Machine Learning — Past and Future

Ray J. Solomonoff
Visiting Professor, Computer Learning Research Center
Royal Holloway, University of London

IDSIA, Galleria 2, CH–6928 Manno–Lugano, Switzerland
rjsolo@ieee.org http://world.std.com/˜rjs/pubs.html

August 11, 2009*

Abstract

I will first discuss current work in machine learning – in particular, feedforeword Artificial Neural Nets (ANN), Boolean Belief Nets (BBN), Support Vector Machines (SVM), Radial Basis Functions (RBF) and Prediction by Partial Matching (PPM). While they work quite well for the types of problems for which they have been designed, they do not use recursion at all and this severely limits their power.

Among techniques employing recursion, Recurrent Neural Nets, Context Free Grammar Discovery, Genetic Algorithms, and Genetic Programming have been prominent.

I will describe the Universal Distribution, a method of induction that is guaranteed to discover any describable regularities in a body of data, using a relatively small sample of the data. While the incomputability of this distribution has sharply limited its adoption by the machine learning community, I will show that paradoxically, this incomputability imposes no limitation at all on its application to practical prediction.

My recent work has centered mainly on two systems for machine learning. The first might be called "The Baby Machine" We start out with the machine having little problem specific knowledge, but a very good learning algorithm. At first we give it very simple problems. It uses its solutions to these problems to devise a probability distribution over function space to help search for solutions to harder problems. We give it harder problems and it updates its probability distribution on their solutions. This continues recursively, solving more and more difficult problems.

The task of writing a suitable training sequence has been made much easier by Moore's Law, which gives us enough computer speed to enable large conceptual jumps between problems in the sequence.

*Revision of lecture given at AI@50, The Dartmouth Artificial Intelligence Conference, Dartmouth, N.H. July 13-15, 2006.

A more difficult task is that of updating the probability distribution when new problems are solved. I will be using some of the current techniques for machine learning in this area: PPM and Grammar Discovery are particularly promising. It is my impression that the probability models need not be recursive for the initial learning. Later the system can, with suitable training, work on the problem of updating its own probability distribution, using fully recursive models.

Genetic Programming is my second area of recent research. Koza's Genetic Programming system has been able to solve many difficult problems of very varied kinds. The system itself is extremely slow, however – it has taken a cluster of 1000 Pentiums about a month to solve some of the more difficult problems. I have found a large number of modifications of the system that I expect will dramatically improve its speed and broaden the scope of problems it can solve.

The Future — The cost halving constant in Moore's law is now about 18 months. When A. I. pays a significant role in the reduction of this time constant, we begin to move toward the singularity. At the present time I believe we have a good enough understanding of machine learning, for this to take place. While I hesitate to guess as to just when the singularity will occur, I would be much surprised if it took as much as 20 years. As for the next 50 years of A.I., I feel that predicting what will happen after this singularity is much like guessing how the universe was before the Big Bang – It's a real discontinuity!

Introduction

To start, I'd like to define the scope of my interest in A.I. I am not particularly interested in simulating human behavior. I am interested in creating a machine that can work very difficult problems much better and/or faster than humans can — and this machine should be embodied in a technology to which Moore's Law applies. I would like it to give a better understanding of the relation of quantum mechanics to general relativity. I would like it to discover cures for cancer and AIDS. I would like it to find some very good high temperature superconductors. I would not be disappointed if it were unable to pass itself off as a rock star.

The Moore's Law condition assures us that we can continue to improve the device through affordable increases in computation capacity. My particular area of interest is Machine Learning.

The problem of learning for humans: You have all of the information that you've gathered in your life, plus a certain amount you were born with. How can you best use this information to make decisions?

The problem of probability: Given a certain body of data plus certain a priori information — how do we best make predictions about the immediate future?

It is seen that while the problem of learning is more general, learning and probabilistic prediction are very close. The problem of machine learning is that of getting good approximations to ideal probabilistic prediction. I will discuss this goal in more detail in the section 2.

