AutoBayes: A System for Synthesizing Data Analysis Programs

Bernd Fischer, Johann Schumann
RIACS / NASA Ames, Moffett Field, CA 94035, USA
{fisch,schumann}@ptolemy.arc.nasa.gov

1 Introduction

Statistical approaches to data analysis, which use methods from probability theory and numerical analysis, are well-founded but difficult to implement: the development of a statistical data analysis program for any given application is time-consuming and requires knowledge and experience in several areas. AutoBayes [BFP99, FSP00] is a fully automatic high-level generator system for data analysis programs from statistical models which aims to overcome these barriers. AutoBayes follows the schema-based deductive approach to program synthesis. This means that the programs are constructed by the instantiation of generic algorithm schemas, e.g., EM. This process is supported by logic-based deduction to ensure the consistency between specification (i.e., statistical model) and synthesized program. AutoBayes uses a textual notation which is based on graphical models (more precisely, Bayesian networks) to specify the statistical models. Currently, it synthesizes optimized and fully commented C/C++ code which can be linked dynamically into the Matlab and Octave environments.

In our opinion, the synthesis approach to generate and then compile source-level programs offers several advantages over the use of libraries or direct interpretation of Bayesian networks. Synthesis is easier to use and provides more flexibility than libraries. It works fully automatically and, hence, the user does not need to know about any details of the library functions. Also, the algorithm schemas can be combined during synthesis in more flexible ways than it is possible with traditional libraries. Furthermore, AutoBayes model specifications are much more declarative than programs and thus easier to understand, communicate, and validate. Synthesized code is compiled and thus more efficient than interpretated general-purpose routines. Moreover, if a data analysis problem (or subproblem) can be solved in closed form, a synthesis system can in many cases symbolically generate this closed form, yielding a further speed-up over the usually iterative routines employed in interpreters.

In the following, we present a small example showing the basic features of the model input language. Then we describe the system architecture of AutoBayes and the different layers of synthesis schemas. For further system details see [BFP99, FSP00].

2 An Example: Mixture of Gaussians

Figure 1 shows the well-known finite mixture of Gaussians model in AutoBayes's specification language. The model (called "Mixture of Gaussians" – line 1) assumes that each of the data points (there are `n_points` – line 3) belongs to one of `n_classes` classes; here `n_classes` has been set to three (line 5), but `n_points` is left unspecified. Lines 15 and 16 declare the input vector and distributions for the data points[1]. Each point `x(I)` is drawn from a Gaussian distribution `c(I)` with mean `mu(c(I))` and standard deviation `sigma(c(I))`. The unknown distribution parameters can be different for each class; hence, we declare these values as vectors (line 10). The unknown assignment of the points to the classes (i.e., distributions) is represented by the hidden (i.e., not observable) variable `c`; the class probabilities or relative frequencies are given by the also unknown vector `rho` (lines 8–13). Since each point belongs to one of the classes, the sum of the probabilities `rho(I)` must be equal to one (line 9). Additional constraints (lines 4,6) express further basic assumptions. Finally, we specify the goal inference task (line 18), maximizing the probability `pr(x | rho(I), mu(I), sigma(I))`. Due to Bayes' rule, this calculates the most likely values of the parameters of interest, `rho(I), mu(I)`, and `sigma(I)`.

```
 1  model mog as 'Mixture of Gaussians';
 2
 3  const nat n_points as 'number of data points'
 4        where 0 < n_points;
 5  const nat n_classes := 3 as 'number of classes'
 6        where n_classes << n_points;
 7
 8  double rho(0..n_classes - 1) as 'class probabilites'
 9        where 1 = sum(idx(I, 0, n_classes - 1), rho(I));
10  double mu(0..n_classes - 1), sigma(0..n_classes - 1);
11
12  nat c(0..n_points) as 'class assignment vector';
13  c(_) ~ discrete(rho);
14
15  data double x(0..n_points - 1) as 'data points (known)';
16  x(I) ~ gauss(mu(c(I)),sigma(c(I)));
17
18  max pr(x | {rho,mu,sigma}) wrt {rho, mu, sigma};
```

Figure 1: AutoBayes-specification for the mixture of Gaussians example. Line numbers have been added for reference in the text. Keywords are underlined.

