Interpreting **Zelig**: Everyone's Statistical Software

Christine Choirat, Vito D'Orazio,
James Honaker, Muhammed Y. Idris, Jennifer McGrath

August 9, 2016

Abstract

The **Zelig** library facilities the interpretation of regression models and creates a common call structure for estimating a large number of statistical models in R. Quantities of interest, such as the expected value or the change in expected value, are simulated and visualized. R libraries for estimating statistical models are wrapped and interacted with through a common syntax. We introduce a new version of Zelig that has been written using R's Reference Classes. This simplifies the contribution of new models and diagnostics, new libraries, and new quantities of interest to Zelig. It improves replication by incorporating all necessary arguments into a single object that may then be exported. This new version of Zelig makes the generalized information matrix test available for all appropriate models, and it integrates with R libraries for multiple imputation, counterfactual analysis, and causal inference. Additional features include a help method, improved scalability through integration with **dplyr**, stochastic unit tests to ensure the reliability of included models, and the automatic creation of model-level metadata to facilitate integration with visualization tools, graphical user interfaces, and citations for contributors, among other features.

1 Introduction and history

The R statistical language is a giant open source project that spans all domains of applied statistics, visualization, and data mining. At the time of writing, R contains 7078 different code packages, most of which are written by a unique author. Among the advantages of this decentralized, dispersed organization, are the speed and depth of coverage across statistical domains with which researchers share software and tools they have developed. A drawback of this massive contribution base is that each contributed R package can often have its own definitions for how data should be structured, divided, accessed, how formulas should be expressed, and arguments named. As a result, every researcher has to learn each package's calls and notation, and possibly restructure their data, before seeing if that package has any useful application to their quantitative project.

The Zelig package for R brings together an abundance of common statistical models found across packages into a unified interface, and provides a common architecture for estimation and interpretation, as well as bridging functions to absorb increasingly more models into the collective library (Imai *et al.*, 2008). Zelig allows each individual package, for each statistical model, to be accessed by a common, uniformly structured call and set of arguments. Researchers using Zelig with their data only have to learn one notation to have access to all enveloped models. Moreover, Zelig automates all the surrounding building blocks of a statistical workflow–procedures and algorithms that may be essential to one user's application but which the original package developer perhaps did not use in their own research and thus might not themselves support. These include statistical utilities such as bootstrapping, jackknifing, matching and reweighting of data. In particular, Zelig automatically generates predicted and simulated quantities of interest (such as relative risk ratios, average treatment effects, first differences and predicted and expected values) to interpret and visualize complex models.

It also interfaces seamlessly with packages for preprocessing of data, such as matching algorithms for balanced sample selection, and multiple imputation algorithms for treatment of missing data, and with packages for postprocessing of model results, such as for forecasting and counterfactual inference.[1] Researchers who write a statistical estimator in a new package, and include the three simple bridge functions, can add all of these abilities to the new estimator they have published, without duplication of effort.

1.1 History of the Zelig Project

Imai *et al.* (2008) describe theoretically a Zelig library with five desirable features: (1) a common call structure for estimating a model and defining and simulating quantities of interest; (2) a generalization of the R formula syntax for representing statistical models; (3) a toolkit for developers to add new R libraries to Zelig; (4) a replication object for users to

[1] Many of these widely used packages, such as MatchIt and CEM (for matching), Amelia (multiple imputation), WhatIf (model dependence in counterfactuals), and YourCast (forecasting), are also R packages developed by authors at Harvard IQSS.

easily replicate an analysis; and (5) the model-level metadata necessary to facilitate Zelig's integration with a graphical user interface. While previous versions of Zelig had included these features, developers were limited by the capabilities of R. As a result, the project faced several challenges from both the user's and the developer's perspective.

In the underlying Zelig 3 code every model could have a very different architecture, meaning new maintainers had to learn all the work of everyone who had contributed to the codebase before, and new features had to be added one-by-one to old models. As a result, coverage across models was often incomplete and developers had difficulty contributing new models. Zelig 4 attempted to fix some serious issues in the Zelig 3 series by moving to a modular and object-oriented architecture for Zelig objects. The overarching goal was to make the package easier to maintain as it grew in scope, and to allow contributors to add new models without excessive Zelig expertise. We also wanted to add new capabilities and features to Zelig without having to do so piecemeal across the different models. An object-oriented design, sought to allow more structure to building models, also means contributors adding new models have to do less work, as much is generalized by the architecture. It simplifies the maintainers job by making the moving pieces more uniform.

However, when Zelig 4 was developed, the only option for an object oriented approach was by using S4 classes in R. S4 classes were an preliminary attempt for a new, more object-oriented class in R. The inherent problems in S4, particularly with regard to environments and scoping, manifested in Zelig. For example, it was very difficult for users to put Zelig calls inside of new functions. There were lots of code readability issues as Zelig in S4 relied heavily on environments. The new "Reference classes" (RCs) now in R have fixed these issues. The syntax for using RCs has conventions resembling Python and looks much more object oriented than traditional R code which has been more strongly rooted in functional programming. Under the hood, Zelig code relies heavily on these, and also on **dplyr**, which has its own unique syntax. Users who come to R from Python, or other object-oriented languages can use the Zelig objects in an object oriented fashion.

However, again we set out with the task that Zelig 5 Choirat *et al.* (2015) retain or improve the ease of use of past Zelig versions, and so users who come to Zelig with only a background in R should hopefully continue to find that Zelig is approachable, user-friendly, and accessible through a simple framework of functions. Indeed, although internally the architecture is entirely new, we worked to retain all of the user-level syntax of previous versions. All of the previous commands–zelig(), setx() and sim()–behave in the same fashion, even though there are alternate ways in the new architecture for users to get to these results using more object oriented syntax. Similarly, interpretability of statistical models through graphs, and through quantities of interest such as first differences, is still at the heart of the Zelig architecture.

In section 2 we describe the new Zelig architecture and the ways in which it is used to accomplish a software implementation of the theoretical framework described in Imai *et al.* (2008). Section 3 introduces new features, including automatically generated citations and help, scalability resulting from integration with `dplyr`, and stochastic unit tests. We detail an example model implementation and demonstrate how developers can contribute their

statistical models to Zelig in section 4. The Zelig API contains the model-level metadata necessary for a graphical user-interface and other integrations, and examples uses are included in section 5.

2 Architectural changes

The fundamental motivation behind the architectural design of Zelig 5 Choirat *et al.* (2015) is to take advantage of how statistical models are nested, how different statistical families are related, and how we often pass the same arguments and call the same functions for different models. The general idea is that features should only be implemented in the code once, and if more than one statistical model shares a feature, then that feature is *inherited* by instances of those models. At the top level of the inheritance are the features that are identical across models, including fields such as data and formula. At the bottom level of the inheritance are the features that are unique to that statistical model, such as its link function.

