Convergence of symbolic regression using metaheuristics

Ben Cardoen
ben.cardoen@student.uantwerpen.be

April 29, 2017

1 Abstract

Symbolic regression (SR) fits a symbolic expression to a set of expected values. Amongst its advantages over other techniques is the ability for a practitioner to interpret the resulting expression, determine important features by their usage in the expression, and an insight into the behavior of the resulting model (e.g. continuity, derivatives, extrema). SR combines a discrete combinatoric problem (combining base functions) with a continuous optimization problem (selecting and mutating constants). One of the main algorithms used in SR is Genetic Programming (GP). The convergence characteristics of SR using GP is still an open issue. The continuous aspect of the problem has traditionally been an issue in GP based symbolic regression. This paper will study convergence of a GP-SR implementation on selected use cases known for bad convergence. We introduce modifications to the classical mutation and crossover operators and observe their effects on convergence. The constant optimization problem is studied using a two phase approach. We apply a variation on constant folding in the GP algorithm and evaluate its effects. The hybridization of GP with 3 metaheuristics (Differential Evolution, Artificial Bee Colony, Particle Swarm Optimization) is studied.

2 Introduction

3 Design

In this section we will detail the design of our tool, the algorithm and its parameters.

3.1 Algorithm

3.1.1 Input and output

The algorithm accepts an X = n x k matrix of input data, and an Y = 1 x k vector of expected data. It will evolve expressions that result, when evaluated on X, in an 1 x k vector Y' that approximates Y. N is the number of features, or parameters, the expression can use. K is the number of datapoint per feature. The algorithm makes no assumptions on the domain or range of the expressions or data set. While domain specific knowledge can be of great value, in real world situations such data is not always known.

3.1.2 Implementation

Our tool is implemented in Python. As with any software tool, the

3.2 Fitness

3.2.1 Distance function

The goal of the algorithm is to find f' such that

$$Y = f(X)$$

$$Y' = f'(X)$$

$$dist(Y, Y') = e$$

e is minimal.
Not all distance functions are equally well suited for this purpose. A simple root mean squared error (RMSE) function has the issue of scale, the range of this function is [0, +∞), which makes comparison problematic, especially if we want to combine it with other objective functions. A simple linear weighted sum requires that all terms use the same scale. Normalization of RMSE is an option, however there is no single recommended approach to obtain this NRMSE.
In this work we use a distance function based on the Pearson Correlation Coefficient. Specifically, we define

$$dist_p(Y, Y') = 1 - |r|$$

with r

$$r = \frac{\sum_{i=0}^{n}(y_i - E[Y]) * (y'_i - E[Y'])}{\sqrt{\sum_{j=0}^{n}(y_j - E[Y])^2 * \sum_{k=0}^{n}(y'_k - E[Y'])^2}}$$

This function has a range [0,1] which facilitates comparison across domains and allows combining it with other objective functions. The function reflects the aim of the algorithm. We not only want to assign a good (i.e. minimal) fitness value to a model that has a minimal distance, we also want to consider linearity between Y an Y'.

3.2.2 Diversity

Diversity, the concept of maintaining semantically different specimens, is an important aspect in metaheuristics. Concepts such as niching and partitioning are often used to promote this behavior, amongst other reasons to prevent premature convergence or even to promote multimodality. Our tool uses a simple measure that mimics the more advanced techniques stated above. It should be clear that for any combination of input and output data, there are a huge set of expressions with an identical fitness value. Such expressions can lead to premature convergence, consider a population of 5 where the 3 best samples all have fitness scores of 0.134. Our tool will aim to prevent retaining expressions that have identical fitness values. In contrast, in some metaheuristics [2] allowing replacement of solutions with identical fitness values (not duplication, but replacement), can actually help avoiding local minima.

3.2.3 Predictive behavior

The algorithm evaluates specimens based on training data X_t, with X_t a random sample from input data X of size j = r * k. R is the sampling ratio, the ratio between training and test data. K is the total amount of datapoints available per feature. After completion of the algorithm the population is ordered based on minimized fitness values calculated on the training data. In real world applications practitioners are also interested in the predictive qualities of the generates expression. In other words, how well do the expressions score on unknown data. In the final evaluation we score each expression on the full data to obtain this measure. While this gives us some information on how good an expression is on the full data set, what we would like to know is how the algorithm's progress is related to the value. For example, if we add 10 more generations, or increase the population by a factor 1.5, do we gain or lose in predictive quality of the expressions? To define this we use a correlation between the fitness values using the training data and those of the full data. This provides an immediate link between the two sets of values. Finally, we calculate a correlation trend between the training fitness values at the end of each phase, and the final fitness values calculated on the full data. This gives us good measure of how the algorithm will converge on a good solution on the full data.

