Chapter 1
Algorithmic Probability: Theory and Applications

Ray J. Solomonoff

Abstract We first define Algorithmic Probability, an extremely powerful method of inductive inference. We discuss its completeness, incomputability, diversity and subjectivity and show that its incomputability in no way inhibits its use for practical prediction. Applications to Bernoulli sequence prediction and grammar discovery are described. We conclude with a note on its employment in a very strong AI system for very general problem solving.

1.1 Introduction

Ever since probability was invented, there has been much controversy as to just what it meant, how it should be defined and above all, what is the best way to predict the future from the known past. Algorithmic Probability is a relatively recent definition of probability that attempts to solve these problems.

We begin with a simple discussion of prediction and its relationship to probability. This soon leads to a definition of Algorithmic Probability (ALP) and its properties. The best-known properties of ALP are its *incomputibility* and its *completeness* (in that order). *Completeness* means that if there is any regularity (i.e. property useful for prediction) in a batch of data, ALP will eventually find it, using a surprisingly small amount of data. The *incomputability* means that in the search for regularities, at no point can we make a useful estimate of how close we are to finding the most important ones. We will show, however, that this incomputability is of a very benign kind, so that in no way does it inhibit the use of ALP for good prediction. One of the important properties of ALP is *subjectivity*, the amount of personal experiential information that the statistician must put into the system. We will show that this

R.J. Solomonoff
Visiting Professor, Computer Learning Research Centre, Royal Holloway, University of London, London, UK
http://world.std.com/ rjs, e-mail: rjsolo@ieee.org

DOI: 10.1007/978-0-387-84816-7_1,

is a desirable feature of ALP, rather than a "Bug". Another property of ALP is its diversity – it affords *many* explanations of data giving very good understanding of that data.

There have been a few derivatives of Algorithmic Probability – Minimum Message Length (MML), Minimum Description Length (MDL) and Stochastic Complexity – which merit comparison with ALP.

We will discuss the application of ALP to two kinds of problems: Prediction of the Bernoulli Sequence and Discovery of the Grammars of Context Free Languages. We also show how a variation of Levin's search procedure can be used to search over a function space very efficiently to find good predictive models.

The final section is on the future of ALP – some open problems in its application to AI and what we can expect from these applications.

1.2 Prediction, Probability and Induction

What is Prediction?

"An estimate of what is to occur in the future" – But also necessary is a measure of confidence in the prediction: As a negative example consider an early AI program called "Prospector". It was given the characteristics of a plot of land and was expected to suggest places to drill for oil. While it did indeed do that, it soon became clear that without having any estimate of confidence, it is impossible to know whether it is economically feasible to spend $100,000 for an exploratory drill rig. Probability is one way to express this confidence.

Say the program estimated probabilities of 0.1 for 1,000-gallon yield, 0.1 for 10,000-gallon yield and 0.1 for 100,000-gallon yield. The expected yield would be $0.1 \times 1{,}000 + 0.1 \times 10{,}000 + 0.1 \times 100{,}000 = 11{,}100$ gallons. At $100 per gallon this would give $1,110,000. Subtracting out the $100,000 for the drill rig gives an expected profit of $1,010,000, so it *would* be worth drilling at that point. The moral is that predictions by themselves are usually of little value – it is necessary to have confidence levels associated with the predictions.

A strong motivation for revising classical concepts of probability has come from the analysis of human problem solving. When working on a difficult problem, a person is in a maze in which he must make choices of possible courses of action. If the problem is a familiar one, the choices will all be easy. If it is not familiar, there can be much uncertainty in each choice, but choices must somehow be made. One basis for choices might be the probability of each choice leading to a quick solution – this probability being based on experience in this problem and in problems like it. A good reason for using probability is that it enables us to use Levin's Search Technique (Sect. 1.11) to find the solution in near minimal time.

The usual method of calculating probability is by taking the ratio of the number of favorable choices to the total number of choices in the past. If the decision to use integration by parts in an integration problem has been successful in the past 43%

of the time, then its present probability of success is about 0.43. This method has very poor accuracy if we only have one or two cases in the past, and is undefined if the case has never occurred before. Unfortunately it is just these situations that occur most often in problem solving.

On a very practical level: If we cross a particular street 10 times and we get hit by a car twice, we might estimate that the probability of getting hit in crossing that street is about $0.2 = 2/10$. However, if instead, we only crossed that street twice and we didn't get hit either time, it would be unreasonable to conclude that our probability of getting hit was zero! By seriously revising our definition of probability, we are able to resolve this difficulty and clear up many others that have plagued classical concepts of probability.

What is Induction?

Prediction is usually done by finding inductive models. These are deterministic or probabilistic rules for prediction. We are given a batch of data – typically a series of zeros and ones, and we are asked to predict any one of the data points as a function of the data points that precede it.

In the simplest case, let us suppose that the data has a very simple structure:

$$0101010101010\ldots.$$

In this case, a good inductive rule is "zero is always followed by one; one is always followed by zero". This is an example of deterministic induction, and deterministic prediction. In this case it is 100% correct every time!

There is, however, a common kind of induction problem in which our predictions will not be that reliable. Suppose we are given a sequence of zeros and ones with very little apparent structure. The only apparent regularity is that zero occurs 70% of the time and one appears 30% of the time. Inductive algorithms give a probability for each symbol in a sequence that is a function of any or none of the previous symbols. In the present case, the algorithm is very simple and the probability of the next symbol is independent of the past – the probability of zero seems to be 0.7; the probability of one seems to be 0.3. This kind of simple probabilistic sequence is called a "Bernoulli sequence". The sequence can contain many different kinds of symbols, but the probability of each is independent of the past. In Sect. 1.9 we will discuss the Bernoulli sequence in some detail.

In general we will not always be predicting Bernoulli sequences and there are many possible algorithms (which we will call "models") that tell how to assign a probability to each symbol, based on the past. Which of these should we use? Which will give good predictions in the future?

One desirable feature of an inductive model is that if it is applied to the *known* sequence, it produces good predictions. Suppose R_i is an inductive algorithm. R_i predicts the probability of an symbol a_j in a sequence $a_1, a_2 \cdots a_n$ by looking at the previous symbols: More exactly,

$$p_j = R_i(a_j|a_1.a_2 \cdots a_{j-1})$$

a_j is the symbol for which we want the probability. $a_1, a_2 \cdots a_{j-1}$ are the previous symbols in the sequence. Then R_i is able to give the probability of a particular value of a_j as a function of the past. Here, the values of a_j can range over the entire "alphabet" of symbols that occur in the sequence. If the sequence is a binary, a_j will range over the set 0 and 1 only. If the sequence is English text, a_j will range over all alphabetic and punctuation symbols. If R_i is a good predictor, for most of the a_j, the probability it assigns to them will be large – near one.