2.1 From S4 classes to reference classes (RC)

Packages such as `sp` (see Pebesma and Bivand (2005); Bivand *et al.* (2013)) and large projects such as Bioconductor (`www.bioconductor.org`) rely heavily on S4 classes. Zelig 5 switched to reference classes (RC) that are now built-in in R. RC are written using S4 classes, so any RC implementation could theoretically be rewritten using S4 classes only. However, as (Wickham, 2014, Chapter 7) points out:

> *RC objects are also mutable: they don't use R's usual copy-on-modify semantics, but are modified in place. This makes them harder to reason about, but allows them to solve problems that are difficult to solve with S3 or S4.*

As an illustration, consider a toy Zelig implementation of a least squares model, in which the user provides a formula and dataset name. The simplest S4 version can be written as:

```
> zelig_S4 <- setClass( "zeligS4",
                        slots = c(formula = "formula", data = "data.frame"))
> setGeneric("zelig", function(object) {standardGeneric("zelig")})
> setMethod("zelig",signature(object = "zeligS4"),
            function(object) lm(formula = object@formula, data = object@data))

> z <- zelig_S4(formula = unem ~ gdp + capmob + trade, data = macro)
> zelig(z)
```

On the other hand, a minimal RC implementation is:

```
> zelig_RC <- setRefClass("zeligRC",
                          fields = c(formula = "formula", data = "data.frame"),
                          methods = list(zelig = function()
                            lm(formula = .self$formula,data = .self$data)))

> z <- zelig_RC$new(formula = unem ~ gdp + capmob + trade, data = macro)
> z$zelig()
```

Although derived from S4 classes, both the syntax and the scoping and inheritance rules of the new RC are much closer to the type of object-oriented (OO) paradigm available in languages such as Python or Ruby. Both RC and S4 enforce *type safety*: for example a check is performed at runtime to make sure that the arguments of the `zelig` method are of type `formula` and `data.frame`. Zelig 4 was implemented with S4, but the implementation was only partial: environments and hidden global variables were used, making the codebase quite unstable.

RC are not "better" than S4 classes but they provide a standard OO paradigm, whilst S4 classes are an idiosyncratic R implementation. In our opinion, RC are more intuitive and make Zelig code easier to write, maintain, debug and extend. Also, using RC makes it trivial to switch to other OO implementations within R, for example if a future version of Zelig were to rely on R6 classes (Chang, 2015). Thanks to the design similarity of RC with other OO implementations, Zelig could be ported to other languages (Crosas *et al.*, 2015). A Python version in particular would be easy to achieve.

2.2 The new Zelig object

A typical use of Zelig consists in specifying and estimating a statistical model, setting covariates to values of interest and running simulations:

```
> data(macro)
> z.out <- zelig(unem ~ gdp + capmob + trade, data = macro, model = "ls")
> x.out <- setx(z.out)
> s.out <- sim(z.out, x.out)
```

Conceptually, there is a single, unique Zelig object `z.out` created in the previous code snippet. This object is created by a call to the `zelig` function and modified, not recreated, by calls to `setx` and `sim`. Somewhat unintuitively, the use of the assignment operator for setx() and sim() does not create a new object or overwrite an existing one. They are instead being used to populate a portion of the existing object `z.out`. This use of the assignment operator has been implemented in Zelig 5 for the purposes of backwards compatibility. Below is an example use of Zelig using the more current syntax of RC.

In Zelig 5, the Zelig object is initialized and then incrementally populated. The three core functions of the Zelig methodology are implemented as *methods*, to use OO parlance, and,

just like in previous versions of Zelig, are expected to be called sequentially. For example, to create a *least squares* model, we first initialize a Zelig object, `z.out <- zls$new()`, and then call the `zelig`, `setx` and `sim` methods:

```
> data(macro)
> z.out$zelig(unem ~ gdp + capmob + trade, data = macro)
> z.out$setx()
> z.out$sim()
```

This generates three lists inside the Zelig object: `z.out$zelig.out`, `z.out$setx.out` and `z.out$sim.out`. Unlike previous versions of Zelig, where `z.out` was copied and included into `x.out` which in turn was copied and included into `s.out`, there exists *only one* version of the object, saving memory without sacrificing computational efficiency. Furthermore, it is straightforward to keep track of the stage of construction and it therefore enables methods to have different behaviors at different stages and to provide users with information on what method should be run next:

```
> z.out <- zls$new()
> print(z.out)
Next step: Use 'zelig' method
```

Right after the object is created, the user is told to use the `zelig` method:

```
> z.out$zelig(unem ~ gdp + capmob + trade, data = macro)
> print(z.out)
Model:

Call:
stats::lm(formula = unem ~ gdp + capmob + trade, data = as.data.frame(.))

Residuals:
    Min      1Q  Median      3Q     Max
-5.3008 -2.0768 -0.3187  1.9789  7.7715

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  6.181294   0.450572  13.719  < 2e-16
gdp         -0.323601   0.062820  -5.151 4.36e-07
capmob       1.421939   0.166443   8.543 4.22e-16
trade        0.019854   0.005606   3.542 0.000452

Residual standard error: 2.746 on 346 degrees of freedom
Multiple R-squared:  0.2878,Adjusted R-squared:  0.2817
```

```
F-statistic: 46.61 on 3 and 346 DF,  p-value: < 2.2e-16

Next step: Use 'setx' method
```

Method `setx` is run:

```
> z.out$setx()
> print(z.out)
setx:
  (Intercept)   gdp  capmob trade
1           1 3.254 -0.8914 57.08

Next step: Use 'sim' method
```

and finally simulations are performed using the `sim` method:

```
> set.seed(1234)
> z.out$sim()
> print(z.out)

 sim x :
 -----
ev
      mean        sd      50%     2.5%    97.5%
1 4.990045 0.1458181 4.992802 4.714031 5.275271
pv
      mean        sd      50%     2.5%    97.5%
1 4.990045 0.1458181 4.992802 4.714031 5.275271
```

2.3 Compatibility wrappers

To maintain backward compatibility and thus ensure that Zelig 4 code still runs in Zelig 5, we provide a set of *wrappers* that mimic the syntax of Zelig 4. These wrappers are generated from a JSON file in the Zelig R package automatically created when the package is built. Indeed, every model available is Zelig 5 has a `wrapper` field which gets populated upon object initialization:

```
> z.out <- zls$new()
> print(z.out$wrapper)
[1] "ls"
```

So when Zelig 5 is used as:

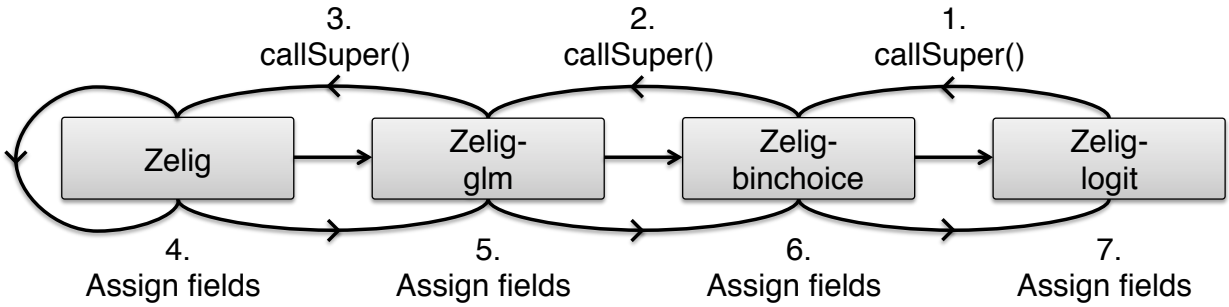


Figure 1: *Inheritance and statistical models.*

```
> z.out <- zelig(unem ~ gdp + capmob + trade, data = macro, model = "ls")
```

the `model` wrapper argument allows the `zelig` call to be unambiguously translated to creating and estimating a `zls` object. The JSON file also contains a description of the model, the vignette URL and information about class hierarchy (see Section 5).

2.4 Inheritance structure

Reference classes allows Zelig to take advantage of how different statistical families are related. For example, in the setting of Generalized Linear Models (GLM), binomial data is often modeled with a logit or a probit. These two models only differ by their link functions, respectively $\ln(p/(1-p))$ and $\Phi^{-}1(p)$, where p is the parameter of the Bernoulli distribution.

