Chapter 14

PROFILING SYMBOLIC REGRESSION-CLASSIFICATION

Michael F. Korns[1] and Loryfel Nunez[1]

[1] *Investment Science Corporation, 1 Plum Hollow, Henderson, Nevada 89052 USA*

Abstract This chapter performance-profiles a composite method of symbolic regression-classification, combining standard genetic programming with abstract grammar techniques, particle swarm optimization, differential evolution, context aware crossover, and age-layered populations. In two previous papers we experimented with combining high performance techniques from the literature to produce a large scale symbolic regression-classification system. In this chapter we briefly describe the current combination of techniques and make the assertion that Symbolic Regression-Classification (via Genetic Programming) is now ready for some industrial strength applications. This aggressive assertion is supported with performance metrics on multiple large scale problems which are scored on testing data which is always distinct from the training data. Each problem contains ten thousand fitness cases (rows) and problems vary in complexity from one to thirty independent variables (columns). In each problem, the dependent variable is generated by a test expression which varies in complexity from simple linear to complex non-linear often including conditionals (if-then-else clauses) with random noise starting at zero percent and rising up to forty percent.

Keywords: artificial intelligence, genetic programming, particle swarm optimization, differential evolution, portfolio selection, data mining, formal grammars, quantitative portfolio management

1. Introduction

This is the continuing story of the issues encountered by Investment Science Corporation in using composite genetic programming techniques to construct a large-scale, time-constrained symbolic regression-classification tool for use with financial data.

In two previous papers (Korns, 2006; Korns, 2007) our pursuit of industrial scale performance with large-scale, time-constrained symbolic regression

problems, required us to reexamine many commonly held beliefs and, of necessity, to borrow a number of techniques from disparate schools of genetic programming and "recombine" them in ways not normally seen in the published literature. We continue this tradition in this current paper, building a symbolic regression-classification tool combining the following disparate techniques from the genetic and evolutionary programming literature:

- Standard tree-based genetic programming
- Vertical slicing and out-of-sample scoring during training
- Grammar template genetic programming
- Abstract grammars utilizing swarm intelligence
- Context aware cross over
- Age-layered populations
- Random noise terms for learning asymmetric noise
- Bagging

For purposes of comparison, all results in this paper were achieved on two workstation computers, specifically an Intel©Core 2 Duo Processor T7200 (2.00GHz/667MHz/4MB) and a Dual-Core AMD Opteron©Processor 8214 (2.21GHz), running our Analytic Information Server software generating Lisp agents that compile to use the on-board Intel registers and on-chip vector processing capabilities so as to maximize execution speed, whose details can be found at *www.korns.com/Document_Lisp Language_Guide.html.*

Fitness Measure

Standard regression techniques often utilize least squares error (LSE) as a fitness measure. In our case we normalize by dividing LSE by the standard deviation of "Y" (dependent variable). This normalization allows us to meaningfully compare the normalized least squared error (NLSE) between different problems.

Of special interest is combining fitness functions to support both symbolic regression and classification of common stocks into long and short candidates. Specifically we would like to measure how successful we are at predicting the future top 10% best performers (*long candidates*) and the future 10% worst performers (*short candidates*) (Korns, 2007).

Briefly, let the dependent variable, Y, be the future profits of a set of securities, and the variable, EY, be the *estimates* of Y. If we were prescient, we could automatically select the best future performers *actualBestLongs, ABL,* and worst

future performers *actualBestShorts, ABS,* by sorting on Y and selecting an equally weighted set of the top and bottom 10%. Since we are not prescient, we can only select the best future estimated performers *estimatedBestLongs, EBL,* and estimated worst future performers *estimatedBestShorts, EBS,* by sorting on EY and selecting an equally weighted set of the top and bottom 10%. If we let the function *avgy* represent the average y over the specified set of fitness cases, then clearly the following will always be the case.

```
-1 <= ((avgy(EBL) - avgy(EBS)) / (avgy(ABL) - avgy(ABS))) <= 1
```