1 Models for Machine Learning: Recursive and Nonrecursive.

In 1957 Frank Rosenblatt (Ros 57) wrote a report on Perceptrons that initiated a burst of enthusiasm in an area of machine learning. About 12 years later, Minsky and Papert (Min 69) wrote an analysis of Perceptrons — showing that they couldn't learn the logical "exclusive or" function. This resulted in a cutoff of federal funds for neural nets for a while, but eventually it was found that artificial neurons that were only slightly different from Perceptrons *could* do "exclusive or" — and that they were *universal* in the sense of being able to approximate any continuous function.

There was a new burst of enthusiasm, (and even funding) for neural nets — marked to some extent in 1986 by Rumelhart and McClelland's "Parallel Distributed Processing" (Rum 86).

A second edition of Minsky and Papert's book (Min 88) appeared in 1988 — pointing out that while neural nets might be able to distinguish between sets that had an even (versus odd) number of elements, they needed a proportionally larger number of neurons and larger data set to do so. They had the same difficulty in dealing with the "greater than" function.

It is clear that in general, neural nets do very poorly in discriminations that are best described recursively. While the nets can (through their particular kind of universality) make such discriminations, they need a large number of neurons and a very large data set to get much precision.

It is characteristic of smart induction models, that they get good predictions with small amounts of data. Humans occasionally do what is called "One shot learning."

Another kind of problem in which neural nets do poorly: we have a data set that consists of a few cycles of a sine wave plus a little noise. Neural nets can extrapolate such data into the *near* future. They will not, however, recognize that the data is, indeed, a sine wave plus noise — which would enable extrapolation into the *more distant* future. If the data set were continued with a noisy sine wave of another frequency and amplitude, the neural net would need lots of data points and many more neurons to extrapolate it. A much more sophisticated learner would realize that two sine waves only need six parameters to characterize them — so it could extrapolate the sine waves with much higher precision using fewer data points.

In general, to do economical prediction, we need (at least) facilities to make recursive definitions.

It should be noted that it is not only feed forward neural nets that lack these facilities. A large number of techniques used in current machine learning are no better: i.e. Radial Basis Functions, Boolean Belief Nets, Support Vector

Machines, and Prediction by Partial Matching are a few: for certain areas of prediction these techniques are fine. They work very well for the problems that are normally given in machine learning contests.

With enough cubical Lego blocks one can make very beautiful buildings — but with the addition of wheels, axles, motors, sensors and computers, one can do much more!

To work really difficult problems, it is necessary to have all possible facilities available. Recursion is one of these facilities. Later, I will discuss more advanced tools.

Some machine learning techniques that get past the "Minsky-Papert recursion barrier": Recurrent Neural Nets, various Evolutionary Techniques, Minimum Description Length/Minimum Message Length, Algorithmic Probability/Universal Distribution, Stochastic Grammars, Inductive Logic Programming.

I will discuss some of these in subsequent sections.

2 The Universal Distribution

At the Dartmouth workshop, there was initially much interest in machine learning — but it was all theoretical, no one had programmed anything. Newell and Simon's "Logical Theorist"(New 57) was a program that solved problems in logic using heuristics. It seemed like a real breakthrough, and most of the A.I. work immediately following the workshop followed that lead.

I, however, continued work on Machine Learning and in 1957 wrote "An Inductive Inference Machine". It described the kind of system I've been working on ever since. At the time, I felt I knew enough about the induction process to be able to compute and optimize all aspects of the system. After a few years of work, it became clear that I did *not* know enough. I began studying induction in general and discovered probabilistic languages. This led in 1960 (Sol 60) to the discovery of the Universal Distribution (also called Algorithmic Probability — ALP). In a 1964 paper (Sol 64a), I described five methods of induction that I felt were probably equivalent. All of them dealt with the extrapolation of a sequence of binary symbols — all induction problems can be put into this form. What we do is assign a probability to any finite binary sequence. We can then use Bayes theorem to compute the probability of any particular continuation of any particular sequence. The big problem is: how do we assign these probabilities to strings? In the second method described in the paper, we have a universal Turing machine with three tapes: a unidirectional input tape, a unidirectional output tape, and a bidirectional work tape. If we feed it a tape with 0's and 1's on it, the machine may print some 0's and 1's on the output — It could print nothing at all or print a finite string and stop or it could print an infinite output string, or it could go into an infinite computing loop with no printing at all.

