Designing a computational system for multi-paradigm modeling

Ben Klemens
U.S. Census Bureau
ben.klemens@census.gov*

March 11, 2010

Abstract

This paper discusses the design of a framework for scientific computing intended to accommodate a wide range of models, from ordinary least squares to agent-based microsimulations, thus allowing easier comparison and discussion across different schools of modeling. The paper begins with theoretical issues in defining a model, and concludes with running code applying some common operations to disparate models.

Keywords: statistic, simulation, modeling, computational analysis, methodology.

Introduction

Members of different schools of modeling literally speak different languages: authors of agent-based models (ABMs) will typically write in a general programming language like Java using one of many ABM-specific libraries, while discrete-event simulation authors will use other libraries (such as SimPy); generalized linear model (GLM) users may use R's GLM-specific syntax for models; Bayesians may prefer Bugs; demographic analysts often work in SAS or SPSS; and so on.

*The author would like to thank Amber Baum, Derrick Higgins, Guy Klemens, Lynne Plettenberg, Rolando Rodríguez, and Aref Dajani for comments and support. This report is released to inform interested parties of ongoing research and to encourage discussion. The views expressed are those of the author and not necessarily those of the U.S. Census Bureau.

This paper discusses some of the challenges, both theoretical and practical, to providing a single computational object that can be used for describing the full range of models from traditional statistical modeling to simulation modeling. Such a framework would facilitate comparing models across paradigms, mixing paradigms, or using tools written by advocates of one paradigm to study models from another. For example, a researcher may wish to use a Probit regression to elucidate her microsimulation results, while another may find that a Loess smoothing step—a common technique from generalized linear modeling—improves his simulation results.

The simplest scenario is comparison across genres: how do the results of a demographic analysis match up with the results from a GLM estimation? Researchers are more likely to ask such questions when they do not need to spend time forcing the formats of inputs and outputs from the models to agree.

A hierarchical model consists of a collection of smaller independent models and a parent model that aggregates statistics from each smaller model; there is no need for parent and child to be from the same modeling paradigm. Consider a model of smoking in a school: the child models could be several network models of individual classrooms, and the parent a school-wide fit of the best-fitting Normal distribution for the count of smokers in each classroom. Given consistent semantics across models, one could swap out the child network model for an OLS regression, or replace the parent distribution with a kernel density estimate. One could do similar paradigm merging with Bayesian updating, where two input models (perhaps from different genres) are combined to produce an output model.

Within a single paradigm, a structured model object also has benefits. The most notable is the provision of defaults for structure elements not explicitly defined. Begin by writing a probability function and nothing more. Given the probability function, estimate parameters via maximum likelihood; then, given parameter estimates, estimate their variance via bootstrapping; then, given variances, test hypotheses about the parameters. As time allows, replace computationally-intensive default routines with model-specific methods for the estimation of parameters or calculation of variances.

This paper begins at the broad level of theory, and proceeds to increasing levels of technical detail. After a brief discussion of some prior literature, it will ask how one could best use mathematical definitions of statistical models for computational

purposes. Given a definition in theory, the discussion turns to some of the many details and issues that arise when the model object is implemented in code. The paper will then present applications of the model structure, including its use in procedures like Bayesian updating or in producing new models. Finally, because this paper strives to address real-world issues of implementation, an appendix presents a working example of the principles laid out in the previous sections, comparing models from different genres.

Prior literature

Few statistics packages and libraries have attempted to unify statistical models under a single framework; this section will contrast two that do: Zelig [Imai et al., 2008] and Lisp-Stat [Tierney, 1990]. Zelig embodies a specific type of workflow by imposing a specific interface on the model, while at the other end of the spectrum, Lisp-Stat goes out of its way to allow any model by imposing as little framework as possible.

