AutoBayes Program Synthesis System —Users Manual—

Johann Schumann
Hamed Jafari
Tom Pressburger
Ewen Denney
Wray Buntine
Bernd Fischer

NASA Ames Research Center
August 2008

National Aeronautics and
Space Administration

Preface

Program synthesis is the systematic, automatic construction of efficient executable code from high-level declarative specifications. AutoBayes is a fully automatic program synthesis system for the statistical data analysis domain; in particular, it solves parameter estimation problems. It has seen many successful applications at NASA and is currently being used, for example, to analyze simulation results for Orion. The input to AutoBayes is a concise description of a data analysis problem composed of 1) a parameterized statistical model and 2) a goal that is a probability term involving parameters and input data. The output of AutoBayes is optimized and fully-documented C/C++ code that, given input data, computes values for those parameters that maximize the probability term. Parameter estimation, clustering, and change point detection type statistical analysis problems can be described in this fashion. The output code can be linked dynamically into Matlab™ [1], Octave, and other environments. AutoBayes uses Bayesian Networks internally to decompose complex statistical models and to derive algorithms for their solution. Its powerful symbolic system enables AutoBayes to solve many subproblems symbolically rather than having to rely on numeric approximation algorithms, thus yielding effective, efficient, and compact code. AutoBayes makes statistical analysis faster and more reliable, because effort can be focused on model development and validation rather than manual development of solution algorithms and code, which instead AutoBayes handles automatically.

[1] Matlab™ is a trademark of Mathworks, Inc.

Contents

List of Figures

List of Tables

Listings

1. Introduction

Data analysis is the transformation of raw data (i.e., pure numbers) into a more abstract form, e.g., summarizing a set of measurements by their mean value and standard deviation as a bare minimum. For most data analysis tasks—especially tasks involving large data sets—computer support is necessary. AutoBayes is a program generator that generates statistical-method based scientific data analysis programs.

The input to AutoBayes is a problem specification as you can see in Figure 1.1. This problem specification is a concise description of a data analysis problem; in fact, a parameter estimation problem. The specification is in the form of a statistical model with declarations, constraints, and a maximum-likelihood (ML) or maximum a posteriori (MAP) goal. AutoBayes internally uses Bayesian Networks to decompose complex statistical models and to derive algorithms for the solution. AutoBayes's output is optimized and fully documented C/C++ code which can then be linked dynamically into Matlab$^{\mathrm{TM}}$, Octave, or other environments. Input data is then given to the generated program to obtain analysis results in the form of estimated parameters.

1.1 Key Features

AutoBayes offers several unique features which result from using program synthesis technology and which make it more powerful and more versatile for statistical analysis than other tools and statistical libraries.

- AutoBayes generates efficient procedural code from a high-level, declarative specification; the specification does not involve algorithmic information such as data or control flow.
- Changes to the statistical model can be made without time consuming re-implementation of a data analysis program.
- The automatically-generated code that solves the statistical problem is efficient and accurate, because it does not rely on numeric approximations when it can solve problems symbolically.
- AutoBayes can generate different programs for the same application, each embodying differing solution algorithms or algorithm variants.

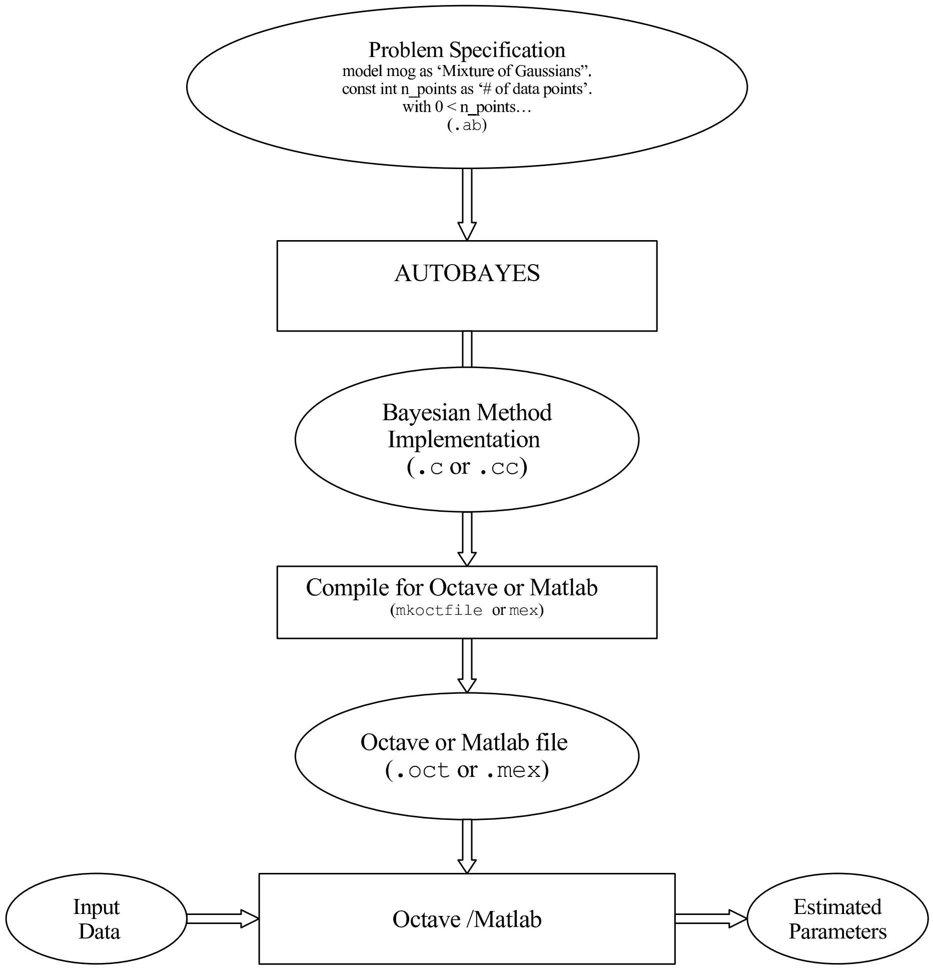


Figure 1.1: AutoBayes system architecture

- With AutoBayes's test data generator, the user is able to generate synthetic data according to the given model specification. This feature can be used to debug and validate the statistical model and to select a synthesized program which best fits the given application profile.

The rest of this chapter discusses some applications that AutoBayes has been used for, giving a sense of its capabilities and breadth of applications. Chapter 2 describes how to install AutoBayes. Chapter 3 shows the step-by-step application of AutoBayes to solve the classic problem of clustering Fisher data about Iris flowers. This chapter explains the clustering problem specification, the commands to invoke AutoBayes and link it to Matlab^{TM}, and the results of the analysis. Chapter 4 briefly describes how AutoBayes works internally; the following chapter describes the kinds of algorithms (e.g., clustering and maximization algorithms) that AutoBayes can embed in its generated code. Chapter 6 describes AutoBayes commands, files, and flags. Chapter 7 defines the problem specification language. We have found that a specification for a new problem is most easily constructed by basing it on existing specifications, so chapter 8 discusses many problems and their specifications, as well as descriptions of the derivation of algorithms constructed by AutoBayes to solve those problems. Appendix A describes command line options accepted by AutoBayes that affect its behavior. Appendix B acknowledges the creators of AutoBayes and NASA support. Lastly, there is a bibliography and an index of terms.

1.2 Applications of AutoBayes

The AutoBayes system has been successfully applied to a wide field of different projects and tasks within NASA. In this section, we will briefly highlight some of the applications in order to give a glimpse of the system's capabilities and applicabilities.

1.2.1 Data Analysis on Large Software Simulations

The analysis of large and complex parameterized software systems, e.g., understanding the results of simulations of aerospace systems, is very complicated and time-consuming due to the large parameter space and the complex, highly-coupled non-linear nature of the different system components. At NASA Ames, tools were developed to facilitate the systematic exploration of parameter spaces. The tools use a unique combination of Monte Carlo, n-factor combinatorial parameter variations, and model-based generation of simulation and test cases [GBSMB08]. These cases (often up to 10,000 of them) are executed by the Trick simulation environment [Vet07], usually generating huge amounts of data.

In order to study the structure of these multivariate data and the dependency on the parameters, AutoBayes clustering models are being used to group these large data sets. A synergistic combination with the machine learning tool TAR3 then facilitates comprehensive root cause analysis. This tool is being used for abort and re-entry scenario analysis for Orion, as well as for a small-satellite guidance, navigation, and control system. Figure 1.2 shows results of a computational model of an earth-based small-satellite simulator entitled the "Hover Test Vehicle" (HTV). The first figure shows the expected variation in trajectory relative to various input parameters (such as center of gravity, mass of fuel, etc.). The clusters were ranked according to their landing velocity and position, with colors ranging from Blue (best outcomes) to Red (worst outcomes) The second plot shows the relationship between landing velocity and the initial wet mass of the vehicle. Through the use of this data, the flight rules for a successful flight were defined. The flight test of the vehicle performed as suggested by the data.

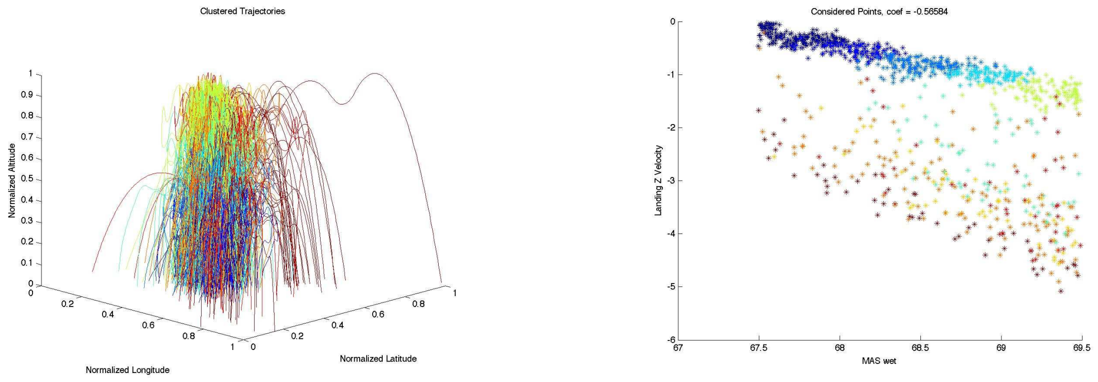


Figure 1.2: HTV trajectories colored according to their class (left). Relationship between landing velocity and wet mass (right).

1.2.2 Data Analysis for Air Traffic Control Data

Modern air traffic control (ATC) systems have to deal with a densely crowded airspace. In order to avoid conflicts and mid-air collisions, air traffic control systems, such as the CTAS (Center Tracon Advisory System), which has been developed at NASA Ames, include algorithms to predict the aircraft's trajectories for several minutes into the future. Of course, the models underlying these algorithms must be as accurate as possible, despite many unknowns and high noise.

AutoBayes was used for a study to perform data mining on actual aircraft trajectories (data from ATC and radar). One task was to find the transition point between a constant CAS (calibrated airspeed) and a constant mach climb, which occurs in typical trajectories. Figure 1.3 illustrates such a scenario. The altitude of the aircraft over time is shown as well as the development of CAS and mach speed. Given the CAS and mach speed over time the task is to find the most likely transition point (vertical line). A simple AutoBayes specification solves this task (for details, see Section 8.3.3). We assume linear, but noisy behavior of the speeds. Thus, the CAS trajectory can be defined by

$$cas_t = \begin{cases} cas_0 & \text{for } t \leq t_0 \\ cas_0 - cas_r(t - t_0) & \text{for } t > t_0 \end{cases}$$

with unknown parameters cas_0, cas_r, and the transition point t_0. The same can be set up for the mach number. If we assume that the actual data are noisy and Gaussian distributed, it is easy to write a statistical specification to estimate the unknown parameters. Figure 1.4 shows the actual result of the AutoBayes analysis. The code generated by AutoBayes has reliably estimated the unknown parameters and the transition point (blue and green). For this analysis, large data sets consisting of more than 10,000 climb scenarios have been analyzed, resulting in figures like Figure 1.5 which shows the most likely mach numbers and altitudes of the transition for a specific type of jet aircraft.

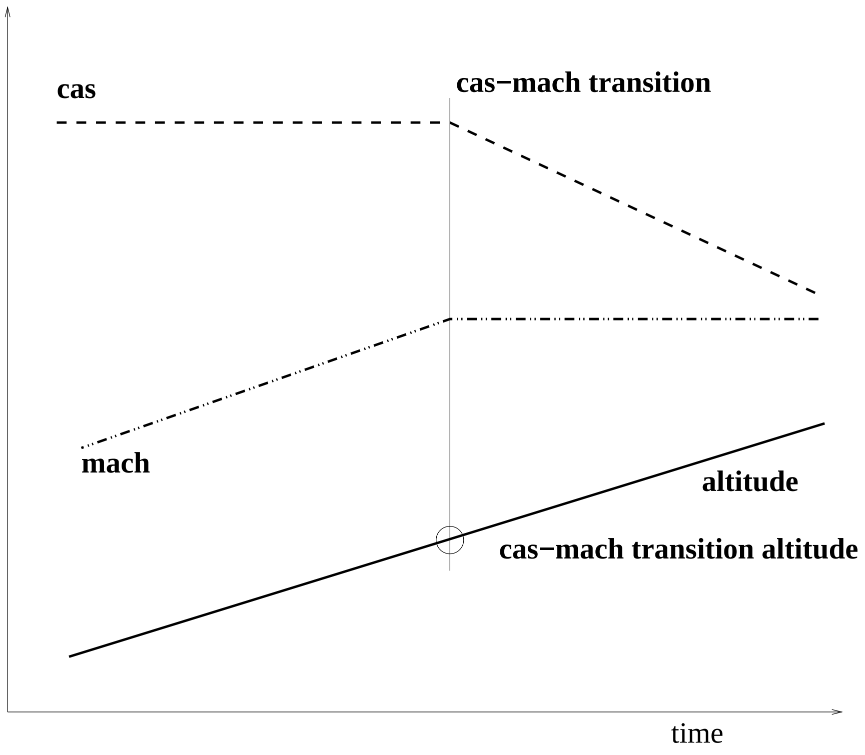


Figure 1.3: Aircraft climb trajectory with altitude profile (solid), CAS (dashed), and mach profile (dash-dot)

1.2.3 Shape Analysis of Planetary Nebulae

Planetary nebulae are remnants of dying stars. Scientists try to understand the physics of these nebulae by analyzing data, in particular by analyzing images taken by the Hubble Space Telescope (HST). Figure 1.6a shows the image of planetary nebula IC418 (the "Spirograph" Nebula). Although coming in most different shapes and forms, different regions of the nebula can be distinguished easily: the dwarf star in the center, the core, and the outer hull. Automatic analysis of pictures requires statistical data analysis models that estimate the center and elliptical extent and orientation

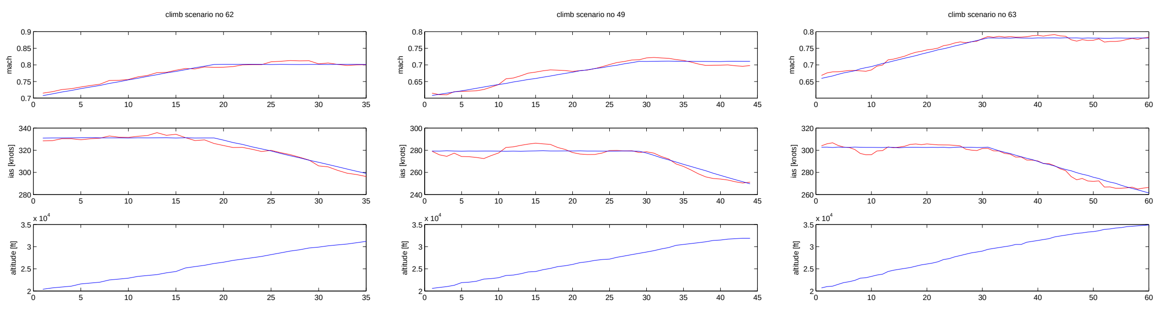


Figure 1.4: Results of analysis with AutoBayes for three climb scenarios. The actual trajectory is shown in red, blue lines correspond to the parameters as estimated by AutoBayes code

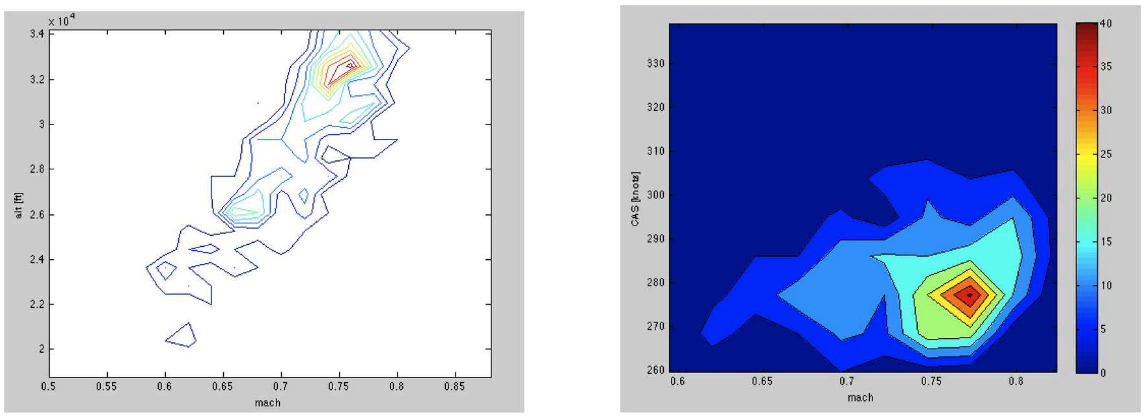


Figure 1.5: Likelihood of transition point in an altitude over mach coordinate system (left) and in a CAS over mach coordinate system (right). Two major transition points at altitudes of approximately 26,000ft and 31,000ft are detected.

of the nebula. In close collaboration with scientists at NASA Ames, several AutoBayes models were defined to estimate center and extent of a nebula. Figure 1.6b shows the manually masked image, which has been taken as the basis for the analysis. Following the approach in [KH02] of using a hierarchy of statistical models to estimate the nebula's parameters, a first model uses a 2-dimensional spherical Gaussian model to estimate center (x_0, y_0), intensity i_0, and radius r of the nebula, where intensity F is modeled as

$$F(x, y) = i_0 \cdot e^{-\frac{(x_0-x)^2+(y_0-y)^2}{2r^2}} \tag{1.1}$$

In the statistical model, the data (pixel intensity) $d(x, y)$ is distributed as $d(x, y) \sim F(x, y) + \eta$ with white noise η.

Figure 1.6c shows the original image superimposed with the estimates of the parameters of the Gaussian model. Figure 1.6d shows sampled data generated using the estimated Gaussian model parameters.

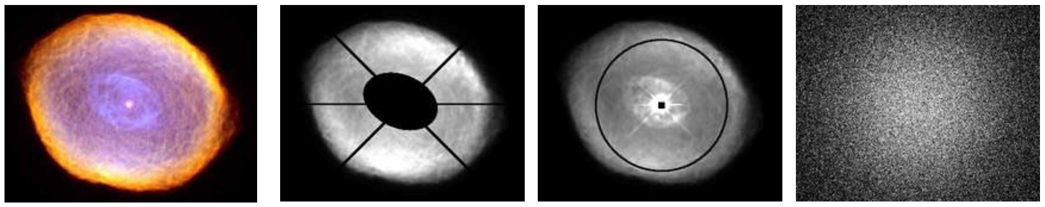


Figure 1.6: Planetary nebula IC 418 or Spirograph Nebula (a) Composite false-color image taken by the HST (Sahai et al., NASA and The Hubble Heritage Team). The different colors (resp. gray-scales) indicate the different chemicals prevalent in the different regions of the nebula; the origin of the visible texture is still unknown. The central white dwarf is discernible as a white dot in the center of the nebula. (b) Manually masked original image. (c) Original image with estimated Gaussian model parameters superimposed. (d) Sample data generated using estimated Gaussian model parameters.

In a second set of models, image segmentation techniques were used. Here, image segmentation via clustering was used. The AutoBayes model, developed for this application is a two-dimensional mixture (of Gaussians) model. AutoBayes generates a customized EM algorithm. Figure 1.7 shows results using clustering-based image segmentation techniques. Models without and with geometric refinement, i.e., constraints about the shape are shown. Note that these AutoBayes models are insensitive enough against several image artifacts (e.g., the "spokes").

Results of this studies were published in [FHKS03, FS03a, KH02].

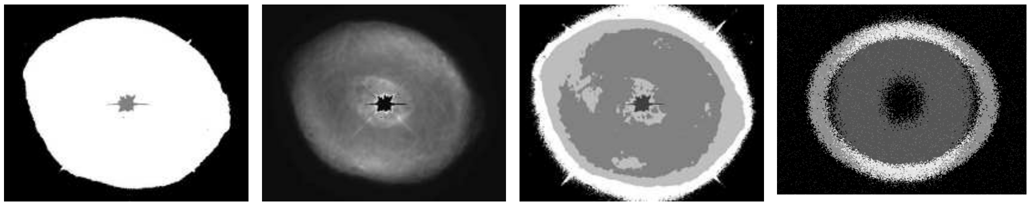


Figure 1.7: Segmentations of IC418 image: (a) three-class segmentation (b) ditto, white class used as mask load to original image (c) five-class segmentation, and (d) sample data from geometrically refined segmentation model; class 1 (white) corresponds to the hull, class 2 (gray) to the core, and class 3 (not shown) to the background.

1.2.4 Clustering for Sloan Digital Galaxy Survey

For understanding the large scale structure of the universe, mapping of structures of all scales (galaxies to large walls) is important. The Sloan Digital Sky Survey (SDSS) contains photometric images including accurate redshifts. The classification of galaxy distances based on redshift using advanced Bayesian data analysis techniques has been explored at NASA Ames. [SSF05] describes an ensemble approach to building Mercer Kernels with prior information. These data adaptive kernels can encode prior information and have been learned directly. For this research work, AutoBayes was used to estimate the parameters for the kernels. A very compact AutoBayes specification (a mixture model with priors) produces an efficient customized variant of the EM algorithm. [SSF05] describes the approach; the mathematical derivation in this paper has been automatically generated and typeset by AutoBayes. Figure 1.8 (left) shows the clustering of a small section of the sky using spectroscopically determined redshifts. Dots indicate galaxies on filaments, crosses indicate field-galaxies (not in filaments). The log-likelihood of the AutoBayes Gaussian mixture model (without priors here) is shown in Figure 1.8 (right) as a function of number of components in the model. This diagram shows that there is substantial variation due to the well-known sensitivity of the EM algorithm to its (random) initialization.

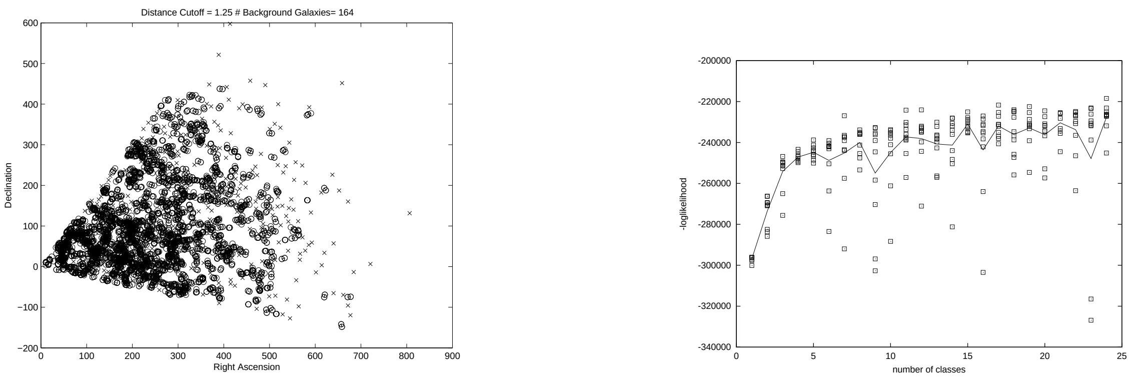
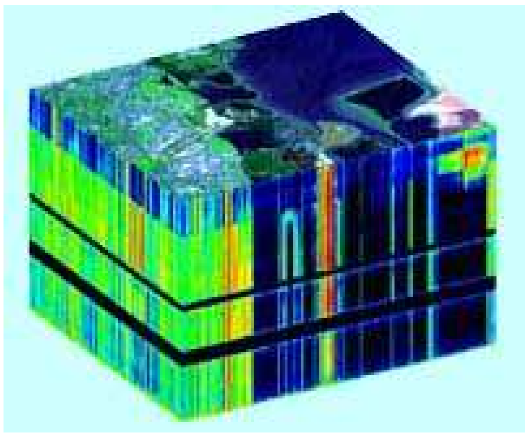


Figure 1.8: Clusters of galaxies (left) and log-likelihood of the AutoBayes model as a function of model components (number of classes, right).

1.2.5 Hyperspectral Clustering of Earth Science Data

Earth-observing satellites like MODIS have a hyperspectral camera. This means that a certain image is taken with a multitude of different wavelengths in the infrared spectrum. The resulting data block is called a hyperspectral data cube (Figure 1.9, left). Since different ground cover (e.g., trees, grass, buildings, water) and different minerals (granite, sand, limestone) reflect the light with a different intensity for different wavelength, scientists can use these data to analyze ground coverage, detect wild fires, or find dying trees.

A simple multivariate mixture model developed with AutoBayes can easily cluster such a hyperspectral cube. The resulting most probable class assignment for each pixel has been used to color the image according to the estimated groups (Figure 1.9, right). The various different ground covers (grass, water, marshland, etc) can be seen easily. In this case, AutoBayes was asked to produce 5 different classes. This clustering gives an easy to interpret result (Figure 1.9(right)). With a larger number of classes more subtle variations are uncovered.



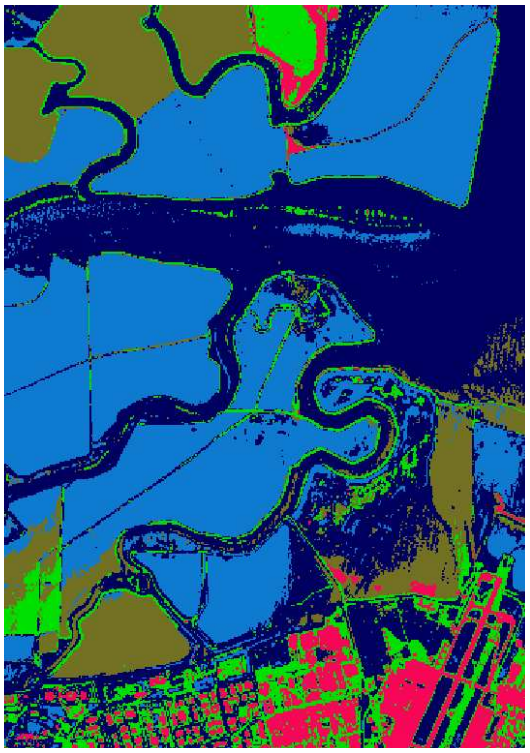


Figure 1.9: Hyperspectral image cube (MODIS) (left) and clustering result for hyperspectral data and 5 classes as produced by AutoBayes (right).

