A Linear Regression Approach to Numerical Simplification in Tree-Based Genetic Programming

Mark Johnston[1], Thomas Liddle[1], and Mengjie Zhang[2]

[1] School of Mathematics, Statistics and Operations Research
Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand
mark.johnston@msor.vuw.ac.nz
liddlethom@msor.vuw.ac.nz
[2] School of Engineering and Computer Science
Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand
mengjie.zhang@ecs.vuw.ac.nz

Abstract. We propose a novel approach to simplification in tree-based Genetic Programming to combat program bloat, based upon numerical relaxations of algebraic rules. We also separate proposal of simplifications (using linear regression, removing redundant children, and replacing small ranges with a constant) from an acceptance criterion that checks the effect of proposed simplifications on the evaluation of training examples, looking several levels up the tree. We test our simplification method on three classification datasets and conclude that the success of linear regression is dataset dependent, that looking further up the tree can catch unwanted bad case simplifications, and that CPU time can be significantly reduced while maintaining classification accuracy on unseen examples.

Victoria University of Wellington
School of Mathematics, Statistics and Operations Research
Research Report Series (ISSN: 1174-2011)
http://msor.victoria.ac.nz/Main/ResearchReportSeries

December 14, 2009

1 Introduction

One problem that limits the effective application of Genetic Programming is *program bloat* [1–4], where program trees tend to grow exponentially in size due to the crossover operator, causing the GP process to be computationally expensive. This is often unnecessary, as trees in the population tend to have some algebraic redundancy, i.e., the mathematical expressions that the trees represent can often be algebraically simplified. Bloat corresponds to "model overfitting" in statistical modelling, i.e., formulating a model that is more complicated than necessary to fit a set of training examples. In addition, program trees are often contrived to make the best use of the available constant values set in the initial population. The usual approach to combating program bloat is to set a maximum depth limit [1] for program trees, thereby "cutting off" potentially useful portions of a tree. This allows the GP process to run in reasonable time, but it is not ideal when it is desired that the GP search covers as much of the search space as possible.

Two distinct approaches to simplification of programs in tree-based GP are the algebraic and numerical approaches. In the *algebraic* approach [5, 6], the rules of algebra are used (in a bottom-up fashion) to directly simplify the mathematical expression that the tree represents. In the *numerical* approach [7, 8], the evaluation of each of the set of training examples are examined to determine if particular subtrees can be approximated by a single constant, removed altogether, or replaced by a smaller subtree. This is similar to "lossy compression" of images and aims for a minimal effect upon the evaluation of training examples.

In this paper, we propose to split the process of simplification into two roles: *proposers* which propose a local change to the program tree; and an *acceptor* which evaluates the proposed local change and determines whether to accept or reject it. The novel aspects are that the proposers use numerical relaxations of algebraic simplification rules, including linear regression, and that the acceptor evaluates the effect of the proposed local change further up the tree to safeguard against myopic decisions. These ideas are developed further in Section 3. We investigate how simplification affects classification performance; in particular considering node-wise (myopic) simplification versus group-wise (local) simplification, investigating sensitivity of program nodes to the significance threshold and propagation effects, and looking for potential bad case examples. We will also study how numerical simplification can actively control program size and achieve parsimony pressure from dynamic simplification versus enforced pruning from a node-count or depth constraint.

Research goals. The overall research goal is to determine how simplification affects classification accuracy and computational effort in controlling program bloat for classification problems. In particular, we wish to balance: the number and severity of simplifications proposed (reduction in tree size or wasted proposals that are not accepted) and the additional workload in evaluating them; how far up the tree should the acceptor look (since looking further up the tree requires increased computational effort but will certainly catch more bad cases); and how often to simplify the population.

Outline of the paper. The remainder of this paper is structured as follows. Section 2 provides background on algebraic and numerical approaches to simplification in GP programs. Section 3 develops our new approach to simplification of GP programs based upon a relaxation of the algebraic rules and separating the roles of simplification proposer and simplification acceptor. Section 4 describes computational experiments on three datasets and Section 5 discusses the results. Finally, Section 6 draws some conclusions and makes recommendations for future research directions.

2 Algebraic and Numerical Approaches to Simplification

In this section we review some existing algebraic and numerical approaches to the simplification of a program in tree-based GP. We consider a simple GP system which includes the basic arithmetic operators: $+$, $-$, $\times$ and protected division $\%$ which returns 0 when the divisor is 0. We also include an `ifpos` operator which returns the middle child if the left child is positive, and otherwise returns the right child.

2.1 Algebraic Simplification

Algebraic simplification of a GP tree involves the *exact* application of the simple rules of algebra to nodes of the tree in order to produce a smaller tree representing an exactly equivalent mathematical expression. Table 1 lists a number of algebraic simplification rules applied to a local node of the program tree. For example, for constants a and c and subtree B, we can replace the subtree $a \times (B\%c)$ with the subtree $b \times B$ where $b = a\%c$ is a new constant node. This can be implemented efficiently using hashing in the finite field $\mathbb{Z}_p$ for prime p [5, 6].

The strength of this approach is that any proposed simplification has *no global* effect on the evaluation of any training example. The weakness is that the rules of algebra are applied exactly, i.e., there is no scope for the rule for approximate equivalence, nor equivalence across the domain of the training examples. There are also some algebraic simplifications that are difficult for a basic set of locally applied algebraic rules to recognise when applied in a bottom-up fashion.

Table 1. Algebraic simplification rules (reproduced from [5]). Lower case letters represent numerical constants, while the upper case letters represent variable/feature terminal nodes or whole subtrees.

Precondition		Result	Precondition		Result
$a+b$	$\rightarrow$	$c, c=a+b$	$a+(b+C)$	$\rightarrow$	$c+C, c=a+b$
$a-b$	$\rightarrow$	$c, c=a-b$	$a+(b-C)$	$\rightarrow$	$c-C, c=a+b$
$a \times b$	$\rightarrow$	$c, c=a \times b$	$a-(b+C)$	$\rightarrow$	$c-C, c=a-b$
$a \div b$	$\rightarrow$	$c, c=a \div b$	$a-(b-C)$	$\rightarrow$	$c+C, c=a-b$
$A \div 1$	$\rightarrow$	A	$a \times (b \times C)$	$\rightarrow$	$c \times C, c=a \times b$
$A \div A$	$\rightarrow$	1	$a \times (b \div C)$	$\rightarrow$	$c \div C, c=a \times b$
$0 \div A$	$\rightarrow$	0	$a \div (b \div C)$	$\rightarrow$	$c \times C, c=a \div b$
$0 \times A = A \times 0$	$\rightarrow$	0	$a+(B+c)$	$\rightarrow$	$b+B, b=a+c$
$A \times 1 = 1 \times A$	$\rightarrow$	A	$a+(B-c)$	$\rightarrow$	$b+B, b=a-c$
$A+0=0+A$	$\rightarrow$	A	$a-(B+c)$	$\rightarrow$	$b-B, b=a-c$
$A-0$	$\rightarrow$	A	$a-(B-c)$	$\rightarrow$	$b-B, b=a+c$
$A-A$	$\rightarrow$	0	$a \times (B \times c)$	$\rightarrow$	$b \times B, b=a \times c$
$A \times \frac{1}{B} = \frac{1}{B} \times A$	$\rightarrow$	$\frac{A}{B}$	$a \times (B \div c)$	$\rightarrow$	$b \times B, b=a \div c$
$A \times \frac{B}{A} = \frac{B}{A} \times A$	$\rightarrow$	B	$a \div (B \div c)$	$\rightarrow$	$b \div B, b=a \times c$

2.2 Numerical Simplification

Numerical simplification of a GP tree involves the replacement of a subtree with a smaller (possibly *approximate*) substitute based upon the *local* effect on the evaluation of the training examples. Two methods previously explored are described below.

Range simplification [7]. In evaluating the training examples, if the range of values a node takes is sufficiently small (less than a *range threshold*), then the node is replaced by a single constant-node (the average value). The strengths of range simplification are that equivalence is based only upon the observed range of the training examples; it deals with nodes that are calculated from constant values; allows for features or subtrees with a very small range of values to be simplified; and it is computationally inexpensive. However, the weakness is that local simplifications can have an adverse effect further up the tree in some cases. Figure 1 shows an example of the potentially bad effect of a local range simplification further up the tree and Appendix D discusses further examples.

Removing redundant children [8]. In evaluating the training examples, if the difference between the values at a parent node and its child are sufficiently small (less than a *redundancy threshold* in this paper) then the parent can be replaced by the child. Song et al [8] use the criterion that the sum of absolute deviations (SAD) be zero over all training examples, i.e.,

$$\sum_i |p_i - c_i| = 0$$

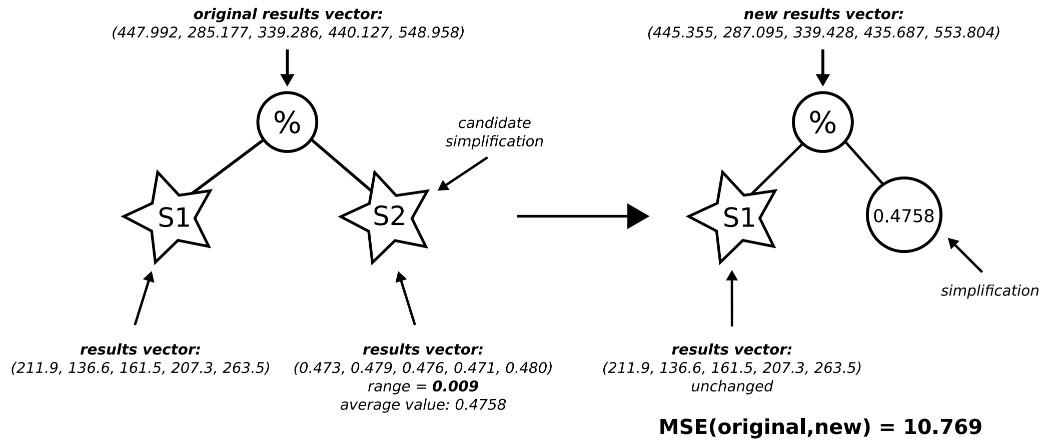


Fig. 1. A bad case example where range simplification causes a significant change to the tree one level up. The left subtree (S1) has relatively large values in its results vector (the evaluation of the subtree on the training examples), and is divided by the right subtree (S2) which has relatively small evaluation values. Even though the range of S2 is only 0.009, the division means the simplification causes a relatively large change to the tree.

where p_i and c_i are the evaluation of the ith training example at the parent and child respectively. This is a slight relaxation of algebraic simplification to the actual range of values taken by the training examples.

3 New Relaxed Approach to Simplification

We propose a new relaxed approach to simplification in which we use numerical evaluation of the training examples to determine if the algebraic rules are approximately satisfied, and then evaluate the numerical effect of any proposed local simplifications further up the tree before accepting them. Hence, we clearly separate the proposal of a local simplification from the acceptance or rejection of the proposal based upon its effect on the numerical evaluation of the training examples. We address the weakness of exact algebraic simplification by covering simple algebraic rules plus more complicated ones, and allow for approximate satisfaction of these rules, thereby extending the possible reduction in tree-size of simplification. We also address the weakness of local numeric simplification by looking further up the tree before accepting a proposed simplification, and therefore catch the bad cases and reject them.

3.1 Proposers

In this paper we use three numerical simplification operators — range simplification and removal of redundant children (as in Section 2.2) and linear regression (described further below) — to numerically evaluate possible algebraic simplifications, relaxing each equality slightly. Between the three operators, we cover almost all the algebraic simplification in Table 1 (except those involving two general subtrees) and some more

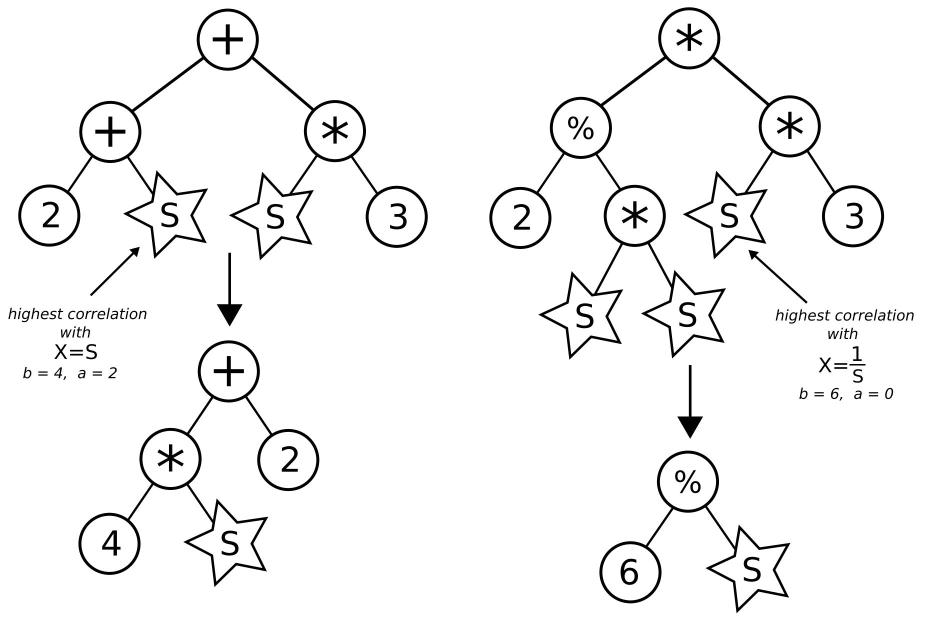


Fig. 2. Simplification examples that are not covered by simple (local) algebraic rules, but are covered by linear regression. Here S represents a particular repeated subtree in each example.

complex ones. We make a small modification to each of the first two operators presented before: for simplicity, we use a constant range threshold for range simplification; and we use mean square error (MSE) for redundancy checking (rather than SAD) for consistency.