Object-oriented systems like C++ or Java require a class or object declaration, which defines the form to which all objects in the class must conform. Zelig follows this lead, defining a class format into which it places a large set of contributed models. The class structure is intended for models that explain a single outcome variable using a set of input variables—basically, the traditional GLM framework—and therefore limits itself to functions useful for that style of analysis, including key routines to estimate model parameters, fix explanatory variables, present attractive outputs, and so on. The specific model template does much to simplify work in this given workflow, but has little utility for problems that stray from that form.

Lisp-Stat follows the mold of object-oriented systems based on an inheritance structure, where specific models inherit from a relatively abstract model. Specific types of GLM, for example, would inherit from a GLM prototype (which Tierney [1991] implements). A model may respond to any from a long list of messages, including simple requests like `:coef-estimates` to get the estimated coefficients, on up to relatively specialized and computation-intensive commands like `:cooks-distance`.

Tierney explicitly rejects class formats as too restrictive [p 206]. Instead, a model is simply a Lisp-style list of data or procedures, and authors may add as many new

items to the list as are useful, without class-defined constraints. Thus, the system's key strength is that it is flexible, and new methods may easily be added to any model. [*Method* is object-oriented jargon for a procedure that is held inside of an object or structure.]

For our purposes, the system's key failing is that it is flexible, and new methods may easily be added to any model. Each class of model will have its own set of functions that make sense for its genre of modeling, so there is no guarantee that the internals of any two models will match sufficiently well that one could be swapped for another in a given procedure.

The problem, then, is to define a class structure that is broad enough to usefully describe any model, but sufficiently limited that we can be guaranteed that every model implements every method included in the definition.

Defining the model object

McCullagh [2002] gives a long list of authors who either explicitly or implicitly use a definition of parameterized statistical model akin to the following, given a sample space $\mathcal{S}$ and the space of all probability distributions $\mathcal{P}$:

Definition 1 *A parameterized statistical model is a parameter set $\boldsymbol{\theta}$ together with a function $P : \boldsymbol{\theta} \rightarrow \mathcal{P}(\mathcal{S})$, which assigns to each parameter point $\boldsymbol{\theta} \in \Theta$ a probability distribution $P_{\boldsymbol{\theta}}$ on $\mathcal{S}$.*

See also Hill [1990, p 119], who gives another definition along McCullagh's mold.

For example, a Normal model in this context is a function in three variables: $P(s, \mu, \sigma)$, where s is a scalar data point, μ the mean, and σ the standard deviation. Selecting a single element in the parameter space, such as ($\mu = 0$, $\sigma = 1$), fixes a single probability distribution, a function of one variable, $P(s)$.

Roughly, the definition describes a model using three characters: a data set $s \in \mathcal{S}$, parameters $\boldsymbol{\theta}$ from the parameter space Θ, and probabilities $P(\cdot)$. It defines the model as the mapping $\boldsymbol{\theta} \rightarrow P(\cdot)$. However, there are reasons to map other elements of the definition to other elements—say, data to parameters. With that in mind, this definition of a model adds useful generality to the prior definition:

Definition 2 *A model intermediates between data, parameters, and likelihoods.*

Presuming that we want to begin with input of fixed data or fixed parameters, the term *intermediates* unfolds to a number of different mappings: $s \rightarrow \boldsymbol{\theta}$, $(\boldsymbol{\theta}, s) \rightarrow P$, or $\boldsymbol{\theta} \rightarrow s$. These three directions are are depicted in Figure 1.

In code, the likelihood function has a signature of the form `likelihood(data, parameters)`, returning a scalar real number. Following the mapping in Definition 1, fixing `parameters` to a constant value reduces this function to a probability distribution $P(\mathcal{S})$.

However, there is more to be done. Given data, one could generate a distribution of parameters: fix s and thus reduce $P(\mathcal{S}, \Theta)$ to $P(\Theta)$. Some authors, most notably Fisher [1934, p 287], are adamant that this is a *likelihood function*, while $P(\mathcal{S})$ is a *probability function*, and the two should remain separate in interpretation. The computer is entirely indifferent to the distinction, and any differences in interpretation are up to the user.

