Creating the ensembles using the best 5 and worst 5 individuals from the last population yielded the best results. This method obtained statistically significant better results than the single best GP individual on 2 data sets.

Lichodzijewski and Heywood [137] used linear GP [1] to evolve individuals which represented bidding behaviours. The bid is used to determine which individual in an ensemble can allocate a class. The individuals in the ensembles are called learners, and an ensemble needs at least two learners. Each learner on its own does not represent a complete solution unlike the other ensemble methods previously

discussed. In this work, three populations are evolved, a population of learners, ensembles and points. The population of points represent subsets of the training data. The outcome of the evolutionary algorithm is an ensemble made up of of several learners. Each learner which represents a bidding behaviour by placing a bid on the test data, and the individual which obtains the highest bid allocates the class to the test data. The proposed approach was tested on 3 publicly available data sets and was outperformed by two other methods, a learning classifier system [138] and a support vector machine [139].

Bhowan *et al.* [101] investigated three methods to reduce negative effects of biased individuals within an ensemble for binary classification problems. The first approach used a weighted majority vote which was based on an individual's fitness. Thus an individual with a weak fitness had a small contribution to the final decision; the converse applies to individuals with a strong fitness. The second approach involved removing those individuals from an ensemble that had an accuracy of less than 50% on either the majority or minority class. The third approach was based on Off-EEL [140]. Off-EEL consists of two phases, in the first phase the classifier is evolved according to some algorithm, and the last population is then used by sorting the individuals according to their fitness from best to worst. The selection then proceeds by iterating through the sorted list and taking the best individual in the list and adding it to the ensemble. This proceeds until the sorted list is empty. Bhowan *et al.* then evaluate all of the ensembles using a majority vote, and the best ensemble is output. Only odd sized ensembles are used in order to avoid a tie between the classes. The proposed methods were tested on 6 publicly available unbalanced data sets. The weighted vote and Off-EEL methods obtained better results than the single objective GP approach, as well as obtained better results when compared to Naïve Bayes and SVM classifiers.

Liu and Xu [141] evolve individuals which consist of sub-ensembles (SE). The proposed method is applied to multiclass classification problems and each SE evolves trees for a class. Thus for a n class problem, there are n SEs. Since a tree is evolved for each class, this representation allows an individual to deal with a multiclass problem in a binary decomposition manner. The researchers make use of a diversity measure which is based on the feature subset and the size of an individual. Additionally a greedy algorithm was implemented to promote the diversity amongst the ensembles. The greedy algorithm places all the SEs in a list, and in the first step removes the SEs which do not exhibit large diversity. In the second step, the algorithm attempts to increase the diversity of the SEs by exchanging those trees which exhibit little diversity on their own. This procedure is repeated several times until no further improvement can be achieved. The class output by each individual in the SE was combined using a weighted majority vote based on accuracy.

4.6.1 Strengths and weaknesses of GP ensembles

The most significant advantage of using ensembles to evolve GP trees is the improvement in classification accuracy over a standard GP approach. This advantage has been reported in [101,133], and furthermore, GP based ensembles outperformed other classification methods in [101, 128, 134, 135, 137, 140, 141]. GP based ensemble approaches have successfully been used to achieve high classification accuracy and is a recommended approach for solving data classification problems. The second advantage is that implementing an ensemble for GP can be achieved without having to modify the entire system; this was demonstrated by Gagné *et al.* [140] whereby ensembles were created using the final population. Evolving ensembles leads to improvement in the evolved classifiers, however there is a challenge one has to face when implementing ensembles. Espejo *et al.* point out that when evolving ensembles, one concern is the way in which the individual members of the ensemble are combined together in order to produce a single classification output. From the literature reviewed, majority voting has typically been employed to address this issue.

4.6.2 Summary of the findings

Based on the studies reviewed in this section, there was no consistency in the number of data sets used for evaluation of the proposed ensemble methods, these varied from 1 to 6 per study. For the majority of the studies the selected data sets were obtained from the UCI repository. This dissertation will make use of a larger number of data sets than the ones used by studies in this section. In terms of the GP algorithm the following findings were obtained. In terms of the initial population, studies reported the use of the ramped half and half method in [101,128,129,135,141] whereas in the other studies the details regarding the exact initialisation method were not provided. In this dissertation the ramped half and half method will be used to create ensembles. Crossover, mutation, and reproduction were used in [128, 136], elitism was included in [101], whereas crossover and mutation were used in [129, 130, 134, 141]. Crossover had the highest application rate, followed by mutation and then reproduction. This dissertation will make use of the crossover and mutation operators as these have successfully been used in evolving GP ensembles, and furthermore similar application rates will be used as a guide for parameter tuning. Tournament selection was employed in [101, 129, 134, 136]. Accuracy based fitness functions were successfully implemented in [128–130, 135, 136], and this dissertation will also make use of a similar function. There was little consistency in the number of GP generations and the population size, with values ranging from 20 to 150, and 51 to 4000 respectively. The evolved classifiers were validated using different methods, the train/test split

was used in [101, 129, 130], cross-validation was used in [128, 134, 136, 141], and the leave-one-out method in [135].

The output of each individual in the ensemble has to be combined together in order to produce a single class for each instance of data. Based on the studies reviewed, this was achieved by using a majority vote [101, 130, 131, 136, 142–145], or a weighted majority vote [128, 129, 134, 136, 141, 146]. The majority vote approach represents a simple approach which has successfully been used in several studies to combine individual outputs, and consequently this dissertation will make use of this approach. The number of individuals within an ensemble has to be defined. Brameier and Banzhaf [147] investigated the effects of the size of the teams evolved and the results reveal that there is no significant improvement when more than four individuals are present in an ensemble. Further investigation regarding the effects of the ensemble size would provide useful to researchers constructing GP ensembles.

4.7 Strengths and Weaknesses of Applying GP to Data Classification

This section highlights some of the strengths and weaknesses of solving data classification problems using GP. The key points were extracted from [11, 92, 148].

4.7.1 Strengths

- Accuracy: the review of Espejo *et al.* [92] investigated 66 papers which compared GP to another technique. From these GP outperformed the other methods in 54.72% of the comparisons.
- Interpretability: GP is able to evolve classifiers which are easy for a human to understand. Decision trees are the simplest to understand, and thus data miners can benefit from this and understand why certain classifications are made. Since the classifiers evolved by GP are easily interpreted, and the relationship between the attributes and the classification output is understood, GP evolved classifiers have been referred to as a white box method. Classifiers created using black box methods on the other hand do not offer as much interpretability, such as neural networks. A classifier created using neural networks provides little insight into the relationships between the attributes and the classification output.
- Automatic feature selection: through the use of a fitness function and the concept of fitness, GP is able to indirectly perform the task of feature selection by using only those attributes which contribute to improved fitness and

thus ignoring those that have no impact on fitness. This avoids an additional method to be put in place for feature selection.

- Stochastic search: GP is stochastic in nature, and thus this randomness can prevent the search algorithm from getting stuck in local optima. Greedy search algorithms are susceptible to converging towards local optimum areas of the search space, whereas GP combines both exploration and exploitation to cover a large area of the search space.

- Variety in solutions: in conjunction with the previous point, the randomness involved in GP leads to a variety of solutions to be found after each run. Thus it is possible - after a few GP runs - that a better solution is found in comparison to the previous ones. The variety in the solutions can result in tree classifiers of different shapes and sizes. Combining these different classifiers can be used to create an ensemble.

- Solutions in the form of a computer program: GP evolved classifiers represent computer program-like solutions. Thus the solutions evolved can readily be applied to additional unseen data.

- Fast execution: since the solutions are in the form of a computer program, the solutions can be applied to unseen data and produce a classification output rapidly.

- Flexibility: GP benefits greatly from its flexibility. Additional fitness functions can be created in order to improve the quality of the classifiers, for instance a fitness function can be extended to favour smaller trees, and thus leading to greater interpretability. The genetic operators can be modified, or additional ones can be included.

4.7.2 Weaknesses

- Evaluation: when using GP, each individual within the population has to be evaluated after each generation. This can be extremely time consuming in the domain of data classification due to the large training sets, since for each tree in the population, all the training instances have to be processed by each tree. Consider a training set with 1000 instances and a population size of 500, after each generation there will be 500,000 evaluations done.

- Introns: introns are a well-known issue when applying GP to any domain. Introns affect the interpretability of the classifiers evolved by adding extra

functions and terminals which increase the complexity of the classifiers. Additionally, introns eventually lead to bloat which further hinders the time to execute a generation.

- Number of parameters: chapter 2 discussed the issue that GP faces in terms of having a large number of parameters. Preliminary runs are usually performed in order to fine tune the parameters. However, as previously mentioned, due to the large execution time which arises from large training sets, it results in a very lengthy process to fine tune the parameters for classification.
- Premature convergence: it was previously mentioned that the variety in the GP classifiers can be beneficial, but simultaneously this causes a problem. Despite the fact that GP benefits from its ability to search a large space, nonetheless GP also suffers from premature convergence toward local optimums. Thus, there is no guarantee that GP will always find an optimal classifier to several data sets from different problem domains. In fact, from the literature it can be noticed that the global optimum solution is seldom found.

Using GP to evolve classifiers has several strengths and weaknesses. GP greatly benefits from its ability to search a large space, but is severely hindered by long execution times and premature convergence. Similar to any other method which could be applied to the task of data classification, there is no method which will obtain the best results for every data set. Nevertheless, GP has been used in numerous studies and continues to present itself as an algorithm worth investigating due to its previous successes.

4.8 Conclusion

This chapter presented previous studies which used GP for the task of data classification. The chapter first discussed studies addressed GP representations for data classification. This will followed by a discussion on GP ensemble methods for data classification. The strengths and weaknesses of applying GP to data classification were then highlighted. The remainder of this conclusion provides a summary of the sections reviewed in this chapter, and provides the rationale behind the investigations which will be performed based on existing work. The next chapter will describe how each of the objectives listed in chapter 1 will be met based on the justifications provided below.

4.8.1 GP for data classification

Based on the findings of the studies reviewed in this chapter, the following observations were made regarding the GP implementations. Not all the studies re-

viewed mention which initial population generation method was used. However the ramped half and half method was reported in [9, 98, 100, 101, 104, 106, 107, 109, 110, 117–119, 123–129, 135, 141]. This dissertation will also make use of the ramped half and half method since it has successfully been used in previous studies and provides greater diversity as discussed in chapter 2. The tournament selection [8, 97, 98, 101, 105, 106, 108, 111, 118, 126, 129, 134, 136] and fitness proportionate selection [95,100,104,109,113,125] were the two most commonly used parent selection methods. Tournament selection was successfully used in a large number of studies reviewed in this chapter. As was discussed in chapter 2, this method applies selection pressure whereas fitness proportionate does not. This method will be used in this dissertation given its popularity and flexibility. Crossover, mutation, and reproduction where the three GOs which were used the most. Typically, when those were used, the crossover had the highest application rate, followed by mutation, and then reproduction, this was observed in [9,101,105,113,118,127,135], where these selected studies used different representations and these also include studies investigating ensembles. This dissertation will make use a similar setting for the GOs. The fitness of the GP individuals were mostly obtained using the accuracy as well as the a measure of complexity such as the size of the individuals, studies which made use of those two measures as a fitness function include [91, 93, 97, 122], alternative fitness functions include the gain ratio measure in [8], or the sensitivity and specificity in [125]. The classification accuracy has been widely employed and this dissertation will also make use of a similar fitness function by taking into consideration the accuracy and the size of the GP individuals. The two most used methods for evaluating the performance of the evolved GP classifiers were k-fold cross-validation [8,97,98,109,111,113,127,134] and the train/test split method [91, 99, 105, 122, 125]. The most common value for k was 10. The train/test split method is computationally less expensive than k-fold cross validation, however only a single test set is used and it is possible that the test set contains instances of data which are easier to classify. Thus, this dissertation will also make use of the 10-fold cross-validation as this method allows each fold to be used as a test set which will lead to more reliable results. The remaining subsections describe the findings obtained for the different areas of GP and data classification which have been drawn out from the literature. The details provided in this subsection will be applied to the objectives justified below.

4.8.2 GP representations for data classification

Linked to the first objective of this dissertation, the three major GP representations for data classification reviewed in this chapter were decision trees, arithmetic trees, and logical trees. Studies for each of these representations were analysed, and there has been no study which has attempted to compare the performance of these GP

representations for data classification. Thus, this dissertation will perform a comparison of the three representations. New researchers in the field of GP and data classification may not know which representation to select given the choice of the three. This serves as the rationale behind this objective. By addressing this objective, new researchers will be able to decide which is the most suitable representation based on the comparison.

Based on the studies reviewed, the following findings were made. Each of these representations makes use of a different function and terminal set. Decision trees use the attributes as functions, and classes and terminals. Decision trees have frequently been used in the literature, and a significant advantage of this representation is that it offers high interpretability. The evaluation is performed by starting at the root node, and following a path to a leaf node by deciding on which branch to follow.

Arithmetic trees use mathematical functions as the function set, and the attributes as the terminal set. For binary classification problems, a threshold value of zero is often used to map the output of a tree to one of the classes. In the case of multiclass classification, static ranges can be used to map the output onto the classes; however, there have been studies which made use of dynamic ranges. Another approach is to make use of binary decomposition and to run the GP algorithm the same number of times as there are classes. For each class i, the evolved GP trees represent a classifier which outputs one of two possible values, a value which denotes that an instance belongs to class i, or a value which denotes that an instance does not belong to class i. Each of the classifiers are then combined together to produce a final classification model. This however provides the additional complexity of dealing with clashes when two or more classifiers output that a particular instance belongs to all of them.

When evolving logical trees using GP, the function set is made up of logical functions, and the terminal set is made up of the attributes. When dealing with numerical attributes, inequalities are usually added to the function set, and the equal sign can be added to the function set when dealing with categorical attributes. Similar to evolving arithmetic trees using GP, logical trees require that binary decomposition is used in order to handle multiclass classification problems.

From the reviewed studies, the following aspects have been drawn out and will be used in this dissertation. Decision trees will make use of the attributes as the function set, and the terminal set will be composed of the classes. A static threshold value of 0 will be used when evolving arithmetic trees as this approach has successfully been implemented in the studies reviewed. This threshold approach will be extended and used with logical representations in order to provide a simple means of discriminating between two classes. Mathematical functions will be used as the function set for arithmetic trees, and the terminal set will be composed of the attributes. Logical

functions will be used as the function set for logical trees, and the terminal set will be composed of the attributes. Additionally, the findings presented in section 4.8.1 will be used in order to implement the three representations.

4.8.3 GP discretisation for data classification

Decision trees can easily be applied to data sets which are made up of categorical attributes; however, discretisation is required when dealing with numerical attributes. Based on the studies reviewed, the existing work has dealt with using discretisation methods prior to the GP run. In terms of incorporating discretisation into evolutionary algorithms, the only attempts where those that made use of ADIs and GAs. There has been no research which combines discretisation into the GP algorithm, thus this dissertation will investigate the combination. This is the second objective of this dissertation. Based on the studies which have investigated ADIs, this dissertation will propose methods for adapting the intervals during the evolutionary process. The findings presented in section 4.8.1 will be used in order to implement the proposed methods.

4.8.4 GP encapsulation for data classification

Chapter 2 discussed the use of modularisation in order to reuse subtrees during the evolutionary process. Programmers often tend to write functions which encapsulate a piece of code that can be reused several times during the execution of the program. In the context of GP, the encapsulation genetic operator was proposed by Koza to achieve a similar task. This genetic operator selects a subtree within a larger tree and encapsulates it, and thus preserving the subtree from the destructive effects of the crossover operator. There has been no previous work which has attempted to incorporate modularisation to the GP algorithm for data classification. This dissertation shall thus investigate the use of the encapsulation GO in the context of data classification in order to determine whether this can enhance the performance of the classifiers. This objective will also determine if, in the context of data classification, the encapsulation operator will preserve the subtrees from the destructive effects of the crossover operator previously mentioned. This represents the third objective of this dissertation. The findings discussed in section 4.8.1 will be used to implement the proposed GP encapsulation approach.

4.8.5 GP ensembles for data classification

An approach to improving the accuracy of the classifiers is to use GP to evolve ensembles. From the literature reviewed, it is apparent that evolving ensemble classifiers can outperform the standard GP approach. Researchers have investigated different

approaches as how to use GP for evolving ensembles. These vary from creating ensembles at the end of the GP execution using the final population, and other approaches whereby the ensemble is evolved during the GP execution. Typically bagging or boosting algorithms can be used with GP in order to create ensembles. A hybridisation between evolutionary algorithms has not previously been investigated in the context of data classification, and thus this investigation serves as the rationale behind the fourth objective. The final objective is to propose a new GP boosting for creating ensembles. The rationale behind this approach is to propose a new method which makes use of a simple approach to use weights in order to allocate a measure of difficulty to the training data. Furthermore, objectives 4 and 5 provide a means of comparing the performance of a population of ensembles to a single evolved ensemble. There has been no previous attempt at such a comparison. The overall issue when evolving ensembles is how to combine the individual classifier outputs into a single final output; typically a majority vote has been used. This dissertation will thus also make use of a majority vote approach. Furthermore, the findings discussed in section 4.8.1 will be used to implement the proposed two ensemble approaches.

Chapter 5

Methodology

5.1 Introduction

This chapter presents the methodology which will be used to address the objectives stated in chapter 1. Section 5.2 describes how each of the objectives outlined in chapter 1 will be achieved. The statistical testing which shall be used in this dissertation is detailed in section 5.3. Section 5.4 describes the characteristics of the 21 data sets used in this dissertation, and additionally provides the rationale for the selected data sets. The overall GP system which will be used in this dissertation is provided in section 5.5. Section 5.6 discusses how the experiments were run and how the results were obtained. Section 5.7 provides details regarding the technical specifications. Finally, this chapter is concluded in section 5.8.

5.2 Addressing the objectives

Based on the implementations of GP for data classification found in the literature, a generational GP algorithm will be implemented. This GP algorithm for data classification shall serve as a baseline approach, and the additional methods which shall be proposed in the following objectives will consequently be compared to this baseline approach. The objectives of this dissertation are listed below along with a discussion on how these objectives will be addressed. Furthermore, for each of the objectives described, the results will be statistically tested in order to determine the effectiveness of the proposed methods. The statistical tests are detailed in section 5.3.

- Objective 1: Incorporating discretisation into GP.

  In order to fulfil this objective, discretisation will be incorporated into the GP algorithm. From the literature reviewed in the previous chapter, it is

apparent that the idea of incorporating discretisation into a GP algorithm has yet to be proposed, and thus, various ways of incorporating discretisation into a GP algorithm will be tested on publicly available data sets which contain only real-valued attributes. The performance of each of the methods will be investigated and compared to one another using statistical tests. Additionally, the proposed methods will be compared to other discretisation methods found in the literature.

- Objective 2: GP representations for data classification.

  The previous chapter presented several studies on each of the three representations, however there has been no study comparing the three. In order to fulfil an investigation on the three commonly used representations, the three GP representations will be implemented and tested on publicly available data sets, and a comparison between each of the three will be made. This comparison will include determining which of the representations obtains the highest training and testing accuracy, and additionally, to compare the size of the evolved representations. Due to the fact that logical trees can only output two values, true or false, this representation can only be applied to binary classification problems. Thus, in order to compare the representations, these methods will only be compared to each other on binary classification problems. Statistical testing will be applied to the results in order to provide a detailed comparison of the performance of the representations.

- Objective 3: Encapsulation genetic operator and data classification.

  In order to accomplish this objective, the encapsulation genetic operator will be included as an additional operator in the baseline GP algorithm. This approach will be investigated and the findings analysed in order to determine the effect of the encapsulation operator. Since this represents the initial attempt at applying the encapsulation genetic operator in a GP algorithm for data classification, the proposed approach will be refined in order to determine additional ways of enabling the encapsulation operator to improve the performance of the classifiers. When the encapsulation operator is applied to GP, encapsulated terminals are created and added to the trees. Thus the number of encapsulated terminals in the evolved classifiers will be measured in order to determine if the number of encapsulated terminals impact the GP trees. The proposed approach will be tested on several publicly available data sets. The results of the proposed encapsulation approach will be statistically compared to the results of the standard GP algorithm without encapsulation (the baseline algorithm) in order to verify the effectiveness of the proposed method.

- Objective 4: Hybridising evolutionary algorithms for classifier ensembles.

  Existing studies detailed in the previous chapter revealed that ensemble methods have proven themselves superior to single classifier methods. However, there has been no attempt to hybridise GP with another evolutionary algorithm in order to create ensembles. This objective will be fulfilled by creating various hybridisations of the GP algorithm with a genetic algorithm in order to evolve a population of ensemble classifiers. By introducing the genetic algorithm there are additional parameters which need to be fine-tuned, and these will be optimised on selected data sets. Ensembles of different sizes will be investigated in order to determine the optimal ensemble size. The performance of these various hybridisations will be compared to each other on several publicly available data sets, and additionally compared to the baseline GP algorithm in order to determine whether the proposed methods can outperform the baseline approach. The results will be statistically tested. Finally, the proposed hybridisations will be compared to other ensemble methods found in literature.

- Objective 5: GP and ensemble construction.

  Both objectives 4 and 5 deal with investigating methods which create ensembles, however, this objective differs from objective 4 in the number of ensembles that are considered. In objective 4, the goal was to hybridise two evolutionary algorithms in order to evolve a population of classifier ensembles and output the best ensemble at the end of the run. This objective, on the other hand, focuses on creating a single ensemble classifier which is output at the end of the GP run. In order to meet this objective, the GP algorithm must incrementally add trees to a single ensemble. Thus, a proposed method for adding trees will be investigated. This will be done by incorporating weights similar to the way they are used in boosting methods. The results for each ensemble size will be investigated and compared to each other. The results from the proposed method will be statistically compared to the results from the baseline GP approach, and additionally, the proposed method will be compared to ensemble methods found in the literature. The proposed approach will be tested on several publicly available data sets. Furthermore, the ensembles proposed in objective 4 will be compared to the proposed method in this objective.

5.3 Statistical testing

According to the central limit theorem [149], as the sample size increases, the distribution of the sample means approaches a normal distribution. A sample size greater

than 30 is sufficient to achieve a normal distribution. Thus, if more than 30 GP runs are performed for each data set, then enough results will be collected to ensure that statistical tests can be conducted.

In this dissertation, when a comparison between methods is conducted, the z-test will be applied with $\alpha = 0.05$. Assume two classification methods A and B are being compared, with means μ_A and μ_B respectively. The first step is to formulate the null and the alternative hypothesis as follows:

$$H_0 : \mu_A = \mu_B$$

$$H_a : \mu_A > \mu_B$$

When applying the z-test, a p-value is computed and compared to the value of α [150]. If the p-value is less than α then the decision is to reject the null hypothesis, and consequently method A outperforms method B. Conversely, if the p-value computed is greater than or equal to α, then the decision is to accept the null hypothesis, and consequently the two classification methods have the same mean performance.

5.4 Data Sets

This section describes the data sets that are used throughout this dissertation. Eleven binary data sets and ten multiclass data sets were selected from the UCI machine learning repository [27].

5.4.1 Characteristics of data sets for data classification problems

A data set is generally described using the following characteristics:

- Number of attributes
- Number of instances
- Number of classes
- Class balance

While it is true that the actual data dictates the complexity of a data set, the characteristics of the data sets also have an impact on the difficulty to create an accurate classifier. It is trivial to create a classifier if for a data set, all the instances belonging to class A have attributes made up of negative values, and all the instances of class B have attributes made up of positive values. Clearly in this situation, the data itself renders the task of creating an accurate classifier simple.

Assuming the data itself is not trivially separable into the correct classes, then the characteristics of the data sets impact the difficulty involved in creating accurate classifiers. For instance, large data sets which contain over thousands of instances can increase the complexity of the classification task.

The class balance can significantly impact the complexity involved in creating classifiers in the case of highly unbalanced data sets. This imbalance can result in the classifier being biased towards the majority classes; consequently which reduces the accuracy on the minority classes [29,151]. The issue of class imbalance is observed in both binary and multiclass classification problems. Medical diagnosis problems are typically formulated as a binary classification problem whereby the class of interest is often the minority class [29]; thus data sets which are unbalanced are of interest.

Finally, the number of attributes can impact the complexity of a data set. Data sets which contain a large number of attributes are affected by the *curse of dimensionality* [152]. As the number of attributes increases, the dimensionality of the problem increases, and consequently creating accurate classifiers becomes more challenging.

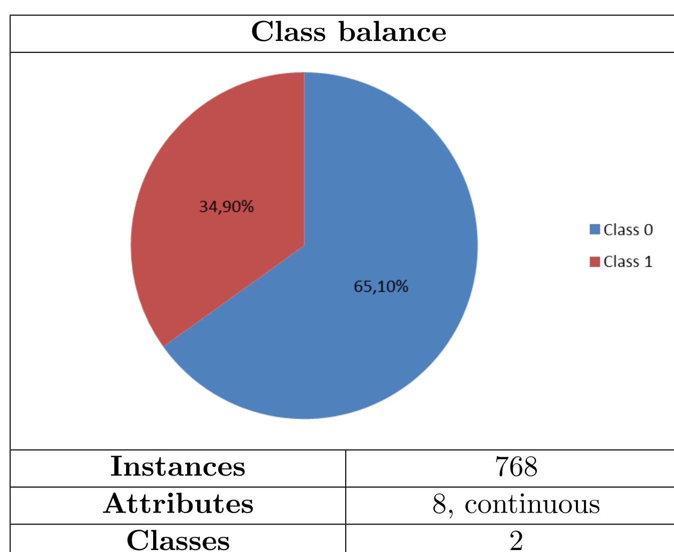
Based on the various characteristics of data sets for classification problems; several data sets were selected in such a way as to address the different characteristics. The selected data sets are presented in the following subsections.

5.4.2 Binary data sets

The eleven binary data sets are described below:

Pima Indians Diabetes (Pima Indians)

Class balance



Instances	768
Attributes	8, continuous
Classes	2

Table 5.1: *Pima Indians* data set characteristics.

Connectionist Bench - Sonar, Mines vs. Rocks (Sonar)

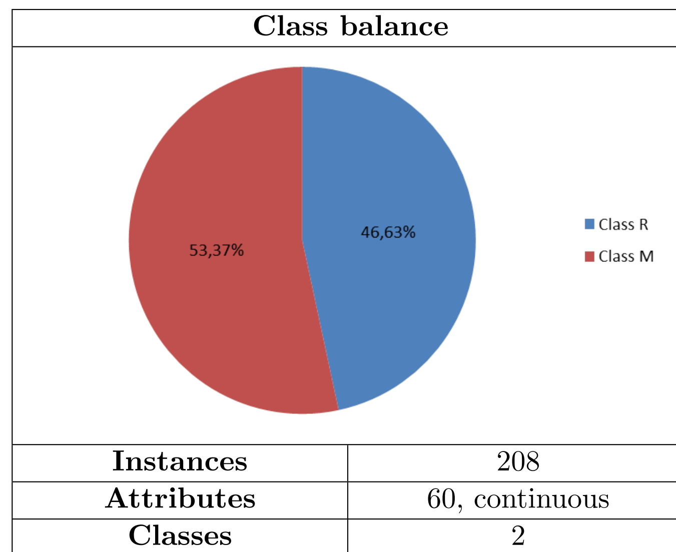
Class balance	
	
Instances	208
Attributes	60, continuous
Classes	2

Table 5.2: *Sonar* data set characteristics.

Breast Cancer Wisconsin - Diagnostic (WDBC)

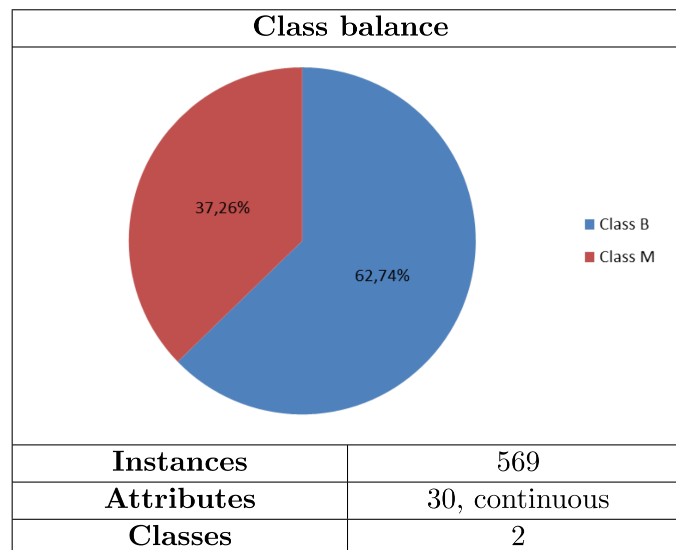
Class balance	
	
Instances	569
Attributes	30, continuous
Classes	2

Table 5.3: *WDBC* data set characteristics.

Parkinsons

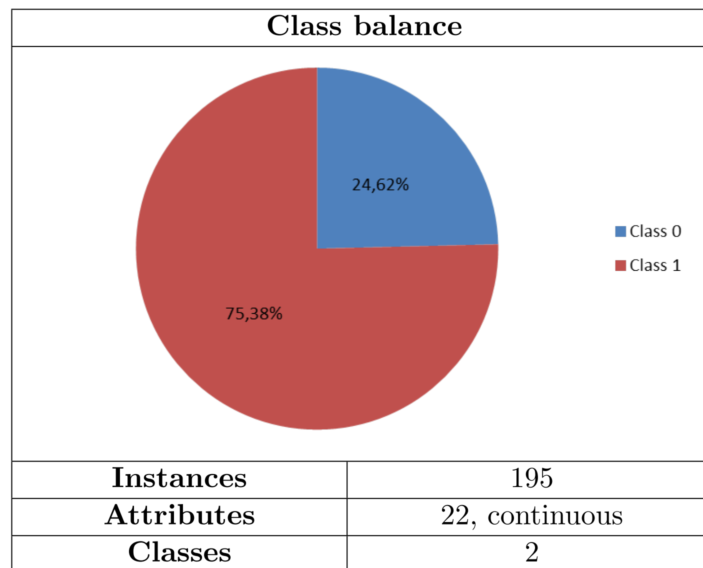
Class balance	
	
Instances	195
Attributes	22, continuous
Classes	2

Table 5.4: *Parkinsons* data set characteristics.

Mammographic Masses (Mammographic)

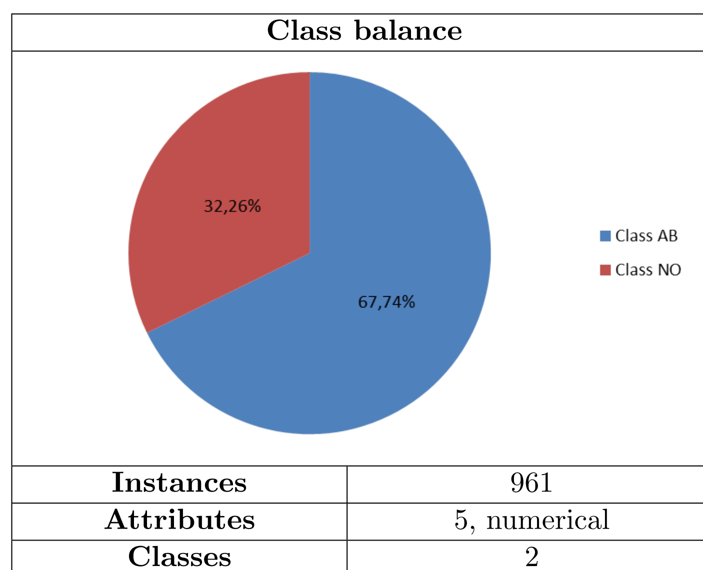
Class balance	
	
Instances	961
Attributes	5, numerical
Classes	2

Table 5.5: *Mammographic* data set characteristics.

