Symbolic Regression of Conditional Target Expressions

Michael F. Korns

Freeman Investment Management, 1 Plum Hollow, Henderson, Nevada 89052 USA
mkorns@korns.com.

Summary. This chapter examines techniques for improving symbolic regression systems in cases where the target expression contains conditionals. In three previous papers we experimented with combining high performance techniques from the literature to produce a large scale, industrial strength, symbolic regression-classification system. Performance metrics across multiple problems show deterioration in accuracy for problems where the target expression contains conditionals. The techniques described herein are shown to improve accuracy on such conditional problems. Nine base test cases, from the literature, are used to test the improvement in accuracy. A previously published regression system combining standard genetic programming with abstract expression grammars, particle swarm optimization, differential evolution, context aware crossover and age-layered populations is tested on the nine base test cases. The regression system is enhanced with these additional techniques: pessimal vertical slicing, splicing of uncorrelated champions via abstract conditional expressions, and abstract mutation and crossover. The enhanced symbolic regression system is applied to the nine base test cases and an improvement in accuracy is observed.

Key words: Abstract Expression Grammars, Differential Evolution, Genetic Programming, Particle Swarm, Symbolic Regression.

1 Introduction

This chapter examines techniques for improving symbolic regression systems in cases where the target expression contains conditionals. In three previous papers (Korns, 2006), (Korns, 2007), and (Korns, 2008), our pursuit of industrial scale performance with large-scale, 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. The evolutionary techniques, as of the three previous papers, vetted for efficacy in symbolic regression are as follows:

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

While the above techniques, described in detail in (Korns 2008), produce a symbolic regression system of breadth and strength, performance metrics across multiple problems show deterioration in accuracy for problems where the target expression contains conditionals. Using the nine base test cases from (Korns, 2007) as a training set, to test for improvements in accuracy, we enhanced our symbolic regression system with these additional techniques which we will show improve accuracy:

- Pessimal vertical slicing
- Splicing
- Abstract mutation and crossover

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*. Furthermore, our Analytic Information Server is in the founding process of becoming an open source software project.

1.1 Testing Regimen

Our testing regimen uses only statistical best practices out-of-sample testing techniques. We test each of the nine test cases on matrices of 10000 rows by 5 columns with no noise, and on matrices of 10000 rows by 20 columns with 40% noise, before drawing any performance conclusions. Taking all these combinations together, this creates a total of 18 separate test cases. For each test a training matrix is filled with random numbers between -50 and +50. The target expression for the test case is applied to the training matrix to compute the dependent variable and the required noise is added. The symbolic regression system is trained on the training matrix to produce the regression champion. Following training, a testing matrix is filled with random numbers between -50 and +50. The target expression for the test case is applied to the testing matrix to compute the dependent variable and the required noise is added. The regression champion is evaluated on the testing matrix for all scoring (i.e. out of sample testing).

1.2 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*)[1].

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

[1] The concept of long short tail classification is described in detail in (Korns, 2007).

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 < 0.50 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 0.30 or less is very good (meaning the average least squared error is less than 0.30 of the standard deviation of Y), while a score of less than 0.50 is okay, NLSE scores greater than 0.50 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 0.50. Finally, a TCE score of less than 0.20 is excellent. A TCE score of less than 0.30 is good; while, a TCE of 0.30 or greater is poor.

2 Previous Results on Nine Base Problems

The previously published results (Korns 2008) of training on the nine base training models on 10,000 rows and five columns with no random noise and only 20 generations allowed, are shown below [2].

In general, training time is very reasonable given the difficulty of some of the problems and the limited number of training generations allowed. Average percent error performance varies from excellent to 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 overfitting.

Surprisingly, long and short classification is fairly robust in most cases with the exception of the *ratio*, and *mixed* test cases. The salient observation is the relative ease of classification compared to regression even in problems with this much noise. In some of the test cases, testing NLSE is either close

[2] The nine base test cases are described in detail in (Korns, 2007).

to or exceeds the standard deviation of Y (not very good); however, in many of the test cases classification is below 0.20. (very good).

