Evolution of a Computer Program for Classifying Protein Segments as Transmembrane Domains Using Genetic Programming

John R. Koza

Computer Science Department
Stanford University
Stanford, CA 94305-2140 USA
Koza@CS.Stanford.Edu
PHONE 415-941-0336

Abstract

The recently-developed genetic programming paradigm is used to evolve a computer program to classify a given protein segment as being a transmembrane domain or non-transmembrane area of the protein. Genetic programming starts with a primordial ooze of randomly generated computer programs composed of available programmatic ingredients and then genetically breeds the population of programs using the Darwinian principle of survival of the fittest and an analog of the naturally occurring genetic operation of crossover (sexual recombination). Automatic function definition enables genetic programming to dynamically create subroutines dynamically during the run. Genetic programming is given a training set of differently-sized protein segments and their correct classification (but no biochemical knowledge, such as hydrophobicity values). Correlation is used as the fitness measure to drive the evolutionary process. The best genetically-evolved program achieves an out-of-sample correlation of 0.968 and an out-of-sample error rate of 1.6%. This error rate is better than that reported for four other algorithms reported at the First International Conference on Intelligent Systems for Molecular Biology. Our genetically evolved program is an instance of an algorithm discovered by an automated learning paradigm that is superior to that written by human investigators.

Introduction

At the First International Conference on Intelligent Systems for Molecular Biology, Weiss, Cohen, and Indurkhya (1993) explored the problem of identifying transmembrane domains in protein sequences. Starting with knowledge about the Kyte-Doolittle hydrophobicity scale (Kyte and Doolittle 1982), they used the SWAP-1 induction technique to discover an algorithm for this classification task. In their first experiment, they equaled the error rate of the best of three human-written algorithms for this classification task.

Genetic programming is a domain-independent method for evolving computer programs that solve, or approximately solve, problems. To accomplish this, genetic programming starts with a primordial ooze of randomly generated computer programs composed of the available programmatic ingredients, and breeds the population or programs using the Darwinian principle of survival of the fittest and an analog of the naturally occurring genetic operation of crossover (sexual recombination). Automatic function definition enables genetic programming to dynamically create subroutines dynamically during the run.

The question arises as to whether genetic programming can evolve a classifying program consisting of initially *unspecified* detectors, an initially *unspecified* iterative calculation incorporating the as-yet-undiscovered detectors, and an initially *unspecified* final calculation incorporating the results of the as-yet-undiscovered iteration. The genetically evolved program in this paper accomplishes this. It achieves a better error rate than all four algorithms described in Weiss, Cohen, and Indurkhya (1993). When analyzed, the genetically evolved program has a simple biological interpretation.

Transmembrane Domains in Proteins

Proteins are polypeptide molecules composed of sequences of amino acids. There are 20 amino acids (also called residues) in the alphabet of proteins. They are denoted by the letters A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, and Y. Broadly speaking, the sequence of amino acids in a protein determines the locations of its atoms in three-dimensional space; this, in turn, determines the biological structure and function of a protein (Anfinsen 1973).

A transmembrane protein is a protein that finds itself embedded in a membrane (e.g., a cell wall) in such a way that part of the protein is located on one side of the membrane, part is within the membrane, and part is on the opposite side of the membrane. Transmembrane proteins often cross back and forth through the membrane several times and have short loops immersed in the different milieu on each side of the membrane. The length of each transmembrane domain and each loop or other non-transmembrane area are usually different. Transmembrane proteins perform functions such as sensing the presence of certain chemicals or certain stimuli on one side of the membrane and transporting chemicals or transmitting signals to the other side of the membrane. Understanding the behavior of transmembrane proteins requires identification of their transmembrane domains.

Biological membranes are of hydrophobic (water-hating) composition. The amino acids in the transmembrane domain of a protein that are exposed to the membrane therefore have a pronounced tendency to be hydrophobic. This tendency toward hydrophobicity is an overall distributional characteristic of the entire protein segment (not of any particular one or two amino acids of the segment). Many transmembrane domains are α-helices, so all the residues of the helix are exposed to the membrane (and are therefore predominantly hydrophobic). Although some transmembrane domains are β-strands (so that only some residues that are actually exposed to the membrane), very few such transmembrane domains are annotated in the computerized databases. Thus, as a practical matter, our discussion here is limited to α-helical transmembrane domains.

Consider, for example, the 161-residue *mouse peripheral myelin protein 22* (identified by the locus name "PM22_MOUSE" in release 27 of the SWISS-PROT computerized database of proteins (Bairoch and Boeckmann 1991). The four transmembrane domains of this protein are located at residues 2–31, 65–91, 96–119, and 134–156.

A successful classifying program should identify a segment such as the following 24-residue segment from positions 96–119:

FYITGFFQILAGLCVMSAAAIYTV, (1)

as a transmembrane domain.

A successful classifying program should also identify the following 27-residue segment between positions 35–61:

TTDLWQNCTTSALGAVQHCYSSSVSEW (2)

as being in a non-transmembrane area of the protein.

