GENETIC PROGRAMMING WITH EMBEDDED FEATURES OF SYMBOLIC COMPUTATIONS

Yaroslav V. Borcheninov and Yuri S. Okulovsky
Institute of Mathematics and Computer Sciences, Ural Federal University, Mira str. 56, Yekaterinburg, Russia

Keywords: Genetic programming, Symbolic computations.

Abstract: Genetic programming is a methodology, widely used in data mining for obtaining the analytic form that describes a given experimental data set. In some cases, genetic programming is complemented by symbolic computations that simplify found expressions. We propose to unify the induction of genetic programming with the deduction of symbolic computations in one genetic algorithm. Our approach was implemented as the .NET library and successfully tested at various data mining problems: function approximation, invariants finding and classification.

1 INTRODUCTION

Genetic programming (Koza, 1992) is a methodology of using the genetic algorithms (Goldberg, 1986) to find a program that performs a user-specified task. We consider the particular case of genetic programming that operates not with arbitrary programs, but with expressions. Genetic programming (GP) is widely used to obtain the analytic form of the experimental data in natural sciences (Schmidt and Lipson, 2009), robotics (Robertson and Dumont, 2002), economics (Koza, 1994), medicine (Zhang and Wong, 2008), etc.

The classic GP approach can shortly be described as follows. The expressions are represented as the operator trees. Initially, the *population* consists of the randomly generated expressions. On each algorithm's iteration, the following actions are performed:

- Mutation. The randomly chosen expression is changed by a replacement of a node.
- Crossover. Two randomly chosen expressions exchange subtrees.
- After all the mutations and crossovers are performed, the resulting expressions' set is subjected to the selection, which evaluates how each expression fits the experimental data. The least valuable expressions are then removed from the population.

The well known problem of GP is the excessive growing of expressions, or *bloating*. Various methods are proposed to resolve the issue: the limitation of the tree's depth; special mutations and crossovers that preserve the expressions' size; selection that sorts out bloated trees (Poli et al., 2008); removal of subtrees that have lesser analogues in the population (Mori et al., 2009).

The obvious way to reduce the expression's size is the algebraic or numerical simplification. If the algorithm has succeeded in finding a correct expression, the expression can be then simplified for a better perception. However, aside from producing non-aesthetic solutions, bloating also significantly reduces the algorithm's performance. Recent studies (Zhang et al., 2006; Kinzett et al., 2008) show the effectiveness of the *online simplification*, when expressions are simplified during the evolution.

The simplification of the expression inevitably leads to the potential growing points' elimination. For example, while approximating the function $(x+1)y^2$, the intermediate solution $(1+1)y^{1+1}$ can be found. This solution will be simplified to $2y^2$, which requires at least two mutations to become a correct answer, e.g. $2y^2 \Rightarrow xy^2 \Rightarrow (x+1)y^2$. The initial solution $(1+1)y^{1+1}$ requires only one mutation $(1+1)y^{1+1} \Rightarrow (x+1)y^{1+1}$. Hence, the simplification hampers the evolution in this case. On other hand, the partial simplification $(1+1)y^{1+1} \Rightarrow (1+1)y^2$ does not show such effect for the function $(x+1)y^2$, but does so for $2y^{x+1}$. Therefore, the question of where to apply the simplification depends on the problem specification, on the particular found expression, etc.

The main idea of our work is to integrate the online simplification — and, more general, arbitrary symbolic computations — with genetic programming on the most basic level. Symbolic computations transform the expression according to some rules and do not change the function, encoded by the expression. Let us call such transformations *deductive*. Transform $x+2x \Rightarrow 3x$ is both deductive and simplifying, while $(x+y)z \Rightarrow xz+yz$ is deductive but not simplifying, since it is not always clear, which form is more preferable. The *inductive* transforms change both the form of the expression and the encoded function. Mutations and crossovers described above are inductive.

