FunctionFinder − A symbolic genetic algorithm to find interpolating functions

Giancarlo Niccolai

Know How 7

Viale Ranzoni 17 − 20100 Milan (IT) − gian@niccolai.ws

Abstract − The problem of interpreting behavior of customers in an organization rises the need to find a behavioral model, that is a function that ties behavior (output) to observations (input). Finding parameters of a given function can be an easy task; but finding an unknown relationship is a problem to be treated with genetic symbolic algorithms.

I.THESIS

Let's assume that we have a vector *Z* of *k* statistical couples, which assumes the form of:

$$Z=\{(X_1,y_1),(X_2,y_2),\ldots,(X_k y_k)\}$$

Where X_i is a vector of *n* variables $X_i=\{x_{1,}\ x_{2,}\ldots,x_n\}$ and y_i is a real number.

We want to find a $f(X_i)$ which is a theory that binds $\mathbb{R}^n$ to $\mathbb{R}$:

$$f(X_i):\mathbb{R}^n\rightarrow\mathbb{R}$$

so that $f(X_i)=y_i$

Note that a simple genetic algorithm is capable to find out that function *f*, but it will not explicitate the function. We could compare a genetic algorithm that evaluates an y_i knowing X_i as an *operator*: an operator is a discrete system with no memory (no status variables), that receives as input a data series, and learns to synthesize a correct value, like that: $M(X_i)=y_i$. That is good if you want to interpolate the series, and you want to predict what will be a result when unknown input conditions are given. But is useless if you are willing to know what is the law of transformation between the output and the input.

To clarify and explicate the relationship between statistical observations and a desired or observed outcome, the genetic algorithm must be symbolic.

That means that the genetic algorithm will not simply find a solution: *it will find a way to find the solution,* and it will tell us how to do. It will explain the rules to get a coherent result in a language familiar to us. The difference is that a simple genetic algorithm is a mechanism capable of obtaining a result, while a symbolic genetic algorithm is like a teacher, that once found a solution, will tell us how the solution have been reached.

II. PRACTICAL CONSEQUENCES

The semantic value of this approach is three−dimensional, compared with the two−dimensional value of information given us by a simple genetic algorithm. In example, we could teach (with the same efforts) to a simple and to a symbolic genetic algorithm to play chess. The first one will learn how to beat us. The second one will tell us how to beat ourselves. That *how* makes great difference, because no information can be extrapolated from the first genetic, while in the latter case we'll have a way to increment how knowledge stock.

In that sense, symbolic genetic algorithms are *knowledge producers*, while simple genetic algorithms are *information producers*. The difference is that knowledge is (or can) be shared, while information generated by simple genetic algorithms is retained inside themselves, and can be "externalized" and transformed in knowledge only after a complex analysis (if it is ever possible).

III. THE SYMBOLIC AGENTS

First of all, while an old styled genetic agent behavior is based upon a genetic code represented by integer numerical values (or, often, just binary data), the genetic code of a symbolic agent is the *self−meaning* symbol [1]−[2]. This means that the material used as genetic code (that regulates the behavior of the agent) has a symbolic value that is meaningful for the agent itself. We can hypothesizes that the symbol carries an intrinsic meaning, known by all the interested agents in the process: the genetic agent, the controlling programs, the researchers and the *final user*.

Every genetic algorithm has an user[3]. If we can exclude the user form the analysis when the genetic is not symbolic, the knowledge generating symbolic genetic agent must have someone that receives the produced knowledge. Well, information becomes knowledge when it's shared across subjects, so there is no sense in saying that some knowledge is produced, if we don't put the user of that information material in the model.

So, the intrinsic meaning of the symbols used by the agent can be shared with the user, or can be useful only for the agent itself; in any case, there will be (simple) a way to translate the symbol in a human readable form, if the meaning carried by the symbol can’t be immediately received by the end user.

It is important to note that the mere presence of a self−meaning symbol sequence does not exclude the ability to create a meta−meaning for a group of related symbols. As an example, the word *blue* carries several self−meanings. Which of them will be used is determined by the context, which is a meta−meaning. I could say: "She has deep blue eyes", and "Today, she feels blue". The self−meaning of blue, in these two sentences is different because the meta−meaning (the context) is different.

Having this example translated in the symbolic genetic agent case, we can see that a context created by the agent can be more important than the sequence of symbols examined separately [4]. As the simple genetic algorithm has a complex behavior, that doesn’t directly depends on the single genes, but on their joint actions, so the symbolic genetic agents can create meta−meanings that are a combined result of the single self−meanings carried by the symbols.

