Large-Scale, Time-Constrained Symbolic Regression-Classification

Michael F. Korns

Investment Science Corporation, 1 Plum Hollow, Henderson, Nevada 89052 USA
mkorns@korns.com

Summary. This chapter demonstrates a novel method combining particle swarm, differential evolution, and genetic programming to build a symbolic regression tool for large-scale, time-constrained regression-classification problems. In a previous paper we experimented with large scale symbolic regression. Here we describe in detail the enhancements and techniques employed to support large-scale, time-constrained regression and classification. In order to achieve the level of performance reported here, of necessity, we borrowed a number of ideas from disparate schools of genetic programming and recombined them in ways not normally seen in the published literature. We discuss in some detail the construction of the fitness function, the use of abstract grammars to combine genetic programming with differential evolution and particle swarm agents, and the use of context-aware crossover. [1]

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

[1] This work was completed with the generous help of Conor Ryan who educated us on Context-Aware Crossover and Timothy May who pioneered the Strategy Central Project at Investment Science Corp.

1 Introduction

This is the story of the problems encountered by Investment Science Corporation in using genetic programming techniques to construct a large-scale, time-constrained symbolic regression tool which could also perform classification.

Without delving in-depth into our financial methods, which is strictly forbidden by our corporate policy, we introduce our financial motivations very briefly and from these motivations quickly construct the requirements of a generic symbolic regression tool which could be used for classifying common stocks into long and short candidates for financial applications.

In our previous paper (Korns01), 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 ideas 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. Of special interest is combining fitness functions to support both symbolic regression and classification of common stocks into long and short candidates, and combining ideas from particle swarm and differential evolution to provide more fine grained control during the Genetic Programming process.

These disparate schools of genetic programming and evolutionary programming were combined into the following unusual hybrids:

- *Combined:* "Hybrid combination of particle swarm agents and GP"
- *Combined:* "Hybrid combination of abstract grammar and tree-based GP"
- *Combined:* "Hybrid combination of multiple island populations and boosting with GP"
- *Combined:* "Hybrid fitness measure supporting symbolic regression and classification of long and short candidates"

This narrative describes an applied research project at Investment Science Corp. involving man years of engineering effort and hundreds of experiments. Each of the hybrid solutions required to overcome the challenges are described in detail. This research project produced a generic symbolic regression-classification tool capable of processing one million row by twenty column data mining tables in less than fifty hours on a single workstation computer (specifically an Intel Core 2 Duo Processor T7200 (2.00GHz/667MHz/4MB), running our Analytic Information Server software generating Lisp agents that maximize the on-board Intel registers and on-chip vector processing capabilities).

1.1 Financial Motivation

Our experimental market-neutral trading system selects, from a universe of the 800 most-liquid exchange-traded common stocks, eighty *(the worst 10%)* securities to sell short, and another eighty *(the best 10%)* to buy long. A successful market-neutral trading system makes a profit greater than that obtained from a market indexing strategy regardless of market conditions.

Consider a quantitative (quant) trading system for the top 800 exchange-traded common stocks with the largest dollar-volume traded in the prior week (YCK04a and CB04a). These securities are so active that we will be able to move millions of dollars in and out of these investments without appreciably perturbing their prices. We will retrain the system weekly using a sliding training window of five years or 1,250 training days of historical data (YCK04a). This allows relatively frequent system retraining (weekly) while providing a relatively long retraining period of fifty hours (the weekend).

Our first challenge is selecting a basket of 20 sample column data points from the over 500 available data points such as Open, High, Low, Close, Volume, EPS, Analyst Rating, etc. We solve this issue by implementing multiple independent trading systems. For instance, one might have a value trading system with one set of 20 training points, a growth trading system with another set of training points, and a chartist trading system with yet another set of training points. If one has a farm of 100 workstations, each workstation could retrain each of 100 independent trading systems once per week.

Our second challenge is to perform a symbolic regression every retraining period on a table of one million rows by twenty columns. If we retrain our market-neutral trading system weekly, using the previous five years of market data, a large volume of data must be fed into the system on every retraining period (1,250 historical daily samples for each of 800 common stocks is 1,000,000 rows of training data by 20 columns). In this paper, we construct a generic regression-classification tool which can perform a single 1,000,000 row by 20 column symbolic regression in less than 50 hours on a single workstation computer (so training can complete over the weekend). Clearly such a tool would prove useful not only in our own financial application but in many other large-scale, time-constrained applications.