Consider S, the product of the probabilities that R_i assigns to the individual symbols of the sequence, $a_1, a_2 \cdots a_n$. S will give the probability that R_i assigns to the sequence as a whole

$$S = \prod_{j=1}^{n} R_i(a_j|a_1, a_2 \cdots a_n) = \prod_{j=1}^{n} p_j \,. \tag{1.1}$$

For good prediction we want S as large as possible. The maximum value it can have is one, which implies perfect prediction. The smallest value it can have is zero – which can occur if one or more of the p_j are zero – meaning that the algorithm predicted an event to be impossible, yet that event occurred!

The "Maximum Likelihood" method of model selection uses S only to decide upon a model. First, a set of models is chosen by the statistician, based on his experience with the kind of prediction being done. The model within that set having maximum S value is selected.

Maximum Likelihood is very good when there is a lot of data – which is the area in which classical statistics operates. When there is only a small amount of data, it is necessary to consider not only S, but the effect of the likelihood of *the model itself* on model selection. The next section will show how this may be done.

1.3 Compression and ALP

An important application of symbol prediction is text compression. If an induction algorithm assigns a probability S to a text, there is a coding method – *Arithmetic Coding* – that can re-create the entire text without error using just $-\log_2 S$ bits.

More exactly: Suppose x is a string of English text, in which each character is represented by an 8-bit ASCII code, and there are n characters in x. x would be directly represented by a code of just $8n$ bits. If we had a prediction model, R, that assigned a probability of S to the text, then it is possible to write a sequence of just $-\log_2 S$ bits, so that the original text, x, can be recovered from that bit sequence without error.

If R is a string of symbols (usually a computer program) that describes the prediction model, we will use $|R|$ to represent the length of the shortest binary sequence that describes R. If $S > 0$, then the probability assigned to the text will be in two parts: the first part is the code for R, which is $|R|$ bits long, and the second part is the code for the probability of the data, as given by R – it will be just $-\log_2 S$ bits in length. The sum of these will be $|R| - \log_2 S$ bits. The compression ratio achieved by R would be

$$\frac{8N}{|R| - \log_2 S}$$

PPM, a commonly used prediction algorithm, achieves a compression of about three for English text. For very large strings of data, compressions of as much as six have been achieved by highly refined prediction algorithms. We can use $|R| - \log_2 S$ bits, the length of the compressed code, as a "figure of merit" of a particular induction algorithm with respect to a particular text.

We want an algorithm that will give good prediction, i.e. large S, and small $|R|$, so $|R| - \log_2 S$, the figure of merit, will be as small as possible and the probability it assigns to the text will be as large as possible. Models with $|R|$ larger than optimum are considered to be *overfitted*. Models in which $|R|$ are smaller than optimum are considered to be *underfitted*. By choosing a model that minimizes $|R| - \log_2 S$, we avoid both underfitting and overfitting, and obtain very good predictions. We will return to this topic later, when we tell how to compute $|R|$ and S for particular models and data sets.

Usually there are many inductive models available. In 1960, I described Algorithmic Probability – ALP [5–7], which uses all possible models in parallel for prediction, with weights dependent upon the figure of merit of each model.

$$P_M(a_{n+1}|a_1,a_2\cdots a_n) = \sum 2^{-|R_i|}\, S_i\, R_i(a_{n+1}|a_1,a_2\cdots a_n) \tag{1.2}$$

$P_M(a_{n+1}|a_1,a_2\cdots a_n)$ is the probability assigned by ALP to the $(n+1)$th symbol of the sequence, in view of the previous part of the sequence.

$R_i(a_{n+1}|a_1,a_2\cdots a_n)$ is the probability assigned by the ith model to the $(n+1)$th symbol of the sequence, in view of the previous part of the sequence.

S_i is the probability assigned by R_i, (the ith model) to the known sequence, $a_1,a_2\cdots a_n$ via (1.1).

$2^{-|R_i|}S_i$ is $1/2$ with an exponent equal to the figure of merit that R_i has with respect to the data string $a_1,a_2\ldots a_n$. It is the weight assigned to $R_i(\ )$. This weight is large when the figure of merit is good – i.e. small.

Suppose that $|R_i|$ is the shortest program describing the ith model using a particular "reference computer" or programming language – which we will call M. Clearly the value of $|R_i|$ will depend on the nature of M. We will be using machines (or languages) that are "Universal" – machines that can readily program any conceivable function – almost all computers and programming languages are of this kind. The subscript M in P_M expresses the dependence of ALP on choice of the reference computer or language.

The universality of M assures us that the value of ALP will not depend *very* much on just which M we use – but the dependence upon M is nonetheless important. It will be discussed at greater length in Sect. 1.5 on "Subjectivity".

Normally in prediction problems we will have some time limit, T, in which we have to make our prediction. In ALP what we want is a set of models of maximum total weight. A set of this sort will give us an approximation that is as close as possible to ALP and gives best predictions. To obtain such a set, we devise a search technique that tries to find, in the available time, T, a set of Models, R_i, such that the total weight,

$$\sum_i 2^{-|R_i|} S_i \tag{1.3}$$

is as large as possible.

On the whole, ALP would seem to be a complex, time consuming way to compute probabilities – though in fact, if suitable approximations are used, these objections are not at all serious.

Does ALP have any advantages over other probability evaluation methods? For one, it's the only method known to be *complete*. The completeness property of ALP means that if there is any regularity in a body of data, our system is guaranteed to discover it using a relatively small sample of that data. More exactly, say we had some data that were generated by an *unknown* probabilistic source, P. Not knowing P, we use instead, P_M to obtain the Algorithmic Probabilities of the symbols in the data. How much do the symbol probabilities computed by P_M differ from their true probabilities, P? The Expected value with respect to P of the total square error between P and P_M is bounded by $-1/2\, lnP_0$.

$$E_P\left[\sum_{m=1}^{n}(P_M(a_{m+1}=1|a_1,a_2\cdots a_m)-P(a_{m+1}=1|a_1,a_2\cdots a_m))^2\right] \leq -\frac{1}{2}\, lnP_0$$
$$lnP_0 \approx kln2 \tag{1.4}$$

P_0 is the a priori probability of P. It is the probability we would assign to P if we knew P.

k is the *Kolmogorov complexity* of the data generator, P. It's the shortest binary program that can describe P, the generator of the data.