We can construct a fitness measure known as tail classification error, TCE, such that

```
TCE = ((1 - ((avgy(EBL) - avgy(EBS)) / (avgy(ABL) - avgy(ABS)))) / 2)
```

and therefore

```
0 <= TCE <= 1
```

A situation where TCE < .5 indicates we are making money speculating on our short and long candidates. Obviously 0 is a perfect score (we might as well have been prescient) and 1 is a perfectly imperfect score (other traders should do the opposite of what we do). Clearly, considering our financial motivation, we are interested in achieving superior regression fitness measures; but, we are also interested in superior classification. In fact, even if the regression fitness (NLSE) is poor but the classification (TCE) is good, we can still have an advantage, in the financial markets, with our symbolic regression-classification tool.

Since both the TCE and NLSE fitness measures are normalized, we can make standard interpretations of results across a wide range of experiments. In the case of NLSE, any score of .3 or less is very good (meaning the average least squared error is less than 30% of the standard deviation of Y), while a score of less than .5 is okay, NLSE scores greater than .5 indicate increasingly poor regression results. Our system automatically averages the estimates of the ten top champions (bagging) whenever the training NLSE of the top champion is greater than .5. Finally, a TCE score of less than .2 is excellent. A TCE score of less than .2 is good; while, a TCE of .5 or greater is poor.

Vertical Slicing

We make use of an out-of-sample testing procedure we call *vertical slicing*, wherein the rows in the training matrix X are sorted in ascending order by the dependent values, Y. Then the sorted rows in X are subdivided into S *vertical slices* by selecting every S-th row to be in each vertical slice. Thus the first vertical slice is the set of training rows as follows *X[0], X[S], X[2*S], ...* . We train on a single vertical slice, but test on all vertical slices (Korns, 2006).

Since Y represents the *behavior* of the system to be learned, sorting X by Y insures that each vertical slice contains training examples equally distributed across the range of behaviors of the system. For complete training coverage,

every epoch we randomly select a different vertical slice as the training subset while still scoring fitness across every fitness example in X.

Vertical slicing reduces training time (which in multiple regression and swarm grammars can be time consuming); while simultaneously reducing over fitting by scoring fitness over all slices (out-of-sample testing).

Abstract Grammar

Recently, informal and formal grammars have been used in genetic programming to enhance the representation and the efficiency of a number of applications including symbolic regression – see overviews in (O'Neill and Ryan, 2003) and (Poli et al., 2008).

Our system implements a hybrid combination of tree-based GP and formal grammars where the head of each sublist is a grammar rule with polymorphic methods for mutation, crossover, etc. Different grammar rules communicate with each other by message passing. We use standard mutation and crossover operations (Koza, 1992) and support both simple regression and multiple regression, as follows: *regress(expression)*; and *regress(expression,...,expression)*.

Our numeric expressions are JavaScript-like containing the variables *x0* through *xm* (where *m* is the number of columns in the regression problem) and real constants such as *2.45 or -34.687*, with the following binary and unary operators +, -, /, %, *, <, <=, ==, ! =, >=, >, expt, max, min, abs, cos, cosh, cube, exp, log, sin, sinh, sqroot, square, tan, tanh, and the ternary conditional expression operator if (...) then ... else ...;

In (Korns, 2006), we implemented both the production and recognition side of our grammars. Later in (Korns, 2007) we developed an abstract grammar which is evaluated using swarm intelligence and provides excellent fine grain control during evolution.

The enhancement of abstract real constants *c0* through *ck* (where *k* is the number of unique abstract real constants in the expression) allows more fine grain control over the evolution of optimal real numbers during the GP process.

For instance, the following concrete expression *regress(3.4*sin(x3))*, when evaluated, has a fitness score based upon regressing 3.4 times the sine of variable three. However, the following abstract expression *regress(c0*sin(x3))*, which must be evaluated in a swarm agent, has a fitness score based upon the swarm agent's choice of optimal real constant for *c0*. Our experience is that swarm intelligence is an excellent method of optimizing specific real constants.