Ionosphere

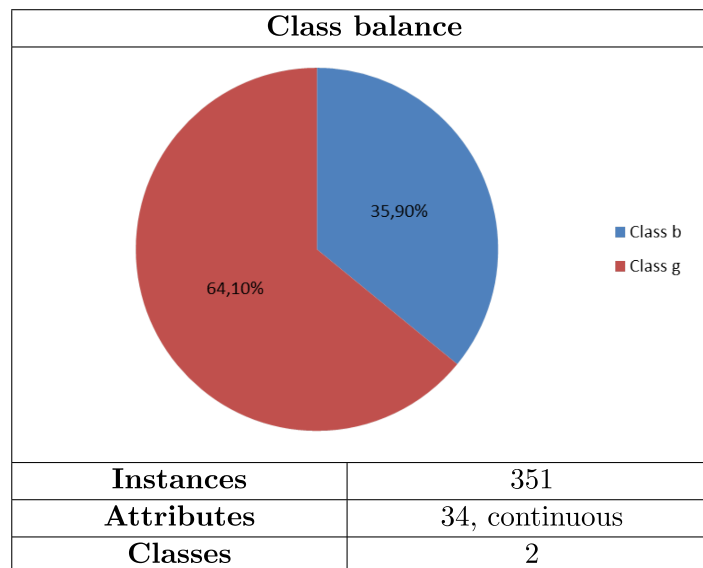
Class balance	
	
Instances	351
Attributes	34, continuous
Classes	2

Table 5.6: *Ionosphere* data set characteristics.

SPECTF Heart (Spectf)

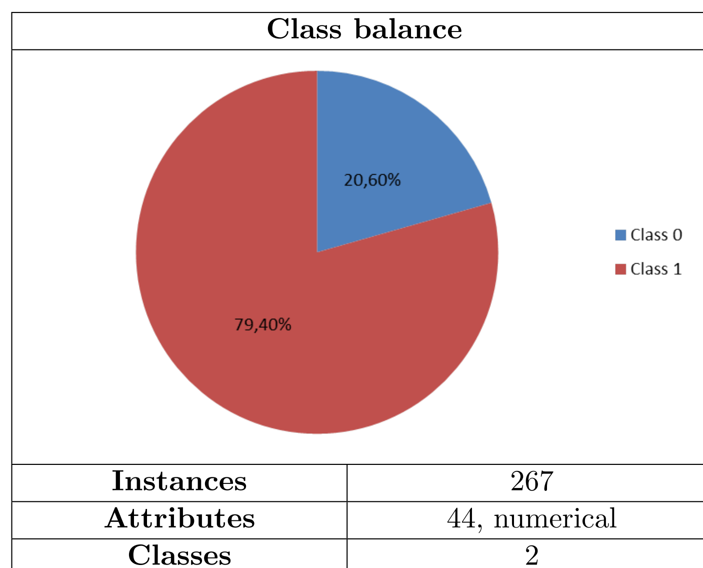
Class balance	
	
Instances	267
Attributes	44, numerical
Classes	2

Table 5.7: *Spectf* data set characteristics.

Climate Model Simulation Crashes (Climate)

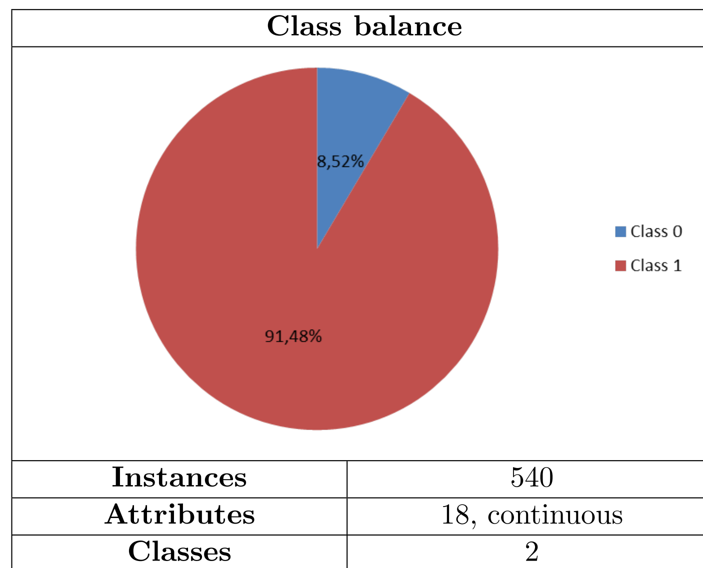
Class balance	
	
Instances	540
Attributes	18, continuous
Classes	2

Table 5.8: *Climate* data set characteristics.

Fertility

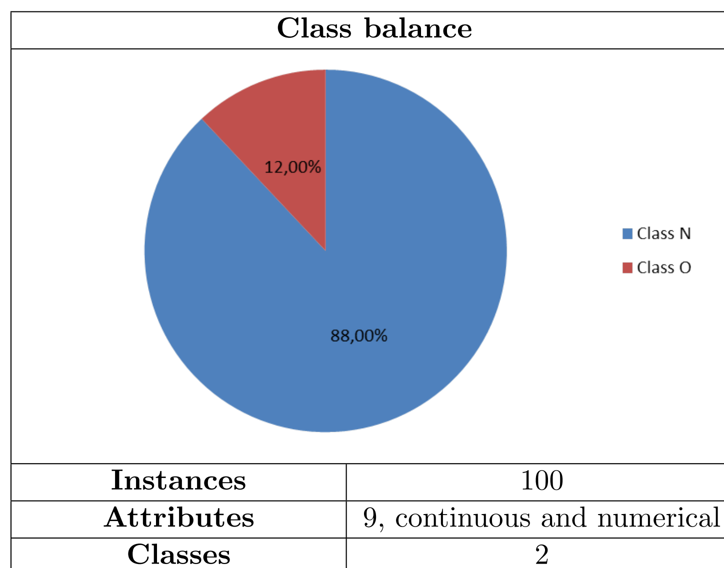
Class balance	
	
Instances	100
Attributes	9, continuous and numerical
Classes	2

Table 5.9: *Fertility* data set characteristics.

MONK's Problems (Monk2)

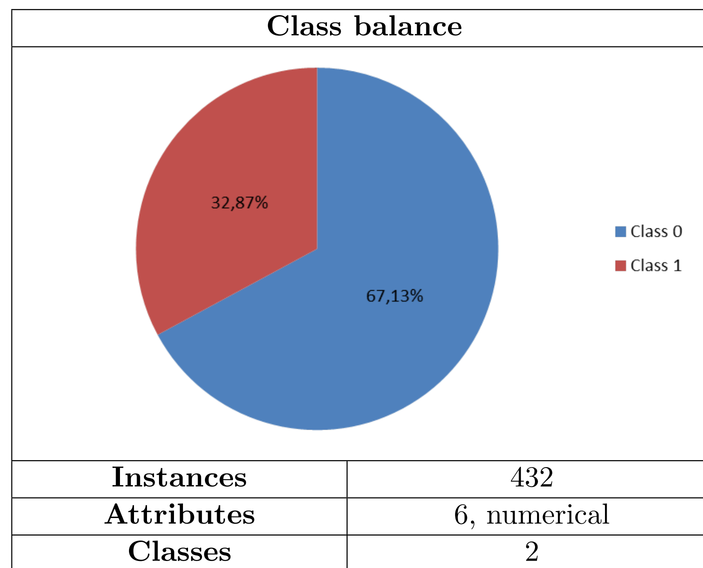
Class balance	
	
Instances	432
Attributes	6, numerical
Classes	2

Table 5.10: *Monk2* data set characteristics.

Tic-tac-toe (TTT)

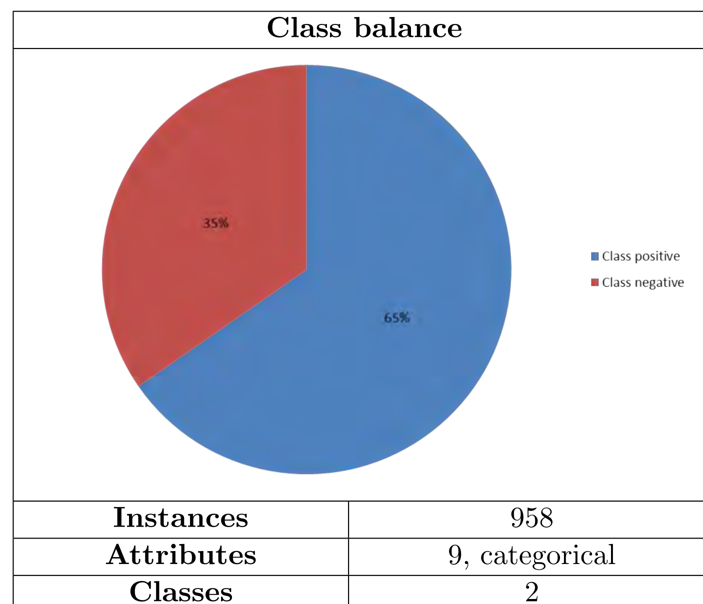
Class balance	
	
Instances	958
Attributes	9, categorical
Classes	2

Table 5.11: *TTT* data set characteristics.

5.4.3 Multiclass data sets

The ten multiclass data sets are described below:

Balance Scale (Balance)

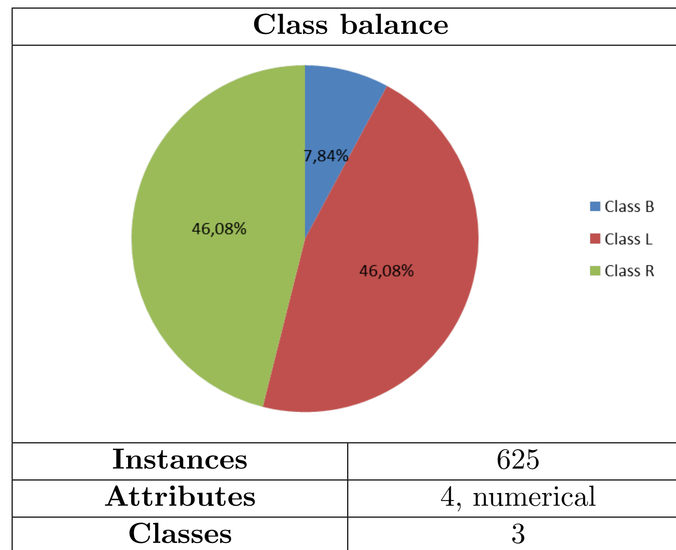
Class balance	
	
Instances	625
Attributes	4, numerical
Classes	3

Table 5.12: *Balance* data set characteristics.

Car Evaluation (Car)

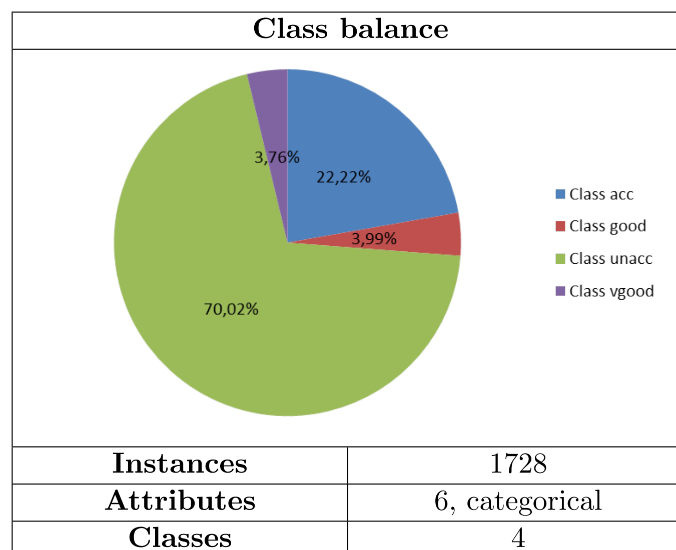
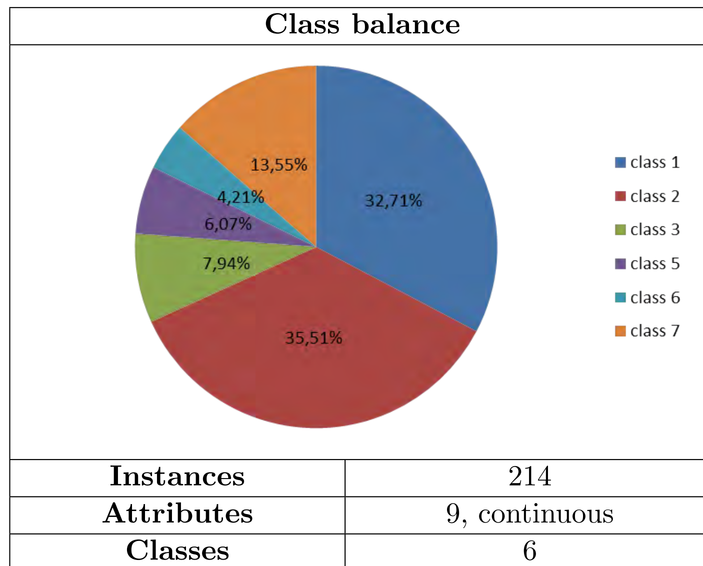
Class balance	
	
Instances	1728
Attributes	6, categorical
Classes	4

Table 5.13: *Car* data set characteristics.

Glass Identification (Glass)

Class balance

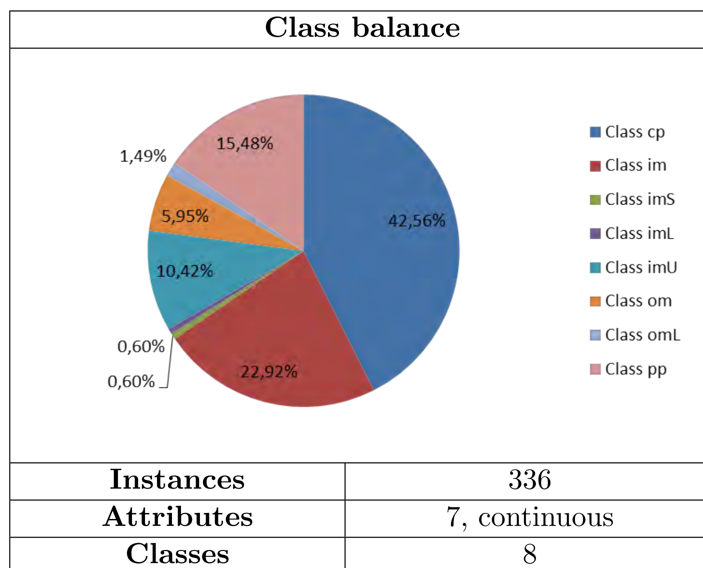


Instances	214
Attributes	9, continuous
Classes	6

Table 5.14: *Glass* data set characteristics.

Ecoli

Class balance

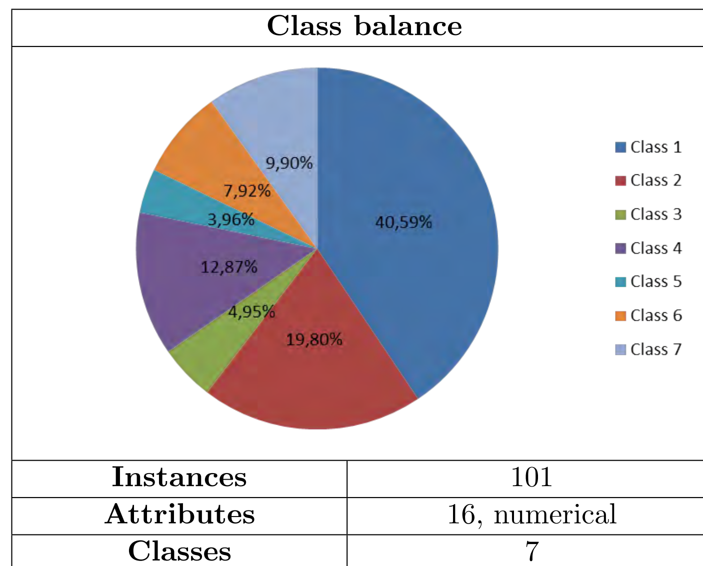


Instances	336
Attributes	7, continuous
Classes	8

Table 5.15: *Ecoli* data set characteristics.

Zoo

Class balance

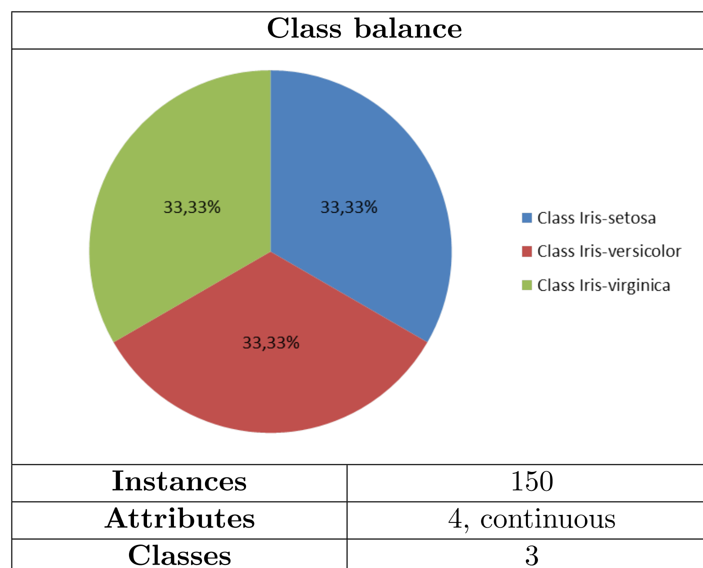


Instances	101
Attributes	16, numerical
Classes	7

Table 5.16: *Zoo* data set characteristics.

Iris

Class balance

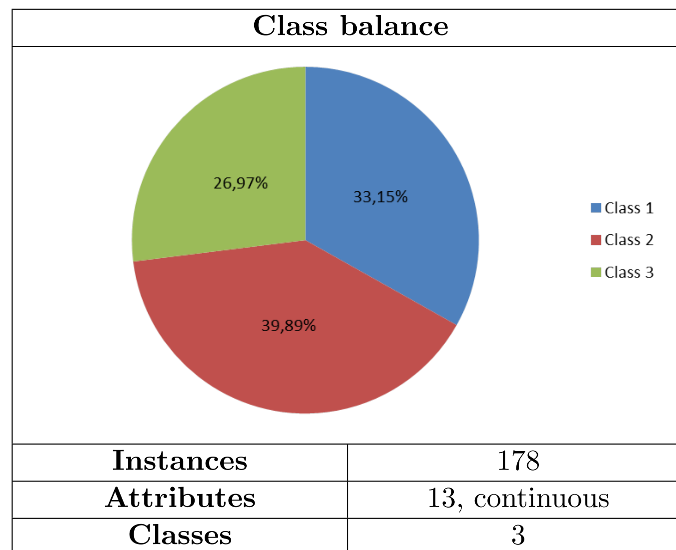


Instances	150
Attributes	4, continuous
Classes	3

Table 5.17: *Iris* data set characteristics.

Wine

Class balance

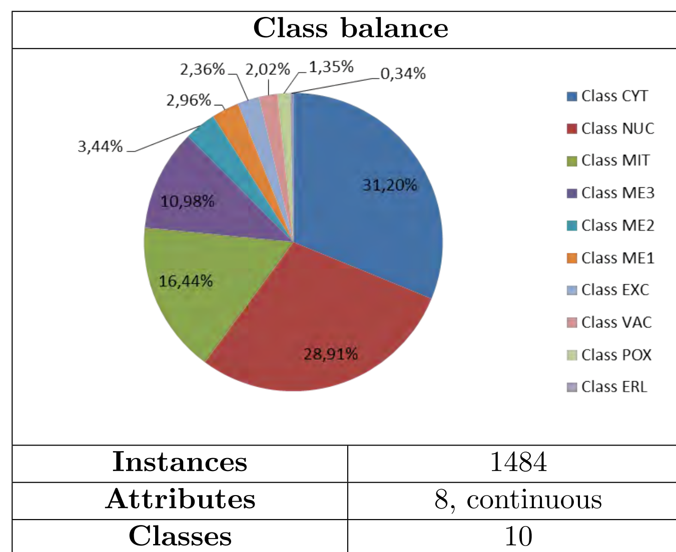


Instances	178
Attributes	13, continuous
Classes	3

Table 5.18: *Wine* data set characteristics.

Yeast

Class balance



Instances	1484
Attributes	8, continuous
Classes	10

Table 5.19: *Yeast* data set characteristics.

Statlog Vehicle Silhouettes (Vehicle)

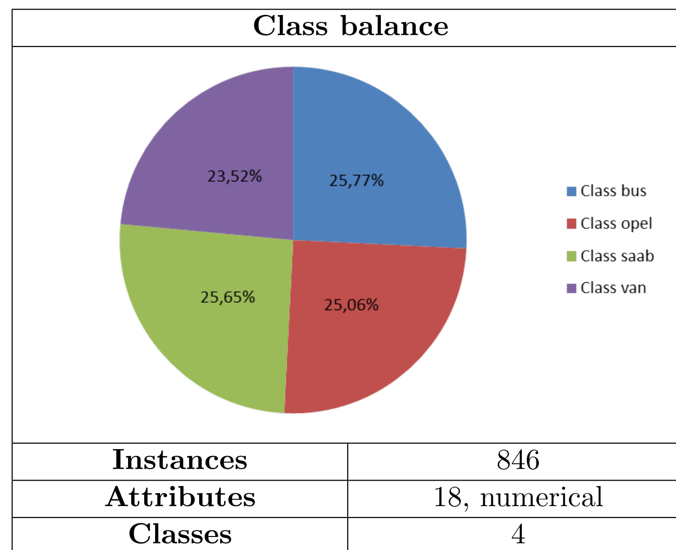
Class balance	
	
Instances	846
Attributes	18, numerical
Classes	4

Table 5.20: *Vehicle* data set characteristics.

Soybean Large (Soybean)

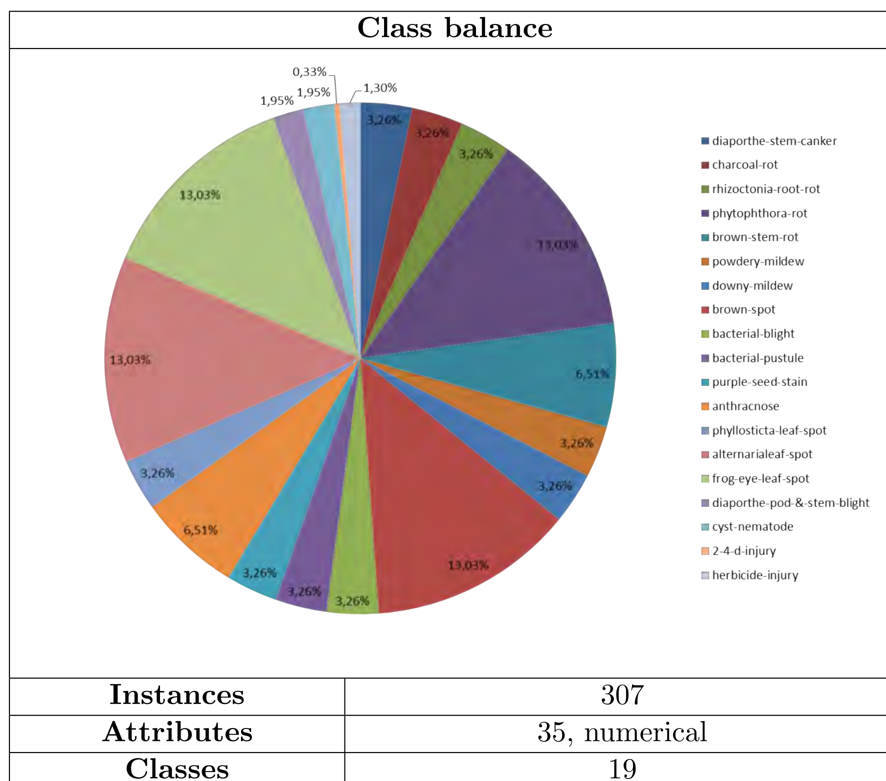
Class balance	
	
Instances	307
Attributes	35, numerical
Classes	19

Table 5.21: *Soybean* data set characteristics.

5.4.4 Rationale behind the selected data sets

Figure 5.1 illustrates the difference between the number of attributes and instances within the binary data sets. These data sets are made up of different characteristics; although certain of these data sets may have a similar number of attributes, they distinguish themselves in the number of instances. For instance, the *Fertility* and *Pima Indians* data sets have a similar number of attributes, however the *Pima Indians* data set has over seven times the number of instances. The selected binary data sets range in the number of instances; from 100 to 916, and additionally there is a variety of attribute dimensionality from 5 to 60. Additionally, certain data sets are well-balanced whilst others are imbalanced. *Mammographic* contains missing numerical values which will be dealt with by imputing each missing value with the corresponding attribute median in which the missing value is found.

The data sets were selected in such a way as to represent different data classification application areas (for example life, physical, and social), and they were selected to contain data sets of dissimilar characteristics. Selecting data sets with similar characteristics would imply that the GP algorithm would be evaluated on problem specific domains, and consequently, the results would not be a true representation of the performance of the proposed methods.

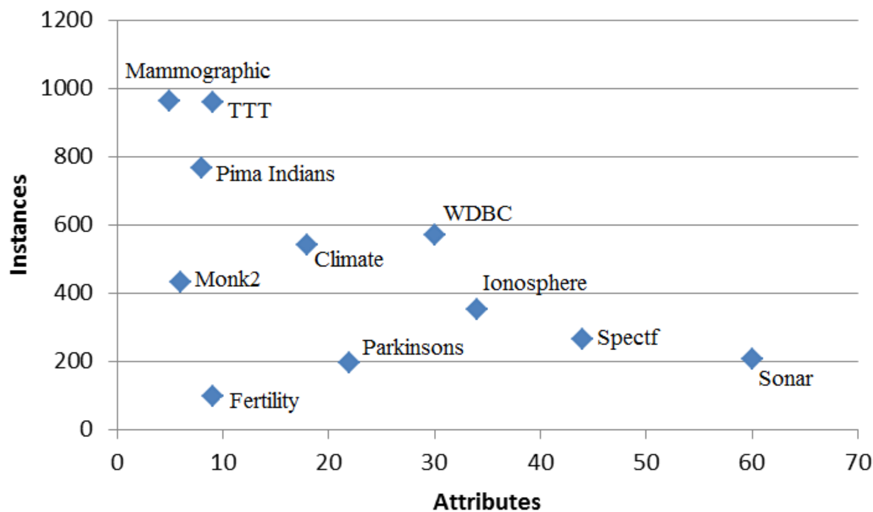


Figure 5.1: Difference in the number of attributes and instances in the binary data sets.

Figures 5.2 and 5.3 illustrate the differences in the number of attributes, instances and classes within the multiclass data sets. Similarly to the binary data sets, these

data sets have different characteristics. The selected multiclass data sets range in the number of instances from 101 to 1728, and additionally, range in the number of attributes from 4 to 35. The number of classes varies from 3 to 19. *Soybean* contains missing categorical values which will be dealt with by imputing each missing value with the mode for each attribute. The selected data sets thus represent different problems with unique challenges in terms of the characteristics described in section 5.4.1.

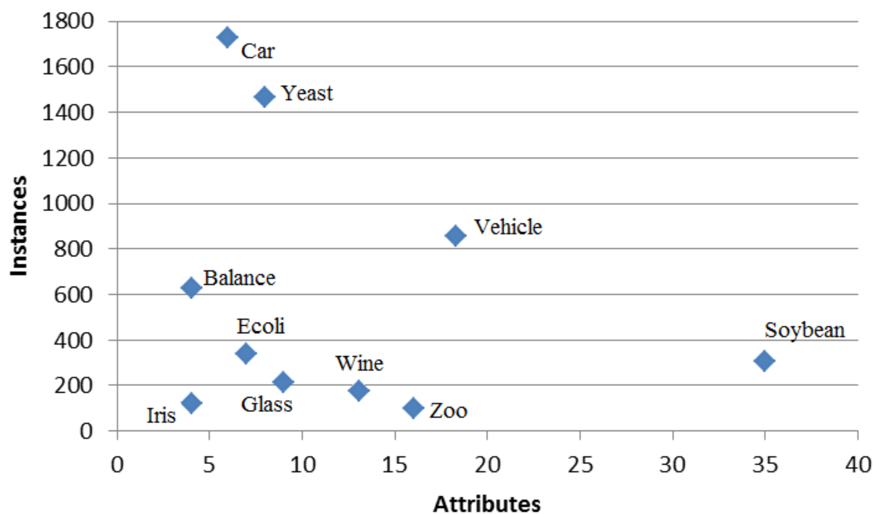


Figure 5.2: Difference in the number of attributes and instances in the multiclass data sets.

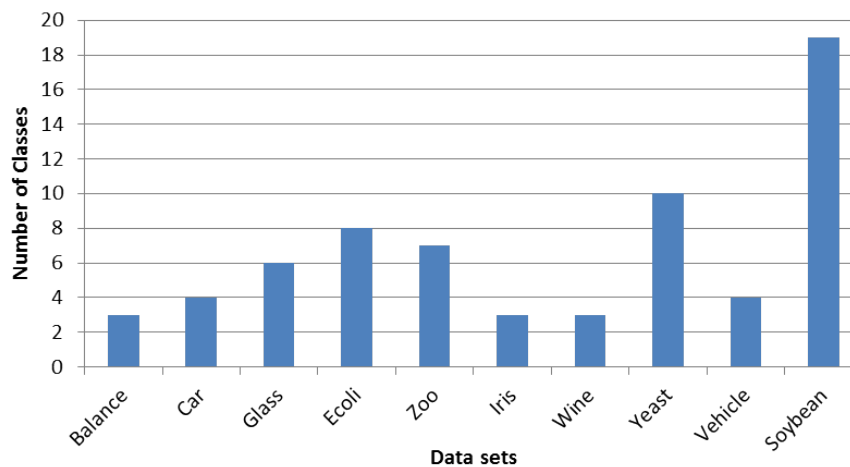


Figure 5.3: Difference in the number of classes in the multiclass data sets.

In the domain of data classification and machine learning, there is no rule of thumb to determine how many data sets should be used in a study in order to

evaluate the performance of an algorithm. McDermott *et al.* [153] surveyed 183 articles in which 20.2% of them were related to data classification. The findings point out that on average only 3.5 data sets were used per article in studies related to data classification. From the literature surveyed in this dissertation, it was apparent that there is a lack of consistency in the number of data sets used in previous studies. It was thus determined that these selected data sets represent a sufficient amount of problems, in terms of their varying characteristics, in order to investigate the objectives of this dissertation.

5.5 GP System

Unless otherwise stated, the generational GP algorithm will be implemented for all of the proposed methods. This was defined in algorithm 2.1. The ramped half and half method will be used to generate the initial population, this was discussed in section 2.7. Tournament selection will be used to select the parents for the GOs, and this was presented in algorithm 2.2. When a parent is to be selected for a GO, the tournament selection method is executed twice and thus two individuals are obtained. These two individuals are first compared in terms of accuracy, and the one with the highest accuracy is returned. If they have the same accuracy, then the number of nodes within each individual is compared, and the one with the smallest number of nodes is returned. Should they both have the same accuracy, and the same number of nodes, then the individual to be returned is selected at random. Crossover and mutation will be used, and these were described in sections 2.10.3 and 2.10.2 respectively. The GP crossover operator makes use of hill climbing in the following manner: the operator runs for 10 attempts and, on each attempt, performs crossover on the original parents. The resulting offspring are examined on each attempt, and if the training accuracy is improved, the fitter of the two offspring is then returned. If there is no improvement across the ten attempts, then the fitter of the two original parents is returned. Accuracy will be used as the fitness function, and was defined in section 3.4. Section 5.6 describes how the GP system will be run and how the results will be collected.