This classification problem will be solved by genetic programming without reference to any knowledge about the hydrophobicity of the 20 amino acids; however, we will use such knowledge to explain the problem (and, later, to interpret the genetically evolved program). Two thirds of the 24 residues of segment (1) are in the category consisting of I, V, L, F, C, M, or A having the highest numerical values of hydrophobicity on Kyte-Doolittle scale. If a human were clustering the 20 hydrophobicity values into three categories with the benefit of knowledge of the Kyte-Doolittle hydrophobicity scale, these seven residues would be categorized into a hydrophobic category. Seven of the 24 residues of segment (1) (i.e., two Gs, two Ts, two Ys, and one S) are in the category consisting of G, T, S, W, Y, P (which the knowledgeable human would cluster into a neutral category). Only one residue of segment (1) (i.e., the Q at position 103) is in the category consisting of H, Q, N, E, D, K, R (which the knowledgeable human would cluster into a hydrophilic category). Even through there are some residues from all three categories in segments(1), segment (1) is predominantly hydrophobic and is, in fact, a transmembrane domain of PM22_MOUSE.

In contrast, 13 of the 27 (about half) of the residues of segment (2) are neutral, eight (about a quarter) are hydrophobic, and six (about a quarter) are hydrophilic. This distribution is very different from that of segment (1). Segment (2) is, in fact, a non-transmembrane area of PM22_MOUSE.

Background on Genetic Programming

John Holland's pioneering 1975 *Adaptation in Natural and Artificial Systems* described how the evolutionary process in nature can be applied to artificial systems using the genetic algorithm operating on fixed length character strings (Holland 1975, 1992). Additional information on current work in genetic algorithms can be found in Goldberg (1989), Forrest (1993), Davis (1987, 1993), and Michalewicz (1992).

Genetic programming is an extension of the genetic algorithm in which the genetic population consists of computer programs (that is, compositions of primitive functions and terminals). As described in *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992), genetic programming is a domain independent method that genetically breeds populations of computer programs to solve problems by executing the following three steps:

(1) Generate an initial population of random computer programs composed of the primitive functions and terminals of the problem.

(2) Iteratively perform the following sub-steps until the termination criterion has been satisfied:

(a) Execute each program in the population and assign it a fitness value according to how well it solves the problem.

(b) Create a new population of programs by applying the following two primary operations. The operations are applied to program(s) in the population selected with a probability based on fitness (i.e., the fitter the program, the more likely it is to be selected).

(i) *Reproduction*: Copy an existing program to the new population.

(ii) *Crossover*: Create two new offspring programs for the new population by genetically recombining randomly chosen parts of two existing programs. The genetic crossover (sexual recombination) operation (described below) operates on two parental computer programs and produces two offspring programs using parts of each parent.

(3) The single best computer program in the population produced during the run is designated as the result of the run of genetic programming. This result may be a solution (or approximate solution) to the problem.

Recent advances in genetic programming are described in Kinnear (1994). A videotape visualization of numerous applications of genetic programming can be found in Koza and Rice (1992) and Koza (1994).

The genetic crossover operation operates on two parental computer programs selected with a probability based on fitness and produces two new offspring programs consisting of parts of each parent.

For example, consider the following computer program (shown here as a LISP symbolic expression):

```
(+ (* 0.234 Z) (- X 0.789)).
```

We would ordinarily write this LISP S-expression as $0.234z + x - 0.789$. This two-input, one-output computer program takes X and Z as inputs and produces a single floating point output.

Also, consider a second program:

```
(* (* Z Y) (+ Y (* 0.314 Z))).
```

This program is equivalent to $zy(y + 0.314z)$.

The crossover operation creates new offspring by exchanging sub-trees (i.e., subroutines, sublists, subprocedures, subfunctions) between the two parents. The two parents are typically of different sizes and shapes. The sub-trees to be exchanged (called crossover fragments) are selected at random by selecting crossover points at random. Suppose that crossover points are the multiplication (*) in the first parent and the addition (+) in the second parent. The two crossover fragments are the underlined sub-programs (sub-lists) in the two parental LISP S-expressions.

The two offspring resulting from crossover are

```
(+ (+ Y (* 0.314 Z)) (- X 0.789))
```

and

```
(* (* Z Y) (* 0.234 Z)).
```

Assuming closure among the functions and terminals of which the parental programs are composed, crossover produces syntactically and semantically valid programs as offspring. Because programs are selected to participate in the crossover operation with a probability based on their fitness, crossover allocates future trials of the search for a solution to the problem to regions of the space of possible computer programs containing programs with promising parts.

Automatic function definition is used to enable genetic programming to evolve subroutines during a run. Automatic function definition can be implemented within the context of genetic programming by establishing a constrained syntactic structure for the individual programs in the population as described in *Genetic Programming II: Scalable Automatic Programming by Means of Automatically Defined Functions* (Koza 1994). Each program in the population contains one (or more) function-defining branches, one main result-producing branch, and possibly other types of branches (such as iteration-performing branches). The function-defining branch(es) define the automatically defined functions ADF0, ADF1, etc. The result-producing branch may invoke the automatically defined functions. The value returned by the overall program consists of the value returned by the result-producing branch.

