Chapter 1

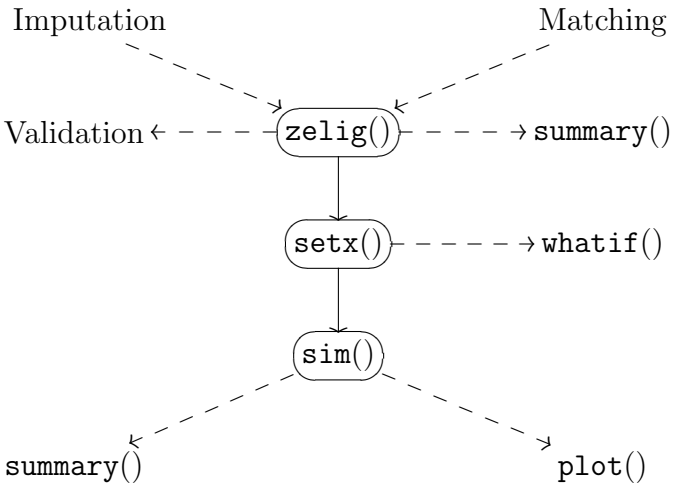
Statistical Commands

1.1 Zelig Commands

1.1.1 Quick Overview

For any statistical model, Zelig does its work with a combination of three commands.

Figure 1.1: Main Zelig commands (solid arrows) and some options (dashed arrows)



1. Use `zelig()` to run the chosen statistical model on a given data set, with a specific set of variables. For standard likelihood models, for example, this step estimates the coefficients, other model parameters, and a variance-covariance matrix. In addition, you may choose from a variety of options:

- Pre-process data: Prior to calling `zelig()`, you may choose from a variety of data pre-processing commands (matching or multiple imputation, for example) to make your statistical inferences more accurate.
- Summarize model: After calling `zelig()`, you may summarize the fitted model output using `summary()`.
- Validate model: After calling `zelig()`, you may choose to validate the fitted model. This can be done, for example, by using cross-validation procedures and diagnostics tools.

2. Use `setx()` to set each of the explanatory variables to chosen (actual or counterfactual) values in preparation for calculating quantities of interest. After calling `setx()`, you may use WhatIf to evaluate these choices by determining whether they involve interpolation (i.e., are inside the convex hull of the observed data) or extrapolation, as well as how far these counterfactuals are from the data. Counterfactuals chosen in `setx()` that involve extrapolation far from the data can generate considerably more model dependence (see **?**, **?**, **?**).

3. Use `sim()` to draw simulations of your quantity of interest (such as a predicted value, predicted probability, risk ratio, or first difference) from the model. (These simulations may be drawn using an asymptotic normal approximation (the default), bootstrapping, or other methods when available, such as directly from a Bayesian posterior.) After calling `sim()`, use any of the following to summarize the simulations:

   - The `summary()` function gives a numerical display. For multiple `setx()` values, `summary()` lets you summarize simulations by choosing one or a subset of observations.
   - If the `setx()` values consist of only one observation, `plot()` produces density plots for each quantity of interest.

Whenever possible, we use `z.out` as the `zelig()` output object, `x.out` as the `setx()` output object, and `s.out` as the `sim()` output object, but you may choose other names.

1.1.2 Examples

- Use the `turnout` data set included with Zelig to estimate a logit model of an individual's probability of voting as function of race and age. Simulate the predicted probability of voting for a white individual, with age held at its mean:

```
> data(turnout)
> z.out <- zelig(vote ~ race + age, model = "logit", data = turnout)
> x.out <- setx(z.out, race = "white")
> s.out <- sim(z.out, x = x.out)
> summary(s.out)
```

- Compute a first difference and risk ratio, changing education from 12 to 16 years, with other variables held at their means in the data:

```
> data(turnout)
> z.out <- zelig(vote ~ race + educate, model = "logit", data = turnout)
> x.low <- setx(z.out, educate = 12)
> x.high <- setx(z.out, educate = 16)
> s.out <- sim(z.out, x = x.low, x1 = x.high)
> summary(s.out)                                          # Numerical summary.
> plot(s.out)                                             # Graphical summary.
```