This dissertation will make use of a multithreaded architecture in order to deal with the issue of large runtimes. In order to achieve this, the GOs will be performed in parallel based on the number of threads allocated to the task. For each execution of a GO, if n offspring are to be created, and t threads are allocated, then each thread will create $\frac{n}{t}$ offspring in parallel. For instance, if 8 threads are allocated, and 800 offspring are to be created using a GO, then each of the 8 threads will be responsible for creating 100 offspring in parallel. This will lead to shorter runtimes than creating the offspring sequentially.

5.6 Performance Measures

Due to the random nature of the GP algorithm, several runs of the proposed methods need to be run. The randomness may cause the GP algorithm to not perform consistently across all runs, and thus a total of fifty runs will be executed in order to obtain a sufficient amount of results. Additionally, by performing a large number of runs, in this case fifty, there is enough data to apply statistical tests in order to verify the statistical significance of the results. In order to determine the performance of the evolved GP classifiers on the data sets, the 10-fold cross-validation will be implemented as follows:

1. Randomly create 10 folds from the original data.
2. Use one fold for testing and the remaining 9 folds for training. Repeat this process ten times ensuring that each fold is used exactly once for testing while the remaining folds are used for training.
3. Repeat steps 1 and 2 five times.

Following the implementation described above, a total of fifty GP classifiers are created for each data set. The average classification accuracy on the training and test data is recorded and averaged across the fifty runs. A random seed is created for each GP run.

5.7 Technical Specifications

The algorithms proposed in this dissertation were written in Java 1.6 using Netbeans 7.3. The technical specifications of the computer used to develop the proposed algorithms are as follows: Intel Core i7-3630QM @ 2.4GHz with 8GB RAM running Windows 8.1. The statistical tests were performed using Microsoft Excel 2010. The simulations were performed on the Center for High Performance Computing.

5.8 Conclusion

This chapter describes the methodology which will be used to achieve the objectives described in chapter 1. Based on the literature reviewed in chapter 4, it was observed that there was no consistency between studies regarding the data sets selected. Furthermore, no rationale was provided for the selected data sets in the existing studies. The data sets which will be used in order to determine the effectiveness of the proposed algorithms were described in this chapter; along with an explanation on how the statistical tests will be conducted. Data sets with varying characteristics were

selected in order to represent a variety of data classification problems. The overall proposed GP system details were presented. Finally, the method for determining the performance of the proposed algorithms was presented.

Chapter 6

Incorporating Adaptive Discretisation into Genetic Programming for Data Classification

6.1 Introduction

GP using a decision tree representation has been used numerous times for data classification problems when the attributes are categorical. One of the shortcomings of decision trees is that they cannot directly handle continuous data. Discretisation, discussed in chapter 3 section 3.7.4, is often required as a pre-processing step in order to apply algorithms to data sets having continuous data. This chapter investigates how GP can incorporate adaptive discretisation for data sets containing continuous attributes.

Two approaches for incorporating discretisation into the GP algorithm are described in section 6.2. Furthermore, section 6.2 describes a new GO for altering the intervals dynamically. Section 6.3 describes the experimental setup, lists the data sets which will be used for testing, and provides details on the GP parameters. Finally, section 6.4 concludes this chapter.

6.2 Proposed Discretisation Methods for GP

When GP is applied to data classification using a decision tree representation, each node within the tree represents an attribute and the leaf node represents the class value.

An example of such a GP tree is illustrated in figure 6.1. In this case the attribute represents categorical data and a branch is created for the two possible values, *male*

and *female*. This example also illustrates the shortcoming of decision trees when dealing with continuous data since creating a branch for each numerical value would not be feasible.

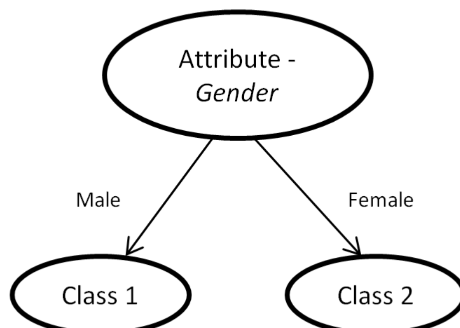


Figure 6.1: Example of a GP tree using a decision tree representation.

Two major approaches were developed in order to determine which discretisation method is most effective. The first creates intervals of equal width and those intervals remain constant throughout the GP execution; this approach was named Equal Width Intervals (EWI). The second approach creates random intervals and alters those intervals during the GP execution; this approach was named GP Evolved Intervals (GPEI). Thus the approaches serve as a comparison between a static and a dynamic interval approach.

An arity has to be specified when creating intervals in order to split them. Two approaches were investigated, a fixed and a varying approach, in order to examine the effect that the arity of the nodes has on the proposed discretisation methods. The *fixed arity approach* sets an arity, and each node in the GP decision trees has to take on that arity. The arity is never changed during the execution of the GP algorithm. The *varying arity* approach allows GP to randomly select the arity during execution of the GP algorithm. In this approach, the GP algorithm randomly selects the arity of a node when it is being created. Nodes are created during initial population generation and when the mutation operator is applied. The rationale behind this approach was to determine whether decision trees with a fixed arity at every node results in the same classification accuracy as trees created with nodes of varying arity values. In this research, the minimum arity was 2 and the maximum was 4, thus when the varying arity was applied, GP could select an arity of 2, 3, or 4 when creating a node. In both approaches, the number of intervals created for a node is equal to the node's arity.

6.2.1 Equal Width Intervals (EWI)

The EWI approach creates intervals of equal width depending on the arity of the node. The intervals are created by taking the minimum and maximum values for an attribute and dividing by the selected arity. The selected arity values for EWI are 2, 3, and 4. Values larger than 4 were not selected in order to prevent GP from creating large trees, and to consequently restrict the search space. EWI can be used with either the fixed or varying arity method. In the case where EWI is used with the fixed arity approach, intervals of equal width are created based on the set arity. In the case where EWI is used with the varying arity method, intervals of equal width are created based on the arity chosen by GP when creating each node. Since each attribute in a data set may have a different range of values, the intervals for each attribute node have the following structure: the lower bound of the first interval has to be the same value as the minimum value for the attribute. The upper bound of the last interval must be equal to the maximum value of the attribute.

An example of intervals created by the EWI approach is given in figure 6.2, the arity used in this example is 3, and the data used to create the intervals is presented in table 6.1. Each interval has an equal length of 2.0 units. The evaluation of the attribute is as follows:

- if $0 \leq attribute_data < 2.0$ then visit left branch
- if $2 \leq attribute_data < 4.0$ then visit left branch
- if $4 \leq attribute_data \leq 6.0$ then visit left branch

In the case of the fixed arity method, the GP trees are initialised to the fixed arity value, and the intervals are never altered during the GP algorithm. In the case of the varying arity method, the GP trees are initialised based on the arity chosen by GP, and the intervals are also never altered during the GP algorithm.

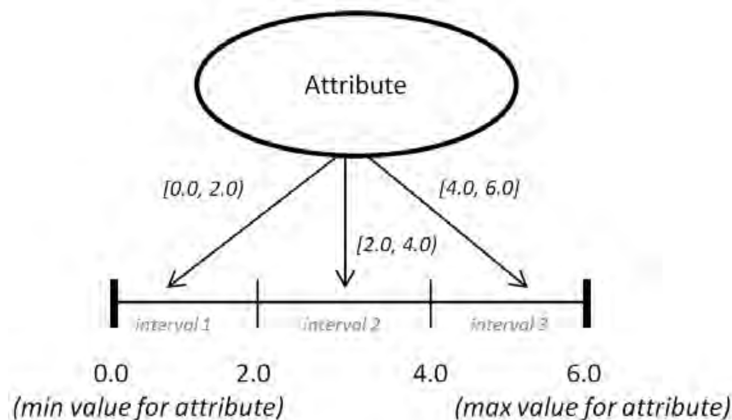


Figure 6.2: Intervals created using EWI.

In the following discussion the terms lower and upper bound are used. Consider the interval $[a, b]$, a represents the lower bound and b represents the upper bound. From the example in figure 6.2 the lower bound in interval 1 is closed, as well as the upper bound in interval 3. Between two adjacent intervals, say interval x and interval y in numerical order, the upper bound of interval x should be open, and the lower bound of interval y should be closed as can be seen between interval 1 and interval 2, and interval 2 and interval 3 in figure 6.2.

Attribute data
0.0
1.0
2.0
3.0
4.0
5.0
6.0

Table 6.1: Sample data for an attribute.

6.2.2 GP Evolved Intervals (GPEI)

The GPEI approach allows the GP algorithm to randomly create intervals based on the arity of each node. In this approach, the GP algorithm is allowed to create intervals of any size for a particular attribute that must adhere to the following:

- The lower bound of the first interval has to be the same value as the minimum value for the attribute.
- The upper bound of the last interval must be equal to the maximum value of the attribute.
- The intervals should be disjoint from each other.
- There should be no gap between intervals, i.e. the intervals are disjoint from each other but represent a continuous flow between the values of adjacent intervals and no discontinuity should exist between adjacent intervals.

Provided that the algorithm follows the above mentioned rules, it is free to create any cut-off point for an interval. These cut-off points are selected randomly during the execution of the GP algorithm when a node is being created, i.e. when a node is being added to the decision tree. Nodes are created during initial population generation, and when the mutation operator is applied. When a node is created, the intervals for that node are also created. Once a node has been created along with its intervals, these values can be changed using the alter interval GO which is described below.

An example of an interval created using GPEI is shown in figure 6.3.

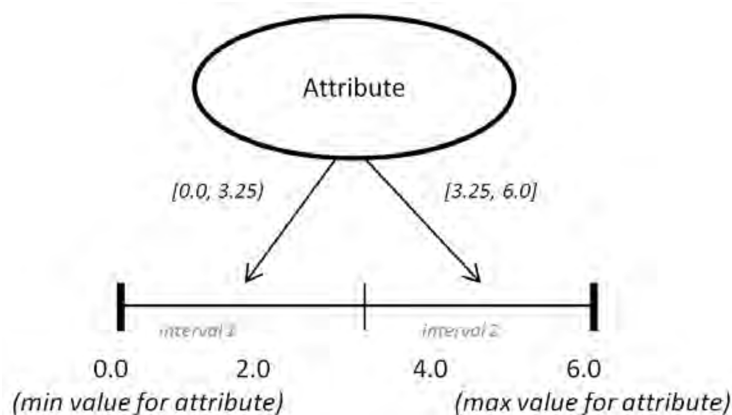


Figure 6.3: Intervals created using GPEI.

The pseudocode for the GPEI approach is presented in algorithm 6.1. Two general observations about this approach are made below:

- When the GP algorithm is executed several times, the intervals for nodes representing the same attribute may be different.
- If a decision tree contains several nodes for a particular attribute, the intervals may be different for each of them.

In step 2 of the GPEI pseudo-code the reason for selecting arity values between 2 and 4 is due to the fact that a value greater than 4 would render the GP program search space large and would affect the algorithm's ability to find a solution. The remainder of this section presents a new GP operator called alter interval. This GO was created in order to allow the GP algorithm to randomly select a node within a tree and to alter the interval in such a way as to improve the accuracy of that tree.

Since the method for creating the GPEI is random in nature it cannot guarantee that a new random interval will improve the accuracy. For this reason, hill climbing was incorporated in an attempt to improve the accuracy of the tree over ten attempts. At each attempt, a random interval is created and if the accuracy of the tree is improved, then the hill climbing halts and the tree is returned by the GO. If the accuracy is not improved, then another new random interval is created until all ten attempts are exhausted. Preliminary trial runs were performed and revealed that hill climbing improved the performance of the algorithm. The pseudocode for the alter interval GO is presented in algorithm 6.2. The hill climbing mechanism is executed in steps 9 to 12 in algorithm 6.2.

The alter interval GO only selects a single node and alters that node's interval. The alter interval GO is a local search operator since the structure of the tree is not

---

Algorithm 6.1: Pseudocode for creating an attribute node using GPEI.

---

input: input_tree
output: input_tree containing a new attribute node

1 **begin**
2 Randomly select an attribute within the function set. Create the attribute random_attr.
3 Allocate an arity value, arity, for random_attr.
4 *If the fixed arity method is used then set the arity based on the user parameter.*
5 *If the varying arity method is used then ranomly select an arity between 2 and 4 inclusively.*
6 Initialise empty intervals based on the arity value determined in step 2.
7 Set the leftmost interval to have a lower bound value equal to the minimum value for random_attr
8 Set current to the value obtained in step 7.
9 random = random real number between current and max value for random_attr. random cannot be equal to the max value for random_attr
10 Create an interval between current and random.
11 Set current to random.
12 Steps 9 to 11 created the first interval. Repeat these steps to create the remaining intervals
13 Set the rightmost interval to have an upper bound value equal to the maximum value for random_attr.
14 **end**

---

affected, and since the search is focused on a single node.

Algorithm 6.2: Alter interval genetic operator.

```
   input: input_tree
   output: A tree with a new random interval for an attribute node.
 1 begin
 2     parent ← TournamentSelection();
 3     parent_copy ← CreateCopy(parent);
 4     random_node ← RandomNodeFromTree(parent_copy);
 5     original_interval ← CopyInterval(random_node);
 6     if random_node ≠ terminal node then
 7         busy ← 0;
 8         initial_accuracy ← ComputeAccuracy(parent_copy);
 9         while new_accuracy ≤ initial_accuracy AND busy ≠ 10 do
10             random_node ← CreateRandomInterval (random_node);
11             new_accuracy ← ComputeAccuracy (parent_copy);
12             busy ← busy + 1;
13         end
14     end
15     if new_accuracy < initial_accuracy then
16         random_node ← SetInterval(original_interval);
17     end
18 end
```

Figures 6.4 and 6.5 illustrate a node which has been modified by the alter interval GO. From the figures, attribute 3 was selected, and it has an arity of 3. The lower bound of the first interval is not changed, neither is the upper bound of the last interval. Typically when a researcher makes use of GP to solve a problem the crossover and mutation operators are used. Since the crossover operator is a local search operator, a small percentage of its application rate can be allocated to the alter interval operator as it is also a local search operator, allowing the GP algorithm to make use of the three operators.

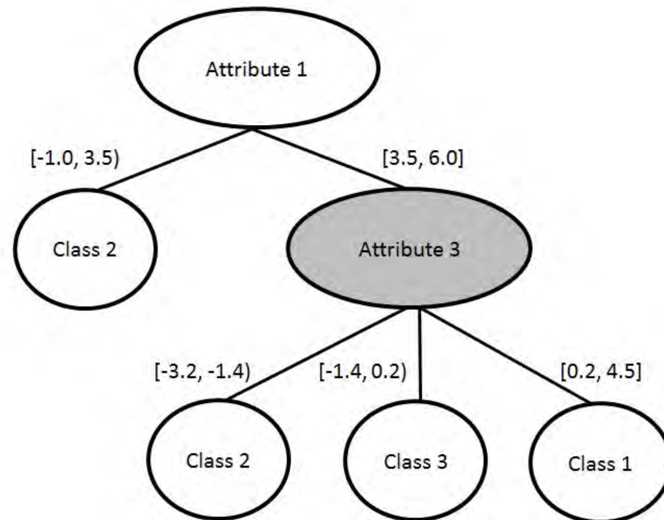


Figure 6.4: Illustrating the alter interval GO. The algorithm selected attribute 3 (highlighted in grey) for modification.

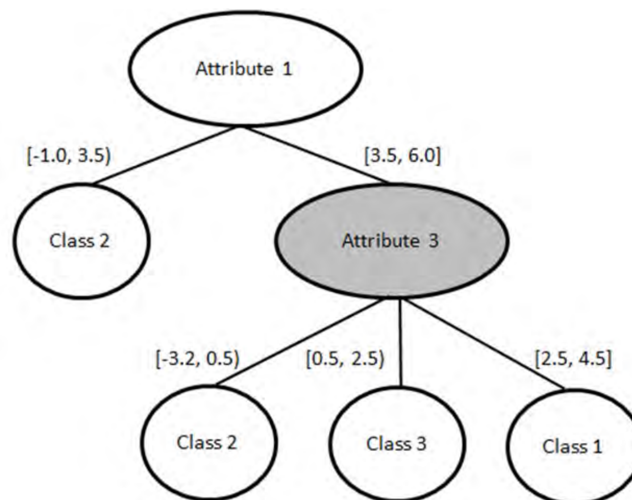


Figure 6.5: Illustrating the alter interval GO. The intervals for attribute 3 were altered which resulted in three new intervals.

6.3 Experimental Setup

Eight experiments were run, each consisting of a unique combination of methods in order to investigate the performance of the proposed adaptive discretisation methods. The proposed discretisation methods implemented the GP system described in section 5.5. The initial population generation, selection methods, fitness evaluation and GOs were implemented as described in that section. The results were obtained using the approach described in section 5.6. The experiments are listed along with

their combination of characteristics in table 6.2, and represent every possible combination between the GPEI and EWI methods, with the fixed and varying arity methods.

ID	Description	Arity	GPEI	Varying Arity
1	GPEI varying arity	n/a	true	true
2	GPEI arity 2	2	true	false
3	GPEI arity 3	3	true	false
4	GPEI arity 4	4	true	false
5	EWI arity 2	2	false	false
6	EWI arity 3	3	false	false
7	EWI arity 4	4	false	false
8	EWI varying arity	n/a	false	true

Table 6.2: Experiments conducted and their different combination of parameters.

Experiments 1 to 4 used the GPEI approach whereas experiments 5 to 8 used EWI. The column labelled “arity” denotes the arity used for that experiment, in the case where a number is present this means that a fixed arity was used, and in the case where “n/a” is present it means that the GP algorithm randomly selected the arity as described previously. The columns “GPEI” and “varying arity” denote whether or not GPEI and the varying arity approach was used. A value of “true” means it was applied, and “false” means it was not applied to that experiment. For instance, experiment ID 3 corresponds to an experiment in which the arity for all the GP trees is set to 3, and the GP algorithm alters the intervals during the execution.

6.3.1 Data sets

Twelve binary and multiclass data sets were used to examine the performance of the proposed methods; the data sets are listed in table 6.3. These data sets were selected as they represent binary and multiclass classification problems, and have varying characteristics. Further details about the data sets were presented in chapter 5.

Ecoli	Sonar
Fertility	Spectf
Glass	Vehicle
Ionosphere	WDBC
Iris	Wine
Pima Indians	Yeast

Table 6.3: Selected data sets for the adaptive discretisation experiments.

6.3.2 GP parameters

The GP parameters used throughout all the experiments in this chapter are listed in table 6.4. The crossover operator application rate varied depending on whether or not the alter interval operator was used. These parameters were determined empirically through trial runs.

GP Parameter	**Value**
Population Size	700
Parent Selection Method-arithmetic	Tournament selection, size 7
Initial Population Maximum Tree Size	7
Initial Population Generation Method	Ramped half and half
Maximum Offspring Size	7
Crossover Application Rate	If GPEI was not used: 70% If GPEI was used: 60%
Mutation Application Rate	30 %
Alter Interval Application Rate	If GPEI was not used: 0% If GPEI was used: 10%
Maximum Number of Generations	200
GP Model	Generational model
GP Tree Representation	Decision trees

Table 6.4: GP parameters.

6.4 Conclusion

This chapter investigates the incorporation of an adaptive discretisation mechanism into the GP algorithm for data classification. Two methods were examined. EWI creates intervals of equal width, whereas GPEI allows GP to evolve the intervals. A new GO was proposed in order to alter the intervals dynamically during the execution of the GP algorithm. Both of these methods could use either a fixed arity which was set in advance, or let GP select the arity each time a node is created. Discretisation has not previously been incorporated in the GP algorithm, and thus the proposed algorithms represent a novel approach.

Chapter 7

Comparing Genetic Programming Representations for Binary Classification

7.1 Introduction

The first decision to make when implementing a GP algorithm for data classification is to decide upon a GP representation. Chapter 4 discussed previous studies which made use of three principle representations, namely arithmetic trees, logical trees, and decision trees. In previous studies, researchers provided no justification for their choice of representation when using GP for data classification. This immediately leads to several questions. Why was a particular representation selected over another? Does a particular representation result in classifiers which obtain higher accuracies? These questions are the rationale behind the investigation proposed in this chapter.

This chapter compares the three major representations in the context of GP and binary data classification. The representations are described in section 7.2. This is followed by the experimental setup in section 7.3 which describes the function and terminal set which will be used for each representation, as well as the data sets used. Finally, this chapter is concluded in section 7.4.

7.2 GP Representations for Binary Classification

7.2.1 Arithmetic trees

Arithmetic trees represent mathematical expressions which can discriminate between classes. In the case of binary classification, one tree can discriminate between two

classes using a threshold value. The function set consists of mathematic operators such as $+, -, \times, /, log, tan, exp$. Leaf nodes represent attributes whereas the non-left nodes are the mathematical operators. A tree represents a single mathematical expression which outputs a single real valued number. The output is then compared to a threshold value. Assume a classification problem has classes "a" and "b". If the output is less than the threshold value, then the class value "a" is the resulting classification value for that particular tree. If the output is greater than or equal to the threshold value, then the class value "b" is the classification value. Thus, in this representation, the tree does not directly encode the class in the representation. Figure 7.1 illustrates an example of a GP arithmetic tree for data classification. In the example x_{20} represents attribute 20 within the data set, similarly x_4 and x_{15} represent attributes 4 and 15 respectively. There are two mathematical functions in this example, the addition and multiplication operators. The tree represents the following expression $(x_{20} \times x_4) + x_{15}$.

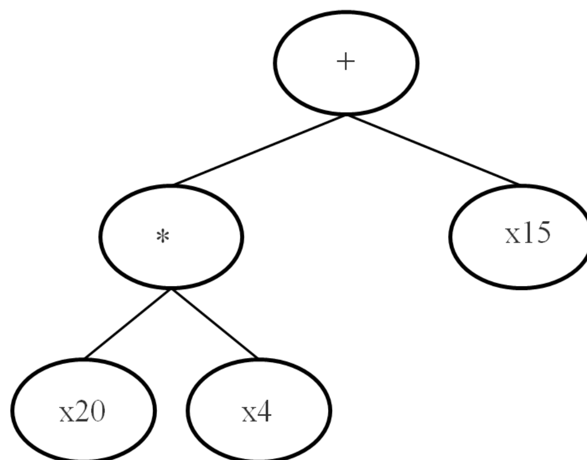


Figure 7.1: Arithmetic tree representation for GP.

7.2.2 Decision trees

Decision trees do not represent an expression like arithmetic trees. Decision trees represent a path from the root node to one leaf node. Each node within the tree represents one attribute and the leaf nodes represent one class. When a decision tree is traversed, a choice as to which branch will be visited next is determined by the attribute. Figure 7.2 illustrates a simple decision tree. The root node is the attribute *temperature*. Thus for any instance in a data set if the *temperature* value is "Hot", the left node is visited next, and if the *temperature* value is "Cold", then the right node is visited next. The classes are directly encoded in the tree. In figure 7.2, when the left branch is visited, a leaf node is reached and the classification output is "Class 1", and similarly if the right branch is visited, the classification output is

"Class 2". The GP decision trees will make use of GPEI discussed in chapter 6, and a fixed arity of 2 will be used.

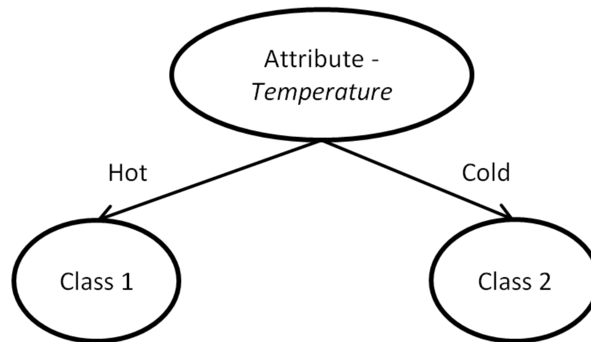


Figure 7.2: Decision tree representation for GP.

7.2.3 Logical trees

Logical trees can be evaluated using a similar threshold approach which is used with arithmetic representations. A logical tree outputs either *true* or *false* since it is made up of logical operators. Assume a classification problem has classes "a" and "b". The threshold approach for logical trees can be achieved as follows: if a tree outputs *true*, then the classification output is class "a", and if a tree outputs *false*, then the output is class "b". This allows a single tree to discriminate between two classes in a similar manner to arithmetic trees. Figure 7.3 illustrates an example of a logical tree. The tree first checks if x_{20} is less than x_4. Then the tree checks if x_8 is greater than x_{12}. The logical *OR* operator is applied to the result of the two comparisons.

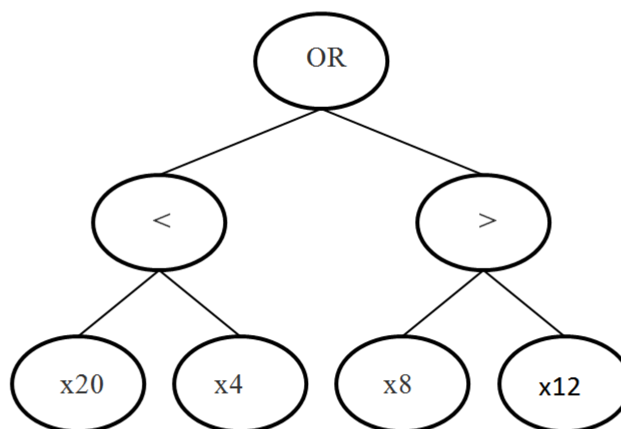


Figure 7.3: Logical tree representation for GP.

Typically inequality functions are used with this representation. A *between* func-

tion was proposed in order to allow an attribute to be compared to two values simultaneously. The *between* function takes three parameters of which the first parameter is always an attribute, and the other two parameters can be attributes or constants. The function is defined as follows:

$$\textit{between}\ (\text{x, y, z}) = \begin{array}{c} \textit{true}, \text{ if } y \leq x \leq z, \\ \textit{false}, \text{ otherwise.} \end{array}$$

Figure 7.4 illustrates an example of the between node.

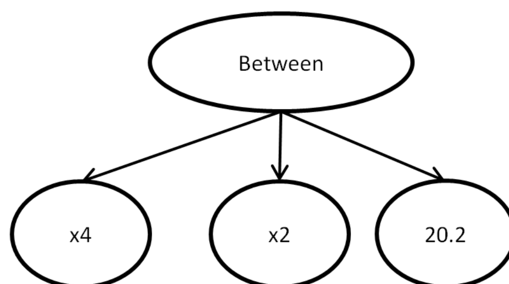


Figure 7.4: The between GP operator for logical tree representations.

Thus, from figure 7.4, the tree will return *true* if the value of x_4 is greater than x_2 and simultaneously smaller than 20.2. De Falco e*t al.* [9] used a similar function to the proposed *between* function in this study. De Falco e*t al.* proposed two functions, *IN* and *OUT*, which compare an attribute with two range values. These functions also take three parameters. The *OUT* function checks if an attribute is outside the range of the two parameters, and the *IN* function checks if an attribute in within the range of the two parameters. In this study, the *OUT* function can be achieved by preceding the between function with a *NOT* operator as illustrated in figure 7.5. De Falco *et al.* do not determine whether or not these additional functions are useful in evolving classifiers. A comparison of the performance of the proposed method without these two additional functions was performed.

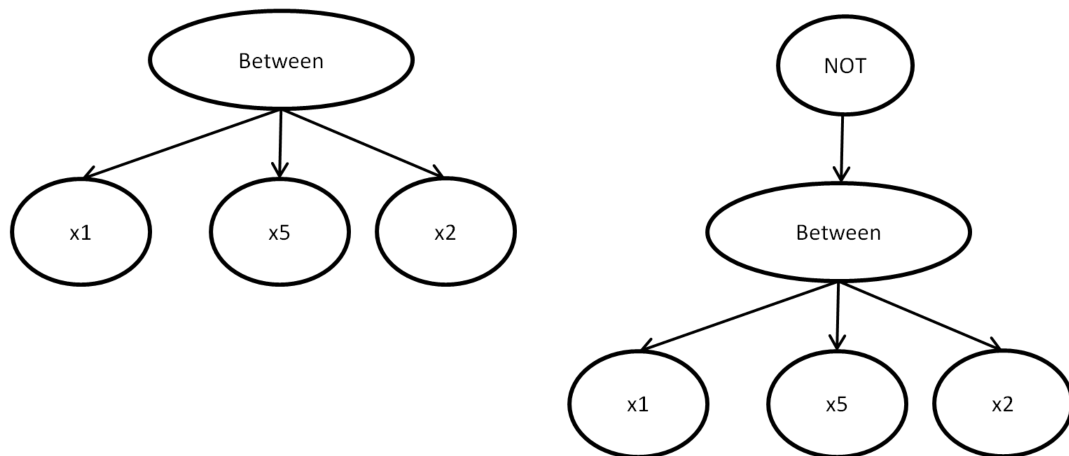


Figure 7.5: Creating the *OUT* function by preceding the between with a *NOT* operator.

7.3 Experimental Setup

Five experiments were carried out in order to compare the three different representations; but also to compare how certain elements of the function set impact the performance of the GP algorithm. Table 7.1 lists the five experiments along with their function and terminal sets. The *arithmetic-with-if* and *arithmetic-without-if* experiments were performed in order to determine the usefulness of the *if* statement. Similarly, *logical-with-between* and *logical-without-between* were conducted to investigate the difference in performance when the *between* function is and is not used. The representations implemented the GP system described in section 5.5. The initial population generation, selection methods, fitness evaluation and GOs were implemented as described in that section. The results were obtained using the approach described in section 5.6.

Representation	**Function Set**	**Terminal Set**
Arithmetic-with-if	$+, -, \times, /$	Attributes
Arithmetic-without-if	$+, -, \times, /, if, <, >$	Attributes
Decision tree	Attribute	Classes
Logical-with-between	$AND, OR, NOT, <, >$	Attributes and random constants
Logical-without-between	$AND, OR, NOT, Between, <, >$	Attributes and random constants

Table 7.1: Characteristics of the five experiments comparing different GP representations for binary data classification.

7.3.1 Data sets

Ten binary publicly available data sets were used to compare each of the GP representations for binary data classification; these are listed in table 7.2. These data sets were selected as they have varying characteristics in terms of number of instances, number of attributes and class balance. Additional details about the data sets were presented in chapter 5.

Climate	Parkinsons
Fertility	Pima Indians
Ionosphere	Spectf
Mammographic	Sonar
Monk2	WDBC

Table 7.2: Selected binary data sets for the GP representation experiments.

7.3.2 GP parameters

The GP parameters which were used throughout all the experiments in this chapter are listed in table 7.3. These parameters were determined empirically through trial runs.

GP Parameter	**Value**
Population Size	700
Parent Selection Method	Tournament selection, size 7
Initial Population Maximum Tree Size	7
Initial Population Generation Method	Ramped half and half
Maximum Offspring Size	7
Crossover Application Rate	70% (for arithmetic and logical representations) 60% (for decision trees)
GPEI Application Rate (for decision trees only)	10 %
Mutation Application Rate	30 %
Maximum Number of Generations	200
GP Model	Generational model

Table 7.3: GP Parameters for comparison of different GP representations for data classification.