Table 1. Result For 10K rows by 5 columns no Random Noise

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

The previously published results (Korns 2008) 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 below.

Table 2. Result for 10K rows by 20 columns with 40% Random Noise

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

Clearly the symbolic regression system performs most poorly on the test cases *hidden*, *mixed* and *ratio* with conditional target expressions. There is evidence of over-fitting shown by the extreme differences between training error and testing error. Plus, the testing TCE is very poor in both ratio test cases. Taken together, these scores portray a symbolic regression system which is not ready to handle industrial strength problems containing conditional target expressions.

Enhancements which will improve our regression scores on the two conditional base test cases, without also greatly reducing the efficiency of the

symbolic regression system on the other test cases, is the subject of the remainder of this chapter.

3 Pessimal Vertical Slicing

In (Korns 2006) we describe 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], ...* .

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. We train on a single vertical slice, but score 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) [3].

Our normal vertical slicing sampling size is one out of every hundred training cases. Of course with difficult conditional target expressions, while this sampling size reduces training time, it also reduces accuracy. So we face a conundrum. Increasing the sampling rate increases accuracy; but, also greatly increases the time to manage easier test cases.

One solution is to leave our normal sampling size as it is (one out of every hundred training cases) until an emergency is declared. If we get to the end of the first training epoch (currently sent to ten generations) and the champion NLSE is .50 or higher, then we declare an emergency. The emergency sampling rate is one out of every four training cases. Increasing the emergency sampling rate increases accuracy for the difficult problems; and, has no impact on the easier test cases.

For complete training coverage, we intersperse randomly selected vertical training slices with pessimally selected vertical training slices with respect to the current best-of-breed champion. The pessimal verstical slice, with respect to the current best-of-breed champion, is the verticle slice on which the current champion has the worst fitness scores. Regardless of which vertical slice is selected, as the training subset, we still score across every fitness example in X. Choosing randomly selected training subsets forces complete training coverage while still maintaining the out-of-sample scoring so important for avoiding overfitting. Choosing the pessimal training subset forces the system to learning those test cases which have been difficult for the current champion while still maintaining the out-of-sample scoring.

[3] The implementation of Vertical Slicing is described in detail in (Korns, 2006).

4 Splicing *Background*

Our system uses a technique known as *aged-layered population structure* (ALPS), devised to minimize premature population convergence [4]. During the course of an ALPS training run we keep track of an elitist pool of all-time champions. As an enhancement, to support splicing, we simultaneously keep track of a second elitist pool of all-time champions *which are uncorrelated to the champions in the elitist pool*. Unfortunately managing the uncorrelated champion pool requires that we perform a standard statistical correlation test for every new champion above a certain NLSE. This process is not free, and therefore it will degrade the performance on the easier test cases to some extent. Nevertheless maintaining an uncorrelated champion pool will allow us to splice uncorrelated champions together using conditional abstract expressions.

Before we can reasonably describe the splicing process in detail, we must provide a brief background on abstract expression grammars as they are used in this symbolic regression system.

In the literature, 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, 2003) and (Poli, 2008). Using a hybrid combination of tree-based GP and formal grammars, where the head of each s-expression is a grammar rule, the standard genetic programming population operators of mutation and crossover can be used without alteration. We use standard mutation and crossover operations (Koza, 1992) and support both simple regression and multiple regression.

4.1 A Concrete Expression Grammar

A simple concrete expression grammar suitable for use in most symbolic regression systems would be a C-like grammar with the following basic elements.

- **Real Numbers**: 3.45, -.0982, and 100.389
- **Row Features**: x1, x2, and x5.
- **Operators**: +, *, /, %, <, <=, ==, ! =, >=, >
- **Functions**: sqrt(), log(), cube(), sin(), tan(), max(), etc.
- **Conditional**: (expr1 < expr2) ? expr3 : expr4

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

[4] The implementation of age-layered population structure is described in detail in (Hornby, 2006).