Suppose we want to find the ALP of finite string x. We feed random bits into the machine. There is a certain probability that the output will be a string that starts out with the string x. *That* is the Universal/Algorithmic Probability of string x.

To compute the ALP of string x:

$$P_M(x) = \sum_{i=0}^{\infty} 2^{-|S_i(x)|}$$

Here $P_M(x)$ is the universal probability of string x with respect to machine, M.

There are many finite string inputs to M that will give an output that begins with x. Say $S_i(x)$ is the i^{th} such string. We may call $S_i(x)$ "a description of x". $|S_i(x)|$ is the length in bits of the string, $S_i(x)$. [1]

$2^{-|S_i(x)|}$ is the probability that the random input will be $S_i(x)$.

$P_M(x)$ is then the sum of the probabilities of all the ways that a string beginning with x, could be generated.

This definition has some interesting properties:

First, it assigns high probabilities to strings with short descriptions — This is in the spirit of Ockham's razor. It is the converse of Huffman coding that assigns short codes to high probability symbols.

Second, the value is somewhat independent of what universal machine is used, because codes for one universal machine can always be obtained from another universal machine by the addition of a finite sequence of translation instructions.

A third property is incomputability: The equation for $P_M(x)$ tells us to find all strings that are "codes for x." Because of the Halting problem, it is impossible to tell whether certain strings are codes for x or not. While it is easy to make approximations to $P_M(x)$, the fact that it is incomputable has given rise to the common misconception that ALP is little more than an interesting theoretical model with no direct practical application. We will discuss this a bit later.

I didn't discover the fourth property until 1968 — eight years later and didn't publish a proof of this result until 1978 (Sol 78). It is the most important property of all. $P_M(x)$ is *complete*. This means that if there is any describable regularity in a batch of data, P_M will find it, using a relatively small amount of the data. At this time, it is the *only* induction method known to be complete.

More exactly: suppose $\mu(x)$ is a probability distribution on finite binary strings. For each $x = x_1, x_2 \cdots x_i$, μ gives a probability that the next bit, x_{i+1} will be 1:

$\mu(x_{i+1} = 1 | x_1, x_2 \cdots x_i)$

From P_M we can obtain a similar function
$P(x_{i+1} = 1 | x_1, x_2 \cdots x_i)$.

Suppose we use μ to generate a sequence, x, Monte Carlo-wise. μ will assign a probability to the $i + 1^{th}$ bit based on all previous bits. Similarly, P_M will assign a probability to the

[1]We include in the set, $S_i(x)$ only "minimal codes for x" – i.e. codes, S_i such that if we delete the last bit of S_i, the result is no longer a code for x. These S_i form a "prefix set", which assures us that the sum of all of the probabilities is ≤ 1.

$i+1^{th}$ bit of x. If P_M is a very good predictor, the probability values obtained from μ and from P_M will be very close, on the average, for long sequences. What I proved was:

$$\mathrm{E}_\mu \sum_{i=1}^{n} (\mu(x_{i+1}=1|x_1,x_2\cdots x_i)$$
$$-P(x_{i+1}=1|x_1,x_2\cdots x_i))^2 \leq \frac{1}{2} k \ln 2$$

The expected value of the sum of the squares of the differences between the probabilities is bounded by about $.35k$. k is the minimum number of bits that M, the reference machine, needs to describe μ. If the function μ is describable by functions that are close to M's primitive instruction set, then k will be small. — But whether large or small, the squared error in probability must converge faster than $\frac{1}{n}$ (because $\sum \frac{1}{n}$ diverges).