1.2.6 Clustering and Mapping of Geospatial Data

Census data are in general multivariate data. For each household, different variables, like size of household, income, renting or owning are assembled. These data are related to the ZIP code. With a simple AutoBayes multivariate clustering model, such data can be processed and visualized as shown in Figure 1.10. AutoBayes can be easily interfaced into visualization tools such as NASA's World Wind[1] or Google Earth. Figure 1.10 shows the results of a multivariate clustering of US census data along the dimensions household size, rented/owned living quarters, age, and male-female ratio for kids and adults. In the given coding, a total of 11 variables were used. These data were available with the ZIP code as their index. Figure 1.10 reveals pretty distinct classes for the region around Anchorage, the North-Western part, and the Aleutes and can form the basis for further analyses and interpretation.

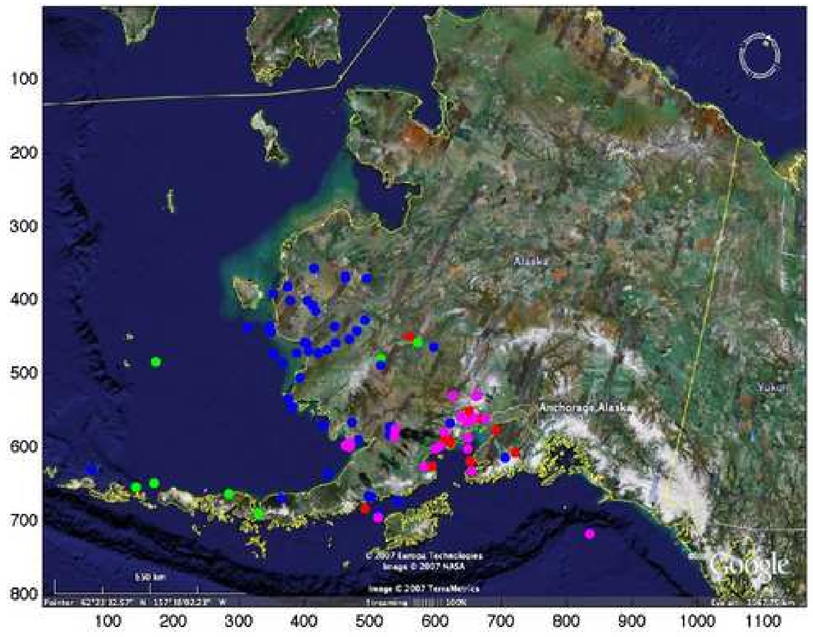


Figure 1.10: Multivariate clustering of census data and mapping onto geographical centers of ZIP codes in parts of Alaska.

1.2.7 Detection of Gamma-ray Spikes

Gamma-ray bursts are very powerful electromagnetic flashes of gamma rays. It is thought that these bursts have to do with the collapse of a high-mass star into a black hole. Instruments in orbit, e.g., the BATSE (Burst and Transient Source Explorer) on the Compton Gamma Ray Observatory (launched 1991) continuously measure

[1]`http://worldwind.arc.nasa.gov`

the intensity of gamma rays. A simple AutoBayes model can be used to detect and isolate such burst events. We assume that the inter-arrival time of photons is exponentially distributed. A detector for a switchpoint, i.e., the transient between a (low) arrival rate and a higher arrival rate can be specified in the AutoBayes specification language in a few lines. The generated program successfully isolates recognized bursts.

2. Installation

AutoBayes runs under Linux and is implemented in SWI-Prolog. The prerequisites for AutoBayes are as follows:

- CPP (C Preprocessor)
- SWI Prolog
- DOT Language (For graphical representation of Bayesian Networks)
- Matlab™ or Octave

2.1 Hardware Requirements

- Solaris
- Mac OSX
- Linux
- Windows with cygwin

2.2 Installation Requirements

2.2.1 C Preprocessor

CPP is the preprocessor for the C programming language and is a requirement for AutoBayes. It is included, e.g., with the GNU C-compiler and is by default available for most platforms.

2.2.2 Installing SWI-Prolog

The stable version (Version 5.6.x or higher) of SWI-Prolog can be downloaded from `http://www.swi-prolog.org/`.

2.2.3 GraphViz

The GraphViz tool is used for generating layouts of the graphical representations of Bayesian Networks. Please refer to the Graphviz website `http://www.graphviz.org/`, and click on the Download tab and follow the instructions in order to have this package installed on your system. Only the tool `dot` is used with the AutoBayes system.

2.2.4 Matlab™ or Octave

Matlab™ or Octave is needed to compile and run the generated synthesized code. Matlab™ is available at `http://www.mathworks.com` and Octave can be obtained from `http://www.octave.org`. Octave is an open source program system for scientific computation which is very similar to Matlab™.

2.3 Getting AutoBayes

AutoBayes can be obtained through two methods: by the source tar-file, or by checking out from a CVS repository. Currently, the CVS repository can only be accessed from NASA Ames, Code TI machines.

2.3.1 Source TAR File

You can unpack the AutoBayes source tar-file `autobayes`$X.Y$`.tgz`, where $X.Y$ is the version number, in the Unix environment by using the following command:

% `tar zxvf autobayes`$X.Y$`.tgz`

or the `gtar` command:

% `gtar zxvf autobayes`$X.Y$`.tgz`

Note: Unpacking a tar-file will write its contents to the current directory.

In the sequel, the directory where AutoBayes was unpacked will be called *autobayeshome*.

2.3.2 AutoBayes CVS Repository

You can use CVS to checkout copies of projects if you have permission for CVS user access. CVS is a good method for keeping track of modification made to project source files.

Obtain the AutoBayes tool through a CVS checkout from the top-level directory using the command line:

`cvs -d :pserver:`*username*`@`*projectname.domain.net*`:/cvs co` *projectname/top-level-directory*

For example:
```
cvs -d wow.arc.nasa.gov:/home/user/cvs co PN
```

2.4 Building and Setting Up AutoBayes

After unpacking the AutoBayes tar-file or checking out the tool from the repository, edit the file *autobayeshome/*`Makefile` so that it contains the correct path for Prolog. The variable `PL` should be set either to `pl` or the exact path where SWI-Prolog has been installed.

In addition, variables in the `Makefile` in the following directory *autobayeshome/system/SWI/* have to be set to the appropriate paths as follows.

The variable `INCLUDEDIR` must be set to the location of the file `SWI-Prolog.h`. For example

```
INCLUDEDIR=/usr/local/pl-5.6.29/lib/pl-5.6.29/include
```

The variable `PN_LIB` must be set to the location of the file `libpl.a`. For example

```
PN_LIB=/usr/local/pl-5.6.29/lib/pl-5.6.29/lib/i686-linux/
```

We can now use the `make` command into the `bash` shell to compile and build AutoBayes:

```
% cd autobayeshome
% make autobayes
```

This command starts the Prolog system, loads the source files, and produces an executable file named `autobayes` in the *autobayeshome* directory.

You have to set the shell variable `AUTOBAYESHOME` to the *autobayeshome* directory; e.g., in your `.bashrc` file:

```
export AUTOBAYESHOME=autobayeshome
```

Then move `autobayes` into a binary directory or adjust the `PATH` accordingly; e.g.,

```
export PATH=./:autobayeshome:${PATH}
```

3. Iris Classical Example

The Fisher Iris flower data set is a multivariate data set introduced by Aylmer Fisher in 1936 as an example of discriminant analysis. The data set contains 50 samples from each of three types of Iris flowers (*Iris setosa*, *Iris Virginia*, and *Iris versicolor*). Each sample has four dimensions, petal length, petal width, sepal length, and sepal width. Although the data set is labeled, for the purpose of this example, we just consider the 4-dimensional unlabeled data set and ask for a classification.

This dataset was used by Fisher in his initiation of the linear discriminant model to classify the flower types based on the combination of the four dimensions.
In this chapter we will:

1. construct a model for the Iris example,
2. invoke AutoBayes on the model and generate program code,
3. compile and run the generated program,
4. provide the Iris data set as input and analyze the results.

3.1 Constructing an AutoBayes Model for Iris Flower Set

As we already know, the first step is to construct a statistical model using the AutoBayes specification language. This section will guide the user through the AutoBayes model for this example. Chapter 8 provides a more complete and thorough description of various AutoBayes input model examples. Chapter 7 discusses details of the AutoBayes specification language.

In Fisher's classical data set, there are 3 classes of 50 instances each. The 3 classes represent the types of the Iris flower: *Setosa*, *Versicolor* and *Virginica*. In our example, we want unsupervised clustering, that is, we will ignore the labels on the data set. We furthermore assume that all measurements are Gaussian distributed, which means that we have a mixture of Gaussians problem. In order to keep the specification compact, we represent the 150 data sets with 4 features each as a matrix of size 4×150.

Please note that AutoBayes uses a 0-based indexing of all vectors and arrays. In our example, the data matrix is indexed `data(0..3, 0..149)`.

Listing 3.1 shows the specification. First, we declare the name of the model (Line 1) and the various constants and parameters. Each of the statements ends with a period ".", and comments are attached to a statement with **as**, with strings delimited by a single quote "'". Line comments are preceded by a %. In order to be as flexible as possible, all the dimensions are declared as symbolic constants. In the declaration, we write **const nat** to obtain a natural number constant that can be zero or a positive integer. We know that the number of classes (3) is positive and much smaller than the number of data points (150). We therefore can provide this knowledge as additional constraints. These guide the AutoBayes system to select the right algorithms and they are checked when the generated code is called. If such a constraint is violated, e.g., if we want to cluster 10 data points into 3 classes (which does not make any sense from a statistical point of view), a run-time error is produced.

The unknown parameters of our model are

- the type frequency *phi* (ϕ) is a probability vector returning how often each class occurs in the data set. As a probability vector, it adds up to one, i.e.,

  $$\sum_{i=0}^{2} \phi_i = 1$$

  Line 11 reflects this constraint. Please note that in the AutoBayes specification language the (scalar) assignment is denoted by a :=, whereas "=" is used for equality comparison.

- mean values μ_{ij} and standard deviations σ_{ij}, declared as matrices, denoting the parameters for the Gaussian for each feature i and each class j. Their data type is **double**. Since we know $\sigma_{ij} > 0$, we specify this as an additional constraint in Line 15. The underscores mean "all values" in the sense of the ":" operator in Matlab™.

```
 1 model iris as
 2     'Simple multivariate clustering model for classical Iris flower
          example'.
 3
 4 const nat n_variables as 'Number of features'.
 5 const nat n_points as 'Number of data points'.
 6 const nat n_classes as 'Number of classes'.
 7         where 0 < n_classes.
 8         where n_classes ≪ n_points.
 9
10 double phi(0..n_classes-1) as 'Class probability vector.' .
11         where sum(I := 0 .. n_classes-1, phi(I)) = 1.
12
```

```
13 double mu(0..n_variables-1, 0..n_classes-1) as 'Matrix of means' .
14 double sigma(0..n_variables-1, 0..n_classes-1) as 'Matrix of std devs'
      .
15   where 0 < sigma(_,_).
16
17 output nat class_assignment(0..n_points-1) as 'Class of each point' .
18 class_assignment(_) ~ discrete(vector(I := 0 .. n_classes-1, phi(I))).
19
20 data double iris_data(0..n_variables-1, 0..n_points-1).
21 iris_data(C,I) ~ gauss(mu(C,class_assignment(I)), sigma(C,
      class_assignment(I))).
22
23 max pr( { iris_data } | { phi, mu, sigma }) for { phi, mu, sigma }.
```

Listing 3.1: AutoBayes model for Fisher's Iris data set.

We specify a vector class_assignment, which stores the most probable class for each data point. This means that it is of length 150, and its values can be 0 (class I), 1 (class II), and 2 (class III). This class assignment is unknown and is to be estimated. Since we want to have the estimated vector returned, this statement is marked with the keyword **output**. Lines 20–21 declare the data iris_data as a matrix, which is Gaussian distributed. The distribution is along the features with separate standard deviations for each class and feature.

Line 23 contains the goal statement, telling AutoBayes the kind of statistical problem that needs to be solved. In our case, we want to estimate the unknown parameters of the mixture; i.e., we need to maximize the probability of the data given the parameters

$$\max Pr(\texttt{iris\_data}|\{\phi, \mu, \sigma\})$$

for the unknown parameters. The specification is a direct transcript.

At the end, we save our model as `iris.ab` and use it in the following steps.

3.2 Invoking AutoBayes on the Iris Input Model

As we mentioned earlier, AutoBayes is a code generator for scientific data analysis programs. We now have a statistical model for our scientific data set (Fisher's data set). The next step is to apply AutoBayes to the model we just constructed and to generate program code to use for our final goal of data analysis. For our example, we want to generate a data analysis function that can be invoked from either the Octave environment [1] or Matlab^{TM}, depending on your working environment preference.

[1] Octave is an open source program system for scientific computation, which is very similar to Matlab^{TM}. Octave can be downloaded from `http://www.octave.org`

3.2.1 Generating Code for Octave

To generate C++ code which can be directly called from Octave as a function, invoke AutoBayes as follows:

```
% autobayes -instrument iris.ab
```

In our example, a file with the name `iris.cc` will be generated. The file name (in our case `iris`) is the same as the model name in the AutoBayes specification (Line 1 in Listing 3.1). In the sequel, we will use this name as the default.

3.2.2 Generating Code for Matlab™

Other target languages and platforms can be selected using the `-target` <*option*> command line option. To generate code that can be called from Matlab™, invoke AutoBayes as follows:

```
% autobayes -instrument -target matlab iris.ab
```

This will generate the program in C. In our example, a file with the name `iris.c` will be generated. The file name (in our case `iris`) is the same as the model name in the AutoBayes specification (Line 1 in Listing 3.1). In the sequel, we will use this name as the default.

3.2.3 Flags

The `-instrument` flag after the `autobayes` command is used to display and store the convergence error at each iteration cycle of the clustering algorithm that it employs in the generated code.

Additional command line options can cause the AutoBayes system to produce different variants of the data analysis algorithms, varying the kind of algorithm (e.g., EM, k-means), initialization, or termination conditions. Details about command line options are described in Appendix A.

3.3 Compiling and Running the Iris Generated Program

Now that we have a generated C++ or C program, we want to dynamically link it into the Octave or Matlab™ environment. This section will go through the steps to compile the generated program and create an `oct`-file for the Octave environment or `mex`-file (using the file created with `-target matlab`) for the Matlab™ environment.

3.3.1 Compiling and Running the Generated Program for the OCTAVE Environment

For us to be able to run the program, we will need to first compile the file `iris.cc` and generate a binary file with the extension `.oct`. We will use the `mkoctfile` command to generate this file:

```
> mkoctfile -I$AUTOBAYESHOME/system/octave/include iris.cc
```

This command will generate the file `iris.oct`. We then enter the OCTAVE environment by typing `octave` into the command line. Once we enter OCTAVE we can call the program we just generated using previous command by typing its name and the appropriate arguments as described in Section 3.4.

3.3.2 Compiling and Running the Generated Program for the MATLAB™ Environment

For working in the MATLAB™ environment, we use the `iris.c` file that we generated using the `-target matlab` flag along with the `mex` command. We type into the MATLAB™ or shell command line:

```
>> mex -I$AUTOBAYESHOME/system/matlab/include iris.c
```

This command will generate a dynamically linked binary file `iris.x` where `x` depends on the operating system (e.g., mexmaci: Mac OSX, mexglx: Linux, mexw32: Windows 32-bit). We now simply call the model by typing its name `iris` into the MATLAB™ command line and it will show the proper usages for the model.

3.4 Providing Input to the Iris Program and Analyzing Results

In order to run the generated program, we enter the OCTAVE or MATLAB™ environment. In the following, we will use OCTAVE, but MATLAB™ works the same. We enter the OCTAVE environment by typing `octave` to the command line prompt.

The next step is to call the model from the OCTAVE environment; in our case, we labeled our model `iris`. If we simply type the model name to the OCTAVE command-line, the model usage is returned[2]:

```
octave:1> iris

usage: [vector class_assignment,matrix mu,vector phi,
                         matrix sigma, vector errors] =
```

[2]We reformatted the output to fit within the margins of this page.

```
                              iris(matrix iris_data,int n_classes,
                                  double tolerance,int maxiteration)
```

In order to cluster the data, we now load the input file that contains the entire data set of 150 samples for the Iris flower. In our case, we have named the file `iris_data.m`. This script sets the variables `Class`, `Petal_length`, `Petal_width`, `Sepal_length`, and `Sepal_width`. The script can be loaded simply by typing its name to the Octave command prompt:

```
octave:4> iris_data
```

One way to make sure your model is linked to your dataset is to use the `who` command after calling the input script. The `who` command will show you the dynamically-linked functions and also the local user variables for the Iris model, for example:

```
*** dynamically linked functions:

iris

*** local user variables:

Class         Petal_length  Petal_width   Sepal_length  Sepal_width

octave:6>
```

The variable `Class` contains the labels, as strings. This variable is not used for our example as we will do unsupervised clustering. The other variables (`Petal_length`, `Petal_width`, `Sepal_length`, and `Sepal_width`) are vectors of size 1×150 which contain the measurements.

You can use the `whos` command which returns the dynamically linked function and also the local user variables along with row and column information for each variable:

```
octave:6> whos

*** dynamically linked functions:

prot  type                        rows   cols  name
====  ====                        ====   ====  ====
 r--  dynamically-linked function      -      -  iris_classical_example

*** local user variables:
```

```
prot  type                          rows   cols  name
====  ====                          ====   ====  ====
 rwd  string                           1   2000  Class
 rwd  matrix                           1    150  Petal_length
 rwd  matrix                           1    150  Petal_width
 rwd  matrix                           1    150  Sepal_length
 rwd  matrix                           1    150  Sepal_width

octave:7>
```

Now we need to define a matrix containing all the measured data. As defined in the AutoBayes model, this matrix is of size 4 × 150 (n_features × n_points). We will call this matrix `data` and set its values as below:

```
octave:7> data = [Petal_length; Petal_width; Sepal_length; Sepal_width ];
```

The next step is to invoke the `iris` program with input and output parameters:

```
[c, mu, phi, sigma, errors]=iris(data,3,0.00001,30);
```

where:

`c`	class assignment vector
`mu`	mean
`phi`	class frequency (class probability)
`sigma`	standard deviation
`errors`	vector to store errors
`3`	number of classes
`data`	matrix containing Iris data
`0.00001`	convergence error for termination
`30`	maximum number of iterations

When this command is called, you should receive an output similar to the following:

```
octave:10> [c,mu,phi,sigma,errors] = iris(data, 3, 0.00001, 30);
 pvar(89) = 3.25039
 pvar(89) = 1.36758
 pvar(89) = 0.445656
 pvar(89) = 0.733875
.
.
```

```
.
 pvar(89) = 0.00627092
 pvar(89) = 0.00575855
 pvar(89) = 0.00528546
 pvar(89) = 0.00484893
```

Please note that since the initialization process is random, each time you get different results.

For each iteration of the expectation-maximization (EM) algorithm (Section 5.1.2), the current error is displayed (Section 8.2.2). This error should become smaller as the algorithm proceeds, but "jumps" are possible. If the error becomes smaller than the given "tolerance" (10^{-5} in our example) or if the number of iterations exceeds "max iterations" (30 in our example), the algorithm terminates and sets the result variables. Figure 3.1 shows the development of the error; this is just a semilog plot of the variable "errors". This plot can easily be generated by typing `semilogy(errors)`.

3.4.1 Executing Commands and Reading Outputs

Now that you have executed the model and generated results, it is time to view them. As specified in the AutoBayes model, the algorithm estimates the parameters of the Gaussians μ, σ^2 for each class as well as the class frequency of ϕ. Also the most likely class assignment `c` is returned as it has been declared as an output variable. `c` is a vector of size 150 with entries 0, 1, 2 which correspond to the class assignment (classes I, II, and III, respectively) for each measurement. We will use the variable `c` to visualize the clustering.

Type `mu` into the command line to view the means for all three classes and their features. For example you should get similar outputs to below where columns are classes I through III, and the rows are features 1 through 4 (petal length, petal width, sepal length, sepal width):

```
octave:15> mu

mu =

  5.49353  1.46400  4.23318
  1.99709  0.24400  1.30830
  6.62647  5.00600  5.84463
  3.01786  3.41800  2.70497

octave:16>
```

You can also type `phi` or `sigma` to view the probability of classes I–III and standard deviations respectively.

For example:

```
octave:16> phi

phi =

  0.35589
  0.33333
  0.31078

octave:17> sigma

sigma =

  0.56852  0.17177  0.47892
  0.28738  0.10613  0.18796
  0.57168  0.34895  0.48216
  0.28793  0.37719  0.29656

octave:18>
```

3.4.2 Interpretation of Results

With the results provided by AutoBayes, the means μ, the standard deviations σ^2, the class frequencies ϕ, and the most-likely class assignment c, we now can start to interpret and visualize the results.

It is hardly surprising that the class frequency ϕ is, for each of the classes, around 0.3, which means that roughly 30% of the measurements fall into each of the three classes. The original set is actually constructed such that exactly 1/3 of the measurements (50 out of 150) belong to each of the three types of Iris flowers (classes). Interestingly, however, the numbers found by the AutoBayes model are not exactly those, which means that some of the data points have been misclassified. A closer look at the data points reveals more information. Please note that we do *not* use the class label, which is provided with the original data set. Rather, our aim was to separate the data as best as possible into three classes *without* knowing the answer.

Figure 3.2 shows a simple scatter plot of the 150 data points along the various pairs of features. Because there are 4 variables, a 4×4 matrix is needed. This figure shows

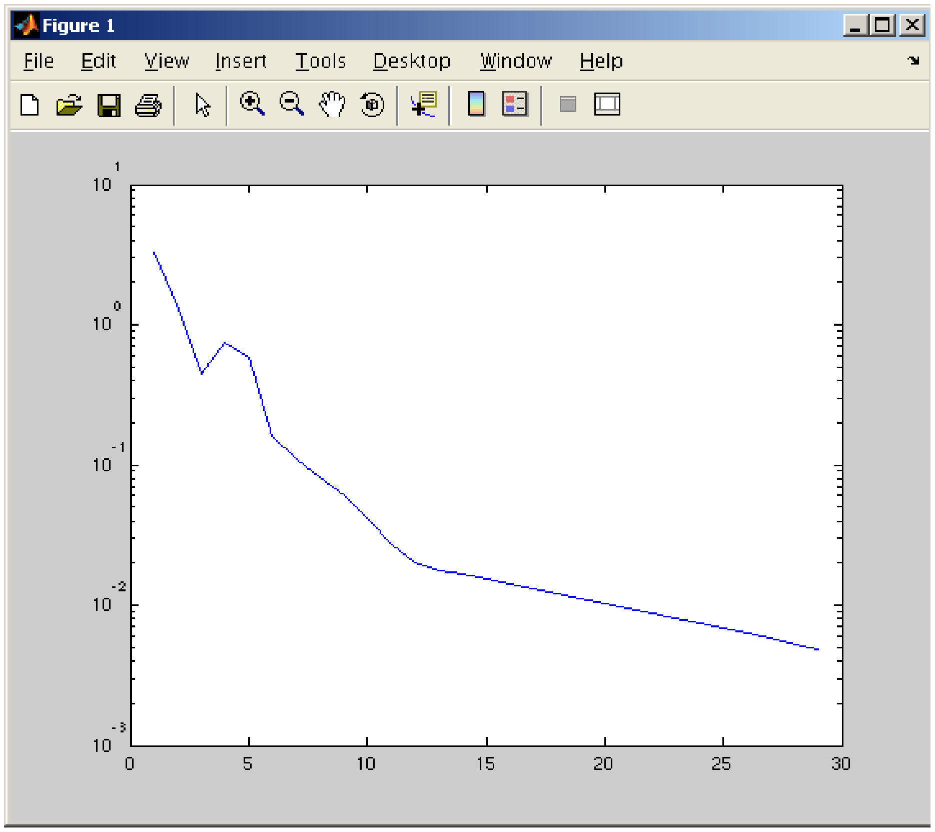


Figure 3.1: This graph shows the logarithm of the error term convergence on the y-axis and time (iteration) on the x-axis.

only the upper triangular part of the scatter matrix. In all plots, two groups can easily distinguished. For the human eye, a classification of the data into three classes seems to be impossible.

After running the AutoBayes-generated code, we present the same scatter plots, however this time, each data point is colored according to its most probable class membership c_i (Figure 3.3).

Finally, the governing parameters of the three Gaussians (μ_i, σ_i^2) can be plotted into such a scatter-plot. Figure 3.4 shows such a plot for a selected pair of features. The thick magenta dots represent the center of the Gaussian, the cyan ellipsis corresponds to a $1\sigma^2$ ellipsis. Also, we have indicated Fisher's given classification by plotting three different plot symbols ($\otimes$, $\odot$, $\oplus$). In a perfect classification, each kind of plot symbol would be the same color. With such a visualization, it can easily be seen how the data have been separated and where potential mis-classifications occur. These figures are only a few possibilities for visualization of the results.

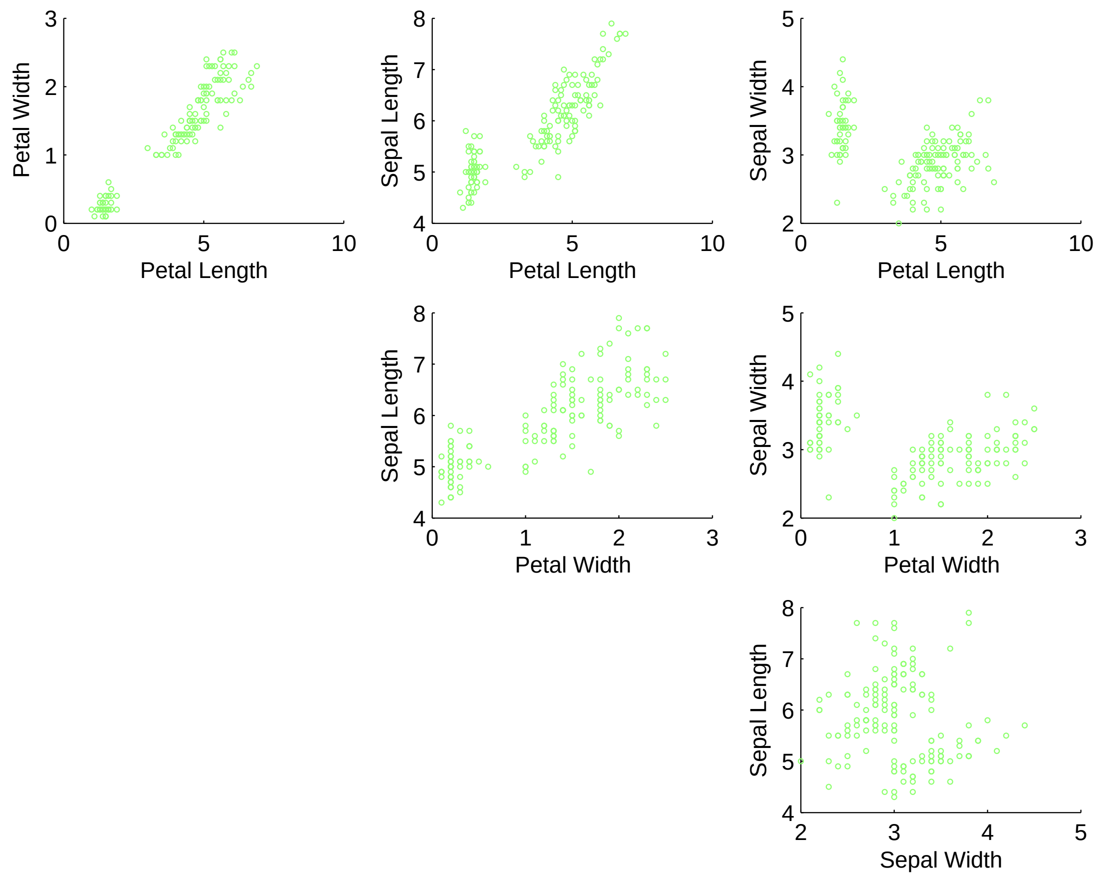


Figure 3.2: Scatter-plot for each of the four variables of the iris data set.