This is an extremely small error rate. The error in probability approaches zero more rapidly than $1/n$. Rapid convergence to correct probabilities is a most important feature of ALP. The convergence holds for any P that is describable by a computer program and includes many functions that are formally *incomputable*. Various kinds of functions are described in the next section. The convergence proof is in Solomonoff [8].

1.4 Incomputability

It should be noted that in general, it is impossible to find the truly best models with any certainty – there is an infinity of models to be tested and some take an unacceptably long time to evaluate. At any particular time in the search, we will know the best ones so far, but we can't ever be sure that spending a little more time will not give *much* better models! While it is clear that we can always make approximations to ALP by using a limited number of models, we can never know how close these approximations are to the "True ALP". ALP is indeed, *formally incomputable.*

In this section, we will investigate how our models are generated and how the incomputability comes about – why it is a *necessary, desirable feature* of any high performance prediction technique, and how this incomputability *in no way* inhibits its use for practical prediction.

How Incomputability Arises and How We Deal with It

Recall that for ALP. we added up the predictions of all models, using suitable weights:

$$P_M = \sum_{i=1}^{\infty} 2^{-|R_i|}\, S_i R_i\,. \tag{1.5}$$

Here R_i gives the probability distribution for the next symbol as computed by the ith model. Just what do we mean by these models, R_i?

There are just four kinds of functions that R_i can be:

1. Finite compositions of a finite set of functions
2. Primitive recursive functions
3. Partial recursive functions
4. Total recursive functions

Compositions are combinations of a small set of functions. The finite power series

$$3.2 + 5.98 * X - 12.54 * X^2 + 7.44 * X^3$$

is a composition using the functions *plus* and *times* on the real numbers. Finite series of this sort can approximate any continuous functions to arbitrary precision.

Primitive Recursive Functions are defined by one or more *DO* loops. For example to define $Factorial(X)$ we can write

```
$Factorial(0) \leftarrow 1$
$DO \quad I = 1, X$
$Factorial(I) \leftarrow I * Factorial(I-1)$
$EndDO$
```

Partial Recursive Functions are definable using one or more *WHILE* loops. For example, to define the factorial in this way:

```
$Factorial(0) \leftarrow 1$
$I \leftarrow 0$
$WHILE\ I \neq X$
$I \leftarrow I+1\ Factorial(I) \leftarrow I * Factorial(I-1)$
$EndWHILE$
```

The loop will terminate if X is a non negative integer. For all other values of X, the loop will run forever. In the present case it is easy to tell for which values of X the loop will terminate.

A simple *WHILE* loop in which it is *not* so easy to tell:

```
$WHILE\ X > 4$
$IF\ X/2$ is an integer $THEN\ X \leftarrow X/2$
$ELSE\ X \leftarrow 3 * X + 1$
$EndWHILE$
```

This program has been tested with X starting at all positive integers up to more than sixty million. The loop has always terminated, but no one yet is certain as to whether it terminates for *all* positive integers!

For any *Total Recursive Function* we know *all* values of arguments for which the function has values. Compositions and primitive recursive functions are all total recursive. Many partial recursive functions are total recursive, but some are not. As a consequence of the insolvability of Turing's "Halting Problem", it will sometimes be impossible to tell if a certain *WHILE* loop will terminate or not.

Suppose we use (1.2) to approximate ALP by sequentially testing functions in a list of all possible functions – these will be the partial recursive functions because this is the only *recursively enumerable* function class that includes *all* possible predictive functions. As we test to find functions with good figures of merit (small $(|R_i| - \log_2 S_i)$) we find that certain of them don't converge after say, a time T, of 10 s. We know that if we increase T enough, eventually, all converging trials will converge and all divergent trials will still diverge – so eventually we will get close to true ALP – but we cannot recognize when this occurs. Furthermore for any finite T, we cannot *ever* know a useful upper bound on how large the error in the ALP approximation is. That is why this particular method of approximating ALP is called "incomputable". Could there be another *computable* approximation technique that *would* converge? It is easy to show that any computable technique cannot be "complete" – i.e. having very small errors in probability estimates.

Consider an arbitrary *computable* probability method, R_0. We will show how to generate a sequence for which R_0's errors in probability would *always* be 0.5 or more. We start our sequence with a single bit, say zero. We then ask R_0 for the most probable next bit. If it says "one is more probable", we make the continuation zero, if it says "zero is more probable", we make the next bit one. If it says "both are equally likely" we make the next bit zero. We generate the third bit in the sequence in the same way, and we can use this method to generate an arbitrarily long continuation of the initial zero.

For this sequence, R_0 will always have an error in probability of at least one half. Since *completeness* implies that prediction errors approach zero for all finitely describable sequences, it is clear that R_0 or any other *computable* probability method *cannot* be *complete*. Conversely, any *complete* probability method, such as ALP, *cannot* be *computable*.

If we cannot compute ALP, what good is it? It would seem to be of little value for prediction! To answer this objection, we note that from a practical viewpoint, we *never* have to calculate ALP exactly – we can *always* use approximations. While it is impossible to know how close our approximations are to the true ALP, *that information is rarely needed for practical induction.*

What we actually need for practical prediction:

1. Estimates of how good a particular approximation will be in future problems (called "Out of Sample Error")
2. Methods to search for good models
3. Quick and simple methods to compare models

For 1., we can use *Cross Validation* or *Leave One Out* – well-known methods that work with most kinds of problems. In addition, because ALP does not overfit or underfit there is usually a better method to make such estimates.

For 2. In Sect. 1.11 we will describe a variant of Levin's Search Procedure, for an efficient search of a very large function space.

For 3., we will always find it easy to compare models via their associated "Figures of Merit", $|R_i| - \log_2(S_i)$.

In summary, it is clear that all computable prediction methods have a serious flaw – they cannot ever approach completeness. On the other hand, while approximations to ALP *can* approach completeness, we can never know how close we are to the final, incomputable result. We can, however, get good estimates of the future error in our approximations, and *this is all that we really need* in a practical prediction system.

That our approximations approach ALP assures us that if we spend enough time searching we will eventually get as little error in prediction as is possible. *No computable probability evaluation method can ever give us this assurance.* It is in this sense that the incomputability of ALP is a *desirable* feature.