The estimation problem goes from input data to parameters, $s \rightarrow \boldsymbol{\theta}$. A maximum likelihood estimation (MLE) would execute the mapping $s \rightarrow P(\Theta)$, then find the most likely value of $\boldsymbol{\theta}$. But there are often other methods available. For the Normal model, a full maximum-likelihood search would be silly: the most likely μ is the data mean, and the most likely σ the square root of the sample variance.

Given parameters, one can generate new data ($\boldsymbol{\theta} \rightarrow s$). By the traditional model, random data generation involves fixing the parameters, then making a large volume of draws from the distribution $P(\mathcal{S})$. Computationally, this method is typically the worst case. Rather, there is a thick book of tricks to make efficient random draws without explicitly evaluating likelihoods (i.e. Devroye [1986]). The expected value is another salient point in the space $\mathcal{S}$ that could be derived from $\boldsymbol{\theta}$ via $P(\cdot)$; again, one can find it via millions of draws from $P(\mathcal{S})$ or, for many models, a simple closed-form computation.

Definition 2 expands Definition 1 by including all mappings among data, parameters, and likelihoods in the model definition, thus making room for specification in code of the simple estimation routine for the Normal model or the many tricks for drawing random numbers. Perhaps more importantly, the definition indicates what is not in the model: Bayesian updating and Cook's distance, for example, will be implemented as functions outside the model object that will take models as inputs.

Simulations versus distributions

The discussion above took the core of the model as a probability distribution, but simulation-type models, centered around explicitly describing the interaction among several elements of a system, often do not center around a probability density. Nonetheless, under appropriate assumptions, a probability distribution does materialize.

For a function to be a probability distribution, it must meet two criteria: it is a function over a given space that integrates to one, and is always nonnegative.

Even the requirement that the function integrate to one is not necessary for most purposes: maximum likelihood routines make no use of the requirement, and strategies to make random draws from an arbitrary function typically use only relative values of the function from which draws are made. Of the methods described in the application section below, only Kullback-Leibler divergence and, in some cases, Bayesian updating use this requirement.

Consider a simulation intended to approximate observed data. The distance between the simulated output and the observed value(s) can be easily be calculated using an appropriate metric, and, given distance d, p can be defined as a monotonically decreasing function of distance, such as $1/(1+d)$ or e^{-d}. This short paper is not an appropriate venue for discussing the choice of transformation, but note that the Information Equality indicates that the choice of transformation is frequently not relevant; see, e.g., Pawitan [2001].

Without target data, models with a stochastic component generate probability distributions over output values directly. Let input data be a vector $\mathbf{X}$, and outputs a scalar Y; then each run is a draw from the distribution $P(Y|\mathbf{X})$ implicit to the assumptions and stochastic elements of the simulation. A random number generation scheme implies a probability distribution (and vice versa), so we again have enough for full use of the framework below.

The methods in the section on applications, below, include a number of common statistical methods that I feel are underused outside of the tradition of closed-form statistics. The next section will show that, once an author has written down an inverse-distance function to evaluate a set of model parameters, the rest of a full statistical model can quickly be filled in, and the statistical methods used directly.

Implementation

To this point, the computational model object would include `p` (or `probability`), random `draw`, `estimate`, and `expected_value` (or, as discussed below, `predict`) methods. These functions would be declared as part of the model class specification, and thus would have the same signature for all models. The Bayesian updating routine below will make use of a `name` element, which is naturally also useful for any output routines.

A model intermediates between data, parameters, and likelihoods, so the model object will naturally require a link to the original `data` set used for estimation and a list of `parameters`.

As a side-note, the authors of the R package, Gentleman and Ihaka [2000], advocate lexical scoping to bind the values of the general form, like fixing $\mathcal{N}(x, \mu, \sigma)$ with a given specific set of parameters (like $\mu = 0$, $\sigma = 1$) to produce $P(x)$. When a function is declared, it is stored in a larger list that includes both the function and the environment in which the function was declared. Whatever values μ and σ had when this list was initialized are stored in the environment, and used as the function's fixed values. Thus, R generates an *ad hoc* structure using the environment to store parameters and data for a single method. An explicitly-defined model object can simply hold the parameter values directly.