IV. THE FUNCTION FINDER ALGORITHM

I will now illustrate how a working Function Finder, or a working symbolic genetic algorithm, can be obtained. I will assume this as an example of the potentials of this method.

The algorithm I developed is based upon a representation of mathematical symbols. Genetics agents have a genetic code that is composed by the following symbols:

- Variables (one for each variable of the vector X)
- Constants (each constant in the genetic code is different)
- Arithmetical signs (’+’, ’−’, ’*’, ’/’)
- Nth Power, or function powers: $f(x)=g(x)^{h(x)}$
- Natural logarithm and exponential
- Negation

Precedence of the symbols (parenthesis) depends on how the sequence is mounted, or, if you prefer, on the context (meta−meaning).

A. Startup

On startup, a standard population is created. First of all, we create some agents with codes meaning base interpolation models: linear, quadratic, geometrical, and so on. The genetic code of the firsts agents will signify:

$$k_0 x_0 + k_1 x_1 + \ldots + k_n x_n$$
$$k_0 x_0^2 + k_1 x_1^2 + \ldots + k_n x_n^2$$
$$k_0 e^{x_0} + k_1 e^{x_1} + \ldots + k_n e^{x_n}$$

... and so on.

To fill up a base population (30 to 50 agents are a good match) the remaining agents are created completely random, but in a way that only generates grammatically valid mathematical expressions.

B. Selection

Each turn, every agent is evaluated: the meaning of it’s genetic code is calculated (as a mathematical expression), feeding the agent with the X_i vector values. The result is compared with the corresponding y_i value. This is repeated with all the k X vectors and the k y results; the square distances between the evaluated results and the results are summed up; then the square root of this sum is divided by the mean of the results:

$$Fitness(g_j)=1-\frac{\sqrt{\sum_{i=0}^{k-1}(f(X_i)-y_i)^2}}{\bar{y}\cdot k}$$

In this way, every agent obtains a valuation of its *fitness* in deducting the observed results knowing independent variables, expressed as a proportional mean error. The lesser is the error, the better is the agent.

The world in which agents move in, (a mathematical space), hasn’t any random element; for this reason, survival of the wrong agents is limited to a small number of turns. In my experiments, I got the best results with survival of the worst elements limited to 0. I let only a part of the population to survive: the best 33% agents carries on to the next turns.

C. Reproduction and evolution

After the selection has been performed, here comes the reproduction. A cross over reproduction algorithm, with some degree of random mutation, has proven to lead to faster results in my experiments. Two random parts of genetic code, coming from two randomly picked up survived agents, are combined to form a new agent. More over, putting in randomly generated agent every generation (one to 10% of max. population), perturb the population enough to notice an increased capacity to exit from local optimum functions.

The process is started over, until an agent gets 0,00% of mean error or until an iteration number limit is hit. I found that 10.000 iterations are often enough to reach a good result.

The winning agent is the one that has the lower mean error.

D. Constants optimization

Since constants are one of the genotype used in generating the symbolic agents, particular care must be taken. The first version of the algorithm created *fixed* constants; an agent that carried a linear interpolation model would have been represented as $4x_1+12x_2+...$; casual mutation had the duty to optimize the functions. In other words, if an agent mutated a constant, and had a lesser error value, it would have survived.

The result has been the following: whole populations become simple variants of a given function. The population reproduced over and over the best agent found in a early stage of the process, varying the given constants.

To avoid this problem, it is necessary to bring constants to the rank of variables. When a new agent is born, the algorithm needs to optimize it's constants. From a starting random value, constants are brought to a best−fit value, using a linear optimization algorithm. This means that an agent carries a family of functions (determined by the attribution of a concrete value to the constants), and the best fitting one will "represent" the whole family.

The problem is: in a non−linear multidimensional worlds, there is no guarantee that a local optimum of a function may be the best global optimum. An agent could have been ill−optimized, so that it's mean error figures (let's say) of 10%, while a better optimization (a better allocation of concrete values to the constants) could result in a mean error of 1%.

I didn't deal directly with this problem, but used randomness to lesser it's consequences: each time an agent is derived, I give different random values to the constants. In that way, optimization *could* bring to different (hopefully better) results. After the optimization step, equivalent older agents with lower fitness values are removed.

A more robust algorithm should handle this problem in a more robust fashion.

