Symbolic Regression using Abstract Expression Grammars

Michael F. Korns
Freeman Investment Management
1 Plum Hollow
Henderson, Nevada 89052
1 (702) 837 3498
mkorns@korns.com

ABSTRACT

Abstract Expression Grammars have the potential to integrate Genetic Algorithms, Genetic Programming, Swarm Intelligence, and Differential Evolution into a seamlessly unified array of tools for use in symbolic regression. The features of abstract expression grammars are explored, examples of implementations are provided, and the beneficial effects of abstract expression grammars are tested with several published nonlinear regression problems.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming - Program Synthesis.

General Terms

terms: Algorithms

Keywords

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

1. INTRODUCTION

Large scale general nonlinear regression is currently practiced largely using genetic programming under the name Symbolic Regression [4]. However, as a nonlinear regression algorithm, genetic programming has a number of difficulties including expression bloat and lack of fine grain control over the solution produced. On the other hand, swarm intelligence [1] and differential evolution [7] provide excellent fine grain control but are not easily linked to general algebraic expression grammars.

Abstract expression grammars can be used to combine these disparate machine learning techniques into a seamlessly unified algorithm for general nonlinear regression. In fact abstract expression grammars are currently being used inside a symbolic regression system for large scale symbolic regression [2] [3].

2. Linear Regression Systems

A general linear regression system accepts an input matrix, X, of N rows and M columns and a dependent variable vector, Y, of length N. The dependent vector Y is related to X thru the (hopefully but not necessarily linear) function, f, as follows: Y[n] = f(X[n]). The output of a linear regression system will be a coefficient vector, C, of length M, such that the inner product of C with each row of X produces an estimate vector, EY, which minimizes the least square error between EY and Y. General linear regression systems can easily be constructed using Gaussian methods.

For our purposes in the remainder of this paper, we will normalize all least squared error measurements by dividing by the standard deviation of Y. We will call this the Normalized Least Squared Error (NLSE).

3. Nonlinear Regression Systems

A general nonlinear regression system accepts an input matrix, X, of N rows and M columns and a dependent variable vector, Y, of length N. The dependent vector Y is related to X thru the (quite possibly nonlinear) function, f, as follows: Y[n] = f(X[n]). The output of a nonlinear regression system will be a program object (which we will hereinafter call an Estimator Agent), f, such that invoking f on each row of X produces an estimate vector, EY, which minimizes the normalized least square error between EY and Y. General nonlinear regression systems can be constructed using genetic programming methods[2] [3] [4] [6].

The capabilities of the agent, f, will be constrained by some sort of algebraic expression grammar built into the nonlinear regression system. More to the point, the computer codes executed upon invoking f will be a compilation of some statement in said grammar.

4. A Concrete Expression Grammar

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

4.1 Real Numbers: 3.45, -.0982, and 100.389

4.2 Row Features: x1, x2, and x5.

4.3 Operators: +, *, /, %, <, <=, ==, !=, >=, >

4.4 Functions: sqrt(), log(), cube(), sin(), tan(), max(), etc.

4.5 Conditional: (expr1 < expr2) ? expr3 : expr4

A nonlinear regression system might create its final estimator agent using mutation, cross over, or any number of techniques; but, the final estimator agent might easily be a compilation of a basic concrete expression such as:

4.6 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.

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

5. Abstract Constants

Suppose that we are satisfied with the form of the expression in (4.6); 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 (4.6).

We can enhance our nonlinear regression system with the ability to optimize individual real constants by adding abstract constant rules to our built-in algebraic expression grammar.

5.1 Abstract Constants: c1, c2, and c10

Abstract constants represent placeholders for real numbers which are to be optimized by the nonlinear regression system. To further optimize f we would alter the expression in (4.6) as follows.

5.2 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 (5.2). 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 (5.2) 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 [1] and differential evolution [7] 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 a new population operator can be defined which converts abstract expressions into their concrete counterparts. For instance, the estimator agent in (5.2) is optimized with:

5.3 f.C == < 45.396 >

Then the optimized estimator agent in (5.2) has a concrete conversion counterpart as follows:

5.4 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 nonlinear regression system will operate 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.

