NUANCE v3.0: A User's Guide

Geoff Hollis & Chris Westbury
Department of Psychology, University of Alberta
P220 Biological Sciences Building
T6G 2E9 Edmonton, Alberta, Canada

Draft of November 9, 2005

Abstract

This manual explains how to use the Naturalistic University of Alberta Nonlinear Correlation Explorer (NUANCE) version 3.0. It describes the various parameters that NUANCE has, and gives examples of how to use the program effectively. Comments, suggestions, or queries for clarification should be addressed to Geoff Hollis at hollis @ ualberta dot ca.

Contents

Introduction

The first public release of NUANCE was version 2.0 (Hollis & Westbury, in press). Since then, we have made significant progress in improving the functionality and interface for the program. This document serves as a reference manual to the program, explaining its various parameters and modes of running. For users uninterested in dealing with the program parameters themselves, a basic mode is provided. This mode uses heuristics to pick program parameters for the user, greatly simplifying the setup process. A reference for the basic mode is provided at the end of this manual. The bulk of what is explained below concerns the advanced mode of running. We will explain the purpose of each parameter that can be adjusted in the advanced mode, and how its manipulation affects the performance of NUANCE. It is assumed that users already have a basic understanding of Genetic Programming (Koza, 1992; Hollis & Westbury, in press; Westbury, Buchanan, Anderson, Rhemtulla, & Phillips, 2003, for introductions). Questions or comments about the program should be directed to hollis @ ualberta dot ca.

Parameters

NUANCE utilizes a large number of parameters that allow users to optimize their ability to find interesting relationships within datasets. These parameters loosely fall into one of four categories: search breadth, function evaluation, stopping conditions, and miscellany. Within this manual, NUANCE's parameters will be organized by and discussed in the context of these headings.

Search Breadth

Population Size. Genetic Programming is a multi-agent search process. It traverses a search space by creating a group of agents, evaluating their position in the search space, taking the ones with the best position, mating them, and repeating this process until a stopping condition is met. The size of a population represents how many agents can exist in the search space at a time. The larger the population size, the more territory in the search space the agents will cover. Broader search typically entails better solutions. However, a larger population size also means slower search. There is no hard and fast rule for choosing an ideal population size. However, convention is to use a population size of 500-1000 for tough problems. NUANCE can handle much larger populations if they are needed; the limiting factors are the amount of available RAM and the amount of time one is willing to devote to the computation.

Runs. The time between seeding an initial population and meeting a stopping condition is called a run. During a run, a population of agents will typically begin converging to a small area in the search space. The area a population converges to is not necessarily an optimal one. Performing multiple runs allows for a broader coverage of the search space because it allows a population to retry the problem after converging to a local optimum. By default, NUANCE is set to perform 10 runs. This is by no means an ideal value for exploring a complex search space. Some research employing GP will use as many as 50 runs (Keijzer, 2003; Langdon & Poli, 1997, for instance). However, combined with other default parameters, a 10-run exploration should be able to provide a reasonable coverage of

a search space and finish in a reasonable amount of time (under 5 minutes) on most modern computers.

Mutation Rate. New agents are created by copying and randomly swapping subtrees between two parents. Through this process, fit subtrees will propagate throughout the population. Eventually, the population will reach a state in which all of the agents within it share very similar properties (i.e. it has converged). Sometimes, a population will converge on a suboptimal solution. However, if all of the agents within that population share a very similar genetic makeup, it is difficult for the population as a whole to come up with better solutions. Mutation (i.e. addition/deletion of random subtrees after mating) can help address this problem. However, it may be that mutation is completely superfluous. Experiments have demonstrated that, at least in some situations, mutation does not change the outcome of a run in any way (Koza, 1992). As such, the default mutation rate for NUANCE is 0%. This value can range between 0% and 100%, and reflects how likely it is that a new agent will be mutated after being created. A mutation will swap one random subtree in an agent with a different randomly-generated subtree of depth 3.

