Predicting Whether Or Not a 60-Base DNA Sequence Contains a Centrally-Located Splice Site Using Genetic Programming

Simon Handley
Computer Science Department
Stanford University
Stanford, CA 94305
(415) 723-4096 shandley@cs.stanford.edu
URL: http://www-leland.stanford.edu/~shandley

Abstract

An evolutionary computation technique, genetic programming, was used to create programs that classify DNA sequences into one of three classes: (1) contains a centrally-located donor splice site, (2) contains a centrally-located acceptor splice site, and (3) contains neither a donor nor an acceptor. The performance of the programs created are competitive with previous work.

1. INTRODUCTION

In eukaryotes, not all of the messenger RNA (the RNA transcribed from the DNA) necessarily ends up being expressed as protein. After an mRNA sequence is transcribed from a DNA sequence, and before it is translated into a protein, contiguous subsequences of the mRNA sequence are removed, or *spliced* out. The subsequences that are removed are called *introns*; the intervening subsequences that get expressed as protein are called *exons*. The interstitial position between an intron and an exon (and vice versa) is called a *splice site*. The splice site between the end of an intron and the start of an exon is called an *acceptor* site, the site between the end of an exon and the start of an intron is called a *donor* site. Singer and Berg (Singer & Berg 1991) discuss eukaryotic translation in detail.

The huge amounts of DNA currently being produced by the various sequencing projects is unannotated or raw: it is just a very long string of As, Cs, Gs and Ts. An annotated DNA sequence additionally has various features located on it. One particularly interesting feature of a DNA sequence is the gene: a roughly-contiguous subsequence of DNA that codes for a particular protein. Since proteins catalyze most reations in cells, determining the sequnce, structure and function of the particular proteins in cells is very important.

How are the locations of genes in raw DNA determined? Because our understanding of the structure of genes in incomplete, most gene-finding programs (Guigó et al. 1992; Krogh, Mian, & Haussler 1994; Roberts 1991; Uberbacher & Mural 1991; Xu et al. 1994; Xu et al. 1995) start with a collection of imprecise feature-detectors (programs that return the probability of a feature being at a particular location) and use dynamic programming to find the most probable collection of features.

In this paper I use genetic programming to evolve splice site detectors, i.e. programs that can distinguish between acceptor splice sites, donor splice sites and ordinary DNA.

2. PREVIOUS WORK

Kudo et al. (Kudo, Iida, & Shimbo 1987) induced an automaton that recognizes 5'-splice sites. They started with 156 mammalian 5'-splice sites that they converted into 156 separate automata, each of which recognizes one and only one splice site (the conversion is the obvious one). They then combine all these 156 separate automata into one big automaton, by adding a start state (plus ε transitions to the 156 start states in the 156 smaller automata) and an end state (plus similar ε transitions). This new automaton recognizes a sequence if and only if it is one of the original 156 splice sites: there is no generalization. To get a more general automaton, the automaton is compressed: states are merged if they either have the same, say, 7 immediate predecessors or the same 7 successors. The resulting more general automaton gets $Q_1 = 55\%$ [1] on a testing set of 20 splice sites (it's unclear what the ratio of positive to negative examples is).

Brunak et al. (Brunak, Engelbrecht, & Knudsen 1990) trained a 19·4×20×2 neural net to recognize splice sites in DNA. They trained on 33 human genes and discovered 7 incorrect database entries that had inhibited learning.

Towell (Towell 1991; Towell, Craven, & Shavlik 1991) compared various machine learning algorithms on the

[1] Q_1 measures the number of positive examples correctly classified, $\frac{\text{number of positives correctly predicted}}{\text{number of positives}}$.

```
function genetic-programming( fitness-measure, class-of-programs, pop-size, termination-criterion )
  gen ← 0
  for i ← 0 ... pop-size - 1 do
    population[0][i] ← randomly-generated-program( class-of-programs )
    compute & store fitness-measure( population[0][i] )
  end
  while not termination-criterion( population[gen] ) do
    create population[gen+1] by crossing-over and copying probably-highly-fit programs from
              population[gen]
    for i ← 0 ... pop-size - 1 do compute & store fitness-measure( population[gen+1][i] )
    gen ← gen + 1
  end
  return fittest individual in population[gen]
end
```