Additionally, the enhancement of abstract variables *v0* through *vj* (where *j* is the number of unique abstract variables in the expression) allows more fine grain control when optimizing formulas with a specific grammatical format.

For instance, the following abstract expression *regress(c0*sin(v0))*, which must be evaluated in a swarm agent, has a fitness score based upon the swarm

agent's choice of actual real constant for *c0* and the swarm's choice of actual concrete variable *v0*. An example of an optimized champion from the swarm agent might be *regress(-.416*sin(x10))*. One is always assured that the final swarm optimized champion will be in a form compatible with the abstract grammar expression supplied. The possibilities for addressing bloat are obvious.

This year, as an extension of our previous experiments with abstract grammar, we introduce abstract random noise terms *e0* through *ek* for learning asymmetric noise (Schmidt and Lipson, 2007). For instance, the following abstract expression *regress(c0*sin(v0+e0))*, which must be evaluated in a swarm agent, is evaluated iteratively an additional number of times with random values between -1 and +1 replacing the *e0* noise term. The fitness score is based upon the swarm agent's choice of actual real constant for *c0* and the choice of actual variable for *v0*, while the noise term *e0* is randomly fluctuated between -1 and +1 as described in greater detail in (Schmidt and Lipson, 2007).

By using an abstract expression grammar with imbedded swarm optimization, we achieve more fine-grained control, of numeric constant optimization, expression bloat, and learning asymmetric noise.

Context-aware Crossover

In (Majeed and Ryan, 2006) an extension of standard GP crossover was devised. In standard GP crossover (Koza, 1992), a randomly chosen snip of genetic material from the father s-expression is substituted into the mother s-expression in a random location. In context-aware crossover, a randomly chosen snip of genetic material from the father s-expression is substituted into the mother s-expression *at all possible valid locations*. Where standard crossover produces one child per operation, context-aware crossover can produce many children *depending upon the context* (Korns, 2007).

We further extended context-aware crossover such that *all possible valid* snips of genetic material from both the mother and father are substituted into both parents *at all possible valid locations*. Each of the many offspring are evaluated with only the survivors being added to the pool. We add our extended context-aware crossover to all GP runs in our system with a increasing probability with each additional generation.

Context-aware crossover holds-forth the promise of greater coverage of the local search space, as defined by the candidate s-expressions' roots and branches, and, therefore, a greater control of the evolutionary search at a fine grain level.

Age-Layered Populations

In (Hornby, 2006) a technique is introduced, known as *aged-layered population structure* (ALPS), devised to minimize premature population convergence.

ALPSs evolve populations in several significant ways. New random populations are generated at irregular intervals over the duration of the evolutionary process. Populations are segmented by the age of the individuals with evolutionary competition restricted to individuals within roughly the same age cohort (young individuals are not allowed to be swamped by more mature individuals). At irregular intervals, younger champions are promoted into populations allowing competition with older individuals (Hornby, 2006).

ALPS differs from a typical evolutionary process by segregating individuals into different age-layers and by regularly introducing new, randomly generated individuals in the youngest layers. The introduction of randomly generated individuals at regular intervals results in an evolutionary process that is never completely converged and is always exploring new parts of the fitness landscape. By restricting the age of competitors and breeding within an age cohort, younger individuals are able to develop without being swamped by older ones. Analysis of the behavior of ALPS finds that individuals that are randomly generated mid-way through a run are able to move the population out of mediocre local optima to better parts of the fitness landscape if they are allowed to develop in their youth, free from unfair competition by more mature adults.