1.5 Subjectivity

The subjectivity of probability resides in a priori information – the information available to the statistician before he sees the data to be extrapolated. This is independent of what kind of statistical techniques we use. In ALP this a priori information is embodied in M, our "Reference Computer". Recall our assignment of a $|R|$ value to an induction model – it was the length of the program necessary to describe R. In general, this will depend on the machine we use – its instruction set. Since the machines we use are Universal – they can imitate one another – the length of description of programs will not vary widely between most reference machines we might consider. But nevertheless, using small samples of data (as we often do in AI), these differences between machines can modify results considerably.

For quite some time I felt that the dependence of ALP on the reference machine was a serious flaw in the concept, and I tried to find some "objective" universal device, free from the arbitrariness of choosing a particular universal machine. When I thought I finally found a device of this sort, I realized that I really didn't want it – that I had no use for it at all! Let me explain:

In doing inductive inference, one begins with two kinds of information: First, the data itself, and second, the a priori data – the information one had before seeing the data. It is possible to do prediction without data, but one cannot do prediction without a priori information. In choosing a reference machine we are given the opportunity to insert into the a priori probability distribution any information about the data that we know before we see it.

If the reference machine were somehow "objectively" chosen for all induction problems, we would have no way to make use of our prior information. This lack of an objective prior distribution makes ALP very subjective – as are all Bayesian systems.

This certainly makes the results "subjective". If we value objectivity, we can routinely reduce the choice of a machine and representation to certain universal "default" values – but there is a tradeoff between objectivity and accuracy. To obtain the best extrapolation, we must use whatever information is available, and much of this information may be subjective.

Consider two physicians, A and B: A is a conventional physician: He diagnoses ailments on the basis of what he has learned in school, what he has read about and his own experience in treating patients. B is not a conventional physician. He is "objective". His diagnosis is entirely "by the book" – things he has learned in school that are universally accepted. He tries as hard as he can to make his judgements free of any bias that might be brought about by his own experience in treating patients. As a lawyer, I might prefer defending B's decisions in court, but as a patient, I would prefer A's intelligently biased diagnosis and treatment.

To the extent that a statistician uses objective techniques, his recommendations may be easily defended, but for accuracy in prediction, the additional information afforded by subjective information can be a critical advantage.

Consider the evolution of a priori in a scientist during the course of his life. He starts at birth with minimal a priori information – but enough to be able to learn to walk, to learn to communicate and his immune system is able to adapt to certain hostilities in the environment. Soon after birth, he begins to solve problems and incorporate the problem solving routines into his a priori tools for future problem solving. This continues throughout the life of the scientist – as he matures, his a priori information matures with him.

In making predictions, there are several commonly used techniques for inserting a priori information. First, by restricting or expanding the set of induction models to be considered. This is certainly the commonest way. Second, by selecting prediction functions with adjustable parameters and assuming a density distribution over those parameters based on past experience with such parameters. Third, we note that much of the information in our sciences is expressed as definitions – additions to our language. ALP, or approximations of it, avails itself of this information by using these definitions to help assign code lengths, and hence a priori probabilities to models. Computer languages are usually used to describe models, and it is relatively easy to make arbitrary definitions part of the language.

More generally, modifications of computer languages are known to be able to express any conceivable a priori probability distributions. This gives us the ability to incorporate whatever a priori information we like into our computer language. It is certainly more general than any of the other methods of inserting a priori information.

1.6 Diversity and Understanding

Apart from accuracy of probability estimate, ALP has for AI another important value: Its multiplicity of models gives us many different ways to understand our data.

A very conventional scientist understands his science using a *single* "current paradigm" – the way of understanding that is most in vogue at the present time. A more creative scientist understands his science in *very many* ways, and can more easily create new theories, new ways of understanding, when the "current paradigm" no longer fits the current data.

In the area of AI in which I'm most interested – Incremental Learning – this diversity of explanations is of major importance. At each point in the life of the System, it is able to solve with acceptable merit, all of the problems it's been given thus far. We give it a new problem – usually its present Algorithm is adequate. Occasionally, it will have to be modified a bit. But every once in a while it gets a problem of real difficulty and the present Algorithm has to be *seriously revised*. At such times, we try using or modifying *once sub-optimal algorithms*. If that doesn't work we can use parts of the sub-optimal algorithms and put them together in new ways to make new trial algorithms. It is in giving us a broader basis to learn from the past, that this value of ALP lies.

1.6.1 ALP and "The Wisdom of Crowds"

It is a characteristic of ALP that it averages over all possible models of the data: There is evidence that this kind of averaging may be a good idea in a more general setting. "The Wisdom of Crowds" is a recent book by James Serowiecki that investigates this question. The idea is that if you take a bunch of very different kinds of people and ask them (independently) for a solution to a difficult problem, then a suitable average of their solutions will very often be better than the best in the set. He gives examples of people guessing the number of beans in a large glass bottle, or guessing the weight of a large ox, or several more complex, very difficult problems.

He is concerned with the question of what kinds of problems can be solved this way as well as the question of when crowds are wise and when they are stupid. They become very stupid in mobs or in committees in which a single person is able to strongly influence the opinions in the crowd. In a wise crowd, the opinions are individualized, the needed information is shared by the problem solvers, and the individuals have great diversity in their problem solving techniques. The methods of combining the solutions must enable each of the opinions to be voiced. These conditions are very much the sort of thing we do in ALP. Also, when we *approximate* ALP we try to preserve this diversity in the *subset* of models we use.

1.7 Derivatives of ALP

After my first description of ALP in 1960 [5], there were several related induction models described, minimum message length (MML) Wallace and Boulton [13], Minimum Description Length (MDL) Rissanen [3], and Stochastic Complexity, Rissanen [4]. These models were conceived independently of ALP – (though Rissanen had read Kolmogorov's 1965 paper on minimum coding [1], which is closely related to ALP). MML and MDL select induction models by minimizing the figure of merit, $|R_i| - \log_2(S_i)$ just as ALP does. However, instead of using a weighted sum of models, they use only the single best model.

MDL chooses a space of computable models then selects the best model from that space. This avoids any incomputability, but greatly limits the kinds of models that it can use. MML recognizes the incomputability of finding the best model so it is in principle much stronger than MDL. Stochastic complexity, like MDL, first selects a space of computable models – then, like ALP it uses a weighted sum of all models in the that space. Like MDL, it differs from ALP in the limited types of models that are accessible to it. MML is about the same as ALP when the best model is much better than any other found. When several models are of comparable figure of merit, MML and ALP will differ. One advantage of ALP over MML and MDL is in its diversity of models. This is useful if the induction is part of an ongoing process of learning – but if the induction is used on one problem only, diversity is of much less value. Stochastic Complexity, of course, does obtain diversity in its limited set of models.