Figure 1. Genetic programming pseudocode.

splice sites problem. His data (Towell, Noordewier, & Shavlik 1992) was all 3190 examples of splice sites in primate sequences in Genbank 64.1 (Benson et al. 1994). Towell compared the following algorithms on a 1000-site subset of the 3190 primate sequences (half of which were negative examples): nearest-neighbour, Cobweb, perceptron (neural net with no hidden neurons), ID3, PEBLS, back-propagation, and KBANN (a neural net training algorithm developed by Towell). The above seven algorithms were divided into two classes: those whose performance depended on the ordering of the training examples and those whose performance was ordering-independent. For the ordering-dependent algorithms (Cobweb, the perceptron, PEBLS, back-propagation and KBANN) Towell ran each algorithm on 11 different permutations of the 1000 training examples. He then defined a training case as correctly classified if a majority of the 11 independent runs correctly classify that training case. Such a methodology is inappropriate for a Monte Carlo-style algorithm such as genetic programming (Koza 1992; Koza 1994b; Koza & Rice 1992) because a random classifier can appear to do arbitrarily well by just looking at more and more permutations of the training data. So, to provide a benchmark for the results in this paper I used the two order-independent algorithms: nearest-neighbour and ID3. The performance measure used by Towell to compare the above seven algorithms was the relative information score (RIS) (Kononenko & Bratko 1991).

Lapedes et al (Lapedes et al. 1990) also tried to predict whether or not a *H. sapiens* DNA sequence that contains "AG" or "GT" is a splice site. The training set size isn't given, the testing set contains 50 examples. They tried windows of 11, 21 and 41 bases around the "AG"/"GT". They got averaged Q_1 values of 91.2% on acceptor sites and 94.5% on acceptor sites.

3. THE TECHNIQUE: GENETIC PROGRAMMING

Genetic programming (Koza 1992; Koza 1994b; Koza & Rice 1992) can be thought of as a black box whose output is a program that is (hopefully) fit according to some fitness measure. See figures 1 and 2.

Genetic programming starts with a population of randomly generated computer programs, a fitness measure, a class of programs (i.e., a function set and a terminal set) and a termination criterion and uses artificial selection and sexual reproduction to produce increasingly fit populations of computer programs:

Genetic programming has been applied to the following sequence analysis problems: predicting whether or not a residue is in a helix (Handley 1993; Handley & Klingler 1993); recognizing the cores of helices (Handley 1994a); predicting the degree to which a residue in a protein is exposed to solvent (Handley 1994b); predicting whether or not a DNA sequence contains an *E. coli* promoter (Handley 1995b); predicting if an mRNA sequence is an intron or an exon (Handley 1995a); predicting whether or

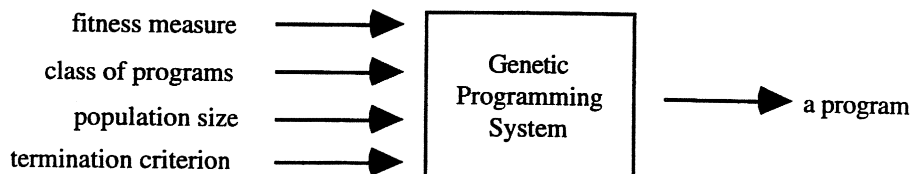

Figure 2. Genetic programming as a black box.

not a sequence contains an omega loop (Koza 1994b); and predicting whether or not a sequence contains a transmembrane domain (Koza 1994a; Koza 1994b).

4. APPLYING GENETIC PROGRAMMING: CHOOSING VALUES FOR PARAMETERS

The black box in figure 2 has four inputs: the fitness measure, the class of programs, the population size and the termination criterion.

4.1. THE FITNESS MEASURE

The fitness cases are the positive and negative examples of splice sites (each positive example is further split into donors and acceptors). The fitness of a program is a measure of the number of correct predictions.