The Zelig implementation takes advantage of how these statistical models are naturally nested, as shown in figure 1. We define a `Zelig-glm` class that derives from the main `Zelig` object via a *single inheritance* mechanism:

```
zglm <- setRefClass("Zelig-glm",
                    contains = "Zelig",
                    fields = list(family = "character",
                                  link = "character",
                                  linkinv = "function"))
```

We then define a `Zelig-binchoice` class, where we specify that the `family` field is "binomial" and calculate the quantities of interest in a virtual way, that is without the need to define a link function:

```
zbinchoice <- setRefClass("Zelig-binchoice",
                          contains = "Zelig-glm")
```

At this point, we only need to specify the link functions ("logit" and "probit") to implement the full logit and probit models.

```
zlogit <- setRefClass("Zelig-logit",
                      contains = "Zelig-binchoice")
zlogit$methods(
  initialize = function() {
    callSuper()
    .self$link <- "logit"
    [...]
  }
)
```

Models such as "logit GEE" or "logit Bayes" rely on *multiple inheritance*:

```
zlogitbayes <- setRefClass("Zelig-logit-bayes",
                           contains = c("Zelig-bayes",
                                        "Zelig-logit"))
```

Estimation and simulations are performed using `Zelig-bayes`, whilst quantities of interest are taken from `Zelig-logit`. Multiple inheritance avoids duplicating code and is therefore a less error-prone software development strategy.

Objects, user-defined or belonging to other R packages, can be derived from main Zelig class. The ZeligChoice package (https://github.com/IQSS/ZeligChoice) is an illustration of this inter-package inheritance.

2.5 Quantities of interest

Quantities of interest available across all statistical models in Zelig include predicted values, expected values, and first differences. These are calculated using the algorithm described in King *et al.* (2000). Algorithm 1 describes simulating predicted values, and algorithm 2 describes simulating expected values. First differences are estimated as the difference in two expected values, simulated at different chosen values of the independent variables.

Steps 1, 2, and 3 are identical across statistical models, and are therefore handled at the top level of inheritance. With respect to steps 1 and 2, model estimates are always placed in the same location inside the Zelig object, so they are consistently retrievable. Once retrieved, it is straightforward to simulate the parameters. The same applies to the chosen values of the independent variables, which are set using the setx() method. The model-specific components in these algorithms are the functions $f(\cdot)$ and $g(\cdot)$, representing the systematic and stochastic components. These functions are specified in the qi() method in the bottom level of inheritance.

Algorithm 1: Simulating Predicted Values

1 Gather estimated model parameters $\gamma = vec(\beta, \alpha)$.
2 Draw simulations $\tilde{\gamma} \sim N(\hat{\gamma}, V(\hat{\gamma}))$.
3 Set independent variables at selected values x_c.
4 Compute systematic parameter $\tilde{\theta} = g(x_c, \hat{\beta})$.
5 Simulate predicted value $\tilde{y}_c \sim f(\tilde{\alpha}, \tilde{\theta})$.
6 Repeat steps 2 through 5, m times.

Algorithm 2: Simulating Predicted Values

1 Gather estimated model parameters $\gamma = vec(\beta, \alpha)$.
2 Draw simulations $\tilde{\gamma} \sim N(\hat{\gamma}, V(\hat{\gamma}))$.
3 Set independent variables at selected values x_c.
4 Compute systematic parameter $\tilde{\theta} = g(x_c, \hat{\beta})$.
5 Simulate predicted value $\tilde{y}_c \sim f(\tilde{\alpha}, \tilde{\theta})$.
6 Repeat step 5, k times, and average $\tilde{E}(\tilde{y}_c|\tilde{\gamma}) = \sum_{i=1}^{k} \tilde{y}_c^{(i)}/k$
7 Repeat steps 2 through 6, m times.

2.6 An integrated workflow for analysis

Pre-processing and post-processing steps are often important or necessary additions to a statistical estimator to construct meaningful quantitative reasoning: missing data is endemic in observational social science, and generally needs to be corrected for to avoid bias and inefficiency in statistical estimates (Schafer, 1997; King *et al.*, 2001; Honaker and King, 2010); failing to account for known measurement error leads to attenuation in the simplest cases, and unpredictable bias in many others (Blackwell *et al.*, 2016a,b); matching as a preprocessing step reduces model dependence (Ho *et al.*, 2007) and is often central to causal reasoning (Stuart, 2010); counterfactual inference becomes increasingly fraught and model dependent as it reaches the support of the data (King and Zeng, 2006, 2007).

In order to facilitate ease of the statistical workflow, Zelig accepts the output objects of many packages for preprocessing, in the place of datasets, seamlessly in the `data` argument, and can include the required output objects for postprocessing packages in the Zelig object. These include the **Amelia** package for missing data and measurement error Honaker *et al.* (2011), the **MatchIt** Ho *et al.* (2011) and **CEM** Iacus *et al.* (2009) packages for constructing matched datasets, and the **WhatIf** package for counterfactual reasoning Stoll *et al.* (2005).

3 New features

3.1 Citation methods

While Zelig brings many novel methods for model interpretation, it relies foundationally on the work of the community of R package authors who construct models, and those who bridge those packages into the Zelig architecture, as well as the packages that provide a large number of pre- and post- processing steps in a statistical workflow that integrate with Zelig (see section 2.6). Giving proper credit to all the constituent contributions is important to any open source project. Also, in statistical software there can too easily be a disjuncture between the academic literature that describes statistical or algorithmic theory, and the codebase that implements these developments; it is centrally important to provide information to the user as to where to learn more about the computational methods the user has employed in their work.

To bring important focus to these issues, Zelig has built a novel architecture for tracking a workflow and accumulating citation and reference information by means of `citation()` and `reference()` methods available to every Zelig object.

As we describe, many of these references are constructed by package authors themselves, however, some items have complex layers of contribution. We work under the analogy that calls to Zelig models are like items in an edited volume, and cite them using such conventions. Zelig is the collected volume, while the contributors who bridge functions may be subeditors who edit an article for integration into a collected volume, and the wrapped package represents the original primordial work.

The citation architecture has several new features for automatically constructing citations for Zelig models and constructing lists of references to describe a Zelig workflow:

1. **Model Citations:** Citations for contributing authors to individual Zelig models are handled through Zelig's `cite()` method. Each individual Zelig model object has fields designated for citation information in the Zelig Reference Class. Examples of such fields include author and year. When a model is added, these fields are inherited into the RC and populated by the author. This provides the benefit of declaring relevant fields and how those fields are to be formatted into a citation at the top-level of the inheritance. As a result, model authors only need to specify their citation fields. This allows for a consistent and reliable method of citation across all contributed models.