1.2 Experimental Setup

Our experimental universe consists of a set of nine different fictitious markets driven by different model functions ranging from simple linear to more difficult multi-modal. Statical best-practice is employed to rigorously separate training and testing data sets so that all experiments are scored on testing data sets very different from the data sets they were trained on. We have crafted nine separate test cases (model formulas), from simple to complex. All of our test cases are trained on one million row by M column randomly generated training matrices (where M is either 1, 5, or 20). Then a separate randomly generated one million row by M column testing matrix is used for scoring. All of our nine test case formulas are shown below (generated with five columns).

Test Case Formulas

linear
```
y = 1.57 + (1.57*x0) - (39.34*x1) +
        (2.13*x2) + (46.59*x3) + (11.54*x4);
```

hidden
```
y = 1.57 + (2.13*sin(x2));
```

cubic
```
y = 1.57 + (1.57*x0*x0*x0) -
        (39.34*x1*x1*x1) + (2.13*x2*x2*x2) +
        (46.59*x3*x3*x3) + (11.54*x4*x4*x4);
```

elipse
```
y = 0.0 + (1.0*x0*x0) + (2.0*x1*x1) + (3.0*x2*x2) +
        (4.0*x3*x3) + (5.0*x4*x4);
```

hyper
```
y = 1.57 + (1.57*tanh(x0*x0*x0)) -
        (39.34*tanh(x1*x1*x1)) + (2.13*tanh(x2*x2*x2)) +
        (46.59*tanh(x3*x3*x3)) + (11.54*tanh(x4*x4*x4));
```

cyclic
```
y = 14.65 + (14.65*x0*sin(x0)) -
        (6.73*x1*cos(x0)) - (18.35*x2*tan(x0)) -
        (40.32*x3*sin(x0)) - (4.43*x4*cos(x0));
```

cross
```
y = -9.16 - (9.16*x0*x0*x0) -
        (19.56*x0*x1*x1) + (21.87*x0*x1*x2) -
        (17.48*x1*x2*x3) + (38.81*x2*x3*x4);
```

mixed
```
if ((mod(x0,4) == 0)
    {
    y = (1.57*log(.000001+abs(x0))) -
```

```
            (39.34*log(.000001+abs(x1))) +
            (2.13*log(.000001+abs(x2))) +
            (46.59*log(.000001+abs(x3))) +
            (11.54*log(.000001+abs(x4)));
    }
else
if ((mod(x0,4) == 1)
    {
    y = (1.57*x0*x0) - (39.34*x1*x1) +
            (2.13*x2*x2) + (46.59*x3*x3) +
            (11.54*x4*x4);
    }
else
if ((mod(x0,4) == 2)
    {
    y = (1.57*sin(x0)) - (39.34*sin(x1)) +
        (2.13*sin(x2)) + (46.59*sin(x3)) +
        (11.54*sin(x4));
    }
else
if ((mod(x0,4) == 3)
    {
    y = (1.57*x0) - (39.34*x1) +
        (2.13*x2) + (46.59*x3) +
        (11.54*x4);
    }
```

ratio

```
if ((mod(x0,4) == 0)
    {
    y = ((1.57*x0)/(39.34*x1)) +
            ((39.34*x1)/(2.13*x2)) +
            ((2.13*x2)/(46.59*x3)) +
            ((46.59*x3)/(11.54*x4));
    }
else
if ((mod(x0,4) == 1)
    {
    y = ((1.57*x0)%(39.34*x1)) +
            ((39.34*x1)%(2.13*x2)) +
            ((2.13*x2)%(46.59*x3)) +
            ((46.59*x3)%(11.54*x4));
    }
else
if ((mod(x0,4) == 3)
    y = 0.0 - (39.34* log(.000001+abs(x1))) +
            (2.13* log(.000001+abs(x2))) +
            (46.59*log(.000001+abs(x3))) +
            (11.54* log(.000001+abs(x4)));
    }
```

Our nine test cases vary from simple (linear) to complex (formulas with embedded if-then-else expressions). Finally, to add difficulty, we sometimes train and test our nine test cases with random noise added using the following formula.

```
;; Modify each y in Y or each ty in TY with random noise.
y = (y * .80) + (y * random(.40));
```

The addition of random noise makes each test case inexact and theoretically undiscoverable. Nevertheless, given our application, we need to test our symbolic regression tool against inexact data.