The fitness measure used was the relative information score (RIS) that Towell used (see the "Previous work" section). RIS is defined as $RIS = (I_a/E) \times 100\%$ where

$$I_a = \frac{\sum_{\text{fitness cases}, i} I_i}{N_{fc}},$$

$$I_i = \begin{cases} -\lg(P(C_j)) + \lg(P'(C)) & \text{if } P'(C) \geq P(C_j) \\ \lg(1 - P(C_j)) - \lg(1 - P'(C)) & \text{otherwise} \end{cases},$$

and

$$E = - \sum_{\text{classes}, j} P(C_j) \lg(P(C_j)).$$

P'(C) is the probability of fitness case *i* being correctly classified (= 1 if correct, 0 otherwise). $P(C_j)$ is the prior probability of class j. N_{fc} is the number of fitness cases.

4.2. THE CLASS OF PROGRAMS

Programs that classify DNA sequences must be able to do two sorts of computation: first, they must be able to look at multiple bases of the sequence; second, they must be able to condition on individual bases.

The ability to look at different bases of a sequence is satisfied by the use of a "turtle" that moves along the sequence. Two functions are used—`lft` and `rght`—that each take one argument and that, when executed, first move the turtle one base left or right, respectively, and then evaluate their argument. A third function, `home`, records the turtle position, evaluates its sole argument and then "homes" the turtle back to the recorded position.

The ability to conditionally execute different parts of a program depending on the bases seen is supplied by the following functions. (First, note that because the sequences are of finite length it is useful to add a fifth base '-' that stretches infinitely far both upstream and downstream of each promoter and non-promoter.)

- `a?`, `c?`, `g?`, `t?` and `-?`. These functions take two arguments and either execute the first one (if the indicated base is in the position pointed to by the turtle) or execute the second one (otherwise). For example, the program `(c? +3 -4)` returns +3 if the base pointed to by the turtle is "C" and returns -4 otherwise.
- `switch` (a.k.a. `sw`). This function takes five arguments and executes one of the five, depending on the base pointed to by the turtle. That is, the program `(switch <a-arg> <c-arg> <g-arg> <t-arg> <--arg>)` evaluates, for example, to "<c-arg>" if the base pointed to by the turtle is a "C".
- `if>=0`. This function returns its second argument if its first argument is >= 0.0 and returns its third argument otherwise.
- `av`, `cv`, `gv`, `tv` and `-v`. These functions return either +1.0 or -1.0 depending on the base pointed to by the turtle. For example, `av` is equivalent to `(a? +1.0 -1.0)`.

In addition to the above two classes of functions, some miscellaneous functions were added:

- arithmetic functions: `+`, `-`, `*` and `%`, where $(\%XY) = \begin{cases} X/Y, & Y \neq 0 \\ 1, & \text{otherwise} \end{cases}$. Note that `(% X Y)`'s second argument, `Y`, is evaluated before its first argument, `X`.
- random numbers between -10.0 and +10.0 were added to generation 0 programs as zero-argument constant functions.

4.3. THE POPULATION SIZE

The genetic algorithm is designed to search spaces that are highly dimensional and that are moderately epistatic[2]. The population is used to implicitly store multiple partial solutions to sub-spaces of the search space. For this reason, the size of the population is directly proportional to the difficulty (the degree of epistasis and the number of dimensions) of the problem that can be solved. Based loosely on my experience with other DNA motif problems and the available computer resources I chose a population size of 64,000 individuals.

4.4. THE TERMINATION CRITERION

A run was terminated if one or more of the following conditions were met:

- a total of 75 generations had been evolved (i.e., 76 populations: 1 randomly created and 75 evolved), or
- the computer crashed, or
- the population started to overfit the data.

---

[2] By "epistasis" I mean the degree of interconnection or correlation between the "independent" variables of a problem. A problem with little epistasis can trivially be solved by independently optimizing each variable. A problem with a high degree of epistasis can only be solved by enumerating the entire search space (consider, for example, a spike function: having the value ∞ at a single point and 0 elsewhere). A moderately epistatic problem can be solved by an adaptive search procedure that exploits regularities in the search space.

run	Generation	Training set	Testing set	Evaluation set
1	16	83.01	81.93	85.15
2	14	81.93	87.22	85.16
3	24	82.04	89.00	86.59

Table 1. Fitness (RIS) on the training set, the testing set and the evaluation set of the best-of-generation (on the training set) individual that scores best on the testing set. The "Generation" column is the generation that the indicated individual first appeared in.

