Using Genetic Programming to Obtain a Closed-Form Approximation to a Recursive Function

Evan Kirshenbaum and Henri J. Suermondt

HP Labs, Palo Alto, CA
{Evan.Kirshenbaum, Jaap.Suermondt}@hp.com

Abstract. We demonstrate a fully automated method for obtaining a closed-form approximation of a recursive function. This method resulted from a real-world problem in which we had a detector that monitors a time series and where we needed an indication of the total number of false positives expected over a fixed amount of time. The problem, because of the constraints on the available measurements on the detector, was formulated as a recursion, and conventional methods for solving the recursion failed to yield a closed form or a closed-form approximation. We demonstrate the use of genetic programming to rapidly obtain a high-accuracy approximation with minimal assumptions about the expected solution and without a need to specify problem-specific parameterizations. We analyze both the solution and the evolutionary process. This novel application shows a promising way of using genetic programming to solve recurrences in practical settings.

1 Introduction

In monitoring applications, the false positive rate of a detector is an important statistic used to characterize the practical applicability of the detector. False alarms are not just annoying, but they also tend to make those responsible for responding to alarms less likely to respond to a real alarm and are a frequent reason for people to turn alarms (or even the underlying detector) off. Even in unsupervised environments, false alarms lead to large event logs, fruitless investigations, and unnecessary cost. Therefore, when developing novel detectors for use in time-series monitoring, we needed (among other things) a good way to estimate the expected number of false positives in a negative sequence of a given length, as a core performance statistic by which we judged the detector.

Unfortunately, the performance data available on the detector does not lend itself to easy characterization of the expected number of false positives in a sequence of arbitrary length. Instead, we find ourselves with a multi-parameter nonlinear recurrence relation for which conventional methods do not yield a closed form or even a closed-form approximation.

The recurrence relation results from the following problem definition. Since the detector is stateful and the elements of the series are not independent, we cannot simply assume an independent probability of getting a false positive at each new element. The detector will be run on the sequence until it signals a detection and then will be retrained for a fixed number of points *T*. In our particular case, the data is assumed to follow an ARIMA model [7], and the detector uses Holt-Winters exponential smoothing [14]. In an earlier stage of the investigation, we had tested detectors on randomly generated negative sequences of arbitrary fixed length *w*. In this setup, all the sequences were negative, so any time a detector signaled an event, it should be considered a false positive. We had derived functions that would estimate, for any parameterization of the data stream and detector, the expected false positive rate *FPR*, viewed as the proportion of these randomly generated negative sequences for which the detector would spuriously signal detection, and the expected penetration *p* into a sequence before a false positive occurs, averaged over those sequences for which a false positive occurs.

So the basic notion is that $1-FPR$ of the time, the detector will get through a negative sequence of length *w* without signaling detection, and the remaining *FPR* of the time it will get on average *p* in before signaling, retrain for *T*, and start over.

We make two minor yet practical simplifying assumptions. First, in a sequence that is longer than *w* (the length of our test sequences), the behavior of the detector after successfully scanning a window of size *w* is the same as its behavior when looking from scratch at another sequence generated from the same model (in other words, the detector does not take advantage of its knowledge of the first *w* data points). Second, after the retraining (recalibration) period of length *T,* the detector also behaves as it does when looking from scratch at a sequence generated from the same model. If we define

$$d = p + T \tag{1}$$

as the expected number of points consumed by a false positive, these assumptions yield as a first-order approximation for the expected number of false positives the recurrence relation

$$NFP_{len} = (1 - FPR)NFP_{len-w} + FPR(1 + NFP_{len-d}) \ . \tag{2}$$

Unfortunately, for our purposes, we needed to be able to determine this quantity a large number of times for different parameterizations of the detector, and a recursive definition was not efficient enough, so we needed to find a closed form solution, or at least a good approximation. When $d = w$ (i.e., the case where our test sequences happen to be the same length as the expected number of points consumed by a false positive), this trivially reduces to

$$\begin{aligned} NFP_{len} &= NFP_{len-w} + FPR \\ &= \frac{FPR \cdot len}{w} \end{aligned} \quad , \tag{3}$$

but when $d \neq w$ (as it does, since *d* is a function of w), things rapidly get messy.