Our symbolic regression system creates its regression champion using mutation, and cross over; but, the final regression champion will be a compilation of a basic concrete expression such as:

- (*E1*): f = (log(x3)/sin(x2*45.3))>x4 ? tan(x6) : cos(x3)

Computing an NLSE score for f requires only a single pass over every row of X and results in an attribute being added to f by executing the score method compiled into f as follows.

- f.NLSE = f.score(X,Y).

4.2 Abstract Constants

Suppose that we are satisfied with the form of the expression in (E1); but, we are not sure that the real constant 45.3 is optimal. The standard genetic programming algorithm does not provide a mechanism for optimizing the real constant, 45.3, other than running the symbolic regression system for more iterations; and, then we are not guaranteed of receiving an improved answer in the same form as in (E1). We can enhance our symbolic regression system with the ability to optimize individual real constants by adding abstract constant rules to our built-in algebraic expression grammar.

- *Abstract Constants*: c1, c2, and c10

Abstract constants represent placeholders for real numbers which are to be optimized by the symbolic regression system. To further optimize f we would alter the expression in (E1) as follows.

- (*E2*): f = (log(x3)/sin(x2*__c1__))>x4 ? tan(x6) : cos(x3)

The compiler adds a new real number vector, C, attribute to f such that f.C has as many elements as there are abstract constants in (E2). Optimizing this version of f requires that the built-in score method compiled into f be changed from a single pass to a multiple pass algorithm in which the real number values in the abstract constant vector, f.C, are iterated until the expression in (E2) produces an optimized NLSE. This new score method has the side effect that executing f.score(X,Y) also alters the abstract constant vector, f.C, to optimal real number choices. Clearly the particle swarm (Eberhardt 2001) and differential evolution algorithms provide excellent candidate algorithms for optimizing f.C and they can easily be compiled into f.score by common compilation techniques currently in the main stream. Summarizing, we have a new grammar term, c1, which is a reference to the 1st element of the real

number vector, f.C (in C language syntax c1 == f.C[1]). The f.C vector is optimized by scoring f, then altering the values in f.C, then repeating the process iteratively until an optimum NLSE is achieved. Two important features of abstract expression grammars are worth mention here. The overall genetic programming algorithms within the nonlinear regression system do not have to be altered because the swarm and differential learning enhancements are hidden inside the score method by the abstract expression compiler when appropriate. Furthermore, as Riccardo Poli (Poli 2008) has pointed out, a new population operator can be defined which converts abstract expressions into their concrete counterparts. For instance, if the regression champion agent in (E2) is optimized with:

- f.C == < 45.396 >

Then the optimized regression champion agent in (E2) has a concrete conversion counterpart as follows:

- f = (log(x3)/sin(x2*__45.396__))>x4 ? tan(x6) : cos(x3)

Since abstract expressions are not grammatically excessively different than concrete expressions, the genetic programming logic in the symbolic regression system will be able to apply the same type of operations (crossover, mutation, etc.) on either type of expression. At different stages in the evolutionary process population operators can be introduced which convert abstract expressions into their optimized concrete counterparts, or even new mutation operators which convert concrete expressions into abstract expressions.

4.3 Abstract Features

Suppose that we are satisfied with the form of the expression in (E1); but, we are not sure that the features, x2, x3, and x6, are optimal choices. The standard genetic programming algorithm does not provide a mechanism for optimizing these features other than running the symbolic regression system for more iterations; and, then we are not guaranteed of receiving an improved answer in the same form as in (E1). We can enhance our symbolic regression system with the ability to optimize individual features by adding abstract feature rules to our built-in algebraic expression grammar.

- *Abstract Features*: v1, v2, and v10

Abstract features represent placeholders for features which are to be optimized by the nonlinear regression system. To further optimize f we would alter the expression in (E1) as follows.

