Chapter 1

Adding Models and Methods to Zelig

Zelig is highly modular. You can add methods to Zelig *and*, if you wish, release your programs as a stand-alone package. By making your package compatible with Zelig, you will advertise your package and help it achieve a widespread distribution.

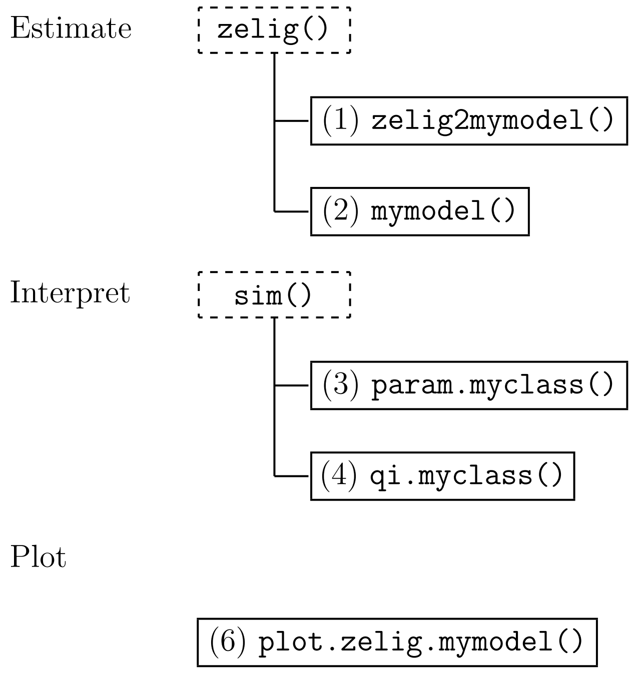
This chapter assumes that your model is written as a function that takes a user-defined formula and data set (see Chapter **??**), and returns a list of output that includes (at the very least) the estimated parameters and terms that describe the data used to fit the model. You should choose a class (either S3 or S4 class) for this list of output, and provide appropriate methods for generic functions such as `summary()`, `print()`, `coef()` and `vcov()`.

To add new models to Zelig, you need to provide six R functions, illustrated in Figure **??**. Let `mymodel` be a new model with class `"myclass"`.

These functions are as follows:

1. `zelig2mymodel()` translates `zelig()` arguments into the arguments for `mymodel()`.
2. `mymodel()` estimates your statistical procedure.
3. `param.myclass()` simulates parameters for your model. Alternatively, if your model's parameters consist of one vector with a correspondingly observed variance-covariance matrix, you may write *two* simple functions to substitute for `param.myclass()`:
   (a) `coef.myclass()` to extract the coefficients from your model output, and
   (b) `vcov.myclass()` to extract the variance-covariance matrix from your model.
4. `qi.myclass()` calculates expected values, simulates predicted values, and generates other quantities of interest for your model (applicable only to models that take explanatory variables).
5. `plot.zelig.mymodel()` to plot the simulated quantities of interest from your model.
6. A **reference manual page** to document the model. (See Section **??**)
7. A function (`describe.mymodel()`) describing the inputs to your model, for use with a graphical user interface. (See Section **??**).

Figure 1.1: Six functions (solid boxes) to implement a new Zelig model



8. An optional **demo script** `mymodel.R` which contains commented code for the models contained in the example section of your reference manual page.

1.1 Making the Model Compatible with Zelig

You can develop a model, write the model-fitting function, and test it within the Zelig framework without explicit intervention from the Zelig team. (We are, of course, happy to respond to any questions or suggestions for improvement.)

Zelig's modularity relies on two R programming conventions:

1. **wrappers**, which pass arguments from R functions to other R functions or to foreign function calls (such as C, C++, or Fortran functions); and

2. **classes**, which tell generic functions how to handle objects of a given class.

Specific methods for R generic functions take the general form: `method.class()`, where `method` is the name of the generic procedure to be performed and `class` is the class of the object. You may define, for example, `summary.contrib()` to summarize the output of your model. Note that for S4 classes, the name of generic functions does not have to be `method.class()` so long as users can call them via `method()`.

To Work with `zelig()`

Zelig has implemented a unique method for incorporating new models which lets contributors test their models *within* the Zelig framework *without* any modification of the `zelig()` function itself.

Using a wrapper function `zelig2contrib()` (where `contrib` is the name of your new model), `zelig2contrib()` redefines the inputs to `zelig()` to work with the inputs you need for your function `contrib()`. For example, if you type

```
zelig(..., model = "normal.regression")
```

`zelig()` looks for a `zelig2normal.regression()` wrapper in any environment (either attached libraries or your workspace). If the wrapper exists, then `zelig()` runs the model.

