Zelig: Everyone's Statistical Software[1]

Kosuke Imai[2] Gary King[3] Olivia Lau[4]

Version 2.7-4
November 10, 2006

[1]The current version of this software is available at `http://gking.harvard.edu/zelig/`, free of charge and open-source (under the terms of the GNU GPL, v. 2).

[2]Assistant Professor, Department of Politics, Princeton University (Corwin Hall, Department of Politics, Princeton University, Princeton NJ 08544; `http://Imai.Princeton.Edu`, `KImai@Princeton.Edu`).

[3]David Florence Professor of Government, Harvard University (Institute for Quantitative Social Sciences, 1737 Cambridge Street, Harvard University, Cambridge MA 02138; `http://GKing.Harvard.Edu`, `King@Harvard.Edu`, (617) 495-2027).

[4]Ph.D. Candidate, Department of Government, Harvard University (1737 Cambridge Street, Cambridge MA 02138; `http://www.people.fas.harvard.edu/~olau`, `OLau@Fas.Harvard.Edu`).

Contents

Acknowledgments

The Zelig project would not have been possible without considerable help from many sources. Our special thanks go to the R core team for providing an excellent modular, open-source platform for the entire statistics and methodological community.

The authors of the following R packages have provided some of the models available through Zelig: `MASS` by William N. Venables and Brian D. Ripley; `MCMCpack` by Andrew D. Martin and Kevin M. Quinn; `survival` by Terry Therneau and Brian D. Ripley; and `VGAM` by Thomas Yee.

The authors of the following R packages have provided some of the auxiliary statistical procedures and methods available through Zelig: `boot` by Angelo Canty and Brian Ripley; `coda` by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines; `sandwich` by Achim Zeileis; and `zoo` by Achim Zeileis and Gabor Grothendieck.

Our appreciation also goes to Ferdinand Almahdi, Ben Goodrich, Justin Grimmer, Yang Lin, and Ying Lu, who have contributed either code or documentation; Dan Hopkins, Ian Yohai, and others who have given valuable feedback from the courses in which they have used Zelig as a teaching tool.

For research support, we thank the National Institutes of Aging (P01 AG17625-01), the National Science Foundation (SES-0318275, SES-0550873, IIS-9874747, SES-0112072), the Mexican Ministry of Health, the U.S. Library of Congress (PA# NDP03-1), and the Princeton University Committee on Research in the Humanities and Social Sciences.

As usual, all errors are our responsibility.

Chapter 1

Introduction

1.1 What Zelig and R Do

Zelig[1] is an easy-to-use program that can estimate and help interpret the results of an enormous and growing range of statistical models. It literally *is* "everyone's statistical software" because Zelig's unified framework incorporates everyone else's (R) code. We also hope it will *become* "everyone's statistical software" for applications, and we have designed Zelig so that anyone can use it or add their models to it.

When you are using Zelig, you are also using R, a powerful statistical software language. You do not need to learn R separately, however, since this manual introduces you to R through Zelig, which simplifies R and reduces the amount of programming knowledge you need to get started. Because so many individuals contribute different packages to R (each with their own syntax and documentation), estimating a statistical model can be a frustrating experience. Users need to know which package contains the model, find the modeling command within the package, and refer to the manual page for the model-specific arguments. In contrast, Zelig users can skip these start-up costs and move directly to data analyses. Using Zelig's unified command syntax, you gain the convenience of a packaged program, without losing any of the power of R's underlying statistical procedures.

In addition to generalizing R packages and making existing methods easier to use, Zelig includes infrastructure that can improve all existing methods and R programs. Even if you know R, using Zelig greatly simplifies your work. It mimics the popular Clarify program for Stata (and thus the suggestions of King, Tomz, and Wittenberg, 2000) by translating the raw output of existing statistical procedures into quantities that are of direct interest to researchers. Instead of trying to interpret coefficients parameterized for modeling convenience, Zelig makes it easy to compute quantities of real interest: probabilities, predicted values, expected values, first differences, and risk ratios, along with confidence intervals, standard errors, or full posterior (or sampling) densities for all quantities. Zelig extends Clarify by

[1] Zelig is named after a Woody Allen movie about a man who had the strange ability to become the physical reflection of anyone he met — Scottish, African-American, Indian, Chinese, thin, obese, medical doctor, Hassidic rabbi, anything — and thus to fit well in any situation.

seamlessly integrating an option for bootstrapping into the simulation of quantities of interest. It also integrates a full suite of nonparametric matching methods as a preprocessing step to improve the performance of any parametric model for causal inference (see MatchIt). For missing data, Zelig accepts multiply imputed datasets created by Amelia (see King, Honaker, Joseph, and Scheve, 2001) and other programs, allowing users to analyze them as if they were a single, fully observed dataset. Zelig outputs replication data sets so that you (and if you wish, anyone else) will always be able to replicate the results of your analyses (see King, 1995). Several powerful Zelig commands also make running multiple analyses and recoding variables simple.

Using R in combination with Zelig has several advantages over commercial statistical software. R and Zelig are part of the open source movement, which is roughly based on the principles of science. That is, anyone who adds functionality to open source software or wishes to redistribute it (legally) must provide the software accompanied by its source free of charge.[2] If you find a bug in open source software and post a note to the appropriate mailing list, a solution you can use will likely be posted quickly by one of the thousands of people using the program all over the world. Since you can see the source code, you might even be able to fix it yourself. In contrast, if something goes wrong with commercial software, you have to wait for the programmers at the company to fix it (and speaking with them is probably out of the question), and wait for a new version to be released.

We find that Zelig makes students and colleagues more amenable to using R, since the startup costs are lower, and since the manual and software are relatively self-contained. This manual even includes an appendix devoted to the basics of advanced R programming, although you will not need it to run most procedures in Zelig. A large and growing fraction of the world's quantitative methodologists and statisticians are moving to R, and the base of programs available for R is quickly surpassing all alternatives. In addition to built-in functions, R is a complete programming language, which allows you to design new functions to suit your needs. R has the dual advantage that you do not need to understand how to program to use it, but if it turns out that you want to do something more complicated, you do not need to learn another program. In addition, methodologists all over the world add new functions all the time, so if the function you need wasn't there yesterday, it may be available today.

1.2 Getting Help

You may find documentation for Zelig on-line (and hence must be on-line to access it). If you are unable to connect to the Internet, we recommend that you print the pdf version of this document for your reference.

If you are on-line, you may access comprehensive help files for Zelig commands and for each of the models. For example, load the Zelig library and then type at the R prompt:

```
> help.zelig(command)                # For help with all zelig commands.
```

[2] As specified in the GNU General License v. 2 `http://www.gnu.org/copyleft`.

```
> help.zelig(logit)                    # For help with the logit model.
```

In addition, `help.zelig()` searches the manual pages for R in addition to the Zelig specific pages. On certain rare occasions, the name of the help topic in Zelig and in R are identical. In these cases, `help.zelig()` will return the Zelig help page by default. If you wish to access the R help page, you should use `help(topic)`.

In addition, built-in examples with sample data and plots are available for each model. For example, type `demo(logit)` to view the demo for the logit model. Commented code for each model is available under the examples section of each model reference page.

Please direct inquiries and problems about Zelig to our listserv at zelig@latte.harvard.edu. We suggest you subscribe to this mailing list while learning and using Zelig: go to `http://lists.hmdc.harvard.edu/index.cgi?info=zelig`. (You can choose to receive email in digest form, so that you will never receive more than one message per day.) You can also browse or search our archive of previous messages before posting your query.

Part I

User's Guide

Chapter 2

Installation

To use Zelig, you must install the statistical program R (if it is not already installed), the Zelig package, and some R libraries (coda, MCMCpack, sandwich, VGAM, and zoo).

Note: In this document, > denotes the R prompt.

If You Know R

We recommend that you launch R and type

```
> source("http://gking.harvard.edu/zelig/install.R")
> library(Zelig)
```

then proceed to Section 4.1.1. For Windows R, you may edit the `Rprofile` file to load Zelig automatically at launch (after which you will no longer need to type `library(Zelig)` at startup). Simply add the line:

```
options(defaultPackages = c(getOption("defaultPackages"), "Zelig"))
```

If You Are New to R

If you are new to R, we recommend that you read the following section on installation procedures as well as the overview of R syntax and usage in Section 6.