Default methods

If two models are similar, then the author of the second should be able to use the methods from the first, rather than rewriting them. Inheritance from a parent model is easy: just copy the parent model and thus all of its methods, then replace any individual methods that differ.

This is useful when two models are similar, but should the author have no model to work off of, the system should also provide default methods. One could implement default methods via a model at the base of a formal inheritance tree; in the appendix, they will be implemented via a set of wrapper functions.

In the simple C-like pseudocode to follow, the = symbol indicates assignment, and `model.p` refers to the `p` element held inside the `model` structure.

Definition 1 focused on a probability function, so say that an author writes only the probability function for a new model:

```
new_model.p = carefully_written_probability_fn(., .)
new_model.log_likelihood = <undefined>
new_model.predict = <undefined>
...
```

The author provided nothing more, but every other method can be derived via a computationally-intensive method. To give the simplest example, a `log_likelihood` function would take in a data set and a model, and return the result of a simple if-then:

```
if model.log_likelihood exists
    return model.log_likelihood(data, model)
else
    return log(model.p(data, model))
```

That is, if there is a log likelihood function to be had, then use it; if not, then find the log likelihood via the log of p. In practice, one or the other is typically more precise and easier to calculate—statisticians lean toward log likelihood, while simulations tend to arrive at a simple p.

Other methods will also use the specific model's function if available, and a fill-in default otherwise. For example, an ordinary least squares (OLS) model would have an `estimate` method solving the linear algebra equation for OLS parameters, while elaborate models with only a `p` function would resort to the default of MLE.

Models with no explicit random number generator (RNG) could use Adaptive Rejection Markov Sampling [Gilks et al., 1995]. The score (i.e. the gradient of the log likelihood function) can be calculated using a delta method. Conversely, a model that provides only an RNG can produce a probability distribution either by binning several draws into a probability mass function (a PMF, see below) and possibly smoothing the PMF via kernel smoothing, Loess, or other methods.

Now consider missing data, prediction, and expected values. Begin with a tuple of data of the form $[YX]$, and say that Y is missing, leaving $[\texttt{NaN}\ X]$ (where `NaN` is read as *Not a number*). This is a common setup for maximum likelihood (ML)

imputation: given the model and existing data, find the most likely fill-in for the `NaN` value. This is also the prediction story for GLMs: the full data set $[Y\,X]$ is known when estimating parameters, but Y is missing and to be filled in when predicting. For a distribution, given data $[\texttt{NaN}\;X]$, the best fill-in is typically the conditional expected value, $E(Y|X)$. The unconditional expected value is the special case where no data is available, and all values must be filled in. Thus, a single `predict` function with an ML imputation default can act as a missing data, prediction, and expected value routine. As with log likelihood and maximum likelihood, ML imputation requires only a `p` method to find a computationally-intensive solution, but specific models can add special cases when closed-form solutions exist.

At this point, every model has a small set of functions that has the same form in every model. As long as the model author wrote some methods for the model (probably the log likelihood function), then all of the other methods are guaranteed to work and return good values via the default dispatch functions, although the results may be more computationally intensive than necessary.

Applications

This section shows some uses of the model object, including functions that take models as inputs, and transformations of models to produce new models, such as fixing most of the parameters of a multidimensional model to produce a unidimensional model.

Some applications are immediate. Producing covariances via bootstrapping and jackknifing involves generating a series of subsets of a main data set, re-running the `estimate` method of a model on each subset, and finding the overall variance using the list of subset parameters. Tierney's example of Cook's distance also asks only an `estimate` method of a model. In all of these cases, the procedure uses only one element of the model interface, so the procedure can be a function that calls a model as input rather than being a method internal to the model itself.

Data sets as models

Consider drawing from an urn filled with black, white, and red balls, where 150 black balls, 50 white, and 50 red balls were pulled. This set of observations can be read as a data set of 250 elements, or as a probability model, where drawing from the urn is a Bernoulli draw with observed odds of black equal to 60%, and odds of white and of red equal to 20% each.