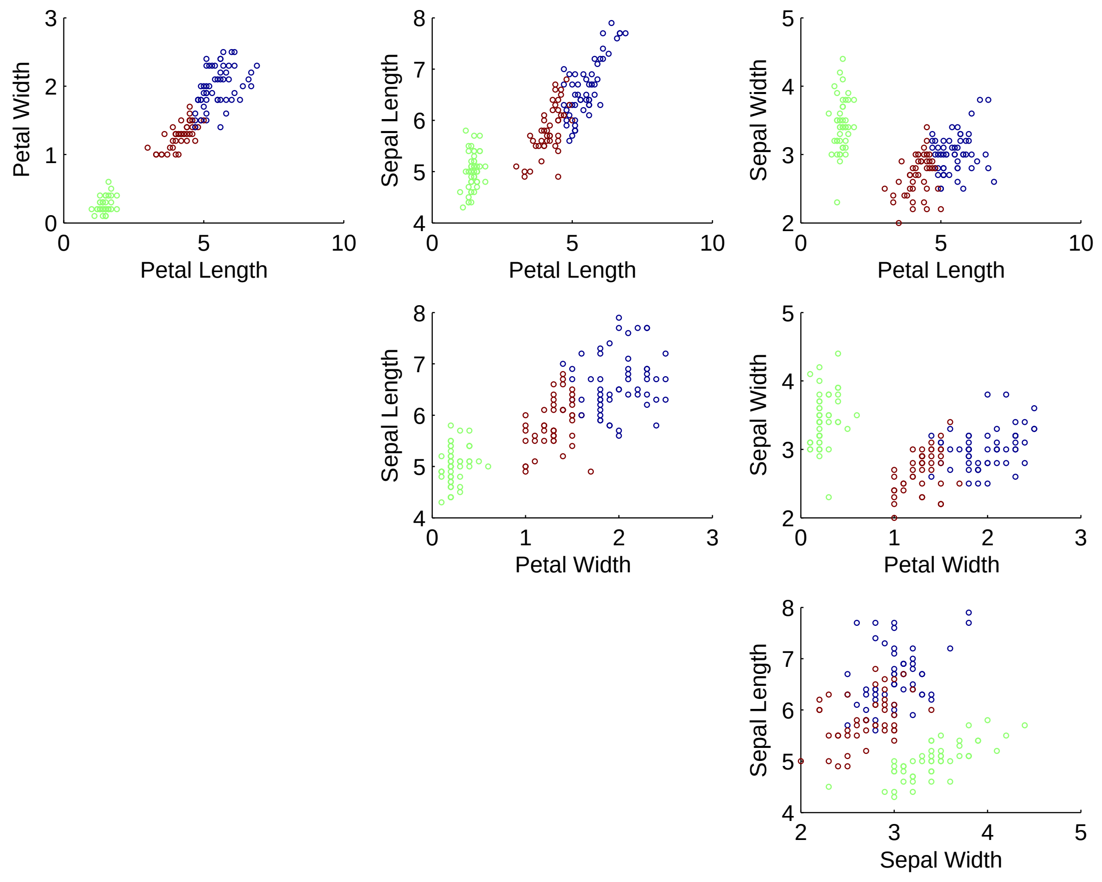


Figure 3.3: Scatter-plot for each of the four variables of the Iris data set. Most likely classes for each data point are shown in different colors.

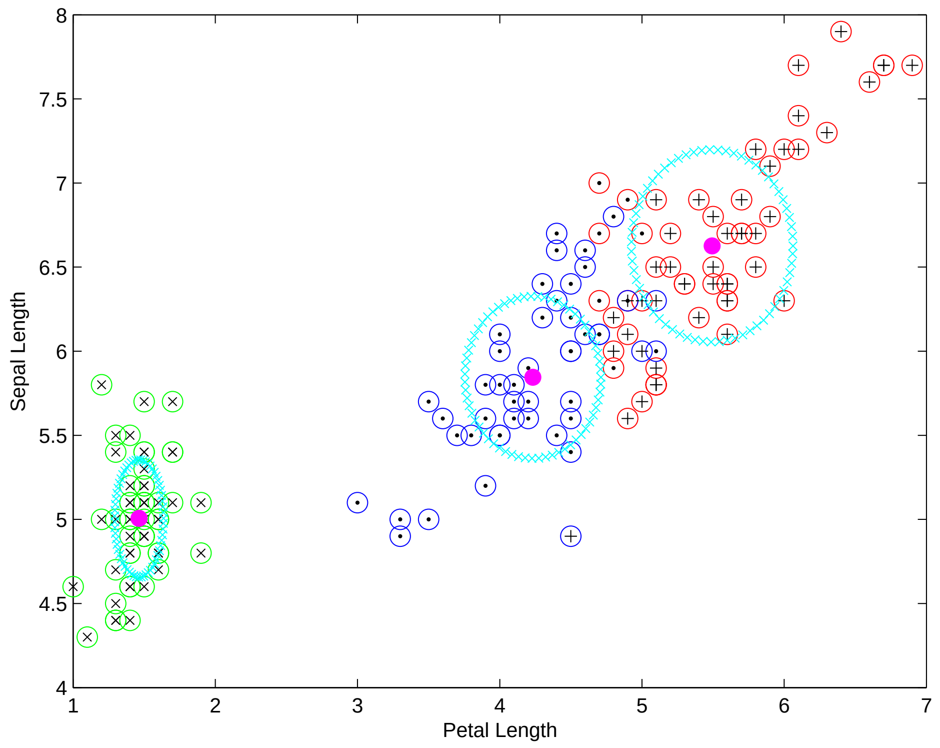


Figure 3.4: Scatter-plot for the features Petal length versus Sepal length with estimated class parameters μ, σ^2 shown.

3.4.3 Matlab™ Scripts for Iris Plots

Listings 3.2 and 3.3 show the Matlab™ scripts used to generate Figures 3.2 and 3.3, and Figure 3.4, respectively. The AutoBayes-generated clustering algorithm has been executed beforehand, so the estimated parameters *mu*, *sigma*, and class assignment *c* are available.

```
figure;
S=5;

subplot(3,3,1);
scatter(Petal_length,Petal_width,S,c);
xlabel('Petal Length');ylabel('Petal Width');

subplot(3,3,2);
scatter(Petal_length,Sepal_length,S,c);
xlabel('Petal Length');ylabel('Sepal Length');

subplot(3,3,3);
scatter(Petal_length,Sepal_width,S,c);
xlabel('Petal Length');ylabel('Sepal Width');

subplot(3,3,5);
scatter(Petal_width,Sepal_length,S,c);
xlabel('Petal Width');ylabel('Sepal Length');

subplot(3,3,6);
scatter(Petal_width,Sepal_width,S,c);
xlabel('Petal Width');ylabel('Sepal Width');

subplot(3,3,9);
scatter(Sepal_width,Sepal_length,S,c);
xlabel('Sepal Width');ylabel('Sepal Length');
```

Listing 3.2: Matlab™ script used for generating Iris scatter plots

```
figure;

S=5;

C=['r','g','b'];
for i=1:3
        cm=find(c==i-1);
        plot(Petal_length(cm),Sepal_length(cm),[ C(i) 'o'],'markersize
            ',10');

        hold on;
end;

plot(Petal_length(1:50),Sepal_length(1:50),'kx');
plot(Petal_length(51:100),Sepal_length(51:100),'k.');
plot(Petal_length(101:150),Sepal_length(101:150),'k+');
xlabel('Petal_Length');
ylabel('Sepal_Length');
i1=1;
i2=3;

hold on;

for cl=1:3,
        plot(mu(i1,cl),mu(i2,cl),'m.','Markersize',25);

        for p=0:0.1:2*pi,
                x = mu(i1,cl) + sigma(i1,cl)*cos(p);
                y = mu(i2,cl) + sigma(i2,cl)*sin(p);
                plot(x,y,'cx');
        end;
end;
```

Listing 3.3: MATLAB™ script for scatter-plot with estimated parameters

4. System Functionality

In this chapter, we will describe the internal workings of the AutoBayes system. First, an overview of the architecture of the AutoBayes system will be given. Then, we will discuss how the code and the documentation is generated. Finally, we will describe how AutoBayes can be used to generate artifical (random) data for the given statistical model.

4.1 Overview

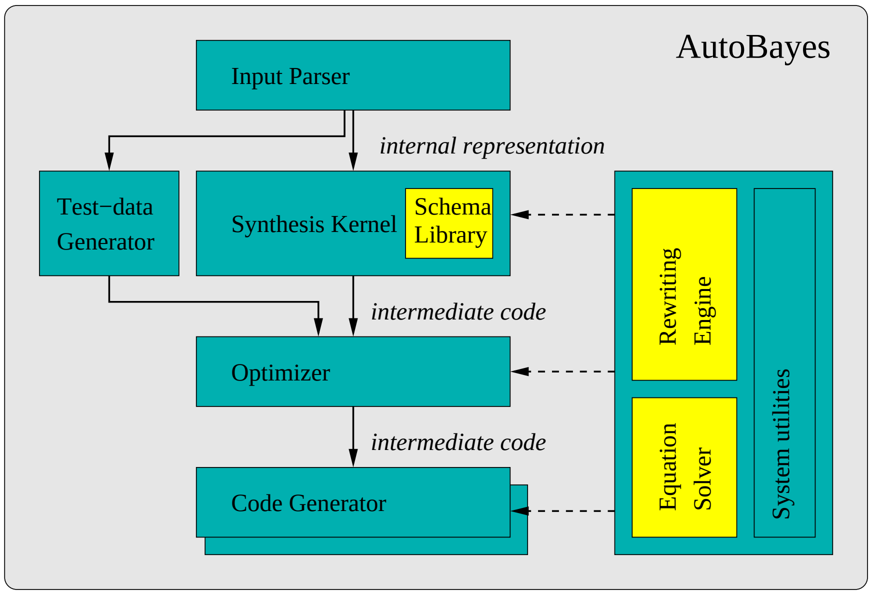


Figure 4.1: Principle Architecture of AutoBayes

In a first processing step, the given specification is parsed, converted into internal form, and the Bayesian network is constructed. This step can also generate an external representation for visualization purposes, using the graph drawing tool graphviz[1]. The *synthesis kernel* then analyzes the network, tries to solve the given optimization task, and instantiates appropriate algorithm schemas, which are given in a schema library.

[1] `http://www.graphviz.org`

Figure 4.1 shows the principle architecture of AutoBayes. The output of the synthesis kernel is a program in a procedural intermediate language. AutoBayes's backend takes the intermediate code, optimizes it and generates code for the chosen target system. Currently, C or C++ code is generated, which can be used in a stand-alone way, or can be linked to the Matlab^{TM} or Octave environment. The synthesis kernel also produces detailed documentation in the form of an HTML design document along with the code. Furthermore, AutoBayes can generate code which generates artificial sampling data for the model, e.g., for visualization and testing purposes.

All parts of the AutoBayes system rely heavily on a symbolic subsystem and some auxiliary system modules (e.g., pretty-printer, set representations, I/O functions). For symbolic mathematical calculations, we implemented a small but reasonably efficient rewriting engine in Prolog. Graph handling, simplification of mathematical expressions, and an equation solver are implemented on top of it. The system architecture is designed in such a way that most of its parts can be re-used for different domains. In particular, the backend and symbolic subsystems are entirely independent of the data analysis domain. For details on the principles of schema based program synthesis see [FS03b].

4.2 Generating Code

The synthesis kernel of AutoBayes generates code in an intermediate language before the code for the actual target system is produced. This intermediate language is a simple procedural language with several domain-specific extensions such as convergence loops, vector normalization, simultaneous vector assignment, as well as assertions and annotations. The domain-specific constructs allow target-specific optimizations and transformations. For example, the `sum` construct of the intermediate language, for calculating the sum of vector elements, can be converted into a "for" loop, an iterator construct for sparse matrices, or a function call to a library. The language supports most basic matrix operations.

The actual target-specific portion of the code generator is rather straightforward and can be adapted to different target languages and enviroments. With the help of rewrite rules all constructs of the intermediate language are transformed into constructs of the target language and printed using a generic pretty-printer. The backend also generates boiler-plate code to interface the algorithm with the target system, and optimizes the code. However, standard optimizations (e.g., evaluation of constant expressions) are left for the subsequent compilation phase. There is no need to perform the same optimization steps that are already in many modern compilers.

4.3 Generating Documentation

Certification procedures for safety-critical applications (e.g., in aircraft or spacecraft) usually mandate manual code inspection. This inspection requires that the code is readable and well documented. Even for programs not subject to certification, understandability is a strong requirement because manual modifications are often necessary, e.g., for performance tuning or system integration. However, existing code generators often produce code that is hard to read and understand. In order to overcome this problem, AutoBayes generates *explanations* along with the programs that show "synthesis decisions": which algorithm schemas have been used, how schema parameters have been instantiated, etc. Model assumptions that were used and proof obligations that could not be discharged during the synthesis are laid out clearly. This makes the synthesis process more transparent and provides traceability from the generated program back to the model specification. In addition to the design document, AutoBayes can also generate the mathematical derivation of the generated algorithm as a LaTeX-document, when the command-line flag `-latex` is used. For illustration purposes, Chapter 8 shows the autogenerated derivations of selected examples, enclosed between **begin autogenerated document** and **end autogenerated document**.

4.4 Generating Artificial Test Data

Visualization and simulation play important roles in the development of data analysis programs. An AutoBayes model specification contains enough information to synthesize code which generates artificial sampling data according to the specification. Generating artifical test data is very helpful in understanding the model and the generated code. If the artificial data does not match the real data set or the scientist's expectations, the specified model might not reflect the reality properly. Artificial data sets can also be used to assess and evaluate the performance of the synthesized code before real data become available. This feature is of particular interest in cases where AutoBayes allows the instantiation of different algorithms for the same specification. For example, if AutoBayes synthesizes different variants for initialization of the hidden variable, their coarse relative performance can be assessed with the generated test data.

Code for the generation of artificial test data can be produced easily by using the commandline flag `-sample` on the AutoBayes specification. The system then generates a file `sample_`*`model`*`.cc` that can be compiled in the usual way. Values for all constants and the parameters which are to be estimated must be provided when this function is called. It then returns random values for the *data* and hidden variables. For example, the Iris example of Chapter 3 would generate the following sampling function

```
[vector class_assignment,matrix iris_data] =
  sample_iris_classical_example(matrix mu,int n_points,vector phi,matrix sigma)
```

Figure 4.2 shows a scatterplot of an artificially generated data set with 150 data points. A comparison with Figure 3.3 on page 3.3 shows that the artificial sampling data, generated by our statistical model, are very similar to the original data set.

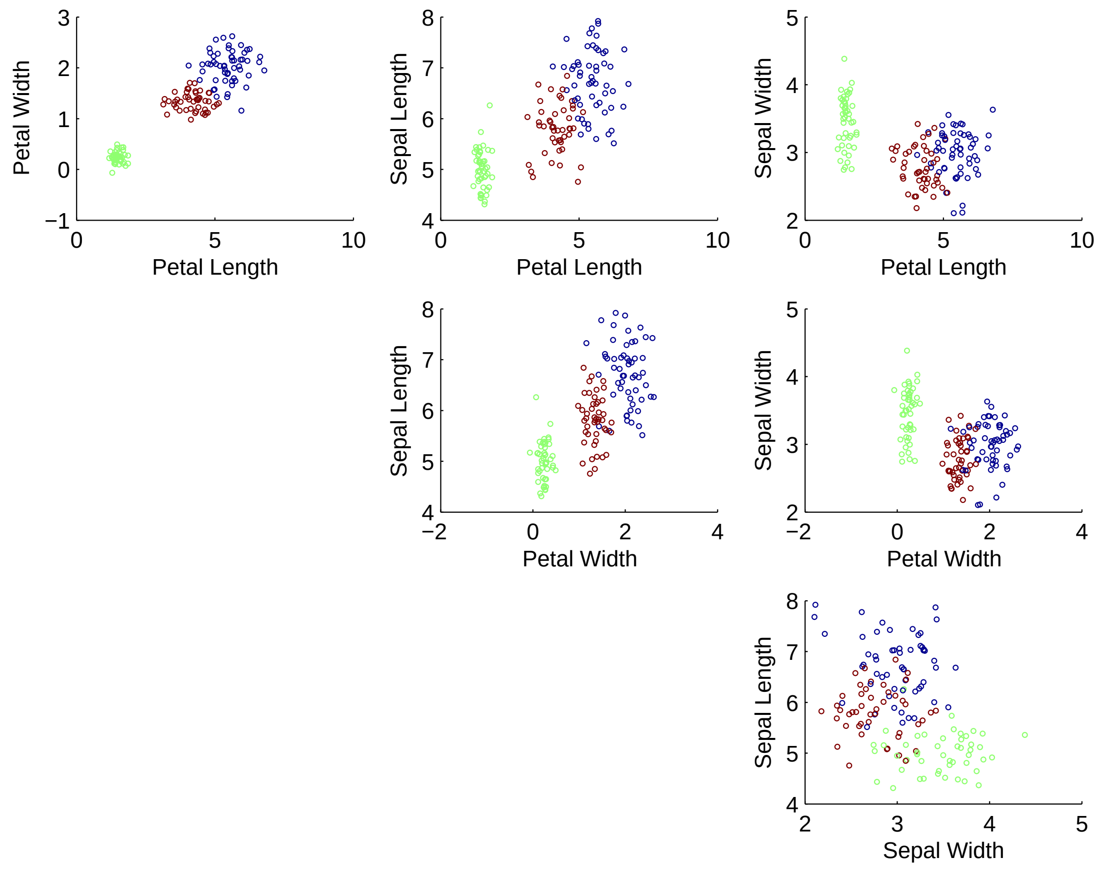


Figure 4.2: Scatter-plot shown artificially generated sampling data from the AUTO-BAYES iris model (Listing 3.1)

5. Generated Algorithms

AutoBayes generates appropriate algorithms from algorithm skeletons with the help of symbolic calculations. Typical algorithms include:

- clustering
- numeric optimization

5.1 Clustering Algorithms

AutoBayes uses clustering algorithms to generate code from certain statistical models. Clustering deals with finding a *structure* in a collection of unlabeled data. A brief definition of clustering would be “the process of organizing objects into groups whose members are similar in some way”. A *cluster* is therefore a collection of objects which are “similar” and are “dissimilar” to the objects belonging to other clusters. AutoBayes currently implements two such algorithms, namely k-means and the EM algorithm.

5.1.1 k-Means Algorithm

K-means (MacQueen, 1967) is one of the simplest unsupervised learning algorithms that solve the well-known clustering problem. The procedure classifies a given data set into a number (k) of clusters, where k is specified a priori. The main idea is to define k-centroids, one for each cluster. These centroids should be placed initially in a cunning way because different initial locations cause different results. A good choice is to place them as far away from each other as possible. The next step is to take each point from the given data set and associate it to the nearest centroid. This induces an early clustering. At this point we need to recalculate k-new centroids as barycenters of the clusters resulting from the previous step. For each group, this minimizes the objective function that is the sum of the Euclidean distances of the centroid from each point in the group. After we have these new centroids, the step is repeated by associating each data point with the new nearest centroid. These steps are done iteratively until the centroids do not move anymore. The output of the algorithm is the location of the k-centroids.

5.1.2 The EM Algorithm

An expectation-maximization (EM) algorithm is used in statistics for finding maximum probability estimates of parameters in probabilistic models, where the model depends on unobserved latent (hidden) variables. The EM algorithm [MK97] alternates between performing an expectation (E) step, which computes an expectation of the likelihood by including the latent variables as if they were observed, and a maximization (M) step, which computes the maximum likelihood estimates of the parameters by maximizing the expected likelihood found during the E step. The parameters found in the M step are then used to begin another E step, and the process is repeated. A detailed description of the algorithm as applied to a mixture-of-Gaussians problem is given in Section 8.2.

5.2 Numerical Optimization Algorithms

AutoBayes uses numerical optimization algorithms in order to generate code for the statistical model. The numerical optimization task is to find, given a single function that depends on one or more parameters, values for those parameters where the given function achieves its maximum or minimum value. For a comprehensive description of well-known optimization algorithms see [GMW81]. Typical numerical optimization algorithms that AutoBayes uses include:

- Nelder-Mead Downhill Simplex Method
- Golden Section Search Method

5.2.1 Nelder-Mead Downhill Simplex Method

The *downhill simplex method* [PFTV92] is due to Nelder and Mead. This method only requires function evaluations, but not derivatives.Therefore, it is a natural choice in many situations where no derivatives exist, or where they are very expensive to calculate.

A *simplex* is a geometrical figure: in N dimensions, it is the convex hull of $N+1$ independent points (or vertices). In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron (not necessarily a regular tetrahedron). In general we are only interested in simplexes that are non-degenerate, i.e., that enclose a finite inner N-dimensional volume. If any point of a non-degenerate simplex is taken as the origin, then the N other points define vector directions that span the N-dimensional vector space. For multidimensional minimization, the best we can do is give our algorithm a starting guess, that is, an N-vector of independent variables as the first point to try. The algorithm is then supposed to make its own way downhill through the unimaginable complexity of an N-dimensional topography, until it encounters a (local) minimum. The downhill simplex method must be started not just with a single point, but with $N + 1$ points, defining an initial simplex.

5.2.2 Golden-Section Search Method

The Golden-Section search method [PFTV92] is a numerical optimization method for continuous functions of a single variable which does not use derivatives at all (i.e., it does not require either symbolic differentiation or numerical computation or approximation of derivatives). The algorithm optimizes by iteratively bracketing the minimum by a triplet of points, $a < b < c$, such that $f(b)$ is less than both $f(a)$ and $f(c)$ and $b - a$ and $c - b$ are related by the golden ratio. In this case we know that there exists a local minimum in the interval (a, c). The algorithm chooses a new point x, either between a and b or between b and c. Suppose, to be specific, that we make the latter choice. Then we evaluate $f(x)$. If $f(b) < f(x)$, then the new bracketing triplet of points is (a, b, x); otherwise, if $f(b) > f(x)$, then the new bracketing triplet is (b, x, c). This process continues until the distance between the two outer points of the triplet is within a designated tolerance level. Note that the convergence is *linear*, meaning that each iteration gains an additional binary digit of accuracy.

5.2.3 Initialization

General optimization algorithms (in particular the Nelder-Mead Simplex Algorithm) require values for the starting point and values for the initial step size. Poor choices can result in poor performance of the algorithm and even non-termination. AutoBayes provides several ways for finding initial start and step values. Its behavior can be controlled via two pragmas, `schema_control_arbitrary_init_values` and `schema_control_init_values`. The following methods for calculation of the start value s_0 and step δ_0 are provided; Table 5.1 shows how the pragmas need to be set.

Range If the variable under consideration is restricted in its value range by constraints to $[l, \ldots, h]$, then $s_0 = l + (h - l)/2$ and $\delta_0 = (h - l)/10$.

Prior If the variable has a prior, the initial values are determined by the moments of the prior: s_0 becomes the first moment, δ_0 the second moment.

User Additional input arguments for the generated code are generated to provide s_0 and δ_0.

Arbitrary If no additional information is known, AutoBayes uses $s_0 = 1$ and $\delta_0 = 1$.

Pragma	**Range**	**Prior**	**User**	**Arbitrary**
`schema_control_arbitrary_init_values`	`false`	`false`	`true`	`true`
`schema_control_init_values`	automatic	automatic	user	arbitrary

Table 5.1: Control of initialization

5.3 Generic Optimization

Whenever AutoBayes encounters an optimization problem (maximization or minimization), it first attempts to find a closed-form solution. This can be attempted by symbolically calculating the derivatives, setting them to zero, and solving the resulting set of equations. In addition, the second derivatives are calculated (if they exist) and checked for the correct sign. If such a solution cannot be found (e.g., there exists no closed form solutions, or if the symbolic solver in AutoBayes cannot find a solution), the problem is divided into pieces which can be handled individually. If this does not yield results, numerical optimization algorithms are instantiated. If this is not successful, AutoBayes fails to generate a program, unless the pragma `schema_control_use_generic_optimize=true` is set. In that case, a (non-executable) statement `optimize(...)` is generated in the final code for debugging purposes.

6. Using AutoBayes

AutoBayes is a program generator for scientific data analysis programs. The input problem is a file written as a specification of a data analysis problem in the form of a statistical model with declarations, constraints, and a goal. Then AutoBayes is invoked on that input file. From the outside, AutoBayes works very similar to a compiler: AutoBayes reads the input file and command-line options and generates an internal representation (a Bayesian network). Once the algorithms have been generated and optimized, AutoBayes generates the code files as well as various listing and documentation files. In this reference chapter, we will discuss calling conventions, common command-line flags, compilation of generated code, and AutoBayes error messages.

6.1 Invoking AutoBayes on an Input File

To invoke AutoBayes on a specific input file, the `autobayes` command is used:

% `autobayes` *[options] [pragmas] modelfilename*`.ab`

Unless a specific target language is requested, this call will generate C++ code from the input file by default. The extension for model files is "`.ab`". The names of files generated by AutoBayes are derived from the name of the statistical model (given by the **model** declaration in the input specification, i.e,

`model` *modelname* `as '...'.`

or, in one case, from the model's filename. For the following, we assume the model name is *modelname* and the model's filename is *modelfilename*.

Each *option* is specified by a flag on the command line optionally followed by an argument. Each *pragma* is introduced by the flag '`-pragma`' followed by a pragma variable and its setting. The following sections describe options and pragmas in more detail.

6.1.1 Generated File Names

The files below can be generated by AutoBayes.