This distribution works on a variety of platforms, including Windows (see Section 2.1), MacOSX (see Section 2.2), and Linux (see Section 2.3). Alternatively, you may access R from your PC using a terminal window or an X-windows tunnel to a Linux or Unix server (see Section 2.3). Most servers have R installed; if not, contact your network administrator.

There are advantages and disadvantages to each type of installation. On a personal computer, R is easier to install and launch. Using R remotely on a server requires a bit more set-up, but does not tie up your local CPU, and allows you to take advantage of the server's speed.

2.1 Windows

Installing R

Go to the Comprehensive R Archive Network website (`http://www.r-project.org`) and download the latest installer for Windows at `http://cran.us.r-project.org/bin/windows/base/` Double-click the `.exe` file to launch the R installer. We recommend that you accept the default installation options if this your first installation.

Installing Zelig

Once R is installed, you must install the Zelig and VGAM packages. There are three ways to do this.

1. We recommend that you start R and then type:

   ```
   > source("http://gking.harvard.edu/zelig/install.R")
   > library(Zelig)
   ```

2. Alternatively, you may install each component package individually in R:

   ```
   > install.packages("Zelig")
   > install.packages("MCMCpack")
   > install.packages("coda")
   > install.packages("VGAM")
   > install.packages("zoo")
   > install.packages("sandwich")
   > library(Zelig)
   ```

   Zelig will load the optional libraries whenever their functions are needed; it is not necessary to load any package other than Zelig at startup.

3. Alternatively, you may use the drop down menus to install Zelig. This requires four steps.

   (a) Go to the Zelig website and and download the latest release of Zelig. The VGAM, MCMCpack, coda, zoo, and sandwich packages are available from CRAN. Store these `.zip` files in your R program directory. For example, the default R program directory is `C:\Program Files\R\R-2.3.1\`.[1]

   (b) Start R. From the drop-down menus, select the "Packages" menu and then the "Install Files from Local Zip Files" option.

[1] Note that when updating R to the latest release, the installer does not delete previous versions from your `C:\Program Files\R\` directory. In this example, the subdirectory `\R-2.3.1\` stores R version 2.3.1. Thus, if you have a different version of R installed, you should change the last part of the R program directory file path accordingly.

(c) A window will pop up, allowing you to select one of the downloaded files for installation. There is no need to unzip the files prior to installation. Repeat and select the other downloaded file for installation.

(d) At the R prompt, type `library(Zelig)` to load the functionality described in this manual. Note that Zelig will automatically load the other libraries as necessary.

4. An additional *recommended but optional step* is to set up R to load Zelig automatically at launch. (If you skip this step, you must type `library(Zelig)` at the beginning of every R session.) To automate this process, edit the `Rprofile` file located in the R program subdirectory (`C:\Program Files\R\R-2.3.1\etc\` in our example). Using a text editor such as Windows notepad, add the following line to the `Rprofile` file:

```
options(defaultPackages = c(getOption("defaultPackages"), "Zelig"))
```

Zelig is distributed under the GNU General Public License, Version 2. After installation, the source code is located in your R library directory, which is by default `C:\Program Files\R\R-2.3.1\library\Zelig\`.

Updating Zelig

There are two ways to update Zelig.

1. We recommend that you periodically update Zelig at the R prompt by typing:

```
> update.packages()
> library(Zelig)
```

2. Alternatively, you may use the procedure outlined in Section 3a to periodically update Zelig. Simply download the latest `.zip` file and follow the four steps.

2.2 MacOS X

Installing R

If you are using MacOS X, you may install the latest version of R (2.3.1 at this time) from the CRAN website http://cran.us.r-project.org/bin/macosx/. At this time, Zelig is not supported for R on MacOS 8.6 through 9.x.

Installing Zelig

Once R is installed, you must install the Zelig and VGAM packages. There are several ways to do this.

1. **For RAqua**:

(a) We recommend that you start R, and then type:

```
> source("http://gking.harvard.edu/zelig/install.R")
> library(Zelig)
```

(You may ignore the warning messages, unless they say "Non-zero exit status".)

(b) Alternatively, to avoid the warning messages, you need to install each package individually and specify the specific installation path:

```
> install.packages("Zelig", lib = "~/Library/R/library")
> install.packages("MCMCpack", lib = "~/Library/R/library")
> install.packages("coda", lib = "~/Library/R/library")
> install.packages("VGAM", lib = "~/Library/R/library")
> install.packages("zoo", lib = "~/Library/R/library")
> install.packages("sandwich", lib = "~/Library/R/library")
> library(Zelig)
```

where `~/Library/R/library` is the default local library directory. Zelig will load the other libraries whenever their functions are needed; it is not necessary to load these packages at startup.

(c) Alternatively, you may use the drop down menus to install Zelig. This requires three steps.

i. Go to the Zelig website and download the latest release of Zelig. The VGAM, MCMCpack, coda, zoo, and sandwich packages are available from CRAN. Save these `.tar.gz` files in a convenient place.

ii. Start R. From the drop-down menus, select the "Packages" menu and then the "Install Files from Local Files" option.

iii. A window will pop up, allowing you to select the one of the downloaded files for installation. There is no need to unzip the files prior to installation. Repeat and select the other downloaded file for installation.

2. **For command line R:**

(a) Before installing command line R, you need to create a local R library directory. If you have done so already, you may skip to the next step. Otherwise, at the terminal prompt in your home directory, type:

```
% mkdir ~/Library/R ~/Library/R/library
```

(b) Modify your configuration file to identify `~/Library/R/library` as your R library directory. There are two ways of doing this:

i. Open the `.Renviron` file (or create one, if you don't have one) and add the following line:

```
R_LIBS = "~/Library/R/library"
```

ii. *Alternatively*, you may modify your shell configuration file. For a Bash shell, open your `.bashrc` file and add the following line:

```
export R_LIBS="$HOME/Library/R/library"
```

(c) Start R and at the prompt, type:

```
> source("http://gking.harvard.edu/zelig/install.R")
> library(Zelig)
```

(You may ignore the warning messages, unless they say "Non-zero exit status".)

(d) Alternatively, to avoid the warning messages, you need to install each component package separately and specify the installation path:

```
> install.packages("Zelig", lib = "~/Library/R/library")
> install.packages("MCMCpack", lib = "~/Library/R/library")
> install.packages("coda", lib = "~/Library/R/library")
> install.packages("VGAM", lib = "~/Library/R/library")
> install.packages("zoo", lib = "~/Library/R/library")
> install.packages("sandwich", lib = "~/Library/R/library")
> library(Zelig)
```

Although the `lib` argument is optional, we recommend that you set it to the default RAqua directory (`"~/Library/R/library"`), in case you later decide to install the RAqua GUI (which has a different default directory).

At the R prompt, type `library(Zelig)` to load the functionality described in this manual. Note that Zelig will automatically load the other packages as necessary.

Zelig is distributed under the GNU General Public License, Version 2. After installation, the source code is located in your R library directory, `~/Library/R/library/Zelig/`.

Updating Zelig

There are two ways to update Zelig.

1. We recommend that you start R and, at the R prompt, type:

   ```
   > update.packages()
   ```

2. Alternatively, you may remove an old version by command by typing `R CMD REMOVE Zelig` at the terminal prompt. Then download and reinstall the package using the installation procedures Section 2.2 outlined above.

2.3 UNIX and Linux

Installing R

Type `R` at the terminal prompt (which we denote as `%` in this section) to see if R is available. (Typing `q()` will enable you to quit.) If it is installed, proceed to the next section. If it is not installed and you are not the administrator, contact that individual, kindly request that they install R on the server, and continue to the next section. If you have administrator privileges, you may download the latest release at the CRAN website. Although installation varies according to your Linux distribution, we provide an example for Red Hat Linux 9.0 as a guide:

1. Log in as root.
2. Download the appropriate binary file for Red Hat 9 from CRAN. For example, for Red Hat 9 running on the Intel 386 platform, go to `http://cran.r-project.org/bin/linux/`.
3. Type the following command at the terminal prompt:
   `% rpm -ivh R-2.3.1-1.i386.rpm`

Installing Zelig

Before installing Zelig, you need to create a local R library directory. If you have done so already, you can skip to Section 2.3. If not, you must do so before proceeding because most users do not have authorization to install programs globally. Suppose we want the directory to be `~/.R/library`. At the terminal prompt in your home directory, type:

```
% mkdir ~/.R ~/.R/library
```

Now you are ready to install Zelig. There are two ways to proceed.