That is, the data can be read as a probability mass function (PMF), aka a histogram. With a continuous range of options and $n \to \infty$, the histogram becomes a common probability distribution function (PDF).

Only a few methods need be set to generate the model. First, save the original data set, one possibly weighted observation per row. Then write a `p(new_data, model)` function to check the frequency of `new_data` within the saved data set, write a `draw` function that draws a random row from the data set, and leave everything else to default methods.

For an application, Kullback-Leibler divergence is used to measure a pseudo-distance between distributions. Given two models, a and t, the total divergence is

$$\int_{\forall x} \ln\left(t(x)/a(x)\right) t(x) dx.$$

That is, it is the expected value of $\ln(t(x)/a(x))$ given that x is distributed as the probability distribution associated with model t.

Given two models where one has a `draw` method and both have `log_likelihoods`, one can calculate this integral as follows, where `n` is the number of draws (typically thousands or millions):

```
val = 0
repeat n times:
      x = t.draw //i.e.  draw from the t model
      val = val + t.log_likelihood(x) - a.log_likelihood(x)
return val/n
```

Because any given value of `x` is drawn in proportion to `t.p(x)`, this gives the correctly-weighted expected value.

Given a theoretical model `t`, and an observed data set converted into a PMF, one

could find the divergence of the data from the theoretical distribution. Similar things can be done for other data-versus-model routines, such as Chi-square or Kolmogorov-Smirnov tests of data's divergence from a hypothesized model.

Bayesian updating

Bayesian updating is a process that takes in two model objects and produces a new model object as output.

There are some cases where closed-form solutions are known, but they are relatively few, and can be looked up in common tables of conjugate distributions. If the prior and likelihood are conjugate, then the routine can use the `names`, `parameters` list and `estimate` methods to look up and implement the correct formula for updating:

- Look up the model's `name` for both prior and likelihood in the table of conjugates.
- If the likelihood has a non-`NULL` `parameters` element, then use it.
- Else, use the input `data`, and the `estimate` method of the likelihood, to find parameters for the likelihood.
- Combine the likelihood's `parameters` element with the `parameters` element of the prior via the formula from the conjugate table, to produce the set of parameters for the output model.

If the distribution pair is not in the conjugate table, then we must resort to numeric approximation. Gibbs sampling is an appropriate default because it makes minimal assumptions about the underlying distribution. The procedure basically involves using the `draw` method of the prior to select a candidate $\boldsymbol{\theta}$, then using the `log_likelihood` method of the likelihood to find the odds of the candidate $\boldsymbol{\theta}$ given the data, and using the likelihood to reject or accept the candidate. After distributing all accepted values of $\boldsymbol{\theta}$ into bins, the output is a PMF model.

Because so few assumptions are made about the form of the model, one can update models well beyond the typical closed-form distributions. Suppose that a simulation produces some output parameters given input parameters, and the simulation designer later decides that the input parameters are not fixed but have some prior distribution.

Then this function could Bayesian update using a distribution prior and a likelihood provided by a simulation.

Finally, the output of the Bayesian updating routine is always a model object (either from the conjugate table or a histogram), so one can chain updating steps over several new data sets by simply sending the output from one updating step into the next.

Constrained models

An equality constraint, such as fixing $\beta_1 \equiv 0$, reduces an n dimensional model to an $n-1$ dimensional model.

The problem of generating an equality-constrained model is an opportunity to write a function that takes in a model and outputs a new model. A `fix_parameters` function would take in a model and a mask, where the mask is a set of parameters of the type that the given model would usually use, but with markers (such as `NaN`s) in some locations. Where there is a non-marker value, the model retains that value for all uses of the model; where there is a marker, the model will leave that parameter unknown and free to vary.

The function can use the inputs to generate a new model where every method is an intermediary that takes in the abbreviated set of free parameters, fills in the full set of parameters, and then calls the method of the original model.