7.4 Conclusion

This chapter proposed a comparison in the performance of three major GP representations for binary data classification. The three most commonly used representations are arithmetic trees, decision trees, and logical trees. Each representation uses a different function and terminal set. Researchers often select a representation without providing a rationale behind their choice. Furthermore, researchers who are new to the field of GP and data classification have no means of determining which GP representation to use, due to the fact that there has not previously been a comparison between the GP representations for binary data classification. Typically, researchers use similar function sets when using an arithmetic or logical tree representation, however, there are studies which have made use of additional functions such as the *if* and *between* function. This chapter proposed five experiments in order to compare decision trees, arithmetic trees, and logical trees. Furthermore, the experiments are proposed in order to determine the usefulness of the *if* and *between* functions.

Chapter 8

Incorporating GP Encapsulation Within Decision Trees for Data Classification

8.1 Introduction

This chapter serves as a an investigation of the encapsulation genetic operator when used with GP for data classification. Section 2.14.1 described and illustrated the encapsulation operator. There has been no previous attempt at investigating this GO in the context of data classification and thus serves as the rationale behind this study. The proposed encapsulation methods for data classification are discussed in section 8.2. Section 8.3 describes the experimental setup, and finally section 8.4 concludes this chapter.

8.2 Incorporating Encapsulation into GP for Data Classification

8.2.1 Decision trees and encapsulation

The encapsulation operator typically selects any element of the function set within a parse tree and encapsulates the subtree located at that position. In the context of decision trees, this is achieved by randomly selecting an attribute node. Encapsulating a node representing a class will be of no particular benefit. The pseudocode for the encapsulation operator in the context of data classification is presented in algorithm 8.1.

Incorporating the encapsulation operator for classification results in changes to

Algorithm 8.1: Pseudocode for encapsulation in the context of data classification.

```
1 begin
2     Select a random tree T within the current GP population.
3     Select a random attribute node within the tree.
4     Remove the subtree located at the random attribute node and store it in
      memory.
5     Allocate an encapsulated terminal name to the removed subtree. Initially
      named E0, then E1, and so on.
6     Add the encapsulated terminal to the terminal set.
7     Within T, replace the subtree with the encapsulated terminal.
8 end
```

steps 4 and 6 within the standard GP algorithm (refer to algorithm 2.1 in chapter 2). In this study the encapsulation operator is applied after every second generation. Applying the operator at each generation would result in a large number of encapsulated nodes which would consequently impact the performance of the algorithm. Additional trial runs were performed and revealed that applying the operator at every second generation led to improved results. The following changes are made to step 6 in the standard GP algorithm. For odd generations, only the crossover and mutation genetic operators are applied. For even generations, the encapsulation operator is applied in addition to crossover and mutation. The encapsulation operator is performed before crossover and mutation so that the latter two GOs can make use of the newly created encapsulated terminals.

The mutation operator selects a random tree and a random mutation point within that tree. It then creates an entirely new subtree at that point. Given the fact that the encapsulation operator is performed prior to mutation, the mutation operator can then add the new encapsulated terminals to the subtrees. Similarly, the crossover operator exchanges subtrees within two randomly selected parents, and if the resulting offspring are larger than a certain specified depth the trees are pruned. By pruning the trees, any function node at the maximum specified depth is replaced with a terminal node. Since the encapsulated terminals are added to the terminal set, it is thus possible that these terminals which are added during the pruning process can be one of the new terminals created by the encapsulation operator.

Figure 8.1 illustrates a tree which has been pruned. The tree on the left, which was produced by crossover or mutation has a depth of 4. If the maximum depth permitted is 3, all the leaves below the horizontal line have to be pruned. Removing all the nodes below the horizontal line results in an invalid tree because attributes cannot be leaves. Thus, the two attributes at depth 3 have to be replaced with ter-

minals. The tree on the right is the result of the pruning process. From the original tree, the left attribute at depth 3 was replaced with an encapsulated terminal, and the right attribute at depth 3 was replaced with a class.

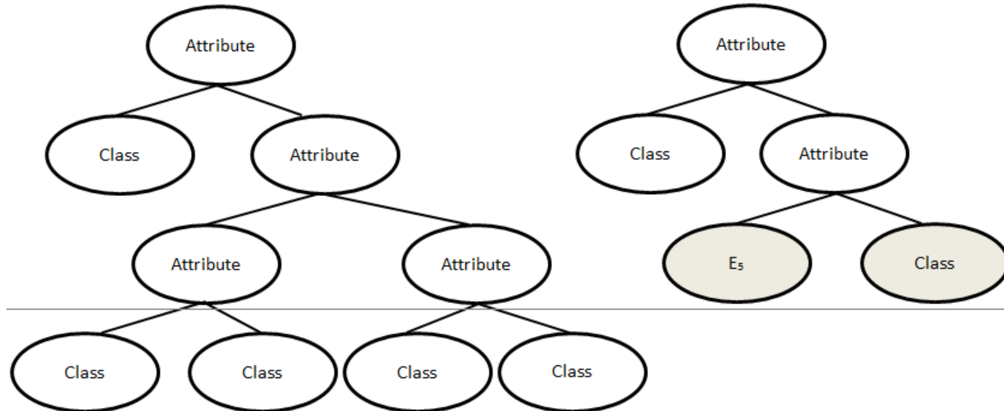


Figure 8.1: Pruning trees and adding encapsulated terminals at the leaves.

Step 4 (the evaluation phase in algorithm 2.1) in the standard GP algorithm is modified to cater for the encapsulated terminals. When a tree is evaluated, and if an encapsulated terminal is reached, the subtree corresponding to that terminal is evaluated. For example, if the following tree is evaluated: *temperature* E_0 *class2*, when the left branch is visited the algorithm will then proceed by evaluating the subtree corresponding to E_0.

It is possible that the algorithm encapsulates a subtree which already contains an encapsulated terminal. The evaluation process will evaluate all the encapsulated terminals recursively until a class is reached. For instance, some tree may have a reference to subtree E_5, and E_5 may in turn have a reference to subtree E_6.

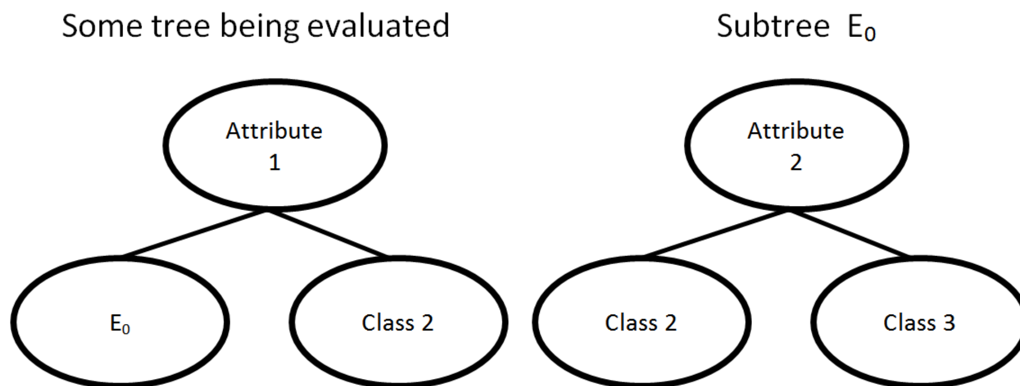


Figure 8.2: Evaluating a tree with an encapsulated terminal.

Assume the left tree in figure 8.2 is evaluated, and if the left branch is visited, the algorithm will then proceed by evaluating the corresponding subtree represented

by the terminal E_0. If the subtree E_0 evaluates to "class 2", then the classification output for the left tree is "class 2". An outline of the modified GP algorithm with encapsulation is presented in algorithm 8.2.

Algorithm 8.2: Pseudocode of proposed GP algorithm with encapsulation.

```
1  begin
2      Create initial population.
3      Evaluate the initial population.
4      generation ← 0.
5      while generation ¡= generation_max do
6          generation ← generation + 1.
7          Apply selection methods.
8          If odd generation, apply crossover and mutation using current
           terminal set.
9          If even generation, apply encapsulation, and add newly created
           terminals to the terminal set.
10         Evaluate the current population. If an encapsulated terminal is
           encountered, then call the corresponding subtree.
11     end
12     return Best solution found.
13 end
```

8.2.2 Maintaining the most called subtrees

The proposed algorithm of incorporating encapsulation to the GP algorithm presented in section 8.2.1 allows the GP algorithm to select from a large list of encapsulated terminals. When the number of encapsulated terminals is very large, this can hinder the performance of the GP algorithm as a large terminal set increases the GP program space. In order to overcome this problem, another method is proposed which maintains a list of encapsulated terminals, this new proposed method is named selective encapsulation. Details about this enhanced proposed method are discussed below.

The list of encapsulated terminals is initialised the first time the encapsulation genetic operator is called. The maintained list has one user defined parameter, *max*, which allows the researcher to control the maximum size of the list. The process of initialising the list is illustrated in algorithm 8.3. In section 8.2.1 all the encapsulated nodes are kept in memory and added to the terminal set; however, in this proposed method only certain elements are kept in the maintained list. In this chapter the term memory represents all the encapsulated terminals created throughout the GP run. Encapsulated terminals in the memory are never deleted. The maintained list contains certain encapsulated terminals from the memory. However unlike the memory, terminals in the maintained list can be removed. When the encapsulation

operator is applied, the newly created encapsulated terminals are automatically added to the memory, but not necessarily to the maintained list. The maintained list is updated by removing encapsulated terminals from the list, and adding new ones.

Algorithm 8.3: Pseudocode for initialising the maintained list.

```
1  begin
2      for Each encapsulated terminal E in the current population do
3          if there are less than max elements in the maintained list then
4              Insert the encapsulated terminal E into the maintained list.
5          end
6          else
7              Insert the encapsulated terminal E into memory but not into the
               maintained list.
8          end
9      end
10 end
```

Algorithm 8.4 presents the pseudocode for updating the maintained list. The list is updated after the encapsulation genetic operator is executed, in this chapter this is performed after every second GP generation.

For a given encapsulated terminal, the number of *calls* is determined by computing the number of times that encapsulated terminal is called within the entire current population. Assume E_0 is a terminal which is found in five trees in the current population, then E_0 has a total of five calls. When the list is updated, an encapsulated terminal which is not in the list, and has the most number of calls is selected. This process only deals with encapsulated terminals which are currently present in the population. For instance, if the terminal E_{44} had a large number of calls in the previous population but is not found in the current population, then E_{44} is not considered as a potential terminal to be selected. Thus, the process finds all the encapsulated terminals within the current population which are not in the maintained list, and determines which one has the highest number of calls. The next step (step 4 in algorithm 8.4) is to determine which encapsulated terminal within the list is called the least within the current population. Finally in step 5, the terminal which is called the least within the list is then replaced with the terminal outside the list which is called the most. Both of the encapsulated terminals remain within memory; however, the terminal which was previously in the maintained list is no longer in the list. Thus, after every second GP generation, one terminal leaves the list and another one enters the list.

After the initial population generation, terminals are only added to trees when the mutation operator is executed, or when trees are pruned. Algorithm 8.5 il-

Algorithm 8.4: Pseudocode for updating the maintained list.

```
1 begin
2     Determine the encapsulated terminal in memory which is not in the
      maintained list, and that has the most number of calls in the current
      population. Call this memTerm.
3     If no element in step 1 is found, then determine the encapsulated terminal
      in memory that has the most number of calls within the current
      population. Call this memTerm.
4     Determine the encapsulated terminal within the maintained list which has
      the least number of calls. Call this listTerm.
5     Swap memTerm with listTerm, i.e. memTerm is now part of the
      maintained list, and listTerm is no longer part of the maintained list,
      however listTerm is still in memory.
6 end
```

lustrates how terminals are selected and added to trees when the maintained list approach is incorporated into the GP algorithm.

If there are no encapsulated terminals in memory, then classes are used as terminals. When the selective encapsulation approach is used, there is a 60% probability that one of the encapsulated terminals from the list is selected. The value of 60% was chosen in such a way so as to slightly bias the algorithm towards selecting terminals from the list, and additionally a value of 60% does not completely bias the choice towards only selecting from the list. Thus, there is a higher probability that GP will select from a smaller range of encapsulated terminals which have been called frequently in the recent population. This contrasts from the initial approach described in section 8.2.1 whereby the GP algorithm can select any encapsulated terminals in memory. This proposed method results in two additional parameters, however, these parameters help further control the method by enabling the GP algorithm to select certain useful encapsulated nodes.

8.3 Experimental Setup

Three experiments were proposed in order to investigate the effects of including the encapsulation GO in GP for data classification problems. The first experiment was a baseline study which did not include the encapsulation operator, this was referred to as the GP encapsulation method. The second experiment, selective encapsulation, included the encapsulation operator; however, there was no restriction as to how GP could use the encapsulated terminals. The algorithm was thus able to select any encapsulated terminal. The third experiment was the selective encapsulation approach which included the encapsulation operator, and the maintained list with a maximum size of ten encapsulated terminals. The value of ten was determined

Algorithm 8.5: Pseudocode which selective encapsulation uses for adding terminals to the GP trees.

```
1  begin
2      if there are no encapsulated terminals then
3          Return a random class.
4      end
5      else
6          Randomly decide whether to select a class or an encapsulated
           terminal.
7          if select a random class then
8              Return a random class.
9          end
10         else
11             One of the two events will occur:
12             With 60% probability, return a random encapsulated terminal
               from the maintained list.
13             With 40% probability, return a random encapsulated terminal
               from memory.
14         end
15     end
16 end
```

through additional trial runs.

Other than algorithms described in this chapter, the overall GP system remained unchanged, and the GP system for the proposed methods was implemented as described in section 5.5. The initial population generation, selection methods, fitness evaluation and GOs were implemented as described in that section. The results were obtained using the approach described in section 5.6. GP decision trees was the selected representation for the proposed methods, namely GPEI with arity 2, and this approach was defined in chapter 6.

8.3.1 Data sets

Twelve publicly available data sets were used to investigate the proposed encapsulation GO, these are presented in table 8.1. These data sets were selected as they represent binary and multiclass classification problems, and have varying characteristics. Additional details about the data sets were presented in chapter 5.

8.3.2 GP parameters

The GP parameters which were used throughout all the experiments in this chapter are presented in table 8.2. These parameters were determined empirically through trial runs.

Climate	Pima Indians
Ecoli	Sonar
Fertility	Spectf
Glass	WDBC
Ionosphere	Wine
Iris	Yeast

Table 8.1: Data sets used for GP encapsulation experiments.

GP Parameter	**Value**
Population Size	700
Parent Selection Method	Tournament selection of size 7
Maximum Initial Population Tree Size	7
Maximum Offspring Size	4
Initial Population Generation Method	Ramped half and half
Crossover Rate	60%
Mutation Rate	30%
Alter Interval Rate	10%
Encapsulation Rate	1% every 2nd generation
Maximum Number of Generations	200
GP Model	Generational model
Function Set	Attributes
Terminal Set	Classes, and encapsulated terminals

Table 8.2: GP parameters used.

8.4 Conclusion

This chapter proposes a novel investigation on the effects of the encapsulation GO for GP in the context of data classification. Two approaches were proposed in order to make use of the encapsulation operator. The first makes use of the encapsulation operator without any restrictions on how the GP algorithm can use the encapsulated terminals. GP is able to select any encapsulated terminal without being biased towards which ones to select. The second approach, selective encapsulation, makes use of a maintained list of encapsulated terminals, and the GP algorithm selects terminals from the maintained list with a 60% probability. The goal behind the second approach is to maintain a list of encapsulated terminals which are frequently used within the GP population, and thus bias the algorithm to select those encapsulated terminals from the list. The proposed methods will be applied to twelve publicly available data sets.

Chapter 9

Hybridising Evolutionary Algorithms for Creating Classifier Ensembles

9.1 Introduction

Instead of creating a single classifier which is responsible for classifying an entire data set, one can also make use of an ensemble. Ensembles were discussed in section 3.7.3, and GP ensembles were discussed in section 4.6.

In this chapter, a genetic algorithm (GA) [52] is hybridised with a GP algorithm in order to create classifier ensembles. This research is aimed at investigating different hybridisation approaches to combine a GP with a GA. Four different approaches are proposed in order to assess the effectiveness of the hybridisation of these two evolutionary algorithms. The proposed ensemble methods provide an alternative to conventional boosting methods which are typically incorporated into the GP algorithm for ensemble construction. The four proposed methods are described in section 9.2. Section 9.3 describes the experimental setup, and finally, section 9.4 concludes this chapter.

9.2 Proposed Hybridisation of GP and GA

Four different approaches were examined in order to determine the effects of hybridising the GA with the GP algorithm. The pseudocode for the GA algorithm used in this study is presented in algorithm 9.1. The two GA genetic operators which will be used are presented in algorithms 9.2 and 9.3 , representing mutation and one point crossover respectively.

Algorithm 9.1: Pseudocode of genetic algorithm for ensemble representation.

```
   input: generation_max: maximum number of GA generations
 1 begin
 2     Create an initial population of chromosomes repsented as ensembles.
 3     Evaluate the initial population.
 4     generation ← 0.
 5     while generation ≤ generation_max do
 6         generation ← generation + 1.
 7         Select parents.
 8         Perform mutation.
 9         Perform one point crossover.
10         Replace current population with offspring.
11     end
12 end
```

Algorithm 9.2: Pseudocode for GA mutation.

```
   input  : probability: probability of which a gene within a chromosome is
            mutated
   input  : parent_chromosome: parent chromosome obtained using tournament
            selection
   input  : length: length of parent
   input  : final_GP_population: the final population of GP trees
   output: A child chromosome which has been mutated.
 1 begin
 2     Create an empty chromosome (child) with length equal to length.
 3     for i ← 0 to length do
 4         if GenerateRandomDouble ¡ probability then
 5             random_GP_tree ← GetRandomGPTreeFrom(final_GP_population)
 6             AddTreeToPosition (random_GP_tree, i)
 7         end
 8         else
 9             AddTreeToPosition (GetGeneFromParentChromosome (i), i)
10         end
11     end
12     return child
13 end
```

Algorithm 9.3: Pseudocode for GA one point crossover.

```
   input: probability: probability of which the crossover operator will be applied
   input: parent1_chromosome: parent chromosome obtained using tournament
          selection
   input: parent2_chromosome: parent chromosome obtained using tournament
          selection
   input: length: length of parent chromosomes
   output: A child chromosome.
 1 begin
 2     Create an empty chromosome (child1) with length equal to length.
 3     Create an empty chromosome (child2) with length equal to length.
 4     if GenerateRandomPosition ¡ probability then
 5         crossover_point ← GenerateRandomPosition(length)
 6         for i ← 0 to length do
 7             if i < crossover_point then
 8                 In child1, set the gene at position i, with the gene at position i
                   in parent1_chromosome
 9                 In child2, set the gene at position i, with the gene at position i
                   in parent2_chromosome
10             end
11             else
12                 In child1, set the gene at position i, with the gene at position i
                   in parent2_chromosome
13                 In child2, set the gene at position i, with the gene at position i
                   in parent1_chromosome
14             end
15         end
16     end
17     return The child with the highest trainning accuracy.
18 end
```

9.2.1 GA encoding

In this study, a chromosome represents an ensemble. Each gene corresponds to a GP decision tree which represent classifiers on its own.. Figure 9.1 illustrates an example of a chromosome. From the figure, part *(1)* illustrates the GA representation with a chromosome length of three. The correspondence of each gene to a GP decision tree is illustrated in part *(2)*.

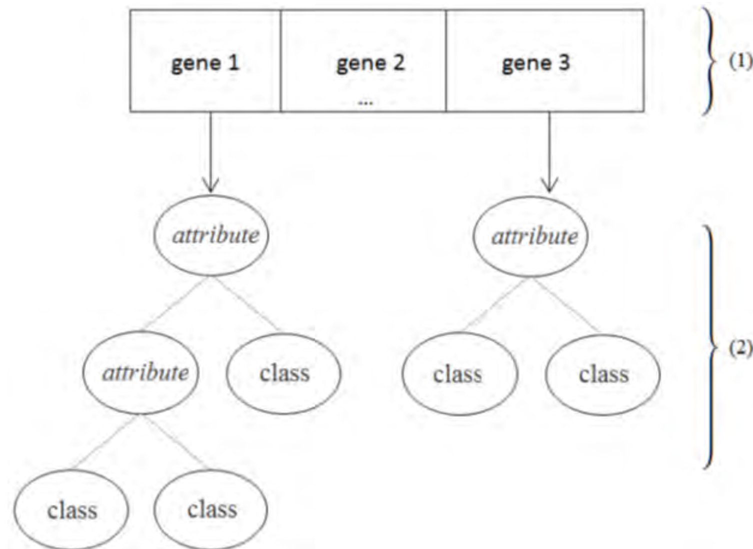


Figure 9.1: Illustrating an ensemble.

In previous studies, a weighted approach is often used, whereby certain members of the ensemble have a greater impact on the final output of the chromosome due to their weight. In this study, however, each tree in the ensemble has an equal vote. A chromosome is evaluated on a particular instance of data as follows. Each tree within the chromosome is evaluated on the instance of data, and the statistical mode of all the individual tree outputs is computed. For example, if the outputs for some chromosome of length 3 on an instance of data are $\{class1, class2, class1\}$, then the output of the chromosome is "*class1*" because that is the statistical mode of the tree outputs. Chromosomes in this study have an odd length in order to reduce the complexity of dealing with clashes. A chromosome is evaluated on each of the instances of data, and the final accuracy of the chromosome is computed as the total number of instances which it correctly classifies.

9.2.2 GA run after the last GP generation (GA-at-end)

The first approach, *GA-at-end*, represents the initial attempt at hybridising the GA with the GP algorithm. In this approach the GA is only executed once the GP algorithm has completed (after step 8 in chapter 2, algorithm 2.1). The GA makes use of the final GP population, and is initialised by randomly creating ensembles in which the genes correspond to GP trees from the last GP generation. The GA is then run for several GA generations, and the best ensemble is output. Several different ensemble sizes will be compared (namely 3, 5, 7, 9 and 11) in order to determine which ensemble size is the most successful. When a particular ensemble size is used, the chromosomes will not change size during the execution of the hybridised algorithm. Thus, when *GA-at-end* is run with an ensemble size of 7, the size of

each chromosome is initialised to 7, and additionally the chromosomes maintain a size of 7 during the GA run. Mutation and one point crossover will be used for this approach. The pseudocode for *GA-at-end* is presented in algorithm 9.4.

Algorithm 9.4: Pseudocode for *GA-at-end*.

```
  input: Ensemble_size
  output: An ensemble represented by a GA chromosome
1 begin
2   Perform a run of the GP algorithm.
3   Initialise the GA chromosomes with size equal to ensemble_size.
4   Randomly select trees from the final GP population and add the trees to
    the initialised chromosomes.
5   Perform a run of the GA algorithm.
6   Output the best chromosome found during the GA algorithm.
7 end
```

9.2.3 GA run after each GP generation (GA-after-each-gen)

In *GA-at-end*, the GA optimisation is only performed after the final GP generation. It is possible that an ensemble with higher accuracy can be obtained using GP trees from any generation and not specifically the last GP generation. In this second hybridisation approach, the GA ensemble optimisation is performed after each GP generation; this approach is consequently named *GA-after-each-gen*.

In *GA-after-each-gen* the GA is initialised after step 7 in algorithm 2.1 (chapter 2). Upon initialising the GA, the algorithm is free to select any chromosome size from the set $\{3, 5, 7, 9, 11\}$. Once a size is chosen, all the chromosomes for that specific GA run are to be initialised to that particular size. The chromosome sizes are not altered during a GA execution. For instance, if the algorithm selects a size of 5, then all the chromosomes are initialised to that size, and the chromosomes maintain a size of 5 during the GA optimisation. After the next GP generation, the GA is reinitialised, and once again free to select any size. Similarly to *GA-at-end*, *GA-after-each-gen* makes use of mutation and one point crossover.

Since the GA is executed afresh after each GP generation, the genes are made up of GP trees from the current GP generation. The GA is run for a certain number of GA generations, and the best chromosome for each GA run is stored in memory. Once the GP algorithm has completed, the chromosome with the highest training classification accuracy found in memory is output as the final classifier . The pseudocode for *GA-after-each-gen* is presented in algorithm 9.5.

Algorithm 9.5: Pseudocode for *GA-after-each-gen*.

output: An ensemble represented by a GA chromosome

1 **begin**
2 $\mathsf{i} \leftarrow 0$
3 Perform GP generation i.
4 Randomly select an esemble size from {3, 5, 7, 9, 11}.
5 Initialise the chromosomes with the size selected in step 3.
6 Randomly select trees from GP generation i and add the trees to the initialised chromosomes.
7 Perform the GA algorithm.
8 Store the best chromosome (found in step 6) in memory.
9 $\mathsf{i} \leftarrow \mathsf{i} + 1$
10 Repeat steps 3 to 9 until i generations are completed.
11 Output the chromosome in memory which has the highest training accuracy.
12 **end**

9.2.4 GA with hill climbing (GA-with-HC)

An extension of *GA-after-each-gen* is proposed whereby hill climbing is incorporated into the GA's recombination operator (one point crossover); this approach is named *GA-with-HC*. The recombination operator is applied to two GA parent chromosomes, and two offspring are consequently created. In *GA-with-HC*, the recombination operator is repeated five times, and if during these five attempts, one of the two offspring has a higher classification accuracy than both of the parents, then that offspring is returned. However, if none of the offspring result in a higher classification accuracy after the five attempts, then the best of two parents is returned. Similarly to *GA-after-each-gen*, the GA in *GA-with-HC* is initialised and run after each GP generation.

Before hill climbing is performed, a copy of the parent chromosomes is made, and thus at each attempt, the offspring chromosomes are generated from the original parents. Hill climbing is incorporated in order to determine if it will impact the performance of the overall method. Trial runs were performed in order to determine the optimal number of times that hill climbing should be applied, and the most optimal value was found to be five.

9.2.5 Steady state GA (SSGA-GP)

In this approach, a steady state GA (SSGA) model will be used to create ensembles which contain GP trees from different GP generations; this approach is named *SSGA-GP*.

In the previously proposed approaches in this study, the chromosomes are made

up of GP trees from one particular GP generation. In *GA-at-end*, the chromosomes are made up of GP trees from the final GP generation. In *GA-after-each-gen* and *GA-with-HC*, the chromosomes are made up of GP trees from each GP generation respectively. For instance, in GP generation 4, the chromosomes are made up of GP trees strictly from the GP population in generation 4. Thus, *SSGA-GP* proposes an alternative approach where the chromosomes can be made up of GP trees from several different GP generations.

For this hybridisation of GP with GA, two separate populations are maintained and coevolved simultaneously. The GP population is responsible for driving the evolution of the GP trees. The GP algorithm implemented corresponds to the standard GP algorithm, and the GP crossover and mutation operators are performed in the conventional way. The SSGA population is maintained separately. The SSGA population maintains the chromosome ensembles in which each gene corresponds to GP trees from different GP generations. It is possible that a chromosome may contain several GP trees from the same GP generation. The SSGA population remains fixed in terms of the number of chromosomes within the population.

Figure 9.2 illustrates an example of a *SSGA-GP* chromosome from the GA population, whereby the genes correspond to GP trees from different GP generations. In this example, *gene 1* corresponds to a tree from GP generation 13; *gene 2* corresponds to a tree from GP generation 4, and so on. The initial SSGA population is created after step 2 in algorithm 2.1 (chapter 2) with genes corresponding to randomly selected GP trees from the initial GP population. Since the chromosomes represent ensemble classifiers, the chromosomes are evaluated by determining the class for each GP tree in the chromosome on an instance of data, and computing the statistical mode of the class outputs. The statistical mode represents the final classification for a chromosome on an instance of data, and the classification accuracy is then used to determine the fitness of the chromosome.

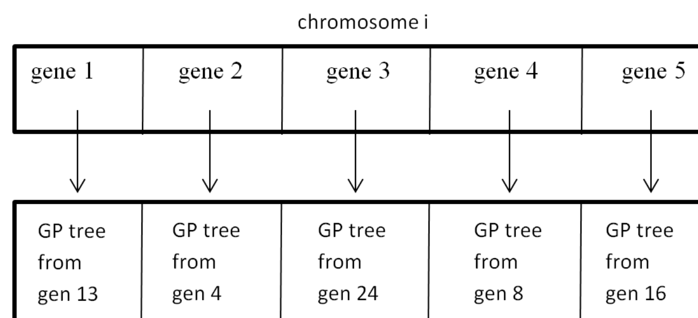


Figure 9.2: Example of a chromosome created using *SSGA-GP*. The genes correspond to GP trees which have been added from different GP generations.

In order to add a GP tree from the GP population to the *SSGA-GP* chromosomes, a variation of the GA mutation genetic operator is proposed, and is named the modified mutation operator. The pseudocode for the modified mutation operator is presented in algorithm 9.6. This operator allows GP trees from different GP generations to be added to the chromosomes.

Algorithm 9.6: Modified mutation GA operator.

```
1  - arithmetic
   input: GP and GA population
   output: GA chromosome offspring
2  begin
3      parent ← TournamentSelection(current_GA_population);
4      parent_fitness ← EvaluateEnsemble(parent);
5      attempt ← 0;
6      repeat
7          attempt ← attempt + 1;
8          copy ← CopyChromosome (parent);
9          random_position ← RandomInteger (copy_length);
10         random_tree ← GetRandomTree (GP_Population);
11         copy ← ReplaceTreeAtPosition (random_position, random_tree);
12         new_fitness ← EvaluateEnsemble(copy);
13         if new_fitness > parent_fitness then
14             return copy
15         end
16     until attempt ≠ 10;
17     return parent
18 end
```

The pseudocode for *SSGA-GP* is presented in algorithm 9.7. The modified mutation operator implements hill climbing by attempting to improve the classification accuracy of the offspring ten times. For each of the ten attempts, a copy of the original parent chromosome is made. Thus, once this operator is performed, if the accuracy of the chromosome is improved, the resulting offspring will contain one tree from the current GP population.

Since *SSGA-GP* makes use of a SSGA, a replacement strategy must be defined. The replacement strategy implemented is the inverse tournament selection method. This is performed in a similar manner to the tournament selection, however in the inverse tournament selection the chromosome with the lowest ensemble accuracy is returned.

Algorithm 9.7: Pseudocode for *SSGA-GP*.

input: ensemble_size
input: num_gen: total number of GP generations to perform before applying the modified mutation operator
input: num_offspring: total number of offspring to create using modified mutation operator
input: num_replace: total number of chromosomes to replace using inverse tournament selection
output: An ensemble represented by a GA chromosome

1 **begin**
2 Create the initial GP population.
3 Initialise the GA chromosomes (with size = ensemble_size) by randomly selecting trees from the initial GP population.
4 Perform num_gen number of GP generations.
5 Perform the modified mutation GA operator on the GA population and create num_offspring offspring.
6 Perform the inverse tournament selection and repalce num_replace chromosomes in the GA population with the offspring created in step 5.
7 Evaluate all the GA chromosomes and store the chromosome with the highest trainning accuracy in memory.
8 Repeat steps 4 to 7 until the maximum number of GP generations has been met.
9 Output the chromosome which obtained the highest trainning accuracy.
10 **end**

After each GP generation, the modified mutation operator is executed. This operator generates offspring chromosomes which may contain GP trees from the current GP population, and then the offspring replace the weaker chromosomes within the SSGA population. The number of offspring to replace is a user defined parameter.

In terms of GOs, the crossover and mutation operators are responsible for optimizing the GP population of GP classifiers, whereas the modified mutation GA operator is responsible for optimizing the SSGA population of GA chromosome ensembles. *SSGA-GP* does not make use of GA crossover or the conventional GA mutation; only the modified mutation operator is applied in order to investigate the effectiveness of this operator.