1. Recommended procedure:
   (a) Open the `~/.Renviron` file (or create it if it does not exist) and add the following line:

   ```
   R_LIBS = "~/.R/library"
   ```

   You only need to perform this step once.

   (b) Start R. At the R prompt, type:

   ```
   > source("http://gking.harvard.edu/zelig/install.R")
   > library(Zelig)
   ```

   (You may ignore the warning messages, unless they say "Non-zero exit status".)

(c) Alternatively, you can avoid the warning messages by installing each component package separately and specifying the installation path:

```
> install.packages("Zelig", lib = "~/.R/library")
> install.packages("MCMCpack", lib = "~/.R/library")
> install.packages("coda", lib = "~/.R/library")
> install.packages("VGAM", lib = "~/.R/library")
> install.packages("zoo", lib = "~/.R/library")
> install.packages("sandwich", lib = "~/.R/library")
> library(Zelig)
```

(d) Finally, create a `.Rprofile` file in your home directory, containing the line:

```
library(Zelig)
```

This will load Zelig every time you start R.

2. Alternatively:

(a) Add the local R library directory that you created above (`~/.R/library` in the example) to the environmental variable `R_LIBS`.

(b) Download the latest bundles for Unix from the Zelig website, and (for the VGAM, MCMCpack, coda, sandwich, and zoo packages) from the CRAN website.

(c) If `XX` is the current version number, at the terminal prompt, type:

```
% R CMD INSTALL Zelig_XX.tar.gz
% R CMD INSTALL MCMCpack_XX.tar.gz
% R CMD INSTALL coda_XX.tar.gz
% R CMD INSTALL VGAM_XX.tar.gz
% R CMD INSTALL zoo_XX.tar.gz
% R CMD INSTALL sandwich_XX.tar.gz
% rm Zelig_XX.tar.gz VGAM_XX.tar.gz zoo_XX.tar.gz sandwich_XX.tar.gz
```

(d) Create a `.Rprofile` file in your home directory, containing the line:

```
library(Zelig)
```

This will load Zelig every time you start R.

Zelig is distributed under the GNU General Public License, Version 2. After installation, the source code is located in your R library directory. If you followed the example above, this is `/.R/library/Zelig/`.

Updating Zelig

There are two ways to update Zelig.

1. We recommend that you start R and, at the R prompt, type:

```
> update.packages()
```

2. Alternatively, you may remove an old version by command by typing `R CMD REMOVE Zelig` at the terminal prompt. Then download and reinstall the package using the installation procedure Section 2.3 outlined above.

2.4 Version Compatability

In addition to R itself, Zelig also depends on several R packages maintained by other development teams. Although we make every effort to keep the latest version of Zelig up-to-date with the latest version of those packages, there may occasionally be incompatabilities. See B.1 in the Appendix for a list of packages tested to be compatabile with a given Zelig release. You may obtain older versions of most packages at `http://www.r-project.org`.

Chapter 3

Data Analysis Commands

3.1 Command Syntax

Once R is installed, you only need to know a few basic elements to get started. It's important to remember that R, like any spoken language, has rules for proper syntax. Unlike English, however, the rules for intelligible R are small in number and quite precise (see Section 3.1.2).

3.1.1 Getting Started

1. To start R under Linux or Unix, type R at the terminal prompt or M-x R under ESS.
2. The R prompt is `>`.
3. Type commands and hit enter to execute. (No additional characters, such as semicolons or commas, are necessary at the end of lines.)
4. To quit from R, type `q()` and press enter.
5. The `#` character makes R ignore the rest of the line, and is used in this document to comment R code.
6. We highly recommend that you make a separate working directory or folder for each project.
7. Each R session has a workspace, or working memory, to store the *objects* that you create or input. These objects may be:
   (a) *values*, which include numerical, integer, character, and logical values;
   (b) *data structures* made up of variables (vectors), matrices, and data frames; or
   (c) *functions* that perform the desired tasks on user-specified values or data structures.

After starting R, you may at any time use Zelig's built-in help function to access on-line help for any command. To see help for all Zelig commands, type `help.zelig(command)`, which will take you to the help page for all Zelig commands. For help with a specific Zelig or R command substitute the name of the command for the generic `command`. For example, type `help.zelig(logit)` to view help for the logit model.

3.1.2 Details

.

Zelig uses the syntax of R, which has several essential elements:

1. R is case sensitive. `Zelig`, the package or library, is not the same as `zelig`, the command.

2. R functions accept user-defined arguments: while some arguments are required, other optional arguments modify the function's default behavior. Enclose arguments in parentheses and separate multiple arguments with commas. For example, `print(x)` or `print(x, digits = 2)` prints the contents of the object `x` using the default number of digits or rounds to two digits to the right of the decimal point, respectively. You may nest commands as long as each has its own set of parentheses: `log(sqrt(5))` takes the square root of 5 and then takes the natural log.

3. The `<-` operator takes the output of the function on the right and saves them in the named object on the left. For example, `z.out <- zelig(...)` stores the output from `zelig()` as the object `z.out` in your working memory. You may use `z.out` as an argument in other functions, view the output by typing `z.out` at the R prompt, or save `z.out` to a file using the procedures described in Section 3.2.3.

4. You may name your objects anything, within a few constraints:

   - You may only use letters (in upper or lower case) and periods to punctuate your variable names.
   - You may *not* use any special characters (aside from the period) or spaces to punctuate your variable names.
   - Names cannot begin with numbers. For example, R will not let you save an object as `1997.election` but will let you save `election.1997`.

5. Use the `names()` command to see the contents of R objects, and the `$` operator to extract elements from R objects. For example:

```
# Run least squares regression and save the output in working memory:
> z.out <- zelig(y ~ x1 + x2, model = "ls", data = mydata)
# See what's in the R object:
> names(z.out)
```

```
 [1] "coefficients"  "residuals"  "effects"  "rank"
# Extract and display the coefficients in z.out:
> z.out$coefficients
```

6. All objects have a class designation which tells R how to treat it in subsequent commands. An object's class is generated by the function or mathematical operation that created it.

7. To see a list of all objects in your current workspace, type: `ls()`. You can remove an object permanently from memory by typing `remove(goo)` (which deletes the object `goo`), or remove all the objects with `remove(list = ls())`.

8. To run commands in a batch, use a text editor (such as the Windows R script editor or emacs) to compose your R commands, and save the file with a `.R` file extension in your working directory. To run the file, type `source("Code.R")` at the R prompt.

If you encounter a syntax error, check your spelling, case, parentheses, and commas. These are the most common syntax errors, and are easy to detect and correct with a little practice. If you encounter a syntax error in batch mode, R will tell you the line on which the syntax error occurred.

3.2 Data Sets

3.2.1 Data Structures

Zelig uses only three of R's many data structures:

1. A **variable** is a one-dimensional vector of length n.

2. A **data frame** is a rectangular matrix with n rows and k columns. Each column represents a variable and each row an observation. Each variable may have a different class. (See Section 3.3.1 for a list of classes.) You may refer to specific variables from a data frame using, for example, `data$variable`.

3. A **list** is a combination of different data structures. For example, `z.out` contains both `coefficients` (a vector) and `data` (a data frame). Use `names()` to view the elements available within a list, and the `$` operator to refer to an element in a list.

For a more comprehensive introduction, including ways to manipulate these data structures, please refer to Chapter 6.

3.2.2 Loading Data

Datasets in Zelig are stored in "data frames." In this section, we explain the standard ways to load data from disk into memory, how to handle special cases, and how to verify that the data you loaded is what you think it is.

Standard Ways to Load Data

Make sure that the data file is saved in your working directory. You can check to see what your working directory is by starting R, and typing `getwd()`. If you wish to use a different directory as your starting directory, use `setwd("dirpath")`, where `"dirpath"` is the full directory path of the directory you would like to use as your working directory.