1.8 Extensions of ALP

The probability distribution for ALP that I've shown is called "The Universal Distribution for *sequential* prediction". There are two other universal distributions I'd like to describe:

1.8.1 A Universal Distribution for an Unordered Set of Strings

Suppose we have a corpus of n unordered discrete objects, each described by a finite string a_j: Given a new string, a_{n+1}, what is the probability that it is in the previous set? In MML and MDL, we consider various algorithms, R_i, that assign probabilities to strings. (We might regard them as *Probabilistic Grammars*). We use for prediction, the grammar, R_i, for which

$$|R_i| - \log_2 S_i \tag{1.6}$$

is minimum. Here $|R_i|$ is the number of bits in the description of the grammar, R_i. $S_i = \prod_j R_i(a_j)$ is the probability assigned to the entire corpus by R_i. If R_k is the best stochastic grammar that we've found, then we use $R_k(a_{n+1})$ as the probability of a_{n+1}. To obtain the ALP version, we simply sum over all models as before, using weights $2^{-|R_i|}S_i$.

This kind of ALP has an associated convergence theorem giving very small errors in probability. This approach can be used in linguistics. The a_j can be examples of sentences that are grammatically correct. We can use $|R_i| - \log_2 S_i$ as a likelihood that the data was created by grammar, R_i. Section 1.10 continues the discussion of Grammatical Induction.

1.8.2 A Universal Distribution for an Unordered Set of Ordered Pairs of Strings

This type of induction includes almost all kinds of prediction problems as "special cases". Suppose you have a set of question answer pairs, $Q_1,A_1;Q_2,A_2;\ldots Q_n,A_n$: Given a new question, Q_{n+1}, what is the probability distribution over possible answers, A_{n+1}? Equivalently, we have an unknown analog and/or digital transducer, and we are given a set of input/output pairs $Q_1,A_1;\ldots$ – For a new input Q_i, what is probability distribution on outputs? Or, say the Q_i are descriptions of mushrooms and the A_i are whether they are poisonous or not.

As before, we hypothesize operators $R_j(A|Q)$ that are able to assign a probability to any A given any Q: The ALP solution is

$$\begin{aligned} P(A_{n+1}|Q_{n+1}) &= \sum_j 2^{-|R_j|} \prod_{i=1}^{n+1} R_j(A_i|Q_i) \\ &= \sum_j a_n^j R_j(A_{n+1}|Q_{n+1}), \\ a_n^j &= 2^{-|R_j|} \prod_{i=1}^{n} R_j(A_i|Q_i). \end{aligned} \tag{1.7}$$

$2^{-|R_j|}$ are the a priori probabilities associated with the R_j.

a_n^j is the weight given to R_j's predictions in view of it's success in predicting the data set $Q_1,A_1 \ldots Q_n,A_n$.

This ALP system has a corresponding theorem for small errors in probability. As before, we try to find a set of models of maximum weight in the available time. Proofs of convergence theorems for these extensions of ALP are in Solomonoff [10]. There are corresponding MDL, MML versions in which we pick the single model of maximum weight.

1.9 Coding the Bernoulli Sequence

First, consider a binary Bernoulli sequence of length n. It's only visible regularity is that zeroes have occurred n_0 times and ones have occurred n_1 times. One kind of model for this data is that the probability of 0 is p and the probability of 1 is $1-p$. Call this model R_p. S_p is the probability assigned to the data by R_p

$$S_p = p^{n_0}(1-p)^{n_1} \ . \tag{1.8}$$

Recall that ALP tells us to sum the predictions of each model, with weight given by the product of the a priori probability of the model $(2^{-|R_i|})$ and S_i, the probability assigned to the data by the model ..., i.e.:

$$\sum_i 2^{-|R_i|} \, S_i \, R_i(\,) \ . \tag{1.9}$$

In summing we consider all models with $0 \le p \le 1$.

We assume for each model, R_p, precision Δ in describing p. So p is specified with accuracy, Δ. We have $\frac{1}{\Delta}$ models to sum so total weight is 1

$$2^{-|R_i|} = \Delta,$$

$$S_i = S_p = p^{n_0}(1-p)^{n_1},$$

$$R_i(\,) = R_p = p.$$

Summing the models for small Δ gives the integral

$$\int_0^1 p^{n_0}(1-p)^{n_1} p \, dp = \int_0^1 p^{n_0+1}(1-p)^{n_1} dp. \tag{1.10}$$

This integral can be evaluated using the Beta function, $B(\ ,\ )$

$$\int_0^1 p^x(1-p)^y dy = B(x+1, y+1) = \frac{x!\,y!}{(x+y+1)!}. \tag{1.11}$$

So our integral of (1.10) equals $\frac{(n_0+1)!\,n_1!}{(n_0+n_1+2)!}$.

We can get about the same result another way: The function $p^{n_0}(1-p)^{n_1}$ is (if n_0 and n_1 are large), narrowly peaked at $p_0 = \frac{n_0}{n_0+n_1}$. If we used MDL we would use the model with $p = p_0$. The a priori probability of the model itself will depend on how accurately we have to specify p_0. If the "width" of the peaked distribution is Δ, then the a priori probability of model M_{p_0} will be just $\Delta \cdot p_0^{n_0}(1-p_0)^{n_1}$.

It is known that the width of the distribution is just $2\sqrt{\frac{p_0(1-p_0)}{n_0+n_1+1}}$.[1] As a result the probability assigned to this model is $\sqrt{\frac{p_0(1-p_0)}{n_0+n_1+1}} \cdot p_0^{n_0}(1-p_0)^{n_1} \cdot 2$. If we use Sterling's approximation for $n!$ ($n! \approx e^{-n}n^n\sqrt{2\pi n}$), it is not difficult to show that

$$\frac{n_0!n_1!}{(n_0+n_1+1)!} \approx p_0^{n_0}(1-p_0)^{n_1}\sqrt{\frac{p_0(1-p_0)}{n_0+n_1+1}} \cdot \sqrt{2\pi}, \tag{1.12}$$

$\sqrt{2\pi} = 2.5066$ which is roughly equal to 2.

To obtain the probability of a zero following a sequence of n_0 zeros and n_1 ones: We divide the probability of the sequence having the extra zero, by the probability of the sequence without the extra zero, i.e.:

$$\frac{(n_0+1)!n_1!}{(n_0+n_1+2)!} \Big/ \frac{n_0!n_1!}{(n_0+n_1+1)!} = \frac{n_0+1}{(n_0+n_1+2)}. \tag{1.13}$$

This method of extrapolating binary Bernoulli sequences is called "Laplace's Rule". The formula for the probability of a binary sequence, $\frac{n_0!n_1!}{(n_0+n_1+1)!}$ can be generalized for an alphabet of k symbols.