- Calculate expected values for every observation in your data set:

```
> data(turnout)
> z.out <- zelig(vote ~ race + educate, model = "logit", data = turnout)
> x.out <- setx(z.out, fn = NULL)
> s.out <- sim(z.out, x = x.out)
> summary(s.out)
```

- Use five multiply imputed data sets from **?** in an ordered logit model:

```
> data(immi1, immi2, immi3, immi4, immi5)
> z.out <- zelig(as.factor(ipip) ~ wage1992 + prtyid + ideol,
                 model = "ologit",
                 data = mi(immi1, immi2, immi3, immi4, immi5))
```

- Use the nearest propensity score matching via *MatchIt* package, and then calculate the conditional average treatment effect of the job training program based on the linear regression model:

```
> library(MatchIt)
> data(lalonde)
> m.out <- matchit(treat ~ re74 + re75 + educ + black + hispan + age,
                   data = lalonde, method = "nearest")
> m.data <- match.data(m.out)
> z.out <- zelig(re78 ~ treat + distance + re74 + re75 + educ + black +
                 hispan + age, data = m.data, model = "ls")
> x.out0 <- setx(z.out, fn = NULL, treat = 0)
> x.out1 <- setx(z.out, fn = NULL, treat = 1)
> s.out <- sim(z.out, x=x.out0, x1=x.out1)
> summary(s.out)
```

- Validate the fitted model using the leave-one-out cross validation procedure and calculating the average squared prediction error via *boot* package. For example:

```
> library(boot)
> data(turnout)
> z.out <- zelig(vote ~ race + educate, model = "logit", data = turnout)
> cv.out <- cv.glm(z.out, data = turnout)
> print(cv.out$delta)
```

1.1.3 Details

1. `z.out <- zelig(formula, model, data, by = NULL, ...)`

   The `zelig()` command estimates a selected statistical model given the specified data. You may name the output object (`z.out` above) anything you desire. You must include three required arguments, in the following order:

   (a) `formula` takes the form `y ~ x1 + x2`, where `y` is the dependent variable and `x1` and `x2` are the explanatory variables, and `y`, `x1`, and `x2` are contained in the same dataset. The `+` symbol means "inclusion" not "addition." You may include interaction terms in the form of `x1*x2` without having to compute them in prior steps or include the main effects separately. For example, R treats the formula `y ~ x1*x2` as `y ~ x1 + x2 + x1*x2`. To prevent R from automatically including the separate main effect terms, use the `I()` function, thus: `y ~ I(x1 * x2)`.

   (b) `model` lets you choose which statistical model to run. You must put the name of the model in quotation marks, in the form `model = "ls"`, for example. See Section **??** for a list of currently supported models.

   (c) `data` specifies the data frame containing the variables called in the formula, in the form `data = mydata`. Alternatively, you may input multiply imputed datasets in the form `data = mi(data1, data2, ...)`.[1] If you are working with matched data created using MatchIt, you may create a data frame within the `zelig()` statement by using `data = match.data(...)`. In all cases, the data frame or MatchIt object must have been previously loaded into the working memory.

   (d) `by` (an optional argument which is by default `NULL`) allows you to choose a factor variable (see Section **??**) in the data frame as a subsetting variable. For each of the unique strata defined in the `by` variable, `zelig()` does a separate run of the specified model. The variable chosen should *not* be in the formula, because there will be no variance in the `by` variable in the subsets. If you have one data set for all 191 countries in the UN, for example, you may use the `by` option to run the same model 191 times, once on each country, all with a single `zelig()` statement. You may also use the `by` option to run models on MatchIt subclasses.

[1] Multiple imputation is a method of dealing with missing values in your data which is more powerful than the usual list-wise deletion approach. You can create multiply imputed datasets with a program such as Amelia; see King, Honaker, Joseph, Scheve (2000).

(e) The output object, `z.out`, contains all of the options chosen, including the name of the data set. Because data sets may be large, Zelig does not store the full data set, but only the name of the dataset. Every time you use a Zelig function, it looks for the dataset with the appropriate name in working memory. (Thus, it is critical that you do *not* change the name of your data set, or perform any additional operations on your selected variables between calling `zelig()` and `setx()`, or between `setx()` and `sim()`.)