There are many techniques for solving recurrences [5, 6, 10, 13]. Most such techniques work only for recurrences of a particular form, typically they involve trial-and-error and guesswork, and many of them become unworkable when the element being determined depends on the value of an element far away, as they do in our problem, in which NFP_{len} depends on NFP_{len-w}. Symbolic math packages can automatically solve specific forms of recurrence relations. *Mathematica* [22], for example, can solve linear recurrence relations for a sequence in which s_n is defined in terms of expressions in s_{n+i} or $s_{q^i n}$, for constant values of i and where there are a fixed number of base cases. In the problem we are addressing, the referenced subscript offsets are non-constant and the number of base cases depend on the parameter w.

Finding a closed-form solution to (2) is exactly the sort of non-linear functional regression problem that genetic programming [1, 12] is asserted to be good at solving, and this gave us an opportunity to test whether it could provide a solution without any special problem knowledge or tweaking of the GP parameters.

2 Experimental Setup

The training set consisted of 1,000 cases randomly drawn over

$$\begin{aligned} fpr &\in [0,1] \\ w &\in [100,200] \\ d &\in [20,150] \\ len &\in [300,1000] \end{aligned}.$$

For each case, we computed NFP_{len}, where

$$NFP_i = \begin{cases} 0 & \text{if } i < d \\ fpr(1+NFP_{i-d}) & \text{if } d \le i < w \\ (1-fpr)\,NFP_{i-w} + fpr(1+NFP_{i-d}) & \text{if } i \ge w \end{cases}. \tag{4}$$

The range of values for NFP_{len} was from 0.0031 to 37.8361.

2.1 GP Parameters

For the genetic programming component, we used GPLab, a genetic programming tool developed at HP Labs [9] and used both for investigating and extending genetic programming and for solving real-world problems. We decided that it would be particularly valuable to establish how well GPLab (and genetic programming in general) worked "out of the box", so we left the many parameters that control the run set to default values established as ones that have seemed to work well on problems in the past.

In particular, we ran with a population of 10,000 candidates in each generation. The initial population consisted of expression trees between four and seven levels deep over an operator set consisting of the variables *fpr*, *w*, *d*, and *len*, real-valued addition, subtraction, multiplication, and division, integer constants between –10 and 10, and real constants between –10 and 10, with a resolution of 0.001. Division was defined such that division by zero resulted in a value of one. We were frankly skeptical that a good solution would be found with such a restricted set of operators, but it seemed the most honest way to get a baseline.

Half the cases (500) were randomly selected for use as training cases. The other half were split into two validation sets to be used in monitoring the progress of the run and choosing the "winner". A separate set of 1,000 cases was generated for testing the chosen function after the run. Each candidate function was presented with a set of 50 training cases, biased toward those adjudged "more difficult" by a reinforcement learning mechanism. The fitness pressure was toward a lower mean relative error. Ties were broken by the number of nodes in the expression tree and then by the mean absolute error in the prediction. Candidates which attempted to evaluate more than 100 operators were judged "infinitely bad".

In each generation, a randomly chosen 10% of the top 25% of the population (250 candidates overall) were tested on the entire set of training and validation cases, and the ones that performed better than the current best on the training set, either of the two validation sets, or the entire set were preserved for later analysis. The overall winner would be the final leader over all 1,000 cases.

Reproduction occurred generationally, with parents chosen by tournament selection with a tournament size of three. In each generation the best candidate on the tests encountered was preserved. For the remainder, 77% (10/13) were a product of crossover, 8% (1/13) were copied from the parent generation, 8% (1/13) were modified by point mutation (the replacement of the operator of a node by another operator with a compatible signature), and the remaining 8% (1/13) were modified by applying a Gaussian drift to a constant. "Classical" genetic programming mutation, in which a new subtree is grown, was not used.

The experiment ran for 1,000 generations.

3 Other Methods

Before discussing the results obtained through genetic programming, it is worthwhile to consider, as a baseline, the performance that can be obtained via other machine learning methods. Note that the results presented in this section do not represent a serious investigation into the optimal parameterization of any of the models used, nor are we experts in the use of most of them. Even so, the results indicate the level of difficulty that this problem presents for traditional machine learning techniques.

The experiments in this section were performed using the WEKA toolkit (version 3.4) [20]. For purposes of comparison, all negative predicted values were clipped to zero. All predictors were trained on the same 1,000 training cases and evaluated on the same 1,000 testing cases used in the genetic programming experiments.

Linear regression resulted in the following model:

$$E[nfp] = 8.0691\,fpr - 0.0056\,w - 0.0275\,d + 0.0053\,len - 1.0497 \ , \tag{5}$$

which has a mean relative error of 264.2%, a median of 32.158%, a maximum of 60,550%, and a 95th percentile of 550.2%. Some of this error is due to predicting negative numbers, which are physically impossible. When negative predictions are clipped to zero, the model has a mean relative error of 64.03%, a median of 32.16%, a maximum of 4,637%, and a 95th percentile of 121.19%. The correlation with the actual values is 0.855. Clearly, such a model would be useless as a predictor.