After setting your working directory, load data using one of the following methods:

1. If your dataset is in a **tab- or space-delimited .txt file**, use `read.table("mydata.txt")`

2. If your dataset is **a comma separated table**, use `read.csv("mydata.csv")`.

3. To import **SPSS, Stata, and other data files**, use the foreign package, which automatically preserves field characteristics for each variable. Thus, variables classed as dates in Stata are automatically translated into values in the date class for R. For example:

```
> library(foreign)                              # Load the foreign package.
> stata.data <- read.dta("mydata.dta")          # For Stata data.
> spss.data <- read.spss("mydata.sav", to.data.frame = TRUE) # For SPSS.
```

4. To load data in R format, use `load("mydata.RData")`.

5. For sample data sets included with R packages such as Zelig, you may use the `data()` command, which is a shortcut for loading data from the sample data directories. Because the locations of these directories vary by installation, it is extremely difficult to locate sample data sets and use one of the three preceding methods; `data()` searches all of the currently used packages and loads sample data automatically. For example:

```
> library(Zelig)                                # Loads the Zelig library.
> data(turnout)                                 # Loads the turnout data.
```

Special Cases When Loading Data

These procedures apply to any of the above `read` commands:

1. If your file uses the **first row to identify variable names**, you should use the option `header = TRUE` to import those field names. For example,

   ```
   > read.csv("mydata.csv", header = TRUE)
   ```

   will read the words in the first row as the variable names and the subsequent rows (each with the same number of values as the first) as observations for each of those variables. If you have additional characters on the last line of the file or fewer values in one of the rows, you need to edit the file before attempting to read the data.

2. The R missing value code is `NA`. If this value is in your data, R will recognize your missing values as such. If you have instead used a place-holder value (such as -9) to represent missing data, you need to tell R this on loading the data:

   ```
   > read.table("mydata.tab", header = TRUE, na.strings = "-9")
   ```

   Note: You must enclose your place holder values in quotes.

3. Unlike Windows, the file extension in R does not determine the default method for dealing with the file. For example, if your data is tab-delimited, but saved as a `.sav` file, `read.table("mydata.sav")` will load your data into R.

Verifying You Loaded The Data Correctly

Whichever method you use, try the `names()`, `dim()`, and `summary()` commands to verify that the data was properly loaded. For example,

```
> data <- read.csv("mydata.csv", header = TRUE)             # Read the data.
> dim(data)                    # Displays the dimensions of the data frame
[1] 16000  8                    #  in rows then columns.
> data[1:10,]                  # Display rows 1-10 and all columns.
> names(data)                  # Check the variable names.
[1] "V1" "V2" "V3"             # These values indicate that the variables
                               #  weren't named, and took default values.
> names(data) <- c("income", "educate", "year")     # Assign variable names.
> summary(data)                # Returning a summary for each variable.
```

In this case, the `summary()` command will return the maximum, minimum, mean, median, first and third quartiles, as well as the number of missing values for each variable.

3.2.3 Saving Data

Use `save()` to write data or any object to a file in your working directory. For example,

```
> save(mydata, file = "mydata.RData")     # Saves 'mydata' to 'mydata.RData'
                                          #  in your working directory.
> save.image()                            # Saves your entire workspace to
                                          #  the default '.RData' file.
```

R will also prompt you to save your workspace when you use the `q()` command to quit. When you start R again, it will load the previously saved workspace. Restarting R will not, however, load previously used packages. You must remember to load Zelig at the beginning of every R session.

Alternatively, you can recall individually saved objects from .RData files using the `load()` command. For example,

```
> load("mydata.RData")
```

loads the objects saved in the `mydata.RData` file. You may save a data frame, a data frame and associated functions, or other R objects to file.

3.3 Variables

3.3.1 Classes of Variables

R variables come in several types. Certain Zelig models require dependent variables of a certain class of variable. (These are documented under the manual pages for each model.) Use `class(variable)` to determine the class of a variable or `class(data$variable)` for a variable within a data frame.

Types of Variables

For all types of variable (vectors), you may use the `c()` command to "concatenate" elements into a vector, the `:` operator to generate a sequence of integer values, the `seq()` command to generate a sequence of non-integer values, or the `rep()` function to repeat a value to a specified length. In addition, you may use the `<-` operator to save variables (or any other objects) to the workspace. For example:

```
> logic <- c(TRUE, FALSE, TRUE, TRUE, TRUE) # Creates 'logic' (5 T/F values).
> var1 <- 10:20                             # All integers between 10 and 20.
> var2 <- seq(from = 5, to = 10, by = 0.5)  # Sequence from 5 to 10 by
                                            #  intervals of 0.5.
> var3 <- rep(NA, length = 20)              # 20 'NA' values.
> var4 <- c(rep(1, 15), rep(0, 15))         # 15 '1's followed by 15 '0's.
```

For the `seq()` command, you may alternatively specify `length` instead of `by` to create a variable with a specific number (denoted by the `length` argument) of evenly spaced elements.

1. **Numeric** variables are real numbers and the default variable class for most dataset values. You can perform any type of math or logical operation on numeric values. If `var1` and `var2` are numeric variables, we can compute

   ```
   > var3 <- log(var2) - 2*var1      # Create 'var3' using math operations.
   ```

   `Inf` (infinity), `-Inf` (negative infinity), `NA` (missing value), and `NaN` (not a number) are special numeric values on which most math operations will fail. (Logical operations will work, however.) Use `as.numeric()` to transform variables into numeric variables. Integers are a special class of numeric variable.

2. **Logical** variables contain values of either `TRUE` or `FALSE`. R supports the following logical operators: `==`, exactly equals; `>`, greater than; `<`, less than; `>=`, greater than or equals; `<=`, less than or equals; and `!=`, not equals. The `=` symbol is *not* a logical operator. Refer to Section 3.3.2 for more detail on logical operators. If `var1` and `var2` both have n observations, commands such as

   ```
   > var3 <- var1 < var2
   > var3 <- var1 == var2
   ```

   create n `TRUE`/`FALSE` observations such that the ith observation in `var3` evaluates whether the logical statement is true for the ith value of `var1` with respect to the ith value of `var2`. Logical variables should usually be converted to integer values prior to analysis; use the `as.integer()` command.

3. **Character** variables are sets of text strings. Note that text strings are always enclosed in quotes to denote that the string is a value, not an object in the workspace or an argument for a function (neither of which take quotes). Variables of class character are not normally used in data analysis, but used as descriptive fields. If a character variable is used in a statistical operation, it must first be transformed into a factored variable.

4. **Factor** variables may contain values consisting of either integers or character strings. Use `factor()` or `as.factor()` to convert character or integer variables into factor variables. Factor variables separate unique values into levels. These levels may either be ordered or unordered. In practice, this means that including a factor variable among the explanatory variables is equivalent to creating dummy variables for each level. In addition, some models (ordinal logit, ordinal probit, and multinomial logit), require that the dependent variable be a factor variable.

3.3.2 Recoding Variables

Researchers spend a significant amount of time cleaning and recoding data prior to beginning their analyses. R has several procedures to facilitate the process.

Extracting, Replacing, and Generating New Variables

While it is not difficult to recode variables, the process is prone to human error. Thus, we recommend that before altering the data, you save your existing data frame using the procedures described in Section 3.2.3, that you only recode one variable at a time, and that you recode the variable outside the data frame and then return it to the data frame.

To extract the variable you wish to recode, type:

```
> var <- data$var1              # Copies 'var1' from 'data', creating 'var'.
```

Do *not* sort the extracted variable or delete observations from it. If you do, the ith observation in `var` will no longer match the ith observation in `data`.

To replace the variable or generate a new variable in the data frame, type:

```
> data$var1 <- var                  # Replace 'var1' in 'data' with 'var'.
> data$new.var <- var               # Generate 'new.var' in 'data' using 'var'.
```

To remove a variable from a data frame (rather than replacing one variable with another):

```
> data$var1 <- NULL
```

Logical Operators

R has an intuitive method for recoding variables, which relies on logical operators that return statements of `TRUE` and `FALSE`. A mathematical operator (such as `==`, `!=`, `>`, `>=` `<`, and `<=`) takes two objects of equal dimensions (scalars, vectors of the same length, matrices with the same number of rows and columns, or similarly dimensioned arrays) and compares every element in the first object to its counterpart in the second object.