The initial random generation of the population (generation 0) is created so that every individual program in the population has a constrained syntactic structure consisting of the problem's particular arrangement of branches. Each branch is composed of functions and terminals appropriate to that branch. This constrained syntactic structure must be preserved as the run proceeds from generation to generation. Structure-preserving crossover is implemented by limiting crossover to points lying within the bodies of the various branches (branch typing). The crossover point for the first parent is randomly selected, without restriction, from the body of any one of the branches. However, once this selection is made for the first parent, the crossover point of the second parent is randomly selected from the body from the same type of branch. This method of performing crossover preserves the syntactic validity of all offspring throughout the run. As the run progresses, genetic programming will evolve different function-defining branches, different result-producing branches, and different ways of calling these automatically defined functions from the result-producing branch.

Preparatory Steps

In applying genetic programming with automatic function definition to a problem, there are six major preparatory steps. These steps involve determining

(1) the set of terminals for each branch,
(2) the set of functions for each branch,
(3) the fitness measure,
(4) the parameters and variables for controlling the run,
(5) the criterion for designating a result and terminating a run, and
(6) the architecture of the overall program.

We begin by deciding that the overall architecture of the yet-to-be-evolved classifying program will have to be capable of categorizing the residues into useful categories, then iteratively performing some arithmetic calculations and conditional operations on the categories, and finally performing some arithmetic calculations and conditional operations to reach a conclusion. This suggests an overall architecture for the classifying program of several automatically defined functions (say ADF0, ADF1, ADF2) to serve as detectors for categorization, an iteration-performing branch, IPB0, for performing arithmetic operations and conditional operations for examining the residues of the protein segment using the as-yet-undiscovered detectors, and a result-producing branch, RPB0, for performing arithmetic operations and conditional operations for reaching a conclusion using the as-yet-undiscovered iteration.

Automatically defined functions seem well suited to the role of dynamically defining categories of the amino acids. If the automatically defined functions are to play the role of set formation, each defined function should be able to interrogate the current residue as to which of the 20 amino acids it is. Since we anticipate that some numerical calculations will subsequently be performed on the result of the categorization of the residues, we employ numerical-valued logic, rather than Boolean-valued logic returning the non-numerical values of True and False. One way to implement this approach is to define 20 numerical-valued zero-argument logical functions for determining whether the residue currently being examined is a particular amino acid. For example, (A?) is the zero-argument residue-detecting function returning a numerical +1 if the current residue is alanine (A) but otherwise returning a numerical –1. A similar residue-detecting function is defined for each of the 19 other amino acids. Since we envisage that the automatically defined functions will be used for set formation, it seems reasonable to include the logical disjunctive function in the function set of the automatically defined functions. Specifically, ORN is the two-argument numerical-valued disjunctive function returning +1 if either or both of its arguments are positive, but returning –1 otherwise.

The terminal set $\mathcal{T}_{\text{fd}}$ for each of the three function-defining branches (ADF0, ADF1, and ADF2) contains the 20 zero-argument numerical-valued residue-detecting functions. That is,

$\mathcal{T}_{\text{fd}}$ = {(A?), (C?), ... , (Y?)}.

The function set $\mathcal{F}_{\text{fd}}$ for the three function-defining branches (ADF0, ADF1, and ADF2) contains only the two-argument numerically-valued logical disjunctive function. That is,

$\mathcal{F}_{\text{fd}}$ = {ORN}.

Typical computer programs contain iterative operators that perform some specified work until some condition expressed by a termination predicate is satisfied. When we attempt to include iterative operators in genetically-evolved programs, we face the practical problem that both the work and the termination predicate are initially created at random and are subsequently subject to modification by the crossover operation. Consequently, iterative operators will, at best, be nested and consume enormous amounts of computer time or will, at worst, have unsatisfiable termination predicates and go into infinite loops. This problem can sometimes be partially alleviated by imposing arbitrary time-out limits (e.g., on each iterative loop individually and all iterative loops cumulatively).

In problems where we can envisage one iterative calculation being usefully performed over a particular known, finite set, there is an attractive alternative to permitting imposing arbitrary time-out limits. For such problems, the iteration can be restricted to exactly one iteration over the finite set. The termination predicate of the iteration is thereby fixed and is not subject to evolutionary modification. Thus, there is no nesting and there are no infinite loops.

In the case of problems involving the examination of the residues of a protein, iteration can very naturally be limited to the ordered set of amino acid residues of the protein segment involved. Thus, for this problem, we employ one iteration-performing branch, with the iteration restricted to the ordered set of amino acid residues in the protein segment. That is, each time iterative work is performed by the body of the iteration-performing branch, the current residue of the protein is advanced to the next residue of the protein segment until the end of the entire protein segment is encountered. The result-producing (wrap-up) branch produces the final output of the overall program.

Useful iterative calculations typically require both an iteration variable and memory (state). That is, the nature of the work performed by the body of the iteration-performing branch typically varies depending on the current value of the iteration variable. Memory is typically required to transmit information from one iteration to the next. In this problem, the same work is executed as many times as there are residues in a protein segment, so the iteration variable is the residue at the current position in the segment. Depending on the problem, the iteration variable may be explicitly available or be implicitly available through functions that permit it to be interrogated. For this problem, the automatically defined functions provide a way to interrogate the residues of the protein sequence.