Given a set of examples, enough generations and enough runs, genetic programming will have no problem finding arbitrarily accurate classifiers. If the goal, however, is to discover programs that can classify unseen examples then such highly accurate programs are of little use. Programs that accurately classify the training data *may* not have accurately captured the underlying model that generated that data: they may have learnt the noise as well as the signal. Of course, a highly accurate program *could* be accurately modeling the underlying process. The point is that there is no way to tell, *from the training data alone*, how well a learnt program is modeling the underlying process.

The only way to predict how well an inductively learnt program will perform on unseen data is to partition the data into separate data sets, to train on one and to test on the second. The performance of learnt programs on the second data set (the data that the program wasn't trained on) will be a reliable predictor of its performance on unseen data. I call this second data set the *evaluation set*.

Partitioning the data set into two sets solves the problem of evaluating the resultant programs but doesn't solve the problem of deciding when to terminate each run of genetic programming. So I further divide the first data set (the promoters that the programs are trained on) into two sets: the *training set* and the *testing set*. The training set is used as described above in the genetic-programming() subroutine: these examples define the fitness of each program and this fitness determines which genetic operations that program participates in etc. The testing set is used to evaluate the degree to which the fittest (on the training set) program of each generation is overfitting the training data. I compute a 10-generation moving average of the fitness on the testing set of the fittest (on the training set) program of each generation and stop a run when the moving average starts to get worse: i.e., when the best (on the training set) program of each generation gets less fit on the testing set.

To summarize, the training examples (as described in the "Data" section) are partitioned into three mutually exclusive sets: the training set, the testing set and the evaluation set. The training set is used to drive the evolutionary process, the testing set is used to stop the evolutionary process, and the evaluation set is used to evaluate the best individuals of a number of runs on unseen promoters.

For compatibility with Towell's (Towell 1991) results, I used 10-way jackknifing: the 1000 examples were partitioned into 10 sets of 100 examples each and one of the 10 sets was used to evaluate the performance of the best program trained on the remaining 900 examples. The 900 examples were further divided into 600 (to train on) and 300 (to test for overfitting).

5. THE DATA

These experiments used Towell's (Towell 1991) data: 1000 randomly chosen positive and negative examples from a database of 3190 positive and negative examples of primate splice sites (Towell, Noordewier, & Shavlik 1992). In the original database (Towell, Noordewier, & Shavlik 1992) there is an equal number of donors and acceptors.

6. THE RESULTS

I did three runs of genetic programming on the data described in the previous section ("The data") and with parameters set as described in the "Applying genetic programming" section. Each run was terminated as described in the "Applying genetic programming" section above. Table 1 shows the best (on the testing set) best-of-generation individual from each of the three runs. Figures 3, 4 and 5 show the individuals.

```
(if>=0 (-? (- (-? tv tv) (rht tv))
(home (sw (home (t? tv tv)) (-? (a? tv
-0.616664) (if>=0 -v tv av)) (rht (t?
8.65255 -v)) (sw gv gv cv cv tv) (a? -v
cv)))) (sw (g? (+ -v cv) (- -v tv))
(if>=0 (sw (rht (- av tv)) (- cv av) (+
cv gv) av (if>=0 (lft gv) (c? 7.19598
tv) (sw -v cv tv av gv))) (lft (if>=0
gv av tv)) (% gv 4.99743)) (rht (% (-?
cv 8.13416) (+ cv gv))) (* (+ tv cv)
(rht cv)) (* cv (-? gv tv))) (lft (g?
(if>=0 (sw (sw gv cv 1.8385 gv -v) (-
cv av) (+ cv gv) av (+ tv av)) (lft (a?
0.878068 gv)) (% gv 4.99743)) (rht (sw
av gv gv av -4.99146)))))
```

Figure 3. The individual that scored 85.15 on the evaluation set.