- `==`: checks that one variable "exactly equals" another in a list-wise manner. For example:

  ```
  > x <- c(1, 2, 3, 4, 5)                 # Creates the object 'x'.
  > y <- c(2, 3, 3, 5, 1)                 # Creates the object 'y'.
  > x == y                                # Only the 3rd 'x' exactly equals
    [1] FALSE FALSE  TRUE FALSE FALSE     #  its counterpart in 'y'.
  ```

  (The `=` symbol is *not* a logical operator.)

- `!=`: checks that one variable does not equal the other in a list-wise manner. Continuing the example:

  ```
  > x != y
    [1]  TRUE  TRUE FALSE  TRUE  TRUE
  ```

- `>` (`>=`): checks whether each element in the left-hand object is greater than (or equal to) every element in the right-hand object. Continuing the example from above:

  ```
  > x > y                                   # Only the 5th 'x' is greater
    [1] FALSE FALSE FALSE FALSE  TRUE       #  than its counterpart in 'y'.
  > x >= y                                  # The 3rd 'x' is equal to the
    [1] FALSE FALSE  TRUE FALSE  TRUE       #  3rd 'y' and becomes TRUE.
  ```

- `<` (`<=`): checks whether each element in the left-hand object is less than (or equal to) every object in the right-hand object. Continuing the example from above:

```
> x < y                             # The elements 1, 2, and 4 of ‘x’ are
[1]  TRUE  TRUE FALSE  TRUE FALSE #  less than their counterparts in ‘y’.
> x <= y                            # The 3rd ‘x’ is equal to the 3rd ‘y’
[1]  TRUE  TRUE  TRUE  TRUE FALSE #  and becomes TRUE.
```

For two vectors of five elements, the mathematical operators compare the first element in `x` to the first element in `y`, the second to the second and so forth. Thus, a mathematical comparison of `x` and `y` returns a vector of five `TRUE`/`FALSE` statements. Similarly, for two matrices with 3 rows and 20 columns each, the mathematical operators will return a 3×20 matrix of logical values.

There are additional logical operators which allow you to combine and compare logical statements:

- `&`: is the logical equivalent of “and”, and evaluates one array of logical statements against another in a list-wise manner, returning a `TRUE` only if both are true in the same location. For example:

```
> a <- matrix(c(1:12), nrow = 3, ncol = 4)      # Creates a matrix ‘a’.
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> b <- matrix(c(12:1), nrow = 3, ncol = 4)      # Creates a matrix ‘b’.
> b
     [,1] [,2] [,3] [,4]
[1,]   12    9    6    3
[2,]   11    8    5    2
[3,]   10    7    4    1
> v1 <- a > 3                   # Creates the matrix ‘v1’ (T/F values).
> v2 <- b > 3                   # Creates the matrix ‘v2’ (T/F values).
> v1 & v2                       # Checks if the (i,j) value in ‘v1’ and
      [,1] [,2] [,3]  [,4]      #  ‘v2’ are both TRUE.  Because columns
[1,] FALSE TRUE TRUE FALSE      #  2-4 of ‘v1’ are TRUE, and columns 1-3
[2,] FALSE TRUE TRUE FALSE      #  of ‘var2’ are TRUE, columns 2-3 are
[3,] FALSE TRUE TRUE FALSE      #  TRUE here.
> (a > 3) & (b > 3)             # The same, in one step.
```

  For more complex comparisons, parentheses may be necessary to delimit logical statements.

- `|`: is the logical equivalent of “or”, and evaluates in a list-wise manner whether either of the values are `TRUE`. Continuing the example from above:

```
> (a < 3) | (b < 3)                # (1,1) and (2,1) in 'a' are less
      [,1]  [,2]  [,3]  [,4]       #  than 3, and (2,4) and (3,4) in
[1,]  TRUE FALSE FALSE FALSE       #  'b' are less than 3; | returns
[2,]  TRUE FALSE FALSE  TRUE       #  a matrix with 'TRUE' in (1,1),
[3,] FALSE FALSE FALSE  TRUE       #  (2,1), (2,4), and (3,4).
```

The `&&` (if and only if) and `||` (either or) operators are used to control the command flow within functions. The `&&` operator returns a `TRUE` only if every element in the comparison statement is true; the `||` operator returns a `TRUE` if any of the elements are true. Unlike the `&` and `|` operators, which return arrays of logical values, the `&&` and `||` operators return only one logical statement irrespective of the dimensions of the objects under consideration. Hence, `&&` and `||` are logical operators which are *not* appropriate for recoding variables.

Coding and Recoding Variables

R uses vectors of logical statements to indicate how a variable should be coded or recoded. For example, to create a new variable `var3` equal to 1 if `var1` $<$ `var2` and 0 otherwise:

```
> var3 <- var1 < var2              # Creates a vector of n T/F observations.
> var3 <- as.integer(var3)         # Replaces the T/F values in 'var3' with
                                   #  1's for TRUE and 0's for FALSE.
> var3 <- as.integer(var1 < var2)  # Combine the two steps above into one.
```

In addition to generating a vector of dummy variables, you can also refer to specific values using logical operators defined in Section 3.3.2. For example:

```
> v1 <- var1 == 5                  # Creates a vector of T/F statements.
> var1[v1] <- 4                    # For every TRUE in 'v1', replaces the
                                   #  value in 'var1' with a 4.
> var1[var1 == 5] <- 4             # The same, in one step.
```

The index (inside the square brackets) can be created with reference to other variables. For example,

```
> var1[var2 == var3] <- 1
```

replaces the ith value in `var1` with a 1 when the ith value in `var2` equals the ith value in `var3`. If you use `=` in place of `==`, however, you will replace all the values in `var1` with 1's because `=` is another way to assign variables. Thus, the statement `var2 = var3` is of course true.

Finally, you may also replace any (character, numerical, or logical) values with special values (most commonly, `NA`).

```
> var1[var1 == "don't know"] <- NA   # Replaces all "don't know"'s with NA's.
```

After recoding the `var1` replace the old `data$var1` with the recoded `var1`: `data$var1 <- var1`. You may combine the recoding and replacement procedures into one step. For example:

```
> data$var1[data$var1 =< 0] <- -1
```

Alternatively, rather than recoding just specific values in variables, you may calculate new variables from existing variables. For example,

```
> var3 <- var1 + 2 * var2
> var3 <- log(var1)
```

After generating the new variables, use the assignment mechanism `<-` to insert the new variable into the data frame.

In addition to generating vectors of dummy variables, you may transform a vector into a matrix of dummy indicator variables. For example, see Section 7.3 to transform a vector of k unique values (with n observations in the complete vector) into a $n \times k$ matrix.

Missing Data

To deal with missing values in some of your variables:

1. You may generate multiply imputed datasets using Amelia (or other programs).

2. You may omit missing values. Zelig models automatically apply list-wise deletion, so no action is required to run a model. To obtain the total number of observations or produce other summary statistics using the analytic dataset, you may manually omit incomplete observations. To do so, first create a data frame containing only the variables in your analysis. For example:

   ```
   > new.data <- cbind(data$dep.var, data$var1, data$var2, data$var3)
   ```

   The `cbind()` command "column binds" variables into a data frame. (A similar command `rbind()` "row binds" observations with the same number of variables into a data frame.) To omit missing values from this new data frame:

   ```
   > new.data <- na.omit(new.data)
   ```

   If you perform `na.omit()` on the full data frame, you risk deleting observations that are fully observed in your experimental variables, but missing values in other variables. Creating a new data frame containing only your experimental variables usually increases the number of observations retained after `na.omit()`.