To combine inductive and deductive transforms, we introduce the following changes in the classic genetic programming algorithm. For the mutations, a collection of rules is defined. Each rule transforms an expression in inductive or deductive way. When we need to perform the mutation operation, we randomly select a rule from the collection and apply it to an expression. Crossover is also defined by the set of rules. Crossover's rules have a slightly different format, they accept two expressions and produce one.

Inductive transforms bloat the tree, while deductive transforms hamper the induction. Therefore, inductive and deductive tendencies are in the opposition. Therefore, we should measure the fitness for both tendencies independently, to find the appropriate balance between them. In our variation of GP, the selection is performed based on various metrics. We calculate these metrics for each expression, obtain their weighted total, and then remove the expressions that have the least weighted total in the population.

Our approach was implemented in C# language (Drayton et al., 2002) as a library for .NET framework, and tested in various data mining problems. The project will be released under GPL v.3. license.

2 ALGORITHM'S ESSENTIALS

An expression is represented as a tree of *nodes*. Three types of nodes are considered: *constants*, *variables* and *operators*. Each node has a *return type*, which is an arbitrary C# type. Different return types can be used in one expression.

Each tree can be compiled into .NET lambda expression. Suppose $f(x_1,\dots,x_n)$ is a function encoded by a tree. Let a be an array of the arguments $a=(x_1,\dots,x_n)$. A node for a constant c is compiled into the lambda $a \mapsto c$. A node for the i-th variable is compiled to $a \mapsto a[i]$. If a node encodes an operator $g(y_1,\dots,y_k)$, it is compiled into $a \mapsto g(c_1(a),\dots,c_2(a))$, where c_i is the compiled i-th child of the node. The compilation of the nodes is possible due to the abstract syntax trees, one of .NET features. It improves the performance of the evaluation.

Trees can be modified according to rules. A rule consists of a condition and an action. The first stage of a rule's application is finding all the tuples of nodes that satisfy the condition. The second stage is to apply the action to one of the selected tuples. Let us consider some examples of the rules.

```
select  ?A    where  A.Type=double
              mod    A→Plus(A,c)              (R1)
```

Here *A* is an identifier of selected node, *c* is a random constant. The rule R1 processes a tree and selects all its nodes of `double` type. It is possible to apply the rule to one of the selected nodes, and replace the node with a new subtree.

The R1 rule allows us introducing an addition in a tree. Due to the type check `A.Type=double`, the `Plus` operation can only be applied to a `double` node, and therefore the tree remains correct. That shows how the rules assure the correctness of the mutations and crossover.

The following R2 rule shows, how the operation can be removed from a tree.

```
select  ?A(.B)   where  A.Type=B.Type
                 mod    A→B                   (R2)
```

The rule R2 searches for all pairs (A,B), where *A* is an arbitrary node, *B* is an arbitrary child of *A*, and types of *A* and *B* coincide. In each such pair, *A* can be replaced with *B*.

We also need rules for simplification of expressions. The following rules R3 and R4 are examples of such rules.

```
select  ?A(.B,.C)   where  A is Plus &&
                           B is Const &&
                           C is Const          (R3)
                    mod    A→B.Val+C.Val

select  ?A(B(C))    where  A is Minus &&
                           B is Minus          (R4)
                    mod    A→C
```

Crossover can also be based on the rules. The following rule R5 is the simplest crossover that exchanges subtrees.

```
select  ?A,?B   where    A.Type=B.Type
                produce  A→B; ret A.Root      (R5)
```

This rule accepts two trees, searches for a pair (A,B) where *A* is from the first tree, *B* is from the second tree, and their types coincide. Since the rule accepts two trees instead of one, it is not clear in which of them the crossover's result is stored. To resolve this, the **mod** clause is replaced with the **produce** clause, which modifies trees and returns some node as the result of the rule's application. More complex crossover schemata are available. For example, the following rule R6

select A,B **where** A.Type=double &&
B.Type=double (R6)
produce **ret** Div(Plus(A,B),2)

is applicable only to the trees' roots A and B and returns their half-sum.