Missing data

In the typical estimation, there are a set of unknown parameters and data that is fixed and immutable, and the MLE searches the space of parameters. In a maximum likelihood imputation, the holes in the data are unknown, parameters are assumed known beforehand, and the MLE searches the space of missing data points.

In this case, we treat our parameters as data, and our data as parameters. Further, most of the data are fixed at observed values, leaving us to search over what are hopefully only a few missing points.

One could again implement these modifications via a simple transformation at each method call. First, the `ml_imputation` method would take in data (with markers) and a parameterized model `m`, from which the data is claimed to have been derived (the

best-fitting Multivariate Normal is a common choice).

The `p`, `likelihood`, and `estimate` methods for the fixed model (name it `mf`) can again be simple intermediaries, which swap the parameters with the data, then call the appropriate method of `m`.

This fixes the problem of doing an ML search over data using a system built around conducting ML searches over parameters. But the search is not over all data points, but only those marked missing. This problem was solved above: The `fix_parameters` function can take in `mf` and generate a new model that has only free parameters at the markers.

Conclusion

From my own experience, Definition 2, defining a model as intermediating between data, parameters, and likelihoods draws a good balance between being as descriptive as possible and maintaining simplicity. The model form is brief enough that it can easily be filled in for new models (initially making heavy use of defaults), but has enough substance that functions that call models can do a great deal of work. Because the list of methods in a model is relatively short and defaults are provided, one can produce transformed models with relatively little effort; outside-the-model functions such as Bayesian updating will then work on the new models with no further modification.

Although all models are constrained to fit one form, this form has proven to be a comfortable fit for models from diverse genres. The library discussed in the appendix already includes model objects from the generalized linear model family; distributions of one, two, or many parameters and data dimensions; and nonparametric models such as histograms, Loess, or kernel densities. In my own work in agent-based modeling [Klemens, 2007], I have used this form to write microsimulations and, thanks to the standard form, apply tools originally written for the above more traditional models.

Grammar is irrelevant to the discussion here, and the sort of model object described could be implemented in C, Java, Lisp-stat, R, or any other modern language. What is important is the selection of a structure at the right level of detail, so that models and tools from all paradigms can be described in a useful way.

Appendix: A worked example using Apophenia

Apophenia is a library of functions and models for scientific and statistical computing, written in C. Its original intent was to preserve the speed needed for large-scale simulation, while providing the tools and conveniences familiar to users of high-level statistics packages. For a full overview of Apophenia and the tools one would need to efficiently and conveniently do statistics in C, see Klemens [2008].

Figure 2 gives an overview of the `apop_model` object. The top of the figure provides a partial list of elements, with some notes on their format. The default methods are implemented not via a parent object from which new models inherit, but via dispatch functions that take in a model, check for the given method, and then continue appropriately. The bottom of the figure presents the key dispatch functions.

The remainder of this appendix implements a motivating example from the beginning of the paper: estimating and evaluating a series of models against the same data. Figure 3 presents a basic textbook-style example in C, using the Apophenia library and its implementation of a model object. No important details have been hidden in pseudocode: this example can be compiled and run as-is. Those unfamiliar with C should be able to follow the gist, if not the precise details; C experts will note several points of bad style for the sake of making the code easier for non-experts.

The program (I) generates some synthetic data, (II) estimates the parameters of three different models using that data, then (III) generates another small data set and gets three sets of predicted values using the three estimated models and the small data set.

Here is a quick outline of the code. The focus will be on the comparison of disparate models, so other details of coding will be set aside.

- To keep the example self-contained, phase (I) generates synthetic data for the analysis. `MVN` is a copy of the base Multivariate Normal, with parameters allocated and then set at $[\mu|\Sigma] = \left[\begin{array}{c|cc} 1 & 2.7 & 1.5 \\ 0 & 1.5 & 1.7 \end{array}\right]$. The dimension of the parameters indicate that this copy is a distribution over $\mathbb{R}^2$. The `make_draws` function will make a single draw from this distribution, so applying it to the rows of the `testdata` matrix produces a matrix with one two-element draw per row. It uses

the GNU Scientific Library's RNG system for randomization [Gough, 2003].