If you have a pre-existing model, writing a `zelig2contrib()` function is quite easy. Let's say that your model is `contrib()`, and takes the following arguments: `formula`, `data`, `weights`, and `start`. The `zelig()` function, in contrast, only takes the `formula`, `data`, `model`, and `by` arguments. You may use the `...` to pass additional arguments from `zelig()` to `zelig2contrib()`, and `<- NULL` to omit the elements you do not need. Continuing the Normal regression example from Section **??**, let `formula`, `model`, and `data` be the inputs to `zelig()`, `M` is the number of subsets, and `...` are the additional arguments not defined in the `zelig()` call, but passed to `normal.regression()`.

```
zelig2normal.regression <- function(formula, model, data, M, ...) {
  mf <- match.call(expand.dots = TRUE)                      # [1]
  mf$model <- mf$M <- NULL                                  # [2]
  mf[[1]] <- as.name("normal.regression")                   # [3]
  as.call(mf)                                               # [4]
}
```

The bracketed numbers above correspond to the comments below:

1. Create a call (an expression to be evaluated) by creating a list of the arguments in `zelig2normal.regression()`, including the extra arguments taken by `normal.regression()`, but not by `zelig()`. All wrappers must take the same standardized arguments (`formula`, `model`, `data`, and `M`), which may be used in the wrapper function to manipulate the `zelig()` call into the `normal.regression()` call. Additional arguments to `normal.regression()`, such as `start.val` are passed implicitly from `zelig()` using the `...` operator.

2. Erase extraneous information from the call object `mf`. In this wrapper, `model` and `M` are not used. In other models, these are used to further manipulate the call, and so are included in the standard inputs to all wrappers.

3. Reassign the first element of the call (currently `zelig2normal.regression`) with the name of the function to be evaluated, `normal.regression()`.

4. Return the call to `zelig()`, which will evaluate the call for each multiply-imputed data set, each subset defined in `by`, or simply `data`.

If you use an S4 class to represent your model, say `mymodel`, within `zelig.default()`, Zelig's internal function, `create.ZeligS4()`, automatically creates a new S4 class called `ZeligS4mymodel` in the global environment with two additional slots. These include `zelig`, which stores the name of the model, and `zelig.data`, which stores the data frame if `save.data=TRUE` and is empty otherwise. These names are taken from the original call. This new output inherits the original class `mymodel` so all the generic functions associated with `mymodel` should still work. If you would like to see an example, see the models implemented using the VGAM package, such as multinomial probit.

To Work with `setx()`

In the case of `setx()`, most models will use `setx.default()`, which in turn relies on the generic R function `model.matrix()`. For this procedure to work, your list of output must include:

- `terms`, created by `model.frame()`, or manually;
- `formula`, the formula object input by the user;
- `xlevels`, which define the strata in the explanatory variables; and
- `contrasts`, an optional element which defines the type of factor variables used in the explanatory variables. See `help(contrasts)` for more information.

If your model output does not work with `setx.default()`, you must write your own `setx.contrib()` function. For example, models fit to multiply-imputed data sets have output from `zelig()` of class `"MI"`. The special `setx.MI()` wrapper pre-processes the `zelig()` output object and passes the appropriate arguments to `setx.default()`.

Compatibility with `sim()`

Simulating quantities of interest is an integral part of interpreting model results. To use the functionality built into the Zelig `sim()` procedure, you need to provide a way to simulate parameters (called a `param()` function), and a method for calculating or drawing quantities of interest from the simulated parameters (called a `qi()` function).

Simulating Parameters Whether you choose to use the default method, or write a model-specific method for simulating parameters, these functions require the same three inputs:

- `object`: the estimated model or `zelig()` output.
- `num`: the number of simulations.
- `bootstrap`: either `TRUE` or `FALSE`.

The output from `param()` should be either

- If `bootstrap = FALSE` (default), an matrix with rows corresponding to simulations and columns corresponding to model parameters. Any ancillary parameters should be included in the output matrix.
- If `bootstrap = TRUE`, a vector containing all model parameters, including ancillary parameters.

There are two ways to simulate parameters:

1. Use the `param.default()` function to extract parameters from the model and, if bootstrapping is not selected, simulate coefficients using asymptotic normal approximation. The `param.default()` function relies on two R functions:

   (a) `coef()`: extracts the coefficients. Continuing the Normal regression example from above, the appropriate `coef.normal()` function is simply:

   ```
   coef.normal <- function(object)
     object$coefficients
   ```

   (b) `vcov()`: extracts the variance-covariance matrix. Again continuing the Poisson example from above:

   ```
   vcov.normal <- function(object)
     object$variance
   ```

2. Alternatively, you can write your own `param.contrib()` function. This is appropriate when:

   (a) Your model has auxiliary parameters, such as σ in the case of the Normal distribution.