- (*E3*): f = (log(**v1**)/sin(**v2***45.3))>**v3** ? tan(**v4**) : cos(**v1**)

The compiler adds a new integer vector, V, attribute to f such that f.V has as many elements as there are abstract features in (E3). Each integer element in the f.V vector is constrained between 1 and M, and represents a choice of feature (in x). Optimizing this version of f requires that the built-in score method compiled into f be changed from a single pass to a multiple pass algorithm in which the integer values in the abstract feature vector, f.V, are iterated until the expression in (E3) produces an optimized NLSE. This new score method has the side effect that executing f.score(X,Y) also alters the abstract feature vector, f.V, to integer choices selecting optimal features (in x). Clearly the genetic algorithm (Man 1999), discrete particle swarm (Eberhardt 2001), and discrete differential evolution (Price 2005) algorithms provide excellent candidate algorithms for optimizing f.V and they can easily be compiled into f.score by common compilation techniques currently in the main stream. Summarizing, we have a new grammar term, v1, which is an indirect feature reference thru to the 1st element of the integer vector, f.V (in C language syntax v1 == x[f.V[1]]). The f.V vector is optimized by scoring f, then altering the values in f.V, then repeating the process iteratively until an optimum NLSE is achieved. For instance, the regression champion agent in (E3) is optimized with:

- f.V == < 2, 4, 1, 6 >

Then the optimized regression champion agent in (E3) has a concrete conversion counterpart as follows:

- f = (log(**x2**)/sin(**x4***45.396))>**x1** ? tan(**x6**) : cos(**x2**)

4.4 Abstract Functions

Similarly, we can enhance our nonlinear regression system with the ability to optimize individual features by adding abstract functions rules to our built-in algebraic expression grammar.

- *Abstract Functions*: f1, f2, and f10

Abstract functions represent placeholders for built-in functions which are to be optimized by the nonlinear regression system. To further optimize f we would alter the expression in (E2) as follows.

- (*E4*): f = (**f1**(x3)/**f2**(x2*45.3))>x4 ? **f3**(x6) : **f4**(x3)

The compiler adds a new integer vector, F, attribute to f such that f.F has as many elements as there are abstract features in (E4). Each integer element in the f.F vector is constrained between 1 and (number of built-in functions available in the expression grammar), and represents a choice of built-in function. Optimizing this version of f requires that the built-in score method compiled into f be changed from a single pass to a multiple pass algorithm in which the integer values in the abstract function vector, f.F, are iterated until the expression in (E4) produces an optimized NLSE. This new score method has the side effect that executing f.score(X,Y) also alters the abstract function vector, f.F, to integer choices selecting optimal built-in functions. Clearly the genetic algorithm (Man 1999), discrete particle swarm (Eberhardt 2001), and discrete differential evolution (Price 2005) algorithms provide excellent candidate algorithms for optimizing f.F and they can easily be compiled into f.score by common compilation techniques currently in the main stream. Summarizing, we have a new grammar term, f1, which is an indirect function reference thru to the 1st element of the integer vector, f.F (in C language syntax f1 == funtionList[f.F[1]]). The f.F vector is optimized by scoring f, then altering the values in f.F, then repeating the process iteratively until an optimum NLSE is achieved. For instance, if the valid function list in the expression grammar is

- f.functionList = < log, sin, cos, tan, max, min, avg, cube, sqrt >

And the regression champion agent in (E4) is optimized with:

- f.F = < 1, 8, 2, 4 >

Then the optimized regression champion agent in (E4) has a concrete conversion counterpart as follows:

- f = (**log**(x3)/**cube**(x2*45.3))>x4 ? **sin**(x6) : **tan**(x3)

The built-in function argument arity issue is easily resolved by having each built-in function ignore any excess arguments and substitute defaults for any missing arguments. Furthermore random noise functions, such as in (Schmidt 2007), can easily be added to the list of available built-in functions in the expression grammar.

5 Splicing *Details*

Assume that we have reached the end of the first training epoch and the best of breed NLSE is so high that we declare an emergency. What action do we take to address this declared emergency?