A sequence of k different kinds of symbols has a probability of

$$\frac{(k-1)!\prod_{i=1}^{k} n_i!}{(k-1+\sum_{i=1}^{k} n_i)!}. \tag{1.14}$$

n_i is the number of times the ith symbol occurs.

This formula can be obtained by integration in a $k-1$ dimensional space of the function $p_1^{n_1}p_2^{n_2}\cdots p_{k-1}^{n_{k-1}}(1-p_1-p_2\cdots-p_{k-1})^{n_k}$.

Through an argument similar to that used for the binary sequence, the probability of the next symbol being of the jth type is

$$\frac{n_j+1}{k+\sum_{i=1}^{k} n_i}. \tag{1.15}$$

A way to visualize this result: the body of data (the "corpus") consists of the $\sum n_i$ symbols. Think of a "pre-corpus" containing one of each of the k symbols. If we think of a "macro corpus" as "corpus plus pre-corpus" we can obtain the probability of the next symbol being the jth one by dividing the number of occurrences of that symbol in the macro corpus, by the total number of symbols of all types in the macro corpus.

[1] This can be obtained by getting the first and second moments of the distribution, using the fact that $\int_0^1 p^x(1-p)^y dp = \frac{x!y!}{(x+y+1)!}$.

It is also possible to have different numbers of each symbol type in the pre-corpus, enabling us to get a great variety of “a priori probability distributions” for our predictions.

1.10 Context Free Grammar Discovery

This is a method of extrapolating an unordered set of finite strings: Given the set of strings, $a_1, a_2, \cdots a_n$, what is the probability that a new string, a_{n+1}, is a member of the set? We assume that the original set was generated by some sort of probabilistic device. We want to find a device of this sort that has a high a priori likelihood (i.e. short description length) and assigns high probability to the data set. A good model R_i, is one with maximum value of

$$P(R_i) \prod_{j=1}^{n} R_i(a_j). \tag{1.16}$$

Here $P(R_i)$ is the a priori probability of the model R_i. $R_i(a_j)$ is the probability assigned by R_i to data string, a_j.

To understand *probabilistic* models, we first define *non-probabilistic* grammars. In the case of context free grammars, this consists of a set of *terminal* symbols and a set of symbols called *nonterminals*, one of which is the initial starting symbol, S.

A grammar could then be:

$$\begin{aligned} S &\to Aac \\ S &\to BaAd \\ A &\to BAaS \\ A &\to AB \\ A &\to a \\ B &\to aBa \\ B &\to b \end{aligned}$$

The capital letters (including S) are all *nonterminal* symbols. The lower case letters are all *terminal* symbols. To generate a legal string, we start with the symbol, S, and we perform either of the two possible substitutions. If we choose $BaAd$, we would then have to choose substitutions for the nonterminals B and A. For B, if we chose aBa we would again have to make a choice for B. If we chose a terminal symbol, like b for B, then no more substitutions can be made.

An example of a string generation sequence:

S, $BaAd$, $aBaaAd$, $abaaAd$, $abaaABd$, $abaaaBd$, $abaaabd$.

The string $abaaabd$ is then a legally derived string from this grammar. The set of all strings legally derivable from a grammar is called the *language* of the grammar. The language of a grammar can contain a finite or infinite number of strings. If we replace the deterministic substitution rules with probabilistic rules, we have a *probabilistic* grammar. A grammar of this sort assigns a probability to every string it can generate. In the deterministic grammar above, S had two rewrite choices, A had three, and B had two. If we assign a probability to each choice, we have a probabilistic grammar.

Suppose S had substitution probability 0.1 for Aac and 0.9 for $BaAd$. Similarly, assigning probabilities 0.3, 0.2 and 0.5 for A's substitutions and 0.4, 0.6 for B's substitutions.

$$\begin{array}{ll} S & 0.1\ Aac \\ & 0.9\ BaAd \\ A & 0.3\ BAaS \\ & 0.2\ AB \\ & 0.5\ a \\ B & 0.4\ aBa \\ & 0.6\ b \end{array}$$

In the derivation of *abaaab* of the previous example, the substitutions would have probabilities 0.9 to get $BaAd$, 0.4 to get $aBaaAd$, 0.6 to get $abaaAd$, 0.2 to get $abaaABd$, 0.5 to get $abaaaBd$, and 0.6 to get $abaaabd$. The probability of the string $abaabd$ being derived this way is $0.9 \times 0.4 \times 0.6 \times 0.2 \times 0.5 \times 0.6 = 0.01296$. Often there are other ways to derive the same string with a grammar, so we have to add up the probabilities of all of its possible derivations to get the total probability of a string.

Suppose we are given a set of strings, *ab*, *aabb*, *aaabbb* that were generated by an unknown grammar. How do we find the grammar?

I wouldn't answer that question directly, but instead I will tell how to find a sequence of grammars that fits the data progressively better. The best one we find may not be the true generator, but will give probabilities to strings close to those given by the generator.

The example here is that of A. Stolcke's, PhD thesis [12]. We start with an ad hoc grammar that *can* generate the data, but it $overfits$ . . . it is too complex:

$$\begin{array}{ll} S & \rightarrow ab \\ & \rightarrow aabb \\ & \rightarrow aaabbb \end{array}$$

We then try a series of modifications of the grammar (*Chunking* and *Merging*) that increase the total probability of description and thereby decrease total description length. *Merging* consists of replacing two nonterminals by a single nonterminal. *Chunking* is the process of defining new nonterminals. We try it when a string or substring has occurred two or more times in the data. *ab* has occurred three times so we define $X = ab$ and rewrite the grammar as

$$\begin{aligned} S &\to X \\ &\to aXb \\ &\to aaXbb \\ X &\to ab \end{aligned}$$

aXb occurs twice so we define $Y = aXb$ giving

$$\begin{aligned} S &\to X \\ &\to Y \\ &\to aYb \\ X &\to ab \\ Y &\to aXb \end{aligned}$$

At this point there are no repeated strings or substrings, so we try the operation *Merge* which coalesces two nonterminals. In the present case merging S and Y would decrease complexity of the grammar, so we try:

$$\begin{aligned} S &\to X \\ &\to aSb \\ &\to aXb \\ X &\to ab \end{aligned}$$

Next, merging S and X gives

$$\begin{aligned} S &\to aSb \\ &\to ab \end{aligned}$$

which is an adequate grammar. At each step there are usually several possible *chunk* or *merge* candidates. We chose the candidates that give minimum description length to the resultant grammar.