   When a model is run, a cite for that model is constructed and printed to screen. This follows the ordering previously discussed (Zelig Author -if no- Package Author -if no- Model (wrapper) Author) within the edited volume analogy. These cites now appear in the docs on zeligproject.org for every example. Below is a screenshot when a Model Author (wrapper) exists for a model:

   ```
   > zam.out <-zelig(yb ~ xx + zz, data = data, model= "normal.bayes",
   verbose = FALSE)
   How to cite this model in Zelig:
   ```

```
.   Ben Goodrich, and Ying Lu.  2013.
.   normal-bayes:  Bayesian Normal Linear Regression
in Christine Choirat, James Honaker, Kosuke Imai, Gary King, and Olivia Lau,
.   "Zelig:  Everyone's Statistical Software," http://zeligproject.org/
```

And here is a screen shot when there is no Model Author which defaults to the author of the original package:

```
> zam.out <-zelig(yb ~ xx + zz, data = data, model= "negbin",
verbose = FALSE)
How to cite this model in Zelig:
.   William N. Venables, and Brian D. Ripley.  2008.
.   negbin:  Negative Binomial Regression for Event Count Dependent Variables
in Christine Choirat, James Honaker, Kosuke Imai, Gary King, and Olivia Lau,
.   "Zelig:  Everyone's Statistical Software," http://zeligproject.org/
```

2. **Automatic Reference Sections:** As a new feature, as a workflow propagates through a Zelig object, the Zelig object builds up an automatic reference list automatically, that the user can ask for at any time. For example, if their dataset is an imputed Amelia object, then they get an Amelia cite. If they use a model in Zelig, then Zelig attaches cites to that model (for example, there are two literature cites (King and Zeng, 2001a,b) that explain the Relogit model if that is the model used in Zelig) as well as automatically creating a cite to any package we wrapped to implement the model (Like the **Survival** package we use for Weibull models). If they then run a GIM test, then it attaches a cite to King and Roberts King and Roberts (2015) explaining the assumptions of that test diagnostic. And so it can continue. When Zelig is finished running, you have simultaneously created a list of all the citations you might want to the pieces and procedures you used, and can ask for this with a utility method. As an example, in the project page documentation, every single model page always has a short reference section, and none of these have been written by hand. All of these are automatically constructed by Zelig, via a call to the citation method on the zelig object that resulted from the explanatory example in the documentation. Here's an example:

```
> zn.out$references()
Honaker J, King G and Blackwell M (2011).  "Amelia II: A Program for
Missing Data." *Journal of Statistical Software*, **45** (7), pp.  1-47.
<URL:http://www.jstatsoft.org/v45/i07/>.

King G and Roberts M (2014).  "How Robust Standard Errors Expose
Methodological Problems They Do Not Fix, and What to Do About It."
Political Analysis*, pp.  1-21.  <URL: http://j.mp/InK5jU>.
```

```
R Core Team (2014) *R: A Language and Environment for Statistical
Computing*.  R Foundation for Statistical Computing, Vienna, Austria.
<URL:http://www.R-project.org/>.
```

Package authors can declare a CITATION file in their package with the ways they would like their package cited, including both a citation to the R package and to related journal articles. If that field exists we use that to credit package authors in the way they most prefer, otherwise we build our own citation from the meta data available in that package's DESCRIPTION file. In this fashion, most of the references Zelig can construct do not have to be collected or coded by hand, or kept up to date. They are automatically generated from the wrapped package files, and kept up to date, in a distributed fashion, by the original package authors.

3. **Zelig CITATION file:** Relatedly, Zelig now has such a CITATION file, if any other author was to use one of the utilities in R that exist to ask Zelig for its own preferred citation. This is distinct from the model citation created above in item 1. If someone queries the Zelig package for citations, presently, it reads:

```
To cite Zelig in publications please use:

Choirat C, Honaker J, Imai K, King G and Lau O (2015).  *Zelig:  Everyone's
Statistical Software*.  Version 5.0-3,
<URL: http://zeligproject.org/>

Imai K, King G and Lau O (2008).  "Toward A Common Framework for Statistical
Analysis and Development." *Journal of Computational Graphics and Statistics*,
*17*(4), pp.  892-913.
<URL: http://j.mp.msE15c>.
```

3.2 Help methods

Zelig's help() method works in a similar way. Every model has an associated vignette available on zeligproject.org. Call the help() method opens this URL in the user's browser. This allows fast access to help files and also the ability to view the help files in the user's browser while continuing work in R. Defaults for the URL are specified at the top level of the inheritance. For example, zeligproject.org is the default URL. However, if not models are added the contributor has the option to overwrite the default URLs by populating the URL fields further down the inheritance.

3.3 Architecture for Weighting

Weights are often added to statistical models to adjust the observed sample distribution in the data to an underlying population of interest. For example, some types of observations may have been intentionally oversampled, and need to be downweighted for population inferences, or weights may have been created by a matching procedure to create a dataset with treatment and control groups that resemble randomized designs and achieve balance in covariates.

Weights can now be added to any and every model in Zelig. The weights argument, can be a vector of weight values, or a name of a variable in the dataset.

Not all the R implementations of statistical models that Zelig uses have been written to accept weights or use them in estimation. When weights have been supplied by the user, but weights are not written into the package for that model, Zelig is still able to use the weights by one of two procedures:

- If the supplied weights are all integer values, then Zelig rebuilds a new version of the dataset by duplicating observations according to their weight (and removing observations with zero weight).
- If the weights are continuously valued, Zelig bootstraps the supplied dataset, using the relative weights as bootstrap probabilities.

3.4 Scalability with `dplyr` package

Zelig leverages two important features of `dplyr` (Wickham and Francois, 2015): (1) it is data-source agnostic, and can work with a local in-memory data frame as well as a connection to a SQL database server; and (2) it provides highly-efficient grouped operations. Grouped operations are a very strong benefit of using `dplyr`. Previous versions of Zelig were striving to have a `by` argument to perform model analysis along categorical variables. The behavior of the `by` function in base R

```
by(warpbreaks, warpbreaks[,"tension"],
   function(x) lm(breaks ~ wool, data = x))
```

can be simplified with Zelig and does not require defining an anonymous function

```
z.out <- zelig(formula = breaks ~ wool, data = warpbreaks,
               model = "ls", by = "tension")
```

Besides allowing for running zelig along categorical variables, the dplyr-based `by` argument is flexible enough to be instrumental in combining Amelia's imputed datasets (Honaker *et al.*, 2011) and creating weighting schemes, such as those used in the bootstrap or in survey models.

3.5 Unit tests for stochastic functions

Another new feature is an automated unit testing framework that assesses the accuracy and reliability of statistical models ported in Zelig. The fundamental issue in testing statistical models is to extract a property of an estimator (e.g., bias or efficiency) and determine a way to test for that property *without rewriting the estimator*. One way of doing this is to specify an estimator's data generating process and simulating a true quantity of interest which can be used to consider different properties of an implemented model. We begin with the premise that it is generally desirable for statistical estimators to be unbiased and we test for this property using expected values and the simulations-based approach described in King *et al.* (2000).

Unit testing, and regression testing more broadly, are important components in large-scale team-sourced software projects to ensure that as dependencies change, or as features are continually adapted, no underlying functionality is unwittingly broken. Unit tests typically consist of small functionality tests of key modules of the code which should always give a known, verifiable output from a specified test set of inputs. Unit testing is a type of regression testing, but regression testing also includes broader functional performance tests of the entire system. In a software platform like Zelig, many of the outputs are themselves stochastic, and thus vary in output even given the same inputs[2]. This makes unit testing of stochastic modules such as statistical estimators a challenging task.

To guarantee that as the packages Zelig depends on are revised, and as contributors make new additions to Zelig functionality, the underlying statistical results can be continually verified we developed a testing framework, based on the Monte Carlo methods commonly employed by researchers to test unbiasedness of newly developed statistical estimators. Our testing framework is based on a novel algorithm for unit testing stochastic model and is designed to (1) ensure that implementations of statistical estimators are correct (unit testing), (2) ensure that dependencies have been ported correctly (module testing), and (3) ensure that changes in dependency package do not invalidate the implementations of statistical and analytical methods (regression testing). The algorithm leverages Monte Carlo simulations and compares the distribution of simulated quantities of interest to true quantities of interest. Algorithm 3 describes the algorithm implemented in our method.