6. Abstract Features

Suppose that we are satisfied with the form of the expression in (4.6); 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 (4.6).

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

6.1 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 (4.6) as follows.

6.2 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 (6.2). 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 (6.2) 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 [6], discrete particle swarm [1], and discrete differential evolution [7] 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 estimator agent in (6.2) is optimized with:

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

Then the optimized estimator agent in (6.2) has a concrete conversion counterpart as follows:

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

7. 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.

7.1 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 (4.6) as follows.

7.2 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 (7.2).

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 (7.2) 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 [6], discrete particle swarm [1], and discrete differential evolution [7] 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

7.3 < log, sin, cos, tan, max, min, avg, cube, sqrt >

And the estimator agent in (7.2) is optimized with:

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

Then the optimized estimator agent in (7.2) has a concrete conversion counterpart as follows:

7.5 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.

8. Putting it all Together

We can put all these expression grammar rules together by adding a final grammar rule as follows.

9.1 Phrase: p1, p2, and p3

The phrase placeholders represent unrestricted valid expressions, such as would be ordinarily returned by a genetic programming algorithm. Therefore, if we request the nonlinear regression system to optimize, **f = p1**, we are requesting results similar to those returned by genetic programming algorithms. The following optimization requests provide a few examples of the range of control we can exhibit over the nonlinear regression system via the use of abstract expression grammars.

9.2 f = p1

9.3 f = (log(x3)/sin(x2*c1))>x4 ? tan(x6) : cos(x3)

9.4 f = (f1(v1)/f2(v2*c1))>v3 ? f3(v4) : f4(v5)

9.5 f = (p1 < p2 ? log(x2) : sin(x3)

Expression (9.2) requests a GP run. Expression (9.3) requests that a specific abstract constant, c1, be optimized. Expression (9.4) requests that a number of abstract constants, features, and functions be optimized but in exactly the specified form. Expression (9.5) requests that two general GP expressions be evolved which optimally predict whether log(x2) or sin(x3) is the correct regression model.

9. Testing these Concepts

In testing nonlinear regression using abstract expression grammars we borrow the "hyper" test case published in [2].

10.1 hyper: y = 1.57 + (1.57*tanh(cube(x1)))
- (39.34*tanh(cube(x2)))
+ (2.13*tanh(cube(x3)))
+ (46.59*tanh(cube(x4)))
+ (11.54*tanh(cube(x5)))

Using the nonlinear regression system described in [2] [3], we construct an X matrix 10,000 by 5, filled with random numbers between -50 and +50, and run the "hyper" training model (10.1) on each row of X to create the Y dependent vector. When we request (*quickly training for only 25 generations*) the nonlinear regression system to optimize, **f = p1**, the following results are returned.

Table 10.2: Regression Results after 20 Generations

Training Model	y = 1.57 + (1.57*tanh(cube(x1))) - (39.34*tanh(cube(x2))) + (2.13*tanh(cube(x3))) + (46.59*tanh(cube(x4))) + (11.54*tanh(cube(x5)))
Hint	f = p1
Resulting Estimator	f = avg(-93.27*tanh(x4),-77.83*tanh(x2))
NLSE	.19

Clearly the genetic programming algorithm has guessed some of the features of the training model; but, many more generations of training would be required for the GP algorithm to figure the whole training model out. Even so the NLSE, of .19, is not too bad.

However, when we request (*quickly training for only 25 generations*) the nonlinear regression system to optimize,

10.3 f = sum(c1, c2*tanh(cube(v1)),
c3*tanh(cube(v2)),
c4*tanh(cube(v3)),
c5*tanh(cube(v4)),
c6*tanh(cube(v5)))

the following results are returned.

Table 20.4: Regression Results after 20 Generations

Training Model	y = 1.57 + (1.57*tanh(cube(x1))) - (39.34*tanh(cube(x2))) + (2.13*tanh(cube(x3))) + (46.59*tanh(cube(x4))) + (11.54*tanh(cube(x5)))
Hint	f = sum(c1, c2*tanh(cube(v1)), c3*tanh(cube(v2)), c4*tanh(cube(v3)),

	c5*tanh(cube(v4)), c6*tanh(cube(v5))
Resulting Estimator	f = sum(1.57, -39.34*tanh(cube(x2)), 2.13*tanh(cube(x3)), 1.57*tanh(cube(x1)), 11.54*tanh(cube(x5)), 46.59*tanh(cube(x4)))
NLSE	.00

Clearly the more specific hint has allowed the nonlinear regression system to guess the features and constants of the training model exactly. The NLSE, of .00, is perfect.