We have developed an elegant way to define rules in C#. The rules can be programmed in the almost natural way, by only defining its logic, without excessive code to adopt this logic to C#. That was achieved with the intensive use of lambda-expressions, generics and code generation. For example, rule R1 can be programmed with the code in List. 1.

Listing 1: Rule R1 definition in C#.

```
var rule=Rule
  .New("Intro +")
  .Select("?A")
  .Where<INode>(c=>c.A.Type
                    ==typeof(double))
  .Mod(c=>c.A.Replace(new Plus(c.A,0)))
```

Rules are very numerous and their categorization is necessary. The first category is universal rules that are applicable to any expressions. The rules R2 and R5 are in this category. The second category of rules describes data types. The following rules are required for each data type:

T1 Introduction of the constant: replacement of a subtree with the return type T to a constant of the same type;

T2 Introduction of the varuable: replacement of a subtree with the return type T to a variable of the same type;

T3 Adjustment of the constant: replacement of a constant by another constant with the close value. For example, floating point constant c may be replaced to a random number from an interval $[c(1-\varepsilon), c(1+\varepsilon)]$.

Rules of the third category describe *domains* of operations: the sets of operations that are often used in expressions together. For example, the algebraic domain consists of addition, subtraction, multiplication, and so on. In each domain, the following types of rules should be developed:

D1 Introduction of each operator (R1);

D2 Calculation rules for each operator (R3);

D3 Deductive rules for the operators (R4, distributivity laws, De Morgan's laws);

D4 Special crossover rules, if they are available (R6).

In the programming implementation, an arbitrary amount of *tags* can be chosen for each rule. Tags indicate the category of the rule, the domain it belongs to, whether the rule is purposed for mutation or crossover, etc. During the work of the algorithm, each tag is associated with its weight. We calculate the weight of each rule as a product of associated tags' weights. The weight of the rule determines how often it will be used. The probability of applying a rule with weight w is w/W, where W is a total sum of all rules' weights. Tags and weights allow us managing the algorithm. For example, on the early stage, when the optimal solution is not found, inductive rules should be applied more often. When the optimal solution is found and we need to get its acceptable presentation, we should use calculation and deductive rules.

3 APPLICATION AREAS AND METRICS

In the *function approximation* problem we are given a set of tuples $\{(x_{i,1}, x_{i,2}, \ldots, x_{i,m}, y_i) \ : \ i = 1, \ldots, n\}$, where $y_i = f(x_{i,1}, \ldots, x_{i,m}) \cdot c_i$ and c_i is a random number from the interval $[1-\alpha, 1+\alpha]$. The goal is to find the analytic form of f. To do that, we use our algorithm with the following two metrics. *Fitness* metric for the function g, found by the algorithm, is calculated as

$$\rho(g) = \left(1 + \sum_{i=1}^{n} |g(x_{i,1}, \ldots, x_{i,m}) - y_i|\right)^{-1}.$$

Taking the reciprocal value is important, because it allows bounding the value of ρ, and provides correspondence between a higher value of ρ and a better expression.

Length metric $\lambda(g)$ is a number, reciprocal to the count of operations in g. Valuation of an expression is determined as a weighted total $e(g) = w_\rho \rho(g) + w_\lambda \lambda(g)$. Typically, $w_\rho = 1$ and $w_\lambda = 0.1$. In our implementation of the algorithm, we allow user adjusting metrics' weights during the algorithm's work. Such adjustment leads to interesting effects. For example, setting the weight of the length metric to a negative value can drive the algorithm out of the local minimum. On the other hand, when the average ρ of the population is high, increasing the length metric to 0.2–0.3 allows finding the most compact form of g.