(f) If you would like to view the regression output at this intermediate step, type `summary(z.out)` to return the coefficients, standard errors, t-statistics and p-values. We recommend instead that you calculate quantities of interest; creating `z.out` is only the first of three steps in this task.

2. `x.out <- setx(z.out, fn = list(numeric = mean, ordered = median, others = mode), data = NULL, cond = FALSE, ...)`

   The `setx()` command lets you choose values for the explanatory variables, with which `sim()` will simulate quantities of interest. There are two types of `setx()` procedures:

   - You may perform the usual *unconditional* prediction (by default, `cond = FALSE`), by explicitly choosing the values of each explanatory variable yourself or letting `setx()` compute them, either from the data used to create `z.out` or from a new data set specified in the optional `data` argument. You may also compute predictions for all observed values of your explanatory variables using `fn = NULL`.
   - Alternatively, for advanced uses, you may perform *conditional* prediction (`cond = TRUE`), which predicts certain quantities of interest by conditioning on the observed value of the dependent variable. In a simple linear regression model, this procedure is not particularly interesting, since the conditional prediction is merely the observed value of the dependent variable for that observation. However, conditional prediction is extremely useful for other models and methods, including the following:
     - In a matched sampling design, the sample average treatment effect for the treated can be estimated by computing the difference between the observed dependent variable for the treated group and their expected or predicted values of the dependent variable under no treatment (**?**).
     - With censored data, conditional prediction will ensure that all predicted values are greater than the censored observed values (**?**).
     - In ecological inference models, conditional prediction guarantees that the predicted values are on the tomography line and thus restricted to the known bounds (**??**).
     - The conditional prediction in many linear random effects (or Bayesian hierarchical) models is a weighted average of the unconditional prediction and the value of the dependent variable for that observation, with the weight being

an estimable function of the accuracy of the unconditional prediction (see **?**). When the unconditional prediction is highly certain, the weight on the value of the dependent variable for this observation is very small, hence reducing inefficiency; when the unconditional prediction is highly uncertain, the relative weight on the unconditional prediction is very small, hence reducing bias. Although the simple weighted average expression no longer holds in nonlinear models, the general logic still holds and the mean square error of the measurement is typically reduced (see **?**).

In these and other models, conditioning on the observed value of the dependent variable can vastly increase the accuracy of prediction and measurement.

The `setx()` arguments for **unconditional** prediction are as follows:

(a) `z.out`, the `zelig()` output object, must be included first.

(b) You can set particular explanatory variables to specified values. For example:

```
> z.out <- zelig(vote ~ age + race, model = "logit", data = turnout)
> x.out <- setx(z.out, age = 30)
```

`setx()` sets the variables *not* explicitly listed to their mean if numeric, and their median if ordered factors, and their mode if unordered factors, logical values, or character strings. Alternatively, you may specify one explanatory variable as a range of values, creating one observation for every unique value in the range of values:[2]

```
> x.out <- setx(z.out, age = 18:95)
```

This creates 78 observations with with age set to 18 in the first observation, 19 in the second observation, up to 95 in the 78th observation. The other variables are set to their default values, but this may be changed by setting `fn`, as described next.

(c) Optionally, `fn` is a list which lets you to choose a different function to apply to explanatory variables of class

- `numeric`, which is `mean` by default,
- `ordered` factor, which is `median` by default, and
- `other` variables, which consist of logical variables, character string, and unordered factors, and are set to their `mode` by default.

While any function may be applied to numeric variables, `mean` will default to median for ordered factors, and mode is the only available option for other types of variables. In the special case, `fn = NULL`, `setx()` returns all of the observations.

[2] If you allow more than one variable to vary at a time, you risk confounding the predictive effect of the variables in question.

(d) You cannot perform other math operations within the `fn` argument, but can use the output from one call of `setx` to create new values for the explanatory variables. For example, to set the explanatory variables to one standard deviation below their mean:

```
> X.sd <- setx(z.out, fn = list(numeric = sd))
> X.mean <- setx(z.out, fn = list(numeric = mean))
> x.out <- X.mean - X.sd
```

(e) Optionally, `data` identifies a new data frame (rather than the one used to create `z.out`) from which the `setx()` values are calculated. You can use this argument to set values of the explanatory variables for hold-out or out-of-sample fit tests.