- `cpp_`*modelfilename*`.ab` The model file preprocessed by `cpp`.
- *modelname*`.cc` The generated C++ code.
- *modelname*`.c` The generated C code.
- *modelname*`.cc.html` The generated C++ code in html format.
- *modelname*`.c.html` The generated C code.in html format
- *modelname*`.log` The default log file generated when the '`-log`' option is specified. A particular file can be specified using the '`-logfile`' option.
- *modelname*`_design.html` The html design document generated when the '`-designdoc`" flag is specified.
- *modelname*`.dot` The graphical representation of the Bayes net generated when either the '`-dot`' or '`-designdoc`' flag is specified.
- *modelname*`.`*stage*`.list` where *stage* is `synt`, `iopt`, `inst`, `lang`, `prop`, `ann`, or `lopt`. These are results of intermediate stages of the program synthesis process, generated when the '`-list`' option is specified. The interesting ones are `synt` which is the synthesized program written in the high-level intermediate language, and `lang`, which is the transformed program just before printing out the C/C++.
- *modelname*`.`*stage*`.dump` Same as the above files, but presented in the internal Prolog format.
- *modelname*`.`*stage*`.tex` The LaTeXfile of results of the intermediate stages of the program synthesis process. This file is generated when the '`-tex`' flag is specified. Providing '`synt`' as the stage generates a LaTeXfile giving the derivation of the generated program.
- `sample_`*modelname*`.cc` The C++ program synthesized to generate artificial data (see Section 4.4).
- `sample_`*modelname*`.c` The C program synthesized to generate artificial data (see Section 4.4).

6.2 Command-Line Flags

AutoBayes command line options are set by specifying flags with the `autobayes` command. For instance '`-help`' is a flag that can be used along with the `autobayes` command to view a list of all available flags. Appendix A.1 lists all flags.

For example:

% `autobayes -help`

The general format is:

% `autobayes -`*flag flag-options*

6.2.1 Help Flag

Using the `'-help'` flag alone will display a list of flags and their options and usages.

For example:

% `autobayes -help`

Providing a flag as argument to the help flag, as in '`-help` ***flag***', displays the usage of the option associated with the particular flag *flag*.

6.2.2 Design Document Flag

The '`-designdoc`' flag will generate a software design document for the specified model in html format. It will also generate in a '`.dot`' file a visualization of the Bayes net[1]. For example:

% `autobayes -designdoc` *modelfilename*`.ab`

You can specify a name for the generated document by using:

% `autobayes -designdoc` *designdocfilename modelfilename*`.ab`

The design document contains the following information.

- Summary

  List of File Names

- Input Specification

  Textual Input Specification

  Graphical Representation

- The Code Generation Process

  AutoBayes Command Line Parameters

[1]See Section 8.2.1 for an example visualization.

- Generated code

    Interface

        Input and Output Parameters for Generated Code

        Assertions and Error Handling

    Intermediate Code

    Final Code

- Warnings/Errors

    Synthesis Constraints

    Compiler Warnings

The '`.dot`' file can be converted to a '`.jpg`' format using the '`dot`' command; for example:

% `dot -Tjpg` *filename*`.dot >` *filename*`.jpg`

6.2.3 Target Flag

This flag gives the user the option to specify the language of the generated code. By default it is set to generate C++ code, but the '`-target`' flag controls the generated code language. Available language and runtime environments are:
`c_standalone matlab modula2 octave spark`

For example:

% `autobayes -target c_standalone` *modelfilename*`.ab`

will generate output in C code and

% `autobayes -target matlab` *modelfilename*`.ab`

will generate C code which can be dynamically linked into the Matlab^{TM} environment.

6.2.4 Artificial Data Flag

The flag '`-sample`' specifies generation of a program to generate artificial sample data. See Section 4.4.

6.3 Pragmas

AutoBayes pragmas are low-level flags and commands to control specific actions in the AutoBayes system. Their main purpose is to help the developer and advanced user guide the AutoBayes system in a specific way. Pragmas are specified in the command line as follows:

% `autobayes` ... `-pragma` *pragma*=*value* ...

There should not be any spaces around the equal sign. The complete list of pragmas and allowable values for each is given in Appendix A.2. Also, a complete list of AutoBayes pragmas can be obtained using the '`help`' option:

```
% autobayes -help pragmas
```

Several pragmas control aspects of the EM algorithm: `em`, `em_log_likelihood_convergence`, `em_q_output`, and `em_q_update_simple`; see the Appendix for details and Section 8.2.1.

6.4 Compiling and Running the Generated Code

AutoBayes generates standalone C code or dynamically linked Matlab^{TM}/Octave C/C++ code. In order to provide input and run the generated stand-alone C code, a driver program is needed. The generated program needs to be compiled in order to have it run under the Matlab^{TM}/Octave systems.

If the code generated is going to run under Matlab^{TM}/Octave environment, then the following steps need to be taken after code generation:

- Load the data
- Call the AutoBayes-generated statistical model code on the data
- Generate plots and analyze the results.

In the Matlab^{TM}/Octave environment:

```
load 'data.dat'
[mu, sigma,..] = mog (...)
plot (...)
```

6.4.1 Compiling and Running the Generated Program: Octave

To run the code, we first need to compile the generated program and create an Octave file. For compiling and creating an Octave file we use the `mkoctfile` command.

% `mkoctfile -I$AUTOBAYESHOME/system/octave/include` *modelname*`.cc`

For the mixture of Gaussians example in Section 8.2.1:

```
% mkoctfile -I$AUTOBAYESHOME/system/octave/include mog.cc
```

This will produce an Octave file. We then enter the Octave environment by typing `octave` to the command line. To see the usage of synthesized code, invoke the generated code by entering the name of the model in the Octave environment:

```
 octave:1> mog
```

For our example, vector input is provided and parameter values are returned:

```
usage: [vector c,vector mu,vector rho,vector sigma] =
                mog(vector x,
                    double tolerance,
                    int maxiteration)
```

6.4.2 Compiling and Running the Generated Program: Matlab^{TM}

To run the code, we first need to compile the generated program and create a mex-file. For compiling and creating a mex-file we use the `mex` command.

% `mex -I$AUTOBAYESHOME/system/matlab/include` *modelname*`.c`

Please note *modelname*`.cc` is for Octave and *modelname*`.c` is for Matlab^{TM}.

For the mixture of Gaussians example in Section 8.2.1:

```
% mex -I$AUTOBAYESHOME/system/matlab/include mog.c
```

This command will produce a Matlab^{TM} file; then we enter the Matlab^{TM} environment by typing `matlab` to the command line. We call the generated code by typing the name of the model to the Matlab^{TM} command-line; for example:

```
>> mog
```

This returns the usage of the synthesized code. For the mixture of Gaussians example, vector input is provided and parameter values are returned:

```
usage: [vector c,vector mu,vector rho,vector sigma] =
                mog(vector x,
                    double tolerance,
                    int maxiteration)
```

6.5 AutoBayes Error and Warning Messages

At different stages the user may encounter different error messages. Below we have categorized the types of error messages the user may encounter during the process of using AutoBayes.

6.5.1 Interface (command-line) Errors

A command-line error can occur when the user sets a pragma to a wrong or unidentifiable value, or when an erroneous value is given as the argument to a flag.

6.5.2 Syntax Errors

A syntax-error message occurs when there is a syntax error in the specification of a statistical model. A syntax error could be in any segment of the model specification. Below is a list of syntax error messages with corresponding sections of specification model.

- Commented Declarations

  Error Message: '`error in declaration`'

  This could be a result of not following the conventions for making comments. The proper keyword for making comments is '`as`'.

- Equations

  Error Message: '`error in equation`'

  This could be a result of not using the proper operator which is '`:=`' for defining equations.

- Distributions

  Error Message: '`error in distribution`'

  This could be a result of not using the proper distribution operator which is '`~`' for distributions.

- Simple Declarations

  Error Message: '`error in {const, data, datastream, output, external, function} declaration`'

  This could be a result of using the wrong type for declaring a variable. Make sure only `int`, `nat`, or `double` are used.

- Constraints

  Error Message: '`error in constraint`'

  This could be a result of not using the proper keywords for setting constraints. The proper keywords are `where` and `with`.

- Approximations

  Error Message: '`error in approximation declaration`'

  This could be a result of not using the proper operators for setting approximations between two terms within a given bound. The proper keywords are: `~~`, `witherror`.

- Complexity Constraints

  Error Message : `'error in complexity declaration'`

  This could be a result of not using the proper operators for defining complexity constraints. The proper keywords are: `complexityof, withbound`.

6.5.3 Code-Generation Errors

Basically these are semantic errors that can occur during the process of code generation. A common code-generation error is '`no code generated`'. This indicates that no synthesized code has been generated.

6.6 Debugging AutoBayes Specifications

6.6.1 Running AutoBayes

`no programs generated` For the given specification, no program could be generated. This can have a multitude of reasons. For debugging, setting the pragma
`-pragma schema_control_use_generic_optimize=true`
can help. In this case, AutoBayes generates a "call" to a generic optimizer for all subproblems it cannot solve. An inspection of these pieces of code can yield helpful insights.

6.6.2 Running AutoBayes-generated Code

- A run-time message `assertion violation n_classes > 10 * n_points` means that a constraint given in the specification is violated. For example, `assertion` `violation n_classes << n_points` means the number of data points in the clustering problem is too small. Try to reduce the number of classes.

Often, this error indicates that the data vector is in the wrong order. Try to call the generated code with the transposed data array.

7. Specification Language

AutoBayes is a program synthesis tool for the statistical data analysis domain. Its input specification language is a concise description of a data analysis problem in the form of a statistical model; its output is optimized and fully documented C/C++ code which can be linked dynamically into Matlab^{TM} and Octave environments[1].

In general, the format for the statistical model input (in a file with the extension `.ab`) is as follows:

1. Model header: **model** M **as** 'model comment'.
2. Constant declarations
3. Priors (hyperparameters, distributions, constraints)
4. Data (distributions, declarations, parameters and constraints)
5. Goal, which is estimating parameters that maximize a probability term.

Chapter 8 gives many examples of AutoBayes models, which are almost self-explanatory. This chapter introduces the syntax of AutoBayes models.

7.1 Model Declarations and Syntax

Comments in the model follow the percent character '`%`'. Comments can also be enclosed in '`/*`' and '`*/`' in a C-style manner. A comment can be associated with a variable by writing **as** 'comment' (see below).

All declarations and statements end in a period ('.').

Names of constants, parameters and statistical variables can consist of the alphabetic (lower and upper case) and numeric characters, as well as underscore ("`_`"), starting with a lowercase letter, e.g., `n_points`. Index variables, usually used to quantify over a vector or matrix, must start with a capital letter, e.g. `x(I)`. This indicates that the declaration that it occurs in is true for all values of the variable; i.e., the declaration is universally quantified over `I`. For example, x(I) ~**gauss**(mu(I), sigma(I)) means

$$\forall\, 0 \leq i < length(x) : x_i \sim N(\mu_i, \sigma_i)$$

[1]Other target languages include C for stand-alone applications and Ada.

Sometimes a variable appears only once in a declaration, in which case, as in Prolog, an underscore (“`_`”) can be used instead of a variable name, but this still means universal quantification. Its meaning is similar to the MATLAB$^{\text{TM}}$ expression “`1:end`”

In order to facilitate a style more aligned with mathematical notation, the pragma `prolog_style` can be set to `false`. Then, variables can start with a lower or upper-case letter and index variables must start with an underscore (‘`_`’), e.g., `X(_i)`.

A variable may be specified with one of four *modes*:

- `const`, in which case the value of the variable will be given in the specification with a ‘`:=`’ or is an input to the generated function (or in the case it is the length of input data, it is computed from the input data given to the generated function);
- `data`, in which case it will be an input to the generated function;
- `output`, in which case it will be an output of the generated function;
- (empty).

The input and output synopsis for the generated code is established according to the following rules:

- data variables are *inputs*.
- constants are *inputs* unless

  they have an assigned value using ‘`:=`’

  they are defined by the size of a data variable, e.g., `data x(0..n)`.
- parameters for specific purposes are inputs. These are generated by AUTOBAYES according to internally-used algorithms. Typical examples are error thresholds (`tolerance`) and upper bounds for the number of iterations (`maxiteration`).
- estimated parameters in the goal statement are *outputs*.
- variables explicitly declared “output” are *outputs*.
- system-generated information, e.g., a convergence vector or the log-likelihood, are *outputs*. This is controlled by the generated algorithm or specific pragmas.
- all parameters, input and output, are sorted alphabetically.

Scalar variables, and the entries of vector or matrix-valued variables, may be of *type* **double**, **int**, or **nat**, where the latter means the value is an integer greater than or equal to zero.

Scalar variables are declared by writing:

mode type varname **as** 'COMMENT'.

where the **as** 'COMMENT' is optional but recommended.

Vector variables are declared by writing:

mode type varname(0 .. max_index)**as** 'COMMENT'.

Matrix variables are declared by writing:

mode type varname(0 .. max_row_index, 0 .. max_column_index)**as** 'COMMENT'.

All lower indices must be 0 (C-style array indexing).

A variable may be constrained by either specifying a distribution for it, a value, or by specifying range restrictions:

- var $\sim$ distribution
- expr $\sim$ distribution (the expression involves the random variable)
- var := value
- **where** range−restriction . Range restrictions may be x in min .. **max** or a boolean expression involving the variable and `=`, `<`, `>`, `=<`, `>=`, `<<`, or `>>`.

Table 7.1 lists the predefined distributions.

The `discrete` distribution takes as argument a probability vector v, say of length n. For a discrete random variable X that ranges over $[0 \ldots n-1]$, the constraint x$\sim$**discrete**(v) means

$$\forall 0 \leq j < n : Pr(X = j) = v(j)$$

The `discrete` distribution is used in conjunction with the `vector` construct. The construct vector(I := 0 .. n−1, expr(I)) creates an n element vector whose ith element is the value of `expr`(i). We have seen an example use in the Iris model in Listing 3.1:

```
class_assignment(_) ~ discrete(vector(I := 0 .. n_classes-1, phi(I))).
```

This specifies that the probability of an entry in the class assignment vector being j is $\phi(j)$, i.e.,

$$\forall 0 \leq i < \text{length(class\_assignment)} : Pr(\text{class\_assignment}(i) = j) = \phi(j)$$

AutoBayes recognizes these operators in arithmetic expressions: `+`, `-`, `*`, `/`, `**` (exponentiation), sqrt, log, exp (e^x), sum(I := min .. **max**, arith_expr(I)),

Distribution	Keyword
Bernoulli	x ∼ **bernoulli**(p)
Beta	x ∼ **beta**(alpha, **beta**)
Binomial	x ∼ **binomial**(n, p)
Cauchy	x ∼ **cauchy**(alpha, **beta**)
Discrete	x ∼ **discrete**(p(_))
Dirichlet	x ∼ **dirichlet**(alpha)
Exponential	x ∼ **exponential**(lambda)
Gamma	x ∼ **gamma**(k, theta)
Gauss	x ∼ **gauss**(mu, sigma)
Inverse Gamma	x ∼ **invgamma**(k, invtheta)
Mixture	x ∼ **mixture**(E **cases** [val1 −>dist1, ...])
Poisson	x ∼ **poisson**(lambda)
Uniform	x ∼ **uniform**(min_val .. max_val)
vonMises	x ∼ **vonmises**(N, mu, kappa)
Weibull	x ∼ **weibull**(alpha_val, beta_val)

a

Table 7.1: List of Probability Distributions

cond(bool_expr, true_arith_expr , false_arith_expr), and expr cases [val −> distribution, ...]. An example of the use of `log` for transforming data is given Listing 8.5 (Page 81). We have already seen in the Iris model in Listing 3.1 (Page 32) a use of the `sum` construct to specify the constraint that ϕ is a probability vector that sums to 1. The `cond` construct returns the value of the `true_arith_expr` if `bool_expr` evaluates to true; otherwise it returns the value of `false_arith_expr`. It is used in the random walk model in Listing 8.16 (Page 106) and in the change point detection model in Listing 8.17 (Page 107). The `cases` construct is used in the mixture model given in Listing 8.14 (Page 102).

The AutoBayes goal expression is of the form:
max pr({vars1} | {vars2}) **for** { vars3 }.
Data variables typically appear in vars1 of the expression. The goal is to find values for the variables vars3 that maximize the conditional probability expression.

Directives to AutoBayes, like pragmas, may be included in the model by writing `pragma` *pragma=value*. A list of directives is given in Appendix A. Arbitrary Prolog code may also be included in the model. It can be used to give hints to AutoBayes as to how to produce a closed-form solution to a subproblem. An example of this is given in Section 8.2.4 (Page 99).

Here we provide the set of keywords for the AutoBayes input model syntax.

pragmas	**pragma**
version header	**version**
include	**include**
comments	**as** 'comment'
distribution	expr $\sim$ distribution
assignment	var := value
vector declaration	var(0 .. max_index)
matrix declaration	var(0 .. max_row_index, 0 .. max_col_index)
expression operators	+, −, ∗, /, ∗∗, sqrt, log, exp, sum, **cond**, **cases**
vector constructor	vector(I := 0 .. n, expr(I))
types	**double**, **int**, **nat**
mode	**const**, **data**, **output**
constraints	**where**
goal	**max pr**({data_1,...} | {param_1, ...}) **for** { param_1, ... }

8. Statistical Models — Examples

In this chapter, we will present a larger number of AutoBayes specifications from various application areas. The purpose of this chapter is twofold. On one hand, these examples have been collected to demonstrate the capabilities of the AutoBayes system. For several examples, we also present the detailed mathematical derivation of the problem, as it is *automatically* generated and typeset (in LaTeX) by AutoBayes by specifying the `-tex synt` flag option. These derivations demonstrate that for the synthesis of a customized data analysis algorithm, often substantial amounts of mathematical derivations have to be performed before the problem can be solved—something traditional libraries cannot offer.

The presented examples also show how compact and flexible the specification language for AutoBayes is. Table 8.2 (Page 118) lists all examples from this section, the lengths of their specifications, file names (in the AutoBayes distribution), and the length of the generated C++ code (including comments). Although these numbers should be viewed with caution, they demonstrate AutoBayes's excellent code-to-specification ratio. Also, the point is that an AutoBayes specification is a high-level, understandable description of a statistical problem, whereas the program is a low-level description of its solution; there are cases where the solution can be expressed succinctly in closed form, but only after a lengthy derivation.

The second purpose of this chapter is to provide the user with a wealth of pre-defined specifications that can be used, modified, and refined for a specific purpose.

Notes

- Automatically generated derivations are enclosed between **begin autogenerated document** and **end autogenerated document**.
- AutoBayes generates internal variables in many places, e.g., index variables. They have the syntax `pv`n, where n is a number. For example, `pv65` is such an autogenerated variable name.

- AutoBayes's Gaussian is defined with respect to standard deviation; i.e., both x˜gauss(mu, sigma) and x˜gauss(mu, sqrt(sigma_sq)) mean $x \sim N(\mu, \sigma^2)$. This is a deviation from the form used by Matlab™ and Octave.
- In order to facilitate pretty-printing of the generated explanations, variables ending in `_sq` are assumed to be squares; e.g., `sigma_sq` is typeset as σ^2. This rule is for typesetting in LaTeX only and such variable names have no specific semantics.

8.1 Introductory Examples

8.1.1 Normal Distributed Data

This very primitive example shows the basic input syntax for AutoBayes and demonstrates its symbolic calculation capabilities: given n data points $x_1, \ldots, x_n$, which are Gaussian distributed with an unknown mean μ and variance σ^2, i.e., $x_i \sim N(\mu, \sigma^2)$, the task is to estimate the unknown μ and σ^2. Listing 8.1 shows the AutoBayes specification for this problem.

```
1  model normal as 'Normal Distributed Data'.
2
3  const nat n as 'number of data points'.
4
5  double mu as 'unknown mean'.
6  double sigma_sq as 'unknown variance'.
7           where 0 < sigma_sq.
8
9  data double x(0..n-1) as 'given data points'.
10
11 x(_) ~ gauss(mu, sqrt(sigma_sq)).
12
13 max pr( x | {mu, sigma_sq} ) for {mu, sigma_sq}).
```

Listing 8.1: AutoBayes specification for normal distributed data.

Line 1 specifies the name of the model. AutoBayes will generate a function with the name `normal`. In Line 3, the number of data points is specified to be a constant. It has to be known during runtime. Lines 5–6 specify the unknown parameters μ and σ^2 with the constraint that $\sigma^2 > 0$.

In Line 9, the input data variable x is declared. This vector of data is provided at runtime to the synthesized function. From this vector, the value of n is calculated automatically.

Line 11 describes in statistical terms the distribution of the data. The underscore _ indicates that each element of the data vector has the same distribution. Finally, Line 13 specifies the data analysis task: finding the values of μ and σ^2 that maximize the probability of the data given μ and σ^2.

It is obvious that the expected result for this problem is the following two equations (this problem can be solved in closed form):

$$\begin{aligned}\mu &= \frac{1}{n}\sum_{i=1}^{n} x_i \\ \sigma^2 &= \frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2\end{aligned}$$

AutoBayes produces a piece of code which exactly contains these sums calculated with nested `for` loops. However, these equations are not hard-coded into the system; rather, a detailed derivation of these equations is produced. In the following, the main steps of the derivation are presented, exactly as they are produced by AutoBayes (with the option `-tex synt`)[1]:

begin autogenerated document
The conditional probability $\mathbf{P}(x \mid \mu, \sigma^2)$ is under the dependencies given in the model equivalent to

$$\prod_{i=0}^{-1+n} \mathbf{P}(x_i|\mu,\sigma^2)$$

The probability occurring here is atomic and can thus be replaced by the respective probability density function given in the model. This yields the log-likelihood function

$$\log \prod_{i=0}^{-1+n} \exp\left(\frac{-\frac{1}{2}\,(x_i-\mu)^2}{(\sigma^2)^{\frac{1}{2}2}}\right)\frac{1}{\sqrt{2\,\pi}\,(\sigma^2)^{\frac{1}{2}}}$$

which can be simplified to

$$-\frac{1}{2}\,n\,\log 2 + -\frac{1}{2}\,n\,\log\pi + -\frac{1}{2}\,n\,\log\sigma^2 + \\ -\frac{1}{2}\,(\sigma^2)^{-1}\sum_{i=0}^{-1+n}(-1\,\mu + x_i)^2$$

[1]The only changes applied were to break long formulas. AutoBayes automatically typesets a variable whose name is a Greek letter as the Greek letter, and a variable whose suffix is '`_sq`' is typeset as the square of the prefix.

This function is then optimized w.r.t. the goal variables μ and σ^2.

The summands

$$-\frac{1}{2}\,n\,\log 2$$
$$-\frac{1}{2}\,n\,\log \pi$$

are constant with respect to the goal variables μ and σ^2 and can thus be ignored for maximization.

The factor

$$\frac{1}{2}$$

is non-negative and constant with respect to the goal variables μ and σ^2 and can thus be ignored for maximization.

The function

$$-1\,n\,\log \sigma^2 + -1\,(\sigma^2)^{-1}\sum_{i=0}^{-1+n}(-1\,\mu + x_i)^2$$

is then symbolically maximized w.r.t. the goal variables μ and σ^2. The partial differentials

$$\begin{aligned}\frac{\partial f}{\partial \mu} &= -2\,\mu\,n\,(\sigma^2)^{-1} + 2\,(\sigma^2)^{-1}\sum_{i=0}^{-1+n} x_i\\ \frac{\partial f}{\partial \sigma^2} &= -1\,n\,(\sigma^2)^{-1} + (\sigma^2)^{-2}\sum_{i=0}^{-1+n}(-1\,\mu + x_i)^2\end{aligned}$$

are set to zero; these equations yield the solutions

$$\begin{aligned}\mu &= n^{-1}\sum_{i=0}^{-1+n} x_i\\ \sigma^2 &= n^{-1}\sum_{i=0}^{-1+n}(-1\,\mu + x_i)^2\end{aligned}$$

end autogenerated document

8.1.2 Working with Priors

A more Bayesian flavor can be given to the above example if we assume that we have some prior knowledge about one or both of the unknown parameters μ or σ^2.

```
model normal_known_variance as 'NORMAL MODEL WITH PRIOR ON MEAN
AND KNOWN VARIANCE'.

const nat n as 'NUMBER OF DATA POINTS'.

        % PRIORS (HYPERPARAMETERS & DISTRIBUTION)

const double mu_0  as 'PRIOR ON MU (MEAN OF MEANS)'.
const double tau_0 as 'PRIOR ON MU (VARIANCE OF MEANS)'.
        where 0 < tau_0.

double mu ~ gauss(mu_0, sqrt(tau_0)).

const double sigma_sq as 'SIGMA SQUARED'.
  where 0 < sigma_sq.

data double x(0..n-1) as 'GIVEN DATA POINTS'.

x(_) ~ gauss(mu, sqrt(sigma_sq)).

max pr({x, mu} | sigma_sq) for {mu}.
```

Listing 8.2: Specification for normal distributed data with priors.

Listing 8.2 shows the specification for the case where the variance is given exactly, and we have some prior knowledge about the mean, i.e., $\mu \sim N(\mu_0, sqrt(\tau_0))$. This model has been adapted from [Bis95]. Again, AutoBayes finds the closed-form solution:

$$\mu \;=\; \mu_0 \frac{\sigma^2}{(\sigma^2 + n\,\tau_0)} + \frac{\tau_0}{(\sigma^2 + n\_points\,\tau_0)} \sum_{i=0}^{-1+n} x_i$$

```
model normal as 'NORMAL MODEL WITH CONJUGATE PRIORS'.

const nat kappa_0 as 'NUMBER PRIOR DATA POINTS'.
  where 0 < kappa_0.
const double mu_0 as 'PRIOR ON MU (MEAN OF MEANS)'.

double mu ~ gauss(mu_0, sqrt(sigma_sq/kappa_0)).

const double sigma_0_sq as 'PRIOR ON SIGMA_SQ'.
  where 0 < sigma_0_sq.
```

```
const double delta_0 as 'DEGREE OF BELIEF IN SIGMA_0_SQ'.
  where 0 < delta_0.

double sigma_sq ~ invgamma(delta_0/2+1, sigma_0_sq*(delta_0/2)).
  where 0 < sigma_sq.

const nat n_points as 'NUMBER OF DATA POINTS'.
  where 0 < n_points.

data double x(0..n_points-1) as 'CURRENT DATA POINTS (KNOWN)'.
x(_) ~ gauss(mu, sqrt(sigma_sq)).

max pr({x, mu, sigma_sq}) for {mu, sigma_sq}.
```

Listing 8.3: Specification for normal distributed data with conjugate priors

Listing 8.3 shows the specification for the case, where conjugate prior information is available for μ and $\sigma_s q$. The priors are specified as the respective conjugate priors for the Gaussian distribution, i.e., μ is distributed as Gaussian itself, and σ^2 is distributed as an inverse gamma.