How do we calculate the length of description of a grammar and its description of the data set?

Consider the grammar

$$\begin{aligned} S &\to X \\ &\to Y \\ &\to aYb \\ X &\to ab \\ Y &\to aXb \end{aligned}$$

There are two kinds of terminal symbols and three kinds of nonterminals. If we know the number of terminals and nonterminals, we need describe only the right hand side of the substitutions to define the grammar. The names of the nonterminals (other than the first one, S) are not relevant. We can describe the right hand side by the string $Xs_1Ys_1aYbs_1s_2abs_1s_2aXbs_1s_2$. s_1 and s_2 are punctuation symbols. s_1 marks the end of a string. s_2 marks the end of a sequence of strings that belong to the same nonterminal. The string to be encoded has seven kinds of symbols. The number of times each occurs: X, 2; Y, 2; S, 0; a, 3; b, 3; s_1, 5; s_2, 3. We can then use the formula

$$\frac{(k-1)!\prod_{i=1}^{k} n_i!}{(k-1+\sum_{i=1}^{k} n_i)!} \tag{1.17}$$

to compute the probability of the grammar: $k=7$, since there are seven symbols and $n_1=2$, $n_2=2$, $n_3=0$, $n_4=3$, etc. We also have to include the probability of 2, the number of kinds of terminals, and of 3, the number of kinds of nonterminals.

There is some disagreement in the machine learning community about how best to assign probability to integers, n. A common form is

$$P(n) = A2^{-\log_2^* n}\,, \tag{1.18}$$

where $\log_2^* n = \log_2 n + \log_2\log_2 n + \log_2\log_2\log_2 n \cdots$ taking as many positive terms as there are, and A is a normalization constant. There seems to be no good reason to choose 2 as the base for logs, and using different bases gives much different results. If we use natural logs, the sum diverges.

This particular form of $P(n)$ was devised by Rissanen. It is an attempt to approximate the shortest description of the integer n, e.g. the Kolmogorov complexity of n. Its first moment is infinite, which means it is very biased toward large numbers. If we have reason to believe, from previous experience, that n will not be very large, but will be about λ, then a reasonable form of $P(n)$ might be $P(n) = Ae^{-n/\lambda}$, A being a normalization constant.

The forgoing enables us to evaluate $P(R_i)$ of (1.16). The $\prod_{j=1}^{n} R_i(a_j)$ part is evaluated by considering the choices made when the grammar produces the data corpus. For each nonterminal, we will have a sequence of decisions whose probabilities can be evaluated by an expression like (1.14), or perhaps the simpler technique of (1.15) that uses the "pre-corpus". Since there are three nonterminals, we need the product of three such expressions.

The process used by Stolcke in his thesis was to make various trials of chunking or merging in attempts to successively get a shorter description length – or to increase (1.16). Essentially a very greedy method. He has been actively working on Context Free Grammar discovery since then, and has probably discovered many improvements. There are many more recent papers at his website.

Most, if not all CFG discovery has been oriented toward finding a *single best grammar*. For applications in AI and genetic programming it is useful to have large sets of *not necessarily best* grammars – giving much needed diversity. One way to implement this: At each stage of modification of a grammar, there are usually several different operations that can reduce description length. We could pursue such paths in parallel ... perhaps retaining the best 10 or best 100 grammars thus far.

Branches taken early in the search could lead to very divergent paths and much needed diversity. This approach helps avoid *local optima* in grammars and *premature convergence* when applied to Genetic Programming.

1.11 Levin's Search Technique

In the section on incomputability we mentioned the importance of good search techniques for finding effective induction models. The procedure we will describe was inspired by Levin's search technique [2], but is applied to a different kind of problem.

Here, we have a corpus of data to extrapolate, and we want to search over a function space, to find functions ("models") $R_i(\ )$ such that $2^{-|R_i|}S_i$ is as large as possible. In this search, for some R_i, the time needed to evaluate S_i, (the probability assigned to the corpus by R_i), may be unacceptably large – possibly infinite.

Suppose we have a (deterministic) context free grammar, G, that can generate strings that are programs in some computer language. (Most computer languages have associated grammars of this kind.) In generating programs, the grammar will have various choices of substitutions to make. If we give each substitution in a k-way choice, a probability of $1/k$, then we have a *probabilistic* grammar that assigns *a priori probabilities* to the programs that it generates. If we use a functional language (such as LISP), this will give a probability distribution over all functions it can generate. The probability assigned to the function R_i will be denoted by $P_M(R_i)$. Here M is the name of the functional computer language. $P_M(R_i)$ corresponds to what we called $2^{-|R_i|}$ in our earlier discussions. $|R_i|$ corresponds to $-\log_2 P_M(R_i)$. As before, S_i is the probability assigned to the corpus by R_i. We want to find functions $R_i(\ )$ such that $P_M(R_i)S_i$ is as large as possible.

Next we choose a small initial time T – which might be the time needed to execute 10 instructions in our Functional Language. The initial T is not critical. We then compute $P_M(R_i)S_i$ for all R_i for which $t_i/P_M(R_i) < T$. Here t_i is the time needed to construct R_i and evaluate its S_i.

There are only a finite number of R_i that satisfy this criterion and if T is very small, there will be very few, if any. We remember which R_i's have large $P_M(R_i)S_i$.

$t_i < T \cdot P_M(R_i)$, so $\sum_i t_i$, the total time for this part of the search takes time $< T \cdot \sum_i P_M(R_i)$. Since the $P_M(R_i)$ are a priori probabilities, $\sum_i P_M(R_i)$ must be less than or equal to 1, and so the total time for this part of the search must be less than T.

If we are not satisfied with the R_i we've obtained, we double T and do the search again. We continue doubling T and searching until we find satisfactory R_i's or until we run out of time. If T' is the value of T when we finish the search, then the total time for the search will be $T' + T'/2 + T'/4 \cdots \approx 2T'$.

If it took time t_j to generate and test one of our "good" models, R_j, then when R_j was discovered, T' would be no more than $2t_j/P_M(R_j)$ – so we would take no more time than twice this, or $4t_j/P_M(R_j)$ to find R_j. Note that this time limit depends on R_j only, and is independent of the fact that we may have aborted the S_i evaluations of many R_i for which t_i was infinite or unacceptably large. This feature of Levin Search is a mandatory requirement for search over a space of partial recursive functions. Any weaker search technique would seriously limit the power of the inductive models available to us.