Experimental Setup

Our goal is to profile the performance of our *genetic symbolic regression-classification machine* (GSM) on a series of symbolic regression-classification problems. All of our symbolic regression-classification problems consist of a training phase and a distinct testing phase. In the training phase, an (N x M) real number matrix, X, is filled with random numbers[1]. Then a training model, f(x), is selected to create a real number N-dimensional vector of dependent variables, Y, as follows:

```
Y = f(X) + noise;   vector equation
```

Training models vary from simple linear all the way to complex nonlinear. For example:

```
y = 1.57 + (1.57*x0) - (39.34*x1);   simple linear
y = ((1.57*x0)%(39.34*x1))
      + (if (log(x0) == x2) {sin(2.13*x2)}
    else {((39.34*x1)%(2.13*x2))});   complex nonlinear
```

In (Korns, 2007) we published nine base training models which vary from simple linear to complex nonlinear. In this paper we generate numerous training models at random. These training models vary from simple linear to complex

[1]N refers to number of rows in the matrix (always 10,000 rows unless otherwise stated). M refers to the number of columns (columns vary from 1 to 30 depending upon test difficulty). The range of the random numbers is from -50 to +50 unless otherwise stated.

nonlinear[2]. Furthermore, our problem set varies from one column problems to more difficult thirty column problems, and from zero random noise to problems with 40% random noise, which is generated as follows:

```
y = (y * 0.8) + (y * random(0.4));
```

The output of the training phase is a champion estimator model, *ef*, which, when evaluated on the training matrix, generates an estimated N vector, EY:

```
EY = ef(X) + noise;   estimator model
```

In the testing phase, another (N x M) real number matrix, TX, is filled with random numbers. Then the original training model, f(x), is evaluated on TX to create a real number N vector of dependent testing variables, TY. Then the champion estimator model, ef, is evaluated on the testing matrix, TX, and the normalized least squares error (NLSE) difference between EY and TY is the final out-of-sample testing score.

For each of the numerous experimental problems attempted for this profiling study, we collected the following important data: Unique Problem Identifier; Training generations to completion; Training noise; Training minutes to completion; Training normalized least squared error (NLSE); Testing normalized least squared error (NLSE); Testing classification error (TCE); Champion estimator model; and the Training model.

Results on Nine Base Problems

The results of training on the nine base training models on 10,000 rows and twenty columns with 40% random noise and only 20 generations allowed, are shown in Table 14-1.

Fortunately, training time is very reasonable given the difficulty of some of the problems and the limited number of training generations allowed. In general, average percent error performance is poor with the *linear* and *cubic* problems showing the best performance. Extreme differences between training error and testing error in the *mixed* and *ratio* problems suggest over-fitting. Surprisingly, long and short classification is fairly robust in most cases with the exception of the *hidden*, *ratio* and *hyper* test cases. If we were to run a market neutral hedge on hypothetical markets, driven by these nine test models, we would have made money in all but one of the markets, made a little money in the markets driven by the *hidden* and *hyper* models, broken even in the market driven by the *ratio* model, and made excellent money in all other markets.

The salient observation is the relative ease of classification compared to regression even in problems with this much noise. In five of the test cases, testing NLSE is either close to or exceeds the standard deviation of Y (not very good); however, in six of the test cases classification is below 20

[2]The complete set of randomly generated training models are described in detail on www.korns.com.

Table 14-1. Result for 10,000 rows by 20 columns with Random Noise. The columns are: *Test:* The name of the test case; *Minutes:* The number of minutes required for training; *Train-Error:* The average percent error score for the training data; *Test-Error:* The average percent error score for the testing data; and *Classify:* The classification score for the testing data.

Test	*Minutes*	*Train-Error*	*Test-Error*	*Classify*
cross	9	0.80	0.80	0.19
cubic	10	0.11	0.11	0.00
hyper	9	0.96	0.96	0.36
elipse	12	0.45	0.46	0.05
hidden	10	0.99	0.99	0.45
linear	10	0.11	0.11	0.00
mixed	12	0.69	1.85	0.07
ratio	26	0.95	1.18	0.46
cyclic	8	0.39	0.91	0.18

Table 14-2. Result For 10,000 rows by 5 columns no Random Noise. The columns are the same as in Table 14-1.