Linear regression. Consider the nodes Y and S in a GP tree, where S is a child or grandchild subtree of Y. If we can approximate Y by

$$Y = b \times S + a \tag{1}$$

or

$$Y = b\%S + a \tag{2}$$

sufficiently closely for some constants a and b, then we may be able to significantly reduce the size of the tree. This is an extension of simple algebraic rules and allows for *approximate linearity* of node Y against subtree S (or $\frac{1}{S}$). Figure 2 gives two examples in which linear regression will reduce a tree where other simplification methods do not. A candidate simplification's tree size using this method will be a maximum of $4 + |S|$ nodes, with a possible simplification to $2 + |S|$ under certain conditions on a and b, where $|S|$ is the number of nodes in subtree S. To evaluate linearity, we use Pearson's correlation coefficient. We consider all children and grandchildren of Y as S for simplification and choose the one with the highest value of Pearson's r^2 greater than a *regression threshold*. The proposal is to replace node Y by the simplest version of equa-

Algorithm 1 Pseudocode for Linear Regression Simplification Proposer

```
Require: subtree $Y$
  if node $Y$ is a terminal or has 2 terminal children then
      return subtree $Y$
  else
      calculate $S_{YY}$ and $\bar{Y}$
      for all children and grandchildren of node $Y$ do
          calculate $S_{XX}$, $S_{XY}$ and correlation coefficient $r$
      end for
      for child or grandchild with the highest $r^2$ do
          {either for $X = S$ or $X = 1/S$, giving preference to granchildren in the case of a tie}
          if $r^2$ > regression threshold then
              calculate $a$ and $b$
              return simplest version of new subtree (+(*bX)a) or (+(%bX)a)
          else
              return subtree $Y$
          end if
      end for
  end if
```

tion (1) or (2) as appropriate. For completeness, the formulae for Pearson's correclation coefficient and calculation of the regression equation are given in Appendix A.

An important consideration when implementing the linear regression testing on a particular subtree is how many levels down the tree to go for child comparison (to get S). This design decision is discussed in some detail in Appendix B. As a result, Algorithm 1 specifies which candidate linear regression simplifications to propose.

3.2 Acceptor

In order to check that a proposed simplification won't cause a significant change further up the tree, we compare the results vectors (the evaluation of the subtree on all training examples) of the old and new (simplified) tree. Figure 3 illustrates which nodes are checked against for different values of n. We go to the ancestor node n levels up and calculate the mean square error (MSE) at that node, i.e.,

$$\sum_i (new_i - old_i)^2$$

where old_i and new_i are the original and newly simplified evaluations of the ith training example respectively. If the MSE is less than an *acceptance threshold*, then we accept the simplification and make the change to the tree; if it is not, then we reject the simplification and keep the old tree. In this way we aim to change the tree's fitness as little as possible by catching *bad* simplifications and rejecting them.

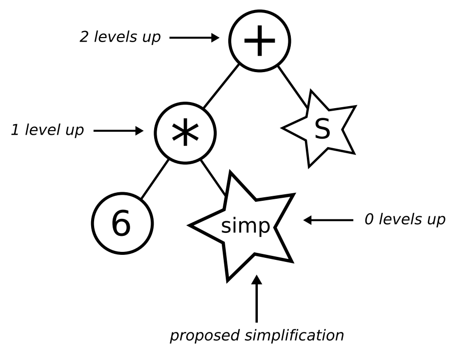


Fig. 3. The acceptor evaluates the effect of a proposed simplification n levels up the tree. Here, arrows point to the node that the MSE calculation applies.

4 Experimental Design

Software package. For our implementation, we modified the RMIT GP (1.5) package [9], written in C++ (see Appendix C for further details). Experiments were carred out on a computational grid (Sun Grid Engine [10]) with each computer having the following specifications: NetBSD (5.0_STABLE) i386; Core 2 Duo 3.0GHz, Intel Q43 (ICH10D) Express Chipset; 4096MB DDR SDRAM.

Datasets. To test our simplification system we ran experiments on three different classification datasets: Coins (14 features, 3 classes, [5, 8]), Wine (13 features, 3 classes, [11]) and Breast-Cancer Wisconsin (9 features, 2 classes, [12]). Coins consists of 600 images (each 64×64 pixels) of five cent pieces against a random noisy background. Wine gives the result of a chemical analysis of Italian wines from three cultivars (the classes).

GP system setup. All experiments were run with the following setup: population size 100, number of generations 100, maximum depth of tree 40 (effectively unlimited), mutation rate 28%, crossover rate 70%, elitism rate 2%. The terminal set consists of the features and random float numbers in the range $[-10, 10]$. The function set consists of $+$, $-$, $\times$, $\%$ (protected division) and `ifpos`. We used *static range selection* [13] to choose the class from the tree output and ten-fold cross validation to evaluate each tree in the population.

Simplification frequency. We perform simplification checks on the whole population every k generations, simplifying the population before the selection process occurs for the next generation. Reductions in tree size will allow more of the search space to be explored over the subsequent generations. We do not simplify the initial population as this may remove too many of the useful "building blocks" present.

Choice of threshold values. For the operators we have implemented there are six different thresholds that we need to test in our experiments: the *proposal thresholds* (range width, redundant MSE and regression r^2); the *acceptance thresholds* (acceptance MSE and the number of levels to look up n); and simplifying the population every k generations.

Preliminary experiments. We performed some initial experiments on the Coins dataset to determine what values of each threshold could be good to test in more extensive experiments on multiple datasets. We ran each configuration of the thresholds on the same 10 random seeds, and only ran these for 50 generations. The values tested were as follows

- Range threshold: $\{0.0001, 0.01, 0.1, 1, 3\}$
- Redundancy threshold: $\{0.0001, 0.01, 0.1, 1, 3\}$
- Regression threshold: $\{0.9999, 0.99, 0.95\}$
- Looking n levels up: $\{0, 1, 2, 3, 4\}$
- Acceptance threshold: $\{0.0001, 0.01, 0.1, 1, 3\}$
- Simplify every k generations: $\{1, 2, 3, 4, 5\}$

Considering all combinations we have $3 \times 5^5 = 9375$ configurations in total.

Preliminary Results. Table 2 shows the results of the different threshold values tested in these experiments. All average test accuracies performed worse than the base system, but this can be understood as each average is over all configurations of the other thresholds, which includes the poor performing value choices (such as 0 levels up). These results indicated that looking 3 or 4 levels up shows no significant benefit, and looking 0 levels up performs poorly. It is useful to include 0 levels up as a comparison however, since this was a key part of our research goals. We also concluded that we wanted to include an even more relaxed regression value (0.8) as well as performing no regression at all, also for comparison. Whilst allowing larger values for the acceptance threshold gave much faster results, it degraded the test accuracy significantly as well, so we chose smaller values for the final experiments.

Table 2. Preliminary experiments results. Average CPU time taken (in seconds) and test classification accuracy (as a proportion) grouped by different thresholds for the Coins dataset. Results for each of the five levels of the range threshold are collected over $3 \times 5^4 = 1875$ combinations of the other five thresholds, etc.

Threshold	Time	(s.d.)	T.Acc	(s.d.)
Base	2.18	0.62	0.8098	0.04

Range Threshold

0.0001	3.00	2.28	0.7761	0.04
0.01	2.99	2.18	0.7761	0.04
0.1	2.96	2.26	0.7770	0.04
1	2.62	1.92	0.7756	0.04
3	2.41	1.96	0.7594	0.05

Regression Threshold

0.9999	2.90	2.20	0.7765	0.04
0.99	2.82	2.15	0.7733	0.04
0.95	2.68	2.07	0.7686	0.04

Acceptance Threshold

0.0001	4.35	3.07	0.7929	0.04
0.01	3.44	2.20	0.7907	0.04
0.1	2.48	1.36	0.7822	0.04
1	1.89	1.05	0.7567	0.04
3	1.83	1.01	0.7415	0.04

Threshold	Time	(s.d.)	T.Acc	(s.d.)
Base	2.18	0.62	0.8098	0.04

Redundancy Threshold

0.0001	3.20	2.26	0.7942	0.03
0.01	3.03	2.11	0.7939	0.03
0.1	2.70	2.05	0.7843	0.03
1	2.56	2.01	0.7500	0.05
3	2.51	2.18	0.7417	0.05

Levels Up

0	1.71	1.19	0.7320	0.06
1	2.91	2.21	0.7813	0.04
2	3.08	2.30	0.7830	0.03
3	3.11	2.19	0.7838	0.03
4	3.19	2.22	0.7840	0.03

Simplify Every k Generations

1	4.17	3.31	0.7720	0.05
2	2.80	1.92	0.7724	0.04
3	2.51	1.55	0.7752	0.04
4	2.35	1.36	0.7732	0.04
5	2.16	1.19	0.7713	0.04

5 Results and Discussion

Preliminary experiments suggested a reasonable range of values of each threshold. The set of values for each threshold used in our more extensive experiments are as follows.

- Range: $\{0.1, 0.5, 1.0\}$
- Redundancy: $\{0.01, 0.05, 0.1\}$
- Regression: $\{\text{none}, 0.99, 0.95, 0.80\}$
- Looking n levels up: $\{0, 1, 2\}$
- Acceptance: $\{0.01, 0.05, 0.1\}$
- Simplify every k generations: $\{3, 4, 5\}$

Considering all combinations we have $3^5 \times 4 = 972$ configurations in total, and we ran each configuration on the same set of 100 random seeds.

We now discuss the results of the final experiments in terms of the research goals outlines in Section 1. Over the following several pages, Table 3 summarises the experimental results for each dataset, and Figures 4–12 show graphical representations of the performance of each configuration, colour coded by the values of the different thresholds, each graph pertaining to one threshold. The base result is a standard GP with no simplification (and recall that the maximum tree depth is 40), for comparison with all other results.

5.1 Classification Accuracy vs Computational Effort

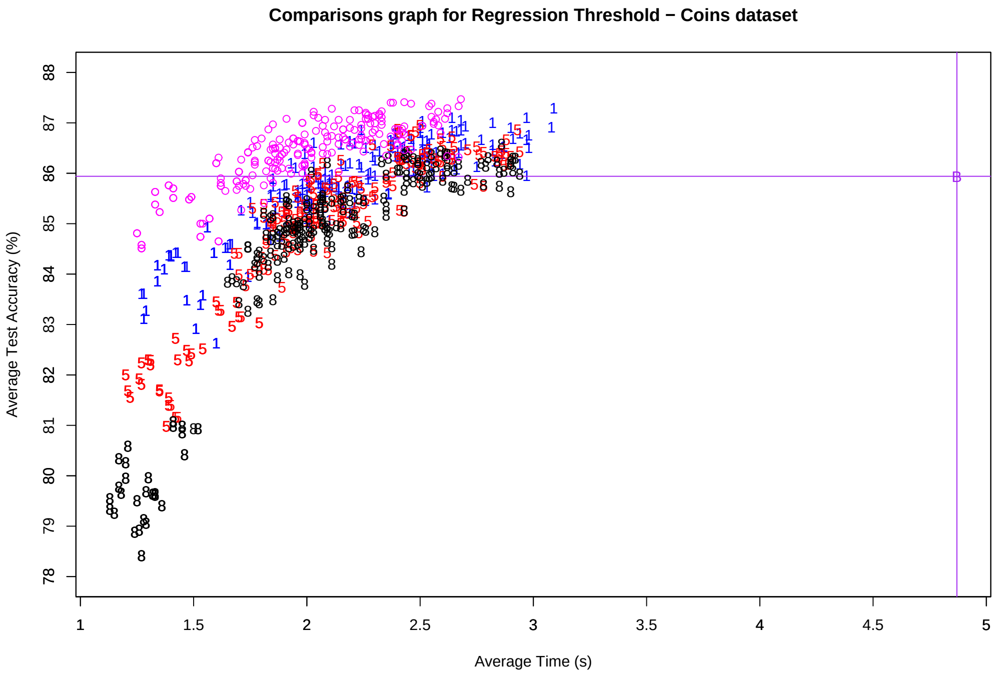
All datasets performed differently in our tests. Regarding average test accuracy, the Coins dataset fluctuated greatly over all configurations, some performing much worse than the base system, but some also significantly better (see the top graph in Figure 4). On the other hand, the Wisconsin dataset's average test accuracy is virtually unchanged in the range $[95.22\%, 95.71\%]$, while the Wine dataset is at least 8–9% worse than the base system. When considering computational effort (CPU time), all datasets show significant savings. The biggest 'reasonable' time savings (meaning not too much degradation in test accuracy) for the Coins dataset is approximately 75% savings, Wisconsin roughly 60%, and Wine around 35%. The Wine dataset runs so quickly, however, that changes in CPU time are difficult to measure accurately, and the time taken across all configurations varies within approximately 0.1 of a second.