Our approach is to introduce an end-of-epoch splicing algorithm to fit together uncorrelated champions using abstract conditional expressions. Selecting the fittest champion, *G*, from the elitist all-time champion pool and selecting the fittest champion, *H*, from the uncorrelated champion pool, we create several new candidate champions by splicing together the well formed formulas *G.wff* and *H.wff* via various predefined abstract conditional expressions as follows.

- *B1*: y = (**v1**>**c1**) ? G.wff : H.wff
- *B2*: y = (**c1**>**v1**) ? G.wff : H.wff
- *B3*: y = (**v1**>**v2**) ? G.wff : H.wff
- *B4*: y = (**f1(v1,v2)**>**c1**) ? G.wff : H.wff
- *B5*: y = (**f1(v1,v2)**<**c1**) ? G.wff : H.wff
- *B6*: y = (**f1(v1,v2)**>**v3**) ? G.wff : H.wff
- *B7*: y = (**f1(v1,v2)**<**v3**) ? G.wff : H.wff
- *B8*: y = (**f1(v1,v2)**<**f2(v3,v4)**) ? G.wff : H.wff

Finally, at the end of each epoch, the splicing algorithm introduces each of the above abstract expressions into the evolutionary pool trying to improve the NLSE over that of the current best of breed champion. Each of the above splicings is optimized and their ***optimized concrete conversions*** are stored in the appropriate population.

For example, suppose our target expression is shown in (E5) below, our best of breed champion is such that *G.wff = tan(x6)*, and our best uncorrelated champion is such that *H.wff = cos(x3)*, then we have final training situation as follows.

- (*E5*): f = (log(x3)>x4) ? tan(x6) : cos(x3)
- G.wff = tan(x6)
- H.wff = cos(x3)

Clearly, given the above situation, the splicing algorithm would attempt to train the following several spliced abstract conditional champions.

- *B1*: y = (**v1**>**c1**) ? tan(x6) : cos(x3)
- *B2*: y = (**c1**>**v1**) ? tan(x6) : cos(x3)
- *B3*: y = (**v1**>**v2**) ? tan(x6) : cos(x3)
- *B4*: y = (**f1(v1,v2)**>**c1**) ? tan(x6) : cos(x3)
- *B5*: y = (**f1(v1,v2)**<**c1**) ? tan(x6) : cos(x3)
- *B6*: y = (**f1(v1,v2)**>**v3**) ? tan(x6) : cos(x3)
- *B7*: y = (**f1(v1,v2)**<**v3**) ? tan(x6) : cos(x3)
- *B8*: y = (**f1(v1,v2)**<**f2(v3,v4)**) ? tan(x6) : cos(x3)

If the splicing algorithm is behaving optimally, we would expect the final concrete conversion of the fully trained (B6) to be as follows.

- *B6 (concrete)*: y = (log(x3)>x4) ? tan(x6) : cos(x3)

  This is of course the correct answer.

6 Abstract Mutation and Crossover

In standard mutation and crossover, random segments of program code are selected for mutation and swapping. In abstract mutation and crossover, these randomly selected segments are *abstracted*. In both abstract mutation and abstract crossover, a set of simple rules define the process of abstracting an expression segment, as follows:

- *Real Numbers*: **3.45**, **-.0982** *are converted to* **c1**, **c1**
- *Row Features*: **x1**, **x4** *are converted to* **v1**, **v2**
- *Operators*: **+**, ***** *are converted to* **f1()**, **f2()**
- *Functions*: **sqrt()**, **log()** *are converted to* **f1()**, **f1()**

Using these simple rules, the abstract mutation population operator works as in the following example:

- f = (log(x3)/**sin(x2*45.3)**)>x4 ? tan(x6) : cos(x3)
- The selected segment **sin(x2*45.3)** is abstracted into **f1(f2(v1,c1))**
- where f1 = sin, f2 = *, v1 = x2, and c1 = 45.3
- which is then inserted below
- f = (log(x3)/**f1(f2(v1,c1))**)>x4 ? tan(x6) : cos(x3)