The prior on μ takes a constant μ_0 as mean; this prior mean has been established from κ_0 prior observations. The standard deviation $\sqrt{\sigma/\kappa_0}$ is largely a mathematically convenient choice which in the end eliminates all dependencies between μ and σ^2. Other values should be possible as well. The prior on σ^2 takes constants σ_0^2 and δ_0 as parameters where δ_0 can be considered to be the degree of belief that σ^2 is the "right" variance. This model is adapted from the normal model with unknown parameters in [BS94, GCSR95].

begin autogenerated document
The joint probability $\mathbf{P}(\mu, \sigma^2, x)$ is under the dependencies given in the model equivalent to

$$\mathbf{P}(\mu \mid \sigma^2)\,\mathbf{P}(\sigma^2) \prod_{i=0}^{-1+n} \mathbf{P}(x_i|\mu, \sigma^2)$$

All probabilities occurring here are atomic and can thus be replaced by the respective probability density functions given in the model. This yields the log-likelihood function

$$\log \exp\left(-\frac{\frac{1}{2}\,\delta_0\,\sigma_0^2}{\sigma^2}\right)\ \exp\left(\frac{-\frac{1}{2}\,(\mu-\mu_0)^2}{(\sigma^2\,\kappa_0^{-1})^{\frac{1}{2}2}}\right)\ (\sigma^2)^{-(1+1+\frac{1}{2}\,\delta_0)}$$

$$\frac{1}{\sqrt{2\,\pi}\,(\sigma^2\,\kappa_0^{-1})^{\frac{1}{2}}}\,\frac{(\frac{1}{2}\,\delta_0\,\sigma_0^2)^{1+\frac{1}{2}\,\delta_0}}{\Gamma(1+\frac{1}{2}\,\delta_0)}\,\prod_{i=0}^{-1+n}\exp\left(\frac{-\frac{1}{2}\,(x_i-\mu)^2}{(\sigma^2)^{\frac{1}{2}2}}\right)\,\frac{1}{\sqrt{2\,\pi}\,(\sigma^2)^{\frac{1}{2}}}$$

which can be simplified to

$$\begin{aligned}-1\log\Gamma(1+\frac{1}{2}\,\delta_0)+-\frac{5}{2}\,\log\sigma^2+-\frac{3}{2}\,\log 2+-\frac{1}{2}\,\delta_0\,\sigma_0^2\,(\sigma^2)^{-1}+\\ -\frac{1}{2}\,\delta_0\,\log 2+-\frac{1}{2}\,\delta_0\,\log\sigma^2+-\frac{1}{2}\,\kappa_0\,(\sigma^2)^{-1}\,(\mu+-1\,\mu_0)^2+-\frac{1}{2}\,n\,\log 2+\\ -\frac{1}{2}\,n\,\log\pi+-\frac{1}{2}\,n\,\log\sigma^2+-\frac{1}{2}\,\log\pi+\\ -\frac{1}{2}\,(\sigma^2)^{-1}\,\sum_{i=0}^{-1+n}(-1\,\mu+x_i)^2+\frac{1}{2}\,\delta_0\,\log\delta_0+\frac{1}{2}\,\delta_0\,\log\sigma_0^2+\frac{1}{2}\,\log\kappa_0+\log\delta_0+\log\sigma_0^2\end{aligned}$$

This function is then optimized w.r.t. the goal variables μ and σ^2.

The summands

$$\begin{aligned}&-1\,\log\Gamma(1+\frac{1}{2}\,\delta_0)\\ &-\frac{3}{2}\,\log 2\\ &-\frac{1}{2}\,\delta_0\,\log 2\\ &-\frac{1}{2}\,n\,\log 2\\ &-\frac{1}{2}\,n\,\log\pi\\ &-\frac{1}{2}\,\log\pi\\ &\frac{1}{2}\,\delta_0\,\log\delta_0\\ &\frac{1}{2}\,\delta_0\,\log\sigma_0^2\\ &\frac{1}{2}\,\log\kappa_0\\ &\log\delta_0\\ &\log\sigma_0^2\end{aligned}$$

are constant with respect to the goal variables μ and σ^2 and can thus be ignored for maximization.

The factor

$$\frac{1}{2}$$

is non-negative and constant with respect to the goal variables μ and σ^2 and can thus be ignored for maximization.

The function

$$-5\,\log\sigma^2 + -1\,\delta_0\,\sigma_0^2\,(\sigma^2)^{-1} + -1\,\delta_0\,\log\sigma^2 + -1\,\kappa_0\,(\sigma^2)^{-1}\,(\mu + -1\,\mu_0)^2 + -1\,n\,\log\sigma^2 + -1\,(\sigma^2)^{-1}\sum_{i=0}^{-1+n}(-1\,\mu + x_i)^2$$

is then symbolically maximized w.r.t. the goal variables μ and σ^2. The partial differentials

$$\begin{aligned}
\frac{\partial f}{\partial \mu} &= -2\,\kappa_0\,\mu\,(\sigma^2)^{-1} + -2\,\mu\,n\,(\sigma^2)^{-1} + 2\,\kappa_0\,\mu_0\,(\sigma^2)^{-1} + 2\,(\sigma^2)^{-1}\sum_{i=0}^{-1+n} x_i \\
\frac{\partial f}{\partial \sigma^2} &= -5\,(\sigma^2)^{-1} + -1\,\delta_0\,(\sigma^2)^{-1} + -1\,n\,(\sigma^2)^{-1} + \delta_0\,\sigma_0^2\,(\sigma^2)^{-2} + \\
&\quad \kappa_0\,(\sigma^2)^{-2}\,(\mu + -1\,\mu_0)^2 + (\sigma^2)^{-2}\sum_{i=0}^{-1+n}(-1\,\mu + x_i)^2
\end{aligned}$$

are set to zero; these equations yield the solutions

$$\begin{aligned}
\mu &= \kappa_0\,\mu_0\,(\kappa_0 + n)^{-1} + (\kappa_0 + n)^{-1}\sum_{i=0}^{-1+n} x_i \\
\sigma^2 &= \delta_0\,\sigma_0^2\,(5 + \delta_0 + n)^{-1} + \kappa_0\,(5 + \delta_0 + n)^{-1}\,(\mu + -1\,\mu_0)^2 + \\
&\quad (5 + \delta_0 + n)^{-1}\sum_{i=0}^{-1+n}(-1\,\mu + x_i)^2
\end{aligned}$$

end autogenerated document

8.1.3 Combining Measurements

This example was motivated by the introduction to Kalman filters presented in Section 1.5 of [MAY79]. Suppose we have two imperfect measuring devices. Each is modeled as returning a Gaussian-distributed measurement with a known bias and standard deviation around the actual value. If a measurement is made with each, how should the two measurements be combined to obtain a better estimate of the true value? The AutoBayes model for this situation is given in Listing 8.4.

```
model biased_measurements as 'ESTIMATE TRUE VALUE GIVEN TWO BIASED
    MEASUREMENTS'.

const double bias_1.
const double bias_2.

const double sigma_1.
  where 0 < sigma_1.
const double sigma_2.
  where 0 < sigma_2.

double mu.

data double x_1.
data double x_2.

x_1 ~ gauss(mu + bias_1, sigma_1).
x_2 ~ gauss(mu + bias_2, sigma_2).

max pr( {x_1, x_2} | {mu, bias_1, bias_2, sigma_1, sigma_2} ) for {mu
    }.
```

Listing 8.4: AutoBayes specification for two biased measurements

AutoBayes is able to solve this problem symbolically, yielding

$$\frac{x_1\sigma_2^2 + x_2\sigma_1^2 - \text{bias}_1\sigma_2^2 - \text{bias}_2\sigma_1^2}{\sigma_1^2 + \sigma_2^2}$$

begin autogenerated document
The conditional probability $\mathbf{P}(x_1, x_2 \mid \mu)$ is under the dependencies given in the model equivalent to

$$\mathbf{P}(x_1 \mid \mu)\,\mathbf{P}(x_2 \mid \mu)$$

All probabilities occuring here are atomic and can thus be replaced by the respective probability density functions given in the model. This yields the log-likelihood function

$$\log \exp\left(\frac{-\frac{1}{2}\,(x_1 - (bias_1 + \mu))^2}{\sigma_1^2}\right)\ \exp\left(\frac{-\frac{1}{2}\,(x_2 - (bias_2 + \mu))^2}{\sigma_2^2}\right)\ \frac{1}{\sigma_1\,\sqrt{2\,\pi}}\,\frac{1}{\sigma_2\,\sqrt{2\,\pi}}$$

which can be simplified to

$$-1\,\log 2 + -1\,\log\pi + -1\,\log\sigma_1 + -1\,\log\sigma_2 + -\frac{1}{2}\,{\sigma_1}^{-2}\,(x_1 + -1\,\mathit{bias}_1 + -1\,\mu)^2 \\ + -\frac{1}{2}\,{\sigma_2}^{-2}\,(x_2 + -1\,\mathit{bias}_2 + -1\,\mu)^2$$

This function is then optimized w.r.t. the goal variable μ. The summands

$-1\,\log 2$

$-1\,\log\pi$

$-1\,\log\sigma_1$

$-1\,\log\sigma_2$

are constant with respect to the goal variable μ and can thus be ignored for maximization. The factor

$$\frac{1}{2}$$

is non-negative and constant with respect to the goal variable μ and can thus be ignored for maximization.

The function

$$-1\,{\sigma_1}^{-2}\,(x_1 + -1\,\mathit{bias}_1 + -1\,\mu)^2 + -1\,{\sigma_2}^{-2}\,(x_2 + -1\,\mathit{bias}_2 + -1\,\mu)^2$$

is then symbolically maximized w.r.t. the goal variable μ. The differential

$$-2\,\mathit{bias}_1\,{\sigma_1}^{-2} + -2\,\mathit{bias}_2\,{\sigma_2}^{-2} + -2\,\mu\,{\sigma_1}^{-2} + -2\,\mu\,{\sigma_2}^{-2} + 2\,x_1\,{\sigma_1}^{-2} + 2\,x_2\,{\sigma_2}^{-2}$$

is set to zero; this equation yields the solution

$$-1\,\mathit{bias}_1\,\sigma_2^2\,(\sigma_1^2+\sigma_2^2)^{-1} + -1\,\mathit{bias}_2\,\sigma_1^2\,(\sigma_1^2+\sigma_2^2)^{-1} + x_1\,\sigma_2^2\,(\sigma_1^2+\sigma_2^2)^{-1} + x_2\,\sigma_1^2\,(\sigma_1^2+\sigma_2^2)^{-1}$$

end autogenerated document

As part of its optimization phase, AutoBayes generates code that computes common subexpressions only once, as is evident in the generated C/C++ code:

```
pv0 = sigma_2 * sigma_2;
pv1 = sigma_1 * sigma_1;
mu = (x_1 * pv0 + x_2 * pv1 - bias_1 * pv0 - bias_2 * pv1) / (pv0 + pv1);
```

8.1.4 Transformations: log-normal and square-normal

Data which has undergone some transformations can be modeled in AutoBayes. A typical example is log-normal distributed data (Listing 8.5). Other examples for transformation of the input data are:

- the log-it transformation. Here, the ratio of $x_i/(1-x_i)$ is log-normal distributed. The corresponding AutoBayes line is as:
  log(x(I)/(1−x(I))) ∼**gauss**(mu, sqrt(sigma_sq)).
- the square transformation with a distribution. Here, the square of each data value, x_i, is normal distributed. In AutoBayes-syntax, we write
  x(I)**2 ∼**gauss**(mu, sqrt(sigma_sq)).

Note that the current version of AutoBayes only allows the user to specify a limited set of transformations.

```
model log_normal as 'Log-normal model'.

... % specification as "normal" above

data double x(0..n_points-1) as 'current data points (known)'.
log(x(_)) ~ gauss(mu, sqrt(sigma_sq)).

max pr(x | {mu, sigma_sq}) for {mu, sigma_sq}.
```

Listing 8.5: AutoBayes specification for log-normal distributed data.

8.1.5 Other distributions: Cauchy

Of course, data are not always Gaussian distributed. Rather, some other probability density function for the data (or even a mixture) is required. A typical data analysis task which requires a non-Gaussian data model has been adapted from [Siv96], attributed to [Gul88]:

> "A lighthouse is somewhere off a piece of straight coastline [of given length] at a position $light_x$ along the shore and a distance $light_y$ out at sea. It emits a series of short highly collimated flashes at random intervals and hence at random azimuths. These pulses are intercepted on the coast by photo-detectors [each at position x_i] that record only the fact that a flash has occurred, but not the angle from which it came. $N_{flashes}$ have so far been recorded at positions $\{x(i)\}$. Where is the lighthouse?"

Listing 8.6 captures this problem and synthesizes code to estimate the position of the lighthouse.

```
model lighthouse as 'Lighthouse example [Sivia96]'.

const nat length    as 'length of the shore'.
const nat n_flashes as 'number of flashes'.

        %
        % Priors (hyperparameters & distribution)
        %
double light_x as 'x-position of the lighthouse'.
double light_y as 'y-position of the lighthouse'.

light_x ~ uniform(-length/2, length/2).
light_y ~ uniform(0, length/2).

        %
        % Data
        %
data double x(0..n_flashes - 1) as 'x-positions of triggered sensors'.
x(_) ~  cauchy(light_x, light_y).

        %
        % Goal
        %
max pr(x | {light_x, light_y}) for {light_x, light_y}.
```

Listing 8.6: Lighthouse example

8.1.6 Discrete

Typical examples with discrete probability distributions always include models of tossed coins. Tossing one coin with a bias can be modelled using the Bernoulli distribution (Listing 8.7). A real-valued *bias* with values between 0 and 1 is defined to describe the bias. Since the coin is tossed only once, the value of `head` can only be 0 or 1 (coin lands on head). This (somewhat degenerate) example obviously calculates the value of `head` as 1 if bias ≥ 0.5 and 0 otherwise. This example shows that AutoBayes can find simple closed-form solutions. Listing 8.8 shows the straight-forward generalization to tossing a biased coin n times. Note that in both specifications, the declaration of a scalar statistic variable (`heads`) and the definition of its pdf can be done in one statement.

If the bias is not known, AutoBayes can estimate it using prior information. In this case, a beta-distributed prior on the value of *bias* is used. Its parameters are a prior on getting head or getting tails. Listing 8.9 shows the AutoBayes specification.

```
model biased_coin as 'Biased coin toss model'.

data double bias as 'bias towards heads'.
        where 1 > bias.
        where 0 < bias.

nat heads ~ bernoulli(bias).
        where 1 ≥ heads.
        where 0 =< heads.
        where heads in 0..1.

max pr(heads | bias) for heads.
```

Listing 8.7: AutoBayes model for tossing a biased coin.

```
model biased_coins as 'Biased coin toss model (multiple tosses)'.

const nat n as 'number of tosses'.

data double bias as 'bias towards heads'.
        where 1 > bias.
        where 0 < bias.

        % Head count
nat heads ~ binomial(n, bias).
        where n ≥ heads.
        where 0 =< heads.
        where heads in 0..n.

max pr(heads | {n, bias}) for heads.
```

Listing 8.8: AutoBayes model for repeatedly tossing a biased coin.

```
model biased_coins_prior as 'Biased coin toss model with prior'.

const nat n as 'number of tosses'.

const nat prior_tails as 'prior count of tails'.
        where 0 < prior_tails.
const nat prior_heads as 'prior count of heads'.
        where 0 < prior_heads.

double bias as 'bias towards heads'.
        where 1 > bias.
        where 0 < bias.

bias ~ beta(prior_tails, prior_heads).

```

```
nat heads ~ binomial(n, bias).
        where n ≥ heads.
        where 0 =< heads.
        where heads in 0..n.

max pr({heads, bias}) for {heads, bias}.
```

Listing 8.9: AUTOBAYES model for tossing a biased coin with prior.

8.2 Clustering Examples

8.2.1 Mixture of Gaussians

The "mixture of Gaussians" [EH81] specification is a good example of how a straightforward and simple problem specification unfolds into a complex, iterative algorithm. Listing 8.10 shows the specification of the problem: n_points data points x_i have been generated by n_classes different sources. The data from each source c are normal distributed, i.e., $x_i \sim N(\mu_c, \sigma_c^2)$. All parameters of the model, i.e., μ_c, σ_c, and the relative class frequency ϕ_c are unknown and must be determined.

```
model mog as 'MIXTURE OF GAUSSIANS'.

        % MODEL PARAMETERS
const nat n_points as 'NUMBER OF DATA POINTS'.
        where 0 < n_points.
const nat n_classes as 'NUMBER OF CLASSES'.
        where 0 < n_classes.
        where n_classes ≪ n_points.

        % CLASS PROBABILITIES
double phi(0..n_classes-1).
        where 0 = sum(I := 0 .. n_classes-1, phi(I))-1.

        % CLASS PARAMETERS
double mu(0..n_classes-1).
double sigma(0..n_classes-1).
        where 0 < sigma(_).

        % HIDDEN VARIABLE
output nat c(0..n_points-1) as 'CLASS ASSIGNMENT VECTOR'.
c(_) ~ discrete(vector(I := 0 .. n_classes-1, phi(I))).

        % DATA
data double x(0..n_points-1).
x(I) ~ gauss(mu(c(I)), sigma(c(I))).

```

```
max pr(x|{sigma, mu, phi}) for {sigma, mu, phi}.
```

Listing 8.10: AutoBayes model for a mixture of Gaussians.

The individual parts of the specification have already been described in Section 3.1 on page 31. Since this example is relevant for a large number of AutoBayes problems, the mathematical derivation and the assembly of the clustering algorithm will be discussed in detail.

When AutoBayes processes this model, a number of internal steps are executed by AutoBayes in order to solve this optimization task. In a first step, a Bayesian Network (BN) for the problem is constructed. Figure 8.1 shows a graphical representation of the resulting Bayesian Network. AutoBayes generates this graphical representation whenever it is called with the `-designdoc` or `-dot` command flags.

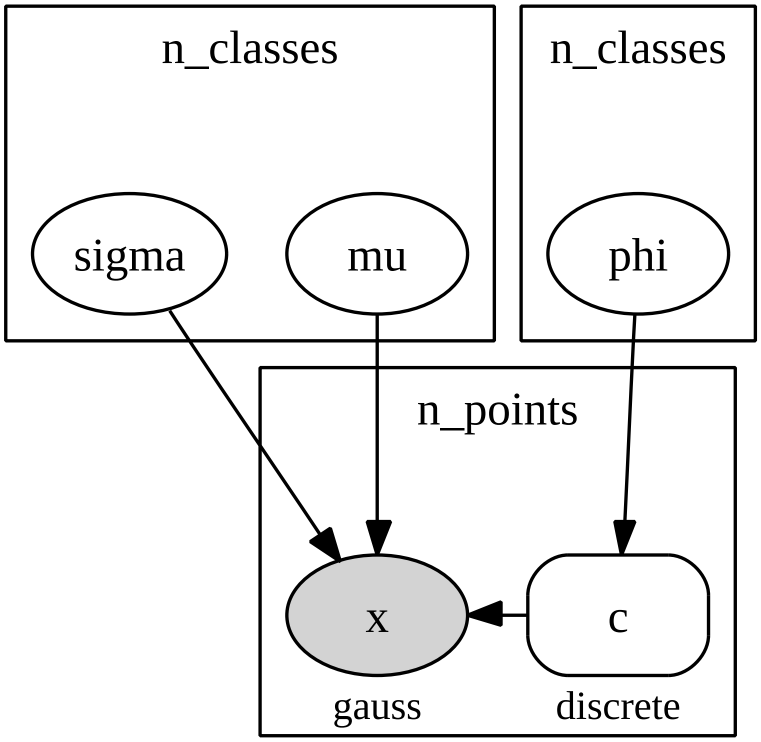


Figure 8.1: Bayesian Network for Mixture of Gaussians. This graph has been auto-generated by AutoBayes and had been visualized using GraphViz.

Statistical variables are shown as vertices in this network, known (data) variables as shaded ellipses. A hidden variable (in our case `c`) is displayed as a rectangle with rounded edges. Since all variables are actually vectors, their dimension and i.i.d status is shown with rectangles in the background, which are labeled with the variable dimension. As usual, arrows mark the dependencies between the variables. From this figure, it is obvious that the problem can be broken down into two subproblems, which can be solved separately:

max pr(c | phi) for phi
max pr(x | {c, mu, sigma}) for {mu, sigma}

In its schema-base, AutoBayes contains a number of rules on how to partition and simplify extended Bayesian Networks. Furthermore, AutoBayes recognizes that this problem describes a discrete latent variable problem (often called a hidden variable problem). In our case, the (known) data variable is x, the hidden variable the class membership vector c.

With these two observations, AutoBayes can now start to solve the problem. Being a hidden variable problem, the application of an instance of a discrete EM algorithm [MK97] can solve this problem.

The model describes a discrete latent (or hidden) variable problem with the latent variable c and the data variable x. The problem to optimize the conditional probability $\mathbf{P}(x \mid \mu, \phi, \sigma)$ w.r.t. the variables μ, ϕ, and σ can thus be solved by an application of the (discrete) EM-algorithm.

The algorithm defines and maintains as central data structure, a class membership table q, such that q_{ij} is the probability that data point x_i belongs to class j, i.e., $q_{ij} = \mathbf{P}([c_i = j])$.

The algorithm consists of an initialization phase for q (Section 8.2.1), followed by a convergence phase, the EM loop, followed by the extraction of the hidden variable c.

Initialization of EM

The q matrix is of the size `n_points` $\times$ `n_classes` and must be initialized before the EM-loop starts. A number of different initialization methods can be selected using the pragma `em`:

`center` In this mode, a center-based initialization is attempted. This means that for each class $0 \leq j \leq$ `n_classes` $- 1$, a center-point is chosen randomly, i.e., $ct_j = x_{rnd}$[2]. Then, all elements of q are initialized with the normalized distance

[2]Please note that this kind of initialization could result in data points could be picked more than

between the data value and the chosen point ct_j. For $0 \leq i \leq \texttt{n\_points} - 1$

$$q_{ij} = \frac{\sqrt{(ct_j - x_i)^2}}{\sum_{k=0}^{\texttt{n\_points}-1} \sqrt{(ct_j - x_k)^2}}$$

`sharp_class` This initialization starts with a random assignment of the class vector c, i.e., $c_j = rnd$ for $0 \leq j \leq \texttt{n\_classes} - 1$. Then the q matrix is initialized such that $q_{ic_i} = 1$ and zero everywhere else ($0 \leq i \leq \texttt{n\_points} - 1$). This means that the EM algorithm starts with a q matrix that is zero almost everywhere.

`fuzzy_class` This initialization also starts with a random assignment of the class vector c, i.e., $c_j = rnd$ for $0 \leq j \leq \texttt{n\_classes} - 1$. Then, however, only a portion δ of the probability is put into q_{ic_i}, the rest is uniformly (but randomly) distributed over the rest of the elements in q. In order to obey the constraint that $\sum_k q_{ik} = 1$, the following algorithm is used:

$$q_{ij} = \begin{cases} \frac{1}{\texttt{n\_classes}}(1 - \delta) \times rnd & j \neq c_i \\ 1 - \sum_{k=0,k\neq j}^{\texttt{n\_classes}-1} q_{ik} & j = c_i \end{cases}$$

`no_pref` This option lets AutoBayes select one of the initialization methods.

Note that there are many possibilities to initialize the q matrix. AutoBayes can be easily extended by schemas to perform other kinds of initialization.

The EM Loop

The EM loop is the central optimization iteration of the algorithm. Each iteration is comprised of two individual steps, the E-step and the M-step. The M-step maximizes the probability expressions and estimates values for the unknown parameters μ, σ, and ϕ. The E-step calculates the "expectation" and updates the q matrix. In contrast to many other implementations, AutoBayes first starts with the M-step followed by the E-step.

The iteration loop is executed until

(a) a given maximal number of iterations has been performed. This number is given as an input parameter `maxiteration` to the generated code, OR

(b) if the iteration metric E is smaller than the given parameter `tolerance`.

once, potentially leading to numerical instability.

In any of these two cases, the EM loop is terminated and the required parameters are extracted and returned.

Although a number of different ways to calculate the "error" E could be used, AutoBayes currently supports two mechanisms.

- The error is calculated as the sum of the normalized differences between all parameters of the current and the previous run. For run t, we have

$$E = \sum_{j=0}^{\texttt{n\_classes}-1} \frac{|\mu_j^t - \mu_j^{t-1}|}{|\mu_j^t| + |\mu_j^{t-1}|} + \sum_{j=0}^{\texttt{n\_classes}-1} \frac{|\sigma_j^t - \sigma_j^{t-1}|}{|\sigma_j^t| + |\sigma_j^{t-1}|} + \sum_{j=0}^{\texttt{n\_classes}-1} \frac{|\phi_j^t - \phi_j^{t-1}|}{|\phi_j^t| + |\phi_j^{t-1}|}$$

- The normalized difference of the log-likelihood L between the current and the previous run is taken, i.e.,
$$E = \frac{|L^t - L^{t-1}|}{|L^t| + |L^{t-1}|}$$
This iteration metric can be activated by setting
`-pragma em_log_likelihood_convergence=true`.
Although the effort in calculation of this metric is higher, the EM algorithm converges usually much faster.

In the following, we present the autogenerated derivation of the M-step, as it contains "the meat" of the problem.

begin autogenerated document
The problem to optimize the conditional probability $\mathbf{P}(c, x \mid \mu, \phi, \sigma)$ w.r.t. the variables μ, ϕ, and σ can under the given dependencies by Bayes rule be decomposed into two independent subproblems:

$\max \mathbf{P}(c \mid \phi)$ for ϕ
$\max \mathbf{P}(x \mid c, \mu, \sigma)$ for μ, σ

The conditional probability $\mathbf{P}(c \mid \phi)$ is under the dependencies given in the model equivalent to

$$\prod_{i=0}^{-1+n\_points} \mathbf{P}(c_i|\phi)$$

The probability occurring here is atomic and can thus be replaced by the respective probability density function given in the model. Summing out the expected variable

c_{pv10} yields the log-likelihood function

$$\sum\nolimits_{j\,\in\,\mathrm{dom}\ c_i \sim q(i,j)}^{i=0\ldots-1+n\_points} \log \prod_{k=0}^{-1+n\_points} \phi_{c_k}$$

which can be simplified to

$$\sum_{i=0}^{-1+n\_classes} \log \phi_i \sum_{j=0}^{-1+n\_points} q(j,i)$$

This function is then optimized w.r.t. the goal variable ϕ.

The expression

$$\sum_{i=0}^{-1+n\_classes} \log \phi_i \sum_{j=0}^{-1+n\_points} q(j,i)$$

is maximized w.r.t. the variable ϕ under the constraint

$$0 = -1 + \sum_{i=0}^{-1+n\_classes} \phi_i$$

using the Lagrange-multiplier l.