Memory can be introduced into any program by means of settable variables, M0, M1, M2, and M3. Settable variables are initialized to some appropriate value (e.g., zero) at the beginning of the execution of the iteration-performing branch. These settable variables typically change as a result of each iteration.

The terminal set $\mathcal{T}_{\text{ipb0}}$ for the iteration-performing branch is

$\mathcal{T}_{\text{ipb0}}$ = {LEN, M0, M1, M2, M3, $\Re$}.

Here $\Re$ represents floating-point random constants between –10.000 and +10.000 with a granularity of 0.001 and LEN is the length of the current protein segment.

Since we envisage that the iteration-performing branch will perform numerical calculations and make decisions based on these calculations, it seems reasonable to include the four arithmetic operations and a conditional operator in the function set. We have used the four arithmetic functions (+, −, *, and %) and the conditional comparative operator IFLTE (If Less Than or Equal) on many previous problems, so we include them in the function set for the iteration-performing branch. The protected division function % takes two arguments and returns one when division by 0 is attempted (including 0 divided by 0), and, otherwise, returns the normal quotient. The four-argument conditional branching function IFLTE evaluates and returns its third argument if its first argument is less than or equal to its second argument and otherwise evaluates and returns its fourth argument.

Since a numerical calculation is to be performed on the results of the categorization performed by the function-defining branches, the functions ADF0, ADF1, and ADF2 are included in the function set for the iteration-performing branch.

We need a way to change the settable variables M0, M1, M2, and M3. The one-argument setting function SETM0 can be used to set M0 to a particular value. Similarly, the setting functions SETM1, SETM2, and SETM3 can be used to set the respective values of the settable variables M1, M2, and M3, respectively. Thus, memory can be written (i.e., the state can be set) with the setting functions, SETM0, SETM1, SETM2, and SETM3, and memory can be read (i.e., the state can be interrogated) merely by referring to the terminals, M0, M1, M2, and M3.

Thus, the function set $\mathcal{F}_{\text{ipb0}}$ for the iteration-performing branch, IPB0, is

$$\mathcal{F}_{\text{ipb0}} = \{\text{ADF0, ADF1, ADF2, SETM0, SETM1, SETM2, SETM3, IFLTE}, +, -, *, \%\}.$$

taking 0, 0, 0, 1, 1, 1, 1, 4, 2, 2, 2, and 2 arguments, respectively.

The result-producing (wrap-up) branch then performs a non-iterative floating-point calculation and produces the final result of the overall program. The settable variables M0, M1, M2, and M3 provide a way to pass the results of the iteration-performing branch to the result-producing branch.

The terminal set $\mathcal{T}_{\text{rpb0}}$ for the result-producing branch, RPB0, is

$$\mathcal{T}_{\text{rpb0}} = \{\text{LEN, M0, M1, M2, M3}, \Re\}.$$

The function set $\mathcal{F}_{\text{rpb0}}$ for the result-producing branch RPB0, is

$$\mathcal{F}_{\text{rpb0}} = \{\text{IFLTE}, +, -, *, \%\}$$

taking 4, 2, 2, 2, and 2 arguments, respectively.

A wrapper is used to convert the floating-point value produced by the result-producing branch into a binary outcome. If the genetically-evolved program returns a positive value, the segment will be classified as a transmembrane domain, but otherwise it will be classified as a non-transmembrane area.

Release 25 of the SWISS-PROT protein data base contains 248 mouse transmembrane proteins averaging 499.8 residues in length. Each protein contains between one and 12 transmembrane domains, the average being 2.4. The transmembrane domains range in length from 15 and 101 residues and average 23.0 in length.

123 of the 248 proteins were arbitrarily selected to create the in-sample set of fitness cases to measure fitness during the evolutionary process. One of the transmembrane domains of each of these 123 proteins was selected at random as a positive fitness case for this in-sample set. One segment of the same length as a random one of the transmembrane segments that is not contained in any of the protein's transmembrane domains was selected from each protein as a negative fitness case. Thus, there are 123 positive and 123 negative fitness cases in the in-sample set of fitness cases.

The evolutionary process is driven by fitness as measured by the set of in-sample fitness cases. However, the true measure of performance for a classifying program is how well it generalizes to different cases from the same problem environment. Thus, 250 out-of-sample fitness cases (125 positive and 125 negative) were created from the remaining 125 proteins in a manner similar to the above. These out-of-sample fitness cases were then used to validate the performance of the genetically-evolved classifying programs.

Fitness will measure how well a particular genetically-evolved classifying program predicts whether the segment is, or is not, transmembrane domain. Fitness is measured over a number of trials, which we call fitness cases. The fitness cases for this problem consist of protein segments.