(f) The `cond` is always `FALSE` for unconditional prediction.

If you wish to calculate risk ratios or first differences, call `setx()` a second time to create an additional set of the values for the explanatory variables. For example, continuing from the example above, you may create an alternative set of explanatory variables values one standard deviation above their mean:

```
> x.alt <- X.mean + X.sd
```

The required arguments for **conditional** prediction are as follows:

(a) `z.out`, the `zelig()` output object, must be included first.

(b) `fn`, which equals `NULL` to indicate that all of the observations are selected. You may only perform conditional inference on actual observations, not the mean of observations or any other function applied to the observations. Thus, if `fn` is missing, but `cond = TRUE`, `setx()` coerces `fn = NULL`.

(c) `data`, the data for conditional prediction.

(d) `cond`, which equals `TRUE` for conditional prediction.

Additional arguments, such as any of the variable names, are ignored in conditional prediction since the actual values of that observation are used.

3. `s.out <- sim(z.out, x = x.out, x1 = NULL, num = c(1000, 100), bootstrap = FALSE, bootfn = NULL, ...)`

   The `sim()` command simulates quantities of interest given the output objects from `zelig()` and `setx()`. This procedure uses only the assumptions of the statistical model. The `sim()` command performs either unconditional or conditional prediction depending on the options chosen in `setx()`.

   The arguments are as follows for **unconditional** prediction:

   (a) `z.out`, the model output from `zelig()`.

(b) `x`, the output from the `setx()` procedure performed on the model output.

(c) Optionally, you may calculate first differences by specifying `x1`, an additional `setx()` object. For example, using the `x.out` and `x.alt`, you may generate first differences using:

```
> s.out <- sim(z.out, x = x.out, x1 = x.alt)
```

(d) By default, the number of simulations, `num`, equals 1000 (or 100 simulations if bootstrap is selected), but this may be decreased to increase computational speed, or increased for additional precision.

(e) Zelig simulates parameters from classical *maximum likelihood* models using asymptotic normal approximation to the log-likelihood. This is the same assumption as used for frequentist hypothesis testing (which is of course equivalent to the asymptotic approximation of a Bayesian posterior with improper uniform priors). See King, Tomz, and Wittenberg (2000). For *Bayesian models*, Zelig simulates quantities of interest from the posterior density, whenever possible. For *robust Bayesian models*, simulations are drawn from the identified class of Bayesian posteriors.

(f) Alternatively, you may set `bootstrap = TRUE` to simulate parameters using bootstrapped data sets. If your dataset is large, bootstrap procedures will usually be more memory intensive and time-consuming than simulation using asymptotic normal approximation. The type of bootstrapping (including the sampling method) is determined by the optional argument `bootfn`, described below.

(g) If `bootstrap = TRUE` is selected, `sim()` will bootstrap parameters using the default `bootfn`, which re-samples from the data frame with replacement to create a sampled data frame of the same number of observations, and then re-runs `zelig()` (inside `sim()`) to create one set of bootstrapped parameters. Alternatively, you may create a function outside the `sim()` procedure to handle different bootstrap procedures. Please consult `help(boot)` for more details.[3]

For **conditional** prediction, `sim()` takes only two required arguments:

(a) `z.out`, the model output from `zelig()`.

(b) `x`, the conditional output from `setx()`.

(c) Optionally, for duration models, `cond.data`, which is the `data` argument from `setx()`. For models for duration dependent variables (see Section **??**), `sim()` must impute the uncensored dependent variables before calculating the average treatment effect. Inputting the `cond.data` allows `sim()` to generate appropriate values.

Additional arguments are ignored or generate error messages.

[3]If you choose to create your own `bootfn`, it must include the the following three arguments: `data`, the original data frame; one of the sampling methods described in `help(boot)`; and `object`, the original `zelig()` output object. The alternative bootstrapping function must sample the data, fit the model, and extract the model-specific parameters.