   (b) Your model performs some sort of correction to the coefficients or the variance-covariance matrix, which cannot be performed in either the `coef.contrib()` or the `vcov.contrib()` functions.

   (c) Your model does not rely on asymptotic approximation to the log-likelihood. For Bayesian Markov-chain monte carlo models, for example, the `param.contrib()` function (`param.MCMCzelig()` in this case) simply extracts the model parameters simulated in the model-fitting function.

   Continuing the Normal example,

   ```
   param.normal <- function(object, num = NULL, bootstrap = FALSE,
                            terms = NULL) {
     if (!bootstrap) {
       par <- mvrnorm(num, mu = coef(object), Sigma = vcov(object))
       Beta <- parse.par(par, terms = terms, eqn = "mu")
   ```

```
    sigma2 <- exp(parse.par(par, terms = terms, eqn = "sigma2"))
    res <- cbind(Beta, sigma2)
  }
  else {
    par <- coef(object)
    Beta <- parse.par(par, terms = terms,  eqn = "mu")
    sigma2 <- exp(parse.par(par, terms = terms, eqn = "sigma2"))
    res <- c(coef, sigma2)
  }
  res
}
```

Calculating Quantities of Interest All models require a model-specific method for calculating quantities of interest from the simulated parameters. For a model of class `contrib`, the appropriate `qi()` function is `qi.contrib()`. This function should calculate, at the bare minimum, the following quantities of interest:

- `ev`: the expected values, calculated from the analytic solution for the expected value as a function of the systematic component and ancillary parameters.
- `pr`: the predicted values, drawn from a distribution defined by the predicted values. If R does not have a built-in random generator for your function, you may take a random draw from the uniform distribution and use the inverse CDF method to calculate predicted values.
- `fd`: first differences in the expected value, calculated by subtracting the expected values given the specified `x` from the expected values given `x1`.
- `ate.ev`: the average treatment effect calculated using the expected values `ev`. This is simply `y - ev`, averaged across simulations for each observation.
- `ate.pr`: the average treatment effect calculated using the predicted values `pr`. This is simply `y - pr`, averaged across simulations for each observation.

The required arguments for the `qi()` function are:

- `object`: the zelig output object.
- `par`: the simulated parameters.
- `x`: the matrix of explanatory variables (created using `setx()`).
- `x1`: the optional matrix of alternative values for first differences (also created using `setx()`). If first differences are inappropriate for your model, you should put in a `warning()` or `stop()` if `x1` is not `NULL`.

- y: the optional vector or matrix of dependent variables (for calculating average treatment effects). If average treatment effects are inappropriate for your model, you should put in a `warning()` or `stop()` if conditional prediction has been selected in the `setx()` step.

Continuing the Normal regression example from above, the appropriate `qi.normal()` function is as follows:

```
qi.normal <- function(object, par, x, x1 = NULL, y = NULL) {
  Beta <- parse.par(par, eqn = "mu")                         # [1]
  sigma2 <- parse.par(par, eqn = "sigma2")                   # [2]
  ev <- Beta %*% t(x)                                        # [3a]
  pr <- matrix(NA, ncol = ncol(ev), nrow = nrow(ev))
  for (i in 1:ncol(ev))
    pr[,i] <- rnorm(length(ev[,i]), mean = ev[,i],           # [4]
                    sigma = sd(sigma2[i]))
  qi <- list(ev = ev, pr = pr)
  qi.name <- list(ev = "Expected Values: E(Y|X)",
                  pr = "Predicted Values: Y|X")
  if (!is.null(x1)){
    ev1 <- par %*% t(x1)                                     # [3b]
    qi$fd <- ev1 - ev
    qi.name$fd <- "First Differences in Expected Values: E(Y|X1)-E(Y|X)"
  }
  if (!is.null(y)) {
    yvar <- matrix(rep(y, nrow(par)), nrow = nrow(par), byrow = TRUE)
    tmp.ev <- yvar - qi$ev
    tmp.pr <- yvar - qi$pr
    qi$ate.ev <- matrix(apply(tmp.ev, 1, mean), nrow = nrow(par))
    qi$ate.pr <- matrix(apply(tmp.pr, 1, mean), nrow = nrow(par))
    qi.name$ate.ev <- "Average Treatment Effect: Y - EV"
    qi.name$ate.pr <- "Average Treatment Effect: Y - PR"
  }
  list(qi=qi, qi.name=qi.name)
}
```

There are five lines of code commented above. By changing these five lines in the following *four* ways, you can write `qi()` function appropriate to almost any model:

1. Extract any systematic parameters by substituting the name of your systematic parameter (defined in `describe.mymodel()`).