Even providing the nonlinear regression system with an incomplete hint such as,

10.5 f = sum(c1, c2*tanh(cube(v1)),

C3*tanh(cube(v2)))

provides quick results that are somewhat superior to no hint at all.

Table 30.6: Regression Results after 20 Generations

Training Model	y = 1.57 + (1.57*tanh(cube(x1))) - (39.34*tanh(cube(x2))) + (2.13*tanh(cube(x3))) + (46.59*tanh(cube(x4))) + (11.54*tanh(cube(x5)))
Hint	f = sum(c1, c2*tanh(cube(v1)), c3*tanh(cube(v2))
Resulting Estimator	f = sum(1.31, -39.01*tanh(cube(x2)), -46.65*tanh(cube(x4)))
NLSE	.18

Even an incomplete hint produces a quick NLSE slightly better than no hint at all, .18 versus .19.

10. The Future

While some of the concepts of abstract expression grammars have been implemented in [2] [3], at this time there appear to be no nonlinear regression systems in which all of these concepts are used. There is little benefit from keeping such valuable, yet disparate, algorithms as GA, GP, Particle Swarm, Differential Evolution, Support Vectors, Neural Nets, and even Gaussian Regression, separated, in isolation, and prevented from working together smoothly. Abstract Expression Grammars have the potential to integrate Genetic Algorithms, Genetic Programming, Swarm Intelligence, Differential Evolution, Support Vector Machines, Neural Nets, and even Gaussian Regression into a seamlessly unified array of tools for use in nonlinear regression systems.

Abstract expression grammar concepts will find their way into an increasing number of evolutionary learning systems and especially into nonlinear regression systems.

11. ACKNOWLEDGMENTS

Our thanks to Riccardo Poli for his thoughtful suggestions on new population operators possible with abstract expression grammars.

12. REFERENCES

[1] Eberhardt, Russel, Shi, Yuhui, and Kennedy, James. 2001 Swarm Intelligence. Morgan Kaufmann, New York, USA. http://www.amazon.com/Swarm-Intelligence-Morgan-Kaufmann-Artificial/dp/1558605959/ref=pd_bbs_sr_1?ie=UTF8&s=books&qid=1228938121&sr=8-1

[2] Korns, Michael F. 2007. Large-Scale, Time-Constrained Symbolic Regression-Classification. In Riolo, Rick, L, Soule, Terrance, and Wortzel, Bill, editors, Genetic Programming Theory and Practive V, pages 53-68, New York, New York, USA. Springer. http://www.springer.com/computer/artificial/book/978-0-387-76307-1

[3] Korns, Michael F., and Nunez, Loryfel, 2008. Profiling Symbolic Regression-Classification. In Riolo, Rick, L, Soule, Terrance, and Wortzel, Bill, editors, Genetic Programming Theory and Practive VI, pages 215-228, New York, New York, USA. Springer. http://www.springer.com/computer/artificial/book/978-0-387-87622-1

[4] Koza, John, R. 1992 Genetic Programming: On Programming Computers by means of natural Selection. MIT Press, Cambridge, USA. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=5888

[5] Man, Kim-Fung, Tang, Kit-Sang, and Kwong, Sam. 1999. Genetic Algorithms. Springer, New York, USA. http://www.springer.com/engineering/robotics/book/978-1-85233-072-9

[6] O`Neil, Michael, and Ryan, Conor. 2003. Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers, Dortrecht, Netherlands. http://www.alibris.com/booksearch?binding=&mtype=&keyword=Grammatical+Evolution&hs.x=6&hs.y=15

[7] Price, Kenneth, Storn, Rainer, and Lampinen, Jouni 2005. Differential Evolution: A Practical Approach to Global Optimization. Springer, New York, USA. http://www.springer.com/computer/foundations/book/978-3-540-20950-8