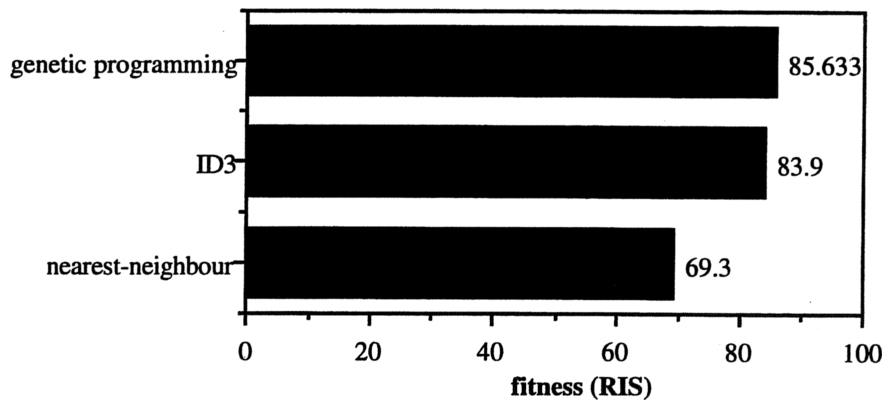

Figure 6. Performance on the evaluation set of three algorithms. The numbers for the nearest-neighbours and ID3 algorithms are from Towell's thesis (Towell 1991).

```
(g? (rht (- (t? (sw (sw 2.42938 av tv
-v 1.53658) (if>=0 8.0244 -v 6.91427)
(lft gv) (- -v tv) (if>=0 -v av cv))
(lft (lft (% av -v)))) (if>=0 (sw (rht
(g? -2.67231 -v)) (% cv cv) (-? tv -v)
(t? gv av) (% cv av)) (a? 9.89077 cv)
(lft (% av -v))))) (% (lft (sw (rht (g?
-2.67231 -v)) (% cv cv) (-? tv -v)
(if>=0 gv gv tv) (% cv av))) (lft cv)))
```

Figure 4 The individual that scored 85.16 on the evaluation set.

```
list1 (g? (rht (t? (+ gv gv) (lft (lft
(lft (a? 9.52972 -v)))))) (- (lft (g?
(t? (lft cv) (lft av)) (rht (- gv
gv)))) (lft (sw (- (home cv) (t?
-7.72872 -v)) (a? (* tv cv) (if>=0 av
-2.39191 gv)) (* (lft (+ gv tv)) (* tv
cv)) gv -2.39191)))))
```

Figure 5. The individual that scored 86.59 on the evaluation set.

7. COMPARISON WITH PREVIOUS WORK

Towell's (Towell 1991; Towell, Craven, & Shavlik 1991) work is used here as the state of the art. As discussed in the "Previous work" section, only two of the algorithms considered by Towell are relevant here: nearest-neighbour and ID3. Figure 6 shows the average fitness (RIS) on nearest-neighbour, ID3 and genetic programming.

8. CONCLUSIONS

This paper shows that an evolutionary computation technique, genetic programming, can create programs that outperform the best classifiers produced by other machine learning algorithms.

8. ACKNOWLEDGEMENTS

I would like to thank the anonymous reviewer for his/her comments.

9. REFERENCES

Benson, D.; Boguski, M.; Lipman, D. J.; and Ostell, J. GenBank. 1994. *Nucleic Acids Research* 22 (17): 3441–4.

Brunak, S.; Engelbrecht, J.; and Knudsen, S. Neural network detects errors in the assignment of mRNA splice sites. 1990. *Nucleic Acids Research* 18 (16): 4797–801.

Guigó, R.; Knudsen, S.; Drake, N.; and Smith, T. Prediction of gene structure. 1992. *Journal of Molecular Biology* 1992 :141–57.

Handley, S. G. 1993. Automated learning of a detector for α-helices in protein sequences via genetic programming. In Proceedings of the Fifth International Conference on Genetic Algorithms, 271–8. Urbana-Champaign, IL.: Morgan Kaufmann.

Handley, S. G. 1994a. Automated learning of a detector for the cores of α-helices in protein sequences via genetic programming. In First IEEE Conference on Evolutionary Computation, 474–9. Walt Disney World Dolphin Hotel, Orlando, FL.: IEEE.