1.3 Fitness Measure

- *Combined:* "Hybrid fitness measure supporting symbolic regression and classification of long and short candidates"

Standard regression techniques often utilize least squares error as a fitness measure; however, we would also like to classify securities 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*).

Let the dependent variable, Y, be the future profits of a set of securities. 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%. Clearly the following will always be the case.

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

A situation where ((EBL - EBS) / (ABL - ABS)) > 0 indicates we are making money speculating on our short and long candidates. Obviously 1 is a perfect score *(we might as well have been prescient)* and -1 is a very imperfect score. Clearly, considering our financial application, we are interested in regression fitness measures which also classify as well as possible. In fact, even if the regression percent error is poor but the classification is good, we can still have an advantage, in the financial markets, with our symbolic regression tool.

We combined a normalized average percent error score with a classification score to produce an optimal fitness measure for our financial application as follows.

- avgDifY = average of abs(Y[n]-avgY) for all n in N
- avgErrY = average of abs(EY[n]-Y[n]) for all n in N

Our regression error and our classification scores as constructed as follows.

- errPct = avgErrY / avgDifY
- classify = (((EBL - EBS) / (ABL - ABS)) + 1) / 2

Finally, our fitness score is constructed as follows.

- fitness = (errPct + (.001 * classify)))

1.4 Abstract Grammars

- *Combined:* "Hybrid combination of abstract grammar and tree-based GP"

Recently, informal and formal grammars have been used in genetic programming (ONeil03) to enhance the representation and the efficiency of a number of applications including symbolic regression. In (Korns01), we discovered that alternative genome representations and evolutionary operators provided less added value than the use of multiple grammars themselves.

Therefore we settled on a hybrid combination of tree-based GP and formal grammars where the head of each sublist is a grammar rule agent with polymorphic methods for mutation, crossover, etc. Different grammar rules communicate with each other by message passing (a staple of object-oriented and agent-oriented software engineering). We use standard mutation and crossover operations (Koz92i) and support two regression grammar rules, one for simple regression and one for multiple regression as follows.

- *REG Grammar:* regress(EXP);
- *MVL Grammar:* mvlregress(EXP,EXP,...,EXP);

Our numeric s-expressions, the EXP grammar, are standard JavaScript-like numeric expressions with 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*, and with the following binary and unary operators + - / % * < <= == ! = >= > expt max min abs cos cosh cube exp log sin sinh sqroot square tan tanh. To these we add the ternary conditional expression operator *(...) ? ... : ... ;*

In this chapter we add an additional abstract numeric expression grammar, the AXP grammar, which is identical to the EXP grammar except that AXP expressions contain abstract real constants *c0* through *ck* (where k is the number of unique abstract real constants in the expression), and abstract variables *v0* through *vj* (where j is the number of unique abstract variables in the expression).

For instance, the following concrete expression *regress(3.4 * sin(x3/x5)*, when evaluated, has a fitness score based upon regressing 3.4 times the sine of column three divided by column five. However, the following abstract expression *regress(c0 * sin(v0/v1)*, which must be evaluated in a particle swarm agent, has a fitness score based upon the particle swarm's choice of actual real constant for c0 and the choices of actual columns v0 and v1.

It is our intent, by using an abstract expression grammar with imbedded particle swarm evaluation, to experiment with more fine-grained control during the genetic programming process.

1.5 Overview of Symbolic Regression Tool

In (Korns01) we constructed a large agent complex for high volume symbolic regression applications consisting of one million rows and from five to twenty columns. Due to the heavy resources required to evaluate a candidate well-formed-formula (WFF) across one millions rows, we cannot afford to evaluate the same candidate twice. Therefore, every WFF which we have ever evaluated is saved during the course of a single training cycle. All WFF candidates are saved in a collection sorted by their fitness scores. A user option setting restricts the survivor WFF population to the "F" most fit WFF candidates. User option settings support single or multiple island populations and other potentially usefully clustering of candidate WFFs. Parenthetically, all the tool's user option settings are available at run time; therefore, it might be possible for the tool to evolve itself, although we have not attempted anything of that nature.