The summand

$$l$$

is constant with respect to the goal variable ϕ_{pv21} and can thus be ignored for maximization. The function

$$-1\,l \sum_{i=0}^{-1+n\_classes} \phi_i + \sum_{i=0}^{-1+n\_classes} \log \phi_i \sum_{j=0}^{-1+n\_points} q(j,i)$$

is then symbolically maximized w.r.t. the goal variable ϕ_{pv21}. The differential

$$-1\,l + \phi_{pv21}^{-1} \sum_{i=0}^{-1+n\_points} q(i,pv21)$$

is set to zero; this equation yields the solution

$$l^{-1} \sum_{i=0}^{-1+n\_points} q(i,pv21)$$

The conditional probability $\mathbf{P}(x \mid c, \mu, \sigma)$ is under the dependencies given in the model equivalent to

$$\prod_{i=0}^{-1+n\_points} \mathbf{P}(x_i | c_i, \mu, \sigma)$$

The probability occurring here is atomic and can thus be replaced by the respective probability density function given in the model. Summing out the expected variable c_{pv10} yields the log-likelihood function

$$\sum\nolimits_{j\,\in\,\mathrm{dom}\ c_i \sim q(i,j)}^{i=0\ldots-1+n\_points} \log \prod_{k=0}^{-1+n\_points} \exp\left(\frac{-\frac{1}{2}\left(x_k - \mu_{c_k}\right)^2}{\sigma_{c_k}^2}\right) \frac{1}{\sigma_{c_k}\sqrt{2\,\pi}}$$

which can be simplified to

$$-1 \sum_{i=0}^{-1+n\_classes} \log \sigma_i \sum_{j=0}^{-1+n\_points} q(j,i) + -\frac{1}{2}\, n\_points\, \log 2 + -\frac{1}{2}\, n\_points\, \log \pi + -\frac{1}{2} \sum_{i=0}^{-1+n\_classes} \sigma_i^{-2} \sum_{j=0}^{-1+n\_points} (-1\,\mu_i + x_j)^2\, q(j,i)$$

This function is then optimized w.r.t. the goal variables μ and σ.

The summands

$$-\frac{1}{2}\, n\_points\, \log 2$$
$$-\frac{1}{2}\, n\_points\, \log \pi$$

are constant with respect to the goal variables μ and σ and can thus be ignored for maximization.

Index decomposition: The function

$$-1 \sum_{i=0}^{-1+n\_classes} \log \sigma_i \sum_{j=0}^{-1+n\_points} q(j,i) + -\frac{1}{2} \sum_{i=0}^{-1+n\_classes} \sigma_i^{-2} \sum_{j=0}^{-1+n\_points} (-1\,\mu_i + x_j)^2\, q(j,i)$$

can be optimized w.r.t. the variables μ_i and σ_i element by element (i.e., along the index variable i) because there are no dependencies along that dimension.

The factor

$$n\_classes$$

is non-negative and constant with respect to the goal variables μ_{pv31} and σ_{pv31} and can thus be ignored for maximization.

The function

$$-1\,\log\sigma_{pv31}\sum_{i=0}^{-1+n\_points} q(i,pv31) + -\frac{1}{2}\,\sigma_{pv31}^{-2}\sum_{i=0}^{-1+n\_points}(-1\,\mu_{pv31}+x_i)^2\,q(i,pv31)$$

is then symbolically maximized w.r.t. the goal variables μ_{pv31} and σ_{pv31}. The partial differentials

$$\begin{aligned}
\frac{\partial f}{\partial\mu_{pv31}} &= -1\,\mu_{pv31}\,\sigma_{pv31}^{-2}\sum_{i=0}^{-1+n\_points} q(i,pv31) + \sigma_{pv31}^{-2}\sum_{i=0}^{-1+n\_points} x_i\,q(i,pv31)\\
\frac{\partial f}{\partial\sigma_{pv31}} &= -1\,\sigma_{pv31}^{-1}\sum_{i=0}^{-1+n\_points} q(i,pv31) + \sigma_{pv31}^{-3}\sum_{i=0}^{-1+n\_points}(-1\,\mu_{pv31}+x_i)^2\,q(i,pv31)
\end{aligned}$$

are set to zero; these equations yield the solutions

$$\begin{aligned}
\mu_{pv31} &= cond(0 = \sum_{i=0}^{-1+n\_points} q(i,pv31),\\
&\quad fail(division\_by\_zero),\\
&\quad \sum_{i=0}^{-1+n\_points} q(i,pv31)^{-1}\sum_{i=0}^{-1+n\_points} x_i\,q(i,pv31))\\
\sigma_{pv31} &= cond(0 = \sum_{i=0}^{-1+n\_points} q(i,pv31),\\
&\quad fail(division\_by\_zero),\\
&\quad \frac{1}{2}\,4^{\frac{1}{2}}\sum_{i=0}^{-1+n\_points}(-1\,\mu_{pv31}+x_i)^2\,q(i,pv31)^{\frac{1}{2}}\sum_{i=0}^{-1+n\_points} q(i,pv31)^{-\frac{1}{2}})
\end{aligned}$$

end autogenerated document

Extracting the Hidden Variable

After the EM-loop has terminated, it has calculated the most likely values of the (unknown) parameters μ, σ, ϕ, as well as the (internal) matrix q. If the most likely class assignment, the hidden variable c is desired, AutoBayes performs the following calculation. For $0 \leq i \leq$ `n_points` $- 1$

$$c_i = \mathrm{argmax}_j\, q_{ij}$$

The q matrix itself is returned using the pragma `-pragma em_q_output=true`. Please note that the name of this internal matrix cannot be accessed from the AutoBayes specification language.

8.2.2 Multivariate Mixture of Gaussians

Whenever data sets with more than one statistical variable needs to be clustered, a multivariate mixture model has to be used. In AutoBayes, multivariate mixture models can be specified in two ways:

- each individual variable is specified by its name; means and standard deviations for each variable are returned separately
- all variables are packed into a 2-dimensional matrix. This matrix has the dimension: $N_{dimensions} \times N_{datapoints}$. AutoBayes then returns a matrix for the means and standard deviation of size $N_{dimensions} \times N_{classes}$.

A typical example for multivariate clustering, the Iris data set, has been discussed in detail in Section 3. There, we used the approach of packing the given data into a matrix "`data`".

All multivariate mixture models can have variants with respect to the independence of the dimensions. In general, a multivariate Gaussian is defined as $X \sim N(\mu, \Sigma)$ where μ is a vector of the means, and Σ is a $N_{dimensions} \times N_{dimensions}$ matrix, the covariance matrix. Often, however, only the diagonal elements of the covariance matrix are of interest, i.e., $\Sigma_{ij} = 0$ for $i \neq j$. Please note that the current version of AutoBayes cannot handle the case with full covariance, i.e., $N(\mu, \Sigma)$.

Listing 8.11 shows the AutoBayes specification of a multivariate mixture of Gaussians. Its input is a data matrix `sim_data` of the dimensions number of data points times number of features (or variables). The goal of the specification is to estimate, given the desired number of classes *n_classes*, the most probable class assignment and the class parameters (μ, σ^2). This model is the basis model for clustering of software simulation data as described in Section 1.2.1 and [GBSMB08].

Typically, this model is processed with the following flags and pragmas:

`-instrument` display and save the error value during each iteration of the EM algorithm. The data can be used to monitor the convergence behavior of the algorithm.

`-pragma em_log_likelihood_convergence=true` This option forces the EM algorithm to use the current log-likelihood as its convergence metric. The algorithm stops, when the change in the log-likelihood becomes smaller than the given threshold `tolerance`.

`-pragma em=...` selects the initialization routine for the EM algorithm as discussed above in Section 8.2.1. By default, a center-based initialization is selected.

As with any iterative statistical or numerical algorithm, there are some possible caveats, when AutoBayes is used on mixture models.

- The number of classes must be specified before the run of the EM algorithm. This model thus does not estimate the most probable number of classes. However, with a simple Matlab^{TM}/Octave script, which iterates over a range of classes (e.g., 2:10) and monitoring of the returned log-likelihood, the best number of classes can easily be estimated.
- If the given number of classes is larger than is prompted by the data, the algorithm may return multiples of classes with identical parameters. The reason is that the generated algorithm does not automatically reduce the number of classes in case they become empty. Reducing the number of classes avoids this problem.
- Numerical instability and return of 'NaN' can occur if the range of values in a data variable becomes too large. The reason is that internally, ratios of expressions of the form $e^{x-\mu}$ have to be formed for input data x. If the value of $x - \mu$ becomes too large or to small, NaNs or division-by-zero exceptions can occur.

  This problem can circumvented by normalizing the data prior to processing. Typical ways to do this in Matlab^{TM}/Octave using a data vector x is:

  - $x_n = \frac{(x-\min(x))}{\max(x)-min(x)}$ produces a 0-1 normalized data vector x_n.
  - $x_n = \frac{(x-\mu_x)}{\sigma_x^2}$ produces a $N(0,1)$ normalized data vector x_n.

```
model mult_cluster as
        'SIMPLE MULTIVARIATE CLUSTERING MODEL'.

const nat n_variables as 'NUMBER OF VARIABLES'.
const nat n_points as 'NUMBER OF DATA POINTS'.

const nat n_classes as 'NUMBER OF CLASSES'.
        where 0 < n_classes.
        where n_classes ≪ n_points.


double phi(0..n_classes-1) as 'CLASS PROBABILITIES'.
        where 0 = sum(I := 0 .. n_classes-1, phi(I))-1.

        % CLASS PARAMETERS
double mu(0..n_variables-1, 0..n_classes-1).

double sigma(0..n_variables-1, 0..n_classes-1).
  where 0 < sigma(_,_).

output nat class_assignment(0..n_points-1) as 'HIDDEN VARIABLE'.
class_assignment(_) ~ discrete(vector(I := 0 .. n_classes-1, phi(I))).

data double sim_data(0..n_variables-1, 0..n_points-1).

sim_data(C,I) ~ gauss(mu(C,class_assignment(I)), sigma(C,
    class_assignment(I))).

        % GOAL

max pr({ sim_data } | { phi, mu, sigma }) for { phi, mu, sigma }.
```

Listing 8.11: Multivariate clustering of Gaussians

8.2.3 Working with Priors

As with other statistical models (see Section 8.1.2), mixture models can have priors. In this section, we will discuss how conjugate priors on the means and the standard deviations for each class can be used within AutoBayes (for a one-dimensional problem). Listing 8.12 shows the entire specification. Most of the specification is similar to the standard one-dimensional mixture of Gaussians. The priors are defined by the additional (known) variables (each variable is a vector over the classes): μ_0 as the mean of the means, κ_0 as confidence of μ_0, and σ_0 and δ_0 as the prior for the standard deviation. Then, the model parameters μ and σ^2 are distributed as:

$$\mu \sim N(\mu_0, \kappa_0 \sigma^2)$$

and

$$\sigma^2 \sim \Gamma^{-1}(1 + \frac{\delta_0}{2}, \frac{1}{2}\sigma_0\delta_0)$$

The goal of the specification now looks different, namely

```
max pr({mu, sigma_sq , x} | phi) for {phi , mu, sigma_sq }.
```

For comparison, the goal specification of the standard mixture model is

```
max pr(x|{phi , mu, sigma}) for {phi , mu, sigma}.
```

The internal, autogenerated derivation is now getting much more involved, as all the information about the priors has to be taken into account. Below, we show the second half of the derivation, namely for maximizing $P(\mu, \sigma^2, x|c)$ under the independence assumption, i.e., the class frequency (ϕ) has been factored out already.

begin autogenerated document
The conditional probability $\mathbf{P}(\mu, \sigma^2, x \mid c)$ is under the dependencies given in the model equivalent to

$$\prod_{i=0}^{-1+C} \mathbf{P}(\mu_i|\sigma_i^2) \prod_{i=0}^{-1+C} \mathbf{P}(\sigma_i^2) \prod_{i=0}^{-1+N} \mathbf{P}(x_i|c_i, \mu, \sigma^2)$$

All probabilities occurring here are atomic and can thus be replaced by the respective probability density functions given in the model. Summing out the expected variable c_{pv11} yields the log-likelihood function

$$\sum_{j \in \mathrm{dom}\ c_i \sim q(i,j)}^{i=0 \ldots -1+N} \log \prod_{k=0}^{-1+C} \exp\left(\frac{-\frac{1}{2}(\mu_k - \mu_0)^2}{(\kappa_0\,(\sigma_k^2)^{\frac{1}{2}})^2}\right) \frac{1}{\kappa_0\,(\sigma_k^2)^{\frac{1}{2}}\sqrt{2\,\pi}}$$
$$\prod_{k=0}^{-1+C} \exp\left(-\frac{\frac{1}{2}\,\delta_0\,\sigma_0}{\sigma_k^2}\right) (\sigma_k^2)^{-(1+1+\frac{1}{2}\,\delta_0)} \frac{(\frac{1}{2}\,\delta_0\,\sigma_0)^{1+\frac{1}{2}\,\delta_0}}{\Gamma(1+\frac{1}{2}\,\delta_0)}$$
$$\prod_{k=0}^{-1+N} \exp\left(\frac{-\frac{1}{2}(x_k - \mu_{c_k})^2}{(\sigma_{c_k}^2)^{\frac{1}{2}2}}\right) \frac{1}{\sqrt{2\,\pi}\,(\sigma_{c_k}^2)^{\frac{1}{2}}}$$

which can be simplified to

$$\begin{aligned}
&-1\,C\,\log\kappa_0 + -1\,C\,\log\Gamma(1+\tfrac{1}{2}\,\delta_0)+\\
&C\,\log\delta_0 + C\,\log\sigma_0 + -\tfrac{5}{2}\,\textstyle\sum_{i=0}^{-1+C}\log\sigma_i^2+\\
&-\tfrac{3}{2}\,C\,\log 2 + -\tfrac{1}{2}\,\delta_0\,C\,\log 2+\\
&-\tfrac{1}{2}\,\delta_0\,\sigma_0\,\textstyle\sum_{i=0}^{-1+C}\left(\sigma_i^2\right)^{-1} + -\tfrac{1}{2}\,\delta_0\,\textstyle\sum_{i=0}^{-1+C}\log\sigma_i^2 + -\tfrac{1}{2}\,C\,\log\pi+\\
&-\tfrac{1}{2}\,N\,\log 2 + -\tfrac{1}{2}\,N\,\log\pi + -\tfrac{1}{2}\,\kappa_0^{-2}\,\textstyle\sum_{i=0}^{-1+C}(-1\,\mu_0+\mu_i)^2\left(\sigma_i^2\right)^{-1}+\\
&-\tfrac{1}{2}\,\textstyle\sum_{i=0}^{-1+C}\log\sigma_i^2\,\textstyle\sum_{j=0}^{-1+N}q(j,i)+\\
&-\tfrac{1}{2}\,\textstyle\sum_{i=0}^{-1+C}\left(\sigma_i^2\right)^{-1}\,\textstyle\sum_{j=0}^{-1+N}(-1\,\mu_i+x_j)^2\,q(j,i)+\\
&\tfrac{1}{2}\,\delta_0\,C\,\log\delta_0 + \tfrac{1}{2}\,\delta_0\,C\,\log\sigma_0
\end{aligned}$$

This function is then optimized w.r.t. the goal variables μ and σ^2.

The summands

$$-1\,C\,\log\kappa_0$$
$$-1\,C\,\log\Gamma(1+\frac{1}{2}\,\delta_0)$$
$$C\,\log\delta_0$$
$$C\,\log\sigma_0$$
$$-\frac{3}{2}\,C\,\log 2$$
$$-\frac{1}{2}\,\delta_0\,C\,\log 2$$
$$-\frac{1}{2}\,C\,\log\pi$$
$$-\frac{1}{2}\,N\,\log 2$$
$$-\frac{1}{2}\,N\,\log\pi$$
$$\frac{1}{2}\,\delta_0\,C\,\log\delta_0$$
$$\frac{1}{2}\,\delta_0\,C\,\log\sigma_0$$

are constant with respect to the goal variables μ and σ^2 and can thus be ignored for maximization.

Index decomposition

The function

$$\begin{array}{l}
-\frac{5}{2}\sum_{i=0}^{-1+C}\log\sigma_i^2 + -\frac{1}{2}\,\delta_0\,\sigma_0\sum_{i=0}^{-1+C}(\sigma_i^2)^{-1} + \\
-\frac{1}{2}\,\delta_0\sum_{i=0}^{-1+C}\log\sigma_i^2 + -\frac{1}{2}\,\kappa_0^{-2}\sum_{i=0}^{-1+C}(-1\,\mu_0+\mu_i)^2\,(\sigma_i^2)^{-1} + \\
-\frac{1}{2}\sum_{i=0}^{-1+C}\log\sigma_i^2\sum_{j=0}^{-1+N}q(j,i) + \\
-\frac{1}{2}\sum_{i=0}^{-1+C}(\sigma_i^2)^{-1}\sum_{j=0}^{-1+N}(-1\,\mu_i+x_j)^2\,q(j,i)
\end{array}$$

can be optimized w.r.t. the variables μ_{pv37} and σ^2_{pv37} element by element (i.e., along the index variable *pv37*) because there are no dependencies along that dimension.

The function

$$\begin{array}{l}
-5\,\log\sigma^2_{pv37} + -1\,\delta_0\,\sigma_0\,(\sigma^2_{pv37})^{-1} + -1\,\delta_0\,\log\sigma^2_{pv37} + -1\,\log\sigma^2_{pv37}\sum_{i=0}^{-1+N}q(i,pv37) + \\
-1\,\kappa_0^{-2}\,(-1\,\mu_0+\mu_{pv37})^2\,(\sigma^2_{pv37})^{-1} + -1\,(\sigma^2_{pv37})^{-1}\sum_{i=0}^{-1+N}(-1\,\mu_{pv37}+x_i)^2\,q(i,pv37)
\end{array}$$

is then symbolically maximized w.r.t. the goal variables μ_{pv37} and σ^2_{pv37}. The partial differentials

$$\begin{aligned}
\frac{\partial f}{\partial\mu_{pv37}} &= -2\,\mu_{pv37}\,\kappa_0^{-2}\,(\sigma^2_{pv37})^{-1} + -2\,\mu_{pv37}\,(\sigma^2_{pv37})^{-1}\sum_{i=0}^{-1+N}q(i,pv37) \\
&\quad +2\,\mu_0\,\kappa_0^{-2}\,(\sigma^2_{pv37})^{-1} + 2\,(\sigma^2_{pv37})^{-1}\sum_{i=0}^{-1+N}x_i\,q(i,pv37) \\
\frac{\partial f}{\partial\sigma^2_{pv37}} &= -5\,(\sigma^2_{pv37})^{-1} + -1\,\delta_0\,(\sigma^2_{pv37})^{-1} + -1\,(\sigma^2_{pv37})^{-1}\sum_{i=0}^{-1+N}q(i,pv37) \\
&+\ \delta_0\,\sigma_0\,(\sigma^2_{pv37})^{-2} + \kappa_0^{-2}\,(-1\,\mu_0+\mu_{pv37})^2\,(\sigma^2_{pv37})^{-2} \\
&+\ (\sigma^2_{pv37})^{-2}\sum_{i=0}^{-1+N}(-1\,\mu_{pv37}+x_i)^2\,q(i,pv37)
\end{aligned}$$

are set to zero; these equations yield the solutions

$$\begin{aligned}
\mu_{pv37} &= \mu_0\,(1+\kappa_0^2\sum_{i=0}^{-1+N}q(i,pv37))^{-1} + \kappa_0^2\,(1+\kappa_0^2\sum_{i=0}^{-1+N}q(i,pv37))^{-1}\sum_{i=0}^{-1+N}x_i\,q(i,pv37) \\
\sigma^2_{pv37} &= \delta_0\,\sigma_0\,(5+\delta_0+\sum_{i=0}^{-1+N}q(i,pv37))^{-1} + \kappa_0^{-2}\,(5+\delta_0 \\
&+ \sum_{i=0}^{-1+N}q(i,pv37))^{-1}\,(-1\,\mu_0+\mu_{pv37})^2 + (5+\delta_0+\sum_{i=0}^{-1+N}q(i,pv37))^{-1}
\end{aligned}$$

$$\times \sum_{i=0}^{-1+N} (-1\,\mu_{pv37} + x_i)^2\, q(i, pv37)$$

end autogenerated document

```
model mgp_mu as 'Mixture of Gaussians (with priors)'.

const nat n_points as 'Number of data points'.
        where 0 < n_points.

const nat n_classes as 'Number of classes'.
        where 0 < n_classes.
        where n_classes ≪ n_points.


double phi(0..n_classes-1).
        where 0 = sum(I := 0 .. n_classes-1, phi(I))-1.


const double mu_0    as 'prior on mu'.
const double kappa_0 as 'prior on mu'.
  where 0 < kappa_0.

double mu(0..n_classes-1).
mu(I) ~ gauss(mu_0, sqrt(sigma_sq(I)) * kappa_0).


const double sigma_0 as 'prior on sigma_sq'.
  where 0 < sigma_0.
const double delta_0 as 'prior on sigma_sq'.
  where 0 < delta_0.

double sigma_sq(0..n_classes-1) ~ invgamma(delta_0/2+1, sigma_0*(
    delta_0/2)).
        where 0 < sigma_sq(_).


nat c(0..n_points-1) as 'class assignment vector'.

c(_) ~ discrete(vector(I := 0 .. n_classes-1, phi(I))).

data double x(0..n_points-1).

x(I) ~ gauss(mu(c(I)), sqrt(sigma_sq(c(I)))).

max pr({mu, sigma_sq, x} | phi) for {phi, mu, sigma_sq}.
```

Listing 8.12: Mixture of Gaussians with priors.

8.2.4 Working with Non-Gaussian and Multiple Distributions

Until now, we only considered mixture models, where data were Gaussian distributed. In many applications, this does not have to be the case. Typical examples include discrete features with binomial distribution, or sensor readings with an exponential distribution. In general, two cases can be distinguished:

- different distributions along the different data dimensions. Here, different data sources have different distributions, but the distribution is the same for all drawn data. An example is a two-dimensional data set of temperature (Gaussian distributed) and thermostat status (on/off, discrete binomial distribution).
- data points have a different distribution according to the data source (=class). For example, a sensor returns data which is exponentially distributed. Background noise (from the same sensor), however, is Gaussian distributed. The statistical model now is required to separate the "good" data from the background noise.

Both kinds of problems can be solved by AutoBayes, as we show in the following sections.

Non-Gaussian mixtures

The specification of a mixture model for non-Gaussian distributions looks very similar to one for Gaussian distribution (Listing 8.10). Only the line, specifying the distribution of the data has to be modified and, of course, the names and numbers of distribution parameters has to modified accordingly. Table 8.1 shows a list of available (and tested) distributions available. Please note that for some distributions, no closed form solution for the M step exists or can be found by the symbolic system. In these cases, a numerical optimization routine (usually a Nelder-Mead Simplex Algorithm) is instantiated. Other distributions, most notably the von Mises-Fisher distribution requires additional "help" to find a closed form solution. These expressions are given in the input specification, as Listing 8.13 shows. This hint to solution is taken from a published paper [BaJGS05], where this result is a major result of the paper.

```
...
data double x(0..n_dim-1,0..n_points-1).
x(J, I) ~ vonmises(n_dim, mu(J,c(I)), k(c(I))).

%---------------------------------------------------------------------------
assert(synth_formula([k(I), mu(_,_)], Formula, Constraint,
block(
  local([
        scalar(PV1, double, []),
```

Name	Notation	Closed Form	Remarks
Bernoulli	$x \sim bernoulli(p)$	Y	
Beta	$x \sim beta(\alpha, \beta)$	N	[1]
Binomial	$x \sim binomial(n, p)$	Y	[2]
Cauchy	$x \sim cauchy(x, y)$	N	[1]
Exponential	$x \sim exp(\lambda)$	Y	
Gamma	$x \sim gamma(k, \theta)$	Y	k known
Gamma	$x \sim gamma(k, \theta)$	N	[1]
Gauss	$x \sim gauss(\mu, \sigma^2)$	Y	
Poisson	$x \sim poisson(\lambda)$	Y	
vonMises	$x \sim vonmises(\mu, k)$	Y	[3]
Weibull	$x \sim weibull(\alpha, \beta)$	N	[1]

Table 8.1: Different distributions for mixture models. For some distributions, closed-form solutions are found by AutoBayes, for others not. [1] AutoBayes has to be called with `-pragma schema_control_arbitrary_init_values=true` to obtain iterative solution. [2] PATCHED version of AutoBayes necessary. [3] solution requires customized schema (see text). [4] semi-supported in alpha version.

```
          scalar(PV2, double, []),
          scalar(R, double, []),
          scalar(I1, int, []),
          scalar(I2, int, []),
          vector(V, double, [dim(0,n_dim - 1)], [])
          ]),

series([
          assign(PV1, 0, []),
          for([idx(I1,0,n_dim - 1)],
                    series([
                      assign(select(V, [I1]), 0, []),
                      for([idx(I2, 0, n_points - 1)],
                             assign_plus(select(V,[I1]),
                               *([q(I2, I), x(I1, I2)]),[]),
                             []),
                      assign_plus(PV1, *([select(V, [I1]), select(V, [I1])
                          ]), [])
                      ], []),
                    []),
           assign(PV1, sqrt(PV1),[]),
          assign(PV2, 0, []),
          for([idx(I1,0, n_points - 1)],
                    assign_plus(PV2, q(I1, I), []),
                []) ,
          for ([idx(I1,0, n_dim - 1)],
                assign(select(mu,[I1,I]), select(V,[I1]) / PV1, []),
                []) ,
          assign(R, *([PV1, PV2 ** (-1)]), []) ,
          assign(select(k,[I]),
            (( *([R, n_dim]) - *([R, R, R])) /
            (1 - *([R, R]))), [
                comment(['Approximation of k according to ',
                        '[BanerjeeEtAl03].'])])
          ], []), [
                comment(['Optimization of the expression ',
                expr(Formula), pp_nl,
                'under the constraints ', expr(Constraints),
                'has been given explicitely in the specification.',
                'For a detailed derivation of this solution see ',
                '[Banerjee et al 2003]. A (recursive) approximation ',
                'for k is used: ',
                expr(k_i = (r*n - r*r*r)/(1-r*r))
                ])
                ]
)
)).
```

Listing 8.13: Expressions to support vonMises-Fisher distributions.

Mixture of Distributions along Variables

Using different distributions along the different variables can be specified in AutoBayes in a straight-forward way. Different variable names are used for the different distributions as in the following specification snippet

```
data x_g(I) ~ gauss(mu(c(I)),sqrt(sigma_sq(c(I)))).
data x_e(I) ~ exponential(lambda(c(I))).
...
```

Then the goal statement looks like
max pr({x_g,x_e}|{phi,mu,sigma,lambda} **for** {phi,mu,sigma,lambda}.

A typical example, where this kind of models is necessary, when the data set consists of measurements from different sensors: some of the measurements are Gaussian distributed (e.g., pressure), whereas others are discrete, e.g., valve-open, switch-position. For example, for a rocket the current thrust might be Gaussian distributed, whereas the boolean flag “motor on/off” is certainly not. In many cases, a boolean variable (or a discrete variable in general) can be modelled as a Gaussian by adding Gaussian noise to it: $x_{ab}(I) = x(I) + normal_{rnd}(0, \sigma^2_{noise})$. A more concise and correct specification, however, would directly define the input data vector x using a binomial distribution, where the probability p is unknown and different for each class c:

```
x(I) ~ binomial(1, p(c(I))).
```

Mixture of Distributions along Classes

For this specification variant the AutoBayes construct **cases** is used. The following specification snippet shows the central part of a mixture of beta and Gaussian distributed data. The full specification is shown in Listing 8.14.

```
data nat x(0..n_points-1).

x(I) ~ mixture(c(I) cases
                 [ 0 -> beta(a, b),
                   1 -> gauss(mu, sigma)
                 ]).
```

In this example, we have two classes. Data belonging to Class 1 are Beta distributed with parameters a and b and those, belonging to Class 2 are Gaussian distributed. Please note that for this specification, set the pragma `schema_control_arbitrary_init_values` to true in order to allow AutoBayes to produce a numerical solution.

```
model mix_beta_gauss as 'Disjoint Mixture between Beta and Gaussian'.

const nat n_points as 'Number of data points'.
        where 0 < n_points.

% Mixing proportions
double phi.
        where 0 < phi.
        where phi < 1.

% Parameters
double a.
        where 0 < a.
double b.
        where 0 < b.

double mu.
double sigma.
        where 0 < sigma.

% Hidden variable
nat c(0..n_points-1) ~ bernoulli(phi) as 'class assignment vector'.
```

```


% DATA
data nat x(0..n_points-1).

x(I) ~ mixture(c(I) cases
                [ 0 -> beta(a, b),
                  1 -> gauss(mu, sigma)
                ]).
        where 0 =< x(_).

% GOAL

max pr(x | {mu, sigma, a, b, phi}) for {mu, sigma, a, b, phi}.
```

Listing 8.14: Mixture of Betas (Class 0) and Gaussians (Class 1).