Other methods tried (and their respective mean relative errors) include Radial Basis Function Networks (544.92%), Conjunctive Rules (305.34%), Decision Stump (one-level decision trees, 305.20%), Locally Weighted Learning [2] (247.53%), Neural Network [16] (113.34%), K* [4] (90.34%), Decision Table (70.65%), *k*-Nearest Neighbor [1] (68.41% with $k = 1$ and $k = 5$), Pace Regression [21] (64.03%), Simple (one-term) Linear Regression (53.73%), Sequential Minimal Optimization [17] (47.18%), Least Median Squared Linear Regression [15] (43.16%), REP Trees (35.20%), M5 [8] (15.55%), and M5 Prime (11.22%). None of these methods was able to get an expected error of less than 10%, and only three of them (K*, M5, and M5 prime) were even able to get the median error below 10. The best of the techniques, M5 prime had a median relative error of 5.23%, maximum of 303.45%, a 95th percentile of 41.68%, and a correlation with the actual values of 0.987.

4 Results

In the initial random population of our genetic programming run, the best solution was equivalent to

$$fpr + \frac{fpr(len - d)}{w} \quad . \tag{6}$$

This had a mean relative error of 17.87% over all of the cases. The median relative error was 12.80%, the maximum was 83.87%, and the 95th percentile value was 50.51%. The correlation with the actual values was 0.822. This can be used as an estimate of how well one can expect to do by guessing a function 10,000 times. By contrast, the best linear regression model has a mean relative error of 286.50%, a median relative error of 34.333%, a maximum relative error of 63,469.47%, and a 95th percentile error of 565.25%, but its correlation is slightly better, at 0.832.

Looking at Fig. 1, in which the x-axis represents the cases sorted by the value of the target function, we see that the best random function (medium gray triangles) is a nice match for the target function (black diamonds) when the target is small, but for approximately the last third of the range it begins to seriously underestimate the value. By contrast, the linear model (light gray squares) is closer for the larger values, but underestimates the smaller ones. This leads to its significantly high relative error, even though the correlation is slightly higher.

As the run progressed, the "overall best so far individual" rapidly improved. The mean relative error was below 15% by generation 14, below the 11.22% of the best other method by generation 22, below 10% by generation 30, and below 5% by generation 144. The experiment took place on a 2.0 GHz Pentium 4 processor, and each generation took roughly 8.5 seconds. The entire run of 1,000 generations took two hours and 23 minutes. The progress can be seen in Fig. 2. The heavy black line tracks the mean relative error, the medium gray line at the bottom tracks the median relative error, and the light gray line at the top tracks the 95th percentile relative error. The maximum error (not shown on the chart) begins at 84% and only exceeds 100%

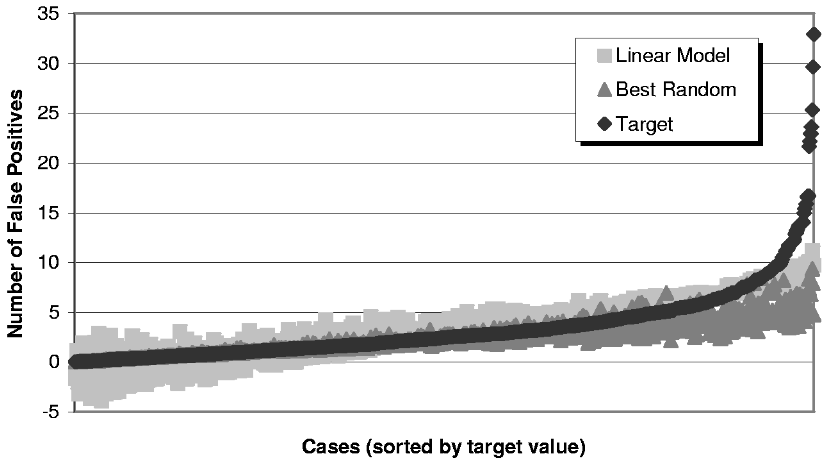

Fig. 1. Target function, best-fit linear model, and best overall candidate from initial random generation. The data points are sorted by target value