Handley, S. G. 1994b. The prediction of the degree of exposure to solvent of amino acid residues via genetic programming. In Second International Conference on

Intelligent Systems for Molecular Biology, 156–60. Stanford University, Stanford, CA.: AAAI Press.

Handley, S. G. 1995a. Classifying Nucleic Acid Sub-Sequences as Introns or Exons Using Genetic Programming. In press.

Handley, S. G. 1995b. Predicting Whether Or Not a Nucleic Acid Sequence is an *E. coli* Promoter Region Using Genetic Programming. In First International Symposium on Intelligence in Neural and Biological Systems INBS '95, 122–7. Herndon, VA.: IEEE Computer Society Press.

Handley, S. G. and Klingler, T. 1993. Automated learning of a detector for α-helices in protein sequences via genetic programming. In *Artificial Life at Stanford 1993*, ed. Koza, J. 144–52. Stanford, CA: Stanford Bookstore.

Kononenko, I. and Bratko, I. Information-based evaluation criterion for classifier's performance. 1991. *Machine Learning* 6 :67–80.

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

Koza, J. R. 1994a. Evolution of a comuter program for classifying protein segments as transmembrane domains using genetic programming. In Second International Conference on Intelligent Systems for Molecular Biology, 244–52. Stanford University, Stanford, CA.: AAAI Press.

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

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

Krogh, A.; Mian, I. S.; and Haussler, D. A hidden markov model that find genes in *E. Coli* DNA. University of California, Santa Cruz, 1994. UCSC-CRL-93-16.

Kudo, M.; Iida, Y.; and Shimbo, M. Syntactic pattern analysis of 5'-splice site sequences of mRNA precursors in higher eukaryote genes. 1987. *Computer Applications in the Biosciences* 3 (4): 319–24.

Lapedes, A.; Barnes, C.; Burks, C.; Farber, R.; and Sirotkin, K. 1990. Application of neural networks and other machine learning algorithms to DNA sequence analysis. In *Computers and DNA, SFI studies in the sciences of complexity*, ed. Bell, G. and Marr, T. 157–82. VII. Addison-Wesley.

Roberts, L. GRAIL seeks out genes buried in DNA sequence. 1991. *Science* 254 (5033): 805.

Singer, M. and Berg, P. 1991. *Genes & genomes: A changing perspective.* Mill Valley, CA: University Science Books.

Towell, G. G. “Symbolic knowledge and neural networks: insertion, refinement, and extraction.” Ph. D., University of Wisconsin-Madison, 1991.

Towell, G. G.; Craven, M. W.; and Shavlik, J. W. 1991. Constructive induction in knowledge-based neural networks. In Eighth International Machine Learning Workshop, Northwestern University.: Morgan Kaufmann.

Towell, G. G.; Noordewier, M.; and Shavlik, J. 1992. Primate splice-junction gene sequences (DNA). UCI Irvine Machine learning database, URL="ftp://ftp.ics.uci.edu/pub/machine-learning-databases".

Uberbacher, E. C. and Mural, R. J. Locating protein-coding regions in human DNA sequences by a multiple sensor-neural network approach. 1991. *Proceedings of the National Academy of Science (USA)* 88 :11261–5.

Xu, Y.; Einstein, J. R.; Mural, R. J.; Shah, M.; and Uberbacher, E. C. 1994. An improved system for exon recognition and gene modeling in human DNA sequences. In Second International Conference on Intelligent Systems for Molecular Biology, 376–84. Stanford University, Stanford, CA.: AAAI Press.

Xu, Y.; Helt, G.; Einstein, J. R.; Rubin, G.; and Uberbacher, E. C. 1995. Drosophila GRAIL: An intelligent system for gene recognition in Drosophila DNA sequences. In First International Symposium on Intelligence in Neural and Biological Systems INBS '95, 128–35. Herndon, VA.: IEEE Computer Society Press.

W. LANGDON.

National Resource Laboratory
for the study of
Brain and Behavior

Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications

Justinian P. Rosca, Editor

Technical Report 95.2
June 1995

UNIVERSITY OF
ROCHESTER