To begin, we calculate y^*, the true expected values, using the data generating process supplied. These expected values are akin to a gold standard to which simulated expected values from $Z(\cdot)$ are compared. To calculate y^*, we set out model's coefficient estimates, β, and any other model parameters, α. We specify the range of covariate values, $[a, b]$, over which we will calculate our expected values. We then construct a sequence, $\vec{x}$, over that range, and calculate θ, the systematic component of our model. We can then calculate $y*$

[2]Setting seeds for pseudorandom number generators can ensure the exact same answers are produced on repeated calls to the same codebase, but small changes in the codebase that do not introduce any errors, such as the order in which different random draws are made, or their method of storage, or even updates to the underlying operating system, can trigger stochastic differences from the same random number seeds while not being caused by any underlying error or fault in the codebase.

Algorithm 3: Algorithm for Monte Carlo Unit Tests

1. There exists an estimator, $Z(\cdot)$, and a data generating process specified by $f(\theta, \alpha)$ and $\theta = g(x, \beta)$.
2. Set model parameters β, α and covariate range a, b.
3. Construct a sequence, $\vec{x}$, over [a,b].
4. Calculate $\theta = g(\vec{x}, \beta)$.
5. Calculate $y^* = E(y|\vec{x}, \beta, \alpha) =$ mean of m draws from $f(\theta, \alpha)$.
6. Simulate $\tilde{x} \sim U(a, b)$.
7. Calculate $\theta = g(\tilde{x}, \beta)$.
8. Simulate $\tilde{y} \sim f(\theta, \alpha)$.
9. Estimate $\hat{Z}(\tilde{x}, \tilde{y})$.
10. Draw $\hat{\beta}, \hat{\alpha}$ from $N(\beta_Z, \alpha_Z)$.
11. Estimate $\hat{y} = E(y|\vec{x}, \hat{\beta}, \hat{\alpha})$.
12. Repeat steps 10 and 11 some large number (perhaps 1,000) of times to estimate a distribution of expected values.

by taking the mean of m draws from $f(\theta, \alpha)$, where an m of 1,000 is sufficient.

At this point we know our true expected value, but we still need a data set on which $Z(\cdot)$ estimates a distribution of expected values to compare against. In our implementation, we found that the appropriate number of observations for this data set should be an N no less than 1,000. Any N greater than 1,000 leads to comparable results and any N lower than 1,000 leads to inconsistent results. To construct such a data set, we draw $\tilde{x}$ from a Uniform distribution with start and end arguments $[a, b]$. We then calculate θ and draw $\tilde{y}$ from $f(\theta, \alpha)$. We now have a data set for which we know the true expected value for this statistical model.

Next, we estimate $\hat{Z}(\tilde{x}, \tilde{y})$, retrieve β_Z and α_Z, and draw $\hat{\beta}$ and $\hat{\alpha}$ from a Normal $N(\beta_Z, \alpha_Z)$. We may now estimate $\hat{y} = E(y|\vec{x}, \hat{\beta}, \hat{\alpha})$ and calculate our test statistical comparing $y*$ and $\hat{y}$. We then repeat the process of simulating a data set, estimating $Z(\cdot)$, and estimating $\hat{y}$ k times where k of 1,000 is likely sufficient. This produces a distribution of simulated expected values to which we visually compare to true predicted values.

Figures 2 and 3 presents such a visual comparison with $E(Y|X)$ in blue with confidence interval and $\hat{y}^*$ in red for a least squares and tobit estimator across a regular sequence of simulated values of X between 0 and 1. A cursory examination of both plots indicates a strong correlation between two quantities of interest suggesting that the both estimators are asymptotically unbiased and working properly.

This Monte Carlo-based approach to unit testing assesses the extent to which implementations of maximum likelihood estimators are asymptotically unbiased. An asymptotically unbiased estimator should (on average) provide the correct answer if implemented correctly. The algorithm described above, provides a way for testing for this property by considering whether an estimator's expected values line up with its functional form (e.g., mean predic-

Figure 2: Least Squares Regression Unit Test,
$\beta_0 = 0, \beta_1 = 2$

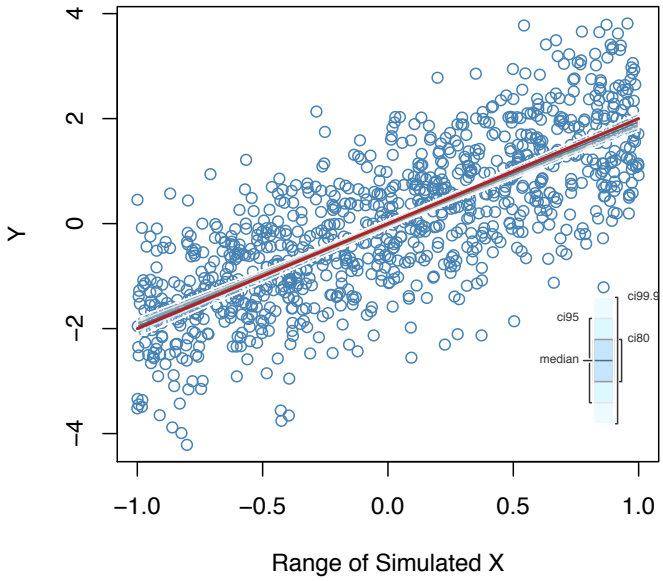
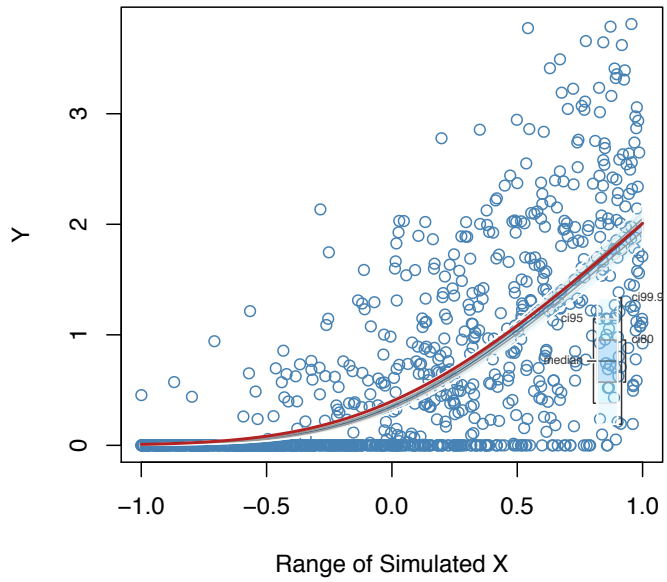


Figure 3: Tobit Regression Unit Test,
$\beta_0 = 0, \beta_1 = 2$



tion). If the estimator is correctly specified, simulations of expected values should agree with the functional form described by a correct data generating process. This data generating

process tell us analytically where the center of the functional form is.

Because this is true for the majority of maximum likelihood estimators, a passed test tells us that the estimator is asymptotically unbiased. Because this is one of (if not the most) desirable property of an estimator for most applications, we interpret a successful result to be a correctly specified model. It is important to note, however, what a passed test does not tell us. It does not provide any information about an estimator's other properties (e.g., efficiency) and because little work has been done on testing for these other properties, we do not consider them there.

A failed unit test signals a mismatch between the expected values simulated from the estimator and its data generating process. This suggests that either the implementation of the estimator or the data generating process is wrong. If the analyst believes that it is easier to write the dgp (and is therefore more confident that it is correctly specified) than the estimator, it is more likely that the estimator is wrong. In this way, the test is like a fire alarm that can indicate that something is wrong without specifying exactly what the problem is. To the extent that the analyst knows the dgp to be correct, this fire alarm is interpreted as the code is wrong and in this way is a unit test. To be sure, it is possible that a incorrectly specified estimator passes the test, but this is reasonably unlikely because both the dgp and the estimator have to be wrong in exactly the same way.