Test	*Minutes*	*Train-Error*	*Test-Error*	*Classify*
cross	107	0.37	0.39	0.02
cubic	0	0.00	0.00	0.00
hyper	369	0.00	0.00	0.00
elipse	0	0.00	0.00	0.00
hidden	3	0.00	0.05	0.00
linear	0	0.01	0.01	0.00
mixed	123	0.24	1.65	0.13
ratio	6	0.03	1.05	0.50
cyclic	4	0.04	0.14	0.06

The obvious comparison with the above difficult problems would be to try easier problems by eliminating the random noise, reducing the number of columns, and increasing the number of training generations allowed. The results of training on the nine base training models on 10,000 rows and *five columns* with *no random noise* and up to *200* generations allowed, are shown in Table 14-2.

Fortunately, reducing the problem difficulty greatly improves the results. In general, average percent error performance is now very good. Extreme differences between training error and testing error in the *mixed* and *ratio* problems

suggest over-fitting. Long and short classification is excellent in most cases with the exception of the *ratio* test case.

Now let us examine the performance of the GSM tool in the many problems whose difficulty falls in between the two extremes shown above. In the next few sections, we will profile the behavior of the tool on a wide range of randomly generated problems.

Results on Advanced Problems. Using the system's production grammar features, we generated thousands of symbolic regression problems ranging from 1 to 30 columns in complexity, from 0% to 40% noise, and containing simple root expressions as well as more difficult modal expressions. We gave each problem 20 generations in which to evolve a solution.

We classify the problems according to the level of difficulty in the column expressions generated for each symbolic regression problem. *Root* problems do not contain conditionals (if then else clauses). *Modal* problems may contain conditionals.

- Example of a one-column root expression:

```
y = (11.3665735467+(16.94203837131*(exp(x0) % abs(x0))));
```

- Example of a one-column modal expression:

```
y = (-20.29753816981+(12.1706446781*(if ((tanh(x0) - x0) >=
(abs(x0) - sqrt(x0))) ((x0*x0*x0) + (x0*x0))
else (exp(x0) - sign(x0)))));
```

There are thousands of separate, independent symbolic regressions tests reported in this paper. For each problem type (Root or Modal) there are 12 unique column-noise combinations. There are a total of 24 test cases all in all.

A test is considered excellent if its NLSE is less than .31 and reasonably good if its NLSE is less than .51. Additionally, a test is considered excellent if its TCE is less than .21 and reasonably good if its TCE is less than .31.

The results of training on the randomly generated training models on 10,000 rows and from 1 to 30 columns with 0 to 40 percent noise are shown in Table 14-3.

For trivial single-column problems, almost all tests show almost perfect classification scores even for problems with 40% noise. As we increase the number of columns, we lose our accuracy as seen from the increasing NLSE score. But even in our most difficult problems we are achieving over 70% excellent classification scores. This is unprecedented.

Effects Number of Column and Noise Levels. We randomly generated tests for the different test cases as seen in Table 14-3. We generated tests for the following number of columns: 1, 10, 20, and 30, with the following noise levels: 0%, 20%, and 40% for simple root expressions and for more difficult modal expressions.

Table 14-3. Result for 10,000 rows by 1-30 columns with 0-40% Random Noise. The top of the results are for the Root tests and the second half is for the Modal tests, as indicated in the left-most column. The rest of the columns are: *Cols:* The number of columns of data (1, 10, 20, 30); and *Noise:* The amount of noise added (0%, 20%, 40%); *Count:* Total number of such tests; *TrainNLSE:* Average training error for all the tests; *TestNLSE:* Average testing error for all tests; *TCE:* Average classify score for all tests; *% GoodReg:* Percent of tests with a test NLSE less than 0.31; and *% GoodClass:* Percent of tests with a TCE Score less than 0.21.