Within the survivor population, mutation and crossover occur in the same fashion as with standard genetic programming. Each WFF survivor is visited once per each evolution. A user-option determines the probability of mutation and another determines the probability of crossover. If warranted, Crossover occurs between the visited individual and another randomly selected individual, from the survivor population. The tool supports multiple grammars in the same training cycle as described previously.

Standard genetic programming practice encourages the use of multiple independent training *runs*. Each run incorporates one initialization step and "G" generational steps during which evolutionary operators are applied. It is standard practice for the experimenter to perform multiple independent training runs of G generations each and then report the results of the fittest individual evolved across all runs (the champion individual).

Since our symbolic regression tool is to be used in a fully automated setting, their can be no human intervention to decide how many independent training runs to perform; therefore, the concept of automatic multiple independent training runs has been incorporated into the tool. A user-option determines the number of evolutions ”without fitness improvement” after which the system starts a new independent training run. After each independent training run, the best-of-breed champions from the previous run are saved and the training cycle restarts from scratch. Thus, a training cycle of ”G” generations may involve more or fewer separate independent training runs depending on the occurrence of long gaps without fitness improvement.

1.6 Vertical Slicing

In (Korns01) we made use of a new procedure *Vertical slicing*. First, the rows in the training matrix X are sorted in ascending order by the dependent values, Y. Then the rows in X are subdivided into S *vertical slices* by simply selecting every Sth 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], ...* . Each vertical slice makes no assumptions about the underlying probability distribution and each vertical slice contains evenly distributed training examples, in X, across the entire range of *ascending* dependent values, in Y.

Vertical slicing reduces training time by subdividing the training data into ”vertical slices” each of which is representative of the whole training data set over ascending values of Y. We then randomly select one of the vertical training data slices as our ”sample” training slice; furthermore, we modify each agent WFF and the memo cache to record the sample-fitness. There are now two different fitness scores for each WFF: sample-fitness and fitness. During evaluation, each WFF is first scored on the ”sample” training slice and its sample-fitness is recorded. Next the sample-fitness of the WFF is compared to the sample-fitness of the least-fit WFF in the survivor population. If the sample-fitness of the WFF is greater than or equal to the sample-fitness of the least-fit WFF in the survivor population, the WFF is then scored against the entire training data and its true fitness is recorded. This approach will produce false negatives but no false positives.

1.7 Adding the AXP Grammar to standard GP

- *Combined:* "Hybrid combination of particle swarm agents and GP"

In this section our goal is to describe the use of our abstract expression grammar, AXP, with standard genetic programming techniques. In a *concrete* grammar expression, such as *regress(EXP)*, obtaining the fitness score requires little more than evaluating the concrete expression once. Therefore the *regress(EXP)* expression can be compiled into a relatively simple agent which evaluates the expression, EXP, at each point and computes the fitness score.

In an *abstract* grammar expression, such as *regress(AXP)*, obtaining the fitness score requires much more than evaluating the abstract expression once. Therefore the *regress(AXP)* expression must be compiled into a complex agent which evaluates the expression, AXP, multiple times at each point and computes the fitness score for each iterative guess at the proper values of the abstract real constants *c0,...,ck* and the proper column choices for each of the abstract variable references *v0,...,vj*.

For instance, the following concrete expression *regress(3.4*sin(x3/x5)*, when evaluated on any point *x=(x0,...,xm)*, has a concrete fitness score. However, the following abstract expression *regress(c0*sin(v0/v1)*, cannot be evaluated on any point *x=(x0,...,xm)*, without choosing concrete values for the *c0, v0, and v1.*

A straightforward method for obtaining a fitness score for this abstract expression, first compiles *regress(c0*sin(v0/v1)* into an agent, then selects three substitutions *(c0=3.4, v0=x3, v1=x5), (c0=-.89, v0=x10, v1=x2), (c0=302.24, v0=x0, v1=x9)* at random, proceeds to evaluate each of the three substitutions, and finally selects the one substitution with the highest fitness. One could then cache the original abstract expression, *regress(c0*sin(v0/v1)*, and its final fitness score, along with a memo denoting the chosen substitution, *(c0=-.89, v0=x10, v1=x2)* which allowed the abstract expression to achieve the fitness score.