4 Example: Implementing the Poisson model

The Poisson model is a discrete count model with stochastic component $f(y_i|\lambda) = \frac{exp(-\lambda)\lambda^{y_i}}{y_i!}$ and systematic component $\lambda = exp(x_i\beta)$ (King, 1989). It is implemented in Zelig using the **glm** library. In this section we use the Poisson to demonstrate how to add a model to Zelig.

The `zelig()` method in Zelig estimates a statistical model by wrapping an existing function for estimating that model. For the Poisson, this is **glm()**. Specifically, using the `glm` function and the "sanction" data in Zelig, a Poisson model may be estimated as:

```
data(sanction)
fit <- glm(formula = num ~ target + coop,
     family=poisson(link="log"), data=sanction)
```

From the developer's perspective, what we are actually doing when we add a model to Zelig is extending the inheritance tree by adding a new RC. Looking at Zelig's inheritance structure, we can see that there exists a Zelig-glm RC that inherits from a Zelig RC. We want to extend this by adding a Zelig-poisson RC that inherits from Zelig-glm, and therefore also from Zelig. We do this in a new R script by:

```
zpoisson <- setRefClass("Zelig-poisson",
     contains = "Zelig-glm", fields = list(theta = "ANY"))
```

Through the contains argument we have inherited the fields and methods of Zelig-glm, and through the fields argument we have added a list of fields to Zelig-poisson, which is here just one field named "theta" whose class is "ANY". Next, we write our method for specifying the required fields to *initialize* Zelig-poisson. There are four required arguments for estimating a poisson regression: `formula`, `family`, `link`, and `data`. The `formula` and `data` fields are required of all models in Zelig and are therefore top-level fields defined by methods in the Zelig RC. This is the value of inheritance: to add the poisson we are not concerned with these top-level fields. The `family` and `link` fields, however, must be specified because they are particular to the poisson.

The `family` and `link` fields are inherited through Zelig-glm, where they are created and restricted to the character class, but not populated. For the poisson, `family` is "poisson" and `link` is "log", and so we specify this in the `initialize` method as:

```
zpoisson$methods(
  initialize = function() {
    callSuper()
    .self$family <- "poisson"
    .self$link <- "log"
  }
)
```

Thus, when any object that inherits from Zelig-glm is instantiated, the `family` and `link` fields are passed to that object and then populated by the `initialize()` method. This is all this is required to estimate a poisson regression using Zelig:

```
z5 <- zpoisson$new()
z5$zelig(num ~ target + coop, data=sanction)
z5$setx()
```

The Zelig method `setx()` is also a top-level method that will not be of concern to many developers. When `setx()` may be of concern is in the case of multiple equations or models with parameters outside of the formula equation that are of interest to set to specific values. For the poisson, this is not the case and the command `z5$setx()` may be used to set default covariate values.

At this point we have extended the Zelig inheritance and successfully wrapped the poisson model from **glm**, but we have not yet populated the fields necessary to simulate quantities of interest. First, we populate the `linkinv` field that is inherited from Zelig-glm. This field is restricted to type function, and for the poisson regression it is equivalent to the systematic component $exp(x_i\beta)$. We may assign this function to `linkinv`, or more easily we may use the one that `glm` is already using by setting `linkinv` to `eval(call(.self$family, .self$link))$linkinv`. Note that `family` is "poisson" and `link` is "log". Equivalently, we can also write: `.self$linkinv <- function(m){exp(m)}`.

Finally, for any contributed model the quantities of interest are specified in its `qi` method. For the poisson, we will just return a list containing the predicted values and the expected values. To do so, we require two arguments: (1) draws of the model's estimated parameters and (2) the covariate values specified by `setx()`. Each of these arguments are higher-level objects that may be passed to `qi()`.[3] Let `simparam` be a matrix of draws for every parameter, and `mm` be the value at which we set our independent variables.

```
zpoisson$methods(
  qi = function(simparam, mm) {
    eta <- simparam %*% t(mm)
    theta.local <- matrix(.self$linkinv(eta), nrow = nrow(simparam))
    ev <- theta.local
    pv <- matrix(NA, nrow = nrow(theta.local), ncol = ncol(theta.local))
    for (i in 1:ncol(theta.local))
      pv[, i] <- rpois(nrow(theta.local), lambda = theta.local[, i])
    return(list(ev = ev, pv = pv))
  }
)
```

The remainder of `qi()` is particular to the poisson model but follows the simulation procedure described in King *et al.* (2000). Specifically, we multiply the drawn parameters (`simparam`) and the covariate values (`mm`). We then apply the inverse link to each row in the resulting matrix. This produces a one-column matrix with a number of rows equal to the number of times we drew from the distribution of model parameters; each cell is our λ for that simulation. To estimate the predicted values we draw from the Poisson distribution using these values of λ. For the poisson, the expected value is simply λ.[4] Our expected values and predicted values are returned as a list that may be later used for visualizations.

5 The Zelig API

The Zelig API is automatically constructed by writing metadata contained inside the Zelig object to a json file. The metadata describes the model and the assumptions on explanatory and dependent variables, provide citations for contributing authors, URLs for documentation and other help files, the inheritance structure, test statistics associated with the model, and additional information. This API facilitates the integration of Zelig with graphical user interfaces, visualization tools, and databases, among other integrations. Its purpose is to provide machine readable information about the contents of the Zelig library and the models it contains. Thus, as Zelig expands, any user interface, documentation, or library description

[3] The covariate values are chosen using `setx()`, and the parameter values are drawn in the `sim()` function in the Zelig RC.

[4] The full script is included in Zelig/R as `model-poisson.R`. Here you will see additional fields that have been populated, include things like `authors` and `description`.

may be automatically updated without the owner having to learn about the underlying changes to Zelig and manually change their code or description to reflect these changes.

The Zelig project website and the TwoRavens tool for statistical analysis are two examples of projects that utilize the Zelig API.[5]

5.1 The Zelig Project Website

One of the difficulties with maintaining project websites is *maintaining* project websites. That is, resources have to be devoted to ensuring the website and the software project are in agreement with one another. Such resources are often scarce, and even when they are available this problem becomes especially difficult when users are encouraged to contribute to the software, as is the case with Zelig. Even when there is a platform such as Github through which there is a clearly marked trail of contributions, it is often the case that different individuals manage the software and the website. This problem is exacerbated when third-parties are also describing and disseminating the software.

The machine readable JSON simplifies the maintenance of the Zelig project website (`zeligproject.org`). Among its features, this site allows users to browse all models included in the Zelig library and explore Zelig's inheritance structure through a visualization created using D3. A subset of this can be see in figure 4. This figure shows, for example, that `logit` inherits from `binchoice`, which inherits from `glm`. As seen in the full visualization, `glm` is included in the Core distribution and inherits from `zelig`.