Trial runs were performed, and it was determined that for *SSGA-GP* the optimal chromosome ensemble size is 7. Thus, *SSGA-GP* initialises the chromosomes to a size of 7 and the modified mutation operator replaces a single gene within a chromosome,

which consequently ensures that the chromosomes remain with a size of 7.

9.3 Experimental Setup

Four experiments were performed in order to examine how effective the GA would be at increasing the classification accuracy when incorporated to the GP algorithm. *GA-at-end* performed a GA run at the end of the GP run, *GA-after-each-gen* performed a GA run after each GP run, *GA-with-HC* was similar to *GA-after-each* however used hill climbing in the GA recombination operator, and finally SSGA-GP coevolved a GP and a GA population. For the proposed hybridisations, the initial GP population generation, GP selection methods, GP fitness evaluation and GP GOs were implemented as described in section 5.5. The results were obtained using the approach described in section 5.6. GP decision trees was the selected representation for the proposed methods. For data sets made up of numerical attributes, GPEI with arity 2 was used, this approach was defined in chapter 6.

9.3.1 Data sets

Twelve publicly available data sets were used to examine the performance of the proposed methods. These data sets were selected as they represent binary and multiclass classification problems, and have varying characteristics. The selected data sets are listed in table 9.1. Further details about the data sets were presented in chapter 5.

Balance	Parkinsons
Climate	Pima Indians
Ecoli	Sonar
Fertility	Soybean
Ionosphere	Spectf
Iris	WDBC

Table 9.1: Selected data sets for the hybridisation experiments.

9.3.2 GP and GA parameters

Table 9.2 presents the GP and GA parameters for the proposed experiments in this chapter. These parameters were determined empirically through trial runs.

Parameter	**Value**
GP Population size	700
GP Parent Selection Method	Tournament selection of size 7
GP Initial Population Maximum Tree Size	7
GP Initial Population Generation Method	Ramped half and half
Maximum GP Offspring Size	7
GP Crossover Rate	70%
GP Mutation Rate	30%
Maximum Number of GP Generations	700
GP Model	Generational model
GP Function Set	Attributes
GP Terminal Set	Classes
GA Population size	1000
GA Parent Selection Method	*GA-at-end*, *GA-after-each* and *GA-with-HC*: Tournament selection of size 7 *SSGA-GP*: Inverse tournament selection of size 7
GA Initial Population Generation Method	Randomly select GP trees based on hybrid method
GA Recombination Rate	*GA-at-end*, *GA-after-each* and *GA-with-HC*: 50% *SSGA-GP*: 0%
GA Mutation Rate	*GA-at-end*, *GA-after-each* and *GA-with-HC*: 30% *SSGA-GP*: 100% (Modified mutation)
Number of individuals to replace from the SSGA population in *SSGA-GP*	30
Maximum Number of GA Generations	*GA-at-end*: 200 *GA-after-each* and *GA-with-HC*: 20

Table 9.2: GP and GA parameters for the hybridisation experiments.

9.4 Conclusion

This chapter proposes four methods which hybridise GA and GP in order to evolve a population of ensembles. The first method executes the GA at the end of the GP run. The second executes the GA after each GP generation, the third proposed method is an extension of the second method and incorporates hill climbing to the GA recombination operator. The last proposed method investigates the hybridisation of steady state GA model and GP. The proposed methods will be tested on 12 publicly available data sets.

Chapter 10

Ensemble Construction for Data Classification using Genetic Programming

10.1 Introduction

This chapter presents an ensemble construction method which creates a single GP ensemble. This approach differs to the approach described in chapter 9 since in this algorithm GP only creates a single ensemble as opposed to evolving a population of ensembles.

The ensemble construction method is introduced in section 10.2. Details on how a GP tree is selected to be added into the ensemble is discussed in section 10.2.1. A description of how the ensemble is evaluated is presented in sections 10.2.2 and 10.2.3. Each instance is allocated a weight, section 10.2.4 describes how the weights are updated. The experimental setup is presented in section 10.3. Finally, section 10.4 concludes this chapter.

10.2 Proposed Ensemble Construction

This section describes the proposed ensemble construction method. The ensemble is a list of GP classifier trees which vote in order to classify instances of data within a data set. This proposed ensemble construction deals with creating one ensemble during one GP run. A tree is added to the ensemble after a certain number of GP generations. At the end of the GP run, the ensemble is output and evaluated on the test set. This section describes how the ensemble is represented, and how a tree is added to the ensemble. Furthermore, this section describes how weights are used

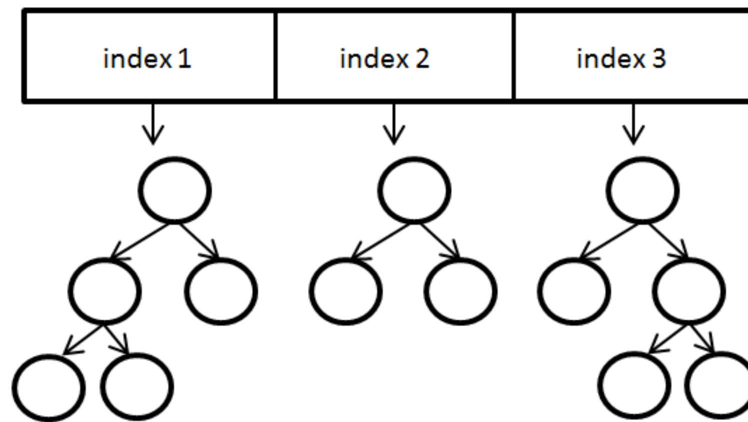


Figure 10.1: Ensemble with corresponding trees at each index.

to train the GP individuals, and how these weights are updated. Additionally, this section provides a discussion on how the GP trees and the ensemble are evaluated. Figure 10.1 illustrates an example of an ensemble where the ensemble size is three. At each index the corresponding GP tree is illustrated. Each tree represents a classifier. In this study two different GP tree representations are used on different data sets. Arithmetic trees are used when the data set contains numerical attributes, and decision trees are used when the data set contains nominal text and discrete integer values.

10.2.1 Selecting a tree to add to the ensemble

Initially, the ensemble is empty and trees are added to the ensemble after a certain number of generations. A user defined parameter, addFrequency, determines after how many GP generations a new tree is added to the ensemble. The pseudocode for adding a tree to the ensemble is illustrated in algorithm 10.1. Before selecting a tree to add to the ensemble, the current fitness of the ensemble is computed as this current fitness will be compared to the fitness of the ensemble after a new tree is added.

When a tree is to be added to the ensemble the tournament selection method is performed on the current GP population. The tree which is selected as a result of tournament selection is then added to the ensemble and the new fitness of the ensemble is computed. This fitness is then compared to the ensemble's previous fitness. In the case where the ensemble was previously empty, the tree which results in the highest ensemble accuracy is simply added to the ensemble.

During each iteration of this algorithm, a single candidate tree is temporarily added to the ensemble in order to compute the new fitness. If the new fitness is greater than the current fitness, then a reference to the candidate tree is stored as

a best candidate. The algorithm is iterated 20 times, and if a best candidate is found, then that candidate is permanently added to the ensemble. If there is no best candidate tree then the original ensemble is returned. The weights are then updated regardless of whether or not a tree has been added into the ensemble. This is further discussed in section 10.2.3.

Algorithm 10.1: Pseudocode for adding a tree to the ensemble.

```
    input: ensemble
    output: ensemble with an additional tree if a suitable candidate is found. If
            no such tree is found, then the original ensemble is returned
 1  begin
 2      tries ← 0;
 3      max_tries ← 20;
 4      best_candidate ← null;
 5      current_fitness ← EvaluateEnsemble(ensemble);
 6      repeat
 7          tries ← tries + 1;
 8          candidate_tree ← TournamentSelection(20);
 9          ensemble ← AddCandidateIntoEnsemble(candidate_tree);
10          new_fitness ← EvaluateEnsemble(ensemble);
11          if new_fitness > current_fitness then
12              best_candidate ← candidate_tree;
13              current_fitness ← new_fitness;
14          end
15          ensemble ← RemoveCandidateFromEnsemble(candidate_tree);
16      until tries < max_tries;
17      if best_candidate ≠ null then
18          ensemble ← AddCandidateIntoEnsemble(best_candidate);
19      else
20          ensemble ← AddCandidateIntoEnsemble(candidate_tree);
21      end
22  end
```

10.2.2 Ensemble evaluation

An ensemble is evaluated on a particular instance of data as follows. Each tree within the ensemble is evaluated on the instance of data, and the statistical mode of all the individual tree outputs is computed. For example, if the outputs for some ensemble of length 3 on an instance of data are $\{class1, class1, class2\}$, then the output of the ensemble is "*class1*" because that is the statistical mode of the tree outputs. Let some instance of training data be labelled with class "B", thus if the output of an ensemble of length 3 is $\{B, A, B\}$, then the ensemble correctly classifies the instance of data since the statistical mode is "B". However, if the output of an

ensemble of length 3 is $\{A, A, A\}$, then the ensemble misclassifies the instance of data. Ensembles in this study have an odd length in order to reduce the complexity of dealing with clashes. An ensemble is evaluated on each instance of data, and the final accuracy of the ensemble is computed as the total number of instances which it correctly classifies.

10.2.3 Evaluating the GP trees using weights

In this study a weight is allocated to each instance of data in the training set. The weights represent how easy or difficult an instance of data is for the ensemble to classify. A negative weight implies that the ensemble is not able to correctly classify a particular instance of data. A weight of zero implies that there is a clash amongst the individuals within the ensemble, for example if the output of the ensemble for a particular instance is $\{class1, class2\}$ then there is a clash. A positive weight implies that the ensemble is able to correctly classify a particular instance of data. Initially the weights are all set to a value of 0.

The magnitude of the weight denotes how well or how badly the ensemble can classify an instance of data. For instance, a weight of "5" implies that the ensemble is better at classifying an instance of weight "1". A weight of "-5" denotes an instance which is much harder to classify than one which has a weight of "-1". Section 10.2.4 describes how the weights are computed.

GP trees are evaluated according to the whether or not it correctly classifies an instance of data. For a GP tree, if an instance of data is correctly classified, then the weight of that particular instance is considered for the tree's fitness calculation, otherwise it does not contribute to the fitness of the tree. The evaluation of GP trees are computed as follows:

$$\sum_{i=1}^{n} f(x_i)$$

where i represents instance i, and n represents the number of instances in the training set. The function $f(x_i)$ is defined as follows:

$$f(x_i) = \begin{cases} 0 & if\ GP\ tree\ incorrectly\ evaluated\ instance\ i \\ g(x_i) & if\ GP\ tree\ correctly\ evaluated\ instance\ i \end{cases}$$

The function $g(x_i)$ has to be constructed in such a way that it caters for three cases. The cases are defined as follows.

- First case: the weight is negative.
- Second case: the weight is equal to 0.

- Third case: the weight is positive.

Negative weights have a greater impact on the fitness. A greater fitness implies that a GP tree is a better classifier. Since a negative weight represents an instance of data which is difficult to classify, a GP tree gains a greater fitness when it correctly classifies one of the difficult instances. The more challenging an instance is, the greater its impact on the fitness. This consequently results in the GP algorithm focusing on classifying the difficult instances whilst still taking into account the instances which are considered easy.

In order to justify the choice of the function $g(x_i)$ several functions were examined. These functions take into account the magnitude of the weight in such a way as to reward a GP tree if it is able to correctly classify an instance of data. GP trees should receive a constant increment in fitness on those instances of data which are easily classified by the ensemble. This is done so that the GP algorithm can focus on optimising the GP trees on the instances which the ensemble could not correctly classify. Thus, let $g(x_i) = 1$ when an instance of data is correctly classified by a GP tree for which the weight is positive (third case).

Possible functions for $g(x_i)$	**Resulting $g(x_i)$ value**	
	First case: W = -1	Second case: W = 0
$\|W\|$	1	0
$\|W\| + 1$	2	1
$\|W\| + 2$	3	2
$\|W\| + 3$	4	3

Table 10.1: Possible functions for $g(x_i)$.

Table 10.1 illustrates four candidate functions for $g(x_i)$. For this discussion let the weight for the first case be "-1" and the weight for the second case be "0". The first function $g(x_i) = |W|$ is not suitable because $g(x_i)$ is equal to 1 for both the first and third case. Such a function would equally reward an instance of data which was previously incorrectly classified, and a instance which was previously correctly classified. The same argument applies for the function $g(x_i) = |W|+1$, this function is not suitable because correctly classified instances having a weight of 0 are equally rewarded as those instances in the third case. The function $g(x_i) = |W| + 2$ is suitable because each of the cases are rewarded differently. The first case receives a reward of 3, the second case receives a reward of 2, and the third case always receives a reward of 1. Increasing the value which is added to W is not necessary (for example $g(x_i) = |W|+3$) as this will simply add a greater bias towards instances which were previously incorrectly classified, and consequently instances which fall under third case have a smaller impact. The final formulation for the function $g(x_i)$

is as follows:

$$g(x_i) = \begin{cases} |W| + 2 & if\ W < 0\ (first\ case) \\ 2 & if\ W = 0\ (second\ case) \\ 1 & if\ W > 0\ (third\ case) \end{cases}$$

Table 10.2 illustrates different weight values and the corresponding value for $g(x_i)$. From the table, it is clear that correctly classifying instances with negative weights will greater impact the fitness of the trees, than correctly classifying instances with positive weights. The more negative the weight, the greater the impact on the fitness. For weights greater than zero, the impact on the fitness is constant regardless of magnitude of the weights. The pseudocode for the evaluation of GP trees is illustrated in algorithm 10.2.

Weight, W	**Case**	**Value for**$g(x_i)$
-3	1	$\|-3\| + 2 = 5$
-2	1	$\|-2\| + 2 = 4$
-1	1	$\|-1\| + 2 = 3$
0	2	$\|0\| + 2 = 2$
1	3	1
2	3	1
3	3	1

Table 10.2: Different weight values and their corresponding value for $g(x_i)$.

Algorithm 10.2: Pseudocode for evaluating a GP tree.

```
   input: A GP tree t
   output: The fitness of t
1  begin
2      fitness ← 0;
3      for each instance i in the training set do
4          DetermineEnsembleOutput (t);
5          if t was correctly classified then
6              if weight_i > 0 then
7                  fitness ← fitness + 1;
8              else
9                  fitness ← fitness + |weight_i| + 2;
10             end
11         end
12     end
13     return fitness
14 end
```

10.2.4 Updating the weights

The weights are only updated when a new tree is added into the ensemble. The ensemble is evaluated on each instance of data in the training set and the weights are updated based on the ensemble's ability to correctly, or incorrectly, classify each instance. The algorithm for updating the weights is presented in algorithm 10.3.

Algorithm 10.3: Pseudocode for updating the weights.

```
    input: intances of training data, and corresponding weight, weight_i, for
           instance i
    output: weights updated for each training instance
 1  begin
 2      for each instance i in the training set do
 3          sum ← 0;
 4          for each tree t in the ensemble do
 5              DetermineTreeOutput (t);
 6              if t was correctly classified then
 7                  sum ← sum + 1;
 8              else
 9                  sum ← sum − 1;
10              end
11          end
12          if sum > 0 then
13              if weight_i ¡ 0 then
14                  weight_i ← 1;
15              else
16                  weight_i ← weight_i + 1;
17              end
18          else
19              if weight_i ¿ 0 then
20                  weight_i ← −1;
21              else
22                  weight_i ← weight_i − 1;
23              end
24          end
25      end
26  end
```

Lines 12 to 17, and 18 to 23, describe how the weights are updated when the ensemble correctly and incorrectly classifies an instance respectively. The weights cannot just be incremented or decremented based on the performance of the ensemble as it is possible that the weights are updated incorrectly. The additional checks in order to correctly update the weights are presented in lines 13 to 15, and 19 to 21. Table 10.3 illustrates the need for lines 13 to 15. In this illustration, an ensemble of size 5 is constructed where each line in the table illustrates the state of the ensemble in terms of the tree outputs for some instance of data. Let class "B" represent

the correct classification. When the third individual is added to the ensemble, the classification is incorrect, and when the fourth individual is added to the ensemble, the classification results in a clash. In both cases the weight is decremented since the ensemble has not correctly classified the instance. However, when the fifth individual is added, the ensemble now correctly classifies the instance (since the mode is B). If the weight was incremented by 1, then the new weight would be "-3", and this weight would be incorrect since the ensemble correctly classifies the instance. Thus, the weight is reset to 1. Lines 19 to 21 describe a similar logic. Once the weights are updated the GP algorithm continues to train with the new weights.

Tree output in ensemble	**Weight (after updating)**	**Sum**	**Classification**
A	-1	-1	Incorrect
A, A	-2	-2	Incorrect
A, A, B	-3	1	Incorrect
A, A, B, B	-4	0	Incorrect (clash)
A, A, B, B, B	1	1	Correct

Table 10.3: Illustrating how the weights are updated. Let the correct class for some instance of data be "B".

10.3 Experimental Setup

Seven experiments were performed in order analyse the performance of the proposed ensemble construction methods. The first three experiments, namely *ensemble5*, *ensemble7* and *ensemble9*, examined different ensemble sizes and had a fixed ensemble size of 5, 7, and 9 respectively. After every 40 generations (parameter determined through trial runs) a tree was added to the ensemble, thus in *ensemble5*, a tree was added to the ensemble in generations 40, 80, 120, 160, and 200. In order to determine how the ensemble construction method compared to the standard GP algorithm three additional experiments were performed. The standard GP experiment were *stdGP200*, *stdGP280*, and *stdGP360*, which ran the standard GP algorithm for 200, 280, and 360 generations respectively. These corresponded to the number of generations for *ensemble5*, *ensemble7* and *ensemble9*. Additionally, these experiments used identical GP parameters as the ensemble experiments, and thus allowed for a comparison between standard GP and the proposed ensemble methods to be made.

Other than algorithms described in this chapter, the overall GP system remained unchanged, and the GP system was implemented as described in section 5.5. The initial GP population generation, GP selection methods, GP fitness evaluation and GP GOs were implemented as described in that section. The results were obtained

using the approach described in section 5.6. Decision trees were evolved when the data sets had categorical attributes, and arithmetic trees were evolved when the data sets had numerical attributes.

10.3.1 GP parameters

The GP parameters used throughout all the experiments in this chapter are presented in table 10.4. These parameters were determined empirically through trial runs.

GP Parameter	**Value**
Population Size	700
Parent Selection Method	Tournament selection of size 7
Maximum Initial Population Tree Size	7
Initial Population Generation Method	Ramped half and half
Crossover Rate	70%
Mutation Rate	30%
Maximum Number of Generations	*Ensemble5*: 200 *Ensemble7*: 280 *Ensemble9*: 360
Add Frequency	40 generations
GP Model	Generational model
Function Set (arithmetic trees)	$+, -, \times, /$
Function Set (decision trees)	Attributes
Terminal Set (arithmetic trees)	Attributes
Terminal Set (decision trees)	Classes

Table 10.4: GP parameters used.

10.3.2 Data sets

The 12 data sets which were used for the experiments are presented in table 10.5. Data sets which have continuous and categorical attributes were investigated, and additionally binary and multiclass data sets were selected. Further details about the data sets were presented in chapter 5.

Balance	Iris
Car	Pima Indians
Climate	Soybean
Ecoli	TTT
Fertility	WDBC
Ionosphere	Zoo

Table 10.5: Data sets used for ensemble construction experiments.

10.4 Conclusion

This chapter proposes a GP ensemble construction method for data classification. In this approach a single ensemble is created during the evolutionary process. The ensemble approach adds individuals into the ensemble by performing hill climbing in order to find an individual which will increase the overall accuracy of the ensemble. The algorithm focuses on instances of data which are more challenging to classify through the use of weights. A weight is allocated to each instance of data, a positive weight represents an instance which is easy to correctly classify, and a negative weight represents an instance which is difficult to correctly classify. The fitness function takes into consideration the magnitude of the weights. Thus, the fitness function rewards the ensemble when it correctly classifies instances of data which have previously been challenging to classify. The weights for each instance are updated based on the ensemble's ability to classify them.

Chapter 11

Results and Discussion

11.1 Introduction

Chapters 6, 7, 8, 9 and 10 proposed several GP and data classification topics for investigation. This chapter presents and discusses the results obtained by the proposed methods on those topics. Statistical testing was performed for all of the results obtained. Details regarding the statistical tests are described in chapter 5, section 5.3. In the results presented in this chapter, a "**" symbol indicates statistical significance, and a "†" symbol indicates statistical insignificance. For each section, the average performance of the algorithm along with the standard deviation of each method across all of the data sets is presented. Section 11.2 discusses the results obtained by incorporating discretisation into the GP algorithm. A discussion on GP representations for binary data classification is presented in section 11.3. The results obtained by the GP encapsulation methods are presented in section 11.4. Section 11.5 discusses the performance of the different methods which hybridised GA and GP. The results obtained by the ensemble construction method are presented in section 11.6. Finally section 11.7 concludes this chapter.

11.2 GP Discretisation

The results for the GP discretisation experiments are presented in tables 11.2 and 11.3. The experiment IDs are defined in table 11.1.

GPEI approach with a fixed arity of 2, obtained the highest overall training accuracy with a value of 87.88%. This approach obtained the best result on 7 out of the 12 data sets. On 4 of these, the results obtained by the GPEI with arity 2 were statistically significant when compared to every other approach. In the case of

ID	**Experiment**	**ID**	**Experiment**
1	GPEI with varying arity	5	EWI with arity 2
2	GPEI with arity 2	6	EWI with arity 3
3	GPEI with arity 3	7	EWI with arity 4
4	GPEI with arity 4	8	EWI with varying arity

Table 11.1: Experiment IDs for the discretisation methods.

Glass and *WDBC*, GPEI with arity 2 failed to achieve statistical significance against only one other method. The next best approach on the training data was the GPEI method with varying arity which obtained an average of 87.20%, and obtained the best result on 2 data sets. The approach which resulted in the weakest average training performance was EWI with arity 2.

Data set	**Experiment ID**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Ecoli	89.06 **	**89.63**	88.76 **	87.84 **	78.79 **	83.72 **	82.56 **	84.38 **
Fertility	95.58 †	95.58 †	95.18 †	93.87 **	94.47 **	93.38 **	92.84 **	**95.78**
Glass	77.50 †	**78.31**	76.54 **	74.13 **	60.45 **	73.41 **	72.24 **	75.94 **
Ionosphere	**97.08**	96.42 **	96.93 **	96.73 **	88.62 **	96.00 **	96.89 **	96.71 **
Iris	98.90 **	**99.21**	**99.21**	99.04 †	79.87 **	97.30 **	93.51 **	98.40 **
Pima Indians	**81.78**	81.44 †	81.70 †	81.12 **	70.92 **	79.36 **	81.59 †	81.74 †
Sonar	92.42 **	94.70 **	93.66 **	91.53 **	85.62 **	96.11 †	95.99 †	**96.52**
Spectf	90.96 †	90.30 †	**91.00**	90.01 **	79.99 **	79.95 **	80.61 **	80.80 **
Vehicle	71.87 **	**72.84**	72.04 **	67.82 **	61.94 **	68.28 **	69.93 **	71.91 **
WDBC	97.26 **	**97.52**	96.94 **	96.46 **	94.12 **	96.49 **	97.34 †	97.29 **
Wine	99.01 **	**99.63**	98.88 **	98.19 **	93.60 **	97.96 **	99.16 **	99.01 **
Yeast	54.93 **	**59.03**	54.29 **	50.94 **	40.82 **	55.05 **	52.98 **	55.39 **
Average	*87.20 ±13.45*	*87.88 ±12.57*	*87.09 ±13.65*	*85.64 ±14.70*	*77.43 ±16.37*	*84.75 ±13.93*	*84.64 ±14.14*	*86.16 ±13.67*

Table 11.2: Training accuracy (%) results for the different GP discretisation methods. For each data set, the best result is highlighted in bold, and was statistically tested with every other result.

In terms of the test results, GPEI with arity 2 also achieved the highest average performance. This method had an average accuracy of 78.38% and obtained the best result on 6 data sets. On the *Yeast* data set, GPEI with arity 2 was statistically significant against every other method. GPEI with arity 2 obtained statistical significance against 6 methods on the *Ionosphere* and *Vehicle* data sets. Similar to the training results, the next best method was GPEI with varying arity, which obtained an average test accuracy of 77.18%; however, this method did not obtain the best result on any data set. Once again, EWI with arity 2 resulted in the weakest accuracy on the test data.

Based on the findings from the test data, GPEI with arity 2 was the best overall method obtaining the highest average, the lowest standard deviation across all of the data sets, and this method obtained the best result on 6 data sets.

Data set	**Experiment ID**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Ecoli	82.06 †	**82.83**	79.04 **	80.86 †	75.35 **	76.65 **	75.25 **	76.89 **
Fertility	80.20 †	81.00 †	81.60 †	**84.40**	80.00 **	82.00 †	82.60 †	81.20 †
Glass	61.91 †	63.17 †	**63.55**	60.05 †	49.27 **	60.72 †	57.09 **	60.33 †
Ionosphere	89.68 †	**90.93**	87.86 **	88.15 **	83.80 **	87.12 **	84.15 **	87.17 **
Iris	94.40 †	95.07 †	95.07 †	93.60 †	76.53 †	**96.00**	89.87 †	94.93 †
Pima Indians	72.05 †	**73.38**	73.05 †	71.58 **	65.21 **	73.20 †	72.50 †	72.08 †
Sonar	70.97 †	**73.00**	69.70 **	68.90 **	67.51 **	71.05 †	69.82 †	69.48 **
Spectf	77.85 †	76.54 **	76.65 **	76.48 **	78.97 †	78.97 †	78.89 †	**79.05**
Vehicle	64.58 †	**65.52**	63.40 **	59.61 **	55.06 **	61.25 **	60.11 **	63.10 **
WDBC	93.32 †	93.99 †	93.39 †	93.07 †	92.30 **	92.97 †	93.42 †	**94.02**
Wine	87.75 **	89.41 †	88.22 †	87.26 **	84.93 **	**90.46**	87.78 **	89.46 †
Yeast	51.39 **	**55.69**	50.94 **	48.01 **	38.83 **	51.82 **	49.77 **	51.42 **
Average	*77.18 ±13.42*	*78.38 ±12.78*	*76.87 ±13.41*	*76.00 ±14.61*	*70.65 ±15.99*	*76.85 ±13.90*	*75.10 ±13.79*	*76.59 ±13.80*

Table 11.3: Test accuracy (%) results for the different GP discretisation methods. For each data set, the best result is highlighted in bold, and was statistically tested with every other result.

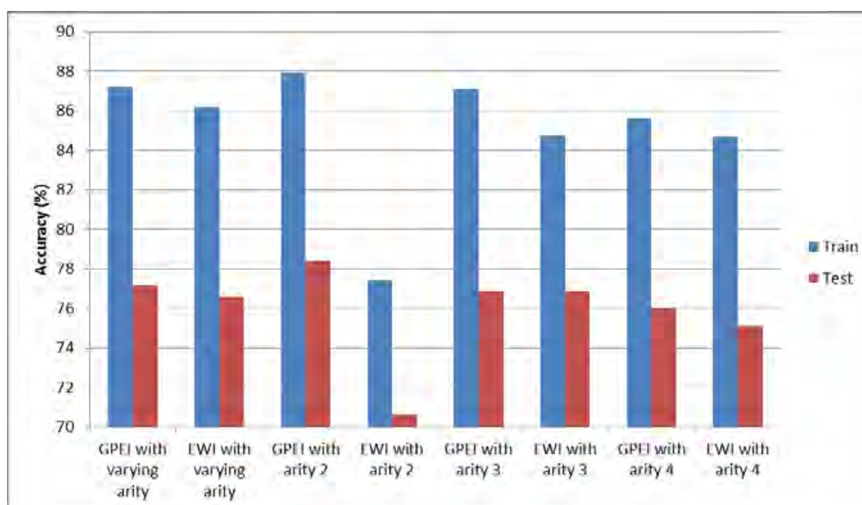


Figure 11.1: Comparing GPEI and EWI in terms of training and test accuracy (%).

Figure 11.1 illustrates the comparison between the GPEI and EWI methods. The figure reveals that for both training and testing, GPEI outperformed the corresponding EWI method. In terms of training, GPEI with varying arity outperformed the corresponding EWI method by 1.04%, and GPEI with arity 2 outperformed the corresponding EWI method by 10.45%. GPEI with arity 3 outperformed its corresponding EWI method by 2.34%, and finally, GPEI with arity 4 outperformed its corresponding EWI method by 1.00%. Similarly, in terms of the test results, GPEI with varying arity outperformed the corresponding EWI method by 0.59%. The largest difference was once again achieved when an arity of 2 was used. GPEI with arity 2 outperformed the corresponding EWI method by 7.73%. The smallest difference, of 0.02%, was achieved when an arity of 3 was used. Finally, GPEI with arity 4 outperformed its corresponding EWI method by 0.89%.

The GPEI approaches randomly create intervals with no guidance on how to create these intervals, and yet the results reveal that even with this randomness driving the search, the GPEI approach is still able to outperform the EWI approaches. Compared to other approaches found in the literature which use additional information to create the intervals, for instance entropy based methods, the results obtained in this section show that randomness in selecting intervals can be successful.

Table 11.4 presents the average tree size for each experiments on all the data sets. The size is measured in terms of the number of nodes in the best GP tree for each method, a smaller value denotes a better result. The lowest average was obtained by the EWI with arity 2. The next lowest average result was obtained by GPEI with arity 2. These results are to be expected since both methods have the least number of branches for each node within the evolved trees. GPEI with arity 2

obtained the smallest result on 4 data sets whereas EWI with arity 2 obtained the smallest result on 6 data sets. The GPEI methods obtained an average size of 79.03 nodes, and the EWI methods obtained an average size of 83.89 nodes.

Data set	**Experiment ID**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Ecoli	67.60 **	34.88 **	84.04 **	140.76 **	**29.32**	92.86 **	150.92 **	67.34 **
Fertility	45.64 **	**27.32**	49.18 **	78.28 **	29.32 †	54.52 **	125.24 **	53.96 **
Glass	64.48 **	39.24 **	77.02 **	143.56 **	**35.64**	81.76 **	144.84 **	82.34 **
Ionosphere	61.96 **	**25.04**	71.62 **	137.64 **	25.20 †	78.70 **	175.56 **	76.58 **
Iris	22.84 **	16.02 **	32.80 **	50.60 **	10.00 **	**7.60**	15.56 **	24.28 **
Pima Indians	107.54 **	47.88 **	105.28 **	215.56 **	**30.16**	113.62 **	299.00 **	154.64 **
Sonar	69.70 **	47.60 †	94.90 **	177.48 **	**46.04**	120.52 **	221.16 **	123.76 **
Spectf	64.14 **	35.52 **	74.44 **	100.44 **	14.48 †	**13.18**	35.88 **	24.76 **
Vehicle	121.60 **	**55.04**	143.50 **	255.08 **	57.32 †	145.66 **	297.96 **	154.84 **
WDBC	36.98 **	28.84 **	51.22 **	94.36 **	**20.88**	53.88 **	109.72 **	50.72 **
Wine	32.80 **	**11.48**	42.06 **	82.88 **	30.00 **	44.96 **	65.00 **	36.96 **
Yeast	85.52 **	51.56 **	97.78 **	191.72 **	**33.16**	117.28 **	153.48 **	95.74 **
Average	*65.07*	*35.04*	*76.99*	*139.03*	*30.13*	*77.05*	*149.53*	*78.83*

Table 11.4: Average size (number of nodes) of the best GP individuals for each method. The smallest size for each data set is highlighted in bold.