When ALP is being used in AI, we are solving a sequence of problems of increasing difficulty. The machine (or language) M is periodically "updated" by inserting subroutines and definitions, etc., into M so that the solutions, R_i to problems in the past result in larger $P_M(R_j)$. As a result the $t_j/P_M(R_j)$ are smaller – giving quicker solutions to problems of the past – and usually for problems of the future as well.

1.12 The Future of ALP: Some Open Problems

We have described ALP and some of its properties:

First, its *completeness*: Its remarkable ability to find any irregularities in an apparently small amount of data.

Second: That any *complete* induction system like ALP must be formally *incomputable*.

Third: That this *incomputability* imposes no limit on its use for practical induction. This fact is based on our ability to estimate the future accuracy of any particular induction model. While this seems to be easy to do in ALP without using Cross Validation, more work needs to be done in this area.

ALP was designed to work on difficult problems in AI. The particular kind of AI considered was a version of "Incremental Learning": We give the machine a simple problem. Using Levin Search, it finds one or more solutions to the problem. The system then updates itself by modifying the reference machine so that the solutions found will have higher a priori probabilities. We then give it new problems somewhat similar to the previous problem. Again we use Levin Search to find solutions – We continue with a sequence of problems of increasing difficulty, updating after each solution is found. As the training sequence continues we expect that we will need less and less care in selecting new problems and that the system will eventually be able to solve a large space of very difficult problems. For a more detailed description of the system, see Solomonoff [11].

The principal things that need to be done to implement such a system:

* We have to find a good reference language: Some good candidates are APL, LISP, FORTH, or a subset of Assembly Language. These languages must be augmented with definitions and subroutines that we expect to be useful in problem solving.
* The design of good training sequences for the system is critical for getting much problem-solving ability into it. I have written some general principles on how to do this [9], but more work needs to be done in this area. For early training, it might be useful to learn definitions of instructions from Maple or Mathematica. For more advanced training we might use the book that Ramanujan used to teach himself mathematics – "A Synopsis of Elementary Results in Pure and Applied Mathematics" by George S. Carr.

* We need a good update algorithm. It is possible to use PPM, a relatively fast, effective method of prediction, for preliminary updating, but to find more complex regularities, a more general algorithm is needed. The universality of the reference language assures us that any conceivable update algorithm can be considered. APL's diversity of solutions to problems maximizes the information that we are able to insert into the a priori probability distribution. After a suitable training sequence the system should know enough to usefully work on the problem of updating itself.

Because of ALP's *completeness* (among other desirable properties), we expect that the *complete* AI system described above should become an extremely powerful general problem solving device – going well beyond the limited functional capabilities of current *incomplete* AI systems.

References

1. Kolmogorov, A.N.: Three approaches to the quantitative definition of information. Problems of Information Transmission **1**(1), 1–7 (1965)
2. Levin, L.A.: Universal search problems. Problemy Peredaci Informacii **9**, 115–116 (1973); Translated in Problems of Information Transmission **9**, 265–266
3. Rissanen, J.: Modeling by the shortest data description. Automatica **14**, 465–471 (1978)
4. Rissanen, J.: Stochastical Complexity and Statistical Inquiry. World Scientific, Singapore (1989)
5. Solomonoff, R.J.: A preliminary report on a general theory of inductive inference. (Revision of Report V–131, Feb. 1960), Contract AF 49(639)–376, Report ZTB–138. Zator, Cambridge (Nov, 1960) (http://www.world.std.com/˜rjs/pubs.html)
6. Solomonoff, R.J.: A formal theory of inductive inference, Part I. Information and Control **7**(1), 1–22 (1964)
7. Solomonoff, R.J.: A formal theory of inductive inference, Part II. Information and Control **7**(2), 224–254 (1964)
8. Solomonoff, R.J.: Complexity-based induction systems: comparisons and convergence theorems. IEEE Transactions on Information Theory **IT–24**(4), 422–432 (1978)
9. Solomonoff, R.J.: A system for incremental learning based on algorithmic probability. In: Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition 515–527 (Dec. 1989)
10. Solomonoff, R.J.: Three kinds of probabilistic induction: universal distributions and convergence theorems. Appears in Festschrift for Chris Wallace (2003)
11. Solomonoff, R.J.: Progress in incremental machine learning. TR IDSIA-16-03, revision 2.0. (2003)
12. Stolcke, A.: On learning context free grammars. PhD Thesis (1994)
13. Wallace, C.S and Boulton, D.M.: An information measure for classification. Computer Journal **11**, 185–195 (1968)

Further Reading

Cover, T. and Thomas, J.: Elements of Information Theory. Wiley, New York (1991) – Good treatments of statistics, predictions, etc.

Li, M. and Vitányi, P.: An Introduction to Kolmogorov Complexity and Its Applications. Springer, New York (1993) (1997) – Starts with elementary treatment and development. Many sections very clear, very well written. Other sections difficult to understand. Occasional serious ambiguity of notation (e.g. definition of "enumerable"). Treatment of probability is better in 1997 than in 1993 edition.

Shan, Y., McKay, R.I., Baxter, R., et al.: Grammar Model-Based Program Evolution. (Dec. 2003) A recent review of work in this area, and what looks like a very good learning system. Discusses mechanics of fitting Grammar to Data, and how to use Grammars to guide Search Problems.

Solomonoff, R.J.: The following papers are all available at the website: world.std.com/ rjs/ pubs.html.

Stolcke, A., Omohundro, S.: Inducing Probabilistic Grammars by Bayesian Model Merging. ICSI, Berkeley (1994) This is largely a summary of Stolcke's: On Learning Context Free Grammars [12].

A Preliminary Report on a General Theory of Inductive Inference. (1960).

A Formal Theory of Inductive Inference. Information and Control, Part I. (1964).

A Formal Theory of Inductive Inference, Part II. (June 1964) – Discusses fitting of context free grammars to data. Most of the discussion is correct, but Sects. 4.2.4 and 4.3.4 are questionable and equations (49) and (50) are incorrect.

A Preliminary Report ... and A Formal Theory ... give some intuitive justification for the way ALP does induction.

The Discovery of Algorithmic Probability. (1997) – Gives heuristic background for discovery of ALP. Page 27 gives a time line of important publications related to development of ALP.

Progress in Incremental Machine Learning; Revision 2.0. (Oct. 30, 2003) – A more detailed description of the system I'm currently working on. There have been important developments since, however.

The Universal Distribution and Machine Learning. (2003) – Discussion of irrelevance of incomputability to applications for prediction. Also discussion of subjectivity.