Similarly, the abstract crossover population operator selects two random segments from two expressions such as:

- *dad* = (**log(x3)**/sin(x2*45.3))
- *mom* = (tan(x3)/cube(**x2*45.3**))
- The selected segments are first swapped and then "abstracted" as follows:
- *dad* = (**x2*45.3**)/sin(x2*45.3)) *abstracted as* = (**f1(v1,c1)**)/sin(x2*45.3))
- *mom* = (tan(x3)/cube(**log(x3)**)) *abstracted as* = (tan(x3)/cube(**f1(v1)**))

After abstract mutation or crossover, the new abstract expressions are optimized by the regression system. Only their ***optimized concrete conversions*** are saved in the proper evolutionary populations. In the enhanced system, 5% of all mutations are abstract mutations and 5% of all crossovers are abstract crossovers.

From first principles, abstract mutation and crossover are compelling because it is less likely that **45.3** will be optimal in a new mutation or location; and, more likely that **c1** will find a local optimum in the new mutation or location. Similar arguments are put forward for **v1**, and **f1**.

7 Enhanced Results on Nine Base Problems

The enhanced results of training on the nine base training models on 10,000 rows and five columns with no random noise and only 20 generations allowed, are shown below in order of difficulty.

Table 3. Result For 10K rows by 5 columns no Random Noise

Test	*Minutes*	*Train-NLSE*	*Train-TCE*	*Test-NLSE*	*Test-TCE*
linear	1	0.00	0.00	0.00	0.00
cubic	1	0.00	0.00	0.00	0.00
cross	145	0.00	0.00	0.00	0.00
elipse	1	0.00	0.00	0.00	0.00
hidden	3	0.00	0.00	0.00	0.00
cyclic	1	0.02	0.00	0.00	0.00
hyper	65	0.17	0.00	0.17	0.00
mixed	233	0.94	0.32	0.95	0.32
ratio	229	0.94	0.33	0.94	0.32

The enhanced 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 below in order as shown above.

Table 4. Result for 10K rows by 20 columns with 40% Random Noise

Test	*Minutes*	*Train-NLSE*	*Train-TCE*	*Test-NLSE*	*Test-TCE*
linear	82	0.11	0.00	0.11	0.00
cubic	59	0.11	0.00	0.11	0.00
cross	127	0.87	0.25	0.93	0.32
elipse	162	0.42	0.04	0.43	0.04
hidden	210	0.11	0.02	0.11	0.02
cyclic	233	0.39	0.11	0.35	0.12
hyper	163	0.48	0.06	0.50	0.07
mixed	206	0.90	0.27	0.94	0.32
ratio	224	0.90	0.26	0.95	0.33

Clearly, in time-constrained training (only 20 generations), the enhanced symbolic regression system is an improvement over the previously published results. While the enhanced system performs poorly on the two test cases *mixed* and *ratio* with conditional target expressions, the obvious over fitting, determined by the extreme differences between training error and testing error in the previously published results, has vanished. In addition, the testing TCE

scores indicate that we can perform some useful classification even in the difficult conditional problems with noise added.

As an acid test of the value of the system enhancements, it would be helpful to know how well the enhanced symbolic regression system performs on the two test cases *mixed* and *ratio* with conditional target expressions when the training is not time-constrained. For instance, do added training generations improve the training NLSE and TCE scores? Does added training time also improve the testing NLSE and TCE scores, or does the harmful over fitting reappear once again?

The results of training on the two test cases *mixed* and *ratio*, with conditional target expressions, on 10,000 rows and five columns with 00% random noise and allocating additional training generations, are shown below.