Selection Strategy. The selection strategy decides who survives at the end of each generation, and who dies. NUANCE supports two selection strategies: greedy overselection, and median pass selection. In greedy overselection, the agents who account for the top 20% of the population's total fitness have an 80% chance of survival, and the remaining 80% of the agents have a 20% chance of survival. In median pass selection, the top N agents accounting for 50% of the population's total fitness survive, and the rest of the population dies. Median pass is extremely conservative when compared to greedy overselection, which is much more exploratory. Greedy overselection generally finds better solutions, but typically at the cost of those solutions being longer and more convoluted (Hollis & Westbury, in press). However, wise use of parsimony pressure can counteract the size problem of solutions generated with greedy overselection. Thus, it is suggested that greedy overselection be used in all but a few specialized cases that will be explained below in the context of discussing age weighting.

Function Evaluation

Fitness Function. The fitness function evaluates how good any agent is on the problem it is trying to solve: e.g. how good it is at predicting the target values from the input set of predictors. NUANCE currently employs the use of two fitness functions: R^2 and percent hit. The agent's predicted values are compared to the target values with one of these three comparison methods. The percent hit fitness function is designed to be used when trying to predict numerical categorical target variables. The R^2 fitness function is designed to be used with any other type of target variable. In most regression problems, it has been formally demonstrated that the R^2 fitness function is one of the most useful fitness functions available to GP users (Keijzer, 2003, 2004). When predicting numerical categorical data, R^2 should be expected to perform better than percent hit by virtue of the fact that it only needs to find trends, not exact values. However, one should also expect the solutions to be harder to interpret, since exact values are not provided as output. Using percent hit versus R^2 for problems with categorical target variables is a tradeoff between interpretability and performance.

Parsimony Pressure. It is not uncommon for agents to grow very large over generations. This phenomenon is typically called *bloat*. Bloat both slows down evolutionary runs and makes it extremely difficult to understand evolved functions. One common method for addressing problems of bloat is to add a parsimony pressure to the fitness function. A parsimony pressure will penalize the fitness of an agent, as a function of how large that agent is. Even miniscule parsimony pressures can reduce average function length by as much as 300%, without degrading fitness (Soule & Foster, 1998). Parsimony values can range between 0% and 100%. By default, NUANCE uses a parsimony value of 0.02% - for every node in an agent, the agent's fitness is decreased by 0.02%.

Age Weight. In the manual for NUANCE v2.0 (Hollis & Westbury, in press), we detailed a parameter called age weight. When working with incomplete or noisy datasets, we often want to avoid over-fitting. One way to go about doing this is to only expose agents to a small subset of the dataset on each generation. However, when working with small datasets, it becomes difficult to expose agents to small subsets without having wildly erratic fluctuations in fitness. We also want to protect genuinely fit agents from being rated poorly simply because they happen to be exposed to unrepresentative subsets of the problem at hand. To improve the performance of the fitness function, we have introduced age weighting. Age weighting changes the way fitness is calculated so that it is not simply a measure of the agent's performance on the current subset but, instead, a weighted sum of performance on all problems it has ever been exposed to. Age weight can range between 0 and 1. An agent's fitness is calculated as follows:

$$f_n \quad = \quad \begin{cases} p_n & \text{if this was the generation the agent was born} \\ (1-\alpha)p_n + \alpha f_{n-1} & \text{otherwise} \end{cases}$$

Where f_n is the agent's fitness on generation n, p_n is the performance on the subset exposed to during generation n, and α is the age weight. If an age weight value is to act as a buffer for good functions against unrepresentative subsets, the age weight value should be less than or equal to 0.5.

Subset Size. Exposing agents to small subsets of the entire problem on each generation is one way to prevent overfitting. The subset size parameter controls the size of the subsets to which each agent is exposed on each generation. The value of subset size can be any non-zero percent value, representing the percent of the total input set that agents are exposed to on each generation. If the value is less than 100%, each subset is composed by randomly drawing distinct elements from the total problem until a subset of the desired size is created. The smaller a subset, the less representative of the total problem it may be. For reliability, subsets with a size less than 30 are not suggested for use with the R^2 fitness function.