- In phase (II), the `model_list` includes an OLS model, a Loess model, and a Multivariate Normal—three very different views of how the two variables relate, and very different methods of computation. The code will make no mention of the specific models after this line.

- Estimation consists of stepping through the list of un-parameterized input models, `model_list`, to produce a list of parameterized output models, `est_list`.

- Having produced an array of estimated models, phase (III) looks at how each predicts outcomes. A small data set is generated, in the same manner as the training set above, then the first column pulled out and replaced with `NaN`s; for the sake of brevity, this routine does not save the original data for calculating and comparing residuals.

- A `for` loop then iterates through the array of estimated models to produce a series of filled-in data sets. Because the `predict` function modifies its input, a copy of the test data is made for each model. The methods produce various outputs, but for brevity and clarity, only the predictions themselves are displayed.

The result (sorted and reformatted for print) is as in Figure 4. OLS and Loess give close predictions all the way across the range, indicating that Loess's greater flexibility adds little for this special case. For the negative end of the range, Loess and OLS give predictions for the first variable that are higher than the Multivariate Normal's predictions. To keep the exposition simple and printable, the example includes only ten test points, but these broad characteristics hold for larger data sets as well.

Bear in mind that the data was generated using a Multivariate Normal model, `MVN`, and the last column of the table is based on a Multivariate Normal with parameters estimated using 10,000 draws of `MVN`. Therefore, the final column yields values very close to the conditional expected value for the true model from which the data was generated.

References

Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

R A Fisher. Two new properties of mathematical likelihood. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 144(852):285–307, March 1934.

Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9(3):491–508, 2000.

Wally R Gilks, N G Best, and K K C Tan. Adaptive rejection Metropolis sampling within Gibbs sampling. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 44(4):455–472, 1995.

Brian Gough, editor. *GNU Scientific Library Reference Manual*. Network Theory, Ltd, 2nd edition, 2003.

Joe R Hill. A general framework for model-based statistics. *Biometrika*, 77(1):115–126, March 1990.

Kosuke Imai, Gary King, and Olivia Lau. Toward a common framework for statistical analysis and development. *Journal of Computational and Graphical Statistics*, 17 (4):892–913, December 2008.

Ben Klemens. *Modeling with Data: Tools and Techniques for Statistical Computing*. Princeton University Press, 2008.

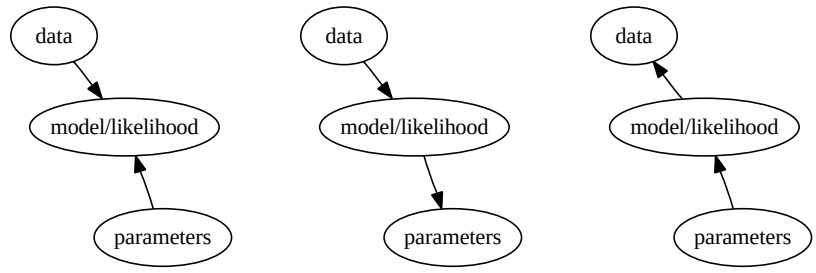
Ben Klemens. Finding optimal agent-based models. Brookings Center on Social and Economic Dynamics Working Paper #49, 2007.

Peter McCullagh. What is a statistical model? *The Annals of Statistics*, 30(5), October 2002.

Yudi Pawitan. *In All Likelihood: Statistical Modeling and Inference Using Likelihood*. Oxford University Press, 2001.

Luke Tierney. Generalized linear models in LISP-STAT. Technical report, 1991.

Luke Tierney. *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*. Wiley-Interscience, 1990.



(a) Probability or log likelihood

(b) Estimation

(c) Random draws, expected value, prediction

Figure 1: A model intermediates between data, likelihoods, and parameters. The type of operation is determined by whether the inputs are data, parameters, or both.