3 System Architecture

AutoBayes takes a model specification as shown in Figure 1 and synthesizes executable C or C++ code from it. The main steps of this process and the corresponding components can be seen in Figure 2. Although the input specification is in a

[1] Vector indices start with 0 in a C/C++ style.

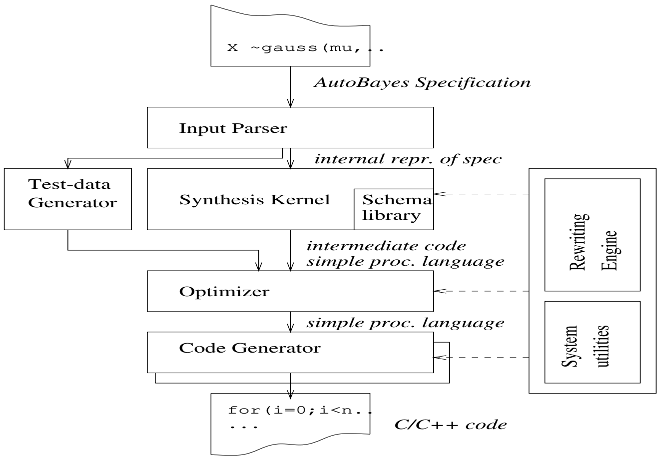

Figure 2: System architecture for AutoBayes.

textual form, AutoBayes internally works on Bayesian networks; its input parser automatically extracts an equivalent Bayesian network from any given specification.

Synthesis proceeds by exhaustive, layered application of *schemas*. A schema consists of a program fragment with open slots and a set of applicability conditions. The slots are filled in with code pieces by the synthesis system. The conditions ultimately constrain how the slots can be filled; they must be discharged (i.e., proven to hold in the given model) by the synthesis before the schema can be applied. Conditions can also be described by specific network patterns; checking then proceeds efficiently by pattern matching. This allows the network structure to guide the application of the schemas and thus to prevent combinatorial explosion of the search space, even if a large number of schemas is available.

AutoBayes currently comprises four different layers of schemas; new schemas can easily be added without restructuring the system. Network decomposition schemas try to break down the network into independent subnets, based on the independence theorems in probability theory. These subnets are fed back into the synthesis process and the resulting programs can be composed to achieve a program for the original problem. AutoBayes is thus able to automatically synthesize large programs by composition of different schemas. Formula and vector decomposition schemas work on complex formulas, e.g., products of conditional probability distributions. The application of these schemas is also guided by the network structure but it requires substantial symbolic computations. This capability is provided by a rewriting engine and different symbolic simplifiers and solvers. The skeleton of the synthesized code is generated by the application of statistical algorithm schemas. AutoBayes currently implements two such schemas, the EM-algorithm and k-Means (i.e., nearest neighbor clustering). After this last network-oriented layer, all probabilities have been converted into an atomic form which can be eliminated by instantiating the appropriate probability density functions. The statistical problem is thus converted into an ordinary numerical optimization problem which can be solved symbolically or numerically. The symbolic algebra kernel allows us to find closed-form solution for many text-book examples; the naive use of commercial tools like Mathematica could compromise the logical soundness of the synthesis system which could then result in incorrect programs. For the numeric solution of optimization problem, AutoBayes currently provides schemas for the Newton-Raphson and Nelder-Mead simplex algorithms. These schemas are instantiated with the function to be optimized. In contrast to using a library function, this open approach allows further symbolic simplifications and optimizations.

All algorithm schemas yield code in a target-independent intermediate language. Figure 3 shows an excerpt for the intermediate code generated for the mixture example. This synthesized code is optimized and finally converted into the language of the target system. AutoBayes can currently generate dynamically linkable C/C++ code for the Matlab and Octave environments, respectively. By using an intermediate language we were able to cleanly separate the synthesis kernel from any target-specific aspects. The system can currently produce programs of up to approximately 1000 lines of C++ code from 25 lines of specification in less than 1/2 minute, thus providing up to 40:1 leverage.

The entire system is implemented in SWI Prolog[2] and comprises more than 20k lines of code. A sampling data generator has been added to AutoBayes. Given a model specification it synthesizes code for generating synthetic random data according to the distributions given in the model. This sampling generator is convenient for debugging, testing and performance tuning purposes.