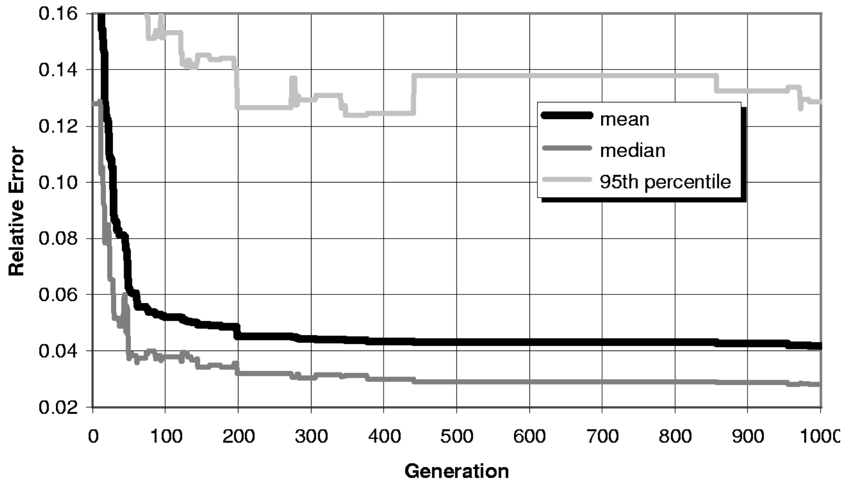

Fig. 2. Statistics for "overall best-so-far" candidate

for two generations (one "best-so-far" candidate). After generation 45, it is only greater than 60% occasionally between generations 280 and 440. After generation 200, not only are nearly all of the errors less than 14% (and half of them less than 3.2%), but the absolute worst case is almost always less than 60%.

The overall best candidate appeared in generation 990, after two hours and 22 minutes of unattended experiment time. This candidate had an overall mean relative error of 4.183%. The median relative error was 2.823%, the 95th percentile relative error was 12.862%, and the maximum relative error was 45.353%. Looking at absolute error (which was *not* a primary fitness pressure), the mean error was 0.147, the median error was 0.051, the 95th percentile error was 0.581, and the maximum error was 8.992. The overall best candidate did only slightly better (4.041% vs. 4.253%) on cases in the training set than it did on cases outside of the training set. When tested on a new set of 1,000 training cases, the overall best candidate did nearly identically. The mean relative error was 4.268%, the median was 3.036%, the 95th percentile was 13.395%, and the maximum was 26.72%. The correlation with the target was 0.993.

The response curve for the overall best candidate can be seen in Fig. 3. Here we see that the solution (gray circles) is a near-perfect fit to the target (black diamonds). The overall best candidate for the run had 95 nodes in its tree, and when simplified by *Mathematica* was equivalent to

$$\frac{fpr\,(\frac{w\,fpr^2}{10}+fpr+d)\,(len-d+\alpha\,fpr+0.4561\,w-\beta)}{w(d-0.07823\,fpr^2\,w)} \quad (7)$$

where

$$\alpha = 1.851479 + len\left(\frac{fpr}{fpr-6.47463}+\frac{len}{len^2-0.2127\,d}-\frac{d}{w}+1\right)$$
$$\beta = \frac{0.0502266\,fpr\cdot w(d\cdot len\cdot w-1)}{(d+9\,fpr)\,len\cdot d} \quad (8)$$

It is relatively easy to see this as a refinement of the

$$fpr+\frac{fpr(len-d)}{w} \quad (9)$$

found in the initial generation. As with many evolved solutions, this one involves a fair number of "magic numbers", and they should always be looked upon with some suspicion. Constants drift slowly during a run, and often there may be a better number in the near vicinity. Very small numbers often indicate that a spurious term is being driven toward zero. But too much should not be made of this skepticism. The suspiciously small constant, 0.07823, multiplied by the square of a variable that averages 0.5, looks to indicate that the term is useless. But if the denominator is replaced by $w\cdot d$, the solution actually becomes significantly worse, having a mean relative error of 8.269%, a median relative error of 5.631%, and a correlation of 0.968. On the other hand, 0.07823, while close, does not appear to be optimal. Investigating by hand within *R* [19] shows that the minimum mean relative error is 4.244% (from 4.268%) when the constant in that position is 0.082496. Such a difference is probably not worth a tremendous amount of effort, but it does show that evolved solutions may not be optimal, especially for a novel data set.