In the *invariants finding* problem we are given the set $\{(x_{i,1}, x_{i,2}, \ldots, x_{i,m}) \ : \ i = 1, \ldots, n\}$ and need to find such f that $f(x_{i,1}, \ldots, x_{i,m}) \approx 0$ (or equals to zero in the absence of the noise). The algorithm requires three metric to solve the problem. The first metric is the length metric λ. The second metric is the *invariance* metric

$$\iota(f) = \left(1 + \sum_{i=1}^{m} f^2(x_{i,1}, \ldots, x_{i,m})\right)^{-1}.$$

However, these two metrics are not enough. The expression $\frac{1}{2^{100}+x}$ is almost invariant on small x, however this expression is not acceptable. The solution is introducing the *tautology* metric

$$\tau(f) = 1 - \left(1 + \sum_{i=1}^{k} f^2(y_{i,1}, \ldots, y_{i,m})\right)^{-1},$$

where $y_{i,j}$ are random numbers. Typical weights of metrics are $w_\iota = 1$, $w_\tau = 1$ and $w_\lambda = 0.1$.

In *classification* problem we are given the set $\{(x_{i,1}, x_{i,2}, \ldots, x_{i,m}, c_i) \ : \ i = 1, \ldots, n\}$, where c_i is a Boolean value indicating whether the corresponded tuple belongs to a class. We need to use the rules for floating point type and associated operators' domains; rules to support Boolean type (defined by items T1–T3 in the section 2); rules to support relation operators $<, >, =$ (only D1 and D2, because these operators do not preserve the operands' types); rules for operators $\vee$, $\wedge$, $\neg$ (D1–D4).

The fitness metric is adjusted as follows

$$\sigma(g) = \left(1 + \frac{|\{i \ : \ g(x_{i,1}, \ldots, x_{i,m}) = c_i\}|}{n}\right)^{-1}.$$

4 CONCLUSIONS AND FUTURE WORK

We have proposed a methodology of genetic programming algorithm that embeds the features of symbolic computations. This approach was implemented in .NET library. We have supported algebraic, trigonometric and comparison operations with floating-points numbers, as well as logical operations with Boolean values. Using the library, we were able to solve the different data mining problems: function approximation, invariants finding and classification.

Our future research will be conducted in the following directions:

- Finding the parameters that provide the most efficient GP performance. By our observation, changing of rules' and metrics' weights leads to significant changes in performance. Moreover, changing the parameters during the algorithm's work has different effect depending on the current state of the population. We believe that the thorough examination of such effects can lead to significant improvements in genetic programming.
- Using genetic programming in new domains: fuzzy numbers, fuzzy logic, temporal logic, etc.
- Exploring substitutions for length metric. Length metric does not seem to catch the intuitive meaning of a "good" expression. We plan to introduce computation complexity and aesthetics metrics instead, and understand how it improves the work of the algorithm.

ACKNOWLEDGEMENTS

The work is supported by the Russian Federation President's program MK-844.2011.1.

REFERENCES

Drayton, P., Albahari, B., and Neward, T. (2002). *C# in a Nutshell*. O'Reilly.

Goldberg, D. (1986). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Kinzett, D., Johnston, M., and Zhang, M. (2008). Numerical simplification for bloat control and analysis of building blocks in genetic programming. *Evolutionary Intelligence*, 4.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA.

Koza, J. R. (1994). Genetic programming for economic modeling. In *Intelligent Systems for Finance and Business*.

Mori, N., McKay, B., Hoai, N. X., Essam, D., and Takeuchi, S. (2009). A new method for simplifying algebraic expressions in genetic programming called equivalent decision simplification. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 13(14):237–238.

Poli, R., Langdon, W. B., McPhee, N. F., and Koza, J. R. (2008). *A Field Guide to Genetic Programming*.

Robertson, A. P. and Dumont, C. (2002). Design of robot calibration models using genetic programming. In Mayorga, R. V. and Rios, A. S.-D. L., editors, *Proceedings of the Third International Symposium on Rob. and Autom.*, volume 3, pages 449–454.

Schmidt, M. and Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85.

Zhang, M. and Wong, P. (2008). Genetic programming for medical classification: a program simplification approach. *Genetic Programming and Evolvable Machines*, 9(2):229–255.

Zhang, M., Wong, P., and Qian, D. (2006). Online program simplification in genetic programming. *Simulated Evolution and Learning - SEAL*, pages 592–600.