The techniques of particle swarm optimization (Eber01) and differential evolution (Price05) offer an approach to training agents which have been compiled from an abstract grammar such as AXP. Both particle swarm and differential evolution techniques can be applied in both a discrete or a continuous vector space. We support either technique at the option of the user.

The compiler prepares WFF agents for optimization by recognizing each abstract constant and each abstract variable reference. Assuming that after compilation there are K abstract constants and J abstract variable references. The compiled agent will contain a vector, *C*, of real values of length K, and a vector, *V*, of integer values of length J. The *C* and *V* vectors will be used to memoize the concrete choices for each abstract constant and each abstract variable reference. The WFF agent's code is then compiled with indirect indexed references into the *C* and *V* vectors. For instance, the expression, *regress(c0*sin(v0/v1)*, is compiled as, *regress(C[0]*sin(x[V[0]]/x[V[1]])*. Assuming that the compiled WFF agent contains vectors *C=(3.4,5.6,-2.5)* and *V=(3,2,0,1)*, after particle swarm or differential evolution have made their choices, then the indirect indexed code, *regress(C[0]*sin(x[V[0]]/x[V[1]])*, is equivalent to, *regress(3.4*sin(x[3]/x[2])*, which is equivalent to, *regress(3.4*sin(x3/x2)*.

Detailed descriptions of the REG, MVL, and EXP grammars plus our parameters for standard GP and our methods of managing well-formed-formula (WFF) agent candidates can be found in (Korns01). Clearly when we add abstract WFFs to the system we add a layer of evolved optimization underneath that of the genetic programming. In our system the handoff is seamless and occurs when the GP machinery tells the compiled WFF agent to compute its fitness score.

In adding particle swarm and differential evolution to our system, the basic algorithmic components are abstract and concrete WFFs, a memo cache of WFFs, the survivor population of the fittest WFFs, and a list of champion WFFs. We used the *regressGSOSR* option settings which are almost directly in line with (Koz92i) and are as follows. At the initialization step of every training "run", exactly 1000 randomly generated WFFs, in the *REG(AXP)* grammar, are evaluated.

Evaluation of each *REG(AXP)* candidate involves a particle swarm (PS) or differential evolution (DE) optimization within the candidate agent. After evaluation, the *C* and *V* vectors will be filled with the concrete choices (for the abstract constants and the abstract variable references) which result in the best fitness score. The size of the PS and/or DE population pool is 25 and the number of generations to optimize is also 25.

All evaluated WFFs are memoized (saved in a memo cache) and also saved in sorted order by fitness score (in the survivor population). The top twenty-five WFFs participate in the genetic operations of mutation and crossover (Koz92i) during each incremental generation. The probability of mutation is 10%, and the probability of crossover is 100%. When a WFF is chosen for crossover, its mate is chosen at random from the WFFs of lower fitness within the top twenty-five fittest individuals (in the survivor population). Crossover is always performed twice with the same parents and always produces two children which are evaluated, memoized, and saved in sorted order by fitness score (in the survivor population). The maximum number of generations before training halts is provided at the start of training time. If ten generations pass with no change in the fittest WFF, then system saves the fittest WFF in its list of champions, clears all WFFs in the survivor population (but not the memo cache) and evaluates 1000 randomly generated WFFs, starting a new "run". Any new "run" does not reset the generation count. Training always proceeds until the maximum number of generations have been reached. If G represents the maximum number of generations allowed for a fully automated training cycle, then the maximum number of independent "runs" is (G/10). Depending upon the progress in training, there may only be a single "run" during the entire training process. At the completion of training, the fittest champion WFF (the fittest WFF ever seen) is chosen as the result of the training process.

1.8 Adding Context-Aware Crossover

In (Maj06) an extension of standard GP crossover is devised. In standard GP crossover (Koz92i), 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*.

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.