5 Overfitting

One of the prime fears when learning a function from data is that one will find a solution that works well on the training cases but which does not generalize to other

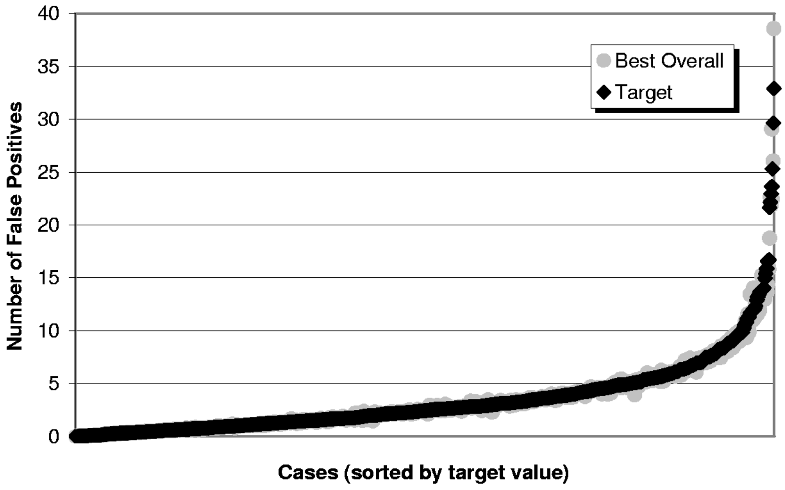

Fig. 3. Target function and best overall candidate, found in generation 986. The data points are sorted by target value

cases. This is known as *overfitting* the training data. Many learning methods, including genetic methods, are susceptible to this problem, but in our experience, genetic programming appears to be less likely to overfit even a reasonably small number of training cases than other methods.

Clearly the overall best candidate did not overfit the data, as shown by its performance on a set of test cases that were completely different from the ones used during its evolution. But ten million candidate programs were evaluated, and 72 were selected as "overall best so far", so if the experiment had been stopped earlier a different candidate would have been chosen as the overall best, and it is possible that that candidate would not have been as general.

One way to get confidence of the generality of a solution is to watch how well the solution does on out-of-set cases. Fig. 4 shows the performance of the best-so-far candidate, selected based on overall performance, evaluated over all the cases (black line) as well as over the training cases (medium gray line) and validation cases (light gray line) separately. If a candidate is overfitting the data, we should see the training error decrease while the out-of-set error remains high. Plainly, this is not happening. Indeed, for a plateau from generation 440 through generation 856, the performance on validation cases was better than the performance on training cases.

It should be stressed that genetic programming is not immune to overfitting, and it is even the case that a single run may alternate general and overfit solutions vying for the "top spot". But in our experience, the technique appears to be much more likely to find a general solution than most other techniques, perhaps because overfitting results in more complicated functions than general solutions, and simpler functions tend to be found first. It certainly seems to be the case that if there *is* as reasonably simple exact solution (that is, one that can be exactly described in an expression containing less than, say, 50 or so operators) it is much more likely to be found than a solution which just happens to fit the data.

6 Repeatability

Having analyzed the results of the run, the question arises as to whether we were merely lucky. Genetic programming is a highly nondeterministic process, and it might be the case that this particular run was anomalous, and that there's no particular reason to believe that other runs will do as well. To test this, 25 more runs were performed with identical parameters and training cases.

As can be seen in Fig. 5, the run described was actually the among the *worst* of the runs. It resulted in a solution with a mean relative error of 4.133% on the training and validation sets, which was worse than all but one of the subsequent runs. On the testing set, the primary run solution's mean relative error of 4.268% was worse than all but two of the 25 subsequent runs.

The mean relative error over the 25 runs averaged 3.500% on the training and validation sets and 3.675% on the testing set. On the testing set, the expected median was 2.473%, the maximum was 34.575%, and the 95th percentile was 11.566%. For the best of the runs, the mean relative error on the testing set was 3.242%, for the worst it was 4.429%, and for the median run it was 3.504%.

The primary run took 986 generations to reach its in-set level of 4.183%, while the other 25 runs only required, on average, 187 generations to surpass that level, with 13 doing so in less than 100 generations, nine in less than fifty generations, and one requiring only 24 generations. By generation seventeen, the expected solution performed better than all of the models produced by the other methods we tried. Some of this is due simply to the increased expressiveness of the genetic programming representation. In the initial random population, the expected mean relative error (equivalent to guessing 10,000 times) was 27.126% and the minimum over all 25 runs (equivalent to guessing 250,000 times) was 20.855%.