4 Conclusions and Future Work

Currently, AutoBayes can synthesize code for a number of standard textbook examples and mixture problems; in fact, we have been able to "automate" most examples from [EH81]. We will extend AutoBayes in different areas. Obviously, its statistical capabilities grow with the number of implemented schemas. We will add more sophisticated numeric solvers as well as schemas to deal with time-series data, e.g., generalized verisions of Kalman filtering and smoothing. We will also add on-line algorithm schemas which enable synthesizing code that does not require that all data are stored in memory at once. Such schemas are especially important for large data sets or embedded applications. The integration of more schemas on all layers and the improvement of the built-in deductive techniques will enable us to generate different solutions for a single problem. Then, users can pick the most suitable one, based on their specific requirements, e.g., speed or memory size.

References

[BFP99] W. L. Buntine, B. Fischer, and T. Pressburger. "Towards Automated Synthesis of Data Mining Programs". In S. Chaudhuri and D. Madigan, (eds.), *Proc. 5th Intl. Conf. Knowledge Discovery and Data Mining*, pp. 372–376, San Diego, CA, August 15–18 1999. ACM Press.

[2] `http://swi.psy.uva.nl/projects/SWI-Prolog`

[EH81] B. S. Everitt and D. J. Hand. *Finite Mixture Distributions.* Monographs on Applied Probability and Statists. Chapman & Hall, London, 1981.

[FSP00] B. Fischer, J. Schumann, and T. Pressburger. "Generating Data Analysis Programs from Statistical Models (Position Paper)". In W. Taha, (ed.), *Proc. Intl. Workshop Semantics Applications, and Implementation of Program Generation, Lect. Notes Comp. Sci.* **1924**, pp. 212–229, Montreal, Canada, September 2000. Springer.

```
// Mixture of Gaussians
proc(mog) {
const:  int n_classes := 3;                  // Number of classes
        int n_points := size(x, 1);          // Number of data points
input:  double x[0:n_points - 1];
output: double mu[0:n_classes-1],rho[0:n_classes-1],sigma[0:n_classes-1];
local: ...
{ ...
  // Initialization
  // Randomize the hidden variable c
  for( [idx(pv64, 0, n_points - 1)])
    c(pv64) := random_int(0, n_classes - 1);
  // Initialize the local distribution; the initialization is "sharp",
  // i.e., q1 is set to zero almost everywhere and to one at the index
  // positions determined by the initial values of the hidden variable.
  for( [idx(pv154, 0, n_points - 1), idx(pv155, 0, n_classes - 1)])
    q1(pv154, pv155) := 0;
  for( [idx(pv156, 0, n_points - 1)])
    q1(pv156, c(pv156)) := 1;
  // EM-loop
  while( converging([vector([idx(pv157, 0, n_classes-1)], rho(pv157)),
            vector([idx(pv158, 0, n_classes-1)], mu(pv158)),
            vector([idx(pv159, 0, n_classes-1)], sigma(pv159))]) )
    {
      // Decomposition I;
      // the problem to optimize the conditional probability
      // pr([c, x] | [rho, mu, sigma]) w.r.t. the variables rho, mu,
      // and sigma can under the given dependencies by Bayes rule be
      // decomposed into independent subproblems.
         ...
      // using the Lagrange-multiplier l1.
      l1 := sum([idx(pv68, 0, n_classes - 1)],
                sum([idx(pv66, 0, n_points - 1)], q1(pv66, pv68)));
      for( [idx(pv68, 0, n_classes - 1)])
        rho(pv68) := l1 ** -1 * sum([idx(pv66, 0, n_points - 1)],
                     q1(pv66, pv68));
      // The conditional probability pr([x] | [sigma, mu, c]) is
      // under the given dependencies by Bayes rule equivalent to
      // prod([idx(pv126, 0, n_points-1)],
      //      pr([x(pv126)] | [c(pv126), mu, sigma]))
      // The probability occuring here is atomic and can be
      // replaced by the respective probability density function.
      ...
      for( [idx(pv64, 0, n_points-1), idx(pv65, 0, n_classes-1)])
        q1(pv64, pv65) := select(norm([idx(pv163, 0, n_classes-1)],
                 exp(-1 / 2 * (-1 * mu(pv163) + x(pv64)) ** 2 *
                 sigma(pv163) ** -2) * rho(pv163) * 2 ** (-1 / 2) *
                 pi ** (-1 / 2) * sigma(pv163) ** -1), [pv65]);
    } } }
```

Figure 3: Pseudo-code for the Mixture of Gaussians example (excerpts).