We further extended context-aware crossover such that *all possible valid* snips of genetic material from the father s-expression are substituted into the mother s-expression *at all possible valid locations*. Whereupon *all possible valid* snips of genetic material from the mother s-expression are then substituted into the father s-expression *at all possible valid locations*.

As an example, this extended context-aware crossover between a father *regress((v0 + c0) / v1)* and a mother *regress(sin(v2*c1))* produces the following children.

- regress(sin(v2*c1) / v1)
- regress((sin(v2*c1) + c0) / v1)
- regress((v0 + sin(v2*c1)) / v1)
- regress((v0 + c0) / sin(v2*c1))
- regress(v2 / v1)
- regress((v2 + c0) / v1)
- regress((v0 + v2) / v1)
- regress((v0 + c0) / v2)
- regress(c1 / v1)
- regress((c1 + c0) / v1)
- regress((v0 + c1) / v1)
- regress((v0 + c0) / c1)
- regress(sin(((v0 + c0) / v1)*c1))
- regress(sin(v2*((v0 + c0) / v1)))
- regress(sin((v0 + c0)*c1))
- regress(sin(v2*(v0 + c0)))
- regress(sin(v0*c1))
- regress(sin(v2*v0))
- regress(sin(c0*c1))

- regress(sin(v2*c0))
- regress(sin(v1*c1))
- regress(sin(v2*v1))

We add our extended context-aware crossover to all GP runs in our system with a varying probability by generation. During the first generation of every GP run, the probability of extended context-aware crossover is 100%. This probability declines linearly until the probability of the extended context-aware crossover is 0% during the final generation.

1.9 Boosting Using Island GP with Multiple Grammars

We introduce genetic diversity via boosting across multiple independent island populations. There is a correlation between the fittest individuals in a training run and genetic diversity in the population (Almal06). Furthermore the fittest individuals, in a training run tend to cluster around a set of common root expressions (Daida05) and (HS04a). Capitalizing on these observations, we set our symbolic regression tool to support boosting across multiple islands with the simple linear regression, *REG*, grammar.

We use as many independent islands as there are columns in the regression problem. Each island is evolved for a total of 10 generations using the abstract grammar *REG(AXP)*. If there are *M* columns, we repeat the boosting process *M* times. When the mth island population has produced its champion WFF agent, the estimation vector *EY* is subtracted from the dependent variable *Y* to produce *Ym* the new dependent variable for the m+1 island population to regress upon.

At the termination of the Mth island symbolic regression, we have M abstract simple linear WFF champions each of which have regressed on the boosted dependent variable. We now assume that the following multiple linear regression will best model that original dependent variable.

- mvlregress(wff0,...,wffM)

Each of the M champions are converted from their abstract AXP grammar representations into concrete EXP grammar representations, and a single *mvlregress(wff0,...,wffM)* candidate, known as *Eve*, is entered into the final island as the first individual. Using extended context-aware crossover on the individual WFF s-expressions contained in *Eve*, we create and additional 1000 individuals. A standard GP run, as in (Korns01), is then run for 10 generations and the resulting most fit *mvlregress(wff0,...,wffM)* model is chosen as our final champion.

1.10 Final Results

Our final experiment was to use the system with multiple island boosting, using the simple linear REG(AXP) grammar and then populate a final "island of champions" using the multiple regression MVL(EXP0,...,EXPM) grammar. [2]. The results of training on the nine test cases, using the *regressGSOBOOST* option settings on 1 million rows and twenty columns with 40% random noise, are shown in the table below.

Table 1. Result For 1M rows by 20 columns Random Noise

Test	*Minutes*	*Train-Error*	*Test-Error*	*Classify*
cross	2820	0.83	0.67	0.72
cubic	2278	0.39	0.40	0.91
hyper	2154	0.85	0.86	0.47
elipse	3171	0.70	0.55	0.82
hidden	2386	0.11	0.00	0.99
linear	2400	0.10	0.01	0.99
mixed	2845	0.67	1.55	0.64
ratio	2582	0.30	0.94	0.00
cyclic	2336	0.43	0.32	0.05

Description of table headings:

- *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
- *Classify:* The classification score for the testing data

Fortunately, training time is mostly within our 3000 minute (50 hour) limit (only the *elipse* test case is slightly over). In general, average percent error performance is poor with the *linear* and *hidden* 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 *cyclic* and *ratio* test cases. If we were to run a market neutral hedge on hypothetical markets, driven by these nine test models, we would have lost money in none of the markets, broken even in the markets driven by the *ratio* and *cyclic* models, and made good money in all other markets.