Presenting Results

1. Use `summary(s.out)` to print a summary of your simulated quantities. You may specify the number of significant digits as:

   ```
   > print(summary(s.out), digits = 2)
   ```

2. Alternatively, you can plot your results using `plot(s.out)`.

3. You can also use `names(s.out)` to see the names and a description of the elements in this object and the `$` operator to extract particular results. For most models, these are: `s.out$qi$pr` (for predicted values), `s.out$qi$ev` (for expected values), and `s.out$qi$fd` (for first differences in expected values). For the logit, probit, multinomial logit, ordinal logit, and ordinal probit models, quantities of interest also include `s.out$qi$rr` (the risk ratio).

1.2 Supported Models

We list here all models implemented in Zelig, organized by the nature of the dependent variable(s) to be predicted, explained, or described.

1. **Continuous Unbounded** dependent variables can take any real value in the range $(-\infty, \infty)$. While most of these models take a continuous dependent variable, Bayesian factor analysis takes multiple continuous dependent variables.

   (a) `"ls"`: The *linear least-squares* (see Section **??**) calculates the coefficients that minimize the sum of squared residuals. This is the usual method of computing linear regression coefficients, and returns unbiased estimates of β and σ^2 (conditional on the specified model).

   (b) `"normal"`: The *Normal* (see Section **??**) model computes the maximum-likelihood estimator for a Normal stochastic component and linear systematic component. The coefficients are identical to `ls`, but the maximum likelihood estimator for σ^2 is consistent but biased.

   (c) `"normal.bayes"`: The *Bayesian Normal* regression model (Section **??**) is similar to maximum likelihood Gaussian regression, but makes valid small sample inferences via draws from the exact posterior and also allows for priors.

   (d) `"netls"`: The *network least squares* regression (Section **??**) is similar to least squares regression for continuous-valued proximity matrix dependent variables. Proximity matrices are also known as sociomatrices, adjacency matrices, and matrix representations of directed graphs.

   (e) `"tobit"`: The *tobit* regression model (see Section **??**) is a Normal distribution with left-censored observations.

(f) `"tobit.bayes"`: The *Bayesian tobit* distribution (see Section **??**) is a Normal distribution that has either left and/or right censored observations.

(g) `"arima"`: Use *auto-regressive, integrated, moving-average* (ARIMA) models for time series data (see Section **??**.

(h) `"factor.bayes"`: The *Bayesian factor analysis* model (see Section **??**) estimates multiple observed continuous dependent variables as a function of latent explanatory variables.

2. **Dichotomous** dependent variables consist of two discrete values, usually $(0, 1)$.

(a) `"logit"`: *Logistic regression* (see Section **??**) specifies $\Pr(Y = 1)$ to be a(n inverse) logistic transformation of a linear function of a set of explanatory variables.

(b) `"relogit"`: The *rare events logistic* regression option (see Section **??**) estimates the same model as the logit, but corrects for bias due to rare events (when one of the outcomes is much more prevalent than the other). It also optionally uses prior correction to correct for choice-based (case-control) sampling designs.

(c) `"logit.bayes"`: *Bayesian logistic regression* (see Section **??**) is similar to maximum likelihood logistic regression, but makes valid small sample inferences via draws from the exact posterior and also allows for priors.

(d) `"probit"`: *Probit regression* (see Section **??**) Specifies $\Pr(Y = 1)$ to be a(n inverse) CDF normal transformation as a linear function of a set of explanatory variables.

(e) `"probit.bayes"`: *Bayesian probit* regression (see Section **??**) is similar to maximum likelihood probit regression, but makes valid small sample inferences via draws from the exact posterior and also allows for priors.

(f) `"netlogit"`: The *network logistic* regression (Section **??**) is similar to logistic regression for binary-valued proximity matrix dependent variables. Proximity matrices are also known as sociomatrices, adjacency matrices, and matrix representations of directed graphs.

(g) `"blogit"`: The *bivariate logistic* model (see Section **??**) models $\Pr(Y_{i1} = y_1, Y_{i2} = y_2)$ for $(y_1, y_2) = (0, 0), (0, 1), (1, 0), (1, 1)$ according to a bivariate logistic density.

(h) `"bprobit"`: The *bivariate probit* model (see Section **??**) models $\Pr(Y_{i1} = y_1, Y_{i2} = y_2)$ for $(y_1, y_2) = (0, 0), (0, 1), (1, 0), (1, 1)$ according to a bivariate normal density.