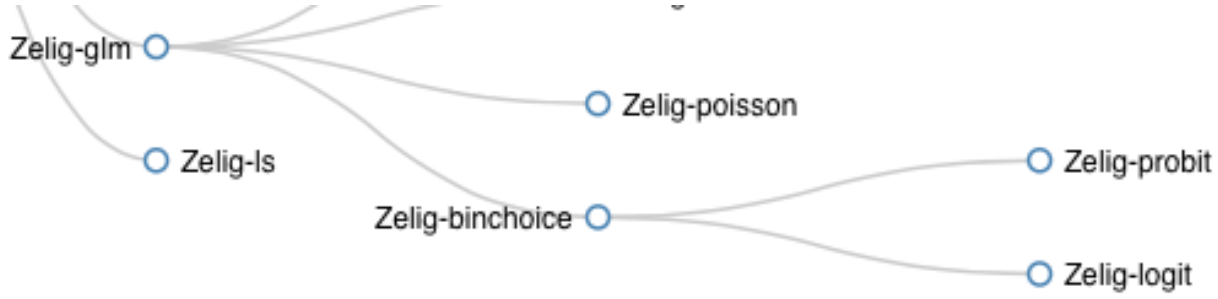


Figure 4: *A subset of the Zelig Inheritance Tree.*

The visualization in figure 4 is constructed dynamically from the Zelig API and is automatically revised if the inheritance structure of Zelig is changed in any way. The terminal nodes on the tree contain the names of the statistical model (e.g., `logit`) and, when clicked, redirect the user to the URL of the help file for that particular model. Again, the URLs are defined dynamically from the API, so a change to the Zelig code base does not require a change to the project site.

[5]See `zeligproject.org` and `github.com/IQSS/TwoRavens` for additional information about these projects.

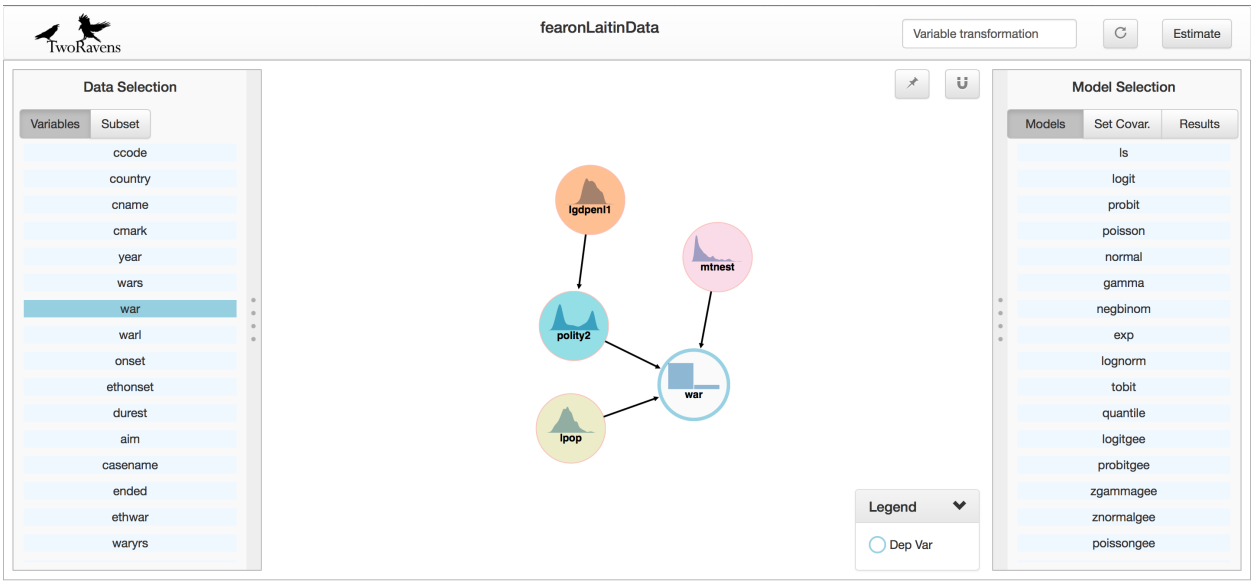


Figure 5: *The TwoRavens interface.*

In short, producing machine readable metadata in the R library itself simplifies the maintenance of the project site and helps to ensure the software and the site contain the same information. It also provides third-party sites and graphical interfaces access to the contents of Zelig, as is the case with TwoRavens.

5.2 The TwoRavens Interface

Zelig's consistent syntax across statistical models, easily retrievable statistical estimates, and API all enable a simple integration with graphical user interfaces such as TwoRavens.

TwoRavens is a gesture-based, Web application for statistical analysis (Honaker and D'Orazio, 2014; D'Orazio and Honaker, 2016). It integrates with data repositories for access to data and statistical software for computation. The interface is designed to be a thin, client-side application that reads only metadata and is usable by researchers with little knowledge of statistical software. The metadata that TwoRavens reads contains information about the data, as well as metadata about the statistical tools available to the user.

An example of the interface being used with data from Fearon and Laitin (2003) is shown in figure 5. Users make their data selections in the left panel, specify relationships and tag variables with certain properties in the center panel, and select their statistical model and view results in the right panel.

The right panel is primarily where the Zelig metadata are used and where statistical estimates are displayed. It contains three tabs: Models, Set Covariates, and Results. The Models tab lists all available models contained in the metadata. On mouseover, users can see brief descriptions of these models, as shown in figure 6 for the negative binomial model.

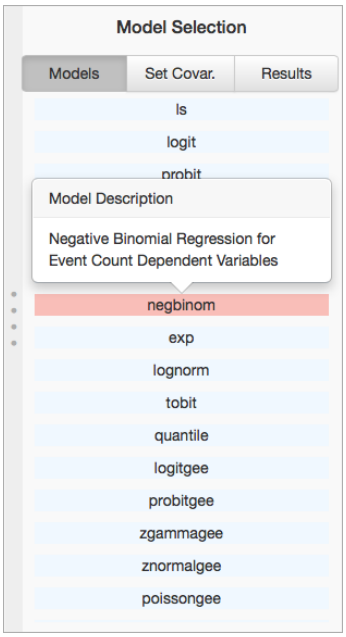


Figure 6: *The Models tab.*

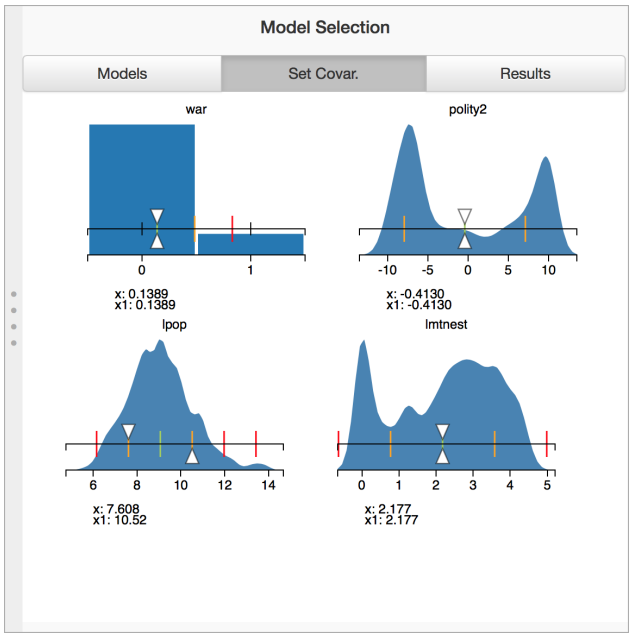


Figure 7: *The Set Covariates tab.*

The Set Covariates tab, shown in figure 7, enables users to select values of the independent variables, as with Zelig's `setx()`.

After the data has been selected, the relationships that the user wishes to model have been specified, and a dependent variable and statistical model have been selected, TwoRavens sends this information to a server for estimation. The consistency of the Zelig syntax means that regardless of which model is chosen, the call to Zelig is essentially the same. The model name will change, as will the formula representation, but those are inputs to a consistent list of arguments required to estimate the model. In the absence of this consistency, TwoRavens would need to construct the R commands differently for every model.