	Cols	Noise	Count	Train NLSE	Test NLSE	TCE	%Good Reg	%Good Class
R	**1**	0%	23	0.016	0.015	0.0007	100.00	100.00
O		20%	25	0.131	0.132	0.032	92.00	98.61
O		40%	25	0.158	0.161	0.014	96.00	95.83
T	**10**	0%	130	0.285	0.286	0.036	65.38	93.85
		20%	107	0.309	0.294	0.062	68.22	88.79
		40%	86	0.369	0.370	0.080	63.95	87.21
	20	0%	66	0.422	0.394	0.074	50.00	90.91
		20%	90	0.460	0.450	0.084	43.33	84.44
		40%	37	0.463	0.423	0.050	45.95	91.89
	30	0%	55	0.604	0.608	0.118	23.64	74.55
		20%	52	0.625	0.560	0.099	26.92	84.62
		40%	51	0.616	0.560	0.095	17.65	84.31
M	**1**	0%	47	0.043	0.034	0.003	97.87	100.00
O		20%	72	0.120	0.121	0.021	94.44	98.61
D		40%	48	0.243	0.246	0.054	77.08	95.83
A	**10**	0%	88	0.405	0.318	0.044	62.50	90.91
L		20%	117	0.439	0.390	0.048	56.41	90.60
		40%	39	0.466	0.462	0.036	46.15	97.44
	20	0%	68	0.444	0.455	0.062	44.12	92.65
		20%	69	0.630	0.577	0.093	30.43	86.96
		40%	85	0.552	0.534	0.059	37.65	89.41
	30	0%	80	0.649	0.579	0.080	30.00	86.25
		20%	65	0.690	0.658	0.101	15.38	83.08
		40%	117	0.730	0.676	0.118	17.09	75.21

On first glance at Table 14-3 and at Figures 14-1 and 14-2, the classification score (*TCE*) and the regression score (*TCE*) get worse as the number of columns and the level of noise increase. To test the separate and mutual effects of the number of columns, and noise levels on the classification score (*TCE*) and the Test error (*NLSE*) , we performed the ANOVA on our test results at 0.01 level of significance.

For the classification score (*TCE*), the ANOVA results are the same for both root and modal problems. The results show that the number of columns and noise level separately have significant F-values, with the noise level posing a

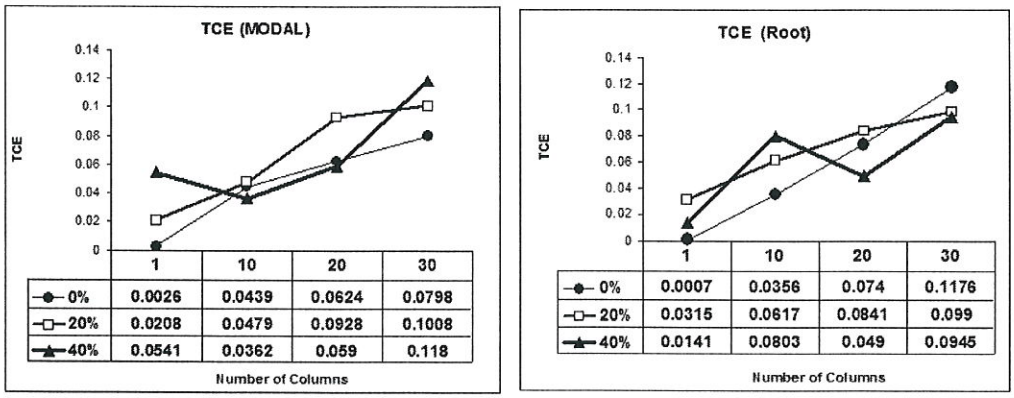

Figure 14-1. TCE Score Results for Root and Modal Tests, for 0%, 20%, and 40% noise levels, and 1, 10, 20 and 40 columns.