5.2 Proposal and Acceptance Thresholds

In general as we increase the value of each of the range width, redundant MSE and acceptance MSE thresholds, thereby relaxing the approximation, CPU time goes down (Wine stays constant however), but so does average test accuracy (except for Coins

Table 3. Average CPU time taken (in seconds) and test classification accuracy (as a proportion) grouped by different thresholds for each dataset. Results for each of the three levels of the range threshold are collected over $3^4 \times 4 = 324$ combinations of the other five thresholds, etc.

	Coins				*Wine*				*Wisconsin*			
	Time	(s.d.)	T.Acc	(s.d.)	Time	(s.d.)	T.Acc	(s.d.)	Time	(s.d.)	T.Acc	(s.d.)
Base	4.87	1.80	0.8594	0.0314	1.09	0.40	0.7346	0.0379	6.66	2.27	0.9532	0.0063
Range Threshold												
0.1	2.07	0.25	0.8490	0.0205	0.70	0.02	0.6567	0.0305	3.93	0.55	0.9546	0.0020
0.5	1.96	0.22	0.8498	0.0202	0.70	0.02	0.6503	0.0295	3.89	0.53	0.9546	0.0021
1.0	1.89	0.20	0.8489	0.0192	0.70	0.02	0.6466	0.0287	3.84	0.52	0.9545	0.0021
Redundancy Threshold												
0.01	2.11	0.26	0.8540	0.0188	0.70	0.02	0.6531	0.0295	3.92	0.54	0.9546	0.0020
0.05	1.94	0.21	0.8491	0.0203	0.70	0.02	0.6507	0.0294	3.88	0.53	0.9546	0.0021
0.10	1.88	0.20	0.8445	0.0205	0.70	0.02	0.6498	0.0296	3.86	0.53	0.9546	0.0021
Regression Threshold												
none	1.96	0.26	0.8632	0.0210	0.70	0.02	0.6399	0.0295	3.42	0.57	0.9541	0.0024
0.99	2.07	0.25	0.8547	0.0220	0.71	0.02	0.6582	0.0314	4.22	0.61	0.9543	0.0022
0.95	1.96	0.21	0.8446	0.0201	0.70	0.02	0.6548	0.0300	4.04	0.53	0.9549	0.0021
0.80	1.90	0.19	0.8343	0.0175	0.70	0.02	0.6520	0.0280	3.86	0.47	0.9549	0.0018
Levels Up												
0	1.49	0.11	0.8301	0.0192	0.68	0.02	0.6241	0.0215	3.15	0.31	0.9550	0.0019
1	2.18	0.28	0.8585	0.0210	0.71	0.03	0.6630	0.0349	4.35	0.70	0.9543	0.0022
2	2.25	0.29	0.8590	0.0208	0.71	0.03	0.6665	0.0343	4.15	0.61	0.9543	0.0023
Acceptance Threshold												
0.01	2.21	0.29	0.8534	0.0193	0.70	0.02	0.6563	0.0306	3.98	0.56	0.9545	0.0021
0.05	1.90	0.20	0.8485	0.0199	0.70	0.02	0.6499	0.0296	3.87	0.53	0.9546	0.0020
0.10	1.80	0.18	0.8457	0.0201	0.70	0.02	0.6474	0.0285	3.81	0.51	0.9546	0.0020
Simply Every k Generations												
3	2.08	0.24	0.8487	0.0200	0.71	0.02	0.6585	0.0309	4.29	0.60	0.9546	0.0021
4	1.94	0.22	0.8490	0.0194	0.70	0.02	0.6488	0.0290	3.81	0.53	0.9546	0.0022
5	1.89	0.22	0.8499	0.0201	0.70	0.02	0.6463	0.0297	3.55	0.48	0.9545	0.0022



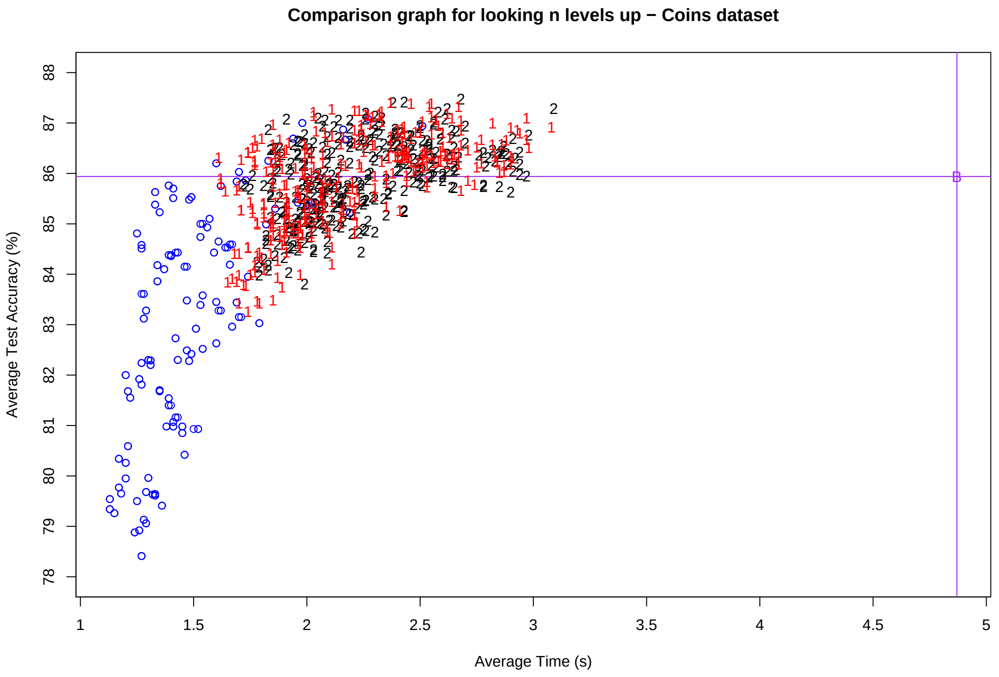
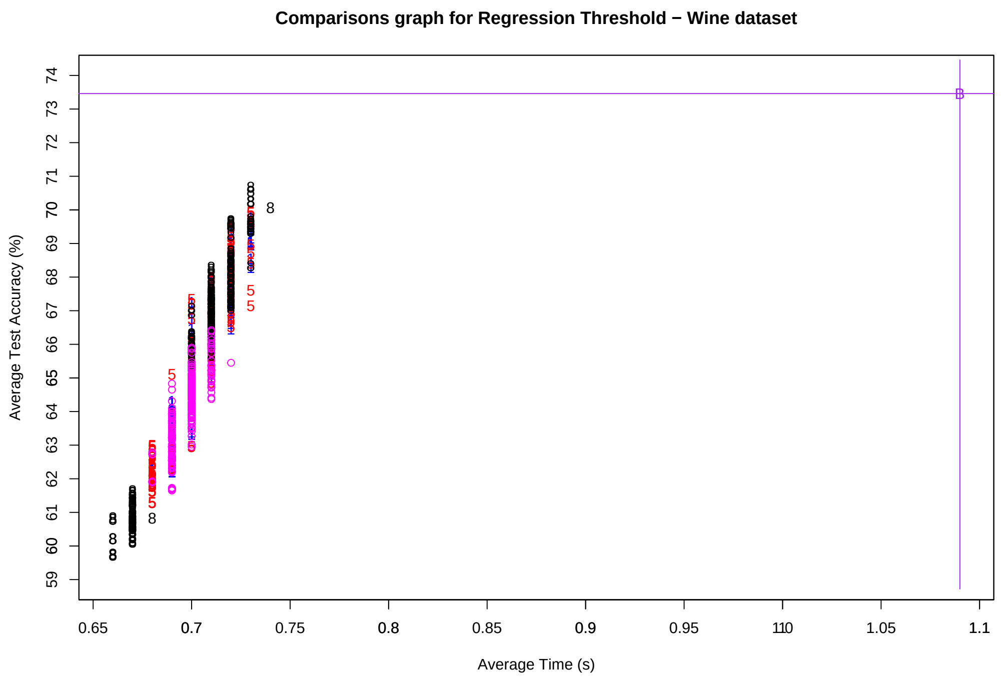


Fig. 4. Two scatter plots showing the average test accuracy vs average CPU time for the Coins dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the regression threshold ('○' = no regression, '1' = 0.99, '5' = 0.95, and '8' = 0.80), and the bottom graph highlights looking n levels up. The lines represent the performance of the base system for comparison.



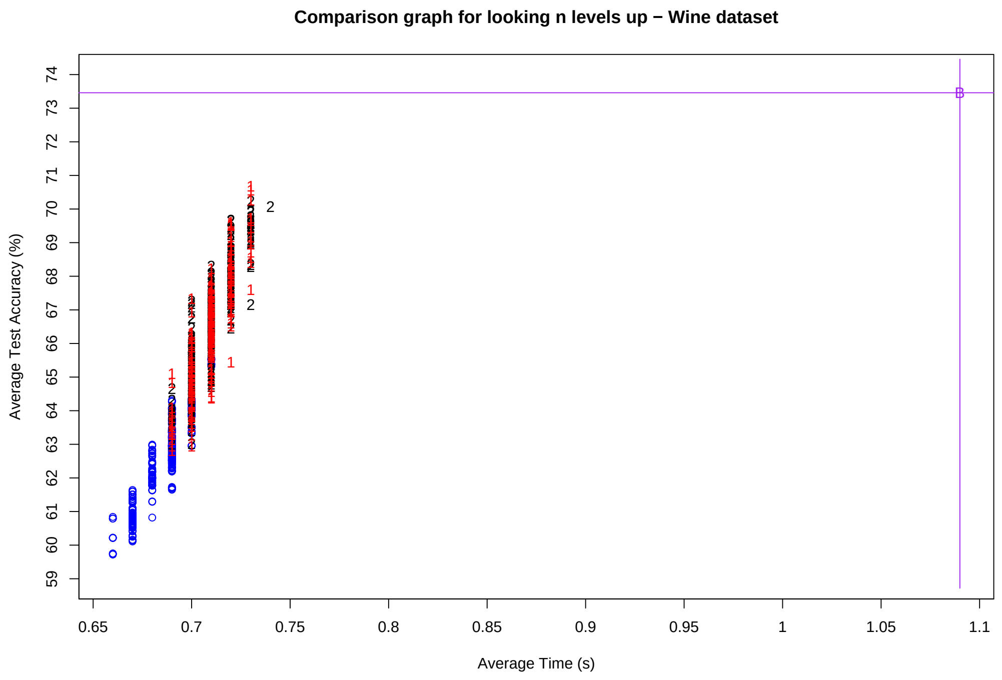
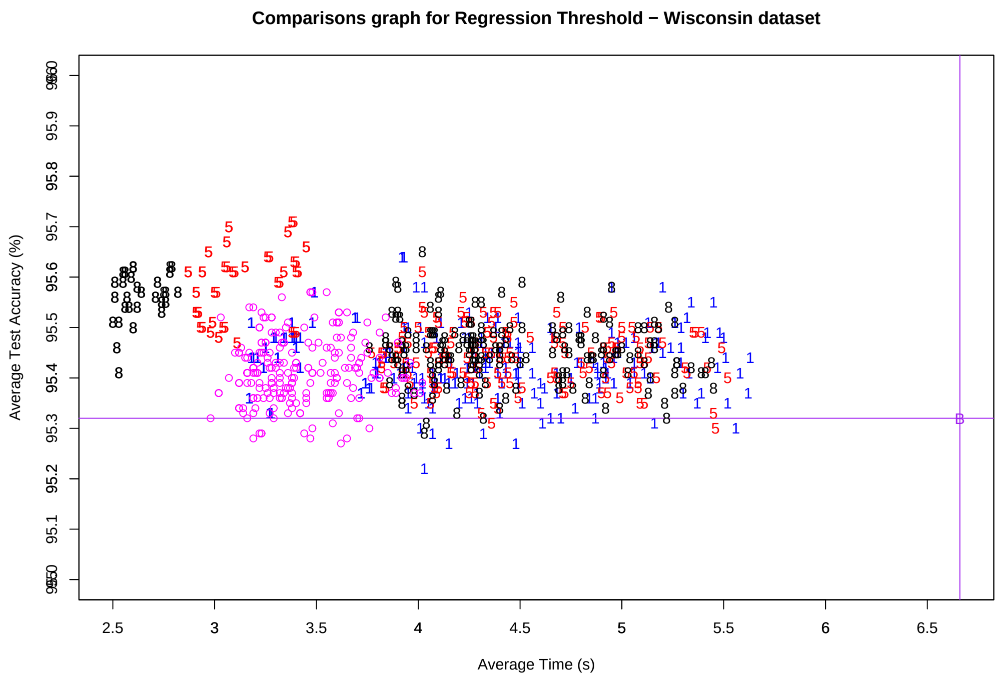


Fig. 5. Two scatter plots showing the average test accuracy vs average CPU time for the Wine dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the regression threshold (‘$\circ$’ $=$ no regression, ‘1’ $= 0.99$, ‘5’ $= 0.95$, and ‘8’ $= 0.80$), and the bottom graph highlights looking n levels up. The lines represent the performance of the base system for comparison.