We were nearly prescient on the *linear*, *hidden*, and *cubic* market models realizing over 90% of theoretically possible profits. We achieved more than

[2] This entire experimental setup can be chosen by selecting the system's *regressGSOBOOST* option settings.

60% of theoretically possible profits even in the more difficult *cross*, *elipse*, and *mixed* market models.

1.11 Summary

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

Clearly we need to experiment with techniques which will improve our performance on the *mixed* and *cyclic* test cases. Areas for future research include: (i) using standard statistical and Bayesian analysis to help build conditional WFFs for multi-modal markets and (ii) experimentation with additional experimentation with grammar expressions to increase the speed of evolutionary training and to develop a better understanding of hill-climbing operators from a root grammar viewpoint.

References

1. *Aho86* "Compiler Principles, Techniques, Tools", Alfred V Aho, Ravi Sethi, Jeffery D. Ullman; Addison-Wesley Publishing; 1986.
2. *Daida05* "Considering the Roles of Structure in Problem Solving by a Computer", in Genetic Programming Theory and Practice II, J Daida; Springer, New York; 2005.
3. *Almal06* "Content Diversity in Genetic Programming and Its Correlation with Fittness", in Genetic Programming Theory and Practice III, A Almal, WP Worzel, E A Wollesen, C D MacLean; Springer, New York; 2006.
4. *YCK04a* "Discovering Financial Technical Trading Rules Using Genetic Programming with Lambda Extraction" in Genetic Programming Theory and Practice II, Tina Yu, Shu-Heng Chen, Tzu-Wen Kuo; Springer, New York; 2005.
5. *HS04a* "Does Genetic Programming Inherently Adopt Structured Design Techniques?" in Genetic Programming Theory and Practice III, John Hall, Terence Soule; Springer, New York; 2006.
6. *Chen02* "Genetic Algorithms and Genetic Programming in Computational Finance" edited by Shu-Heng Chen; Kluwer Academic Publishers, Dordrecht Netherlands; 2002.
7. *Korns01* "Large-Scale, Time-Constrained, Symbolic Regression" in Genetic Programming Theory and Practice IV, Michael Korns; Springer, New York; 2006.
8. *Koz92i* "Genetic Programming: On the Programming of Computers by Means of Natural Selection" John R. Koza; The MIT Press, Cambridge Massachusetts; 1992.
9. *Koz94i* "Genetic Programming II: Automatic Discovery of Reusable Programs" John R Koza; The MIT Press, Cambridge Massachusetts; 1994.
10. *Koz99i* "Genetic Programming III: Darwinian Invention and Problem Solving" John R Koza, Forrest H Bennett III, David Andre, Martin A Keane; Morgan Kaufmann Publishers, San Francisco, California; 1999.
11. *Koz03i* "Genetic Programming IV: Routine Human-Competitive Machine Intelligence" John R Koza, Martin A Keane, Mathew J Streeter, William Mydlowec, Jessen Yu, Guido Lanza; Kluwer Academic Publishers, Dordrecht Netherlands; 2003.
12. *ONeil03* "Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language" Michael O'Neill, Conor Ryan; Kluwer Academic Publishers, Dordrecht Netherlands; 2003.
13. *CB04a* "Lessons Learned Using Genetic Programming in a Stock Picking Context" in Genetic Programming Theory and Practice II, Michael Caplan, Ying Becker; Springer, New York; 2005.
14. *Eber01* "Swarm Intelligence" Russell Eberhart, Yuhui Shi, James Kennedy; Morgan Kaufman, New York; 2001.
15. *Price05* "Differential Evolution: A Practical Approach to Global Optimization" Kenneth Price, Rainer Storn, Jouni Lampinen; Springer, New York; 2005.
16. *Maj06* "Using context-aware crossover to improve the performance of GP" in GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation,Hammad Majeed and Conor Ryan; ACM Press, New York; 2006.