Later research has shown this result to be very robust — we can use a large, (non-binary) alphabet and/or use error functions that are different from total square difference (Hut 02). The probability obtained can be normalized or unnormalized (semi-measure)(Gac 97).

The function μ to be "discovered" can be any describable function — primitive recursive, total recursive, or partial recursive — even a large class of *incomputable* functions.

The desirable aspects of ALP are quite clear. We know of no other model of induction that is nearly as good.

But people immediately ask — what good is it if you can't compute it? Certainly a reasonable question. The answer is that for practical prediction we don't have to know ALP *exactly.* Approximations to it are quite usable and *the closer an approximation is to ALP, the more likely it is to share ALP's desirable qualities*

About the easiest kind of approximation to an incomputable number is making rational approximations to $\sqrt{2}$. We know that there is no rational number whose square is 2, but we can get arbitrarily close approximations. We can also compute an upper bound on the error of our approximation and for most methods of successive approximation we are assured that the errors approach zero. In the case of the universal distribution, the Bad News is that we *cannot* compute useful upper bounds on approximation error — but the Good News is that for few probability applications do we need this information.

On the other hand, for most applications an estimate of prediction error is needed. Cross validation is usually possible, but for ALP it may be unnecessary. ALP itself has no underfitting or overfitting and may be able to use error estimation techniques that are less wasteful of sample size.

Another reasonable question: Suppose you have a computable probability method that seems to work adequately — why bother doing an approximation to ALP? The answer is in ALP's precision. No matter how good a computable prediction method is, it is almost certain that there is an approximation to ALP that is better. In the field of finance — market prediction, derivative evaluation, insurance premium calculation, horse racing ..., it translates directly into financial gain. In other applications, it means better models, better understanding. We get our cure for cancer or our high temperature superconductor in a few years, rather than 50 years or never. Many A.I. problems are quite difficult and we cannot afford to stray very far from using the very best tools available.

The approximation problem for the universal distribution is very similar to that of approximating a solution to the travelling salesman problem, when the number of cities is too large to enable an exact solution. When we make trial paths, we always know the total length of each path — so we know whether one trial is better than another. In approximations for the universal distribution, we also always know when one approximation is better than another — and we know how much better. In some cases, we can combine trials to obtain a trial that is better than either of the component trials. In both TSP and ALP approximation, we never know how far we are from the theoretically best.

The completeness property of ALP is closely associated with its incomputability. Any complete induction system cannot be computable. Conversely, any computable induction system cannot be complete. For any computable induction system, it is possible to construct data sequences for which that system gives *extremely* poor probability values. The sum of the squared errors diverges linearly in the sequence length.

Appendix 1 gives a simple construction of this kind.

We note that the incomputability of ALP makes such a construction impossible and its probability error always converges to zero for *any* finitely describable sequence.

3 SA, The Scientist's Assistant

We will describe recent developments in a system for machine learning that we've been working on for some time (Sol 86, 89, 03a). It is meant to be a "Scientist's Assistant" of great power and versatility in many areas of science and mathematics. It differs from other ambitious work in this area in that we are not so much interested in knowledge itself, as we are in how it is acquired — how machines may learn.

We will begin with the description of a simple kind of inductive inference system. We are given a sequence of Q, A pairs (questions and correct answers). Then, given a new Q, the system must give an appropriate answer. At first, the problems will be mathematical questions in which there is only one correct answer. The system tries to find an appropriate function F so that for all examples, $Q_i, A_i; F(Q_i) = A_i$. We look for F functions that have highest a priori probabilities — that have "short descriptions". In generating such functions, we use compositions of primitive functions built into the system. The overall language used is very close to Lisp, but there is a difference in how recursion is represented. I am not yet certain as to whether this language will be a real improvement over Lisp in the present system.