(i) `"irt1d"`: The *one-dimensional item response* model (see Section **??**) takes multiple dichotomous dependent variables and models them as a function of *one* latent (unobserved) explanatory variable.

(j) `"irtkd"`: The *k-dimensional item response* model (see Section **??**) takes multiple dichotomous dependent variables and models them as a function of k latent (unobserved) explanatory variables.

3. **Ordinal** are used to model ordered, discrete dependent variables. The values of the outcome variables (such as kill, punch, tap, bump) are ordered, but the distance between any two successive categories is not known exactly. Each dependent variable may be thought of as linear, with one continuous, unobserved dependent variable observed through a mechanism that only returns the ordinal choice.

   (a) `"ologit"`: The *ordinal logistic* model (see Section **??**) specifies the stochastic component of the unobserved variable to be a standard logistic distribution.

   (b) `"oprobit"`: The *ordinal probit* distribution (see Section **??**) specifies the stochastic component of the unobserved variable to be standardized normal.

   (c) `"oprobit.bayes"`: *Bayesian ordinal probit* model (see Section **??**) is similar to ordinal probit regression, but makes valid small sample inferences via draws from the exact posterior and also allows for priors.

   (d) `"factor.ord"`: *Bayesian ordered factor analysis* (see Section **??**) models observed, ordinal dependent variables as a function of latent explanatory variables.

4. **Multinomial** dependent variables are unordered, discrete categorical responses. For example, you could model an individual's choice among brands of orange juice or among candidates in an election.

   (a) `"mlogit"`: The *multinomial logistic* model (see Section **??**) specifies categorical responses distributed according to the multinomial stochastic component and logistic systematic component.

   (b) `"mlogit.bayes"`: *Bayesian multinomial logistic* regression (see Section **??**) is similar to maximum likelihood multinomial logistic regression, but makes valid small sample inferences via draws from the exact posterior and also allows for priors.

5. **Count** dependent variables are non-negative integer values, such as the number of presidential vetoes or the number of photons that hit a detector.

   (a) `"poisson"`: The *Poisson* model (see Section **??**) specifies the expected number of events that occur in a given observation period to be an exponential function of the explanatory variables. The Poisson stochastic component has the property that, $\lambda = \mathrm{E}(Y_i|X_i) = \mathrm{V}(Y_i|X_i)$.

   (b) `"poisson.bayes"`: *Bayesian Poisson* regression (see Section **??**) is similar to maximum likelihood Poisson regression, but makes valid small sample inferences via draws from the exact posterior and also allows for priors.

   (c) `"negbin"`: The *negative binomial* model (see Section **??**) has the same systematic component as the Poisson, but allows event counts to be over-dispersed, such that $\mathrm{V}(Y_i|X_i) > \mathrm{E}(Y_i|X_i)$.

6. **Continuous Bounded** dependent variables that are continuous only over a certain range, usually $(0, \infty)$. In addition, some models (exponential, lognormal, and Weibull) are also censored for values greater than some censoring point, such that the dependent variable has some units fully observed and others that are only partially observed (censored).

   (a) `"gamma"`: The *Gamma* model (see Section **??**) for positively-valued, continuous dependent variables that are fully observed (no censoring).

   (b) `"exp"`: The *exponential* model (see Section **??**) for right-censored dependent variables assumes that the hazard function is constant over time. For some variables, this may be an unrealistic assumption as subjects are more or less likely to fail the longer they have been exposed to the explanatory variables.

   (c) `"weibull"`: The *Weibull* model (see Section **??**) for right-censored dependent variables relaxes the assumption of constant hazard by including an additional scale parameter α: If $\alpha > 1$, the risk of failure increases the longer the subject has survived; if $\alpha < 1$, the risk of failure decreases the longer the subject has survived. While `zelig()` estimates α by default, you may optionally fix α at any value greater than 0. Fixing $\alpha = 1$ results in an exponential model.

   (d) `"lognorm"`: The *log-normal* model (see Section **??**) for right-censored duration dependent variables specifies the hazard function non-monotonically, with increasing hazard over part of the observation period and decreasing hazard over another.

7. **Mixed** dependent variables include models that take more than one dependent variable, where the dependent variables come from two or more of categories above. (They do not need to be of a homogeneous type.)