Looking at Fig. 5, it is apparent that several of the runs did strikingly better than the others. When the best-so-far candidates are examined, it becomes apparent that, probably due to initial random variation, these two runs refined a different strategy

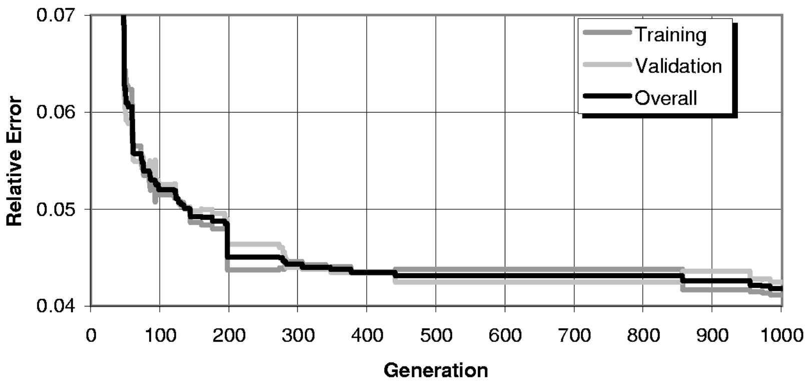

Fig. 4. Mean relative error overall compared with mean relative error on training cases and validation cases

from the other three runs. In runs that converge slowly, the top level of the recorded trees tend to be multiplications by *fpr*. That is, the population discovered early on that the problem could be thought of as finding an appropriate multiplier for the false positive rate, and the evolutionary process discovered better and better approximations to that multiplier. In runs that converge quickly, by contrast, the root of the tree for most of the run is a division operator, and in many cases the numerator is simply *len* or something that contains *len* as a principal component.

In all of the runs, however, both strategies are seen early on, so it appears that the strategy of treating the problem as essentially a division works better *provided* that it does well enough soon enough. Otherwise, it appears that that strategy gets bred out of the population by a somewhat more resilient strategy of multiplying by *fpr*, which improves more slowly and never seems to get to the levels of the division strategy.

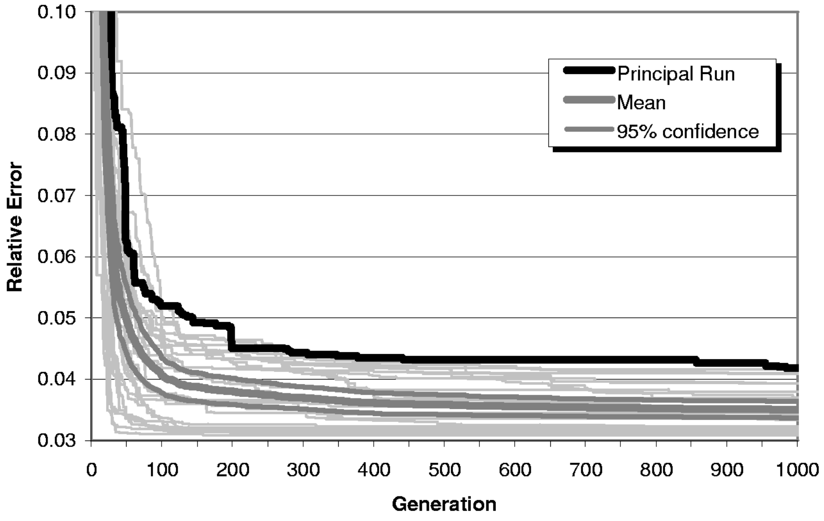

Fig. 5. Mean relative error on overall best-so-far candidates for 25 runs

There are more advanced techniques which can be used to counter—or even take advantage of—such tendencies. On the one hand, there is *deme-based* evolution [18], in which there are multiple (possibly asynchronously) evolving populations which, every so often, send good representatives to one another. This can keep a population from getting stuck in a rut and can permit the combination of mature strategies. On the other hand, there are methods for combining the results of independent runs to get solutions that are more stable than can be expected from any given run—and may, in fact, be better than any of the individual solutions. The latter technique is especially useful when it is feared that an individual solution has a small chance of being unusably bad but there is no way of determining this before it is used. Combining multiple runs can result in a solution that is much less likely to perform poorly, and, indeed can, in some cases, result in a solution that outperforms any individual solution. In the current experiment, sampling 1,000 five-solution "teams" from the 25 solutions generated and considering the prediction of each team to be the mean of the predictions of its members drops the expected mean relative error from 3.675% to

3.366%. Dropping the high and low predictions before taking the mean drops it further to 3.317%.

7 Solution Complexity

Having shown that genetic programming can produce useful results without worrying about customizing the parameters, we now look at what happens when you change one of them.