The Q, A formalism for problems is fairly general. We can express many kinds of information in that form. This makes it easy to put information into the system.

At first, the problems will all be deterministic: only one correct solution for each problem. Later we will allow several possible solutions to each problem and the system must find probabilities for each of them.

The system starts with equal probability for all primitives, and finds solutions to simple problems by combining them — roughly in order of probability of each string of primitives. Because some trials can take a very long time — occasionally not converging at all — we have to find some way to truncate trials. We use a technique called Levin Search (Lsearch) in which a testing time limit is assigned to a trial proportional to the a priori probability of that trial — which is roughly negative exponential in the number of symbols in the trial description. Short descriptions get lots of time, long descriptions, very little time. Symbolically:

$$T_i = P_i \cdot T$$

P_i is the a priori probability assigned to a trial

T_i is the maximum time allowed for the trial.

T is a constant for each run. we start with T set to the time for about five instructions to be executed.

For a single run, we do all possible trials, using $T_i = P_i \cdot T$: Since $\sum P_i \leq 1$, $\sum T_i \leq T$: the total time for a run is $\leq T$. If we don't find an acceptable trial in that run, we double T and do a new run. These runs continue by doubling T until we find an acceptable solution. It is easy to show that if P_j is the probability of an acceptable solution, and it takes T_j time to generate and test this solution, then the entire search time will be $< 2T_j/P_j$.

After we have solved a fair number of simple problems this way, we no longer assign equal probability to all primitives. We take the entire set of problem solutions as a corpus for any of the (initially) non-recursive prediction methods of Section 1: These methods can be used to assign probabilities to various possible continuations of that corpus of problem solutions— creating candidates for problem solutions. We end up with a much higher probability being assigned to the correct solution, so T_i/P_i for that solution is much smaller, and we take much less time to find it.

At first, we use non-recursive prediction methods, because they are usually faster. Eventually, we will use the recursive methods mentioned in Section 1. Probabilistic Grammar Discovery appears to be very promising. For a particular problem, prediction techniques will be tried in expected T_j/P_j order— small values first. The evaluation of this expected value is a kind of *Metaprediction* problem.

I have not yet programmed much of SA. Schmidhuber, however, has programmed OOPS (Sch 02), which is similar to SA in many ways. OOPS has been able to find a generally recursive solution for the "Tower of Hanoi" problem, after having solved a simpler problem with a recursive solution. Its training sequence is not yet fully developed, and it has a weak update algorithm for the probability distribution that guides its search, but these are deficiencies that are readily repaired (if they have not already been dealt with).

I have described what I expect to be doing in the very near future. In the more distant future, I will extend SA so that is able to solve more general problems than the probabilistic Q, A problems. It will have a large corpus of problem-solving methods. It will learn to assign these methods to new problems and will also learn to expand this corpus of problem-solving methods.

After it has worked a suitable training sequence of relevant problems, we will give it the problem of updating the probability distribution that guides its searches. As we pointed out, this probability distribution is initially generated by a nonrecursive prediction algorithm. By replacing it with an algorithm that can hypothesize any "Turing expressible" function, we expect to get more compression — to enable recognition of regularities of arbitrary structure.

4 Genetic Programming

Koza's Genetic Programming (Koz 99)was the development of an idea of Cramer (Cra 85). In normal Genetic Algorithms, a description of a candidate for solution to a problem will be a finite string. To produce new child candidate strings, we "crossover" two parent strings by interchanging sections of the descriptions of parents. For most description methods, this will often produce meaningless descriptions. Genetic Programming uses Lisp programs to define functions. These programs are all "trees". To create a child from two parent programs, we interchange branches. This assures us that the resultant programs are meaningful. The branches interchanged also have reasonable chances of being useful subfunctions.

Kosa has continued to develop important improvements in the system — such as automatically defining loops, recursions, and iterations to facilitate reuse of code.