Since GPEI with arity 2 obtained the best average training and test accuracies, it was compared to other discretisers. Table 11.5 presents the results obtained by other discretisation techniques alongside the results obtained by the GPEI with arity 2. Other studies used different cross-validation techniques and may not have attempted to optimise their algorithm and parameters, thus this comparison serves as a means of determining an estimation on how well GPEI with arity 2 can perform. GPEI with arity 2 was compared with Lui *et al.* [75] which implemented C4.5 and 9 discretisation methods, Bacardit and Garrell [84] which implemented GAssist and reported on results obtained by ID3, and also reported on Fayyad and Irani's discretisation method [83], Hacibeyoglu *et al.* [80] report on k-NN, Naïve Bayes, and C4.5, and

finally a study on ADIs by Bacardit and Garrel [154]. These studies were discussed in chapter 3, section 3.7.4. The comparison indicates that GPEI with arity 2 is competitive with other state-of-the-art discretisation methods. GPEI with arity 2 did not outperform other discretisation methods on *Glass*, *Vehicle* and *Wine*. On the other data sets, the proposed GP discretisation method was able to outperform some of the existing methods.

Data set	**Test accuracy (%) of other state-of-the-art methods**	**Test accuracy (%) of GPEI with arity 2**
Iris	88.77 – 95.99 [75] 93.30 – 95.30 [80]	95.07
Ionosphere	88.10, 93.30, 89.50 [84] 88.02 – 91.48 [75] 92.70 [154] 87.50 – 93.40 [80]	90.93
Glass	67.89, 65.80 [154] 77.57 – 97.69 [75]	63.17
Pima Indians	73.10, 75.30 [154] 62.30 – 77.09 [75]	73.38
Sonar	76.40, 74.80, 72.65 [84] 71.50, 74.30 [154]	73.00
Vehicle	66.70, 73.60 [154] 66.67 – 72.10 [75]	65.52
WDBC	76.40, 74.80, 72.65 [84] 94.00, 93.70 [154]	93.99
Wine	93.00, 94.10 [154] 92.05 – 93.22 [75] 95.00 – 98.30 [80]	89.41

Table 11.5: Comparison between GPEI with arity 2 and other state-of-the-art discretisation methods.

11.3 GP Representations for Binary Classification

The training and test results for the GP representation experiments are presented in tables 11.6 and 11.7 respectively. For each data set, statistical tests were performed by comparing the performance of each of the representations. The representation which obtained the best result for a particular data set was statistically compared to the other representations. In terms of training accuracy, decision trees obtained the highest average accuracy, with an average of 92.40%, and obtained the lowest standard deviation. Decision trees obtained the best result for 5 data sets, and for 4 of these, the results were statistically significant when compared to all the

other representations. In the case of the *WDBC* data set, the results obtained by decision trees was not statistically significant when compared to *arithmetic-with-if*. The second best method was *arithmetic-with-if*, which had an average of 90.13%. This representation obtained the best result on two data sets. *Arithmetic-without-if* obtained the weakest average training accuracy with an average of 88.96%, and obtained the highest standard deviation.

	Arithmetic-without-if	**Arithmetic-with-if**	**Logical-with-between**	**Logical-without-between**	**Decision Trees**
Climate	97.42 †	**97.43**	96.64 **	97.19 †	96.86 **
Fertility	95.20 **	**95.96**	94.09 **	93.40 **	95.87 †
Ionosphere	95.83 **	95.97 **	**98.06**	97.96 †	96.42 **
Mammo-graphic	80.64 **	80.85 **	82.13 **	81.87 **	**82.72**
Monk2	81.94 **	87.46 **	79.25 **	77.07 **	**91.50**
Parkinsons	91.99 **	92.24 **	89.85 **	89.61 **	**95.31**
Pima Indians	74.94 **	75.79 **	77.94 **	78.75 **	**81.92**
Sonar	87.84 **	90.05 **	94.04 †	**94.88**	94.70 **
Spectf	86.62 **	88.31 **	**91.96**	89.58 **	91.15 †
WDBC	97.20 **	97.28 †	95.32 **	95.60 **	**97.52**
Average	*88.96 ±7.46*	*90.13 ± 6.92*	*89.93 ± 7.05*	*89.59 ± 7.35*	*92.40 ± 5.42*

Table 11.6: Training accuracy (%) results for the different representations. For each data set, the best result is highlighted in bold. A "**" indicates that the best result for the data set is statistically significant when compared to that result. A "†" indicates a statistically insignificant result when compared to the best result.

In terms of the training data, there was a difference in performance between the results obtained by the two arithmetic representations and by the two logical representations, in the sense that in 9 out of the 10 data sets, they produced dissimilar performance. This can be visualised in figure 11.2. Both of the arithmetic representations produced higher results than the two logical representations on *Fertility*, *Monk2*, *Parkinsons*, and *WDBC*. Whereas on *Ionosphere*, *Mammographic*, *Pima Indians*, *Sonar* and *Spectf* the opposite performance is observed, i.e. both of the logical representations outperformed the two arithmetic representations. These observations were not present for the *Climate* data set as all of the representations produced similar results.

	Arithmetic-without-if	Arithmetic-with-if	Logical-with-between	Logical-without-between	Decision Trees
Climate	**94.33**	94.15 †	92.15 **	92.78 **	90.78 **
Fertility	82.00 †	82.20 †	84.20 †	**84.80**	80.20 **
Ionosphere	88.71 **	88.94 **	**92.02**	92.01 †	90.93 †
Mammo-graphic	79.52 †	79.64 †	**79.93**	79.73 †	79.29 †
Monk2	76.68 **	79.68 **	75.89 **	73.43 **	**88.76**
Parkinsons	**86.70**	84.72 †	84.83 †	85.17 †	86.69 †
Pima Indians	69.30 **	69.29 **	73.67 †	**73.96**	73.62 †
Sonar	72.47 †	74.30 †	75.16 †	**75.28**	73.00 †
Spectf	**77.77**	76.87 †	75.97 †	76.83 †	76.58 †
WDBC	**95.15**	94.76 †	92.76 **	92.51 †	93.99 **
Average	*82.26 ±8.34*	*82.46 ±7.89*	*82.66 ±7.22*	*82.65 ± 7.45*	*83.38 ± 7.35*

Table 11.7: Test accuracy (%) results for the different representations. For each data set, the best result is highlighted in bold. A "**" indicates that the best result for the data set is statistically significant when compared to that result. A "†" indicates a statistically insignificant result when compared to the best result.

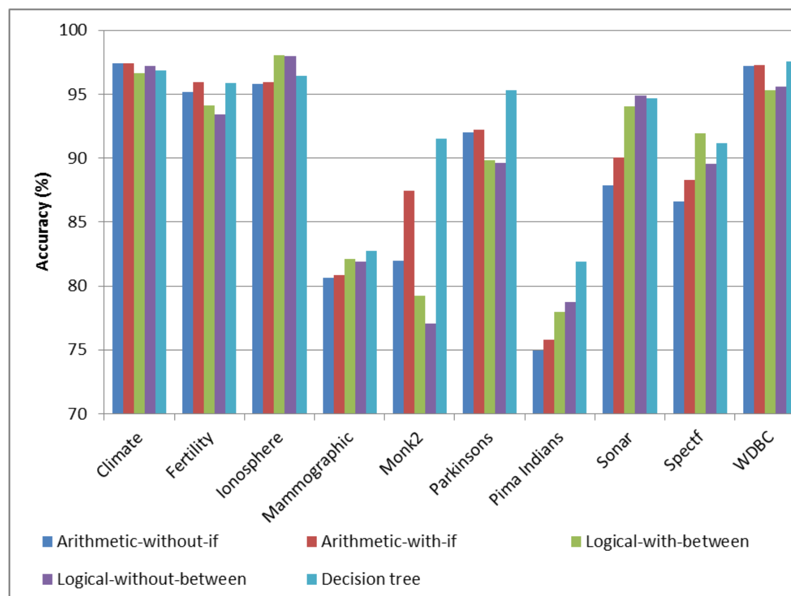


Figure 11.2: Illustrating the average training accuracy (%) for the different representations.

There are no apparent characteristics within the data sets that would stand out as a possible reason for this difference in performance between the arithmetic and logical representations. The data sets for which the arithmetic representations performed better have attributes varying from 9 to 30, and the total number of instances vary from 100 to 569. Whereas the attributes vary from 5 to 44 for data sets on which the logical representations performed better, and the number of instances vary from 208 to 961. Furthermore, the data sets for which the representations produced dissimilar performance have similar class balance. Since the characteristics of the data sets are similar, no conclusion can be made as to why the representations perform differently. The training performance of the decision trees did not show any particular trend when compared to the arithmetic and logical representations.

In terms of the test data, decision trees obtained the highest average with a value of 83.38%, and obtained the second lowest standard deviation. Furthermore, decision trees had the best result for the *Monk2* data set, and this result was statistically significant against all the other methods for this data set. *Logical-with-between* was the next best method, with an average test accuracy of 82.66%. *Arithmetic-without-if* performed the weakest and obtained the highest standard deviation, however it had best result on 3 data sets. Both of the logical representations performed similarly with only a 0.01% difference in average performance.

Based on the findings from the test data, decision trees was the best overall representation. Decision trees obtained the highest average and the second lowest standard deviation across all of the data sets.

Figure 11.3 graphically illustrates the test results for the different representations. Both of the arithmetic representations outperformed the logical representations on the *Climate* and *WDBC* data sets. On the other hand, on the *Fertility*, *Ionosphere*, *Pima Indians* and *Sonar*, both of the logical representations outperformed the arithmetic ones. By combining the findings from the training and test results the following observations are made. The arithmetic representations outperformed the logical representations on the *WDBC* data set in terms of both training and testing. The logical representations outperformed the arithmetic ones on the *Ionosphere*, *Sonar* and *Pima Indians* data sets in terms of training and testing.

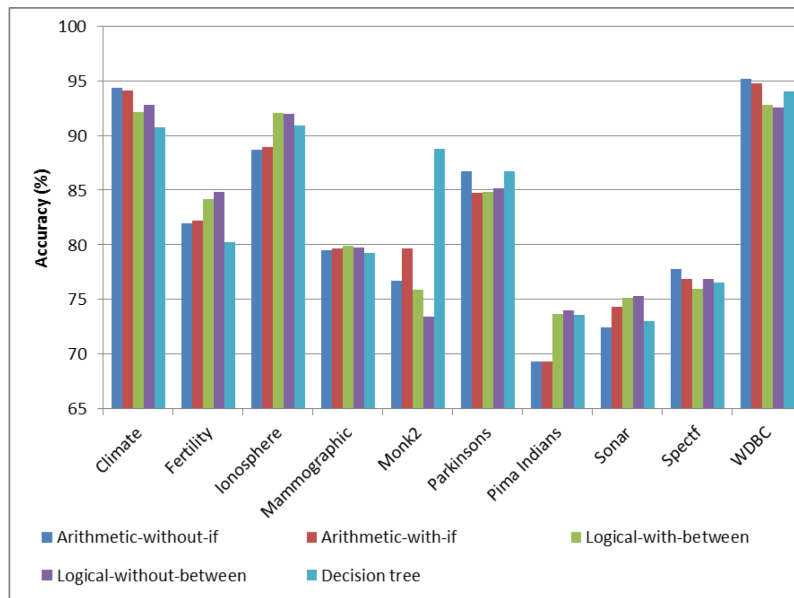


Figure 11.3: Illustrating the average test accuracy (%) for the different representations.

The variance in the average accuracy is presented in table 11.8. All of the representations obtained similar test results on the *Mammographic* and *Spectf*. The variances in the performance of the different methods were not consistent. This indicates that the different representations perform differently for various data sets.

Data sets	**Training variance**	**Testing variance**
Climate	0.10	1.73
Fertility	1.01	2.73
Ionosphere	0.94	2.08
Mammographic	0.62	0.05
Monk2	28.30	28.35
Parkinsons	4.23	0.79
Pima Indians	6.02	4.78
Sonar	8.08	1.28
Spectf	3.70	0.34
WDBC	0.86	1.10

Table 11.8: Variance amongst the different methods for each data set.

Table 11.9 presents the average size for the different representations. The size of the best GP individual was recorded for each representation, and the results were averaged across the 50 GP runs for each data set.

	Arithmetic-without-if	**Arithmetic-with-if**	**Logical-with-between**	**Logical-without-between**	**Decision Trees**
Climate	**26.88**	28.42	33.94	37.72	35.32
Fertility	26.64	28.34	28.36	21.54	**19.76**
Ionosphere	35.56	37.10	45.18	41.90	**25.04**
Mammo-graphic	**30.32**	37.38	46.40	45.26	44.64
Monk2	43.48	70.30	**30.58**	37.24	65.66
Parkinsons	29.12	33.12	26.68	**21.04**	26.00
Pima Indians	41.08	49.20	**31.90**	43.14	51.24
Sonar	**30.28**	38.48	50.36	53.30	47.60
Spectf	**33.16**	43.46	45.72	36.68	38.60
WDBC	**25.72**	32.70	30.84	30.06	28.84
Average	32.22	39.85	37.00	36.79	38.27

Table 11.9: Average size (number of nodes) of the different representations. The smallest result for each data set is highlighted in bold.

Arithmetic-with-if resulted in the largest average size, with a value of 39.85 nodes. The smallest average size was obtained by the *arithmetic-without-if*. By including the *if* statement, the average tree size was always larger than those without it, with an average increase of 7.63 nodes. Although the *if* statement increased the overall complexity of the trees, this did not hinder the performance of the representation, since there was an average improvement in both training and testing when compared to *arithmetic-without-if*. Including the *between* statement did not increase the average size of the trees as drastically as the inclusion of the *if* statement. Adding the *between* statement increased the average tree size by an average of 0.21 nodes. Decision trees resulted in the second largest average size with an average of 38.27 nodes. This represents an average of 6.05 nodes larger than *arithmetic-without-if*. Despite the decision tree representation being larger than *arithmetic-without-if*, it obtained a higher average performance.

11.4 GP Encapsulation

In this section, standard GP refers to standard GP without the encapsulation operator. Tables 11.10 and 11.11 present the training and test accuracy for the two proposed encapsulation methods. For each data set statistical tests were performed by comparing the performance of the encapsulation methods with standard GP.

Data sets	**Standard GP without encapsulation**	**GP encapsulation method**	**Selective encapsulation**
Climate	96.86	**96.88** †	96.71 †
Ecoli	89.63	**90.67** **	90.26 **
Fertility	**95.58**	95.33 **	94.76 **
Glass	78.31	**81.22** **	79.58 **
Ionosphere	96.42	**98.17** **	97.37 **
Iris	99.21	99.57 **	**99.61** **
Pima Indians	81.92	**83.35** **	82.15 **
Sonar	94.70	**96.45** **	94.99 **
Spectf	91.15	**93.20** **	91.91 **
WDBC	97.52	**97.93** **	97.64 †
Wine	**99.63**	**99.63** †	99.46 †
Yeast	59.03	**61.55** **	61.01 **
Average	*90.00 ± 11.83*	*91.16 ± 11.10*	*90.45 ± 11.28*

Table 11.10: Training accuracy (%) results for standard GP and the two proposed encapsulation methods. For each data set, the best result is highlighted in bold, and the results for the encapsulation methods were statistically tested with the results obtained by standard GP. A "**" indicates that the result is statistically significant when compared to the result obtained by standard GP without encapsulation. A "†" indicates a stastically insignificant result compared to standard GP without encapsulation.

In terms of training, the standard GP algorithm obtained the lowest overall accuracy and the highest standard deviation, but obtained the best result on 2 data sets, namely *Fertility* and *Wine*. In the case of the *Fertility* data set, standard GP obtained statistically significant results when compared to the encapsulation methods; however, on the *Wine* data set the best result was statistically insignificant in comparison to the other approaches. The GP encapsulation method obtained an overall average of 91.16% across all the training data sets, which represents an improvement in accuracy of 1.16% over standard GP. The GP encapsulation method obtained the lowest standard deviation on the training data. Furthermore, this method obtained statistically significant results on 10 data sets when compared to standard GP, 8 of which were the best result. On the remaining 2 data sets, *Climate* and *Wine*, it also achieved the best the results but were statistically insignificant.

Data sets	Standard GP without encapsulation	GP encapsulation method	Selective encapsulation
Climate	**90.78**	90.70 †	**90.78** †
Ecoli	**82.83**	82.48 †	82.55 †
Fertility	**81.00**	79.60 **	80.80 †
Glass	63.17	**64.92** †	64.17 †
Ionosphere	**90.93**	90.42 †	90.32 †
Iris	**95.07**	94.67 †	94.67 †
Pima Indians	**73.62**	73.02 †	72.78 **
Sonar	**73.00**	72.41 †	72.41 †
Spectf	76.58	75.98 †	**77.11** †
WDBC	93.99	94.06 †	**94.16** †
Wine	89.41	**90.94** †	90.09 †
Yeast	55.69	56.12 †	**56.62** **
Average	*80.51 ± 12.54*	*80.44 ± 12.36*	*80.54 ± 12.28*

Table 11.11: Test accuracy (%) results for standard GP and GP with the two proposed encapsulation methods. For each data set, the best result is highlighted in bold, and the results for the encapsulation methods were statistically tested with the results obtained by standard GP. A "**" indicates that the result is statistically significant when compared to the result obtained by standard GP without encapsulation. A "†" indicates a statically insignificant result compared to standard GP without encapsulation.

Selective encapsulation made use of the maintained list and this approach obtained an overall training average of 90.45% which is better than standard GP by 0.45%. Selective encapsulation obtained the best result on the *Iris* data set, and this result was statistically significant when compared to standard GP. Although selective encapsulation did not obtain the best result on as many data sets as the GP encapsulation method, selective encapsulation obtained a statistically better result on a total of 9 data sets when compared to standard GP. The findings reveal that both of the GP encapsulation methods obtain a better training accuracy than standard GP.

In terms of the test results, the selective encapsulation approach obtained the highest overall accuracy with a value of 80.54% and obtained the best result on 3 data sets, this however was only statistically significant on the *Yeast* data set. Selective encapsulation outperformed standard GP on 5 data sets, and furthermore obtained the lowest standard deviation. The GP encapsulation method obtained the best result on 2 data sets, and outperformed standard GP on 4 data sets; these were however not statistically significant. The GP encapsulation method obtained the lowest overall test accuracy. Standard GP obtained the best result on 7 test data sets, and achieved an overall average of 80.51%. This method ranked second

best, however, obtained the highest standard deviation.

Based on the test data, the three methods obtained similar performance, however, the selective encapsulation method obtained the highest average accuracy along with the lowest standard deviation.

After each GP run, the number of encapsulated terminals which were present in the best GP tree was recorded. Table 11.12 presents the average number of encapsulated terminals which were found in the best GP individuals for each of the data sets. Selective encapsulation made use of the maintained list, whereas the GP encapsulation method did not. When the maintained list was not used the average number of encapsulated terminals in the best GP tree was 48.46. There was a considerable reduction when the maintained list was used, with an average of 28.64 encapsulated terminals. Furthermore, these results were statistically significant for every data set thus confirming that the selective encapsulation method reduces the number of encapsulated terminals in the trees. Reducing the number of encapsulated terminals implies that the complexity of the tree is also reduced.

Data set	**Without Maintained List**	**With Maintained List**
Climate	45.74	27.60 **
Ecoli	49.98	30.84 **
Fertility	32.22	17.88 **
Glass	52.04	32.52 **
Ionosphere	45.56	24.64 **
Iris	24.14	11.92 **
Pima Indians	72.36	41.18 **
Sonar	55.44	35.68 **
Spectf	61.24	32.50 **
WDBC	40.72	23.18 **
Wine	28.12	18.52 **
Yeast	73.98	47.27 **
Average	*48.46*	*28.64*

Table 11.12: Comparison between the number of encapsulated terminals which were present in the best GP individuals for the two encapsulation methods. All the results obtained by encapsulation with maintained list were statistically significant compared to when the list was not used, this is denoted by "**".

11.5 Hybridisation of GA and GP

Tables 11.13 to 11.16 present the results for the proposed hybrid ensemble methods. The training results and test results are presented separately. The best training and test results for each data set are highlighted in bold. Statistical tests were used to

compare the results obtained by the ensemble method to the standard GP approach in order to determine the effectiveness of the proposed methods.

Data set	**Ensemble Size**					**Standard GP**
	3	5	7	9	11	
Balance	96.77	96.76	96.76	96.76	96.76	**96.81** †
Climate	97.77	97.96	98.04	**98.05** **	98.03	97.71
Ecoli	**82.54** **	82.46	82.22	82.07	81.94	77.39
Fertility	96.18	**96.42** †	96.24	96.11	96.04	95.98
Ionosphere	96.30	96.65	96.82	**96.88** **	96.83	95.67
Iris	96.92	**96.96** **	96.90	96.81	96.73	81.94
Parkinson	92.32	92.76	92.92	92.93	**92.94** †	92.56
Pima Indians	75.79	76.18	76.26	**76.29** **	76.15	75.31
Sonar	89.14	89.64	**89.85** **	89.80	89.69	87.75
Soybean	79.60	79.59	79.59	79.60	79.59	**80.06** †
Spectf	87.75	88.22	88.42	**88.44** **	88.41	87.34
WDBC	97.54	97.69	97.74	**97.76** †	97.69	97.20
Average	*90.72 ± 7.74*	*90.94 ± 7.74*	*90.98 ± 7.75*	*90.96 ± 7.75*	*90.90 ± 7.77*	*88.81 ± 8.34*

Table 11.13: Average training classification accuracy (%) for *GA-at-end*. For each data set, the best result is highlighted in bold, and the best ensemble result is statistically tested against the standard GP result.

GA-at-end was the first proposed method at investigating the use of hybridising the GA with the GP algorithm. Tables 11.13 and 11.14 present the training and test results for *GA-at-end* respectively. Ensemble sizes of 3, 5, 7, 9, and 11 were investigated in order to determine which size would yield the best ensemble. An ensemble size of 3 did not perform as well as the other ensemble sizes for both the training and test data sets. *GA-at-end* with a size of 3 obtained the weakest training accuracy in comparison to the other ensemble sizes, with an average accuracy of 90.72%. Nonetheless, this method was 1.91% better in terms of training accuracy than standard GP, and also obtained a lower standard deviation than standard GP. However, *GA-at-end* with a size of 3 did not obtain the weakest average test accuracy when compared to the other ensemble sizes. This method obtained a better average

than sizes 9 and 11. Furthermore, for both the training and test sets, *GA-at-end* obtained a higher average accuracy for each ensemble size in comparison to standard GP.

Data set	**Ensemble Size**					**Standard GP**
	3	5	7	9	11	
Balance	68.64	68.77	68.77	68.77	68.74	**68.83** †
Climate	94.11	93.96	93.89	94.11	**94.30** †	93.52
Ecoli	77.91	78.02	77.78	77.72	77.84	**80.94** **
Fertility	81.00	82.00	**83.60** †	81.60	81.80	80.60
Ionosphere	90.43	**90.88** †	90.60	90.60	90.03	90.13
Iris	94.13	**94.27** †	**94.27** †	93.73	94.13	93.47
Parkinsons	86.37	86.06	86.26	**86.46** †	86.07	84.42
Pima Indians	69.74	69.45	69.51	**69.95** †	69.63	68.85
Sonar	75.47	74.88	**75.69** †	74.24	75.09	72.40
Soybean	72.49	72.36	72.49	72.56	72.49	**72.96** †
Spectf	75.62	75.46	75.24	75.54	75.23	**76.15** †
WDBC	94.90	94.76	**95.04** †	94.83	95.01	94.66
Average	*81.73* ± 9.86	*81.74* ± 9.93	*81.93* ± 9.91	*81.68* ± 9.87	*81.70* ± 9.90	*81.41* ± 9.77

Table 11.14: Average test classification accuracy (%) for *GA-at-end*. For each data set, the best result is highlighted in bold, and the best ensemble result is statistically tested against the standard GP result.

In terms of training, as the size of *GA-at-end* was increased from 3 to 7, the average accuracy improved for all of the data sets. For each of the different sizes investigated, the average training accuracy was higher than standard GP. The results indicate that *GA-at-end* with a size of 7 resulted in the best training accuracy with an average of 90.98%; which represents an improvement of 2.17% on the standard GP approach. Standard GP obtained a higher classification accuracy than *GA-at-end* when trained on the *Soybean* and *Balance* data sets; these result however were not statistically significant. On 7 of the 12 data sets, *GA-at-end* obtained a statistically significant best result when compared to standard GP.

Similar to the improvement in the average training results as a larger ensemble size was used; such an improvement was also present in the average test results when

the ensemble size was increased from 3 to 7. However, beyond a size of 7, there was a reduction in average test performance. *GA-at-end* with sizes of 9 and 11 performed worse than with a size of 7. This suggests that although increasing the ensemble size improves the average test accuracy of the classifiers, this improvement is no longer noticed when the ensemble is too large. This effect could be due to the fact that adding extra trees to the ensemble increases the overall complexity of the ensemble.

The ensemble size which produced the most consistent test results was that of size 7, which obtained the best result on 4 data sets (*Fertility*, *Iris*, *Sonar*, and *WDBC*), and an average of 81.93% across all of the test data sets. This represents an improvement of 0.52% when compared to the average test accuracy of standard GP. However, *GA-at-end* with a size of 7 obtained the second highest standard deviation in terms of the test data. *GA-at-end*, in general, outperformed standard GP on 8 test sets; however, these results were not statistically significant.

The training and test accuracies of *GA-after-each-gen* are presented in tables 11.15 and 11.16 respectively. This method obtained an average training accuracy of 92.17% which represents an improvement of 3.36% over standard GP. *GA-after-each-gen* outperformed standard GP on all 12 of the data sets, and these results were statistically significant for 10 data sets. *GA-after-each-gen* obtained the best result on 2 data sets. Furthermore, when compared to *GA-at-end*, *GA-after-each-gen* obtained a higher average training accuracy. *GA-after-each-gen* outperformed standard GP in terms of the test accuracy on 8 data sets, with an average improvement of 0.25%. *GA-after-each-gen* obtained a statistically significant result on the *Ecoli* test data set, and statistically insignificant results on 10 data sets. *GA-after-each-gen* obtained a higher average test accuracy than *GA-at-end* with sizes 9 and 11.

In *GA-after-each-gen*, the GA was able to create ensembles of different sizes. Every time the GA was run after each GP generation, the ensemble size was randomly selected from $\{3, 5, 7, 9, 11\}$. The average sizes of the ensembles for each data set are presented in table 11.17. The average size was 7.45 which is close to the best performing method for the *GA-at-end* which was 7. The smallest ensemble size was 4.92 for the *Balance* data set, and the largest was 9.00 for the *Climate* data set. Despite the average size of the ensembles evolved for the *Climate* data set being large, *GA-after-each-gen* when compared to standard GP, was able to obtain a better training and test accuracy of 0.96% and 0.89% respectively on that data set.

The training and test accuracies of *GA-with-HC* are presented in tables 11.15 and 11.16 respectively. Similar to the performance of *GA-at-end* and *GA-after-each-gen*, the *GA-with-HC* ensemble obtained better training results when compared to the standard GP approach. This ensemble method obtained an average training

Data set	GA-after-each-gen	GA-with-HC	SSGA-GP	Standard GP
Balance	**97.07** †	**97.07** †	93.40 **	96.81
Climate	98.67 **	**98.81** **	98.72 **	97.71
Ecoli	83.95 **	**85.00** **	81.90 **	77.39
Fertility	**97.49** **	97.47 **	96.27 †	95.98
Ionosphere	97.83 **	**98.05** **	97.66 **	95.67
Iris	98.98 **	**98.99** **	98.50 **	81.94
Parkinsons	94.55 **	94.64 **	**94.74** **	92.56
Pima Indians	77.27 **	77.36 **	**77.77** **	75.31
Sonar	91.23 **	**92.59** **	92.19 **	87.75
Soybean	**80.58** †	80.51 †	70.51 **	80.06
Spectf	90.15 **	90.15 **	**91.95** **	87.34
WDBC	98.27 **	**98.41** **	98.19 **	97.20
Average	*92.17 ± 7.66*	*92.42 ± 7.58*	*90.98 ± 9.24*	*88.81 ± 8.34*

Table 11.15: Average training classification accuracy (%). The best result for each data set is highlighted in bold. The result for each ensemble method was statistically compared to the result obtained by standard GP. Between each pairwise comparison, a "**" denotes that the higher result is statistically significant. A "†" denotes statistical insignificance.

Data set	GA-after-each-gen	GA-with-HC	SSGA-GP	Standard GP
Balance	69.48 †	70.28 †	**73.05** **	68.83
Climate	**94.41** †	94.19 †	94.11 †	93.52
Ecoli	78.20 **	78.98 †	77.32 **	**80.94**
Fertility	82.00 †	82.00 †	**82.80** †	80.60
Ionosphere	90.76 †	90.59 †	**90.93** †	90.13
Iris	93.47	**93.73** †	92.93 †	93.47
Parkinsons	84.05 †	85.25 †	**86.07** †	84.42
Pima Indians	69.06 †	**69.66** †	69.34 †	68.85
Sonar	73.91 †	**74.81** †	74.22 †	72.40
Soybean	**74.26** †	72.70 †	61.91 **	72.96
Spectf	76.13 †	74.48 †	74.58 †	**76.15**
WDBC	94.76 †	**95.29** †	94.52 †	94.66
Average	*81.71 ± 9.66*	*81.83 ± 9.72*	*80.98 ± 10.82*	*81.41 ± 9.77*

Table 11.16: Average test classification accuracy (%). The best result for each data set is highlighted in bold. The result for each ensemble method was statistically compared to the result obtained by standard GP. Between each pairwise comparison, a "**" denotes that the higher result is statistically significant. A "†" denotes statistical insignificance.

accuracy of 92.42% which represents an improvement of 3.61% when compared to standard GP. *GA-with-HC* outperformed standard GP on every data set, of which 10 of these were statistically significant and 2 results were not. Furthermore, *GA-with-HC* outperformed *GA-after-each-gen* in terms of average training accuracy by 0.25%. *GA-with-HC* obtained a higher accuracy than standard GP on 9 test data sets with an average improvement of 0.42% when compared to standard GP. However, these test results were not statistically significant. There was an average improvement of 0.12% when using *GA-with-HC* in comparison to *GA-after-each-gen* for the test data sets. *GA-with-HC* obtained the lowest standard deviation on the training data when compared to standard GP and the other ensemble methods.

The average size of the ensembles evolved during *GA-with-HC* are presented in table 11.17. *GA-with-HC* obtained an average size of 7.98 across all the data sets. This represents an increase in size of 0.53 in comparison to *GA-after-each-gen*. Similar to the performance of *GA-after-each-gen*, the smallest average size was obtained on the *Balance* data set. Statistical tests were performed on the sizes of the ensembles, and *GA-with-HC* created ensembles which were statistically larger than *GA-after-each-gen* on 5 data sets. Although *GA-with-HC* resulted in larger ensembles on average, this method was still able to obtain better test results than *GA-after-each-gen* which created smaller ensembles.

GA-after-each-gen and *GA-with-HC* are similar methods with the exception that the latter implements hill climbing. The results indicate that better results are achieved by incorporating hill climbing into the algorithm. The results from table 11.17 also suggest that for certain data sets a larger ensemble is required to obtain a high training accuracy. On the *Balance*, *Ecoli*, *Fertility* and *Soybean* data sets, both *GA-after-each-gen* and *GA-with-HC* created small ensembles, whereas both methods created large ensembles for *Climate*, *Spectf*, *Ionosphere*, *Parkinsons*, *Pima Indians*, *Sonar*, *Spectf* and *WDBC*. This indicates that setting a fixed ensemble size may not result in the most optimal training accuracy. Nonetheless, based on the results from *GA-at-end*, *GA-after-each-gen* and *GA-with-HC,* the findings show that the best results are obtained when an ensemble size of 7 is used.