The solutions discovered all bumped up against the evaluation budget of 100 operations used as a run parameter. Since there were no loops, conditionals, or automatically defined functions in the allowed operator set, this means that the resulting expression trees all had nearly 100 nodes. This is a commonly-seen property of longer genetic programming runs, in which the population will experiment with strategies having different sizes and shapes for the first few dozen generations until it settles on the best strategies having at or near the maximal number of operators, spending the rest of the run refining this strategy. With nearly a hundred nodes, the expressions can get, as we have seen, quite complex, and it is often extremely difficult to understand them without a fair bit of interaction with a tool such as *Mathematica.* Of course, often we really don't *need* to understand the result, we merely need a result we can have confidence in. But if understandability is important, it can be useful to investigate whether a useful result can be obtained with a smaller allowable tree size.

When this problem was run with an evaluation budget of fifty operations, the mean relative error on the testing set over 25 runs was 3.716%, only slightly worse than the 3.675% with a budget of 100 operations. The expected median was 2.480%, the expected maximum was 29.339%, and the expected 95th percentile was 12.046%. The best run had a mean relative error of 3.195%, the worst run 5.292%, and the median run 3.309%.

Pushing further, an evaluation budget of only 25 operations appears to be sufficient. The first such run yielded a result which simplified to

$$\frac{fpr\,(len-d+0.273\,w\,(2-fpr))}{w\,(1-fpr)+fpr\cdot d} \tag{10}$$

in generation 301. This solution had a mean relative error of 3.238% (3.340% out-of-set), a median of 2.054%, a 95th percentile of 10.377%, and a maximum of 24.492%. The second run yielded a result which simplified to

$$fpr\left(\frac{len}{w\,(1-fpr)+fpr\,(d+2.7491)}+\frac{0.862299\,w-1.8623\,d}{d+w}\right) \tag{11}$$

in generation 556. This solution had a mean relative error of 3.379% (3.3095% out-of-set), a median of 2.224%, a 95th percentile of 11.241%, and a maximum of 25.643%.

When 25 runs were performed with an evaluation budget of 25 operations, the mean relative error (on the testing set) of the solutions ranged from 3.304% to 9.600%, with a median of 3.744% and a mean of 4.824%.

There does appear to be a lower limit, however. The first run with an evaluation budget of 15 operations was unable to produce an individual better than one equivalent to

$$\frac{fpr \cdot d}{w(1-fpr)+fpr \cdot d} \tag{12}$$

found in generation 44. This candidate had a mean relative error of 7.112% (7.0114% out-of-set), a median of 5.208%, a 95th percentile of 20.533%, and a maximum of 48.520%. When 25 runs were performed with an evaluation budget of 15 operations, the best solution found did no better than a 4.505% mean relative error on the testing set, and the worst run was 11.651%, the median run 5.382%, and the mean was 7.426%.

While clearly these solutions are not as good as the ones obtained when the candidates are allowed to be a bit larger, the fact that several runs investigated hit on similar functions indicates that this is probably to some extent a "strongest component". Unfortunately further runs based on both 15-node and 25-node solutions were unable to find any signal in the residuals.

8 Population Size

Another parameter worth looking at is population size. To investigate this, 25 runs were performed at each of the following population sizes: 20,000, 5,000, 1,000, 500, and 100. All other parameters were kept the same.

As discussed above, over 25 runs of 1,000 generations each with a population size of 10,000 candidates, the solutions ranged from a mean relative error (on the testing set) of 3.242% to 4.429% with a median of 3.504% and a mean of 3.657%.

When the population size is raised to 20,000, the solutions ranged from a mean relative error of 3.193% to 4.300%, with a median of 3.322% and a mean of 3.509%. At a population size of 5,000, the solutions ranged from a mean relative error of 3.207% to 4.969%, with a median of 3.913% and a mean of 3.943%. At a population size of 1,000, the solutions ranged from a mean relative error of 3.397% to 8.181%, with a median of 4.696% and a mean of 5.004%. At a population size of 500, the solutions ranged from a mean relative error of 3.374% to 8.467%, with a median of 5.636% and a mean of 5.694%. At a population size of 100, the solutions ranged from a mean relative error of 3.908% to 37.855%, with a median of 9.842% and a mean of 13.174%.

9 Discussion

Although many techniques for solving or approximating recurrences are known, all of them are somewhat painful for the non-mathematician researcher who has a practical need for solving such recurrences. In this paper, we demonstrate the use of genetic programming as a fully automated, out-of-the-box solver for such a problem.