   (a) The *Bayesian mixed factor analysis* model, in contrast to the Bayesian factor analysis model and ordinal factor analysis model, can model both types of dependent variables as a function of latent explanatory variables.

8. **Ecological inference** models estimate unobserved internal cell values given contingency tables with observed row and column marginals.

   (a) `ei.hier`: The *hierarchical* EI model (see Section **??**) produces estimates for a cross-section of 2×2 tables.

   (b) `ei.dynamic`: *Quinn's dynamic Bayesian* EI model (see Section **??**) estimates a dynamic Bayesian model for 2×2 tables with temporal dependence across tables.

   (c) `ei.RxC`: The $R \times C$ EI model (see Section **??**) estimates a hierarchical Multinomial-Dirichlet EI model for contingency tables with more than 2 rows or columns.

1.3 Replication Procedures

A large part of any statistical analysis is documenting your work such that given the same data, anyone may replicate your results. In addition, many journals require the creation and dissemination of "replication data sets" in order that others may replicate your results (see King, 1995). Whether you wish to create replication materials for your own records, or contribute data to others as a companion to your published work, Zelig makes this process easy.

1.3.1 Saving Replication Materials

Let `mydata` be your final data set, `z.out` be your `zelig()` output, and `s.out` your `sim()` output. To save all of this in one file, type:

```
> save(mydata, z.out, s.out, file = "replication.RData")
```

This creates the file replication.RData in your working directory. You may compress this file using zip or gzip tools.

If you have run several specifications, all of these estimates may be saved in one .RData file. Even if you only created quantities of interest from one of these models, you may still save all the specifications in one file. For example:

```
> save(mydata, z.out1, z.out2, s.out, file = "replication.RData")
```

Although the .RData format can contain data sets as well as output objects, it is not the most space-efficient way of saving large data sets. In an uncompressed format, ASCII text files take up less space than data in .RData format. (When compressed, text-formatted data is still smaller than .RData-formatted data.) Thus, if you have more than 100,000 observations, you may wish to save the data set separately from the Zelig output objects. To do this, use the `write.table()` command. For example, if `mydata` is a data frame in your workspace, use `write.table(mydata, file = "mydata.tab")` to save this as a tab-delimited ASCII text file. You may specify other delimiters as well; see `help.zelig("write.table")` for options.

1.3.2 Replicating Analyses

If the data set and analyses are all saved in one .RData file, located in your working directory, you may simply type:

```
> load("replication.RData")                # Loads the replication file.
> z.rep <- repl(z.out)                     # To replicate the model only.
> s.rep <- repl(s.out)                     # To replicate the model and
                                           #  quantities of interest.
```

By default, `repl()` uses the same options used to create the original output object. Thus, if the original `s.out` object used bootstrapping with 245 simulations, the `s.rep` object will similarly have 245 bootstrapped simulations. In addition, you may use the `prev` option when replicating quantities of interest to reuse rather than recreate simulated parameters. Type `help.zelig("repl")` to view the complete list of options for `repl()`.

If the data were saved in a text file, use `read.table()` to load the data, and then replicate the analysis:

```
> dat <- read.table("mydata.tab", header = TRUE)  # Where `dat' is the same
> load("replication.RData")                       #   as the name used in
> z.rep <- repl(z.out)                            #   `z.out'.
> s.rep <- repl(s.out)
```

If you have problems loading the data, please refer to Section **??**.

Finally, you may use the `identical()` command to ensure that the replicated regression output is in every way identical to the original `zelig()` output.[4] For example:

```
> identical(z.out$coef, z.rep$coef)              # Checks the coefficients.
```

Simulated quantities of interest will vary from the original quantities if parameters are re-simulated or re-sampled. If you wish to use `identical()` to verify that the quantities of interest are identical, you may use

```
# Re-use the parameters simulated (and stored) in the original sim() output.
> s.rep <- repl(s.out, prev = s.out$par)

# Check that the expected values are identical.  You may do this for each qi.
> identical(s.out$qi$ev, s.rep$qi$ev)
```

[4]The `identical()` command checks that numeric values are identical to the maximum number of decimal places (usually 16), and also checks that the the two objects have the same class (numeric, character, integer, logical, or factor). Refer to `help(identical)` for more information.