3.2.4 Convergence limit

As a stopcondition our tool uses a preset number of iterations. The user specifies the number of generations (g) and phases (r), and the algorithm will at most execute g x r iterations. Convergence stalling is implemented by keeping track of the number of successful operations (mutation or crossover). If this value crosses a threshold value convergence has stalled and the algorithm is terminated.

3.3 Initialization

Initialization is done using the 'full' method [1]. The algorithm has a parameter initial depth, each new expression in the population is created using that depth.

3.3.1 Invalid expressions

Generating a random expression is done by generating random subtrees and merging them. A critical point to observe is that randomly generating expressions can be invalid for their domain. The ubiquitous example here is division by zero. Several approaches to solve this problem exist. One can define 'safe' variants of the functions, in case of division by zero, one can return a predefined value that will still result in a valid tree.

The downside of this approach is that the division function's semantics is altered, a practitioner, given a resulting expression, would have to know that some functions are no longer corresponding entirely to their mathematical equivalents. The other option is assigning maximum fitness to an invalid expression. While simple, this approach needs a careful implementation. From a practical standpoint wrapping functions in exception handling code will quickly deteriorate performance. Our approach is quick domain check for each calculation, avoiding exceptions.

Invalidity probability We define the probability that a randomly generated tree of depth d, with n possible variables, k possible base functions, and j data points as q. With more complex problems d will have to be increased. GP is also susceptible to bloat, increasing d even further. This issue will affect generation of subtrees in mutation. With more datapoints the probability of at least one leading to an invalid evaluation will increase linearly. An increase in d will lead to an exponential increase in the number of nodes in the tree. A node can be either a basefunction or a leaf (parameter or constant). For each additional node the probability q increases. We can conclude that q, while irrelevant for small problems and low depths, becomes a major constraint for larger problem statements. A performant approach of this issue is therefore needed.

Bottom up versus top down There are two methods to generate a tree based expression : bottom up and top down. The top down approach is conceptually simpler, we select a random node and add randomly selected child nodes until the depth constraint in satisfied. The problem with this approach is that the expression can only be evaluated at completion, early detection of an invalid subtree is impossible. In contrast a bottom up construction, where we generate small subtrees, evaluate them and if and only if valid merge them into a larger tree, allows for early detection of invalid expressions. A downside of this approach is the repeated evaluation of the subtrees. This can be mitigated by caching the result of the subtree.

Disallowing invalid expressions in initialization We can generate random expressions and let the evolution stage of the algorithm filter them out based on their fitness value, or we can enforce the constraint that no invalid expressions are introduced. The last option is computationally more expensive at first sight, since the algorithm is capable by definition of eliminating unfit expressions from the population. This can lead to unwanted behavior in the algorithm itself. For high q values we can have a significant part of the population that is at any one time invalid. This can lead to premature convergence, similar to a scenario where the population is artificially small or dominated by a set of highly fit individuals. Another observation to make is the algorithm will waste operations (mutation, crossover) on expressions that are improbable to contribute to a good solution. While more expensive computationally, we therefore prohibit generation of invalid expressions in the initialization.

3.4 Evolution

3.4.1 Mutation

3.4.2 Crossover

3.5 Selection

After evolution a decision has to be made on which expressions will be retained in the population. In our tool the population is fixed, so a replacement is in order. We use an elitist approach. If a an expression has a lower (better) fitness value after mutation, it is replaced. In crossover we combine two expressions r, and t, resulting in two offspring s, u. From these four expressions the two with minimal fitness survive to the next generation.

3.6 Archiving

The algorithm holds an archive that functions as memory for best solutions obtained by seeding, from other processes, or the best expressions at the end of a phase. At the end of a phase we store the j best expressions out of the population. J is a parameter ranging from 1 to the population size n. With j == n we risk premature convergence, with j == 1 the risks exists that we lose good expressions from phases which will have to be rediscovered. While there are numerous archiving strategies described in literature, we use a simple elitist approach. This means that there is no guarantee that the best j samples of phase i are retained, if they have fitness values lower than those present in the archive and the archive has no more empty slots, they will be ignored. This leads us to the size of the archive. While no exact optimal value for this exist, in order to function as memory between phases it should be similar in size to the amount of phases. The j parameter will influence this choice as well.

3.7 Representation and data structures

3.7.1 Expression

Tree representation

Base functions

Constants

Features

3.7.2 Population

3.8 Parameters

3.8.1 Depth

3.8.2 Populationsize

3.8.3 Phases

3.8.4 Generations

3.8.5 Samples

3.8.6 Domain

3.8.7 Operator parameters

3.9 Constant Optimization

3.9.1 Initialization revisited

3.9.2 Folding

3.9.3 ABC

3.9.4 PSO

3.9.5 DE

3.10 Features

3.10.1 Incremental support

3.10.2 Statistics

3.10.3 Visualization

3.11 Conclusion

4 Experiments

5 Related Work

6 Conclusion

7 Future work

References

[1] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[2] STORN, R., AND PRICE, K. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization 11*, 4 (1997), 341–359.