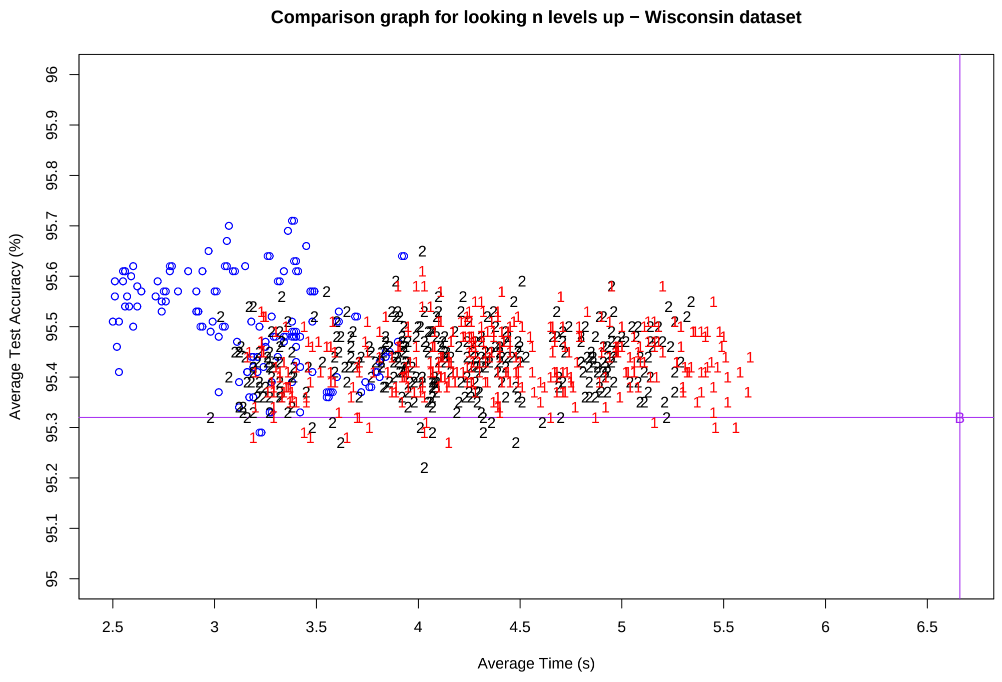
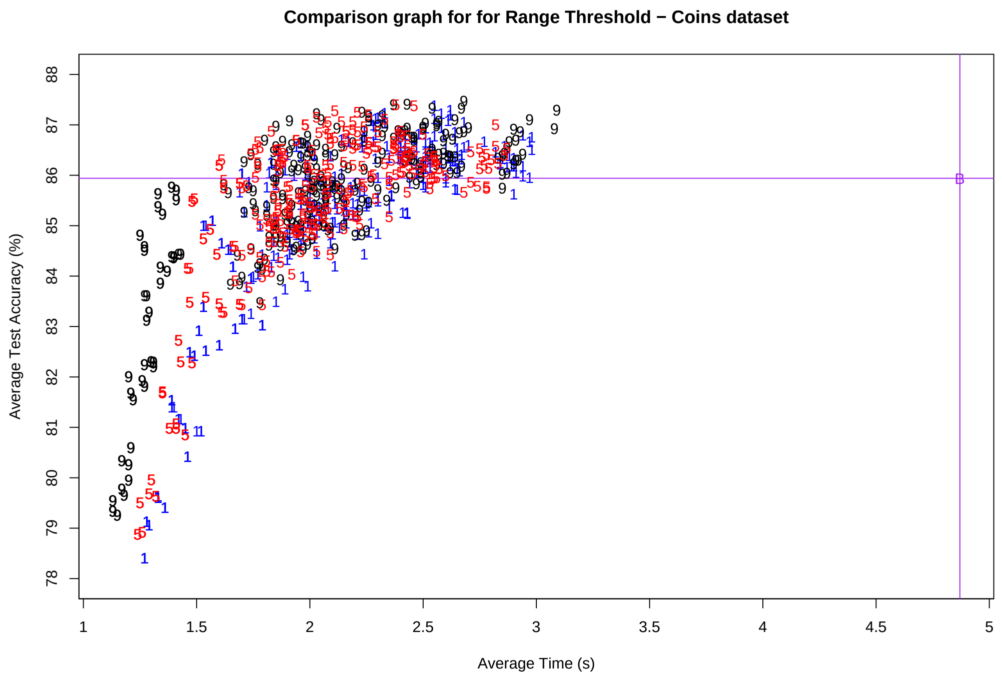


Fig. 6. Two scatter plots showing the average test accuracy vs average CPU time for the Wisconsin dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the regression threshold (‘○’ = no regression, ‘1’ = 0.99, ‘5’ = 0.95, and ‘8’ = 0.80), and the bottom graph highlights looking n levels up. The lines represent the performance of the base system for comparison.



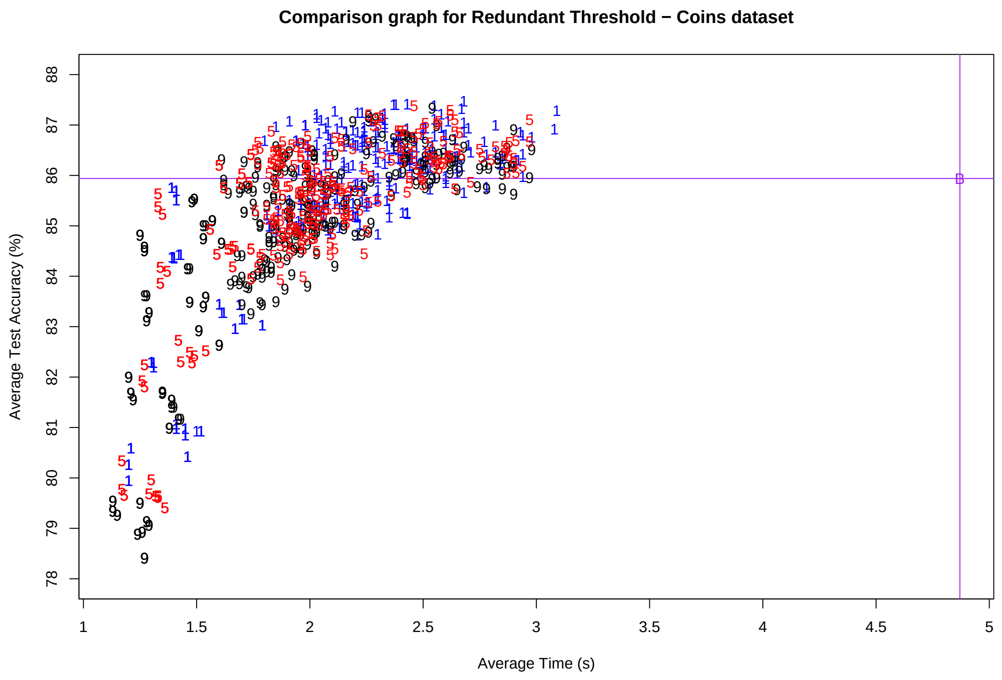
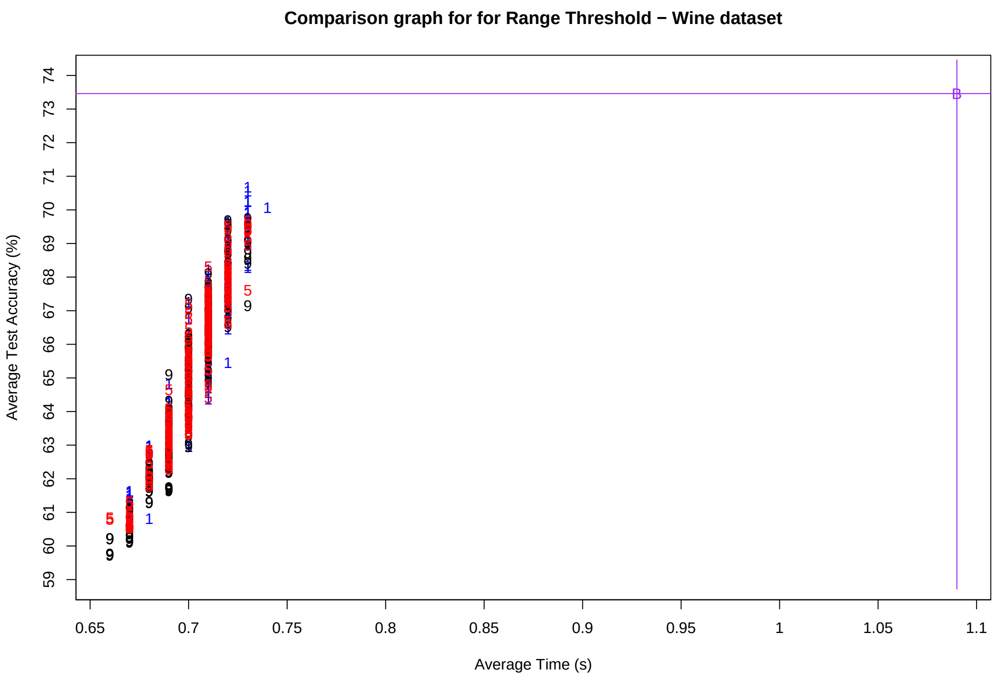


Fig. 7. Two scatter plots showing the average test accuracy vs average CPU time for the Coins dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the range threshold (‘1’ = 0.1, ‘5’ = 0.5, and ‘9’ = 1.0), and the bottom graph highlights the different values for the redundancy threshold (‘1’ = 0.01, ‘5’ = 0.05, and ‘9’ = 0.1). The lines represent the performance of the base system for comparison.



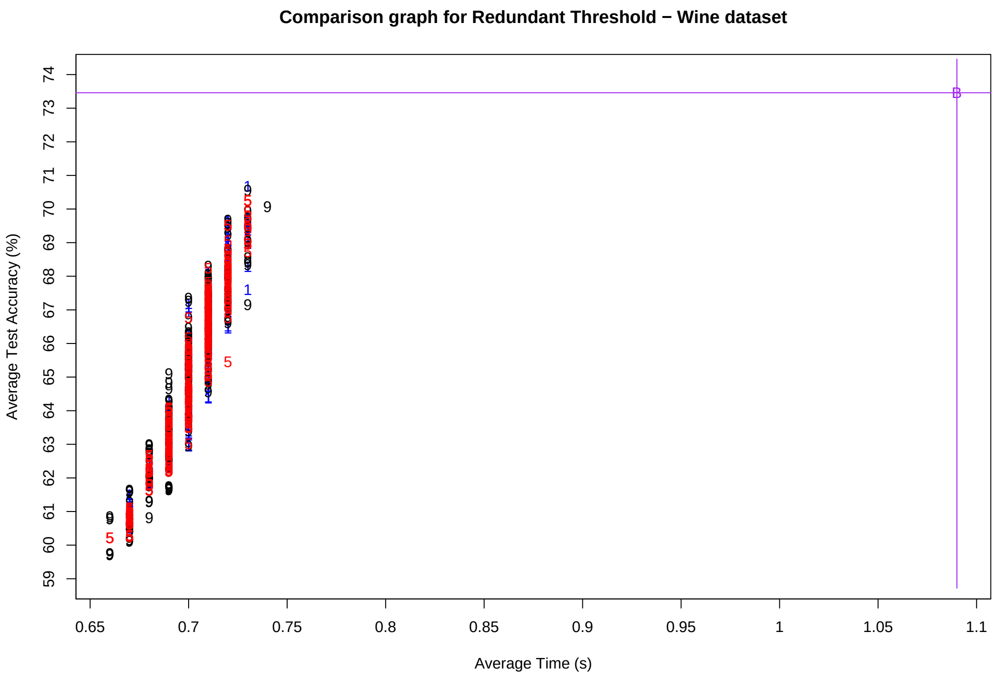
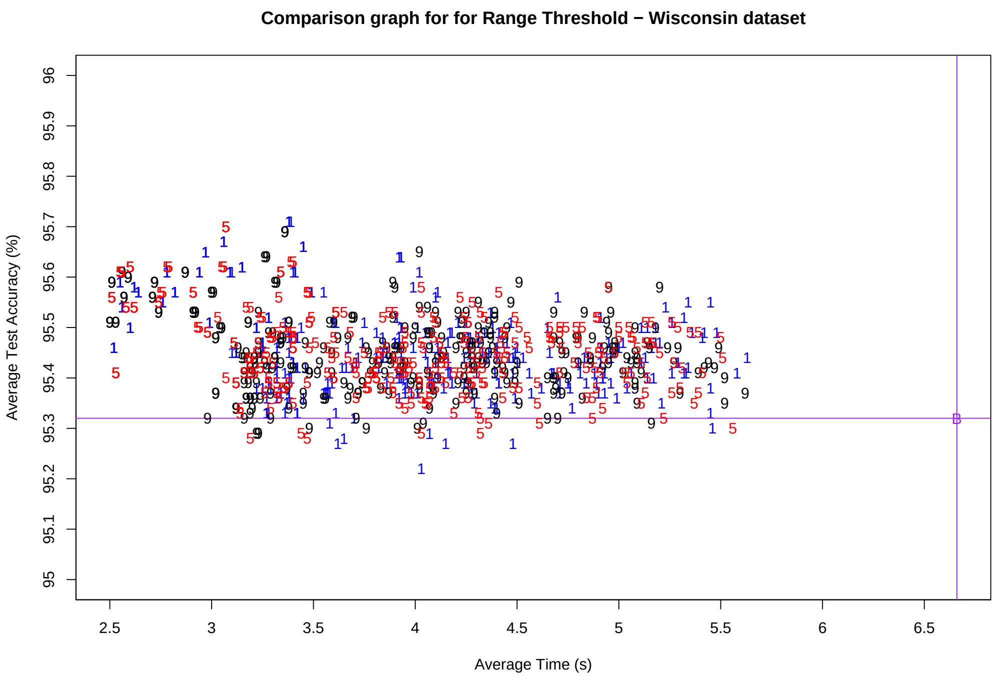


Fig. 8. Two scatter plots showing the average test accuracy vs average CPU time for the Wine dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the range threshold (‘1’ = 0.1, ‘5’ = 0.5, and ‘9’ = 1.0), and the bottom graph highlights the different values for the redundancy threshold (‘1’ = 0.01, ‘5’ = 0.05, and ‘9’ = 0.1). The lines represent the performance of the base system for comparison.