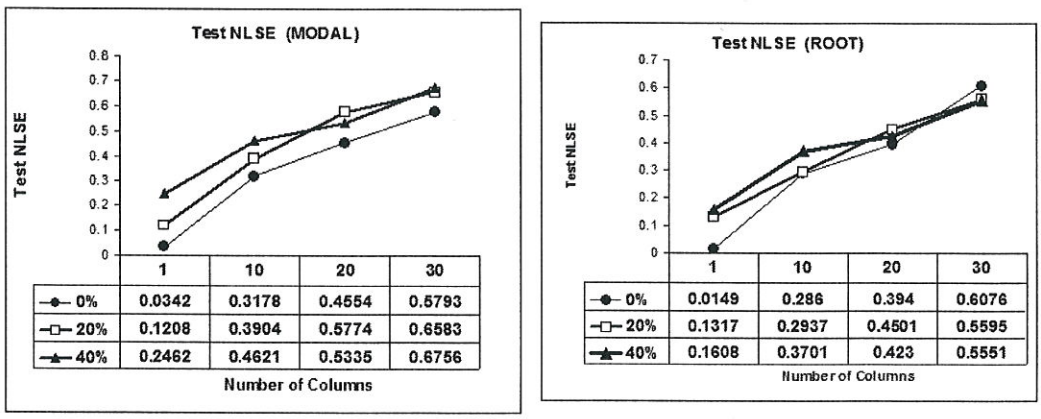

Figure 14-2. Test NLSE Score Results for Root and Modal Tests for 0%, 20%, and 40% noise levels, and 1, 10, 20 and 40 columns.

higher F-score than the number of columns. The ANOVA results for the interaction between noise level and number of columns is not statistically significant.

This means that increasing number of columns or the noise level alone worsens the TCE. Increasing the noise level has more effect in worsening the TCE than increasing the number of columns. Surprisingly, there is no statistical confirmation that an increase in the number of columns with a corresponding increase in noise level will significantly worsen the TCE.

For the regression score (*Test NLSE*), using the F-test at 0.01 level of significance, the ANOVA results for the interaction between noise levels and number of columns show very high F-values for both root and modal problems. However, for root problems, the number of columns or the noise level separately do not pose significant F-values. For modal problems, only the number of columns pose a significant F-value.

This shows that ANOVA results provide no statistically significant confirmation that there will a corresponding increase in the NLSE when we increase the

Table 14-4. 30-column problems with varying Noise Levels. The column headings are as follows: *Test:* Number of Tests; *20Gens:* Average Test NLSE or TCE for 20 generations; *200Gens:* Average Test NLSE or TCE for 200 generations; *Difference:* Test NLSE or TCE in 200 Generations; and subtracted by Test NLSE or TCE in 20 Generations.

			TEST NLSE			TCE		
		Tests	20 Gens	200 Gens	Diff	20 Gens	200 Gens	Diff
Modal	0%	32	0.870	0.632	0.238	0.297	0.125	0.171
	20%	33	0.785	0.714	0.070	0.125	0.146	-0.021
	40%	31	0.661	0.601	0.060	0.112	0.110	0.002
Root	0%	34	0.637	0.615	0.022	0.134	0.332	-0.197
	20%	26	0.716	0.578	0.139	0.138	0.115	0.022
	40%	19	0.568	0.524	0.044	0.083	0.136	-0.053

number of columns or the noise level separately for root problems. This result is different for modal problems where an increase in the number of columns alone would significantly worsen the NLSE. For both root and modal problems, it is an increase in the number of columns coupled with a corresponding increase in the noise level that worsens NLSE as seen from the resulting high F-scores for the noise-column interaction.

Effects Increasing the Number of Training Generations. We have increased the number of generations from 20 to 200 in our 30-column tests to see if the evolved solution is more accurate.

From the results in Table 14-4, Test NLSE Results are improved by increasing the number of generations for both modal and root problems. However, increasing the number of generations do not necessarily improve TCE results. In fact, in some cases due to over fitting, TCE scores after 200 generations are worse.

An additional uncompleted test is currently underway. The system has been given a complicated modal problem with five variables (columns), no noise, and 20,000 generations to train. At the time of this publication we see continuous improvement in the training NLSE. At 20 generations the training NLSE was .75, by 100 generations training NLSE had dropped to .65, by 600 generations training NLSE had fallen to .64, and at 7,000 generations the training NLSE is now down to .37.

There does not appear to be any hard barrier to continuous training improvement in this system (we believe this to be due to the ALPS strategy). However, we will not know until generation 20,000 whether or not the system has over fit (as happened in some cases above after only 200 generations).

Summary

Genetic Programming, from a corporate perspective, is ready for industrial use on *some* large scale, time constrained symbolic regression-classification problems. Adapting the latest research results, has created a symbolic regression tool whose efficiency is exciting.