Interval Arithmetic. Except for the limits imposed by the operator set, there is typically no restriction on the shape of functions that can be evolved by NUANCE. When working with an incomplete dataset, this means NUANCE could output functions that had undefined values in ranges along the range of possible inputs, but which happen not to be present in the dataset equations are evolved on, because of division by zero, logging negative numbers, and other such undefined values. At the best of times, this simply creates equations that are harder to understand. At the worst of times, undefinedness across a range

actually impedes NUANCE's capabilities to evolve agents with good fitness by making the search space harder to explore. Turning interval arithmetic on ensures that NUANCE will not create equations with undefined behavior anywhere in the range of the input variables. It has a cost of additional computational power, often making runs take 50% longer to complete. However, the benefits may be worthwhile, especially if you intend on trying to interpret equations provided by NUANCE. An in-depth introduction to the benefits of interval arithmetic can be found in (Keijzer, 2003).

Stopping Conditions

Generations. Runs have to terminate eventually. A common way of doing this is to put a cap on the maximum number of generations per run. If the value of this parameter is above zero, a run will terminate after the specified number of generations have elapsed. With of a value of zero, there will be no cap on the maximum number of generations per run.

Generations Without Change. This stopping condition will end a run if a specified number of generations have elapsed without NUANCE improving its best solution of that run. A value of zero will disable this stopping condition. This parameter is particularly useful for identifying and terminating runs where NUANCE has had a poor start. By default, NUANCE will terminate after 20 generations have elapsed with no progress. However, it should be noted that some problems exist where NUANCE will often not improve for 50 generations and then suddenly have an "insightful leap", increasing the fitness of its best equation by over 1000% percent (Hollis & Westbury, in press). Thus, this parameter should be treated with extreme care when the goal is a thorough coverage of search space.

Miscellany

Minimum Constant and Maximum Constants. NUANCE has two types of terminal nodes it uses in functions: variables and constants. Constants are static numeric values. Depending on the problem, certain ranges of constants are more appropriate than others. Users can specify the range of constants they would like to use by changing the minimum and maximum constant parameters. By default, the value of these constants are 0 and 1, respectively. Through division, values in this range can emulate any positive real number. Thus, they are ideal default values.

Constant Type. Depending on the problem, it may be more appropriate to use constants pulled from the set of Integer or Real Numbers. This parameter allows users to choose which set of numbers they would like constants to be generated from. When a constant is needed, it is pulled randomly from a uniform distribution over the range of allowable constants.

Running Mode. NUANCE was designed to regress a set of predictors on a single target variable. However, some situations arise when users might want to use NUANCE in different ways. For instance, NUANCE has been used to map out the pattern of interactions between 15 factors thought to be of interest to lexical decision tasks (Hollis, Westbury, & Peterson, submitted). Such tasks require NUANCE to be run in a fundamentally different manner. By default, NUANCE has three run modes: normal, singleton, and pairwise. In

normal mode, NUANCE can use all of the predictors in the input set to estimate the target variable. In singleton mode, NUANCE takes each of the predictors alone and uses them to predict the target variable. In pairwise mode, NUANCE systematically takes each pairwise combination of the predictors and uses them to predict the target variable. Pairwise and singleton mode are remnants from the work done in Hollis et al. (submitted), but have been left as illustrative examples of how to program and add new run modes. They may also be of use to individuals interested in the type of exploratory work carried out in Hollis et al. (submitted).

Host Service. Genetic Programming is easily amenable to parallelization. One common approach to parallelizing a Genetic Programming system is the island model approach. In the island model approach multiple, semi-independent populations evolve alongside each other. They are only semi-independent, because occasional migration between populations occurs. It has been noted that keeping smaller, semi-independent populations typically produces better solutions than one large population (Andre & Koza, 1996).

NUANCE parallelizes genetic programming used the island model approach. When this option is enabled, NUANCE will act as a server and wait for incoming connections from other NUANCEs. When a connection is established, the settings chosen on the server NUANCE will be sent out to the client, which will run its own independent population that will occasionally send or receive emigrants from other clients connected to the service. Currently, service hosting only works in *normal* mode.

To host a service, port 7159 must be available for use.

To connect to a running service, users enter client setup. The DNS or IP of the computer running the service must then be entered. If the service is running, the client will download rules for running set out by the server, and begin running.