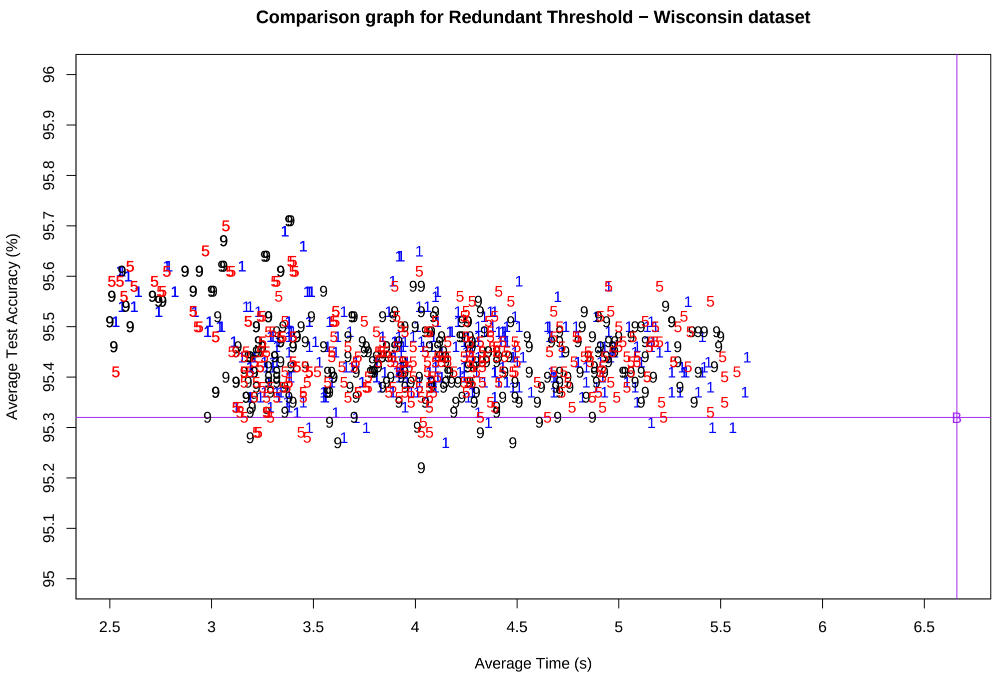
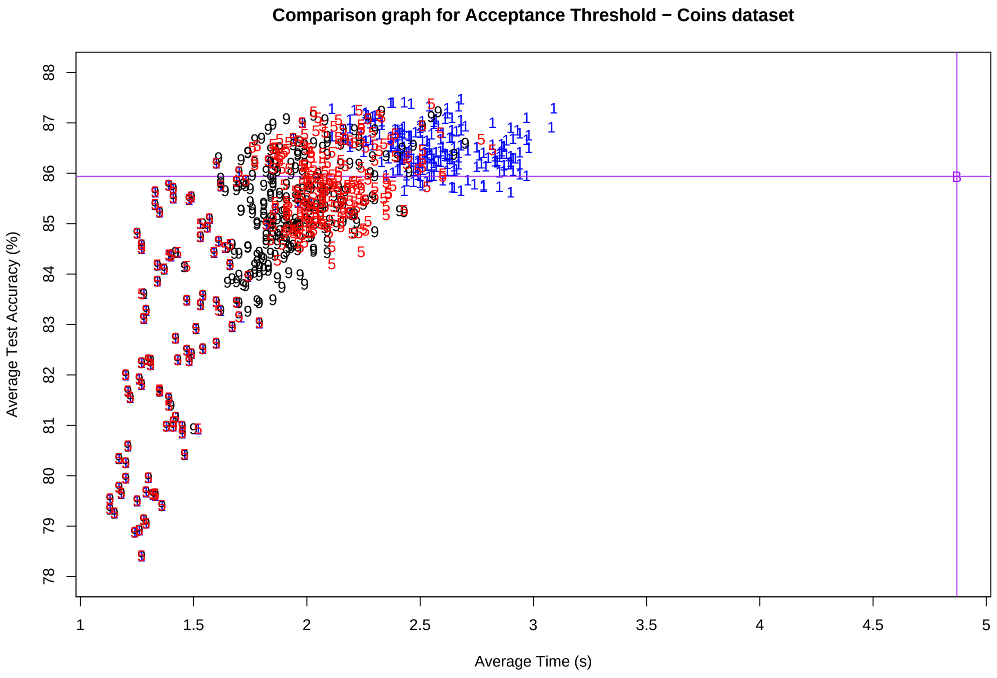


Fig. 9. Two scatter plots showing the average test accuracy vs average CPU time for the Wisconsin dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the range threshold ('1' = 0.1, '5' = 0.5, and '9' = 1.0), and the bottom graph highlights the different values for the redundancy threshold ('1' = 0.01, '5' = 0.05, and '9' = 0.1). The lines represent the performance of the base system for comparison.



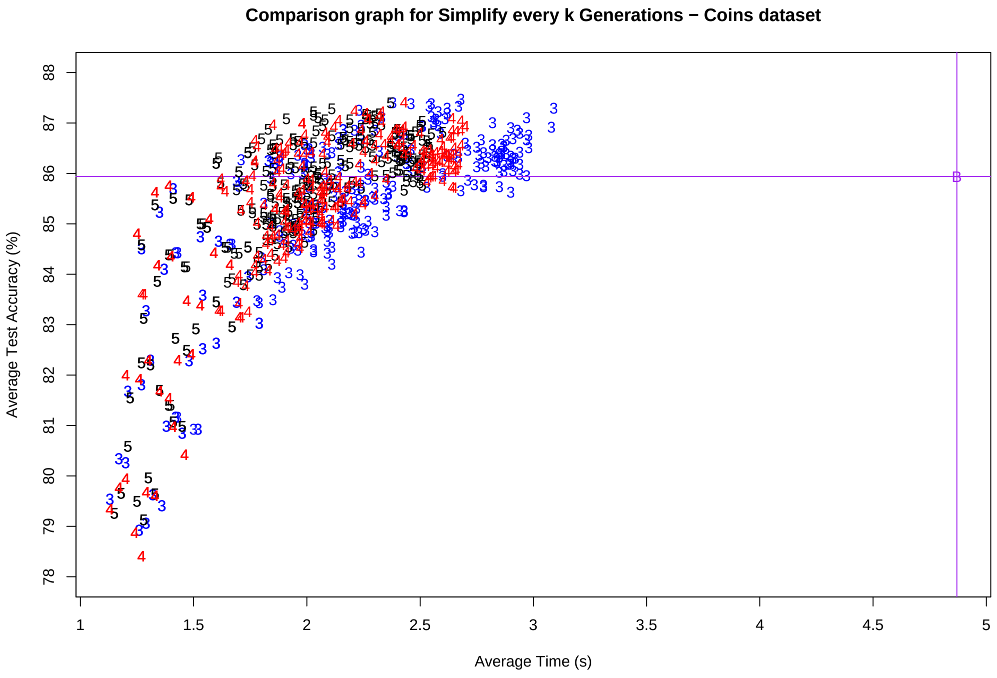
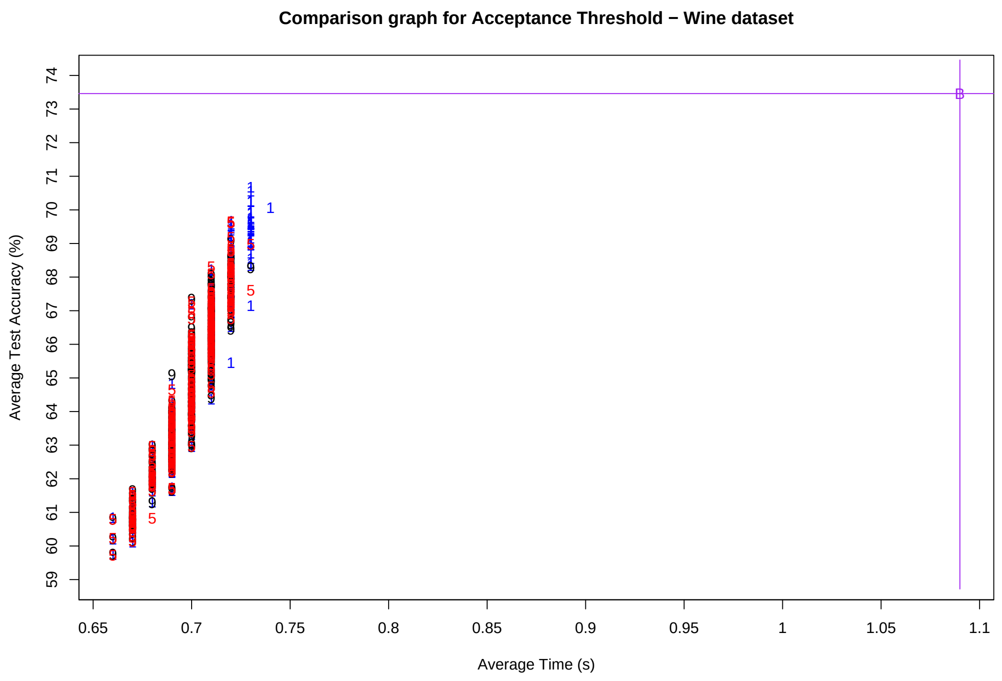


Fig. 10. Two scatter plots showing the average test accuracy vs average CPU time for the Coins dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the acceptance threshold (‘1’ $= 0.01$, ‘5’ $= 0.05$, and ‘9’ $= 0.1$), and the bottom graph highlights simplifying every k generations. The lines represent the performance of the base system for comparison.



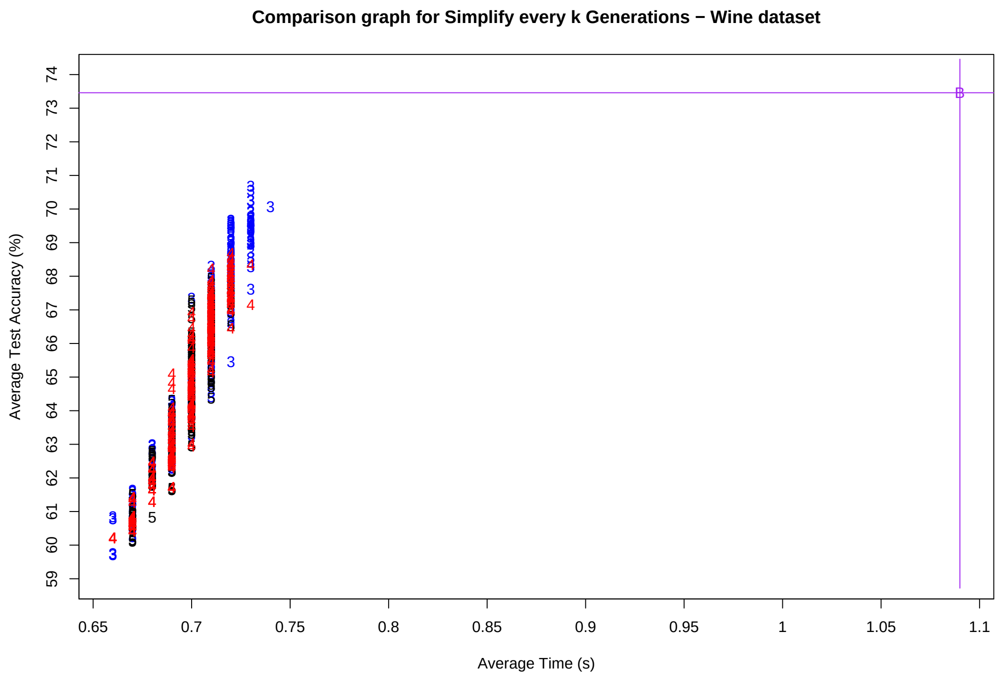
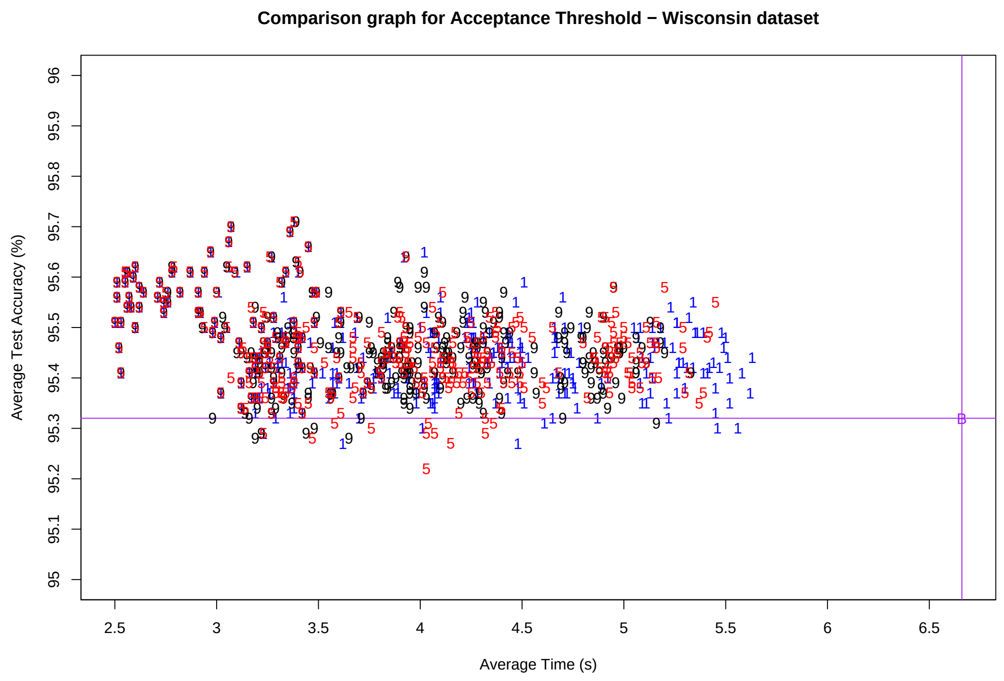


Fig. 11. Two scatter plots showing the average test accuracy vs average CPU time for the Wine dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the acceptance threshold (‘1’ $= 0.01$, ‘5’ $= 0.05$, and ‘9’ $= 0.1$), and the bottom graph highlights simplifying every k generations. The lines represent the performance of the base system for comparison.