The programs have been very successful in solving some really difficult problems. When used for analog circuit design, it discovered many circuits that had previously been patented. Its design for an analog operational amplifier would normally take human engineers several months.

Recently one of its creations was awarded a patent by the US patent office.

While the performance of the program is certainly impressive, I will discuss several serious deficiencies and suggestions on how they may be overcome.

Most notable is the computer resources needed to implement the program. A cluster of 1000 Pentiums operated for about a month to obtain some of the more interesting results.

To understand this difficulty, suppose we were to select a newborn child, and train it for many years in engineering, then have it solve a difficult problem. To solve a new engineering problem, we kill this engineer and start out with a second newborn child as before, train it for many years, then give it a new problem to solve. It would seem more efficient to use the first engineer to solve the second problem, after having solved the first.

In genetic programming, we note that after a program has solved a problem, it has a large population of functions that have been developed. This population contains much potentially useful information — yet to apply it to a new problem is not a trivial task.

Two possible approaches: In the first, we examine a new problem and decide which problems of the past are most similar to it. We then use the final populations of these historical problems, or sampled mixtures of them, as the initial population for the new problem.

How do we find which past problems are related to the new problem? R. Cilibrasi and P. Vitanyi (Cil 05) have been using text compression programs for "clustering". They work something like this: say x is one corpus of text strings and y is another such corpus. We have text string, a. Does it "belong" in set x or set y? To decide, we compress x and y and mixtures of x, y and a.

Say x compresses to $c(x)$ bits. x and a compresses to $c(x, a)$ bits.

Say y compresses to $c(y)$ bits. y and a compresses to $c(y, a)$ bits.

$c(x, a) - c(x)$ is the additional information needed to code a, if x is known. If a is very similar to other strings in x, this additional information will be small. A measure of the relative probability that a belongs in x rather than y, will be $2^{-((c(x,a)-c(x))-(c(y,a)-c(y)))}$.

We can use a technique of this sort to estimate which problems of the past are most like the present new problem. We compare strings of symbols that define each problem.

The efficacy of the method certainly depends on just what compression algorithm is used — on what kinds of features it can recognize, what kind of regularities it can discover. Surprisingly, relatively simple compression algorithms such as PPM and BZIP2 (which is very similar to PPM) were remarkably successful in generating clusters of strings that have proved to be of scientific value.

The second approach does not depend on the agility of a compression scheme: instead, the goal of the genetic program is much modified.

Suppose a_1 is a string that describes our first problem. We then use Genetic Programming to search for a program, f_1 that is able to operate on a_1 to produce an acceptable solution, b_1, i.e. $f_1(a_1) = b_1$. After we have found such a f_1 we go to the second problem, a_2. Starting with the population that solved a_1, we look for a new function f_2 that can solve both a_1 and a_2 — i.e. $f_2(a_1) = b_1$ and $f_2(a_2) = b_2$, the solution to the second problem.

For the third problem, we use as initial population, the population that found f_2. We then search for an f_3 that will solve a_1, a_2 and a_3.

f_3 has to *recognize* as well as *solve* problems — a serious augmentation of Koza's system. We note that these are necessary requirements of human scientists as well — it is not too much to expect of a potentially intelligent machine.

For efficient learning, the problems given to the system would best be in the form of a training sequence. They are designed to introduce various kinds of functions into the system most efficiently. They are similar to the training sequences I'm using in my SA (Scientists Assistant) program.

In normal evolutionary programs, two kinds of modifications of population members are used to create new candidates — the first is mutation — which can be likened to forming a probability distribution from a sample of one. The second technique, crossover, can be likened to using a sample of two. Using larger samples — three, four, five... would enable even better candidates.

A technique has been described that uses the entire population as its sample, to create a distribution over candidates. It uses a probabilistic grammar to model the known population (Pel 00, Sha 03). Candidates can be selected Monte Carlo-wise, using the induced probability distribution.