When a genetically-evolved classifying program in the population is tested against a particular fitness case, the outcome can be a true-positive, true-negative, false-positive, or false-negative. Fitness can be measured by the correlation coefficient C. When the predictions and observations each take on only two possible values, correlation is a general, and easily computed, measure for evaluating the performance of a classifying program. Consider a vector in a space of dimensionality N_{fc} of the correct answers (with the integer 1 representing a transmembrane domain and the integer 0 representing a non-transmembrane area) and the vector of length N_{fc} of the predictions (1 or 0) produced by a particular genetically evolved program. Suppose each vector is transformed into a zero-mean vector by subtracting the mean value of all of its components from each of its components. The correlation, C, is the cosine of the angle in this space of dimensionality N_{fc} between the zero-mean vector of correct answers and the zero-mean vector of predictions. The correlation coefficient indicates how much better a particular predictor is than a random predictor. A correlation C of −1.0 indicates vectors pointing in opposite directions in N_{fc}-space (i.e., greatest negative correlation); a correlation of +1.0 indicates coincident vectors (i.e., greatest positive correlation); a correlation C of 0.0 indicates orthogonal vectors (i.e., no correlation).

The correlation, C, lends itself immediately to being the measure of raw fitness measure for a genetically evolved computer program. Since raw fitness ranges between –1.0 and +1.0 (higher values being better), standardized fitness ("zero is best") can then be defined as $\frac{1-C}{2}$. Standardized fitness ranges between 0.0 and +1.0, lower values being better and a value of 0 being the best. Thus, a standardized fitness of 0 indicates perfect agreement between the predicting program and the observed reality; a standardized fitness of +1.0 indicates perfect disagreement; a standardized fitness of 0.50 indicates that the predictor is no better than random.

The *error rate* is the number of fitness cases for which the classifying program is incorrect divided by the total number of fitness cases. The error rate is a less general measure of performance for a classifying program; however, Weiss, Cohen, and Indurkhya (1993) use the error rate as their yardstick for comparing three methods in the biological literature with their new algorithm created using the SWAP-1 induction technique. Therefore, we present our final results in terms of both correlation and error rate and we use error rate for the purpose of comparing results.

Population size, M, was 4,000. The maximum number of generations to be run, G, was set to 21. The other parameters for controlling the runs of genetic programming were the default values specified in Koza (1994) and which have been used for a number of different problems.

Results

We now describe the two best runs out of out 11 runs of this problem, starting with the second best.

The vast majority of the randomly generated programs in the initial random population (generation 0) of run 1 have a zero or near-zero correlation, C, indicating that they are no better than random in classifying whether a protein segment is a transmembrane domain. However, even in the initial random population, some individuals are better than others.

The best-of-generation classifying program from generation 0 of run 1 has an in-sample correlation of 0.48 as a result of getting 99 true positives, 83 true negatives, 40 false positives, and 24 false negatives over the 246 in-sample fitness cases. This program has a standardized fitness of 0.26. This program myopically looks at only the last residue of the protein segment and categorizes the entire segment based only on one, highly flawed automatically defined function. However, this program is better than any of the other 3,999 programs in the population at generation 0. In the valley of the blind, the one-eyed man is king.

The worst-of-generation classifying program from generation 0 of run 1 has an in-sample correlation of –0.4 and standardized fitness is 0.70. This program achieves this negative value of correlation by using incomplete information in precisely the wrong way.

In generation 2 of run 1, the best-of-generation program achieves an incrementally better value for correlation (0.496 in-sample and 0.472 out-of-sample) by virtue of an incremental change consisting of just one residue in the definition of ADF0.

There is a major qualitative change in generation 5. The best of generation 5 is the first best-of-generation program that makes its prediction based on the entire protein segment. This program contains 62 points (i.e., 62 functions and terminals in the bodies of the branches), has a distinctly better in-sample correlation of 0.764, an out-of-sample correlation of 0.784, and a standardized fitness of 0.12.

```
(progn (defun ADF0 ()
         (values (ORN (ORN (I?) (A?)) (ORN
           (ORN (L?) (G?)) (N?)))))
       (defun ADF1 ()
         (values (ORN (ORN (ORN (ORN (G?)
           (D?)) (ORN (E?) (V?))) (ORN (ORN
           (R?) (E?)) (ORN (T?) (P?)))) (ORN
           (N?) (S?)))))
       (defun ADF2 ()
         (values (ORN (ORN (ORN (L?) (R?))
           (ORN (V?) (P?))) (ORN (G?) (L?)))))
       (progn (looping-over-residues (SETM1 (- (+ M1
           (ADF0)) (ADF1))))
       (values (* (% (+ (% -9.997 M3) M1) 6.602) (+ 6.738
           (% (- M3 L) (+ M3 M2)))))))
```

The iteration-performing branch of this program uses the settable variable M1 to create a running sum of the difference between two quantities. Specifically, as the iteration-performing branch is iteratively executed over the protein segment, M1 is set to the current value of M1 plus the difference between ADF0 and ADF1. ADF0 consists of nested ORNs involving the three hydrophobic residues (I, A, and L), one neutral residue (G), and one hydrophilic residue (N). ADF1 consists of nested ORNs involving one hydrophobic residue (V), four neutral residues (G, T, P, and S), and the four most hydrophilic residues (D, E, R, and N).

