Lecture 2: Applications of Algorithmic Probability

Ray J. Solomonoff

rjsolo@ieee.org http://world.std.com/˜rjs/

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 give 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.

Next, I want to discuss several applications of ALP, as well as several other developments related to machine learning. In the last lecture I will show how to use these ideas to create a very good learning machine.

2 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 M_p.

P_{M_p} is the probability assigned to the data by M_p.

$$P_{M_p} = p^{n_0}(1-p)^{n_1} \tag{1}$$

Let us consider all models M_p with $0 \leq p \leq 1$ We want to sum their probabilities: This gives us

$$\int_0^1 p^{n_0}(1-p)^{n_1}\,dp \tag{2}$$

It is given by

$$B(n_0+1, n_1+1) = \frac{n_0!n_1!}{(n_0+n_1+1)!} \tag{3}$$

Here $B(\cdot,\cdot)$ is the Beta function.

In summing the M_p's, we can define $M_{p\Delta}$ to have precision Δ in specifying p. This means we need $-log_2\Delta$ bits to describe it and its a priori probability is $1/\Delta$. If Δ is very small, then when we add all of the $\Delta p^{n_0}(1-p)^{n_1}$'s together we get the integral above.

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 wanted to use MDL we would use the model with $p = p_0$. The cost of the model itself will depend on how accurately we have to specify p_0. If the "width" of the peaked distribution is w, then the "probability cost" of model M_{p_0} will be just $\frac{1}{w}$.

It is known that the half width of the distribution is just $\sqrt{\frac{p(1-p)}{n_0+n_1+1}}$.[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 $x!$, it is not difficult to show that

$$\frac{n_0!n_1!}{(n_0+n_1+1)!} \approx p_0^{n_0}(1-p_0)^{n_1}\sqrt{\frac{p_0(1-p_0)}{n_0+n_1+1}} \cdot \sqrt{2\pi} \tag{4}$$

$\sqrt{2\pi} = 2.25066$ which is approximately equal to 2.

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

$$\frac{(k-1)!\prod_{i=1}^{k} n_i!}{(k-1+\sum_{i=1}^{k} n_i)!} \tag{5}$$

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

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

3 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, b, 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 M_i, is one with maximum value of

$$P(M_i) \prod_{j=1}^{n} M_i(a_j) \tag{6}$$

Here $P(M_i)$ is the a priori probability of the model M_i.

$M_i(a_j)$ is the probability assigned by M_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:

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

The capital letters (including S) are all non-terminal symbols.

The lower case letters are all terminals.

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 non-terminals B and A.

For B, if we chose aBA we would again have to make choices for B and A. 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 associates a probability with every finite string. 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 .1 for Aac and .9 for $BaAd$. Similarly, assigning probabilities .3, .2 and .5 for A's substitutions and .4, .9 for B's substitutions.

$$\begin{array}{ccl} S & .1 & Aac \\ & .9 & BaAd \\ A & .3 & BAaS \\ & .2 & AB \\ & .5 & a \\ B & .4 & aBa \\ & .9 & b \end{array}$$

In the derivation of *abaaab* of the previous example, the substitutions would have probabilities .9 to get $BaAd$, .4 to get $aBaaAd$, .9 to get $abaaAd$, .2 to get $abaaABd$, .5 to get $abaaaBd$, and .9 to get $abaaabd$.

The probability of the string *abaabd* being derived this way is $.9 \times .4 \times .9 \times .2 \times .5 \times .9 = .02916$. Often there are other ways to derive the same string with this 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, 1994.

We start with an ad hoc grammar that *can* generate the data, but it $overfits$ . . . it is too complex:

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

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 non-terminals by a single non-terminal. $Chunking$ is the process of defining new non-terminals. 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

$$\begin{array}{rcl} S & \rightarrow & X \\ & \rightarrow & aXb \\ & \rightarrow & aaXbb \\ x & \rightarrow & ab \end{array}$$

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

$$\begin{array}{rcl} S & \rightarrow & X \\ & \rightarrow & Y \\ & \rightarrow & aYb \\ X & \rightarrow & ab \\ Y & \rightarrow & aXb \end{array}$$

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

$$\begin{array}{rcl} S & \rightarrow & X \\ & \rightarrow & aSb \\ & \rightarrow & aXb \\ X & \rightarrow & ab \end{array}$$

Next, merging S and X gives

$$\begin{array}{rcl} S & \rightarrow & aSb \\ & \rightarrow & ab \end{array}$$

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

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

There are two kinds of terminal symbols and three kinds of non-terminals.

If we know the number of terminals and non-terminals, we need describe only the right hand side of the substitutions to define the grammar. The names of the non-terminals (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 non-terminal. The string to be encoded has 7 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

$$\frac{(k-1)!\prod_{i=1}^{k} n_i!}{(k-1+\sum_{i=1}^{k} n_i)!} \tag{7}$$

to compute the probability of the grammar: $k = 7$, since there are 7 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 non-terminals.

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

$$P(N) = A2^{-log_2^* n} \tag{8}$$

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 $P(n) = A\alpha^n$ where α and A are arranged so that the expected value of n is λ.

The forgoing enables us to evaluate $P(M_i)$ of equation 1. The $\prod_{j=1}^{n} M_i(a_j)$ part is evaluated by considering the choices made when the grammar produces the data corpus. For each non-terminal, we will have a sequence of decisions whose probabilities can be evaluated by an expression like equation 7. Since there are three non-terminals, we need the product of three such expression.

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 equation 6 — 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 of CFG discovery has been oriented toward finding a *single best grammar*. For applications in A.I. and genetic programming it is useful to have large set 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.

Papers relevant to Lecture II: (All are available on the net)

1. R. Solomonoff, "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 Sections 4.2.4 and 4.3.4 are questionable and equations 49 and 50 are incorrect.

2. Andreas Stolcke, "On Learning Context Free Grammars", PhD Thesis, 1994. Much detailed discussion.

3. A. Stolcke, S. Omohundro, "Inducing Probabilistic Grammars by Bayesian Model Merging", 1994. This is largely a summary of (2).

4. Y. Shan, R.I. McKay, R. Baxter 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.

5. Sepp Hochreiter, Juergen Schmidhuber, "Flat Minimum Search Finds Simple Nets", Dec. 1994. Application of MDL to selection of neural nets for good prediction capability.