The speed of the search can be augmented by carefully selecting the sample of the population that is used to generate the grammar. One way to do this: Say we put the population in fitness order, so that x refers to the x^{th} best member of the population. One way to select a good subpopulation is to accept only population members for which

$x < x_0$

where x_0 defines our acceptance threshold. Another way, corresponding to tournament selection, selects them with probability proportional to $e^{-\lambda x}$. Here, λ corresponds to the size of the tournament. Larger λ for large tournaments and correspondingly large bias toward the elite (i.e. small x).

If we use the x_0 threshold method, the parameter x_0, should be selected so as to maximize

$z = \frac{\sigma}{\mu - M_{in}}$

over the population. Here, M_{in} is the fitness function of the best candidate found thus far. μ and σ are the observed mean and standard deviation of fitness in the population being selected. Maximizing z makes it most likely that a randomly generated child will have a fitness function less than M_{in}, the best thus far.

It is also possible to select λ using the same criterion.

One common bottleneck in genetic programming can be excessive time to compute the fitness function. In early work on circuit design, Koza used the "SPICE" program to evaluate fitness of trials. It's evaluation time was *much much* larger than the time needed to select and generate the candidates.

In such a situation, it would be well to try to devise a fast approximate fitness function to use on early generations.

If fitness evaluation time cannot be reduced, then more time should be spent on selecting candidates. Ideally, about the same time should be spent on fitness evaluation as on candidate selection/generation, presuming that the selection time is wisely spent!

The best way to use the time is through better induction models. ALP is designed to give better probability estimates as one spends more time. Other induction systems may have similar characteristics.

I've mentioned a few ways to improve genetic programs. Most of them have been tried to some extent, but not *all together*. I think some of the improvements would work quite synergistically!

5 The Future of A.I.

How far are we from serious A.I.? It is my impression that we are not very far.

Koza's system is very good, and though it is quite slow, there are several techniques for speeding it up and augmenting its functionality.

Another promising system is Schmidhuber's OOPS (Sch 02). It uses Levin search over a Turing complete set of instructions to find solutions to problems, and has been able to find recursive solutions for them. Though it suffers from various deficiencies, most of them can be corrected with techniques that have been already developed in the machine learning community.

In a more general context, we have just about all the needed tools and parts. It remains only to put them together.

Appendix A:

Given a computable probability function μ, I will show how to generate a deterministic sequence (i.e. probabilities are only 0 and 1)

$Z = Z_1 Z_2 Z_3 \cdots$

to which μ gives probabilities that are *extremely bad*: they are always in error by $> .5$.

Let $\mu(Z_{n+1} = 1 | Z_1 \cdots Z_n)$ be μ's estimate that Z_{n+1} will be 1, in view of $Z_1 \cdots Z_n$.

if $\mu(Z_1 = 1 | \bigwedge) < .5$ then $Z_1 = 1$ else $Z_1 = 0$

if $\mu(Z_2 = 1 | Z_1) < .5$ then $Z_2 = 1$ else $Z_2 = 0$

if $\mu(Z_k = 1 | Z_1, Z_2, \cdots Z_{k-1}) < .5$ then $Z_k = 1$ else $Z_k = 0$.

References

[1] (Cil 05) Cilibrasi, R. and Vitányi, P. 2005. Clustering by Compression. *IEEE Transactions on Information Theory* , Vol 51, No. 4.

[2] (Cra 85) Cramer, N.L. 1985. A Representation for the Adaptive Generation of Simple Sequential Programs. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, Carnegie-Mellon University, July 24–26, 1985, J.J. Grefenstette, ed., Lawrence Erlbaum Associates, Hillsdale, N.J., pp. 183-187.
http://www.sover.net/˜nichael/nlc-publications/icga85/index.html

[3] (Hut 02) Hutter, M. 2002. Optimality of Universal Bayesian Sequence Prediction for General Loss and Alphabet.
http://www.idsia.ch/˜marcus/ai/

[4] (Gac 97) Gács, P. 1997, Theorem 5.2.1. in *An Introduction to Kolmogorov Complexity and Its Applications*, Li, M. and Vitányi, P. , Springer-Verlag, N.Y. pp. 328-331.