Because the neutral G residue and the hydrophilic N residue appear in both ADF0 and ADF1, there is no net effect on the running sum of the differences, M1, calculated by the iteration-performing branch when the current residue is either G or N. There is a positive contribution (from ADF0) to the running sum M1 only when the current residue is I, A, or L (all of which are hydrophobic), and there is a negative contribution (from ADF1) to the running sum M1 only when the current residue is D, E, or R (all of which are hydrophilic). The running sum M1 is a count (based on a sample of only three of the seven hydrophobic residues and only three of the seven hydrophilic residues) of the excess of hydrophobic residues over hydrophilic residues.

When simplified, the result-producing branch is equivalent to $1.17 \times (M_1 + 1)$, so the protein segment is classified as a transmembrane domain whenever M1 is greater than 0. In other words, whenever the number of occurrences of the three particular hydrophobic residues (I, A, and L) equals or exceeds the number of occurrences of the three particular hydrophilic residues (D, E, and R), the segment is classified as a transmembrane domain. This relatively simple calculation is a highly imperfect predictor of transmembrane domains, but it is often correct. Because it examines the entire given protein segment, it is considerably better than any of its ancestors.

In generation 6 of run 1, the best-of-generation program has marginally better values for correlation (0.766 in-sample and 0.834 out-of-sample). This improvement is a consequence of a small, but beneficial, evolutionary change in the definition of ADF1. This small incremental improvement (produced by the crossover operation) is typical of the intergenerational improvements produced by genetic programming.

The 62-point best of generation 8 of run 1 exhibits a substantial jump in performance over all its predecessors from previous generations. In-sample correlation rises to 0.92; out-of-sample correlation rises to 0.89.

```
(progn (defun ADF0 ()
          (values (ORN (ORN (ORN (I?) (M?))
            (ORN (V?) (C?))) (ORN (ORN (L?)
            (G?)) (N?)))))
       (defun ADF1 ()
          (values (ORN (ORN (ORN (ORN (G?)
            (D?)) (ORN (E?) (V?))) (ORN (ORN
            (R?) (E?)) (ORN (T?) (P?)))) (ORN
            (N?) (S?)))))
       (defun ADF2 ()
          (values (ORN (ORN (ORN (L?) (R?))
            (ORN (V?) (P?))) (ORN (G?) (L?)))))
       (progn (looping-over-residues (SETM1 (- (+ M1
          (ADF0)) (ADF1))))
       (values (* (+ M1 M3) (+ 6.738 (% (- M3 L) (+ M3
          M2)))))))
```

In this program, ADF0 tests for four (I, M, C, and L) of the seven hydrophobic residues, instead of three. Moreover, isoleucine (I), the most hydrophobic residue among the seven hydrophobic residues on the Kyte-Doolittle scale, has become one of the residues incorporated into ADF0. More important, ADF1 tests for three neutral residues (T, P, and S) as well as three hydrophilic residues (D, E, and R). The result-producing branch calculates $7.738M_1$. As before, a protein segment is classified as a transmembrane domain whenever the running sum M1 is positive.

The three neutral residues (T, P, and S) in ADF1, play an important role in ADF1 since a positive value of M1 can be achieved only if there are enough sampled hydrophobic residues in the segment to counterbalance the sum of the number of sampled hydrophilic and neutral residues.

In generation 11 of run 1, the 78-point best-of-generation program shown below has an in-sample correlation of 0.94 and a standardized fitness of 0.03. It scored 117 true positives, 122 true negatives, 1 false positive, and 6 false negatives over the 246 in-sample fitness cases. It has an out-of-sample correlation of 0.96 and a standardized fitness of 0.02 as a result of getting 122 true positives, 123 true negatives, 2 false positives, and 3 false negatives over the 250 out-of-sample fitness cases. Its out-of-sample error rate is only 2.0%.

```
(progn (defun ADF0 ()
          (values (ORN (ORN (ORN (I?) (M?))
            (ORN (V?) (C?))) (ORN (ORN (L?)
            (G?)) (N?)))))
       (defun ADF1 ()
          (values (ORN (ORN (ORN (ORN (G?)
            (D?)) (ORN (E?) (V?))) (ORN (ORN
            (R?) (E?)) (ORN (ORN (ORN (ORN (G?)
            (D?)) (ORN (E?) (V?))) (ORN (ORN
            (R?) (K?)) (ORN (T?) (P?)))) (ORN
            (N?) (S?))))) (ORN (N?) (S?)))))
       (defun ADF2 ()
          (values (ORN (ORN (ORN (L?) (Y?))
            (ORN (V?) (P?))) (ORN (G?) (L?)))))
       (progn (looping-over-residues (SETM1 (- (+ M1
          (ADF0)) (ADF1))))
       (values (* (+ M1 M3) (+ 6.738 (% (- M3 L) (+ M3
          M2)))))))
```

The iteration-performing branch of this program uses the settable variable M1 to create a running sum of the difference between two quantities. Specifically, as the iteration-performing branch is iteratively executed over the protein segment, M1 is set to the current value of M1 plus the difference between ADF0 and ADF1.

The result-producing branch calculates $7.738M_1$. Thus, a protein segment will be classified as being a transmembrane domain whenever the running sum M1 is positive.