8.2.5 Multinomial Principal Components Analysis (MPCA)

This is a *k*-means version of the algorithm, without sparse vectors. For details see [BFG03].

```
model mpca as 'MULTINOMIAL PRINCIPLE COMPONENT ANALYSIS (PCA)'.

const nat n_points as 'NUMBER OF POINTS'.
        where 0 < n_points.
const nat n_classes as 'NUMBER OF CLASSES'.
        where 0 < n_classes.
        where n_classes ≪ n_points.
const nat n_feats as 'NUMBER OF WORDS IN EACH DOCUMENT'.
        where 0 < n_feats.
const nat n_words as 'NUMBER OF DISTINCT WORDS'.
        where 0 < n_words.

const double alpha(0..n_classes-1) as 'DIRICHLET PARAMETERS'.

%
%   PARAMETERS:   DISTRIBUTION OVER WORDS
%
double omega(0..n_classes-1,0..n_words-1).
        where 0 =< omega(_,_).
        where 1 = sum(J := 0 .. n_words-1,omega(_,J)).

%
%   HIDDEN VARIABLE M THE TOPIC PROPORTIONS
%
double m(0..n_points-1,0..n_classes-1).
```

```
          where 0 =< m(_,_).
          where 1 = sum(J := 0 .. n_classes-1,m(_,J)).

%
%   HIDDEN VARIABLE K  THE ASSIGNED TOPIC FOR EACH WORD
%
nat k(0..n_points-1,0..n_feats-1).
           where 0 =< k(_,_).
           where k(_,_) < n_classes.

k(I,_) ~ discrete(vector(J := 0 .. n_classes-1,m(I,J))).

%
%  DATA
%
data double x(0..n_points-1,0..n_feats-1).
           where 0 =< x(_,_).
           where x(_,_) < n_words - 1.

x(I,J) ~ discrete(vector(K := 0 .. n_words-1,omega(k(I,J),K))).

max pr({x}|{m,omega}) for {m,omega}.
```

Listing 8.15: AutoBayes model for MCPA

8.3 Time Series Analysis

The models discussed so far, are time independent. In nature and engineering, however, many statistical processes are *time series*. This means they are of the form $x_{t_0}, \ldots, x_t, x_{t+\Delta t}, \ldots$, where t is the time, which is incremented in discrete steps of Δt. Please note that there are many different kinds of time series. Here, we only discuss discrete time series.

Typical examples for discrete time series are sequences of events, or sequences of (noisy) sensor measurements made at a certain sampling rate. Typical data analysis problems are concerned with the detection of basic model parameters (and their change over time).

Although the underlying mechanism of Bayesian Networks (BN) is very powerful to handle time series data, AutoBayes's capabilities in handling of such data is currently fairly restricted. The following sections illustrate which analyses on time series can be performed with AutoBayes.

The most striking restriction for AutoBayes is that all data have to be processed in batch mode, i.e., all data $x_{t_0}, \ldots, x_t, \ldots, x_N$ must be presented to the generated code as one vector of data. In contrast, recursive algorithms can process each data value x_t individually at a given time. Thus, recursive data analysis algorithms are much more amenable to processing streaming data, i.e., data coming in at a specific rate.

8.3.1 Random Walk

A simple random walk can be described by the following equations:

$$\begin{aligned} x_0 &= 0 \\ x_t &= x_{t-1} + \eta \quad \text{for } t > 0 \end{aligned}$$

For a standard random walk, η is Gaussian distributed as $\eta \sim N(0, \sigma^2)$. A biased random walk is described by a similar set of equations. However, a noisy *drift* factor pushes the values of x_t toward a specific "direction". We have

$$\begin{aligned} x_0 &= 0 \\ x_t &= x_{t-1} + b + \nu \quad \text{for } t > 0 \end{aligned}$$

where b is the bias and $\nu \sim N(0, \sigma^2)$.

With this information, we can construct the AutoBayes model shown in Listing 8.16. In this model, a random walk with n_points data points is processed. The aim of the model is to estimate the drift drift_rate and the noise drift_error given the data drift. Thus the goal of the specification is given as

```
max pr( { drift } | { drift_rate , drift_error} )
        for { drift_rate , drift_error }.
```

The distribution of the data is specified exactly as shown in the equations above. Here, the **cond** keyword is used to distinguish the first data element from the subsequent ones.

```
drift(I) ~ gauss(cond(I>0,drift(I-1),0)+drift_rate , drift_error).
```

```
model walk as 'SIMPLE RANDOM WALK WITH BIAS'.

const nat n_points.
  where 0 < n_points.
  where n_points > 1.

%     PARAMETERS
double drift_rate as 'RATE OF DRIFT PER TIME SPLICE'.
double drift_error as 'STANDARD DEVIATION OF DRIFT PER TIME SLICE'.
        where drift_error > 0.

%     DISTRIBUTION
data double drift(0..n_points-1) as 'DRIFT OR GYRO ANGULAR ERROR'.
drift(I) ~ gauss(cond(I>0,drift(I-1),0)+drift_rate , drift_error).


max pr( { drift } | { drift_rate , drift_error} )
        for { drift_rate , drift_error }.
```

Listing 8.16: AutoBayes model for a biased random walk.

8.3.2 Change Point Detection

An important task in the analysis of time series is the detection of an abrupt change. Here, the most probable index $t, 0 \leq t < N$ is estimated where a change occurs. Classical examples include accident rates in coal mining, which changes abruptly after a new safety measure has been introduced, or estimating the point in time when sensor readings go bad.

Listing 8.17 shows the AutoBayes specification (adapted from [OF96]) to detect the most probable change point. Here the process is described by

$$x_t = \begin{cases} N(\mu_1, \sigma^2) & \text{for } t < t_{sw} \\ N(\mu_2, \sigma^2) & \text{for } t \geq t_{sw} \end{cases}$$

In this simple model, the (unknown) noise remains constant over the change.

```
model hinckley as 'Gaussian change point analysis'.

const nat n_points.
  where 0 < n_points.

nat switchpt.
  where switchpt in 1..n_points-2.
  %
  % The following constraint are implied by the range constraint but
      not
  % yet inferred...
  %
  where 0 < switchpt.
  where switchpt < n_points - 1.
  where switchpt < n_points.

double mu1.
double mu2.

double sigma_sq.
  where 0 < sigma_sq.

data double x(0..n_points-1).

x(I) ~ gauss(cond(I < switchpt, mu1, mu2), sqrt(sigma_sq)).

max pr(x|{mu1, mu2, sigma_sq, switchpt}) for {mu1, mu2, sigma_sq,
    switchpt}.
```

Listing 8.17: AutoBayes model for a simple detection of a change point

Of course, variations of this change-point detection model can be specified. In the following, we will just mention some ideas and leave the exact specification as an exercise for the reader.

Instead of a change in the mean value μ, the noise characteristics σ^2 can change abruptly. This can be the case if a sensor suddenly produces a large amount of noise (e.g., due to a broken cable or damaged amplifier. Then, we have

$$x_t = \begin{cases} N(\mu, \sigma_1^2) & \text{for } t < t_{sw} \\ N(\mu, \sigma_2^2) & \text{for } t \geq t_{sw} \end{cases}$$

Also, the detection of a change-point from a constantly growing value to a constant value can be specified easily. A practical example for such a specification is the detection of the CAS-mach transition in aircraft trajectory (Chapter 1.2.2). Here, the values of the data stream are growing with a constant (but unknown) rate x_r, until it switches over to a constant (again unknown) value x_c. The mathematical formulation for this problem is (assuming the noise σ^2 is constant and known).

$$x_t = \begin{cases} N(x_c + x_r(t - t_{sw}), \sigma^2) & \text{for } t < t_{sw} \\ N(x_c, \sigma^2) & \text{for } t \geq t_{sw} \end{cases}$$

A full specification of this problem will be given in the next section, where we will discuss finding the most probable change point in two statistical process variables.

8.3.3 Change Points in Multiple Variables

The detection of CAS-mach transition as discussed in Chapter 1.2.2 calls for a monitoring of two variables: the calibrated air speed (CAS) and the mach number. Listing 1.3 on page 19 shows how these variables develop, when the aircraft is climbing: before the change point t_0, the aircraft climbs with a constant (but noisy) airspeed cas_0. Due to the physics of the atmosphere, at the same time, the mach number increases linearly with an unknown rate $mach_r$. After the aircraft passed the (unknown) transition point t_0, the mach number will remain constant $mach_0$ and the airspeed will decrease linearly. We obtain the two formulas:

$$\begin{aligned} cas_t &= \begin{cases} cas_0 & \text{for } t \leq t_0 \\ cas_0 - cas_r(t - t_0) & \text{for } t > t_0 \end{cases} \\ mach_t &= \begin{cases} mach_0 - mach_r(t - t_0) & \text{for } t \leq t_0 \\ mach_0 & \text{for } t > t_0 \end{cases} \end{aligned}$$

In order to obtain the most likely unknown parameters, we have to optimize for all parameters simultaneously by

$$\max Pr(\langle cas_t, mach_t \rangle | mach_0, mach_r, cas_0, cas_r)$$

Listing 8.18 shows the full specification for this problem.

```
model climb_transition as 'MAG->CAS TRANSITION FOR CLIMB SCENARIOS'.

const nat n_points.
  where 0 < n_points.

nat t_0.
  where t_0 < n_points - 1.
  where 2 < t_0.
  where t_0 in 3..n_points-3.

double mach_0.
double mach_r.
double cas_0.
double cas_r.

const double sigma_sq.
  where 0 < sigma_sq.

data double mach(0..n_points-1).
data double cas(0..n_points-1).

mach(I) ~ gauss(
            cond(I < t_0,
                mach_0 - (I-t_0)*mach_r,
                mach_0
                ),
           sqrt( sigma_sq)).
cas(I) ~ gauss(
           cond(I < t_0,
                cas_0,
                cas_0 - (I-t_0)*cas_r
                ),
            sqrt(sigma_sq)).

max pr({mach,cas}|{mach_r, mach_0, cas_0, cas_r, t_0})
        for {mach_0, mach_r, cas_0, cas_r, t_0}.
```

Listing 8.18: AutoBayes specification for the detection of the CAS-mach transition

The heart of the algorithm, which is generated by AutoBayes is code for solving a large and complicated quadratic equation. By abbreviating subexpressions, which occur more than once, the code can be kept compact. If additional information about the unknown parameters are known, we can add prior information to the specification. Listing 8.19 shows that only few lines of the specification have to be added. Basically, we specify that the unknown parameters cas_r and $mach_r$ are Gaussian distributed around some known mean $\mu_{mach_r}, \mu_{cas_r}$ and a certain confidence $\kappa_{mach_r}, \kappa_{cas_r}$.

```
model climb_transition_prior as
    'MAC->CAS TRANSITION FOR CLIMB SCENARIOS WITH PRIORS'.

const nat n_points.
  where 0 < n_points.

nat t_0.
  where t_0 < n_points - 1.
  where 2 < t_0.
  where t_0 < n_points.
  where t_0 in 3..n_points-3.

double mach_0.
double mach_r.
const double mu_mach_r.
const double kappa_mach_r.
        where 0 < kappa_mach_r.

mach_r ~ gauss(mu_mach_r,sqrt(sigma_sq)*kappa_mach_r).

double cas_0.
double cas_r.
const double mu_cas_r.
const double kappa_cas_r.
        where 0 < kappa_mach_r.

cas_r ~ gauss(mu_cas_r,sqrt(sigma_sq)*kappa_cas_r).

const double sigma_sq.
  where 0 < sigma_sq.

data double mach(0..n_points-1).
data double cas(0..n_points-1).

mach(I) ~ gauss(
          cond(I < t_0,
              mach_0 - (I-t_0)*mach_r,
              mach_0
              ),
          sqrt(sigma_sq)).
cas(I) ~ gauss(
         cond(I < t_0,
              cas_0,
              cas_0 - (I-t_0)*cas_r
              ),
          sqrt(sigma_sq)).

max pr({mach,cas,mach_r,cas_r}|{mach_c, cas_c, switch_pt})
        for {mach_c, mach_r, cas_c, cas_r, switch_pt}.
```

Listing 8.19: Specification for the detection of the CAS-mach transition with priors

8.3.4 Kalman Filters

Kalman filters are recursive least-square algorithms for the estimation of a process state, given a noisy process model and noisy measurements [GA01, BH97]. The automatic generation of code for Kalman filters is the domain of AutoFilter [TOMS04]. However, simple variants of Kalman filters can be easily specified using AutoBayes. One restriction, however, should be noted: traditionally, Kalman filter algorithms work on-line, i.e., they process one measurement or temporal update step at at time. AutoBayes can only generate batch-mode filters, where all data of the time series is present at the same time (given as a vector to the algorithm).

```
model kalman as 'Simplest possible Kalman filter'.

const nat n_points.
  where 0 < n_points.

data double likelihood.

double drift(0..n_points-1) as 'DRIFT'.
drift(I) ~ gauss(cond(I>0, drift(I-1), 0),
                 cond(I>0, 2.0, 1.0)).

     %     QUANTITY BEING PREDICTED, THE ``NEXT" POINT
double drift_next as `future drift'.
DRIFT_NEXT ~ GAUSS(DRIFT(N_POINTS-1), 1.0).

CONST DOUBLE MEAS_ERROR AS 'std.dev. of measurement'.

     %   GAUSSIAN OBSERVATION (MEASUREMENT)
DATA DOUBLE MEAS(0..N_POINTS-1) AS 'measurement'.
MEAS(I) ~ GAUSS(DRIFT(I), MEAS_ERROR).

     %     QUANTITY BEING PREDICTED
DOUBLE MEAS_NEXT AS 'future measurement'.
MEAS_NEXT ~ GAUSS(MEAS(N_POINTS-1), MEAS_ERROR).

MAX PR( { DRIFT_NEXT, MEAS_NEXT, DRIFT } | {  MEAS } )
        FOR { DRIFT_NEXT, MEAS_NEXT, DRIFT  }.
```

Listing 8.20: AutoBayes model for a simple Kalman filter.

Listing 8.20 shows a specification for an extremely simple Kalman filter to estimate the drift rate of a gyro, for example. The drift is defined as a Gaussian random walk (vector), here just with constant σ^2 of 1 and 2. Note the usage of **cond** to specify the recursive equation. The estimation of the next drift value in time drift_next is again Gaussian distributed, as well as the measurements. All unknown parameters, the estimated sequence of the drift values as well as the estimated next drift and next measurement, are calculated, given the sequence of measurements in form of a matrix. Please note, that AutoBayes does not generate online recursive algorithms for Kalman filters.

8.3.5 Kalman Filters with Failure modes

A Kalman filter can also detect if and when sensors (or combination of sensors) fail, even if this failure is not directly observable. Listing 8.21 shows a somewhat complicated example of an aircraft sensor suite consisting of a directional and a pitch gyro as well as a turn indicator. The AutoBayes specification estimates the next states of the system and the most likely failure points for the individual sensors.

```
model dgtc as 'DIRECTIONAL AND PITCH GYRO PLUS TURN COORDINATOR MODEL'
    .

const nat n_points.
  where 0 < n_points.

data double likelihood.

%   DIRGYRO_FAILURE_PT = CHANGE POINT FOR DIRECT. GYRO FAILING
%       PARAMETERS
const double dirgyro_failure_prob as 'PROBABILITY OF FAILURE OF DIR.
    GYRO'.
     dirgyro_failure_prob := 0.5.
nat dirgyro_failure_pt.
  where dirgyro_failure_pt in 1..n_points.
  where dirgyro_failure_pt < n_points+1.
  where 0<dirgyro_failure_pt.
dirgyro_failure_pt ~ discrete(vector(I := 1 .. n_points,
        cond(I<n_points, dirgyro_failure_prob/(n_points-1),
                         1.0-dirgyro_failure_prob))).

%   TC_FAILURE_PT = CHANGE POINT FOR TURN COORDINATOR FAILING
%       PARAMETERS
const double tc_failure_prob as 'PROBABILITY OF FAILURE OF TURN
    COORDINATOR'.
     tc_failure_prob := 0.5.
nat tc_failure_pt.
  where tc_failure_pt in 1..n_points.
  where tc_failure_pt < n_points+1.
  where 0<tc_failure_pt.
tc_failure_pt ~ discrete(vector(I := 1 .. n_points,
        cond(I<n_points,tc_failure_prob/(n_points-1),1.0-
            tc_failure_prob))).
```

```


% ELECT_FAILURE_PT = CHANGE POINT FOR ELECTRICAL FAILING
%     PARAMETERS
const double elect_failure_prob as 'PROBABILITY OF FAILURE OF
    ELECTRICAL'.
     elect_failure_prob := 0.5.
data nat elect_failure_pt.
  where elect_failure_pt in 1..n_points.
  where elect_failure_pt < n_points+1.
  where 0 < elect_failure_pt.
elect_failure_pt ~ discrete(vector(I := 1 .. n_points,
         cond(I<n_points,elect_failure_prob/(n_points-1),
                          1.0-elect_failure_prob))).

% DRIFT = BIASED RANDOM WALK FOR BOTH DIRECT. AND PITCH GYROS
%     PARAMETERS, INCLUDING THE BASE CASE
const double drift_rate as 'RATE OF DRIFT PER TIME SLICE'.
drift_rate := 0.1.
const double drift_error as 'STANDARD DEVIATION OF DRIFT PER TIME
    SLICE'.
         where drift_error > 0.
drift_error := 1.0.
const double drift_base as 'DRIFT INITIALIZATION'.
drift_base := 0.
const double drift_xbase_error as 'DRIFT INITIALIZATION OF STANDARD
    ERROR'.
drift_xbase_error := drift_error*2.

%     DISTRIBUTION WITH BASE CASE AND RECURSIVE CASE
%     EXPLICITLY DEFINED USING COND()
double dirdrift(0..n_points-1) as 'DRIFT OR DIRGYRO ANGULAR ERROR'.
dirdrift(I) ~ gauss(cond(I>0,dirdrift(I-1),drift_base)+drift_rate,
                    cond(I>0,drift_error,drift_xbase_error)).

%     QUANTITY BEING PREDICTED, THE "NEXT" POINT
double dirdrift_next as 'FUTURE DIRDRIFT'.
dirdrift_next ~ gauss(dirdrift(n_points-1)+drift_rate, drift_error).


% ANGLEDIFF = DIFFERENCE OF ANGLE
%     DISTRIBUTION
double anglediff(0..n_points-1) as 'ANGLE'.
anglediff(I) ~ gauss(cond(I>0,anglediff(I-1),0), 0.02).

%     QUANTITY BEING PREDICTED
double anglediff_next as 'FUTURE ANGLE DIFFERENCE'.
anglediff_next ~ gauss(anglediff(n_points-1), 0.02).
```

```


% ANGLE = unbiased random walk
%    distribution
double angle(0..n_points-1) as 'ANGLE'.
angle(I) ~ gauss(cond(I>0,angle(I-1)+anglediff(I-1),0), 0.01).

%    quantity being predicted
double angle_next as 'FUTURE ANGLE'.
angle_next ~ gauss(angle(n_points-1)+anglediff(n_points-1), 0.01).


% DIRGYRO = gaussian conditioned on angle and dirdrift;
%          failure causes the dirgyro to be stuck at last good data

const double gyro_error as 'ERROR RATE FOR GYROS'.
  gyro_error := 0.1.

const double tc_error as 'ERROR RATE FOR TURN COORDINATOR'.

data double dirgyro(0..n_points-1) as 'DIRECTIONAL GYRO MEASUREMENT'.
dirgyro(I) ~ gauss(cond(and([I<dirgyro_failure_pt]),
                        angle(I)+dirdrift(I),
                        angle(dirgyro_failure_pt-1)+dirdrift(
                            dirgyro_failure_pt-1)
                         ),
                        gyro_error).


% TURN_COORD = gaussian conditioned on anglediff

data double turn_coord(0..n_points-1) as
         'TURN COORDINATOR - RATE OF CHANGE OF BANK'.

turn_coord(I) ~ gauss(cond(and([I<tc_failure_pt, I<elect_failure_pt]),
                              anglediff(I),
                              0),
                          tc_error).

max pr( { dirgyro_failure_pt, tc_failure_pt,
          dirgyro, turn_coord, angle_next, dirdrift_next,
              anglediff_next} |
        {elect_failure_pt} )
    for { angle_next, dirdrift_next, anglediff_next,
        dirgyro_failure_pt,
          tc_failure_pt}.
```

Listing 8.21: AutoBayes model for a Kalman filter with sensor failures.

8.4 Reliability Models

Statistical models in software engineering can be used to model failure rates, mean-time-between-failure, and reliability of a piece of software. The models described in this section comprise the standard models found in the literature. The AutoBayes models have been developed by B. Fischer.

Listing 8.22 shows the specification for the basic Jelinski/Moranda (a.k.a., de-eutrophication) model [JM72]. In this model, the elapsed time between failures is modeled by an exponential distribution; the hazard rate is assumed to be proportional to the number of errors remaining in the software and to be constant between two consecutive failures. The Jelinski/Moranda model is thus a finite failure, concatenated failure rate model.

The error repair process is modeled as immediate and perfect, i.e., after each failure the responsible error is identified and repaired and testing resumes only after the repair, and attempts to repair an error are always successful and do not introduce new errors.

Listing 8.22 is a close transcription of the original model, only the parameter loc as number of lines of code has been added. Its purpose is to limit the number of errors (to be less than the number of lines in the code).

Listing 8.23 is a Jelinski-Moranda model with conjugate priors as described in [MS83].

The Goel-Okumoto model ([GO78], Listing 8.24) modifies the basic Jelinski-Moranda model by introducing a parameter p for the error repair rate, i.e., the probability that a detected error is successfully repaired. This relaxes the "perfect repair" assumption of Jelinski-Moranda; however, the assumption that no new bugs are introduced during repair still holds. Obviously, for p = 1, the Jelinski-Moranda model appears as special case of the Goel-Okumoto model.

```
model jm as '(Basic) Jelinski-Moranda model'.

const nat loc as 'lines of code'.
        where n < loc.

double n_error as 'initial number of errors'.
        where n =< n_error.
        where n_error =< loc.

double c as 'fault detection rate'.
        where 0 < c.

% Observation and distribution
```

```
const nat n as 'NUMBER OF OBSERVATIONS'.
        where 0 < n.

data double x(0..n-1) as 'INTERFAILURE TIMES (ELAPSED TIME BETWEEN
    ERRORS)'.
        where 0 < x(I).

x(I) ~ exponential(c * (n_error - I)).

max pr(x|{n_error, c}) for {n_error, c}.
```

Listing 8.22: Jelinski-Moranda software reliability model

```
model ms as 'JELINSKI-MORANDA MODEL WITH CONJUGATE PRIORS'.

double n_error as 'INITIAL NUMBER OF ERRORS'.
        where n =< n_error.
        where n_error =< loc.

double c as 'FAULT DETECTION RATE'.
        where 0 < c.

% PRIORS
const double theta as 'PRIOR ON N_ERROR'.
        where 0 < theta.

n_error ~ poisson(theta).

const double mu    as 'PRIOR ON C (SCALE PARAMETER)'.
const double alpha as 'PRIOR ON C (SHAPE PARAMETER)'.
        where 0 < mu.
        where 0 < alpha.

c ~ gamma(mu, alpha).

% OBSERVATION AND DISTRIBUTION
const nat n as 'NUMBER OF OBSERVATIONS'.
        where 0 < n.

const nat loc as 'LINES OF CODE'.
        where n < loc.

data double x(0..n-1) as 'INTERFAILURE TIMES (ELAPSED TIME BETWEEN
    ERRORS)'.
        where 0 < x(I).

x(I) ~ exponential(c * (n_error - I)).

```

```
max pr({x, n_error, c}) for {n_error, c}.
```

Listing 8.23: Jelinski-Moranda model with conjugate priors

```
model go as 'Goel–Okumoto model'.

double n_error as 'initial number of errors'.
        where n =< n_error.
        where n_error =< loc.

double c as 'fault detection rate'.
        where 0 < c.

double p as 'error repair rate'.
        where 0 =< p.
        where p =< 1.

% Observation and distribution
const nat n as 'number of observations'.
        where 0 < n.

const nat loc as 'lines of code'.
        where n < loc.

data double x(0..n-1) as 'interfailure times (elapsed time between
    errors)'.
        where 0 < x(I).

x(I) ~ exponential(c * (n_error - p * I)).

max pr(x|{n_error, c, p}) for {n_error, c, p}.
```

Listing 8.24: Goel-Okumoto model

Name	Lines of spec	Lines of C++	Listing
normal_known_variance	18	299	8.2
normal	14	199	8.1
normal_priors	22	207	8.3
biased_measurements	19	162	8.4
log_normal	15	155	8.5
lighthouse	23	502	8.6
biased_coin	11	91	8.7
biased_coins	14	132	8.8
biased_coins_prior	20	228	8.9
mog	27	587	8.10
mult_cluster	29	676	8.11
mgp_mu	39	635	8.12
mix_beta_gauss	38	999	8.14
mpca	46	589	8.15
walk	17	168	8.16
hinckley	25	353	8.17
climb_transition	38	888	8.18
climb_transition_prior	49	1023	8.19
kalman	30	285	8.20
dgtc	116	735	8.21
jm	21	1040	8.22
ms	35	1109	8.23
go	26	886	8.24

Table 8.2: AutoBayes specifications and size of generated code

Appendix A. Command Line Options

A.1 AutoBayes Command Line Flags

The AutoBayes system is controlled by a (large) number of command-line options. A list of these options can be obtained by calling

```
autobayes -help
```

The following list gives an alphabetical overview of all available command line flags[1]

`[-O number]` set optimization level to N (default=1)

`[-anngen]` generate annotations and VCs from C or intermediate code file (.lang.dump) with -infer SP

`[-c]` parse C code (with -anngen/vcgen)

`[-certify {array|defuse|init|inuse|norm|symm|wl}]` generate policy-specific annotations for certification

`[-check]` check intermediate code for wellformedness

`[-codegen]` generate code from intermediate code file

`[-compile]` compile and link synthesized code

`[-debug number]` set debug level for code instrumentation

`[-designdoc]` generate design document

`[-designdoc filename]` generate named design document

`[-dir filename]` working directory for all output files

`[-dot]` write output for dot to <specname>.dot

`[-dot filename]` write output for dot to <filename>

`[-dump {all}]` dump intermediate code and proof obligations at all stages to files <specname>.<stage>.dump

[1]This list is autogenerated by AutoBayes with `autobayes -tex -help`.