Table 5. Result for 10K rows by 5 columns with 00% Random Noise

Test	*Minutes*	*Train-NLSE*	*Train-TCE*	*Test-NLSE*	*Test-TCE*	*Gens*
mixed	233	0.94	0.32	0.95	0.32	20
mixed	9866	0.87	0.24	0.88	0.25	200
mixed	15148	0.85	0.23	0.87	0.26	400
ratio	229	0.94	0.33	0.94	0.32	20
ratio	10324	0.87	0.23	0.87	0.25	200
ratio	14406	0.82	0.19	0.82	0.20	400

Clearly removing the time constraint on training, by adding additional training generations, steadily improves the results. There is obvious incremental improvement in the training NLSE and TCE scores for both problems as the number of training generations increases. Furthermore, the testing NLSE and TCE scores for both problems also improve steadily as the number of training generations increases. There is no evidence of a limit on training improvement nor any evidence of over fitting at least up to 400 training generations.

Taken together, these results portray a symbolic regression system which is ready to handle some industrial strength problems containing conditional target expressions.

7.1 Summary

Genetic Programming, from a corporate perspective, is ready for industrial use on *some* large scale, symbolic regression-classification problems. Adapting the latest research results, has created a symbolic regression tool whose efficiency is improving especially on the more difficult 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 nonlinear regression system in the financial

domain. But as new techniques are added and current ones improved, we believe that the system has evolved to be a domain-independent tool that can provide superior regression and classification results for industrial scale nonlinear regression problems.

Clearly we need to experiment with even more techniques which will improve our performance on the conditional test cases. Primary areas for future research should include: experimenting with statistical and other types of analysis to help build conditional WFFs for difficult conditional problems with large amounts of noise; and experimenting with techniques to remove training time constraints while increasing training generations, for instance parallelizing the system on a cloud environment.

References

1. Michael Caplan, Ying Becker (2005). Lessons Learned Using Genetic Programming in a Stock Picking Context, in *Genetic Programming Theory and Practice II*. Springer, New York.
2. Shu-Heng Chen (2002), editor. Genetic Algorithms and Genetic Programming in Computational Finance. Kluwer Academic Publishers, Dordrecht Netherlands.
3. Russell Eberhart, Yuhui Shi, James Kennedy (2001). Swarm Intelligence. Morgan Kaufman, New York.
4. Gregory S Hornby (2006). Age-Layered Population Structure For reducing the Problem of Premature Convergence, in *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM Press, New York.
5. Michael Korns (2006). Large-Scale, Time-Constrained, Symbolic Regression, in *Genetic Programming Theory and Practice IV*. Springer, New York.
6. Michael Korns (2007). Large-Scale, Time-Constrained, Symbolic Regression-classification, in *Genetic Programming Theory and Practice V*. Springer, New York.
7. Michael Korns, Loryfel Nunez (2008). Profiling Symbolic Regression-classification, in *Genetic Programming Theory and Practice V1*. Springer, New York.
8. John R Koza (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge Massachusetts.
9. John R Koza (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. The MIT Press, Cambridge Massachusetts.
10. John R Koza, Forrest H Bennett III, David Andre, Martin A Keane (1999). Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann Publishers, San Francisco California.
11. Riccardo Poli, William Langdon, Nicholas McPhee (2008). A Field Guide to Genetic Programming. LuLu Enterprises.
12. Hammad Majeed and Conor Ryan (2006). Using context-aware crossover to improve the performance of GP, in *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM Press, New York.
13. Michael O'Neill, Conor Ryan (2003). Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers, Dordrecht Netherlands.
14. Kenneth Price, Rainer Storn, Jouni Lampinen (2005). Differential Evolution: A Practical Approach to Global Optimization. Springer, New York.
15. Michael Schmidt and Hod Lipson (2007). Learning Noise, in *GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM Press, New York.
16. Kim-Fung Man, Kit-Sang Tang, Sam Kwong (1999). Genetic Algorithms. Springer, New York.
17. Kenneth Price, Rainer Storn, Jouni Lampinen (2005). Differential Evolution: A Practical Approach to Global Optimization. Springer, New York.