While there is no evidence to suggest that the eight techniques included in this system are the absolute best, it is a credit to the scientists pioneering Genetic Programming that at least this combination of techniques has produced a system, which for hundreds of randomly generated difficult modal problems with 30 columns and 40% noise, we obtained an excellent classification score in 75% of the test cases.

Financial institutional interest in the field is growing while pure research continues at an aggressive pace. Further applied research in this field is absolutely warranted. We are using our *genetic symbolic regression-classification machine* (GSM) in the financial domain. But as new techniques are added and current ones improved, we believe that GSM has evolved to be a domain-independent tool that can provide superior regression and classification results for industrial scale symbolic regression problems.

Additional research separating the effects of each of the eight techniques on training time, and testing NLSE/TCE is necessary. Currently we need a more detailed understanding of the individual effects of each of these techniques and of their effects in specific combinations. Furthermore, we need more research on the effects of longer training times (20,000 generations and more) on system convergence and over fitting.

Clearly we need to experiment with techniques which will improve our performance on the modal test cases. The primary area for future research involves experimenting with statistical other types of analysis to help build conditional WFFs for difficult multi-modal problems.

References

Hornby, Gregory S. (2006). ALPS: the age-layered population structure for reducing the problem of premature convergence. In Keijzer, Maarten, Cattolico, Mike, Arnold, Dirk, Babovic, Vladan, Blum, Christian, Bosman, Peter, Butz, Martin V., Coello Coello, Carlos, Dasgupta, Dipankar, Ficici, Sevan G., Foster, James, Hernandez-Aguirre, Arturo, Hornby, Greg, Lipson, Hod, McMinn, Phil, Moore, Jason, Raidl, Guenther, Rothlauf, Franz, Ryan, Conor, and Thierens, Dirk, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 815–822, Seattle, Washington, USA. ACM Press.

Korns, Michael F. (2006). Large-scale, time-constrained symbolic regression. In Riolo, Rick L., Soule, Terence, and Worzel, Bill, editors, *Genetic Pro-*

gramming Theory and Practice IV, volume 5 of *Genetic and Evolutionary Computation*, chapter 16, pages –. Springer, Ann Arbor.

Korns, Michael F. (2007). Large-scale, time-constrained symbolic regression-classification. In Riolo, Rick L., Soule, Terence, and Worzel, Bill, editors, *Genetic Programming Theory and Practice V*, Genetic and Evolutionary Computation, chapter 4, pages 53–68. Springer, Ann Arbor.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Majeed, Hammad and Ryan, Conor (2006). Using context-aware crossover to improve the performance of GP. In Keijzer, Maarten, Cattolico, Mike, Arnold, Dirk, Babovic, Vladan, Blum, Christian, Bosman, Peter, Butz, Martin V., Coello Coello, Carlos, Dasgupta, Dipankar, Ficici, Sevan G., Foster, James, Hernandez-Aguirre, Arturo, Hornby, Greg, Lipson, Hod, McMinn, Phil, Moore, Jason, Raidl, Guenther, Rothlauf, Franz, Ryan, Conor, and Thierens, Dirk, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 847–854, Seattle, Washington, USA. ACM Press.

O'Neill, Michael and Ryan, Conor (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, Dordrecht Netherlands.

Poli, Riccardo, Langdon, William B., and McPhee, Nicholas Freitag (2008). *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

Schmidt, Michael D. and Lipson, Hod (2007). Learning noise. In Thierens, Dirk, Beyer, Hans-Georg, Bongard, Josh, Branke, Jurgen, Clark, John Andrew, Cliff, Dave, Congdon, Clare Bates, Deb, Kalyanmoy, Doerr, Benjamin, Kovacs, Tim, Kumar, Sanjeev, Miller, Julian F., Moore, Jason, Neumann, Frank, Pelikan, Martin, Poli, Riccardo, Sastry, Kumara, Stanley, Kenneth Owen, Stutzle, Thomas, Watson, Richard A, and Wegener, Ingo, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1680–1685, London. ACM Press.