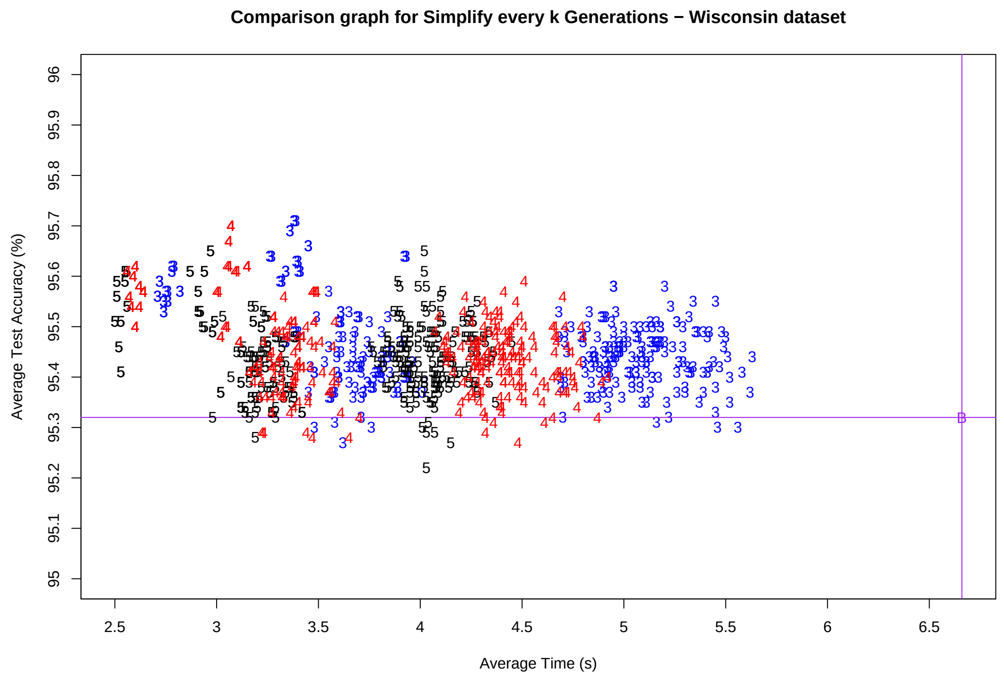


Fig. 12. Two scatter plots showing the average test accuracy vs average CPU time for the Wisconsin dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the acceptance threshold (‘1’ $= 0.01$, ‘5’ $= 0.05$, and ‘9’ $= 0.1$), and the bottom graph highlights simplifying every k generations. The lines represent the performance of the base system for comparison.

when the range threshold is 0.5 and Wisconsin which stays fairly constant). It appears that linear regression is causing more computational overhead than it is worth. The Coins dataset shows this most clearly (see the top graph in Figure 4): the time taken with no regression is similar to that with 0.95 and 0.80 values, but the test accuracy stays higher than all values, i.e., additional computational overhead is not offset by the simplifications made. We see similar CPU time savings without regression in the Wisconsin dataset, but test accuracy remains fairly constant. On the Wine dataset, however, using linear regression has higher test accuracy than not using it, but the test accuracy is still significantly less than that of the base system. Figure 10 (top) shows the effect of relaxing the acceptance threshold, with the clumpings of data moving to the left (faster) and down (worse accuracy) as the value is increased on the Coins dataset. The speed reductions on the Wisconsin dataset can also be seen to a lesser extent in Figure 12. It appears that linear regression is causing more computational overhead than it is worth. The Coins dataset shows this most clearly in Figure 4 (top): the time taken with no regression is similar to that with 0.95 and 0.80 values, but the test accuracy stays higher than all values, i.e., additional computational overhead is not offset by the simplifications made. Figure 6 (top) shows a similar effect (with the exception of a few configurations in the top left corner); we can clearly see that no linear regression runs faster that most configurations. It is hard to see in Figure 5 (top) because of the clumping of all the datapoints, but we can see that linear regression in this case is sitting in the middle of the data, which is not consistent with the other two datasets.

5.3 How Far Up the Tree to Evaluate

In general it seems that as we increase the number of levels we look up before accepting a simplification, the overall average CPU time increases (with the exception of Wisconsin with 2 levels), but so does the test accuracy (Wisconsin's test accuracy remains relatively constant however; see Figure 6 (bottom)). This is best displayed in the Coins dataset where both CPU time and test accuracy change significantly (see Figure 4 (bottom)). In general, looking 0 levels up amounts to a significant time reduction but also a significant reduction in test accuracy (see Figures 4 (bottom) and 5 (bottom)), while looking 1 or 2 levels up only is slightly more computationally expensive but maintains a lot higher test accuracy.

5.4 How Often to Simplify

Overall, there doesn't seem to be much change in test accuracy among the different values for k, although the Wine dataset does show a very slight reduction as we simplify less often, but it is fairly insignificant. As we simplify less often, the CPU time reduces significantly on the Coins and Wisconsin datasets ("bands" can be clearly seen in Figures 10 (bottom) and 12 (bottom)), while the time remains unchanged on Wine. This indicates that it might be useful to investigate higher values (simplifying even less often).

5.5 Comparing Number of Proposals vs Number of Acceptances

Tables 4–6 compare the number of simplifications proposed and accepted, and percentage accepted, for each proposal operator, and shows the effect of increasing the acceptance threshold within each of these for each of the datasets. Some general trends are apparent. Unexpectedly, we see *fewer* proposed simplifications—it is apparent that there may be some "repeat proposing" of simplifications, i.e., a candidate gets rejected but is proposed again later on since it is still a good candidate at the local level (this also explains the higher CPU time for more stringent acceptance threshold values). As expected, the number of accepted proposals increases (except for Coins when the regression threshold is in $\{0.99, 0.95\}$, where the number accepted is relatively similar for acceptance threshold in $\{0.05, 0.10\}$). The average percentage of proposals accepted also increases, although at different rates for each dataset and proposal operator (the best acceptance percentage was just over 50%). The CPU time decreases due to a combination of fewer proposals (lower calculation overhead) and higher number of proposals accepted (overhead incurred in our implementation if a simplification proposal is rejected). The Coins dataset shows the largest reduction in CPU time, while none is observed on the Wine dataset. As expected, the average test accuracy decreases—as we accept less accurate approximations of portions of the tree, this causes the tree itself to have poorer accuracy in general. Again, Wisconsin is an exception, showing little change in test accuracy. Coins shows the highest reduction in test accuracy as well as CPU time seen above, so there seems to be a tradeoff. It is interesting to note, however, that some individual combinations of the simplification operators actually increase the average test accuracy compared to the base system (see Figure 4 (top)). This could mean that simplifications are taking place in an early generation, allowing more of the search space to be covered in less time, but further research would be required to establish this.

5.6 How the Proposal Thresholds Affect the Number of Proposals and Acceptances

Across all datasets, for the linear regression operator, decreasing the value of the threshold increases the number of simplification proposals. Proportionately, the number of proposals accepted increases but on average the percentage of proposals accepted decreases, indicating the acceptance operator is working. Surprisingly, for range simplification and redundancy, as we increase the value of the threshold, we actually see a

Table 4. Comparison of number of simplifications proposed vs number accepted for the Coins dataset. All numbers are averages over all relevant configurations.

	Proposal Thresh.	Accept Thresh.	#Prop	#Acpt	%Acpt	Time (s)	(sd)	T.Acc	(sd)
Base Sys.		-	-	-	-	4.87	1.8	0.8594	0.0314
Range Thresh.	0.1	0.01	468.37	131.54	28.09	2.29	0.52	0.8531	0.0218
		0.05	340.77	142.53	41.82	2.00	0.35	0.8483	0.0199
		0.10	301.51	147.29	48.85	1.91	0.32	0.8455	0.0193
	0.5	0.01	428.95	111.66	26.03	2.18	0.49	0.8535	0.0208
		0.05	305.13	120.69	39.55	1.89	0.31	0.8492	0.0192
		0.10	270.29	125.77	46.53	1.81	0.28	0.8467	0.0188
	1.0	0.01	422.51	95.31	22.56	2.17	0.65	0.8535	0.0223
		0.05	284.44	103.66	36.44	1.83	0.41	0.8481	0.0197
		0.10	228.95	96.33	42.07	1.68	0.31	0.8450	0.0186
Redundant Thresh.	0.01	0.01	486.03	136.64	28.11	2.27	0.51	0.8566	0.0190
		0.05	370.32	159.29	43.01	2.07	0.38	0.8537	0.0180
		0.10	326.42	162.73	49.85	1.98	0.35	0.8517	0.0175
	0.05	0.01	436.42	109.46	25.08	2.20	0.57	0.8537	0.0215
		0.05	291.89	111.38	38.16	1.84	0.31	0.8477	0.0188
		0.10	255.86	114.56	44.78	1.77	0.27	0.8460	0.0183
	0.10	0.01	397.38	92.41	23.26	2.16	0.60	0.8498	0.0235
		0.05	268.14	96.21	35.88	1.80	0.34	0.8442	0.0207
		0.10	218.47	92.10	42.16	1.66	0.24	0.8394	0.0188
Regression Thresh.	none	0.01				2.13	0.42	0.8653	0.0076
		0.05	–	–	–	1.90	0.30	0.8629	0.0066
		0.10				1.83	0.29	0.8614	0.0066
	0.99	0.01	252.68	78.89	31.22	2.31	0.56	0.8580	0.0124
		0.05	181.62	81.38	44.81	2.00	0.37	0.8539	0.0100
		0.10	157.63	81.11	51.46	1.90	0.32	0.8520	0.0093
	0.95	0.01	517.82	142.15	27.45	2.22	0.59	0.8500	0.0201
		0.05	365.83	151.47	41.40	1.89	0.37	0.8439	0.0161
		0.10	313.79	151.09	48.15	1.78	0.30	0.8399	0.0138
	0.80	0.01	989.28	230.30	23.28	2.18	0.65	0.8401	0.0300
		0.05	693.01	256.32	36.99	1.81	0.40	0.8333	0.0254
		0.10	596.26	260.33	43.66	1.70	0.32	0.8296	0.0231

Table 5. Comparison of number of simplifications proposed vs number accepted for the Wine dataset. All numbers are averages over all relevant configurations.

	Proposal Thresh.	Accept Thresh.	#Prop	#Acpt	%Acpt	Time (s)	(sd)	T.Acc	(sd)
	Base Sys.	–	–	–	–	4.87	1.80	0.8594	0.0314
Range Thresh.	0.1	0.01	265.01	99.92	37.70	0.71	0.02	0.6603	0.0266
		0.05	248.77	104.79	42.12	0.70	0.01	0.6559	0.0227
		0.10	241.74	107.76	44.58	0.70	0.01	0.6538	0.0210
	0.5	0.01	247.45	88.85	35.91	0.70	0.02	0.6552	0.0267
		0.05	230.57	92.44	40.09	0.70	0.01	0.6489	0.0220
		0.10	224.91	95.25	42.35	0.70	0.01	0.6468	0.0204
	1.0	0.01	247.33	84.56	34.19	0.70	0.02	0.6533	0.0287
		0.05	219.95	84.75	38.53	0.70	0.01	0.6448	0.0223
		0.10	213.29	86.71	40.65	0.70	0.01	0.6417	0.0205
Redundant Thresh.	0.01	0.01	259.92	95.08	36.58	0.70	0.02	0.6570	0.0266
		0.05	241.88	98.69	40.80	0.70	0.01	0.6522	0.0230
		0.10	235.95	101.6	43.06	0.70	0.01	0.6501	0.0214
	0.05	0.01	251.94	90.36	35.87	0.70	0.02	0.6561	0.0276
		0.05	230.45	92.93	40.33	0.70	0.01	0.6488	0.0223
		0.10	225.15	95.78	42.54	0.70	0.01	0.6471	0.0212
	0.10	0.01	247.93	87.89	35.45	0.70	0.02	0.6557	0.0283
		0.05	226.95	90.36	39.82	0.70	0.01	0.6486	0.0229
		0.10	218.83	92.33	42.19	0.70	0.01	0.6451	0.0207
Regression Thresh.	none	0.01				0.70	0.01	0.6429	0.0114
		0.05	–	–	–	0.70	0.01	0.6391	0.0097
		0.10				0.70	0.01	0.6377	0.0094
	0.99	0.01	175.10	70.42	40.22	0.71	0.01	0.6633	0.0230
		0.05	160.74	72.01	44.80	0.70	0.01	0.6569	0.0184
		0.10	156.70	73.89	47.15	0.70	0.01	0.6543	0.0167
	0.95	0.01	284.97	107.04	37.56	0.71	0.02	0.6606	0.0274
		0.05	262.05	110.68	42.24	0.70	0.01	0.6533	0.0223
		0.10	254.25	113.69	44.72	0.70	0.01	0.6504	0.0207
	0.80	0.01	552.99	186.98	33.81	0.70	0.02	0.6582	0.0371
		0.05	509.59	193.29	37.93	0.70	0.02	0.6503	0.0313
		0.10	495.63	198.70	40.09	0.70	0.02	0.6474	0.0293

Table 6. Comparison of number of simplifications proposed vs number accepted for the Wisconsin dataset. All numbers are averages over all relevant configurations.