SSGA-GP employs the use of a steady-state GA model whereby the ensembles had a fixed size of 7. The training and test accuracies of *SSGA-GP* are presented in tables 11.15 and 11.16 respectively. Once again similarly to the other three ensemble methods, *SSGA-GP* outperformed the standard GP on all the training data sets except on *Balance* and *Soybean*. *SSGA-GP*, however, obtained a higher standard deviation on the training data when compared to standard GP.

Where *SSGA-GP* outperformed standard GP on the training data, the results were statistically significant for all of the data sets except in the case of the *Fertility* data set. *SSGA-GP* obtained an average training accuracy of 90.98% which

represents an improvement of 2.48% when compared to standard GP. *SSGA-GP* outperformed the standard GP approach on 7 test data sets, and obtained a statistically significant better result on the *Balance* data set. The remainder of the best test results were not statistically significant, and standard GP outperformed *SSGA-GP* on the *Soybean* and *Ecoli* test data with statistical significance. Furthermore, *SSGA-GP* was outperformed by standard GP on *Iris*, *Spectf*, and *WDBC*; these results were not statistically significant. The average test accuracy for *SSGA-GP* was 0.48% weaker than standard GP. *SSGA-GP* did not perform as well as the other proposed ensemble methods on the test data, and furthermore, it obtained the highest standard deviation.

The results revealed that based on the test data, *GA-at-end* with a size of 7 obtained the highest overall test accuracy and produced the best result on 3 data sets, however, this method had the third highest standard deviation.

Data set	**GA-after-each-gen**	**GA-with-HC**
Balance	4.92 †	4.56
Climate	9.00	9.60 **
Ecoli	5.32	6.96 **
Fertility	6.08 †	5.80
Ionosphere	8.84	9.60 **
Iris	7.04	7.68 †
Parkinsons	8.80 †	8.59
Pima Indians	7.84	9.40 **
Sonar	8.20	9.12 **
Soybean	6.12 †	5.88
Spectf	8.56	9.60 †
WDBC	8.64	8.96 †
Average	*7.45*	*7.98*

Table 11.17: Comparison of the average size of the ensembles of *GA-after-each-gen* and *GA-with-HC*. For each data set, the larger result was statistically compared to the other result. A "**" indicates that the result is statistical larger than the other method. A "†" denotes a statistically insignificant result.

Data set	Ensemble Size					Other methods
	3	5	7	9	11	
Ionosphere	96.30	96.65	96.82	96.88	96.83	92.40 [13]
Iris	96.92	96.96	96.90	96.81	96.73	98.10 [13]
Parkinson	92.32	92.76	92.92	92.93	92.94	86.60 [13]
Pima Indians	75.79	76.18	76.26	76.29	76.15	72.20 [13]
Sonar	89.14	89.64	89.85	89.80	89.69	79.00 [13]
Spectf	87.75	88.22	88.42	88.44	88.41	81.90 [13]

Table 11.18: Training accuracy (%) comparison between *GA-at-end* and other methods found in literature.

Data set	Ensemble Size					Other methods
	3	5	7	9	11	
Balance	68.64	68.77	68.77	68.77	68.74	91.68 [155] 90.29 [155] 94.40 [8]
Ecoli	77.91	78.02	77.78	77.72	77.84	82.44 [8]
Ionosphere	90.43	90.88	90.60	90.60	90.03	88.50 [13] 85.40–90.52 [13] 82.00, 91.06 [155] 86.30, 89.91 [136] 92.30 [8] 92.30 [126]
Iris	94.13	94.27	94.27	93.73	94.13	96.00, 96.60 [13] 94.80, 95.53 [155] 95.33 [8] 97.90 [126]
Parkinsons	86.37	86.06	86.26	86.46	86.07	84.30 [13]
Pima Indians	69.74	69.45	69.51	69.95	69.63	68.60 [13] 68.30–75.75 [13] 73.70 [126]
Sonar	75.47	74.88	75.69	74.24	75.09	73.30 [13] 38.40–72.42 [13] 74.38, 74.87 [136]
Spectf	75.62	75.46	75.24	75.54	75.23	77.60, 83.20 [13]

Table 11.19: Test accuracy (%) comparison between *GA-at-end* and other methods found in literature.

Data set	GA-after-each-gen	GA-with-HC	SSGA-GP	Other methods
Ionosphere	97.83	98.05	97.66	92.40 [13]
Iris	98.98	98.99	98.50	98.10 [13]
Parkinsons	94.55	94.64	94.74	86.60 [13]
Pima Indians	77.27	77.36	77.77	72.20 [13]
Sonar	91.23	92.59	92.19	79.00 [13]
Spectf	90.15	90.15	91.95	81.90 [13]

Table 11.20: Training accuracy (%) comparison between the proposed ensemble methods and other methods found in literature.

Data set	GA-after-each-gen	GA-with-HC	SSGA-GP	Other methods
Balance	69.48	70.28	73.05	91.68 [155] 90.29 [155] 94.40 [8]
Ecoli	78.20	78.98	77.32	82.44 [8]
Ionosphere	90.76	90.59	90.93	88.50 [13] 85.40–90.52 [13] 82.00, 91.06 [155] 86.30, 89.91 [136] 92.30 [8] 92.30 [126]
Iris	93.47	93.73	92.93	96.00, 96.60 [13] 94.80, 95.53 [155] 95.33 [8] 97.90 [126]
Parkinsons	84.05	85.25	86.07	84.30 [13]
Pima Indians	69.06	69.66	69.34	68.60 [13] 68.30–75.75 [13] 73.70 [126]
Sonar	73.91	74.81	74.8174.22	73.30 [13] 38.40–72.42 [13] 74.38, 74.87 [136]
Spectf	76.13	74.48	74.58	77.60, 83.20 [13]

Table 11.21: Test accuracy (%) comparison between the proposed ensemble methods and other methods found in literature.

Tables 11.18 to 11.21 presents a comparison between the results obtained using the ensemble methods proposed in this dissertation with those results obtained using other GP methods found in Jabeen and Baig [13], Pappa and Freitas [136], Shali *et al.* [8], Eggermont *et al.* [126], Tan *et al.* [124], and AdaBoost Naïve Bayes and

bagging Naïve Bayes found in Kotsiantis and Pintelas [155]. It is not possible to directly compare the performance of the proposed methods in this dissertation to other approaches, due to the fact that other studies may have used different cross-validation techniques and/or different GP parameters, and also due to the fact that other studies may not have attempted to further optimise the performance of their algorithms. However, this comparison serves as an estimation of the performance of the proposed methods to other state-of-the-art methods. The comparison indicates that the proposed methods are able to perform better than other methods on certain data sets in terms of both training and testing.

11.6 GP Ensemble Construction

The training and test accuracy results for the proposed ensemble construction approach are presented in tables 11.22 and 11.23 respectively. Three ensemble sizes were tested, namely, *ensemble5*, *ensemble7* and *ensemble9* which correspond to 200, 280 and 360 GP generations respectively. The three ensemble methods obtained a higher overall average training accuracy than standard GP. The difference between the average ensemble accuracies and standard GP were 1.75%, 2.46% and 3.33% for *ensemble5*, *ensemble7* and *ensemble* 9 respectively. The findings indicate that the training accuracy improves as a larger ensemble is used. The ensembles obtained statistically better training results on all the data sets except in the case of *Balance* and *Soybean*. *Ensemble9* obtained the best result on 11 out of the 12 data sets of which 10 of these results were statistically significant.

Similar to the training results, the test results indicate that the ensembles produce classifiers which are superior to standard GP. The average test accuracy for each ensemble was higher than the results obtained by standard GP. *Ensemble9* obtained the highest average test accuracy, with a value of 85.88% which represents an improvement of 2.11% over standard GP with 360 generations.
Furthermore, *ensemble9* obtained the best result on 4 data sets and 2 of these were statistically significant. *Ensemble7* had a higher overall test accuracy when compared to standard GP with 280 generations, and obtained the best result on 3 data sets; these were statistically significant. On certain data sets, standard GP outperformed its corresponding ensemble method. On *Balance*, *Ecoli* and *Soybean* standard GP with 200 generations outperformed *ensemble5*. Additionally, standard GP with 360 generations outperformed *ensemble9* on 5 data sets, these results were not statistically significant.

Based on the test data, *ensemble9* obtained the highest overall average accuracy, and the best result on 4 data sets. This method ranked 4th best in terms of standard deviation, however, the standard deviation was lower than that of *ensemble5* and

Data sets	**Ensemble Size**			**Standard GP Generations**		
	5	*7*	*9*	*200*	*280*	*360*
Balance	89.43	92.42	94.63	92.40 **	93.70 **	**95.10** **
Car	89.15 **	94.46 **	**95.36** **	79.69	81.90	78.86
Climate	97.72 **	97.91 **	**98.20** **	96.59	96.90	96.71
Ecoli	90.63 **	91.96 **	**92.71** **	89.60	89.91	90.46
Fertility	95.29 **	96.24 **	**96.93** **	93.96	93.91	94.04
Ionosphere	95.85 **	96.42 **	**96.70** **	93.04	93.62	93.29
Iris	99.72 **	99.91 **	**99.94** **	99.30	99.38	99.23
Pima Indians	75.16 **	75.91 **	**76.32** **	72.79	72.93	72.91
Soybean	70.54	73.74	**77.21** †	72.91 **	75.87 **	76.61
TTT	96.50 **	98.48 **	**99.35** **	90.22	91.07	91.85
WDBC	97.24 **	97.54 **	**97.75** **	96.54	96.65	96.47
Zoo	99.89 **	99.89 **	**100.00** **	99.12	99.54	99.60
Average	*91.43 ± 9.46*	*92.91 ± 8.83*	*93.76 ±8.23*	*89.68 ± 9.44*	*90.45 ± 8.88*	*90.43 ±9.12*

Table 11.22: Training accuracy (%) results for the different ensembles and standard GP. The best result for each data set is highlighted in bold. For each data set, *ensemble5* was statistically compared to standard GP 200 generations, *ensemble* 7 to standard GP 280 generations, and *ensemble9* to standard GP 360 generations.

Data sets	**Ensemble Size**			**Standard GP Generations**		
	5	*7*	*9*	*200*	*280*	*360*
Balance	73.12	71.94	72.03	**75.00** **	74.85 **	73.66 †
Car	86.87 **	92.07 **	**92.80** **	78.71	80.52	77.06
Climate	94.41 †	**94.70** **	94.56 †	93.70	93.52	93.89
Ecoli	81.24	82.06	81.98	**83.81** **	82.25 †	83.60 †
Fertility	82.60 †	82.40	**83.00** †	81.60	82.40	80.80
Ionosphere	90.65 †	**91.62** **	91.39 **	89.44	88.77	88.77
Iris	94.80 †	95.07 †	94.93	94.27	94.27	**95.20** †
Pima Indians	**68.91** †	68.33 †	67.34	68.09	68.25	67.94 †
Soybean	64.35	66.70	**71.21** †	67.05 †	71.14 **	69.63
TTT	88.64 **	90.17 **	**91.57** **	84.55	84.30	84.47
WDBC	95.54 †	**95.89** **	95.53 **	94.87	94.72	94.52
Zoo	94.47	94.69 †	94.27	94.47	92.67	**95.67** †
Average	*84.63 ± 10.78*	*85.47 ± 10.97*	*85.88 ± 10.47*	*83.80 ±10.05*	*83.97 ±9.16*	*83.77 ±10.10*

Table 11.23: Test accuracy (%) results for the different ensembles and standard GP. The best result for each data set is highlighted in bold. For each data set, *ensemble5* was statistically compared to standard GP 200 generations, *ensemble* 7 to standard GP 280 generations, and *ensemble9* to standard GP 360 generations.

ensemble7.

Figure 11.4 graphically illustrates the difference in average accuracy between the ensembles and standard GP in terms of the test results. Given that the ensembles and standard GP pairs ran for the same number of generations, it is apparent from the figure that creating ensembles lead to an overall improvement in accuracy. There is no significant difference when comparing the computational effort between the proposed ensemble method and standard GP. The only additional processing which occurs in the ensemble method is the evaluation of the ensemble on the training data, and the updating of the weights. Since a new tree is added to the ensemble every 40 generations, this has little impact on the overall computational effort.

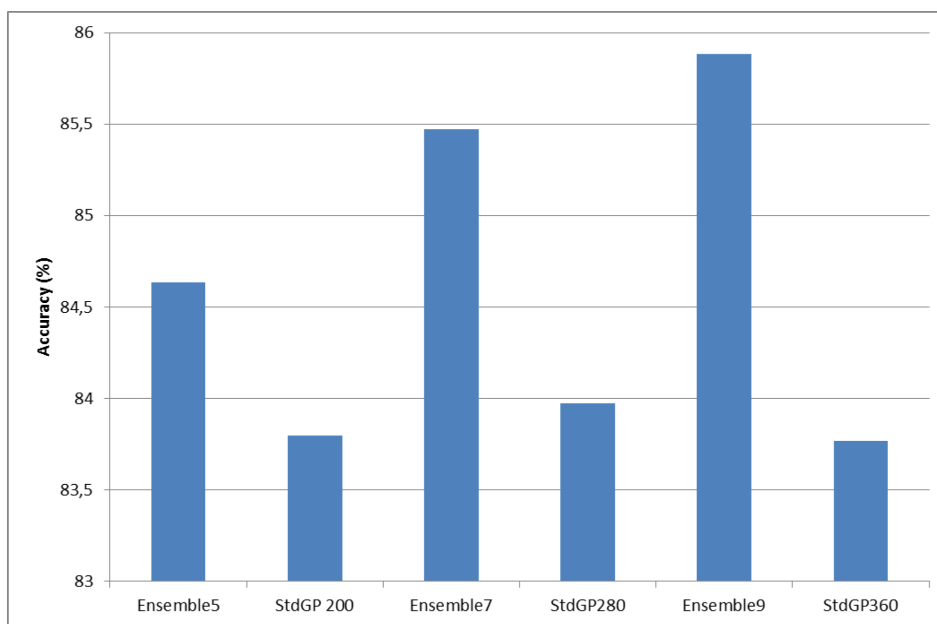


Figure 11.4: Comparison between the average test results for the ensembles and standard GP.

Tables 11.24 and 11.25 present a comparison between the results obtained using the ensemble methods proposed in this dissertation with the results obtained using other GP methods found in Jabeen and Baig [13], Pappa and Freitas [136], Shali *et al.* [8], Eggermont *et al.* [126], Tan *et al.* [124], and AdaBoost Naïve Bayes and bagging Naïve Bayes found in Kotsiantis and Pintelas [155]. It is not possible to directly compare the performance of the proposed methods in this dissertation to other approaches, due to the fact that other studies may have used different cross-validation techniques and/or different GP parameters, and also due to the fact that other studies may not have attempted to further optimise the performance of their algorithms. However, this comparison serves as an estimation of the performance of the proposed methods to other state-of-the-art methods. The comparison re-

veals that the proposed ensemble methods are able to perform just as well as other methods. In terms of the test results, the proposed ensemble method was able to outperform 3 studies on the *Ionosphere* data set, 1 study on the *Iris* data set and 2 studies on the *Pima Indians* data set.

Data sets	**Ensemble Size**			**Other methods**
	5	*7*	*9*	
Ionosphere	95.85	96.42	96.70	92.40 [13]
Iris	99.72	99.91	99.94	98.10 [13]
Pima Indians	75.16	75.91	76.32	72.20 [13]

Table 11.24: Training accuracy (%) comparison between the proposed ensemble construction method and other methods found in literature.

Data sets	**Ensemble Size**			**Other methods**
	5	*7*	*9*	
Balance	73.12	71.94	72.03	91.68 [155] 90.29 [155] 94.40 [8]
Ecoli	81.24	82.06	81.98	82.44 [8]
Ionosphere	90.65	91.62	91.39	88.50 [13] 85.40–90.52 [13] 82.00, 91.06 [155] 86.30, 89.91 [136] 92.30 [8] 92.30 [126]
Iris	94.80	95.07	94.93	96.00, 96.60 [13] 94.80, 95.53 [155] 95.33 [8] 97.90 [126]
Pima Indians	68.91	68.33	67.34	68.60 [13] 68.30–75.75 [13] 73.70 [126]
Zoo	94.47	94.69	94.27	95.07, 97.23 [155] 89.10 [8] 100.00 [124]

Table 11.25: Test accuracy (%) comparison between the proposed ensemble construction method and other methods found in literature.

Table 11.26 presents a comparison between the best results obtained by the ensemble methods created by the hybridisation of GA and GP in section 11.5, and the best results obtained using the ensemble construction approach in this section. The ensemble methods created by the hybrid approach obtained the best result on 7 training sets of which 6 were statistically significant results. Furthermore, the

Data set		**Method**	**Result (%)**	**Method**	**Result (%)**
Balance	Train	*GA-with-HC*	**97.07** **	*Ensemble9*	94.63
	Test	*SSGA-GP*	73.05	*Ensemble5*	**73.12** †
Climate	Train	*GA-with-HC*	**98.81** **	*Ensemble9*	98.20
	Test	*GA-at-end*	94.30	*Ensemble7*	**94.70** †
Ecoli	Train	*GA-with-HC*	85.00	*Ensemble9*	**92.71** **
	Test	*GA-with-HC*	78.98	*Ensemble7*	**82.06** **
Fertility	Train	*GA-after-each-gen*	**97.49** **	*Ensemble9*	96.93
	Test	*GA-at-end-7*	**83.60** †	*Ensemble9*	83.00
Ionosphere	Train	*GA-with-HC*	**98.05** **	*Ensemble9*	96.70
	Test	*SSGA-GP*	90.93	*Ensemble7*	**91.62** †
Iris	Train	*GA-with-HC*	98.99	*Ensemble9*	**99.94** **
	Test	*GA-at-end-5* and *GA-at-end-7*	94.27	*Ensemble7*	**95.07** †
Pima Indians	Train	*SSGA-GP*	**77.77** **	*Ensemble9*	76.32
	Test	*GA-at-end-9*	**69.95** **	*Ensemble5*	68.91
Soybean	Train	*GA-after-each-gen*	**80.58** **	*Ensemble9*	77.21
	Test	*GA-after-each-gen*	**74.26** **	*Ensemble9*	71.21
WDBC	Train	*GA-at-end-9*	**97.76** †	*Ensemble9*	97.75
	Test	*GA-with-HC*	95.29	*Ensemble7*	**95.89** **

Table 11.26: Comparison between the hybrid ensemble methods with the ensemble construction methods. The best training and test result for each data set is highlighted in bold. For each training and test set, a "**" denotes that the best result is statistically significant when compared to the other result, and a "†" denotes statistical insignificance.

hybrid approach obtained the best result on 3 test sets of which 2 were statistically significant. The ensemble construction approach obtained the best result on 2 training sets, both of which were statistically significant. Additionally, the ensemble construction approach obtained the best result on 6 test sets of which 2 were statistically significant. The results reveal that no single method can obtain the best result across all of the different data sets. *GA-after-each-gen* on the *Soybean* data set was however able to obtain the best result for both the training and test set. The results obtained from the hybridisation of GA with GP obtained more statistically significant results. The proposed hybridisation methods were able to obtain a greater number of best results on the training data, whereas the proposed ensemble construction methods obtained a greater number of best results on the test data.

11.7 Conclusion

This chapter presented and discussed the results obtained for the proposed GP and data classification investigations. The first investigation was to incorporate discretisation into the GP algorithm. Eight methods were investigated and based on the findings, GPEI with arity 2 produced the overall best results and EWI with arity 2 performed the weakest overall. The chapter then presented the results obtained by the 5 GP representations for binary data classification. The findings revealed that decision trees performed the best, and that arithmetic-without-if obtained the weakest results. This was followed by a discussion on the performance of the 2 proposed encapsulation methods. The results indicated that including the encapsulation GO leads to an improvement in classification accuracy. The initial encapsulation approach obtained the highest overall average on the training data, whereas the selective encapsulation approach performed the best on the test data. The results for the 4 GA and GP hybrid methods were then discussed. The results revealed that *GA-with-HC* obtained the best performance on the training data, and that *GA-at-end* with size 7 obtained the best test accuracy. *GA-at-end* with a size of 3 performed the weakest on the training data, and *SSGA-GP* performed the weakest on the test data. Finally, the performance of the ensemble construction methods were analysed. When compared to standard GP, *ensemble9* obtained the best overall performance, and amongst the proposed ensemble construction methods, *ensemble3* obtained the weakest overall performance. Furthermore, a comparison between the hybridisation methods and the ensemble construction methods was conducted. The findings revealed that the hybrid methods performed better on the training data, and the ensemble construction methods performed better on the test data. The next chapter lists the objectives of this dissertation and discusses how the objectives were met based on the findings reported in this chapter.

Chapter 12

Conclusions and Future Work

This chapter summarises the findings of this dissertation and provides a conclusion to each of the different areas of GP and data classification which have been researched. The objectives were discussed in chapter 1 and each section below discusses how each objective was met, concludes the findings of each objective, and provides details on how each objective can be extended for future investigations. The objectives of this dissertation are listed below:

- Objective 1: Incorporating discretisation into GP.
- Objective 2: GP representations for binary data classification.
- Objective 3: Creating an encapsulation genetic operator for data classification.
- Objective 4: Hybridising evolutionary algorithms for classifier ensembles.
- Objective 5: Creating a GP ensemble construction method.

12.1 Objective 1 - GP Discretisation

Several experiments were performed in order to determine whether incorporating discretisation into the GP algorithm would be successful. Two major approaches were proposed: EWI, which created intervals of equal width based on the attribute data, and GPEI, which allowed GP to evolve the intervals during the evolutionary process. Furthermore, two methods for selecting the arity of the nodes were proposed. The fixed arity approach set a constant arity for all the nodes within the GP trees, and the varying arity approach which allowed GP to select the arity of the node during the evolutionary process. The proposed approaches were tested on 12 publicly available data sets. GPEI performed better than EWI indicating that evolving the intervals during the evolutionary process yields better results than

using fixed intervals which are not altered. The results revealed that the GPEI with arity 2 performed well and obtained competitive classification accuracies when compared to other discretisation methods found in the literature. GPEI with fixed arity 2 performed the best overall when compared to the other proposed methods, and thus discretisation can successfully be incorporated into the GP algorithm using that approach. The results therefore revealed that randomly altering the intervals during the execution of the GP algorithm can improve the classification accuracy of the classifiers evolved.

Instead of replacing the intervals of the nodes within the trees with new ones as is done in the proposed GPEI method, future work will determine whether altering the arity of existing nodes during the execution of the algorithm can improve the performance of the algorithm.

12.2 Objective 2 - GP Representations for Binary Classification

This study compared the performance of three major representations for GP and binary data classification. A total of five variations of these representations were proposed in order to determine which representation and function set would yield the best results. The representations were tested on 10 publicly available binary data sets.

Decision trees obtained the highest overall average when compared to the other GP representations investigated; however, this representation did not always obtain the best result. The results revealed that there was an improvement in overall accuracy when the *if* function was included in the function set for arithmetic trees. Furthermore, the results revealed that adding the *between* function in the function set for logical trees can improve the overall accuracy of this representation.

The rationale behind this study was that researchers did not provide sufficient justification for their choice of representations. This study empirically showed that researchers investigating the domain of GP and binary data classification can select any of the three major representations, and obtain good results on both the training and test set.

There was no consistency between the arithmetic and logical representations. When one performed well the other did not, and thus future research will aim at determining the cause for such an observation. Future research will also include extending this study to multiclass classification problems in order to determine which representation is the most suitable.

12.3 Objective 3 - GP Encapsulation

This study served as an investigation into the effects of the encapsulation GO for GP in the context of data classification. Two encapsulation methods were investigated, the first made use of the encapsulation operator without any restrictions on how the GP algorithm could use the encapsulated terminals. The second approach, selective encapsulation, made use of a maintained list of encapsulated terminals, and the GP algorithm selected terminals from the maintained list with a 60% probability. The two proposed methods were tested on 12 publicly available data sets. The findings showed that both of the proposed methods improved the average training accuracy, and that selective encapsulation improved the average test accuracy when compared to a standard GP approach without encapsulation.

The outcome of this study showed that the inclusion of the encapsulation operator has the potential to improve the classification accuracies of models created using GP, and thus suggesting that GP modularisation is a promising area of investigation for future research. Future research includes investigating the effect of the encapsulation GO when arithmetic and logical trees are used, as this study only investigated the use of encapsulation on GP decision trees. In this study, the list was updated based on the number of times the encapsulated terminals were called, and thus future research will include an investigation on updating the list based on the performance of the encapsulated terminals, instead of the number of times they are called within the population.

12.4 Objective 4 - Hybridising GA and GP

This study investigated the hybridisation of a GA with a GP algorithm for creating classifier ensembles. Four methods were proposed, and were tested on 12 publicly available data sets. The first approach, *GA-at-end*, being the most simple of the four, showed that by executing a GA on the final GP population, it is possible to create ensembles which obtain higher classification accuracies than the best standard GP individual. The training results obtained by *GA-at-end* outperformed the single best GP tree on all the data sets except for 2. Standard GP outperformed *GA-at-end* on 4 data sets.

The results for *GA-after-each-gen* also reveals the effectiveness of running a GA for a small number of generations after each GP generation. *GA-with-HC* improved the training and test results when compared to *GA-after-each-gen*. *SSGA-GP* did not perform as well as the other three ensemble methods. The best ensemble method for each data set was compared to other GP and ensemble methods found in literature. The findings revealed that the proposed hybrid methods are able to produce

competitive results when compared to other state-of-the-art ensemble methods.

The four proposed methods for creating ensembles provide an alternative approach to creating classifier ensembles which do not make use of bagging or boosting approaches. This study shows that a simplistic approach without the need for weighted votes can improve the classification results. Based on the findings from this study, it is possible for a researcher to hybridise GA and GP, and as a consequence improve the accuracy of the evolved classifiers. Future work will examine the effect of using different tree representations simultaneously within a single chromosome, thus permitting the chromosome to be constructed using various GP representations.

12.5 Objective 5 - GP Ensemble Construction

This study proposed a GP ensemble construction method which evolved only one ensemble throughout the evolutionary process. This differed from the fourth objective since the hybrid approaches evolved a population of ensembles. Three ensemble sizes were examined in order to determine the effect that the ensemble size has on the performance of the algorithm. Weights were allocated to the training instances so that the algorithm could focus on correctly classifying instances of data which were more challenging. The proposed ensemble was tested on 12 publicly available data sets. The findings revealed that for all sizes, the proposed ensemble method outperformed standard GP on the training and test data. Furthermore, the results indicated that a larger ensemble can obtain better training accuracy than a smaller one. The results revealed that an ensemble size of 9 yielded the best overall results. The hybrid approaches, from the previous objective, were compared to the ensemble construction approach, and the results indicated that the hybridised approaches obtained a greater number of statistically significant results than the ensemble construction method. Furthermore, the hybridised approaches performed better on the training data, whereas the ensemble construction methods performed better on the test data. Similar to the future work for the 4th objective, future research for this study will investigate the use of creating ensembles using more than one GP representation.

12.6 Conclusion

This chapter described how the objectives of this dissertation were met. For each objective, several methods were proposed and where possible, the proposed methods were compared to existing GP methods. This dissertation investigated areas of GP and data classification which have not previously been researched.

Dealing with missing values, feature selection, imbalanced data sets, and mixed

attributes are issues which have generally been addressed on their own. Future research will include proposing a GP algorithm which will be able to cater for all the previously described issues in a single GP run. This will ultimately enable the algorithm to be able to cater for any data set. Furthermore, proposing new parallel architectures for GP and data classification will be researched in order to speed up the evolutionary process.

Bibliography

[1] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming.* Lulu Enterprises, UK Ltd, 2008.

[2] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin, *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[3] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection.* Cambridge, MA, USA: MIT Press, 1992.

[4] P. J. Angeline and J. Pollack, "Evolutionary module acquisition," in *Proceedings of the second annual conference on evolutionary programming*, pp. 154–163, 1993.

[5] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 3rd ed., 2011.

[6] T. Fawcett, "An introduction to roc analysis," *Pattern Recogn. Lett.*, vol. 27, pp. 861–874, June 2006.

[7] T. Borovicka, M. Jirina Jr, P. Kordik, and M. Jirina, "Selecting representative data sets," *Advances in Data Mining Knowledge Discovery and Applications. Intech*, 2012.

[8] A. Shali, M. Kangavari, and B. Bina, "Using genetic programming for the induction of oblique decision trees," in *Machine Learning and Applications, 2007. ICMLA 2007. Sixth International Conference on*, pp. 38–43, Dec 2007.

[9] I. De Falco, A. Della Cioppa, and E. Tarantino, "Discovering interesting classification rules with genetic programming," *Applied Soft Computing*, vol. 1, no. 4, pp. 257–269, 2002.

[10] M. Bramer, *Principles of data mining.* Springer, 2007.

[11] H. Jabeen and A. R. Baig, "Review of classification using genetic programming," *International journal of engineering science and technology*, vol. 2, no. 2, pp. 94–103, 2010.

[12] S. Luke and L. Panait, "A survey and comparison of tree generation algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pp. 81–88, 2001.

[13] H. Jabeen and A. R. Baig, "Depthlimited crossover in gp for classifier evolution," *Comput. Hum. Behav.*, vol. 27, pp. 1475–1481, Sept. 2011.

[14] E. Galvan-Lopez, J. M. Swafford, M. O'Neill, and A. Brabazon, "Evolving a ms. pacman controller using grammatical evolution," in *Applications of Evolutionary Computation*, vol. 6024 of *Lecture Notes in Computer Science*, pp. 161–170, Springer Berlin Heidelberg, 2010.

[15] D. J. Montana, "Strongly typed genetic programming," *Evolutionary computation*, vol. 3, no. 2, pp. 199–230, 1995.

[16] K. E. Kinnear, Jr., ed., *Advances in Genetic Programming.* Cambridge, MA, USA: MIT Press, 1994.

[17] J. R. Koza, *Genetic programming II: automatic discovery of reusable programs.* Cambridge, MA, USA: MIT Press, 1994.

[18] S. Luke and L. Panait, "A comparison of bloat control methods for genetic programming," *Evol. Comput.*, vol. 14, pp. 309–344, Sept. 2006.

[19] S. Harding and W. Banzhaf, "Fast genetic programming on gpus," in *Proceedings of the 10th European Conference on Genetic Programming*, EuroGP'07, (Berlin, Heidelberg), pp. 90–101, Springer-Verlag, 2007.

[20] V. Ciesielski and D. Mawhinney, "Prevention of early convergence in genetic programming by replacement of similar programs," in *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, vol. 1, pp. 67–72, May 2002.

[21] E. Alpaydin, *Introduction to Machine Learning.* The MIT Press, 2nd ed., 2010.

[22] S. Sakprasat and M. Sinclair, "Classification rule mining for automatic credit approval using genetic programming," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pp. 548–555, Sept 2007.

[23] H. Al-Sahaf, A. Song, K. Neshatian, and M. Zhang, "Two-tier genetic programming: towards raw pixel-based image classification," *Expert Systems with Applications*, vol. 39, no. 16, pp. 12291 – 12301, 2012.

[24] K. Tan, Q. Yu, C. Heng, and T. Lee, "Evolutionary computing for knowledge discovery in medical diagnosis," *Artif. Intell. Med.*, vol. 27, pp. 129–154, Feb. 2003.