Data Sets

Predictors and target variables must be loaded in from a comma- tab- or space-separated-variable file. The first line in a data set must be a header containing the variable names, separated by one of the proper delimeters. NUANCE will convert all empty cell values (represented as . or n/a) to their column average. Other non-numeric values are assumed to be categories.

Operators

Basic Operators

NUANCE takes a set of symbols and applies transformations to them in an attempt to predict some target value as accurately as possible. Table 1 provides a listing of the operators that come with NUANCE v3.0, and a mathematical description of what they do.

Adding New Operators

NUANCE v2.0 allowed users to define custom operators for use during evolution. This was useful in circumstances where users knew complex operations needed to be performed to get a good fit. However, it was limited in that only new operators that could be created out of NUANCE's basic operators were allowabke. NUANCE v3.0 is open source. This

allows users to write their own operators. Because users now have access to the source code, they have much more flexibility over what their custom operators can do. Custom operators will also evaluate much faster. Creating a new operator is easy, but requires basic knowledge of the Java programming language. Instructions on adding a new operator are given below. However, the instructions use Java jargon, and thus readers are expected to know the basics of Java programming before they attempt to add their own operators:

- A new class for it must be created in the src/agent/s_expression package. The class must implement the Symbol interface. Much of the legwork for this is already done in the BasicSymbol class, so new symbols can extend this rather than implementing the Symbol interface. It is assumed this second approach is taken.
- Write code for the copy(), type(), eval(Problem), and range(Problem) methods that your new operator requires. Documentation on how these methods should work is provided in the src/agent/s_expression/Symbol.java interface.
- Write code for any constructors the new symbol might require. A constructor taking no arguments is required, and it should set all children in the symbol to null.
- In src/agent/s_expression/SymbolicFactory.java, a new entry for the symbol must be added into the all_ops array. It must also be given a unique name (usually, this is the same value returned by the symbol's type() method).
- An entry for this class should be added to the src/agent/s_expression/package.mk makefile so the main makefile knows to compile the new class.
- In the src directory, run *make clean*, and then *make jar* to create a new jar file.

Setup

There are multiple ways a person can set NUANCE up to run. The major setup modes are *basic* and *advanced*. However, there are also two additional modes: *client* and *text*. Each mode will be discussed below in turn, and examples of how to run in each mode are given.

Basic Setup

Basic setup was designed with the casual user in mind. It requires no knowledge of NUANCE's many parameters. Users simply choose their data file, choose the variable they would like to estimate, choose where they would like to log output to, say how many times they would like to run the program, and then run. NUANCE will choose what it thinks the most appropriate settings are to run with the given data file. Basic setup is extremely easy to use and requires little planning to run, but it also lacks control over the more advanced features described above. For instance, basic setup will always start nuance running in normal mode. Custom modes cannot be accessed within the basic setup. Services cannot be hosted when using basic setup. Operators and predictors cannot be chosen in basic setup. Access to these features requires users to enter the advanced setup. However, these are features that the casual user is probably not interested in.

Advanced Setup

Advanced setup is the interface most users of NUANCE will want to become familiar with. It allows full control over every facet of NUANCE. Upon entering the advanced setup,

users are presented a screen full of spinners and option lists with which they can customize how NUANCE will run. All of the options presented in advanced setup are discussed in the *Parameters* section of this manual.

Client Setup

In the advanced setup there is an option to host a service, which is described above in the *Parameters* section of this manual.

Text Setup

Circumstances arise where running NUANCE with a graphical user interface (GUI) is undesirable or impossible. The NUANCE GUI requires a substantial amount of memory (appx. 16mb of RAM), and thus may prevent older computers from effectively utilizing NUANCE. People who wish to execute NUANCE remotely may be completely incapable of interacting with a GUI. To accomodate for such situations, NUANCE has been supplied with a text mode.

To invocate the NUANCE text mode, NUANCE must be started from a terminal window and be supplied with the -textmode runtime parameter (e.g. java -jar nuance.jar -textmode). NUANCE will boot up in a mode analagous to the advanced setup, except it will read in parameter values from the file located at *$NUANCE_ROOT/config/advanced_config.txt*. Each parameter and its value must be listed on a new line, and separated by a colon. All output normally displayed to a run window in graphical mode will be sent to the terminal's standard output.