	Proposal Thresh.	Accept Thresh.	#Prop	#Acpt	%Acpt	Time (s)	(sd)	T.Acc	(sd)
Base Sys.		–	–	–	–	6.66	2.27	0.9532	0.0063
Range Thresh.	0.1	0.01	617.20	154.33	25.00	4.02	0.83	0.9545	9.0E-4
		0.05	582.41	162.67	27.93	3.91	0.75	0.9546	9.0E-4
		0.10	563.67	168.84	29.95	3.85	0.70	0.9546	8.0E-4
	0.5	0.01	610.46	151.25	24.78	3.98	0.83	0.9545	8.0E-4
		0.05	574.45	158.96	27.67	3.87	0.74	0.9545	8.0E-4
		0.10	556.92	165.05	29.64	3.82	0.70	0.9546	8.0E-4
	1.0	0.01	606.05	146.54	24.18	3.95	0.85	0.9545	8.0E-4
		0.05	564.19	152.85	27.09	3.82	0.75	0.9545	8.0E-4
		0.10	543.64	157.83	29.03	3.75	0.69	0.9546	8.0E-4
Redundant Thresh.	0.01	0.01	625.76	156.83	25.06	4.01	0.83	0.9546	8.0E-4
		0.05	589.19	166.80	28.31	3.90	0.74	0.9546	8.0E-4
		0.10	572.08	173.30	30.29	3.86	0.70	0.9546	8.0E-4
	0.05	0.01	609.71	150.27	24.65	3.98	0.84	0.9545	8.0E-4
		0.05	571.07	157.00	27.49	3.85	0.74	0.9546	8.0E-4
		0.10	554.26	163.72	29.54	3.80	0.70	0.9546	8.0E-4
	0.10	0.01	598.24	145.01	24.24	3.96	0.84	0.9545	9.0E-4
		0.05	560.80	150.68	26.87	3.84	0.76	0.9545	8.0E-4
		0.10	537.88	154.70	28.76	3.76	0.69	0.9546	8.0E-4
Regression Thresh.	none	0.01				3.46	0.24	0.9541	6.0E-4
		0.05	–	–	–	3.41	0.23	0.9541	6.0E-4
		0.10				3.39	0.22	0.9542	7.0E-4
	0.99	0.01	371.95	95.94	25.79	4.32	0.75	0.9543	7.0E-4
		0.05	346.96	102.82	29.64	4.20	0.66	0.9544	7.0E-4
		0.10	332.48	107.71	32.39	4.13	0.60	0.9543	6.0E-4
	0.95	0.01	660.15	165.04	25.00	4.15	0.83	0.9548	0.0010
		0.05	619.54	174.43	28.15	4.02	0.74	0.9549	9.0E-4
		0.10	598.75	182.27	30.44	3.95	0.69	0.9550	9.0E-4
	0.80	0.01	1412.85	341.84	24.20	3.99	1.05	0.9549	8.0E-4
		0.05	1328.24	355.39	26.76	3.84	0.93	0.9548	8.0E-4
		0.10	1287.74	365.65	28.40	3.76	0.87	0.9549	7.0E-4

reduction in proposals on average, and therefore a reduction in proposals accepted as well. A possible reason for this reduction could be the nature of the proposal operators: in both cases, once a simplification has occurred, those nodes can no longer be further simplified through these two methods. However, a linear regression simplification could in turn allow for another simplification the next level up the tree, a sort of cascading effect.

6 Conclusions

All configurations of the simplification operators significantly reduced the CPU time for the GP process to run. However, the tradeoff between CPU time and classification accuracy was different for different configurations and different datasets. Range simplification and removing redundant children appear to be useful simplification operators to use because they are simple and computationally efficient. However, the computational tests were inconclusive as to whether the linear regression operator we introduced is worth using (good for Wine, no change for Wisconsin, poor for Coins). Evaluating the effect of proposed simplifications further up the tree (rather than blind acceptance) appears to be very effective in catching bad case simplifications (Coins and Wine show that classification accuracy improves); looking one level up seems to be sufficient. As there is little reduction in test accuracy for any of the acceptance MSE threshold values tested in this paper, a more lenient MSE value may be desired for further CPU time reductions. Finally, when simplifying a population, it seems to be better to do so less often because of the high overhead incurred, so the less often you simplify, the faster the GP process runs (our best results were simplifying every five generations).

Avenues for future research include investigating the effect of these simplification methods on tree size and tree depth across different generations, eliminating the repeat proposing of the same simplification by the regression operator, applying the linear regression operator on more datasets to see if there is any consistency amongst different types of problems, and further investigating simplifying less often to find the optimal balance between size reduction and computational overhead.

Acknowledgement

This work was supported in part by the University Research Fund (URF09-2399/85608) at Victoria University of Wellington.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, Mass. (1992)
2. Soule, T., Foster, J.A., Dickinson, J.: Code growth in genetic programming. In J.R. Koza et al, ed.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (1996) p. 215–223
3. Soule, T., Heckendorn, R.B.: An analysis of the causes of code growth in genetic programming. Genetic Programming and Evolvable Machines (2002) p. 283–309
4. Blickle, T., Thiele, L.: Genetic programming and redundancy. In Hopf, J., ed.: Genetic Algorithms within the Framework of Evolutionary Computation, Max-Planck-Institut für Informatik (MPI-I-94-241) (1994) p. 33–38
5. Wong, P., Zhang, M.: Algebraic simplification of GP programs during evolution. In M. Keijzer et al, ed.: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO 2006). Volume 1., Seattle, Washington, USA, ACM Press (8–12 July 2006) p. 927–934
6. Zhang, M., Wong, P., Qian, D.: Online program simplification in genetic programming. In T. Wang et al, ed.: Proceedings of the 6th International Conference on Simulated Evolution and Learning (SEAL 2006). Volume 4247 of LNCS., Hefei, China, Springer (15–18 October 2006) p. 592–600
7. Kinzett, D., Zhang, M., Johnston, M.: Using numerical simplification to control bloat in genetic programming. In X. Li et al, ed.: Proceedings of the 7th International Conference on Simulated Evolution And Learning (SEAL 2008). Volume 5361 of LNCS., Melbourne, Australia, Springer (7–10 December 2008) p. 493–502
8. Song, A., Chen, D., Zhang, M.: Bloat control in genetic programming by evaluating contribution of nodes. In G. Raidl et al, ed.: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO 2009), Montreal, ACM (8–12 July 2009) p. 1893–1894
9. RMIT University: RMIT-GP: The RMIT university genetic programming system. http://www.cs.rmit.edu.au/∼vc/rmitgp/
10. Sun Software: Sun grid engine. http://www.sun.com/software/sge/
11. Forina, M., Leardi, R., Armanino, C., Lanteri, S.: PARVUS: An Extendable Package of Programs for Data Exploration, Classification and Correlation. Elsevier, Amsterdam (1988)
12. Asuncion, A., Newman, D.J.: UCI Machine Learning Repository (2007) http://www.ics.uci.edu/∼mlearn/MLRepository.html.
13. Zhang, M., Ciesielski, V.: Genetic programming for multiple class object detection. In Foo, N., ed.: 12th Australian Joint Conference on Artificial Intelligence. Volume 1747 of LNAI., Sydney, Australia, Springer-Verlag (6-10 December 1999) 180–192

Appendices

A Pearson's Correlation Coefficient

For completeness, the formulae for Pearson's correlation coefficient is:

$$\begin{aligned} r &= \frac{\sum XY - \frac{1}{n}\sum X \sum Y}{\sqrt{(\sum X^2 - \frac{1}{n}(\sum X)^2)(\sum Y^2 - \frac{1}{n}(\sum Y)^2)}} \\ &= \frac{S_{XY}}{\sqrt{S_{XX}S_{YY}}} \end{aligned}$$

where

$$\begin{aligned} S_{XY} &= \sum XY - \frac{1}{n}\sum X \sum Y \\ S_{XX} &= \sum X^2 - \frac{1}{n}(\sum X)^2 \\ S_{YY} &= \sum Y^2 - \frac{1}{n}(\sum Y)^2 \end{aligned}$$

and n is the number of training examples, X is the results vector (the evaluation of the tree up to this point for each training example) for the child subtree S or $\frac{1}{S}$, and Y is the results vector for the parent node. The regression equation is

$$\widehat{Y} = bX + a$$

where

$$\begin{aligned} b &= \frac{S_{XY}}{S_{XY}} \\ a &= \frac{1}{n}\sum Y - b\frac{1}{n}\sum X \\ &= \overline{Y} - b\overline{X} \end{aligned}$$

B How far down the tree to go for S

An important consideration when implementing the linear regression testing on a particular subtree is how many levels down the tree to go for child comparison (to get S). Since the general case for linear regression would produce a simplified tree of size $4 + |S|$ nodes, it would not be sensible to perform the calculations on subtrees that were either terminal ($|S| = 1$) or had two terminal children ($|S| = 3$). We built this size check into our implementation to save wasted time. This means that for any possible size-reduction benefit (in the smallest-tree worst case scenario, see Figure 13), we would need to look at least to the third level of the subtree (which in the worst case would not reduce the tree size), and probably would need to look to the fourth level

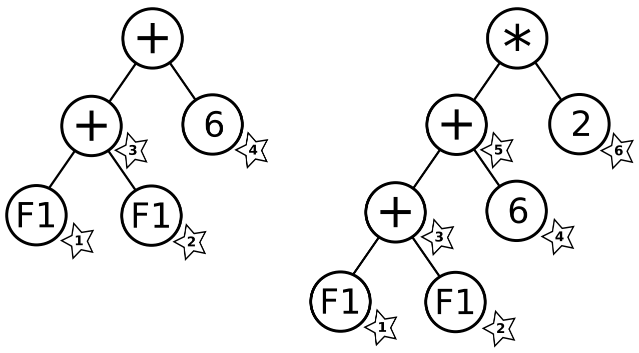


Fig. 13. Examples of the smallest tree sizes for 3 levels (left) and 4 levels (right). The numbers inside the stars indicate the number of nodes linear regression tests will be performed on, and the order in which they are done.

(which, if a suitable linear correlation is found, would guarantee at least a reduction of 2 nodes in the smallest-tree case). The tradeoff that needs to be considered is the number of nodes that the linear correlation tests (which are reasonably computationally expensive) would need to be calculated on. Even though in the worst case, looking down to the second level won't save any tree size, it is possible that one or both sides have a very large subtree, so we considered it worth checking against these children as well. Therefore we did the tests in a 'bottom-up' fashion, from the lowest level, to the second level. If we test to only the third level, we would need to do the calculations on a minimum of $2 + 2^1 = 4$ nodes (or $3 + 3^1 = 6$ if all functions above are `ifpos`) and a maximum of between $2^2 + 2^1 = 6$ and $3^2 + 3^1 = 12$ (if all functions above are `ifpos`) nodes.

Once we go to the fourth level however, the maximum number of nodes in that level stretches out to between $2^3 = 8$ and $3^3 = 27$ (if all functions above are `ifpos`). Because our function set is made up of 4 functions that have 2 children ($+, -, \times, \%$) and 1 function that has 3 children (`ifpos`), our worst-case scenarios are more likely to lean more towards the binary tree cases, which are not so bad, but it is still important to note the ternary worst case (as unlikely as it is). Compounding the size of the fourth level is the fact that if we go this far down, we also need to check the third level as well, in case no simplification candidates are found in the 4th level. So in the worst case (a full tree to the 4th level) this gives us a total of either $8+4+2 = 14$ nodes (a fully binary tree) or up to $27 + 9 + 3 = 39$ (a fully ternary tree) nodes to do linear regression testing on. This is just too expensive.