Elements

`name`	Simple text, for display and lookup.
`parameters`	The parameters estimated by the model; possibly free inputs to $P(\cdot, \cdot)$. Of type `apop_data`: a collection of vectors and matrices.
`data`	The data set: fixed inputs to $P(\cdot, \cdot)$. A collection of vectors and matrices, typically one observation per row.
`info`	Auxiliary information such as covariances or residuals.
`settings`	A list of settings groups, providing details for routines such as for MLE or Bayesian updating, or for models such as PMFs.
`double constraint (data, model)`	Test that the data falls within a constraint. If not, return a penalty. Allows for constrained MLE.
`void prep(data, model)`	Check that data is in proper form, allocate settings groups, &c.

Dispatch functions

`apop_model *apop_estimate(data, model)`	Estimate the parameters of the model given data. Produces a model with `parameters`, `info`, and `settings`. Default via MLE.
`double apop_p(data, model)`	Given a model with `parameters`, the probability of the data, parameters, and model.
`double apop_log_likelihood(data, model)`	Same as `apop_p`, but log likelihood.
`double apop_cdf(data, model)`	Integral of the CDF to the given data point. Default via random draws.
`void apop_score(data, gradient, model)`	Fills in the gradient of the log likelihood. Default via delta method.
`void apop_predict(data, model)`	Fill in `NaN`s in the `data` using the expected value given the available data. Given no data, this is the plain expected value. Default via ML imputation.
`void draw(out, r, model)`	Given a `gsl_rng` named `r` (i.e. a random number generator from the GNU Scientific Library), fill `out` with a draw from `model`. Default via Adaptive Rejection Markov Sampling.
`void print(model)`	Display information about the model in a manner familiar to users from the model's paradigm.

Figure 2: Selected elements of Apophenia's `apop_model`, and the dispatch functions for calling `p`, `log_likelihod`, and other methods.

```
#include <apop.h>
gsl_rng *r;
apop_model *MVN;

static void make_draws(gsl_vector *row){ apop_draw(row->data, r, MVN); }

int main(){
    //Phase I: generate data
    r = apop_rng_alloc(13);
    MVN = apop_model_copy(apop_multivariate_normal);
    MVN->parameters = apop_data_alloc(2,2,2);
    apop_data_fill(MVN->parameters, 1, 2.7, 1.5,
                                        0, 1.5, 1.7);
    apop_data *training_data = apop_data_alloc(0, 10000, 2);
    apop_matrix_apply(training_data->matrix, make_draws);

    //Phase II: estimate
    apop_model model_list[] = {apop_loess, apop_ols, apop_multivariate_normal};
    int list_size = 3;

    apop_model *est_list[list_size];
    for (int i=0; i<list_size; i++)
        est_list[i] = apop_estimate(training_data, model_list[i]);

    //Phase III: predict
    apop_data *testdata2 = apop_data_alloc(0, 10, 2);
    apop_matrix_apply(testdata2->matrix, make_draws);
    Apop_col(testdata2, 0, to_nan);
    gsl_vector_set_all(to_nan, NaN);

    apop_data *predictions[list_size];
    for (int i=0; i<list_size; i++){
        predictions[i] = apop_predict(apop_data_copy(testdata2), est_list[i]);
        apop_data_show(predictions[i]);
    }
}
```

Figure 3: Testing and comparing a list of models. The code is dissected in the text.

X	Loess	OLS	MVN
-2.256	-0.955	-0.975	-1.049
-2.027	-0.757	-0.775	-0.978
-1.144	-3.39e-4	-4.32e-3	-0.178
-0.541	0.506	0.522	0.303
0.129	1.098	1.106	1.110
0.223	1.186	1.188	0.929
1.131	1.971	1.981	1.496
1.281	2.103	2.112	1.930
1.542	2.336	2.340	2.088
3.160	3.839	3.752	3.685

Figure 4: Let data be of the form $[Y\,X]$. The table gives three different predictions for Y. Output from Figure 3.