[25] A. Teredesai and V. Govindaraju, "Issues in evolving gp based classifiers for a pattern recognition task," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 1, pp. 509–515 Vol.1, June 2004.

[26] K. J. Cios, L. A. Kurgan, W. Pedrycz, and R. W. Swiniarski, *Data Mining: A Knowledge Discovery Approach.* Springer Science+ Business Media, LLC, 2007.

[27] K. Bache and M. Lichman, "UCI machine learning repository," 2013.

[28] A. Karahoca, *Advances in Data Mining Knowledge Discovery and Applications.* InTech, 2012.

[29] R. Longadge, S. Dongre, and L. Malik, "Class imbalance problem in data mining: Review," *International Journal of Computer Science and Network*, vol. 2, no. 1, 2013.

[30] E. Fix and J. L. Hodges Jr, "Discriminatory analysis-nonparametric discrimination: consistency properties," tech. rep., DTIC Document, 1951.

[31] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowl. Inf. Syst.*, vol. 14, pp. 1–37, Dec. 2007.

[32] Z. Voulgaris and G. D. Magoulas, "Extensions of the k nearest neighbour methods for classification problems," in *Proceedings of the 26th IASTED International Conference on Artificial Intelligence and Applications*, AIA '08, (Anaheim, CA, USA), pp. 23–28, ACTA Press, 2008.

[33] Y. Bao, N. Ishii, and X. Du, "Combining multiple k-nearest neighbor classifiers using different distance functions," in *Intelligent Data Engineering and Automated Learning - IDEAL 2004* (Z. Yang, H. Yin, and R. Everson, eds.), vol. 3177 of *Lecture Notes in Computer Science*, pp. 634–641, Springer Berlin Heidelberg, 2004.

[34] H. Parvin, H. Alizadeh, and B. Minati, “A modification on k-nearest neighbor classifier,” *Global Journal of Computer Science and Technology*, vol. 10, no. 14, 2010.

[35] N. Suguna and K. Thanushkodi, “An improved k-nearest neighbor classification using genetic algorithm,” *International Journal of Computer Science Issues*, vol. 7, no. 2, pp. 18–21, 2010.

[36] A. Bharathi and E. Deepankumar, “Survey on classification techniques in data mining,” *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 2, pp. 1983–1986.

[37] S. Kotsiantis, “Decision trees: a recent overview,” *Artificial Intelligence Review*, vol. 39, no. 4, pp. 261–283, 2013.

[38] J. R. Quinlan, “Induction of decision trees,” *Mach. Learn.*, vol. 1, pp. 81–106, Mar. 1986.

[39] J. R. Quinlan, *C4.5: Programs for Machine Learning.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[40] R. Barros, M. Basgalupp, A. C. P. L. F. De Carvalho, and A. Freitas, “A survey of evolutionary algorithms for decision-tree induction,” *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 42, pp. 291–312, May 2012.

[41] T. M. Mitchell, *Machine Learning.* New York, NY, USA: McGraw-Hill, Inc., 1 ed., 1997.

[42] I. Rish, “An empirical study of the naive bayes classifier,” in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, pp. 41–46, 2001.

[43] G. Zhang, “Neural networks for classification: a survey,” *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 30, pp. 451–462, Nov 2000.

[44] S. Haykin, *Neural Networks: A Comprehensive Foundation.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd ed., 1998.

[45] P. Jeatrakul and K. Wong, “Comparing the performance of different neural networks for binary classification problems,” in *Natural Language Processing, 2009. SNLP ’09. Eighth International Symposium on*, pp. 111–115, Oct 2009.

[46] S. B. Kotsiantis, “Supervised machine learning: A review of classification techniques,” in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, (Amsterdam, The Netherlands, The Netherlands), pp. 3–24, IOS Press, 2007.

[47] M. Aly, “Survey on multiclass classification methods,” tech. rep., Caltech, USA, 2005.

[48] T. N. Phyu, “Survey of classification techniques in data mining,” in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, vol. 1, pp. 18–20, 2009.

[49] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing.* SpringerVerlag, 2003.

[50] A. Freitas, “A review of evolutionary algorithms for data mining,” in *Soft Computing for Knowledge Discovery and Data Mining* (O. Maimon and L. Rokach, eds.), pp. 79–111, Springer US, 2008.

[51] A. A. Freitas, “A survey of evolutionary algorithms for data mining and knowledge discovery,” pp. 819–845, New York, NY, USA: Springer-Verlag New York, Inc., 2002.

[52] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[53] H. Liu and H. Motoda, *Computational Methods of Feature Selection.* Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, Taylor & Francis, 2007.

[54] Y. Saeys, I. Inza, and P. Larrañaga, “A review of feature selection techniques in bioinformatics,” *bioinformatics*, vol. 23, no. 19, pp. 2507–2517, 2007.

[55] H. Liu and L. Yu, “Toward integrating feature selection algorithms for classification and clustering,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, no. 4, pp. 491–502, 2005.

[56] P. Pujari and J. B. Gupta, “Improving classification accuracy by using feature selection and ensemble model,” *International Journal of Soft Computing*, vol. 2.

[57] J. Yang and V. Honavar, "Feature subset selection using a genetic algorithm," in *Feature extraction, construction and selection*, pp. 117–136, Springer, 1998.

[58] R. R. Jacob, Shomona Gracia, "Discovery of knowledge patterns in clinical data through data mining algorithms: Multi-class categorization of breast tissue data," *International Journal of Computer Applications*, vol. 1, 2011.

[59] K. Kira and L. A. Rendell, "A practical approach to feature selection," in *Proceedings of the ninth international workshop on Machine learning*, ML92, (San Francisco, CA, USA), pp. 249–256, Morgan Kaufmann Publishers Inc., 1992.

[60] I. Kononenko, "Estimating attributes: Analysis and extensions of relief," in *Machine Learning: ECML-94* (F. Bergadano and L. Raedt, eds.), vol. 784 of *Lecture Notes in Computer Science*, pp. 171–182, Springer Berlin Heidelberg, 1994.

[61] P. E. McKnight, K. M. McKnight, S. Sidani, and A. J. Figueredo, *Missing data: A gentle introduction.* The Guilford Press, 2007.

[62] E. Acuna and C. Rodriguez, "The treatment of missing values and its effect on classifier accuracy," in *Classification, Clustering, and Data Mining Applications*, pp. 639–647, Springer, 2004.

[63] B. M. Marlin, *Missing data problems in machine learning.* PhD thesis, University of Toronto, 2008.

[64] P. D. Allison, *Missing data*, vol. 136. Sage publications, 2001.

[65] B. E. T. H. Twala, M. C. Jones, and D. J. Hand, "Good methods for coping with missing data in decision trees," *Pattern Recogn. Lett.*, vol. 29, pp. 950–956, May 2008.

[66] B. Twala, "An empirical comparison of techniques for handling incomplete data using decision trees," *Appl. Artif. Intell.*, vol. 23, pp. 373–405, May 2009.

[67] T. G. Dietterich, "Ensemble methods in machine learning," in *Proceedings of the First International Workshop on Multiple Classifier Systems*, MCS '00, (London, UK, UK), pp. 1–15, Springer-Verlag, 2000.

[68] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, no. 1-2, pp. 1–39, 2010.

[69] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[70] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[71] R. E. Schapire, "The strength of weak learnability," *Machine Learing*, vol. 5, pp. 197–227, July 1990.

[72] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," in *Machine Learning: Proceedings of the Thirteenth International Conference*, pp. 148–156, MORGAN KAUFMANN PUBLISHERS, INC., 1996.

[73] B. Liu, B. McKay, and H. Abbass, "Improving genetic classifiers with a boosting algorithm," in *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, vol. 4, pp. 2596–2602 Vol.4, Dec 2003.

[74] S. Garcia, J. Luengo, J. Saez, V. Lopez, and F. Herrera, "A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 25, no. 4, pp. 734–750, 2013.

[75] H. Liu, F. Hussain, C. L. Tan, and M. Dash, "Discretization: An enabling technique," *Data mining and knowledge discovery*, vol. 6, no. 4, pp. 393–423, 2002.

[76] R. Kerber, "Chimerge: discretization of numeric attributes," in *Proceedings of the tenth national conference on Artificial intelligence*, AAAI'92, pp. 123–128, AAAI Press, 1992.

[77] J. R. Quinlan, *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[78] H. Liu and R. Setiono, "Feature selection via discretization," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 9, no. 4, pp. 642–645, 1997.

[79] F. E. Tay and L. Shen, "A modified chi2 algorithm for discretization," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 3, pp. 666–670, 2002.

[80] M. Hacibeyoglu, A. Arslan, and S. Kahramanli, "Improving classification accuracy with discretization on datasets including continuous valued features," *World Academy of Science, Engineering & Technology*, 2011.

[81] S. Kotsiantis and D. Kanellopoulos, "Discretization techniques: A recent survey," *GESTS International Transactions on Computer Science and Engineering*, vol. 32, no. 1, pp. 47–58, 2006.

[82] J. Bacardit and J. M. Garrell, “Evolution of multi-adaptive discretization intervals for a rule-based genetic learning system,” in *Advances in Artificial Intelligence–IBERAMIA 2002*, pp. 350–360, Springer, 2002.

[83] U. Fayyad and K. Irani, “Multi-interval discretization of continuous-valued attributes for classification learning,” in *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp. 1022–1029, 1993.

[84] J. Bacardit and J. M. Garrell, “Evolving multiple discretizations with adaptive intervals for a pittsburgh rule-based learning classifier system,” in *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartII*, GECCO’03, (Berlin, Heidelberg), pp. 1818–1831, Springer-Verlag, 2003.

[85] J. Aguilar-Ruiz, J. Bacardit, and F. Divina, “Experimental evaluation of discretization schemes for rule induction,” in *Genetic and Evolutionary Computation–GECCO 2004*, pp. 828–839, Springer, 2004.

[86] J. Alcalá-Fdez, L. Sánchez, S. García, M. Jesus, S. Ventura, J. Garrell, J. Otero, C. Romero, J. Bacardit, V. Rivas, J. Fernández, and F. Herrera, “Keel: a software tool to assess evolutionary algorithms for data mining problems,” *Soft Computing*, vol. 13, pp. 307–318, Oct. 2008.

[87] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The weka data mining software: an update,” *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, Nov. 2009.

[88] J. Alcalá-Fdez, A. Fernández, J. Luengo, J. Derrac, S. García, L. Sánchez, and F. Herrera, “Keel data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework,” *Journal of Multiple-Valued Logic and Soft Computing*, 2010.

[89] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[90] A. A. Freitas, *Data mining and knowledge discovery with evolutionary algorithms*. Springer, 2002.

[91] G. Tur and H. A. Guvenir, “Decision tree induction using genetic programming,” in *Proceedings of the Fifth Turkish Symposium on Artificial Intelligence and Neural Networks*.

[92] P. Espejo, S. Ventura, and F. Herrera, "A survey on the application of genetic programming to classification," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 40, no. 2, pp. 121–144, 2010.

[93] M. Shirasaka, Q. Zhao, O. Hammami, K. Kuroda, and K. Saito, "Automatic design of binary decision trees based on genetic programming," in *Proc. The Second Asia-Pacific Conference on Simulated Evolution and Learning (SEAL'98*, 1998.

[94] J. R. Koza, "Concept formation and decision tree induction using the genetic programming paradigm," in *Parallel Problem Solving from Nature*, pp. 124–128, Springer, 1991.

[95] T. Khoshgoftaar, N. Seliya, and Y. Liu, "Genetic programming-based decision trees for software quality classification," in *Tools with Artificial Intelligence, 2003. Proceedings. 15th IEEE International Conference on*, pp. 374–383, Nov 2003.

[96] X. Wang, F. Buontempo, A. Young, and D. Osborn, "Induction of decision trees using genetic programming for modelling ecotoxicity data: adaptive discretization of real-valued endpoints," *SAR and QSAR in Environmental Research*, vol. 17, no. 5, pp. 451–471, 2006.

[97] J. K. Estrada-Gil, J. C. Fernández-López, E. Hernández-Lemus, I. Silva-Zolezzi, A. Hidalgo-Miranda, G. Jiménez-Sánchez, and E. E. Vallejo-Clemente, "Gpdti: A genetic programming decision tree induction method to find epistatic effects in common complex diseases," *Bioinformatics*, vol. 23, no. 13, pp. i167–i174, 2007.

[98] M. Bot and W. Langdon, "Application of genetic programming to induction of linear classification trees," in *Genetic Programming* (R. Poli, W. Banzhaf, W. Langdon, J. Miller, P. Nordin, and T. Fogarty, eds.), vol. 1802 of *Lecture Notes in Computer Science*, pp. 247–258, Springer Berlin Heidelberg, 2000.

[99] H. Etemadi, A. A. Anvary Rostamy, and H. F. Dehkordi, "A genetic programming model for bankruptcy prediction: Empirical evidence from iran," *Expert Systems with Applications*, vol. 36, no. 2, pp. 3199–3207, 2009.

[100] H. Gray, R. Maxwell, I. Martinez-Perez, C. Arus, and S. Cerdan, "Genetic programming for classification of brain tumours from nuclear magnetic resonance biopsy spectra," *Genetic Programming*, p. 424, 1996.

[101] U. Bhowan, M. Johnston, M. Zhang, and X. Yao, "Evolving diverse ensembles using genetic programming for classification with unbalanced data," *Evolutionary Computation, IEEE Transactions on*, vol. 17, pp. 368–386, June 2013.

[102] R. Poli, N. F. McPhee, and L. Vanneschi, "Elitism reduces bloat in genetic programming," in *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, (New York, NY, USA), pp. 1343–1344, ACM, 2008.

[103] K. Hennessy, M. G. Madden, J. Conroy, and A. G. Ryder, "An improved genetic programming technique for the classification of raman spectra," *Know.-Based Syst.*, vol. 18, pp. 217–224, Aug. 2005.

[104] X. Li and V. Ciesielski, "Using loops in genetic programming for a two class binary image classification problem," in *Proceedings of the 17th Australian joint conference on Advances in Artificial Intelligence*, AI'04, (Berlin, Heidelberg), pp. 898–909, Springer-Verlag, 2004.

[105] B. Garcia, R. Aler, A. Ledezma, and A. Sanchis, "Genetic programming for predicting protein networks," in *Advances in Artificial Intelligence - IBERAMIA 2008* (H. Geffner, R. Prada, I. Machado Alexandre, and N. David, eds.), vol. 5290 of *Lecture Notes in Computer Science*, pp. 432–441, Springer Berlin Heidelberg, 2008.

[106] D. Agnelli, A. Bollini, and L. Lombardi, "Image classification: an evolutionary approach," *Pattern Recogn. Lett.*, vol. 23, pp. 303–309, Jan. 2002.

[107] K. Topon and H. Iba, "Classification of scleroderma and normal biopsy data and identification of possible biomarkers of the disease," in *Computational Intelligence and Bioinformatics and Computational Biology, 2006. CIBCB'06. 2006 IEEE Symposium on*, pp. 1–6, IEEE, 2006.

[108] F. d. L. Arcanjo, G. L. Pappa, P. V. Bicalho, W. Meira Jr, and A. S. da Silva, "Semi-supervised genetic programming for classification," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pp. 1259–1266, ACM, 2011.

[109] M. Zhang and P. Wong, "Genetic programming for medical classification: a program simplification approach," *Genetic Programming and Evolvable Machines*, vol. 9, no. 3, pp. 229–255, 2008.

[110] J. Fitzgerald and C. Ryan, "Exploring boundaries: optimising individual class boundaries for binary classification problem," in *Proceedings of the fourteenth*

international conference on Genetic and evolutionary computation conference, GECCO '12, (New York, NY, USA), pp. 743–750, ACM, 2012.

[111] D. Muni, N. Pal, and J. Das, "A novel approach to design classifiers using genetic programming," *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 2, pp. 183–196, 2004.

[112] J. Kishore, L. Patnaik, V. Mani, and V. Agrawal, "Application of genetic programming for multicategory pattern classification," *Evolutionary Computation, IEEE Transactions on*, vol. 4, pp. 242–258, Sep 2000.

[113] T. Loveard and V. Ciesielski, "Representing classification problems in genetic programming," in *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, vol. 2, pp. 1070–1077, IEEE, 2001.

[114] M. Zhang and V. Ciesielski, "Genetic programming for multiple class object detection," in *Advanced Topics in Artificial Intelligence*, pp. 180–192, Springer, 1999.

[115] W. Smart and M. Zhang, "Classification strategies for image classification in genetic programming," in *Proceeding of image and vision computing conference*, pp. 402–407, Palmerston North, New Zealand, 2003.

[116] A. Song, T. Loveard, and V. Ciesielski, "Towards genetic programming for texture classification," in *Proceedings of the 14th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, AI '01, (London, UK, UK), pp. 461–472, Springer-Verlag, 2001.

[117] W. Smart and M. Zhang, "Using genetic programming for multiclass classification by simultaneously solving component binary classification problems," in *Genetic Programming*, pp. 227–239, Springer, 2005.

[118] H. Jabeen and A. R. Baig, "Two-stage learning for multi-class classification using genetic programming," *Neurocomputing*, 2013.

[119] S. Silva and Y.-T. Tseng, "Classification of seafloor habitats using genetic programming," in *Applications of Evolutionary Computing*, vol. 4974 of *Lecture Notes in Computer Science*, pp. 315–324, Springer Berlin Heidelberg, 2008.

[120] B.-C. Chien, J.-y. Lin, and W.-P. Yang, "A classification tree based on discriminant functions," *Journal of information science and engineering*, vol. 22, no. 3, p. 573, 2006.

[121] T. Loveard and V. Ciesielski, "Employing nominal attributes in classification using genetic programming," in *Proceedings of the 4th Asia-Pacific Conference*

on Simulated Evolution And Learning (L. Wang, K. C. Tan, T. Furuhashi, J.-H. Kim, and X. Yao, eds.), pp. 487–491, Nov 2002.

[122] C.-S. Kuo, T.-P. Hong, and C.-L. Chen, "Applying genetic programming technique in classification trees," *Soft Computing*, vol. 11, no. 12, pp. 1165–1172, 2007.

[123] J. Eggermont, A. E. Eiben, and J. I. v. Hemert, "A comparison of genetic programming variants for data classification," in *Proceedings of the Third International Symposium on Advances in Intelligent Data Analysis*, IDA '99, (London, UK, UK), pp. 281–290, Springer-Verlag, 1999.

[124] K. C. Tan, A. Tay, T. H. Lee, and C. M. Heng, "Mining multiple comprehensible classification rules using genetic programming," in *Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress - Volume 02*, CEC '02, (Washington, DC, USA), pp. 1302–1307, IEEE Computer Society, 2002.

[125] C. Bojarczuk, H. Lopes, and A. Freitas, "Genetic programming for knowledge discovery in chest-pain diagnosis," *Engineering in Medicine and Biology Magazine, IEEE*, vol. 19, pp. 38–44, July 2000.

[126] J. Eggermont, J. N. Kok, and W. A. Kosters, "Genetic programming for data classification: partitioning the search space," in *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, (New York, NY, USA), pp. 1001–1005, ACM, 2004.

[127] H. Jabeen and A. R. Baig, "Two layered genetic programming for mixed-attribute data classification," *Appl. Soft Comput.*, vol. 12, pp. 416–422, Jan. 2012.

[128] A. Idris, A. Khan, and Y. S. Lee, "Genetic programming and adaboosting based churn prediction for telecom," in *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pp. 1328–1332, 2012.

[129] R. Thomason and T. Soule, "Novel ways of improving cooperation and performance in ensemble classifiers," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, (New York, NY, USA), pp. 1708–1715, ACM, 2007.

[130] Y. Zhang and S. Bhattacharyya, "Genetic programming in classifying large-scale data: an ensemble method," *Inf. Sci.*, vol. 163, pp. 85–101, June 2004.

[131] G. Folino, C. Pizzuti, and G. Spezzano, "Ensemble techniques for parallel genetic programming based classifiers," in *Genetic Programming* (C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, eds.), vol. 2610 of *Lecture Notes in Computer Science*, pp. 59–69, Springer Berlin Heidelberg, 2003.

[132] G. Folino, C. Pizzuti, and G. Spezzano, "Gp ensembles for large-scale data classification," *Evolutionary Computation, IEEE Transactions on*, vol. 10, pp. 604–616, Oct 2006.

[133] H. Iba, "Bagging, boosting, and bloating in genetic programming," in *Proceedings of the genetic and evolutionary computation conference*, vol. 2, pp. 1053–1060, 1999.

[134] D. A. Augusto, H. J. C. Barbosa, and N. F. F. Ebecken, "Coevolutionary multi-population genetic programming for data classification," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pp. 933–940, ACM, 2010.

[135] T. K. Paul, Y. Hasegawa, and H. Iba, "Classification of gene expression data by majority voting genetic programming classifier," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pp. 2521–2528, IEEE, 2006.

[136] G. Pappa and A. Freitas, "Creating rule ensembles from automatically-evolved rule induction algorithms," in *Advances in Machine Learning I*, vol. 262 of *Studies in Computational Intelligence*, pp. 257–273, Springer Berlin Heidelberg, 2010.

[137] P. Lichodzijewski and M. I. Heywood, "Managing team-based problem solving with symbiotic bid-based genetic programming," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pp. 363–370, ACM, 2008.

[138] J. U. Ryan and H. M. Jason, "Learning classifier systems: A complete introduction, review, and roadmap," *Journal of Artificial Evolution and Applications*, 2009.

[139] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, pp. 273–297, Sept. 1995.

[140] C. Gagné, M. Sebag, M. Schoenauer, and M. Tomassini, "Ensemble learning for free with evolutionary algorithms?," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, (New York, NY, USA), pp. 1782–1789, ACM, 2007.

[141] K.-H. Liu and C.-G. Xu, "A genetic programming-based approach to the classification of multiclass microarray datasets," *Bioinformatics*, vol. 25, no. 3, pp. 331–337, 2009.

[142] J. H. Hong and S. B. Cho, "Ensemble genetic programming for classifying gene expression data," in *Proceedings of the Seventh Asian-Pacific Conference on Complex Systems*, 2004.

[143] D. Nagendra Kumar, S. Satapathy, and J. V. R. Murthy, "A scalable genetic programming multi-class ensemble classifier," in *Nature Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pp. 1201–1206, 2009.

[144] K. Imamura, R. B. Heckendorn, T. Soule, and J. A. Foster, "N-version genetic programming via fault masking," in *Genetic Programming*, pp. 172–181, Springer, 2002.

[145] G. A. Morrison, D. P. Searson, and M. J. Willis, "Using genetic programming to evolve a team of data classifiers," *World Academy of Science, Engineering & Technology*, vol. 72, pp. 261–264, 2011.

[146] S. Hengpraprohm and P. Chongstitvatana, "A genetic programming ensemble approach to cancer microarray data classification," in *Innovative Computing Information and Control, 2008. ICICIC '08. 3rd International Conference on*, pp. 340–340, 2008.

[147] M. Brameier and W. Banzhaf, "Evolving teams of predictors with linear genetic programming," *Genetic Programming and Evolvable Machines*, vol. 2, no. 4, pp. 381–407, 2001.

[148] T. Loveard, *Genetic Programming For Classification Learning Problems.* PhD thesis, Royal Melbourne Institute of Technology, January 2003.

[149] D. S. Moore and G. P. McCabe, *Introduction to the Practice of Statistics.* W.H. Freeman, 1989.

[150] D. F. Groebner, P. W. Shannon, P. C. Fry, and K. D. Smith, *Business Statistics: A Decision Making Approach.* Prentice Hall, 7th ed., 2007.

[151] N. V. Chawla, N. Japkowicz, and A. Kotcz, "Editorial: Special issue on learning from imbalanced data sets," *ACM SIGKDD Explorations Newsletter - Special issue on learning from imbalanced datasets*, vol. 6, pp. 1–6, June 2004.

[152] R. Bellman, R. E. Bellman, R. E. Bellman, and R. E. Bellman, *Adaptive control processes: a guided tour*, vol. 4. Princeton University Press Princeton, 1961.

[153] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly, "Genetic programming needs better benchmarks," in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, (New York, NY, USA), pp. 791–798, ACM, 2012.

[154] J. Bacardit and J. M. Garrell, "Analysis and improvements of the adaptive discretization intervals knowledge representation," in *Genetic and Evolutionary Computation–GECCO 2004*, pp. 726–738, Springer, 2004.

[155] S. B. Kotsiantis and P. E. Pintelas, "Logitboost of simple bayesian classifier," *Informatica*, vol. 29, pp. 53–59, 2005.

Appendices

Appendix A

User Manual

This appendix describes how to run the program.

A.1 Program Requirements

Java must be installed in order to use the program. This can be obtained from http://java.com/en/download/ and once installed the program can be used.

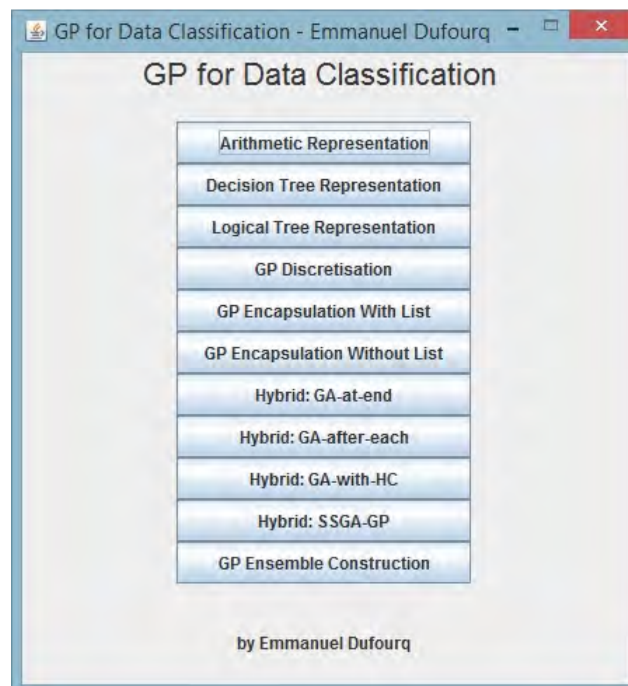


Figure A.1: Main menu.

A.2 Starting the Program

In order to start the program, execute GPforDataClassification.jar which is found on the CD. Once the program has started the main menu will appear as shown in figure A.1.

A.3 Selecting an Experiment to Run

From the main menu, there are 11 options to choose from. Each button on the menu corresponds a proposed method in this dissertation.

- **Arithmetic Representation** - corresponds to the arithmetic representation described in chapter 7.
- **Decision Tree Representation** - corresponds to the decision tree representation described in chapter 7.
- **Logical Tree Representation** - corresponds to the logical tree representation described in chapter 7.
- **GP Discretisation** - corresponds to the proposed GP discretisation methods described in chapter 6.
- **GP Encapsulation With List** - corresponds to the proposed GP encapsulation method with the maintained list described in chapter 8.
- **GP Encapsulation Without List** - corresponds to the proposed GP encapsulation method without the maintained list described in chapter 8.
- **Hybrid: GA-at-end** - corresponds to the proposed GA and GP hybridisation method, *GA-at-end*, described in chapter 9.
- **Hybrid: GA-after-each** - corresponds to the proposed GA and GP hybridisation method, *GA-after-each*, described in chapter 9.
- **Hybrid: GA-with-HC** - corresponds to the proposed GA and GP hybridisation method, *GA-with-HC*, described in chapter 9.
- **Hybrid: SSGA-GP** - corresponds to the proposed GA and GP hybridisation method, *SSGA-GP*, described in chapter 9.
- **GP Ensemble Construction** - corresponds to the proposed GP ensemble construction described in chapter 10.

A.4 Starting an Experiment

Once an experiment has been selected from the main menu, a new window will appear which allows the experimenter to set up the parameters and start the algorithm. The default parameters are set to those used in this dissertation, however, these parameters can be changed. For each experiment all the parameters must be filled in. Furthermore, each experiment can run on one or more threads. The recommended number of threads to use is based on the specifications of the system on which the program is being used. If the program is executed on a computer with 2 physical cores, then a maximum of 4 threads should be used. However, if the computer has 4 physical cores, then a maximum of 8 threads should be used. A greater share of the CPU will be allocated to the program when more threads are used.

A.4.1 Selecting a data set

For each experiment, a data set must be selected, and the corresponding training and test files for that data set must be specified. Figure A.2 illustrates the GP Arithmetic Representation experiment. The *Climate* data set was selected. For training the *climatetrain0.fold1* was selected, and *climatetest0.fold1* was selected as the test set.

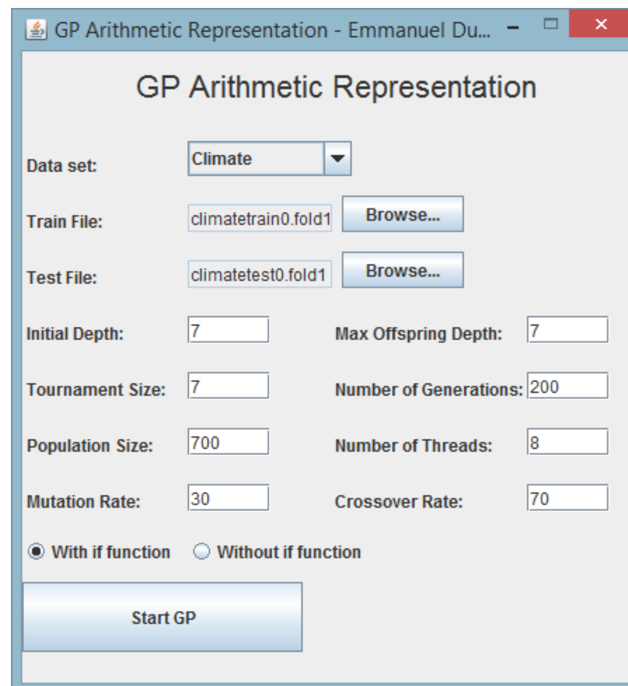


Figure A.2: GP Arithmetic Representation menu.

The training and test files have been supplied on the CD. Each file is in the format

"data_set_name x.fold y" where data_set_name corresponds to the name of the data set, x corresponds to the fold (from 0 to 9 since 10-fold cross-validation is used), and y corresponds to the partition. Ten-fold cross-validation was performed 5 times on each data set which correspond to the 5 partitions. Thus for each experiment, one train and one test must be selected, and the values of x and y must be the same. For example, using the *Climate* data set, the following represent examples of valid selections for the training and test files:

- climatetrain5.fold3 and climatetest5.fold3
- climatetrain2.fold2 and climatetest2.fold2
- climatetrain1.fold3 and climatetest1.fold3
- climatetrain8.fold1 and climatetest8.fold1
- climatetrain9.fold2 and climatetest9.fold2

A.4.2 Executing the experiment

Once the data set has been selected and the parameters entered, the algorithm can be executed by clicked on "Start GP". In order to cancel the execution, click on "Cancel GP Run". A progress bar will appear on the software which indicates the overall progression of the algorithm. Once the algorithm has completed, a popup message will appear as in illustrated in figure A.3. This message indicates the location of the output file. The output file contains information about run, and the last two lines of the output file has the training and test accuracy for the evolved GP classifier.

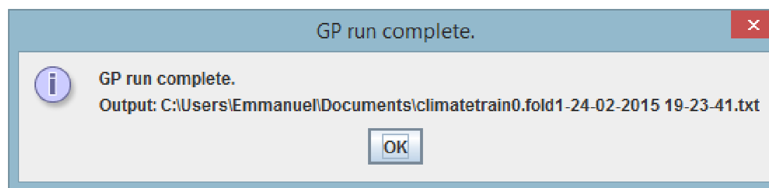


Figure A.3: Popup message which appears at the end of the run.