In order to save computational time, we also built into our implementation a check to see if the subtree Y's size is already $4 + |S|$ nodes, and therefore would not be able to produce any reduction. However, we also note a possible advantage to allowing the linear regression process to "re-arrange" Y anyway. In a special case, where $b = 1$, the re-arranging "formats" the tree in such a way that the regression can catch a simplification case that would normally only be possible by going to the next level down. The following example shows this process (see Figure 14 for visual representation). We allow the re-arranging, and we get a special case regression in Step 3, which reduces the tree's size to $2 + |S|$.

```
Tree: (+ (+ 3 (* 4 F2)) 6)

*** Beginning Simplification

Step 1:
Simplify (* 4 F2):
Redundancy check:
Child 0, sum square error: 2828.32
Child 1, sum square error: 1892.25
Regression check:
Subtree too small to do regression (size = 3)

Step 2:
Simplify (+ 3 (* 4 F2)):
Redundancy check:
Child 0, sum square error: 3364
Child 1, sum square error: 4860
Regression check:
Parent node size 5 (regression won't reduce tree size)
Child 1 size = 1    - F2
Child 1 regression:
r^2: 1, a: 3, b: 4
NewNode: (+ (* F2 4) 3)
Current Simplified Tree: (+ (+ (* F2 4) 3) 6)

Step 3:
Simplify (+ (+ (* F2 4) 3) 6):
Redundancy check:
Child 0, sum square error: 19440
Child 1, sum square error: 15105.8
Regression check:
Parent node size 7
Child 0 size = 3    - (* F2 4)
Child 0 regression:
r^2: 1, a: 9, b: 1
NewNode: (+ (* F2 4) 9)

*** Simplification complete

Original Tree:   (+ (+ 3 (* 4 F2)) 6)
Number of nodes: 7, Depth: 4

Simplified Tree: (+ (* F2 4) 9)
Number of nodes: 5, Depth: 3
```

```
Savings: 2 nodes, 1 level
```

In the above example, the re-arranging in Step 2 doesn't result in size reduction, but in Step 3, re-arranging allows us to catch a simplification the next level down by doing the tests on the subtree `(* F2 4)`. This only works because of the special case $b = 1$ in Step 3's regression tests. Not allowing the re-arranging causes Steps 2 and 3 to stop, and we miss out on catching the special case.

```
Tree: (+ (+ 3 (* 4 F2)) 6)

*** Beginning Simplification

Step 1:
Simplify (* 4 F2):
Redundancy check:
Child 0, sum square error: 2828.32
Child 1, sum square error: 1892.25
Regression check:
Subtree too small to do regression (size = 3)

Step 2:
Simplify (+ 3 (* 4 F2)):
Redundancy check:
Child 0, sum square error: 3364
Child 1, sum square error: 4860
Regression check:
Parent node size 5
Child size = 1   - F2
Subtree already at optimum size, regression won't reduce

Step 3:
Simplify (+ (+ 3 (* 4 F2)) 6):
Redundancy check:
Child 0, sum square error: 19440
Child 1, sum square error: 15105.8
Regression check:
Parent node size 7
Child size = 3   -  (* 4 F2)
Subtree already at optimum size, regression won't reduce

*** Simplification complete

Original Tree:   (+ (+ 3 (* 4 F2)) 6)
Number of nodes: 7, Depth: 4
Simplified Tree: (+ (+ 3 (* 4 F2)) 6)
Number of nodes: 7, Depth: 4
Savings: 0 nodes, 0 levels
```

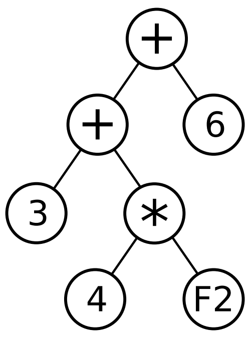


Fig. 14. An example GP tree, that will be simplified if re-arranging is allowed, but won't be if not (because it is a special case where $b = 1$).

Since these special case scenarios are too rare to justify the extra overhead for all the other cases, we do not allow for rearranging. We also limit ourselves to performing regression to the third level, for a combination of possibly large reductions in tree-size while not allowing the computational time to be too expensive. The final algorithm that we used for finding a linear regression simplification candidate can be seen in Algorithm 1.

C Software Package

For our implementation, we modified the RMIT GP (1.5) package [9], written in C++. We needed to make several changes to this package for our simplification system, including the following.

– "Vectorise" the evaluations, meaning that each node in a program tree held a vector of its evaluation results over all training examples for the subtree below it. We use this vector to calculate the range of values that a node takes for range simplification, and also for use in comparing the change to the tree that a candidate simplification will make. This is also much faster than the original system (tests on 10 random seeds showed a reduction from an average of 1.43 seconds to 0.78 seconds on the symbolic regression problem $2x^2 + 4xy + 3y$).
– Convert the symbolic regression example into classification, including adding the `ifpos` operator.
– Modify the program so that it will read in data from file, with a particular pattern. This pattern can be seen in Appendix 3.
– Modify the way the program handles feature terminals – created one class that allows unlimited number of features (terminals) to be added to the GP system, as specified by the data file, instead of having to duplicate the Feature classes for as many as you need.

– Add capacity for multiple command line arguments to be passed into the program for different configurations.
– Add different levels of 'tracing' for debugging and results processing purposes.
– Implement the simplification process, and run it at the end of every k^{th} generation, with the ability to turn simplification off completely.
– Add statistics calculations for the number of simplifications proposed/accepted.

Data File Format

Our implementation takes a single dataset file with all the examples in it, and uses the n-fold cross validation method for training and test accuracy. The file format that our implementation can read requires the following.

– Filename: [dataname].txt
– A header line in the following format:

```
[num_feat] [num_ex] [n-fold] [class_b4] [classes_start_0]
```

- `num_feat`: The number of features in the dataset.
- `num_ex`: The number of examples in the dataset.
- `n-fold`: For cross fold validation. n is the number of subsets the datafile will be split into. (e.g. 10 = 10-fold cross validation)
- `class_b4`: Boolean value (1 or 0) that allows the class to be either at the start of the example line or at the end
- `classes_start_0`: Boolean value that allows the class labels to begin at either 0 or 1. (Use 1 if classes start at 0, 0 otherwise).

– Fill the rest of the file with the data for each example according to the specifications in the header line. Feature values should be separated by a single space only, and each example should begin on a new line.

A couple of limitations to this file formate are that all data should be numeric, class labels must be sequential (e.g. 0,1,2 or 1,2,3), and our system at this stage can only handle 2 or 3 classes, no more. The datafile should be randomised so that cross-validation can have a good chance of picking a wide variety of example combinations.

An example first few lines of the `wine.txt` file are:

```
13 178 10 1 0 3
2 12.17 1.45 2.53 19 104 1.89 1.75 0.45 1.03 2.95 1.45 2.23 355
2 12.37 1.13 2.16 19 87 3.5 3.1 0.19 1.87 4.45 1.22 2.87 420
3 12.85 3.27 2.58 22 106 1.65 0.6 0.6 0.96 5.58 0.87 2.11 570
```

D Bad case examples of range simplification

Here we give some bad case examples of range simplification without checking the effect looking further up the tree.

Example 1. A trivial division.

The following is a trivial example of a bad simplification case-type, that, when allowing a range simplification threshold of 0.05, results in a large change to the tree one level up.

The feature F1 in this example takes values in the range $(-0.01, 0.01)$
The feature F2 in this example takes values in the range $(-20.0, 20.0)$

```
Tree: (+ (% F2 (* 2 F1)) 0)
Number of nodes: 7, Depth: 4
Evaluated range: 32345.6

*** Beginning Simplification

Step 1:
Simplify (* 2 F1):
Evaluated Range = -0.019797 -> 0.0191529 = 0.0389499, Depth: 3
Simplification possible, replacing with average: -0.000322031
Current Simplified Tree: (+ (% F2 -0.0003) 0)

Step 2:
Simplify (% F2 -0.0003):
Redundancy check:
Child 0, sum square error: 6.06993e+10
Child 1, sum square error: 6.06603e+10

Step 3:
Simplify: (+ (% F2 -0.0003) 0)
Redundancy check:
Child 0, sum square error: 0
Simplification possible, replacing parent node with child 0

*** Simplification complete

Original Tree:   (+ (% F2 (* 2.0000 F1)) 0.0000)
Simplified Tree: (% F2 -0.0003)
Number of nodes: 3, Depth: 2
Evaluated range: 117960
Mean Square Error for simplification: 1.17658e+09
Savings: 4 nodes, 2 levels
```

In this case, the range of the node in Step 1 is small enough to be simplified, and is replaced with a constant value which is very small. This becomes the denominator of the fraction one level above, which has a large range of (comparatively) large values. The division operator causes this node's values to change very significantly. This bad case simplification type can be caught by checking the tree one level up before accepting the simplification.

Example 2. A harder case

The following bad case example is one created using the Coins dataset, from simplifying a tree in the initial random population. It will get caught as a bad simplification if we look 2 levels up (to the root).

```
Original Tree: (* (% (% (- (- F12 F8) (- F8 F9))
                     (% drand-42.053000 (if>0 F11 F2 F5)))
                  (if>0 (if>0 (if>0 F3 F3 F3)
                              (if>0 F7 F9 F7)
                              (- F4 F8))
                        (% (- F13 F12) (% F1 F0))
                        (- drand4.173399 (if>0 F6 F1 F1))))
               (% (+ (* (% F9 F2) (+ F3 F1))
                     (- (+ F0 F4) (+ F4 F2)))
                  (+ (- F12 F2)
                  (- (+ F8 drand2.014714) (+ F2 F1)))))
Number of nodes: 69, Depth: 6
Evaluated range: 184.315    (-82.9886 --> 101.326)

*** Beginning Simplification

Search:
Can simplify (% (- (- F12 F8) (- F8 F9))
                (% drand-42.053000 (if>0 F11 F2 F5))):
Evaluated Range = -0.00491354 -> 0.00472195 = 0.00963548, Depth: 3
Simplification possible, replacing with average: -9.57958e-05

*** Simplification complete

Simplified Tree: (* (% drand-0.000096
                       (if>0 (if>0 (if>0 F3 F3 F3)
                                   (if>0 F7 F9 F7)
                                   (- F4 F8))
                             (% (- F13 F12) (% F1 F0))
                             (- drand4.173399 (if>0 F6 F1 F1))))
                    (% (+ (* (% F9 F2) (+ F3 F1))
                          (- (+ F0 F4) (+ F4 F2)))
                       (+ (- F12 F2)
                          (- (+ F8 drand2.014714) (+ F2 F1)))))
Number of nodes: 56, Depth: 6
Evaluated range: 26.9512    (-17.8945 --> 9.05671)
Mean Square Error for simplification: 44.4615
Savings: 13 nodes, 0 levels

Number of nodes in each level:
Before simplifying: 1 2 4 9 19 34
After simplifying:  1 2 4 7 15 27
```

At the level the simplification takes place, it seems a good simplification. Even looking one level above will not reveal a massive change (MSE: 0.00148394) to the original tree. To try and understand why this is a bad simplification two levels up, we need to look at the different parts of the tree. The 'sibling' to the simplification is the following:

```
Sibling tree: (if>0 (if>0 (if>0 F3 F3 F3)
                          (if>0 F7 F9 F7)
                          (- F4 F8))
                    (% (- F13 F12) (% F1 F0))
                    (- drand4.173399 (if>0 F6 F1 F1))))
Number of nodes: 26, Depth: 4
Evaluated range: 3.35834    (-3.36416 --> -0.00581753)
```

This sibling has all negative values, in a relatively small range. Our simplification, though small in range, had a mix of positive and negative values, but was replaced by a negative value, which may be part of the cause. So looking at the parent one level up, and the effects of the simplification:

```
Parent (1 Level Up): (% (% (- (- F12 F8) (- F8 F9))
                           (% drand-42.053000 (if>0 F11 F2 F5)))
                        (if>0 (if>0 (if>0 F3 F3 F3)
                                    (if>0 F7 F9 F7)
                                    (- F4 F8))
                              (% (- F13 F12) (% F1 F0))
                              (- drand4.173399 (if>0 F6 F1 F1))))
Number of nodes: 41, Depth: 5
Evaluated range: 0.237106    (-0.204258 --> 0.0328484)

*** Beginning Simplification

Search:
Can simplify (% (- (- F12 F8) (- F8 F9))
                (% drand-42.053000 (if>0 F11 F2 F5))):
Evaluated Range = -0.00491354 -> 0.00472195 = 0.00963548, Depth: 3
Simplification possible, replacing with average: -9.57958e-05

*** Simplification complete

Simplified Tree: (% drand-0.000096
                    (if>0 (if>0 (if>0 F3 F3 F3)
                                (if>0 F7 F9 F7)
                                (- F4 F8))
                          (% (- F13 F12) (% F1 F0))
                          (- drand4.173399 (if>0 F6 F1 F1))))
Number of nodes: 28, Depth: 5
Evaluated range: 0.0164383    (2.84754e-05 --> 0.0164667)
Mean Square Error for simplification: 0.00148394
Savings: 13 nodes, 0 levels
```

While there is not a significant MSE change at this parent level, we can see that the range of the parent is narrowed quite a lot and all values have become positive. The final factor causing the simplification to be bad is the top root level, which contains a multiplication operator, with the left child being this newly simplified parent, (with all positive values rather than a mix of both positive and negative), while the right child is the following tree which has a large range of large values, both positive and negative:

```
Right Child of Root: (% (+ (* (% F9 F2) (+ F3 F1))
                           (- (+ F0 F4) (+ F4 F2)))
                        (+ (- F12 F2)
                           (- (+ F8 drand2.014714) (+ F2 F1))))
Number of nodes: 27, Depth: 5
Evaluated range: 5495.59    (-2853.92 --> 2641.68)
```

So the multiplication operator, which was previously multiplying a mix of positive and negative values in both children, is now multiplying a mix on one side with all positive values on the right. Compounded with a smaller range of values in the left child, this causes the large change in the resulting simplified tree. This is a hard case to pick up with the naked eye, which gives motivation for the numerical checks up multiple levels of the tree.