2. Extract any ancillary parameters (defined in `describe.mymodel()`) by substituting their names here.

3. Calculate the expected value using the inverse link function and $\eta = X\beta$. (For the normal model, this is linear.) You will need to make this change in two places, at Comment [3a] and [3b].

4. Replace `rnorm()` with a function that takes random draws from the stochastic component of your model.

1.2 Getting Ready for the GUI

Zelig can work with a variety of graphical user interfaces (GUIs). GUIs work by knowing *a priori* what a particular model accepts, and presenting only those options to the user in some sort of graphical interface. Thus, in order for your model to work with a GUI, you must describe your model in terms that the GUI can understand. For models written using the guidelines in Chapter **??**, your model will be compatible with (at least) the Virtual Data Center GUI. For pre-existing models, you will need to create a `describe.*()` function for your model following the examples in Section **??**.

1.3 Formatting Reference Manual Pages

One of the primary advantages of Zelig is that it fully documents the included models, in contrast to the programming-orientation of R documentation which is organized by function. Thus, we ask that Zelig contributors provide similar documentation, including the syntax and arguments passed to `zelig()`, the systematic and stochastic components to the model, the quantities of interest, the output values, and further information (including references). There are several ways to provide this information:

- If you have an existing package documented using the .Rd help format, `help.zelig()` will automatically search R-help in addition to Zelig help.

- If you have an existing package documented using on-line HTML files with static URLs (like Zelig or MatchIt), you need to provide a `PACKAGE.url.tab` file which is a two-column table containing the name of the function in the first column and the url in the second. (Even though the file extension is `.url.tab`, the file should be a tab- or space-delimited text file.) For example:

```
command        http://gking.harvard.edu/zelig/docs/Main_Commands.html
model          http://gking.harvard.edu/zelig/docs/Specific_Models.html
```

  If you wish to test to see if your `.url.tab` files works, simply place it in your R library/Zelig/data/ directory. (You do not need to reinstall Zelig to test your `.url.tab` file.)

- Preferred method: You may provide a LaTeX 2ε `.tex` file. This document uses the book style and supports commands from the following packages: `graphicx`, `natbib`, `amsmath`, `amssymb`, `verbatim`, `epsf`, and `html`. Because model pages are incorporated into this document using `\include{}`, you should make sure that your document compiles before submitting it. Please adhere to the following conventions for your model page:

  1. All mathematical formula should be typeset using the `equation*` and `array`, `eqnarray*`, or `align` environments. Please avoid `displaymath`. (It looks funny in html.)
  2. All commands or R objects should use the `texttt` environment.
  3. The model begins as a subsection of a larger document, and sections within the model page are of sub-subsection level.
  4. For stylistic consistency, please avoid using the `description` environment.

  Each LaTeX model page should include the following elements. Let `contrib` specify the new model.

Help File Template

```
\subsection{{\tt contrib}: Full Name for [type] Dependent Variables}
\label{contrib}

\subsubsection{Syntax}

\subsubsection{Examples}
\begin{enumerate}
\item First Example
\item Second Example
\end{enumerate}

\subsubsection{Model}
\begin{itemize}
\item The observation mechanism, if applicable.
\item The stochastic component.
\item The systematic component.
\end{itemize}

\subsubsection{Quantities of Interest}
\begin{itemize}
\item The expected value of your distribution, including the formula
  for the expected value as a function of the systemic component and
  ancillary paramters.
\item The predicted value drawn from the distribution defined by the
       corresponding expected value.
\item The first difference in expected values, given when x1 is specified.
\item Other quantities of interest.
\end{itemize}

\subsubsection{Output Values}
\begin{itemize}
\item From the {\tt zelig()} output stored in {\tt z.out}, you may
  extract:
   \begin{itemize}
   \item
   \item
   \end{itemize}
\item From {\tt summary(z.out)}, you may extract:
   \begin{itemize}
   \item
   \item
   \end{itemize}
\item From the {\tt sim()} output stored in {\tt s.out}:
   \begin{itemize}
   \item
   \item
   \end{itemize}
\end{itemize}

\subsubsection{Further Information}

\subsubsection{Contributors}
```