V. FUNCTION GENERATION

Having to create random or mutated sequence of meaningful symbol, that must undergo a strict grammar rule set, require to handle genetic code generation in a special way. When the objective probability to generate a meaningless genetic code is low, it is possible to delegate the burden of evolving correctly formed agents to the fitness mechanism: agents having meaningless or useless genetic code won't survive. But when a complex grammar must be respected, and when the objective probability to create a meaningful random sequence is very low, as in the present case, the mutation mechanism must take care to never generate a meaningless genetic code.

Two methods have been developed; they are briefly discussed below.

A. The interpreter approach

In this approach, the symbols composing the genetic code are internally codified as a sequence of integer values; an *interpreter* transforms this sequence in a sequence of mathematical symbols. This can add a dimension of randomness which is not fully predictable in advance, since ambiguous sequences will be forced to be a rightful mathematic expression, by removing some unusable code, or by adding some closing symbol basing on an arbitrary de cision. More than this, a slight change in any part of the sequence could have a great impact on the resulting translated mathematical expression.

For example, it's difficult to say what exactly will be the result of a cross over mutation. A cross over between $a x_1 + b x_2$ with $\frac{x_1}{a}+\frac{x_2}{b}$ an intuitive result should be something like $\frac{x_1}{a}+b x_2$. But with the interference of the translator, you could get an odd result, like

$$\frac{x_1}{a}+\log x_0+b x_1$$

B. The symbol sequence approach

This method consists in using a genetic code that is a sequence of self meaning mathematical symbol, arranged following strict mathematical rules. The internal representation (strings containing readable functions, chains of operators, trees of operators, stacks etc.) is not important, since consequences are the same in all cases: the reproduction and the mutation algorithms must take care to intervene on the genetic code so that the transformed code is still a meaningful genetic sequence (under the provided rules, in this case grammatical rules for mathematical expressions).

The control leak seen in the examples in the section V.A is overwhelmed, but the algorithms used in reproduction and mutation are far more complex. As a side effect, it is possible to have really sporadic programming errors resulting from unhanded exceptions in the configuration of the genetic code to be mutated. A post−mutation control that ensures that the mutation algorithm does not any mistake is a costly overload in terms of calculation resources; and still some hard error can rise deep in the mutation algorithms when handling unexpected situation. The interpreter approach creates a correct sequence by hypothesis using random “directions” given by the genetic code; the symbol sequence approach has higher complexity wired in.

C. Symbolic optimization

The algorithm implements symbolic optimization, so that $\log e^{x_i}$ is transformed into x_i, and $-(-x_i)$ is translated into x_i. Furthermore $a x_i - (-b x_j)$ is translated into $a x_i + b x_j$, and $(x_i^a)^b$ will be transformed into x_i^{ab}. Some other basic mathematical simplifications are provided

D. Agent filtering

The above optimization is useful since, without symbolic reduction, the algorithm tends to span equivalent agents through the population. In a way that resembles the constant optimization problem, when a good enough agent is found, the algorithm generates a series of mutated agents that has the same canonical form of the best one, but are written in different algebraic forms: ie. $x_i + x_j$, $x_i - (-x_j)$, $x_3 - (-\log e^{x_j})$ and so on.

Since the world in which the agents are moving (a given set of data) is deterministic, two equals or equivalent genotypes are only a reduction of variety of the population, that must be avoided at all costs. Before to let a newborn genetic agent in the population, the algorithm checks for the existence of an equal or equivalent agent; if there is already a "copy" of the newborn genotype in the population, the new agent is discarded.

More than this, if there is an agent that "fails" some evaluation with the X_k vector (having some impossible result for same value of X_k, i.e. $\sqrt{x_3} \; when \; x_3 < 0$), the agent is given a strong penalty that will surely bring it to the very bottom of the population error ranking, causing it’s death (provided that there are enough non−failing agents).

REFERENCES

[1] G. Zanarini, “Complessità, auto−organizzazione, significato” in A. Ardigò, G. Mazzoli, “Complessità e Ipercomplessità tra sociosistemica e cibernetica”, Milano, Franco Angeli, 1990.

[2] E. Morin, “La Methode I. La nature de la nature”, Paris, Seuil, 1977

[3] H. v. Foerster, “Observing system”, Seaside, Intersystem, 1984

[4] M. M. Walldrop, “Complexity, the emerging scienge at the edge of order and chaos”, New York, Touchstone, 1993.