It is also possible to run a client process in text mode. This is done by supplying the -c option at runtime, followed by the DNS or IP of the server (e.g. *java -jar nuance.jar -c nuance.psych.ualberta.ca*).

Output

After NUANCE has completed its runs, a log is printed to a folder chosen by the user. The log folder is chosen during setup. In this folder will be a file giving a detailed report of the best function found, and another file that contains all of the configuration options for the run (advanced mode only). Also contained within the log folder will be a subfolder for each run completed. In the subfolder will be a history file for everything that was displayed to the screen. Also, the subfolder will contain a summary file. In the summary file, a generation-by-generation summary of the run will be printed. For each generation, information on the amount of time it took, the fitness of the best function, and the best function to date will be listed.

References

Andre, D., & Koza, J. R. (1996). Parallel genetic programming: A scalable implementation using the transputer network architecture. In P. J. Angeline & K. E. Kinnear, Jr. (Eds.), *Advances in genetic programming 2* (pp. 317–338). Cambridge, MA, USA: MIT Press.

Hollis, G., & Westbury, C.(in press). Nuance: Naturalistic university of alberta nonlinear correlation explorer. *Behavioral Research Methods. Behavioral Research Methods.* in press.

Hollis, G., Westbury, C., & Peterson, J. (submitted). Nuance 3.0: Using genetic programming to model variable relationships. *Behavioral Research Methods. Behavioral Research Methods.* submitted.

Keijzer, M. (2003, 14-16 April). Improving symbolic regression with interval arithmetic and linear scaling. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, & E. Costa (Eds.), *Genetic programming, proceedings of eurogp'2003* (Vol. 2610, pp. 70–82). Essex: Springer-Verlag.

Keijzer, M. (2004, September). Scaled symbolic regression. *Genetic Programming and Evolvable Machines*, *5*(3), 259–269.

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

Langdon, W. B., & Poli, R. (1997, 24 February). *Fitness causes bloat* (Tech. Rep. No. CSRP-97-09). Birmingham, B15 2TT, UK: University of Birmingham, School of Computer Science.

Soule, T., & Foster, J. A. (1998, Winter). Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, *6*(4), 293–309.

Westbury, C., Buchanan, L., Anderson, M., Rhemtulla, M., & Phillips, L. (2003). Using genetic programming to discover nonlinear variable interactions. *Behavioral Research Methods, Instruments, and Computers*, *28*, 202–216.

Table 1: Descriptions of the operators available in the basic release of NUANCE v3.0. Operators can take any number of arguments. In the mathematical description of the operators, these arguments are referred to in the form $x_0, x_1, x_2, ..., x_n$

Operator	Arguments	Description
Addition	2	$x_0 + x_1$
Subtraction	2	$x_0 - x_1$
Division	2	$\frac{x_0}{x_1}$
Multiplication	2	$x_0 x_1$
Square Root	1	$\sqrt{x_0}$
Cube Root	1	$\sqrt[3]{x_0}$
Square	1	x_0^2
Cube	1	x_0^3
Natural Logarithm	1	$\ln x_0$
Antiln	1	e^{x_0}
Absolute Value	1	$\lvert x_0 \rvert$
Round	1	$floor(x_0 + 0.5)$
Sine	1	$\sin x_0$
Cosine	1	$\cos x_0$
Tangent	1	$\tan x_0$
Less Than	4	$\begin{cases} x_2 & x_0 < x_1 \\ x_3 & \text{otherwise} \end{cases}$
Equals	4	$\begin{cases} x_2 & x_0 = x_1 \\ x_3 & \text{otherwise} \end{cases}$
Or	2	$\begin{cases} 1 & \lvert x_0 \rvert + \lvert x_1 \rvert \neq 0 \\ 0 & \text{otherwise} \end{cases}$
And	2	$\begin{cases} 1 & x_0 x_1 \neq 0 \\ 0 & \text{otherwise} \end{cases}$
True	0	1
False	0	0