[5] (Koz 99) Koza, J.R., Bennett III, F.H., Andre, D., Keane, M. 1999. *Genetic Programming III*, Mogan Kaufmann,

[6] (Min 69) Minsky, M.L., and Papert, S. 1969. *Perceptrons: An Introduction to Computational Geometry*, (First Edition), MIT Press, Cambridge, Mass.

[7] (Min 88) Minsky, M.L., and Papert, S. 1969. *Perceptrons: An Introduction to Computational Geometry*, (Expanded Edition), MIT Press, Cambridge, Mass.

[8] (New 57) Newell, A, Shaw, J., and Simon, H. 1963. Empirical Explorations with the Logic Theory Machine. *Proc of the Western Joint Computer Conference*, 15, 218-239. Reprinted in Feigenbaum and Feldman.

[9] (Pel 00) Pelikan, M., Goldberg, D., Lobo, F. 2000. A Survey of Optimization by Building and Using Probabilistic Models. Ill. Genetic Algorithms Laboratory, University of Illinois and Urbana-Champaign.

[10] (Ros 57) Rosenblatt, F. 1957. The Perceptron: A Perceiving and Recognizing Automaton. Report 85-460-1, Project PARA, Cornell Aeronautical Laboratory, Ithaca, N.Y.

[11] (Rum 86) Rumelhart, D. and McClelland, J., *Parallel Distributed Processing*, MIT Press, Cambridge, Mass.

[12] (Sha 03) Shan, Y., McKay, R., Baxter, R, Abbass, H., Essam, D, Nguyen, H. 2003. Grammar Model-Based Program Evolution. Canberra, Australia.

[13] (Sch 02) Schmidhuber, J. 2002. Optimal Ordered Problem Solver. TR IDSIA-12-02, 31 July. http://www.idsia.ch/˜juergen/oops.html

[14] (Sol 60) Solomonoff, R.J. 1960. A Preliminary Report on a General Theory of Inductive Inference. (Revision of Report V–131), Contract AF 49(639)–376, Report ZTB–138, Zator Co., Cambridge, Mass.
http://world.std.com/˜rjs/pubs.html

[15] (Sol 64a) Solomonoff, R.J. 1964. A Formal Theory of Inductive Inference. *Information and Control*, Part I: Vol 7, No. 1, pp. 1–22.
http://world.std.com/˜rjs/pubs.html

[16] (Sol 78) Solomonoff, R.J. 1978. Complexity–Based Induction Systems: Comparisons and Convergence Theorems. *IEEE Trans. on Information Theory*, Vol IT–24, No. 4, pp. 422–432.
http://world.std.com/˜rjs/pubs.html

[17] (Sol 86) Solomonoff, R.J. 1986. The Application of Algorithmic Probability to Problems in Artificial Intelligence. in *Uncertainty in Artificial Intelligence*, Kanal, L.N. and Lemmer, J.F. (Eds), Elsieevr Science Publishers B.V., PP. 473–491.
http://world.std.com/˜rjs/pubs.html

[18] (Sol 89) Solomonoff, R.J. 1989. A System for Incremental Learning Based on Algorithmic Probability. *Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition*, pp. 515–527.
http://world.std.com/˜rjs/pubs.html

[19] (Sol 03a) Solomonoff, R.J. 2003 Progress In Incremental Machine Learning. TR IDSIA-16-03, revision 2.0.
http://world.std.com/˜rjs/pubs.html

[20] (Sol 03b) Solomonoff, R.J. 2003. The Universal Distribution and Machine Learning. *The Computer Journal*, Vol 46, No. 6.
http://world.std.com/˜rjs/pubs.html