In this program, ADF0 tests for four (I, M, C, and L) of the seven hydrophobic residues, including isoleucine (I), the most hydrophobic residue among the seven hydrophobic residues on the Kyte-Doolittle scale.

ADF1 tests for four of the seven hydrophilic residues (D, E, R, and K) and three neutral residues (T, P, and S). D, E, R, and K are the most hydrophilic residues from among the seven hydrophilic residues according to the Kyte-Doolittle scale. The three neutral residues (T, P, and S) in ADF1 play an important role in ADF1 since a positive value of M1 can be achieved only if there are a sufficiently large number of sampled hydrophobic residues in the segment to counterbalance the sum of the number of sampled hydrophilic residues and sampled neutral residues.

The three residues V, G, and N play no role in the calculation of the running sum M1 since they appear in both ADF0 and ADF1.

ADF2 is ignored by the iteration-performing branch and therefore plays no role at all in this program.

The operation of this program from generation 11 of run 1 can be summarized as follows: If the number of occurrences of I, M, C, and L in a given protein segment exceeds the number of occurrences of D, E, R, K, T, P, and S, then classify the segment as a transmembrane domain; otherwise, classify it as non-transmembrane.

After generation 11 of run 1, the in-sample performance of the best-of-generation program continues to improve. For example, the in-sample correlation improves from 0.94 to 0.98 between generations 11 and 18 and the number of in-sample errors (i.e., false positives plus false negatives) drops from 7 to 3. However, this apparent improvement after generation 11 is illusory and is due to overfitting. Genetic programming is driven to achieve better and better values of fitness by the relentless effects of Darwinian natural selection. Fitness for this problem is based on the value of the correlation for the predictions made by the genetically-evolved program on the *in-sample* set of fitness cases. However, the true measure of performance for a classifying algorithm is how well it generalizes to other, previously unseen sets of data (i.e., the *out-of-sample* data). In this run, the out-of-sample correlation drops from 0.96 to 0.94 between generations 11 and 18 and the number of out-of-sample errors increases from 5 to 7. The maximum value of out-of-sample correlation is attained at generation 11. After generation 11, the evolved classifying programs are being fit more and more to the idiosyncrasies of the particular in-sample fitness cases employed in the computation of fitness. The classifying programs after generation 11 are not getting better at classifying whether proteins segments are transmembrane domains. Instead, they are merely getting better at memorizing the in-sample data. In fact, a continuation of this run out to generation 50 produces no result better than that attained at generation 11.

We now consider run 2. This best-of-all run produced the best value of out-of-sample correlation of any run, namely 0.968.

```
(progn (defun ADF0 ()
          (values (ORN (ORN (ORN (I?) (H?))
            (ORN (P?) (G?))) (ORN (ORN (ORN
            (Y?) (N?)) (ORN (T?) (Q?))) (ORN
            (A?) (H?))))))
       (defun ADF1 ()
          (values (ORN (ORN (ORN (A?) (I?))
            (ORN (L?) (W?))) (ORN (ORN (T?)
            (L?)) (ORN (T?) (W?))))))
       (defun ADF2 ()
          (values (ORN (ORN (ORN (ORN (ORN (D?)
            (E?)) (ORN (ORN (ORN (D?) (E?))
            (ORN (ORN (T?) (W?)) (ORN (Q?)
            (D?)))) (ORN (K?) (P?)))) (ORN (K?)
            (P?))) (ORN (T?) (W?))) (ORN (ORN
            (E?) (A?)) (ORN (N?) (R?))))))
       (progn (loop-over-residues (SETM0 (+ (- (ADF1)
          (ADF2)) (SETM3 M0))))
       (values (% (% M3 M0) (% (% (% (- L -0.53) (* M0
          M0)) (+ (% (% M3 M0) (% (+ M0 M3) (% M1
          M2))) M2)) (% M3 M0))))))
```

This high correlation was achieved on generation 20 by the 105-point program above with an in-sample correlation of 0.976 resulting from getting 121 true positives, 122 true negatives, 1 false positive, and 2 false negatives over the 246 in-sample fitness cases. Its out-of-sample correlation of 0.968 is the result of getting 123 true positives, 123 true negatives, 2 false positives, and 2 false negatives over the 250 out-of-sample fitness cases. Its out-of-sample error rate is only 1.6%.

Ignoring the three residues common to the definition of both ADF1 and ADF2, ADF1 returns 1 if the current residue is I or L and ADF2 returns 1 if the current residue is D, E, K, R, Q, N, or P. I and L are two of the seven hydrophobic residues on the Kyte-Doolittle scale. D, E, K, R, Q, and N are six of the seven hydrophilic residues, and P is one of the neutral residues.

In the iteration-performing branch of this program from generation 20 of run 2, M0 is the running sum of the differences of the values returned by ADF1 and ADF2. M0 will be positive only if the hydrophobic residues in the protein segment are so numerous that the occurrences of I and L outnumber the occurrences of the six hydrophilic residues and one neutral residue of ADF2. M3 is the same as the accumulated value of M0 except that M3 lags M0 by one residue. Because the contribution to M3 in the iteration-performing branch of the last residue is either 0 or 1, M3 is either equal to M0 or is one less than M0.