Upon estimation, TwoRavens retrieves the statistical estimates and any visualizations produced by Zelig and displays that information to the user. Regardless of the model, coefficients can be pulled using the `getcoef()` method. This is a simple, but important feature as it is not always the case across R libraries that these estimates are in the same location or that they exist within the object at all. For example, the estimates might only be available through the summary function.

To facilitate interpretability, Zelig provides users with visualizations of quantities of interest. These graphics are uploaded to TwoRavens and displayed alongside the statistical estimates. Although it is increasingly common for developers to provide graphical features for users to visualize their statistical results, there is very little consistency in the syntax for doing so. However, for TwoRavens the graphics are uploaded as-is to the user's browser. Thus, as long as the visualization is produced through Zelig's `plot()` method, the visualization may be uploaded directly to the user's browser for interpretation.

The Zelig API, as well as its consistency in syntax for estimating models and retrieving results, enables interfaces such as TwoRavens to easily integrate with Zelig.

6 Conclusion

Zelig 5 has been designed to take advantage of how statistical models are nested, how different statistical families are related, and how we often pass the same arguments and call the same functions for different models. R's reference classes, through their ability to handle inheritance, enabled us to do this. As much as is possible, features are implemented in the code once and are inherited by instances of models. This facilitates the addition of new models, as well as new features across existing models. Examples include citation and help methods, and stochastic unit tests. Everything necessary to replicate an estimation is stored in the Zelig object. As a result, replication is a simple as saving and loading a single object. Zelig 5 contains an API that may be used for integrating Zelig with other software applications, including graphical user interfaces.

References

Bivand RS, Pebesma E, Gomez-Rubio V (2013). *Applied spatial data analysis with R, Second edition*. Springer, NY. URL http://www.asdar-book.org/.

Blackwell M, Honaker J, King G (2016a). "A Unified Approach to Measurement Error and Missing Data: Details and Extensions." *Sociological Methods and Research*, p. forthcoming. doi:10.1177/0049124115589052. URL http://j.mp/1BQAvH9.

Blackwell M, Honaker J, King G (2016b). "A Unified Approach to Measurement Error and Missing Data: Overview and Applications." *Sociological Methods and Research*, p. forthcoming. doi:10.1177/0049124115585360. URL http://j.mp/1KbrWKX.

Chang W (2015). *R6: Classes with Reference Semantics*. R package version 2.1.1, URL http://CRAN.R-project.org/package=R6.

Choirat C, Honaker J, Imai K, King G, Lau O (2015). "Zelig: Everyone's Statistical Software, Version 5." URL http://ZeligProject.org.

Crosas M, King G, Honaker J, Sweeney L (2015). "Automating Open Science for Big Data." *The ANNALS of the American Academy of Political and Social Science*, **659**(1), 260–273.

D'Orazio V, Honaker J (2016). *A User Guide to TwoRavens: An overview of features and capabilities*. URL http://2ra.vn/papers/tworavens-guide.pdf.

Fearon JD, Laitin DD (2003). "Ethnicity, insurgency, and civil war." *American political science review*, **97**(01), 75–90.

Ho D, Imai K, King G, Stuart E (2007). "Matching as Nonparametric Preprocessing for Reducing Model Dependence in Parametric Causal Inference." *Political Analysis*, **15**, 199–236. URL http://gking.harvard.edu/files/abs/matchp-abs.shtml.

Ho DE, Imai K, King G, Stuart EA (2011). "MatchIt: Nonparametric Preprocessing for Parametric Causal Inference." *Journal of Statistical Software*, **42**(8), 1–28. URL http://www.jstatsoft.org/v42/i08/.

Honaker J, D'Orazio V (2014). "Statistical Modeling by Gesture: A Graphical, Browser-based Statistical Interface for Data Repositories." In *Extended Proceedings of the 25th ACM Conference on Hypertext and Social Media*, volume 1210 of *DataWiz 2014*. ACM. ISSN 1613-0073. URL http://ceur-ws.org/Vol-1210/datawiz2014_05.pdf.

Honaker J, King G (2010). "What to do About Missing Values in Time Series Cross-Section Data." *American Journal of Political Science*, **54**(2), 561–581. URL http://gking.harvard.edu/files/abs/pr-abs.shtml.

Honaker J, King G, Blackwell M (2011). "Amelia II: A Program for Missing Data." *Journal of Statistical Software*, **45**(7), 1–47. URL http://www.jstatsoft.org/v45/i07/.

Iacus SM, King G, Porro G (2009). "CEM: Coarsened Exact Matching Software." *Journal of statistical Software*, **30**. URL http://gking.harvard.edu/cem.

Imai K, King G, Lau O (2008). "Toward A Common Framework for Statistical Analysis and Development." *Journal of Computational Graphics and Statistics*, **17**(4), 1–22. URL http://gking.harvard.edu/files/abs/z-abs.shtml.

King G (1989). *Unifying political methodology: The likelihood theory of statistical inference*. University of Michigan Press.

King G, Honaker J, Joseph A, Scheve K (2001). "Analyzing Incomplete Political Science Data: An Alternative Algorithm for Multiple Imputation." *American Political Science Review*, **95**(1), 49–69. URL http://gking.harvard.edu/files/abs/evil-abs.shtml.

King G, Roberts ME (2015). "How Robust Standard Errors Expose Methodological Problems They Do Not Fix, and What to Do About It." *Political Analysis*, **23**(2), 159–179. URL http://j.mp/1BQDeQT.

King G, Tomz M, Wittenberg J (2000). "Making the Most of Statistical Analyses: Improving Interpretation and Presentation." *American Journal of Political Science*, **44**(2), 341–355. URL http://gking.harvard.edu/files/abs/making-abs.shtml.

King G, Zeng L (2001a). "Explaining Rare Events in International Relations." *International Organization*, **55**(3), 693–715. URL http://gking.harvard.edu/files/abs/baby0s-abs.shtml.

King G, Zeng L (2001b). "Logistic Regression in Rare Events Data." *Political Analysis*, **9**(2), 137–163. URL http://gking.harvard.edu/files/abs/0s-abs.shtml.

King G, Zeng L (2006). "The Dangers of Extreme Counterfactuals." *Political Analysis*, **14**(2), 131–159. URL http://gking.harvard.edu/files/abs/counterft-abs.shtml.

King G, Zeng L (2007). "When Can History Be Our Guide? The Pitfalls of Counterfactual Inference." *International Studies Quarterly*, pp. 183–210. URL http://gking.harvard.edu/files/abs/counterf-abs.shtml.

Pebesma EJ, Bivand RS (2005). "Classes and methods for spatial data in R." *R News*, **5**(2), 9–13. URL http://CRAN.R-project.org/doc/Rnews/.

Schafer JL (1997). *Analysis of incomplete multivariate data*. Chapman & Hall, London.

Stoll H, King G, Zeng L (2005). "WhatIf: Software for Evaluating Counterfactuals." *Journal of Statistical Software*, **15**(4). URL http://www.jstatsoft.org/index.php?vol=15.

Stuart EA (2010). "Matching Methods for Causal Inference: A Review and a Look Forward." *Statistical Science*, **25**(1), 1–21. doi:10.1214/09-STS313.

Wickham H (2014). *Advanced R*. Chapman & Hall/CRC The R Series. Taylor & Francis. ISBN 9781466586963. URL http://adv-r.had.co.nz/.

Wickham H, Francois R (2015). *dplyr: A Grammar of Data Manipulation*. R package version 0.4.2, URL http://CRAN.R-project.org/package=dplyr.