The recursive function took one argument and had three constant parameters, yielding a solution that is a function of four variables. The problem was real, and the solution was not known ahead of time. In fact, when we presented our result to a number of statisticians and operations-research experts, they were pleased with the quality of the solution and the simplicity of its form, and most of all they were surprised by the fact that we were able to find a closed-form approximation at all for the recurrence we had formulated.

In order to investigate just how well genetic programming would work on such a problem without requiring any special expertise on the part of the experimenter, we deliberately ran the experiment with an "out-of-the-box" configuration and what we assumed would be an overly restrictive set of operators. To our surprise, it worked very well. The solutions derived were not merely good enough, they were good—although they were not the perfect solutions we would have liked—and they were much better than the solutions derived using the out-of-the-box configurations of any of a suite of other machine learning techniques, none of which produced a usable result. The method showed absolutely no tendency to overfit, nor did it appear to optimize the single statistic of interest at the expense of others, and multiple runs demonstrated that the initial satisfactory result was a likely outcome of the method, and, indeed, that even better solutions could often be expected.

It is true that genetic programming does take far longer than many other techniques, but it took less than three hours from writing the scripts to generate the data set to having an answer, and for a real-world problem whose solution is to be used in other systems, three hours is perfectly reasonable—indeed for many such problems three *days* would be perfectly reasonable.

One question, of course, is whether we were simply lucky with this particular recurrence. Future work is to characterize the performance of this technique on many other classes of recursive functions, including those on which well-known techniques are known to yield closed forms (as we would like this to work reasonably well without any knowledge or guess as to what the class of solution should be). Early experiments regressing to the Fibonacci function, which has a known-good closed-form approximation, are promising.

References

1. Aha, D.W., Kibler, D., Albert, M.K.: Instance-Based Learning Algorithms. Machine Learning 6 (January, 1991) 37–66
2. Atkeson, C., Moore, A., Schaal, S.: Locally Weighted Learning. Artificial Intelligence Review 11 (April, 1997) 11–73
3. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming: An Introduction. Morgan Kaufmann (1998)
4. Cleary, J.G., Trigg, L.E.: K*: An Instance-Based Learner Using an Entropic Distance Measure. Proc. 12th Int. Conf. on Machine Learning, (1995) 108–114
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press (1990)
6. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics. Addison-Wesley (1989)
7. Hamilton, J.D.: Time Series Analysis. Princeton University Press (1994)
8. Holmes, G., Hall, M., Frank, E.: Generating Rule Sets from Model Trees. Australian Joint Conf. on Artificial Intelligence (1999) 1–12

9. Kirshenbaum, E.: GPLab: A Flexible Genetic Programming Framework. Tech Report HPL-2004-12, Hewlett-Packard Laboratories, Palo Alto, CA (2004)
10. Knuth, D.E.: The Art of Computer Programming, Vol. 1: Fundamental Algorithms. 3rd edn. Addison-Wesley (1997)
11. Kohavi, R.: The Power of Decision Tables. Proc. Euro. Conf. on Machine Learning. Lecture Notes in Artificial Intelligence, Vol. 914. Springer-Verlag (1995) 174–189
12. Koza, J.R.: Genetic Programming. MIT Press (1992)
13. Lueker, G.S.: Some Techniques for Solving Recurrences. ACM Comp. Surv. 12(4) (1980)
14. Makridakis, S, Wheelwright, S.C., Hundman, R.J.: Forecasting: Methods and Applications 3rd edn. John Wiley & Sons (1998)
15. Rousseeuw, P.J., Leroy, A.M.: Robust Regression and Outlier Detection. John Wiley & Sons (1987)
16. Rumelhart, D.E., McClelland, J.L., PDP Research Group: Parallel Distributed Processing. MIT Press (1986)
17. Smola, A.J., Scholkopf, B.: A Tutorial on Support Vector Regression. NeuroCOLT2 Technical Report Series NC2-TR-1998-030, 1998.
18. Tackett, W.A., Carmi, A.: The Donut Problem: Scalability, Generalization and Breeding Policies in Genetic Programming. In: Kinnear, K.E. (ed.), Advances in Genetic Programming. MIT Press (1994)
19. Venables, W.N., Smith, D.M., R Development Core Team: An Introduction to R. http://cran.r-project.org/doc/manuals/R-intro.pdf
20. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools with Java Implementations. Morgan Kaufmann (2000)
21. Wang, Y., Witten, I.H.: Modeling for Optimal Probability Predictions. Proc. 19th Int. Conf. of Machine Learning (2002)
22. Wolfram, S.: The Mathematica Book. 4th edn. Wolfram Media, Inc. and Cambridge University Press (1999)