The result-producing branch is equivalent to

$$\frac{M_3^3}{M_0(M_0+M_3)(Len+0.53)}$$

The subexpression (- LEN -0.53) is always positive and therefore can be ignored in determining whether the result-producing branch is positive or nonpositive. Because of the close relationship between M0 and M3, analysis shows that the result-producing branch identifies a protein segment as a transmembrane domain whenever the running sum of the differences, M0, is greater than 0, except for the special case when M0 = 1 and M3 = 0. This special case occurs only when the running values of M0 and M3 are tied at 0 and when the very last residue of the protein segment is I or L (i.e., ADF1 returns 1).

Ignoring this special case, we can summarize the operation of this overall best-of-all program from generation 20 of run 2 as follows: If the number of occurrences of I and L in a given protein segment exceeds the number of occurrences of D, E, K, R, Q, N, and P, classify the segment as a transmembrane domain; otherwise, classify it as a non-transmembrane area.

Out-of-sample correlation closely tracks the in-sample correlation in the neighborhood of generation 20 of run 2. At generation 20, the out-of-sample correlation is 0.968 and the in-sample correlation is 0.976.

Additional Work

We redid the above work using arithmetic and conditional operations in the function set of the automatically defined functions (rather than the set-creating OR function). Our best-of-all evolved program in this arithmetic-performing version of the transmembrane also scored an out-of-sample error rate of 1.6% (Koza 1994).

Conclusions

Table 1 shows the out-of-sample error rate for the four algorithms for classifying transmembrane domains reviewed in Weiss, Cohen, and Indurkhya (1993) as well as the out-of-sample error rate of our best-of-all genetically-evolved program from generation 20 of run 2 above. We wrote a computer program to test the solution discovered by the SWAP-1 induction technique used in the first experiment of Weiss, Cohen, and Indurkhya (1993). Our implementation of their solution produced an error rate on our test data identical to the error rate reported by them on their own test data (i.e., the 2.5% of row 4 of the table).

Table 1 Comparison of five methods.

Method	Error rate
von Heijne 1992	2.8%
Engelman, Steitz, and Goldman 1986	2.7%
Kyte-Doolittle 1982	2.5%
Weiss, Cohen, and Indurkhya 1993	2.5%
Best genetically-evolved program	1.6%

As can be seen, the error rate of the best-of-all genetically-evolved program from generation 20 of run 2 is better than the error rates of the other four methods reported in the table. This genetically evolved program is an instance of an algorithm discovered by an automated learning paradigm that is superior to that written by human investigators. In fact, our second best genetically evolved program (from generation 11 of run 1) also outscores the other four methods (with an out-of-sample error rate of 2.0%).

In summary, without using foreknowledge of hydrophobicity, genetic programming with automatic function definition was able to evolve a successful classifying program consisting of initially-unspecified detectors, an initially-unspecified iterative calculation incorporating the as-yet-undiscovered detectors, and an initially-unspecified final calculation incorporating the results of the as-yet-undiscovered iteration.

Acknowledgments

James P. Rice of the Knowledge Systems Laboratory at Stanford University did the computer programming of the above on a Texas Instruments Explorer II$^+$ computer.

References

Anfinsen, C. B. 1973. Principles that govern the folding of protein chains. *Science* 81: 223-230.

Bairoch, A. and Boeckmann, B. 1991. The SWISS PROT protein sequence data bank. *Nucleic Acids Research* 19: 2247–2249.

Davis, L. (editor). 1987. Genetic Algorithms and Simulated Annealing. Pittman.

Davis, L. 1991. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.

Engelman, D., Steitz, T., and Goldman, A. 1986. Identifying nonpolar transbilayer helices in amino acid sequences of membrane proteins. *Annual Review of Biophysics and Biophysiological Chemistry*. Volume 15.

Forrest, S. (editor). 1993. *Proceedings of the Fifth International Conference on Genetic Algorithms.* Morgan Kaufmann.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* Ann Arbor, MI: University of Michigan Press 1975. Also available from Cambridge, MA: The MIT Press 1992.

Kinnear, K. E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.

Koza, J. R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT Press.

Koza, J. R., and Rice, J. P. 1992.*Genetic Programming: The Movie.* Cambridge, MA: The MIT Press.

Koza, J. R. 1994. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: The MIT Press.

Kyte, J. and Doolittle, R. 1982. A simple method for displaying the hydropathic character of proteins. *Journal of Molecular Biology.* 157:105-132.

Matthews, B. W. 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochemica et Biophysica Acta*. 405:442-451.

Michalewicz, Z. 1992. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag.

von Heijne, G. Membrane protein structure prediction: Hydrophobicity analysis and the positive-inside rule. *Journal of Molecular Biology.* 225:487–494.

Weiss, S. M., Cohen, D. M., and Indurkhya, N. 1993. Transmembrane segment prediction from protein sequence data. In Hunter, L., Searls, D., and Shavlik, J. (editors). *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology.* Menlo Park, CA: AAAI Press.