`[-dump {synt|iopt|inst|lang|prop|ann|lopt}]` dump intermediate code at stage to file <specname>.<stage>.dump

`[-dump {synt|iopt|inst|lang|prop|ann|lopt} filename]` dump intermediate code at stage to file <filename>

`[-dump {tptp}]` dump final proof obligations to files <taskname>.dump

`[-dump {vc|nvc|lvc|svc}]` dump proof obligations at stage to file <specname>.<stage>.dump

`[-dump {vc|nvc|lvc|svc} filename]` dump proof obligations at stage to file <specname>.<stage>.dump

`[-fastest]` report which is the fastest synthesized program (see -maxprog)

`[-geninit]` DEVELOP: enable generic variable initialization

`[-genopt]` DEVELOP: enable generic optimize schema

`[-help]` display usage and list of options

`[-help atom]` display usage of option

`[-help {pragmas}]` display list of available pragmas

`[-html]` write synthesized code as html to file <specname>.html

`[-html filename]` write synthesized code as html to file <filename>

`[-html {synt|iopt|inst|lang|prop|ann|lopt} filename]` write intermediate code at stage as html to file <filename>

`[-html {synt|iopt|inst|lang|prop|ann|lopt|all}]` write intermediate code at stage as html to file <specname>.<stage>.html

`[-html_in filename]` retrieve html from the html file <filename>

`[-infer {array|init|norm|symm|val|frame}]` infer policy-specific annotations for certification

`[-instrument]` instrument code with conv-vector

`[-interactive]` switch into interactive mode

`[-js filename]` write synthesized code as JavaScript to file <filename>

`[-js {synt|iopt|inst|lang|prop|ann|lopt} filename]` write intermediate code at stage as JavaScript to file <filename>

`[-js {synt|iopt|inst|lang|prop|ann|lopt|all}]` write intermediate code at stage as JavaScript to file <specname>.<stage>.html

`[-lib {gsl|gslran|unuran}]` enable library

`[-lib {gsl|gslran|unuran} filename]` enable library (with include-path)

`[-list {all}]` list intermediate code and proof obligations at all stages to files <specname>.<stage>.list

`[-list {all} -]` list intermediate code and proof obligations at all stages to stdout

`[-list {synt|iopt|inst|lang|prop|ann|lopt}]` list intermediate code at stage to file <specname>.<stage>.txt

`[-list {synt|iopt|inst|lang|prop|ann|lopt} filename]` list intermediate code at stage to file <filename>

`[-list {vc|nvc|lvc|svc}]` list proof obligations at stage to file <specname>.<stage>.txt

`[-list {vc|nvc|lvc|svc} filename]` list proof obligations at stage to file <filename>

`[-log]` write information to log file <specname>.log

`[-log filename]` write information to log file <filename>

`[-matlab {ann}]` (stage 2: ann) write intermediate code as Matlab-readable JavaScript/HTML to file <specname>_<policy>.lang.html, with main file <specname>.certification.html

`[-matlab {lang}]` (stage 1: lang) write synthesized code as Matlab-readable JavaScript/HTML to file <specname>.lang.html, where main file <specname>.certification.html

`[-matlab {proofs}]` (stage 4: proofs) Generates Stage(1-3). Executes run_all prover on VCs and return result in Matlab-readable JavaScript/HTML, where main file is <specname>_<policy>.certification.html

`[-matlab {tasks}]` (stage 3: tasks) write intermediate code as Matlab-readable JavaScript/HTML, where main file <specname>_<policy>.certification.html

`[-maxprog number]` synthesize up to N program versions (default=1)

`[-monitorapproximations]` synthesize code to check that approximation error bounds are respected

`[-monte_carlo]` synthesize monte-carlo data for Kalman filters [Autofilter only]

`[-nocode]` synthesize intermediate code only

`[-nooptimize]` set optimization level to 0 (no optimization)

`[-php filename]` write synthesized code as php to file <filename>

`[-php {synt|iopt|inst|lang|prop|ann|lopt} filename]` write intermediate code at stage as php to file <filename>

`[-php {synt|iopt|inst|lang|prop|ann|lopt|all}]` write intermediate code at stage as php to file <specname>.<stage>.php

`[-pragma atom=atom]` set pragma to value

`[-prover {tptp|esetheo|ivy}]` use specified prover for certification [inactive]

`[-quiet]` reduce log/trace information

`[-sample]` synthesize code for data sampling

`[-silent]` suppress all log/trace information

`[-target {c_standalone|matlab|modula2|octave|spark}]` target language and runtime environment

`[-tex]` write synthesized code as latex to file <specname>.tex

`[-tex filename]` write synthesized code as latex to file <filename>

`[-tex {synt|iopt|inst|lang|prop|ann|lopt} filename]` write intermediate code at stage as latex to file <filename>

`[-tex {synt|iopt|inst|lang|prop|ann|lopt|all}]` write intermediate code at stage as latex to file <specname>.<stage>.tex

`[-timelimit number]` set overall timelimit

`[-timelimit number number number]` set timelimits for simplifier, solver, optimizer

`[-user filename]` user information [only for web autobayes]

`[-vcgen]` generate VCs from C or annotated intermediate code file (.ann.dump / .prop.dump) with -infer SP or -certify SP, resp.

A.2 AutoBayes Pragmas

AutoBayes pragmas are low-level flags and commands to control specific actions in the AutoBayes system. Their main purpose is to help the developer and andvanced user to guide the AutoBayes system in a specific way. A complete list of AutoBayes pragmas can be obtained by

$ `autobayes -help pragmas`

In general, a pragma has the form

`-pragma <NAME>=<VALUE>`

No spaces are allowed between the name of the pragma, the equality sign "=", and the value.

In the following we list all AutoBayes pragmas[2]. For each pragma, its name and type is given. Default values and, where applicable, a list of possible values are shown. AutoBayes supports the following types of pragmas:

boolean are boolean flags, which can take the values `true` or `false`

integer can take arbitrary integer values

atom can take a Prolog atomic value, i.e., a name like `none`, or a string in single quotes, e.g., `'this model'`. If a list of possible values is given, only arguments matching with an element of this list can be used.

callable requires the name of a predicate. This feature enables the developer to call specific predicates in conjunction with this predicate. The arity of such a predicate depends on it actual calling environment. No checks whatsoever are performed.

`assert_display_location` **(boolean)** Display source code locations in trace messages

Default: `-pragma assert_display_location=true`

`certify_browser` **(atomic)** set to open a browser (or browser tab) after execution of code

Default: `-pragma certify_browser=none`

Possible values :

`none` no browser

`firefox` open firefox browser

`mozilla` open mozilla browser

`safari` open safair browser

`netscape` open netscape browser

[2] This list is autogenerated by the command `autobayes -tex -help pragmas`.

`certify_explicit_symm` **(boolean)** use symm predicate in annotations

Default: `-pragma certify_explicit_symm=false`

`certify_external_init` **(boolean)** Treat external declarations as being initialized

Default: `-pragma certify_external_init=false`

`certify_filename_policy` **(boolean)** use safety policy in filename

Default: `-pragma certify_filename_policy=true`

`certify_generate_proof_tasks` **(boolean)** Generate certification proof tasks: gets set to true if -certify or -infer called

Default: `-pragma certify_generate_proof_tasks=false`

`certify_generate_safety_doc` **(boolean)** generate rendered safety document from inference and VC information

Default: `-pragma certify_generate_safety_doc=false`

`certify_generate_true_prooftask` **(boolean)** Generate at least one proof task

Default: `-pragma certify_generate_true_prooftask=true`

`certify_globals_strict` **(boolean)** Globals must be initialized according to decl lists in procs and funcs

Default: `-pragma certify_globals_strict=false`

`certify_globals_visible` **(boolean)** Globals are visible throughout

Default: `-pragma certify_globals_visible=true`

`certify_hotvar` **(atomic)** Active hotvar for annotation inference

Default: `-pragma certify_hotvar=$all`

Possible values :

`$all` Infer annotations for all hotvars

`_` Infer annotations for specified hotvar only

`certify_itar_warning` **(boolean)** insert ITAR warning at top of safety report

Default: `-pragma certify_itar_warning=false`

`certify_label_stage` **(atomic)** The stage at which line numbers are added during annotation inference

Default: `-pragma certify_label_stage=lang`

Possible values :

`source` Assume to already exist in parsed source

`lang` Add before inference

`ann` Add after inference

`certify_limit_defs` **(boolean)** Limit the def patterns to be generator specific

Default: `-pragma certify_limit_defs=true`

`certify_list_defs` **(boolean)** List which defs have been successfully annotated

Default: `-pragma certify_list_defs=true`

`certify_only_defs` **(boolean)** Only annotate the defs

Default: `-pragma certify_only_defs=false`

`certify_ordered_anns` **(boolean)** process pre- and post-conditions in order

Default: `-pragma certify_ordered_anns=false`

`certify_render_lterm` **(boolean)** display lterm with rendered VCs

Default: `-pragma certify_render_lterm=false`

`certify_semantic_labels` **(boolean)** add semantic markup to VCs and display explanations in certification browser

Default: `-pragma certify_semantic_labels=false`

`certify_semantic_labels_order` **(callable)** sort order for display of semantic markup in VCs

Default: `-pragma certify_semantic_labels_order=render_sort_labels_by_line`

Possible values :

`render_sort_labels_by_line`

`certify_stream_vcs` **(boolean)** Output VCs as they are generated

Default: `-pragma certify_stream_vcs=false`

`certify_transparent_anns` **(boolean)** use transparent annotation rules in VCG

Default: `-pragma certify_transparent_anns=false` `certify_transparent_inf` **(boolean)** only annotate opaque barriers during inference

Default: `-pragma certify_transparent_inf=false`

`certify_use` **(atomic)** Active use number for annotation inference

Default: `-pragma certify_use=$all`

Possible values :

`$all` Infer annotations for all uses of a given hotvar

`_` Infer annotations for a specified use of a given hotvar

`certify_use_postconditions` **(boolean)** use postconditions in VCG

Default: `-pragma certify_use_postconditions=true`

`certify_vc_label` **(callable)** Pre-simplifier for proof tasks (labeling and splitting)

Default: `-pragma certify_vc_label=vc_label`

Possible values :

`vc_label` strongest simplification (default)

`vc_label_structure_prop`

`vc_label_structure`

`vc_identity` no simplification

`certify_vc_normalize` **(callable)** Normalizer for proof tasks (integrated into VCG)

Default: `-pragma certify_vc_normalize=vc_normalize`

Possible values :

`vc_normalize` default normalization

`vc_normalize_auxiliary`

`vc_normalize_flist`

`vc_normalize_subst`

`vc_identity` no normalization

`certify_vc_simplify` **(callable)** Simplifier for proof tasks (after conversion into FOL)

Default: `-pragma certify_vc_simplify=vc_simplify`

Possible values :

`vc_simplify` strongest simplification (default)

`vc_identity` no simplification

`cg_comment_style` **(atomic)** select comment style for C/C++ code generator

Default: `-pragma cg_comment_style=cpp`

Possible values :

`kr` use traditional (KR) style comments

`cpp` use C++ style comments //

`cluster_pref` **(atomic)** select algorithm schemas for hidden-variable (clustering) problems

Default: `-pragma cluster_pref=em`

Possible values :

`em` prefer EM algorithm

`no_pref` no preference

`k_means` use k-means algorithm

`codegen_ignore_inconsistent_term` **(boolean)** [DEBUG] ignore inconsistent-term conditional expressions in codegen

Default: `-pragma codegen_ignore_inconsistent_term=false`

`em` **(atomic)** preference for initialization algorithm for EM

Default: `-pragma em=no_pref`

Possible values :

`no_pref` no preference

`center` center initialization

`sharp_class` class-based initialization (sharp)

`fuzzy_class` class-based initialization (fuzzy)

`em_log_likelihood_convergence` **(boolean)** converge on log-likelihood-function

Default: `-pragma em_log_likelihood_convergence=false` `em_q_output` **(boolean)** Output the Q matrix of the EM algorithm

Default: `-pragma em_q_output=false`

`em_q_update_simple` **(boolean)** force the q-update to just contain the density function

Default: `-pragma em_q_update_simple=false`

`ignore_division_by_zero` **(boolean)** DEBUG: Do not check for X=0 in X**(-1) expressions

Default: `-pragma ignore_division_by_zero=false`

`ignore_zero_base` **(boolean)** DEBUG: Do not check for zero-base in X**Y expressions

Default: `-pragma ignore_zero_base=false`

`il_extended` **(boolean)** use extended intermediate language. Set with -c

Default: `-pragma il_extended=false`

`infile_cpp_prefix` **(atomic)** Prefix for intermediate input file after cpp(1) processing

Default: `-pragma infile_cpp_prefix=cpp_`

`instrument_convergence_save_ub` **(integer)** default size of instrumentation vector for convergence loops

Default: `-pragma instrument_convergence_save_ub=999`

`lopt` **(boolean)** Turn on/off optimization of the lang code

Default: `-pragma lopt=false`

`optimize_cse` **(boolean)** enable common subexpression elimination

Default: `-pragma optimize_cse=true`

`optimize_expression_inlining` **(boolean)** enable inlining (instead function calls) of goal expressions by schemas

Default: `-pragma optimize_expression_inlining=true`

`optimize_max_unrolling_depth` **(int)** maximal depth of for-loops w/ constant bound to be unrolled

Default: `-pragma optimize_max_unrolling_depth=3` `optimize_memoization` **(boolean)** enable subexpression-memoization

Default: `-pragma optimize_memoization=true`

`optimize_substitute_constants` **(boolean)** allow values of constants to be substituted into loop bounds

Default: `-pragma optimize_substitute_constants=true`

`pp_html_fixed_font_family` **(atomic)** Font family for fixed fonts in html-output

Default: `-pragma pp_html_fixed_font_family=courier new`

Possible values :

`courier new` default font

other fonts possible

`pp_html_fixed_font_size` **(int)** Font size for fixed fonts in html-output

Default: `-pragma pp_html_fixed_font_size=14`

`pp_html_font_color_active` **(atomic)** Font color for active in html-output

Default: `-pragma pp_html_font_color_active=#6699FF`

Possible values :

`#6699FF` default color

other colors possible

`pp_html_font_color_annotation` **(atomic)** Font color for annotation in html-output

Default: `-pragma pp_html_font_color_annotation=purple`

Possible values :

`purple` default color

other colors possible

`pp_html_font_color_comment` **(atomic)** Font color for comment in html-output

Default: `-pragma pp_html_font_color_comment=green`

Possible values :

`green` default color

other colors possible

`pp_html_font_color_highlight` **(atomic)** Font color for highlight in html-output

Default: `-pragma pp_html_font_color_highlight=red`

Possible values :

`red` default color

other colors possible

`pp_html_font_color_hover` **(atomic)** Font color for hover in html-output

Default: `-pragma pp_html_font_color_hover=#6699FF`

Possible values :

`#6699FF` default color

other colors possible

`pp_html_font_color_label` **(atomic)** Font color for label in html-output

Default: `-pragma pp_html_font_color_label=blue`

Possible values :

`blue` default color

other colors possible

`pp_html_font_color_label_ref` **(atomic)** Font color for label in html-output

Default: `-pragma pp_html_font_color_label_ref=#FF9966`

Possible values :

`#FF9966` default color

other colors possible

`pp_html_font_color_link` **(atomic)** Font color for link in html-output

Default: `-pragma pp_html_font_color_link=#6699FF`

Possible values :

`#6699FF` default color

other colors possible

`pp_html_font_color_schema` **(atomic)** Set font colors for html-output

Default: `-pragma pp_html_font_color_schema=default`

Possible values :

`default`

`bw`

`pp_html_font_color_visited` **(atomic)** Font color for visited in html-output

Default: `-pragma pp_html_font_color_visited=#6699FF`

Possible values :

`#6699FF` default color

other colors possible

`pragmas_detailed_help` **(boolean)** Print detailed information on Pragmas in -help pragmas

Default: `-pragma pragmas_detailed_help=true`

`prolog_style` **(boolean)** Capitalized names are variables

Default: `-pragma prolog_style=true`

`propagate_annotations` **(atomic)** propagate explicit annotations during certification

Default: `-pragma propagate_annotations=true`

Possible values :

`true` Propagate annotations after lang stage

`false` Do no propagation

`infer_ann_pre` Propagate before inference

`infer_ann_post` Propagate after inference

`propagate_index_bounds` **(boolean)** propagate index bounds during certification

Default: `-pragma propagate_index_bounds=true`

`rwr_cache_max` **(integer)** size of rewrite cache

Default: `-pragma rwr_cache_max=2048`

`schema_control_arbitrary_init_values` **(boolean)** enable initialization of goal variables w/ arbitrary start/step values

Default: `-pragma schema_control_arbitrary_init_values=false` `schema_control_init_values` **(atomic)** initialization of goal variables

Default: `-pragma schema_control_init_values=automatic`

Possible values :

`automatic` calculate best values

`arbitrary` use arbitrary values

`user` user provides values (additional input parameters

`schema_control_solve_partial` **(boolean)** enable partial symbolic solutions

Default: `-pragma schema_control_solve_partial=true`

`schema_control_use_generic_optimize` **(boolean)** enable intermediate code generation w/ generic optimize(...)-statements

Default: `-pragma schema_control_use_generic_optimize=false`

`synth_serialize_maxvars` **(integer)** maximal number of solved variables eliminated by serialize

Default: `-pragma synth_serialize_maxvars=0`

`system_os` **(atomic)** generate html for target os

Default: `-pragma system_os=linux`

Possible values :

`linux`

`windows`

`trace_browser_files` **(boolean)** Trace I/O of certification browser files

Default: `-pragma trace_browser_files=false`

`trace_display_solver_obligations` **(boolean)** display proof obligations from the solver

Default: `-pragma trace_display_solver_obligations=true`

`trace_vc_files` **(boolean)** Trace I/O of VC files

Default: `-pragma trace_vc_files=true`

Appendix B. Acknowledgements

Acknowledgements

Program synthesis is the automated construction of correct and efficient executable code from declarative specifications. The RSE (Robust Software Engineering) area at NASA Ames Research Center is developing applications of this technology in the domains of data analysis and state estimation. The RSE area is also interested in correctness issues, specifically regarding the formal certification of synthesized code.

For more information, please refer to RSE's web site `URL: http://ti.arc.nasa.gov/` (click on the link for the "Robust Software Engineering" research area.)

The main developers of AutoBayes are (in alphabetical order) Wray Buntine (NICTA, Australia), Bernd Fischer (University of Southampton, UK), and Johann Schumann (USRA/RIACS). The following researchers (in alphabetical order) substantially contributed to the AutoBayes system: Ewen Denney (USRA/RIACS), Thomas Pressburger (RECOM Inc., QSS Group, Inc., NASA), Phil Oh (QSS Group, Inc.), Grigore Rosu (USRA/RIACS), and Jon Whittle (RECOM Inc., QSS Group, Inc.), with the help of summer students and interns Jutta Eusterbrock, Alexander Gray, Kate Mullen, Rachel Smith, Amber Telfer, Steven Trac, and Mike Whalen. Its development was supported by Michael Lowry (Intelligent Systems Division Chief Scientist) and Joseph Coughlan (RSE area lead).

The authors of this manual would like to acknowledge Kanishka Bhaduri and John Stutz of NASA Ames Research Center who reviewed this manual and provided helpful comments.

The development of AutoBayes was supported by NASA Universities Grant 05106, and the following NASA programs: the Intelligent Data Understanding (IDU) Program, the Intelligent Systems (IS) Program, the Computing, Information, and Communication Technology (CICT) Program, and the Exploration Technology Development Program (ETDP).

Bibliography

[BaJGS05] A. Banerjee, I. Dhilon ans J. Ghosh, and S. Sra. Clustering on the Unit Hypersphere using von Mises-Fisher Distributions. *The Journal of Machine Learning Research*, 6:1345–1382, 2005.

[BFG03] W. Buntine, B. Fischer, and A. Gray. Automatic Derivation of the Multinomial PCA Algorithm. Technical report, 2003. Available at `http://ti.arc.nasa.gov/people/fischer/papers/ais2003-print.ps`.

[BFH$^+$99] W. Buntine, B. Fischer, K. Havelund, M. Lowry, T. Pressburger, S. Roach, P. Robinson, and J. Van Baalen. Transformation Systems at NASA Ames. In Marcelo Sant'Anna, editor, *Proc. ICSE-21 Intl. Workshop Software Transformation Systems*, pages 8–13, Los Angeles, CA, May 1999.

[BFP99] W. Buntine, B. Fischer, and T. Pressburger. Towards Automated Synthesis of Data Mining Programs. In Surajit Chaudhuri and David Madigan, editors, *Proc. 5th Intl. Conf. Knowledge Discovery and Data Mining*, pages 372–376, San Diego, CA, August 15–18 1999. ACM Press.

[BFS00] W. Buntine, B. Fischer, and J. Schumann, editors. *NIPS*2000 Workshop on Software Support for Bayesian Analysis Systems*, Breckenridge, December 2000.

[BH97] R. Brown and P. Hwang. *Introduction to Random Signals and Applied Kalman Filtering.* John Wiley & Sons, 3rd edition, 1997.

[Bis95] Ch. M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon-Press, Oxford, 1995.

[BS94] J. M. Bernardo and A. F. M. Smith. *Bayesian Theory.* J. Wiley & Sons, Chichester, UK, 1994.

[EH81] B. S. Everitt and D. J. Hand. *Finite Mixture Distributions.* Chapman & Hall, 1981.

[FHKS03] B. Fischer, A. Hajian, K. Knuth, and J. Schumann. Automatic Derivation of Statistical Data Analysis Algorithms: Planetary Nebulae and Beyond. In Gary Erickson and Yuxiang Zhai, editors, *Proc. 23rd Intl.*

Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering, pages 276–291. American Institute of Physics, 2003.

[FPRS01] B. Fischer, T. Pressburger, G. Roşu, and J. Schumann. The AutoBayes Program Synthesis System—System Description. In Steve Linton and Roberto Sebastiani, editors, *Proc. 9th Symp. Integration of Symbolic Computation and Mechanized Reasoning*, pages 118–125, Siena, Italy, July 2001.

[FS03a] B. Fischer and J. Schumann. Applying AutoBayes to the Analysis of Planetary Nebulae Images. In John Grundy and John Penix, editors, *Proc. 18th Intl. Conf. Automated Software Engineering*, pages 337–342, Montreal, Canada, October 6–10 2003. IEEE Comp. Soc. Press.

[FS03b] B. Fischer and J. Schumann. AutoBayes: A System for Generating Data Analysis Programs from Statistical Models. *J. Functional Programming*, 13(3):483–508, May 2003.

[FSP00] B. Fischer, J. Schumann, and T. Pressburger. Generating Data Analysis Programs from Statistical Models (position paper). In Walid Taha, editor, *Proc. Intl. Workshop Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lect. Notes Comp. Sci.*, pages 212–229, Montreal, Canada, September 2000. Springer.

[FW99] B. Fischer and J. Whittle. An Integration of Deductive Retrieval into Deductive Synthesis. In Robert J. Hall and Enn Tyugu, editors, *Proc. 14th Intl. Conf. Automated Software Engineering*, pages 52–61, Cocoa Beach, Florida, October 1999. IEEE Comp. Soc. Press.

[GA01] M. S. Grewal and A. P. Andrews. *Kalman Filtering: Theory and Practice Using MATLAB.* Wiley Interscience, 2001. 2nd edition.

[GBSMB08] K. Gundy-Burlet, J. Schumann, T. Menzies, and T. Barrett. Parametric Analysis of Antares Re-entry Guidance Algorithms using Advanced Test Generation and Data Analysis. In *Proc. iSAIRAS 2008 (9th International Symposium on Artifical Intelligence, Robotics and Automation in Space)*, 2008.

[GCSR95] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis.* Chapman & Hall, 1995.

[GFSB03] A. G. Gray, B. Fischer, J. Schumann, and W. Buntine. Automatic Derivation of Statistical Algorithms: The EM Family and Beyond. In

Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 689–696. MIT Press, 2003.

[GMW81] P. Gill, W. Murray, and M. Wright. *Practical Optimization.* Academic Press, 1981.

[GO78] A. L. Goel and K. Okumoto. An Analysis of Recurrent Software Failures on a Real-time Control System. In *ACM Annual Technical Conference*, pages 496–500, 1978.

[Gul88] S. F. Gull. Bayesian Inductive Inference and Maximum Entropy. In G. J. Erickson and C. R. Smith, editors, *Maximum entropy and Bayesian methods in science and engineering*, volume Vol. 1. Kluwer, Dordrecht, 1988.

[JM72] Z. Jelinski and P. B. Moranda. Software Reliability Research. In W. Freiberger, editor, *Statistical Computer Performance Evaluation*, pages 465–484. Academic Press, New York, 1972.

[KH02] K. H. Knuth and A. R. Hajian. Hierarchies of models: Toward Understanding of Planetary Nebulae. In C. Willimans, editor, *Proc. Bayesian Inference and Maximum Entropy Methods in Science and Engineering*, pages 92–103. American Institute of Physics, 2002.

[MAY79] P. S. Maybeck. *Stochastic Models, Estimation, and Control.* Vol. 1, Academic Press, New York, New York, 1979.

[MK97] G. McLachlan and T. Krishnan. *The EM Algorithm and Extensions.* Wiley Series in Probability and Statistics. John Wiley & Sons, New York, 1997.

[MS83] R. J. Meinhold and N. D. Singpurwalla. Bayesian Analysis of a Commonly Used Model for Describing Software Failures. *The Statistician*, 32:168–173, 1983.

[OF96] J. J. K. O'Ruandaidh and W. J. Fitzgerald. *Numerical Bayesian Methods Applied to Signal Processing.* Springer, Berlin, 1996.

[PFTV92] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge Univ. Press, Cambridge, UK, 2nd. edition, 1992.

[Siv96] A. S. Sivia. *Data Analysis - A Bayesian Tutorial.* Oxford University Press, Oxford, UK, 1996.

[SSF05] A. Srivastava, J. Schumann, and B. Fischer. An Ensemble Approach to Building Mercer Kernels with Prior Information. In *IEEE SMC Conference*. IEEE, 2005.

[TOMS04] J. Whittle, J. Schumann. Automating the Implementation of Kalman Filter Algorithms. In *TOMS: ACM Transactions on Mathematical Software*, Vol. 29, issue 5, pages 434-453, December 2004.

[Vet07] K. Vetter. The Trick User's Guide, Trick 2007.5 release, July 2007.

[WSF02] M. Whalen, J. Schumann, and B. Fischer. Autobayes/CC — Combining Program Synthesis with Automatic Code Certification (system description). In Andrei Voronkov, editor, *Proc. 18th Intl. Conf. Automated Deduction*, volume 2392 of *Lect. Notes Artificial Intelligence*, pages 290–294, Copenhagen, Denmark, July 2002. Springer.

Index