Chapter 4

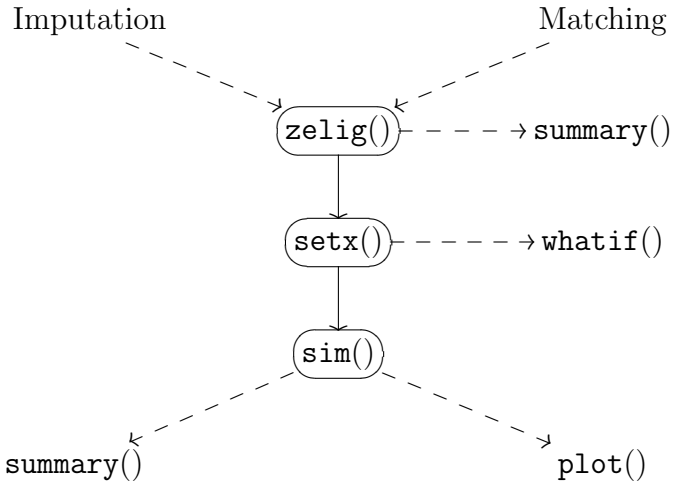
Statistical Commands

4.1 Zelig Commands

4.1.1 Quick Overview

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

Figure 4.1: 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-processing: 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 regression output using `summary()`.

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 King and Zeng (2006), `?`), Stoll et al. (2006)).

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.

4.1.2 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 Scheve and Slaughter (2001) 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)
```

4.1.3 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 4.2 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 2) 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.

   (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()`.)

[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).

(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 (King et al. 1990).
     - In ecological inference models, conditional prediction guarantees that the predicted values are on the tomography line and thus restricted to the known bounds (King 1997; Adolph et al. 2003).
     - 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 Gelman and King 1994). 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 King et al. 2004).

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.

(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
```

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

(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 6), `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.

Presenting Results

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

[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.

```
> 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).

4.2 Supported Models

We list here all models impelmented 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 12.16) 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 12.20) 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 12.21) 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) `"tobit"`: The *tobit* regression model (see Section 12.30) is a Normal distribution with left-censored observations.

   (e) `"tobit.bayes"`: The *Bayesian tobit* distribution (see Section 12.31) is a Normal distribution that has either left and/or right censored observations.

   (f) `"factor.bayes"`: The *Bayesian factor analysis* model (see Section 12.7) 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 12.13) 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 12.29) 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 12.14) 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 12.27) 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 12.28) 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) `"blogit"`: The *bivariate logistic* model (see Section 12.1) 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.

(g) `"bprobit"`: The *bivariate probit* model (see Section 12.2) 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.

(h) `"irt1d"`: The *one-dimensional item response* model (see Section 12.11) takes multiple dichotomous dependent variables and models them as a function of *one* latent (unobserved) explanatory variable.

(i) `"irtkd"`: The *k-dimensional item response* model (see Section 12.12) 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 12.22) specifies the stochastic component of the unobserved variable to be a standard logistic distribution.

   (b) `"oprobit"`: The *ordinal probit* distribution (see Section 12.23) specifies the stochastic component of the unobserved variable to be standardized normal.

   (c) `"oprobit.bayes"`: *Bayesian ordinal probit* model (see Section 12.24) 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 12.9) 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 12.17) specifies categorical responses distributed according to the multinomial stochastic component and logistic systematic component.

   (b) `"mlogit.bayes"`: *Bayesian multinomial logistic* regression (see Section 12.18) 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 12.25) 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 12.26) 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 12.19) 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 12.10) for positively-valued, continuous dependent variables that are fully observed (no censoring).

   (b) `"exp"`: The *exponential* model (see Section 12.6) 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 12.32) 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 12.15) 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 12.4) produces estimates for a cross-section of 2×2 tables.

   (b) `ei.dynamic`: *Quinn's dyanamic Bayesian* EI model (see Section 12.3) estimates a dynamic Bayesian model for 2×2 tables with temporal dependece across tables.

   (c) `ei.RxC`: The $R \times C$ EI model (see Section 12.5) estimates a hierarchical Multinomial-Dirichlet EI model for contingency tables with more than 2 rows or columns.

4.3 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.

4.3.1 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.

4.3.2 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 3.2.2.

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:

[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.

```
> 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)
```

Chapter 5

Graphing Commands

R, and thus Zelig, can produce exceptionally beautiful plots. Many built-in plotting functions exist, including scatter plots, line charts, histograms, bar charts, pie charts, ternary diagrams, contour plots, and a variety of three-dimensional graphs. If you desire, you can exercise a high degree of control to generate just the right graphic. Zelig includes several default plots for one-observation simulations for each model. To view these plots on-screen, simply type `plot(s.out)`, where `s.out` is the output from `sim()`. Depending on the model chosen, `plot()` will return different plots.

If you wish to create your own plots, this section reviews the most basic procedures for creating and saving two-dimensional plots. R plots material in two steps:

1. You must call an output device (discussed in Section 5.3), select a type of plot, draw a plotting region, draw axes, and plot the given data. At this stage, you may also define axes labels, the plot title, and colors for the plotted data. Step one is described in Section 5.1 below.

2. Optionally, you may add points, lines, text, or a legend to the existing plot. These commands are described in Section 5.2.

5.1 Drawing Plots

The most generic plotting command is `plot()`, which automatically recognizes the type of R object(s) you are trying to plot and selects the best type of plot. The most common graphs returned by `plot()` are as follows:

1. If `X` is a variable of length n, `plot(X)` returns a scatter plot of (x_i, i) for $i = 1, \ldots n$. If X is unsorted, this procedure produces a messy graph. Use `plot(sort(X))` to arrange the plotted values of (x_i, i) from smallest to largest.

2. With two numeric vectors `X` and `Y`, both of length n, `plot(X, Y)` plots a scatter plot of each point (x_i, y_i) for $i = 1, \ldots n$. Alternatively, if Z is an object with two vectors, `plot(Z)` also creates a scatter plot.

Optional arguments specific to `plot` include:

- `main` creates a title for the graph, and `xlab` and `ylab` label the x and y axes, respectively. For example,

```
plot(x, y, main = "My Lovely Plot", xlab = "Explanatory Variable",
     ylab = "Dependent Variable")
```

- `type` controls the type of plot you request. The default is `plot(x, y, type = "p")`, but you may choose among the following types:

`"p"`	points
`"l"`	lines
`"b"`	both points and lines
`"c"`	lines drawn up to but not including the points
`"h"`	histogram
`"s"`	a step function
`"n"`	a blank plotting region (with the axes specified)

- If you choose `type = "p"`, R plots open circles by default. You can change the type of point by specifying the `pch` argument. For example, `plot(x, y, type = "p", pch = 19)` creates a scatter-plot of filled circles. Other options for `pch` include:

19	solid circle (a disk)
20	smaller solid circle
21	circle
22	square
23	diamond
24	triangle pointed up
25	triangle pointed down

  In addition, you can specify your own symbols by using, for example, `pch = "*"` or `pch = "."`.

- If you choose `type = "l"`, R plots solid lines by default. Use the optional `lty` argument to set the line type. For example, `plot(x, y, type = "l", lty = "dashed")` plots a dashed line. Other options are dotted, dotdash, longdash, and twodash.

- `col` sets the color of the points, lines, or bars. For example, `plot(x, y, type = "b", pch = 20, lty = "dotted", col = "violet")` plots small circles connected by a dotted line, both of which are violet. (The axes and labels remain black.) Use `colors()` to see the full list of available colors.

- `xlim` and `ylim` set the limits to the x-axis and y-axis. For example, `plot(x, y, xlim = c(0, 25), ylim = c(-15, 5))` sets range of the x-axis to $[0, 25]$ and the range of the y-axis to $[-15, 5]$.

For additional plotting options, refer to `help(par)`.

5.2 Adding Points, Lines, and Legends to Existing Plots

Once you have created a plot, you can *add* points, lines, text, or a legend. To place each of these elements, R uses coordinates defined in terms of the x-axes and y-axes of the plot area, not coordinates defined in terms of the the plotting window or device. For example, if your plot has an x-axis with values between $[0, 100]$, and a y-axis with values between $[50, 75]$, you may add a point at $(55, 55)$.

- **points()** plots one or more sets of points. Use `pch` with `points` to add points to an existing plot. For example, `points(P, Q, pch = ".", col = "forest green")` plots each (p_i, q_i) as tiny green dots.

- **lines()** joins the specified points with line segments. The arguments `col` and `lty` may also be used. For example, `lines(X, Y, col = "blue", lty = "dotted")` draws a blue dotted line from each set of points (x_i, y_i) to the next. Alternatively, `lines` also takes command output which specifies (x, y) coordinates. For example, `density(Z)` creates a vector of x and a vector of y, and `plot(density(Z))` draws the kernel density function.

- **text()** adds a character string at the specified set of (x, y) coordinates. For example, `text(5, 5, labels = "Key Point")` adds the label "Key Point" at the plot location $(5, 5)$. You may also choose the font using the `font` option, the size of the font relative to the axis labels using the `cex` option, and choose a color using the `col` option. The full list of options may be accessed using `help(text)`.

- **legend()** places a legend at a specified set of (x, y) coordinates. Type `demo(vertci)` to see an example for `legend()`.

5.3 Saving Graphs to Files

By default, R displays graphs in a window on your screen. To save R plots to file (to include them in a paper, for example), preface your plotting commands with:

```
> ps.options(family = c("Times"), pointsize = 12)
> postscript(file = "mygraph.eps", horizontal = FALSE, paper = "special",
             width = 6.25, height = 4)
```

where the `ps.options()` command sets the font type and size in the output file, and the `postscript` command allows you to specify the name of the file as well as several additional options. Using `paper = special` allows you to specify the width and height of the encapsulated postscript region in inches (6.25 inches long and 4 inches high, in this case), and the statement `horizontal = FALSE` suppresses R's default landscape orientation. Alternatively, you may use `pdf()` instead of `postscript()`. If you wish to select postscript options for .pdf output, you may do so using options in `pdf()`. For example:

```
> pdf(file = "mygraph.pdf", width = 6.25, height = 4, family = "Times",
+     pointsize = 12)
```

At the end of every plot, you should close your output device. The command `dev.off()` stops writing and saves the `.eps` or `.pdf` file to your working directory. If you forget to close the file, you will write all subsequent plots to the same file, overwriting previous plots. You may also use `dev.off()` to close on-screen plot windows.

To write multiple plots to the same file, you can use the following options:

- For plots on separate pages in the same `.pdf` document, use

  ```
  > pdf(file = "mygraph.pdf", width = 6.25, height = 4, family = "Times",
  +     pointsize = 12, onefile = TRUE)
  ```

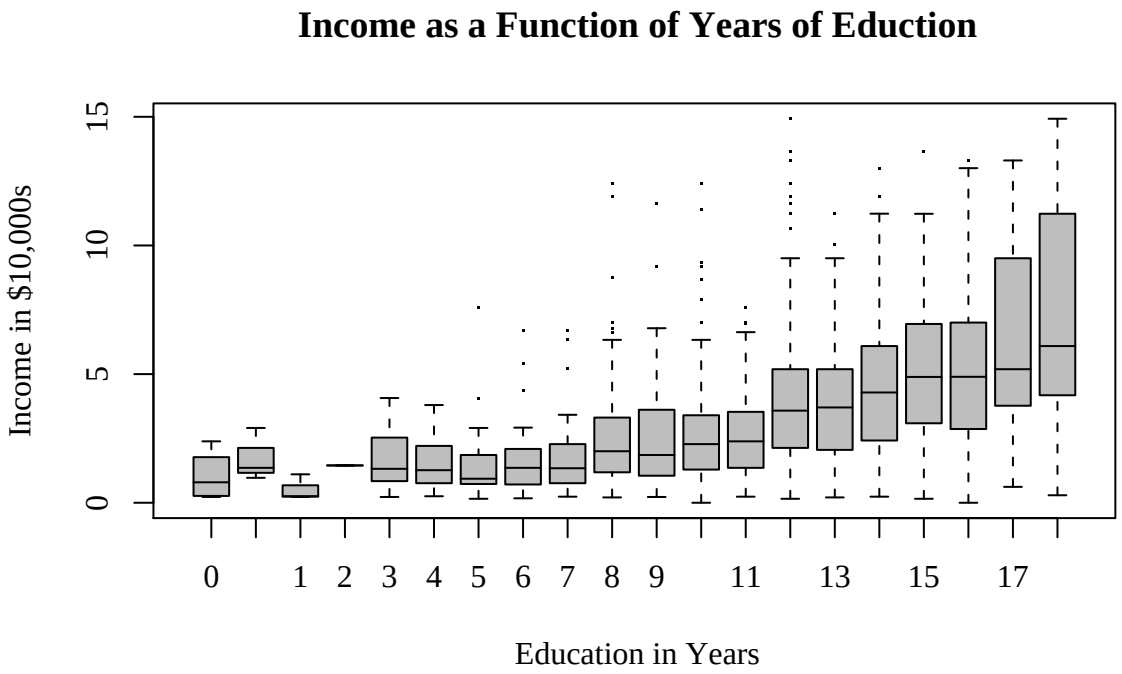
- For multiple plots on one page, initialize either a `.pdf` or `.eps` file, then (before any plotting commands) type:

  ```
  par(mfrow = c(2, 4))
  ```

  This creates a grid that has two rows and four columns. Your plot statements will populate the grid going across the first row, then the second row, from left to right.

5.4 Examples

5.4.1 Descriptive Plots: Box-plots

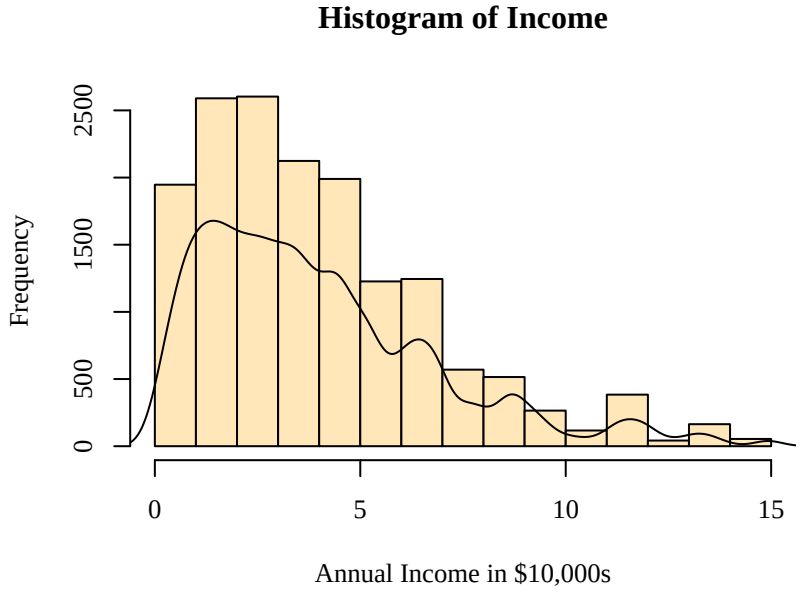


Using the sample `turnout` data set included with Zelig, the following commands will produce the graph above.

```
> library(Zelig)                                # Loads the Zelig package.
> data (turnout)                                # Loads the sample data.
> boxplot(income ~ educate,                     # Creates a boxplot with income
+  data = turnout, col = "grey", pch = ".",     #  as a function of education.
+  main = "Income as a Function of Years of Education",
+  xlab = "Education in Years", ylab = "Income in \$10,000s")
```

5.4.2 Density Plots: A Histogram

Histograms are easy ways to evaluate the density of a quantity of interest.



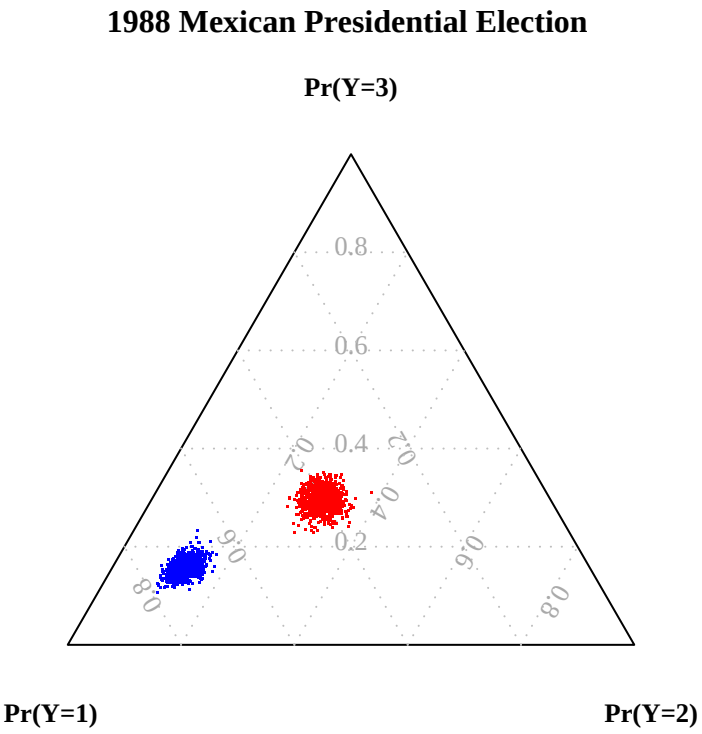
Here's the code to create this graph:

```
> library(Zelig)                              # Loads the Zelig package.
> data(turnout)                               # Loads the sample data set.
> truehist(turnout$income,  col = "wheat1",   # Calls the main plot, with
+      xlab = "Annual Income in $10,000s",    #  options.
+      main = "Histogram of Income")
> lines(density(turnout$income))              # Adds the kernel density line.
```

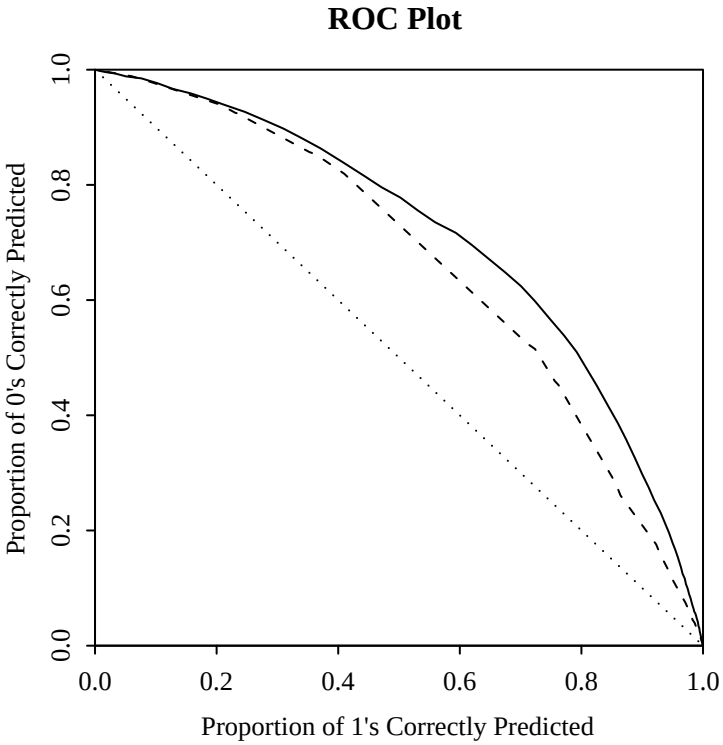
5.4.3 Advanced Examples

The examples above are simple examples which only skim the surface of R's plotting potential. We include more advanced, model-specific plots in the Zelig demo scripts, and have created functions that automate some of these plots, including:

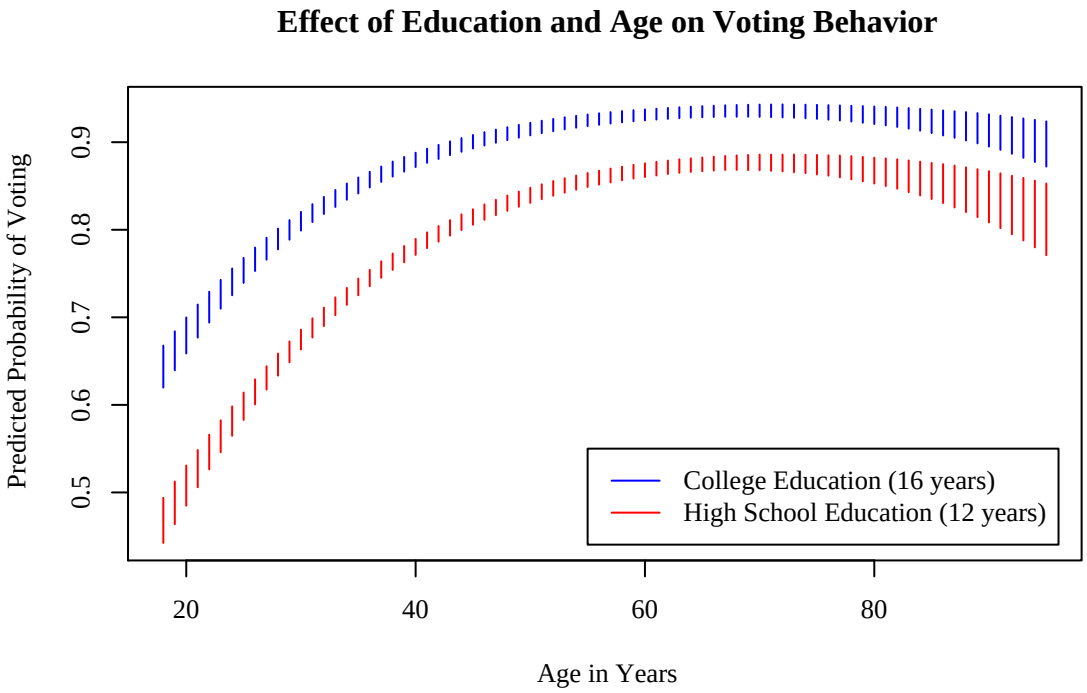
1. **Ternary Diagrams** describe the predicted probability of a categorical dependent variable that has three observed outcomes. You may choose to use this plot with the multinomial logit, the ordinal logit, or the ordinal probit models (Katz and King, 1999). See Section 12.17 for the sample code, type `demo(mlogit)` at the R prompt to run the example, and refer to Section 12.17 to add points to a ternary diagram.



2. **ROC Plots** summarize how well models for binary dependent variables (logit, probit, and relogit) fit the data. The ROC plot evaluates the fraction of 0's and 1's correctly predicted for every possible threshold value at which the continuous Prob($Y = 1$) may be realized as a dichotomous prediction. The closer the ROC curve is to the upper right corner of the plot, the better the fit of the model specification (King and Zeng, 2002*b*). See Section 3 for the sample code, and type `demo(roc)` at the R prompt to run the example.



3. **Vertical Confidence Intervals** may be used for almost any model, and describe simulated confidence intervals for any quantity of interest while allowing one of the explanatory variables to vary over a given range of values (King, Tomz and Wittenberg, 2000). Type `demo(vertci)` at the R prompt to run the example, and `help.zelig(plot.ci)` for the manual page.



Part II

Advanced Zelig Uses

Chapter 6

R Objects

In R, objects can have one or more classes, consisting of the class of the scalar value and the class of the data structure holding the scalar value. Use the `is()` command to determine what an object *is*. If you are already familiar with R objects, you may skip to Section 3.2.2 for loading data, or Section 4.1 for a description of Zelig commands.

6.1 Scalar Values

R uses several classes of scalar values, from which it constructs larger data structures. R is highly class-dependent: certain operations will only work on certain types of values or certain types of data structures. We list the three basic types of scalar values here for your reference:

1. **Numeric** is the default value type for most numbers. An `integer` is a subset of the `numeric` class, and may be used as a `numeric` value. You can perform any type of math or logical operation on numeric values, including:

```
> log(3 * 4 * (2 + pi))          # Note that pi is a built-in constant,
   [1] 4.122270                  #   and log() the natural log function.
> 2 > 3                          # Basic logical operations, including >,
   [1] FALSE                     #   <, >= (greater than or equals),
                                 #   <= (less than or equals), == (exactly
                                 #   equals), and != (not equals).
> 3 >= 2 && 100 == 1000/10       # Advanced logical operations, including
   [1] TRUE                      #   & (and), && (if and only if), | (or),
                                 #   and || (either or).
```

   Note that `Inf` (infinity), `-Inf` (negative infinity), `NA` (missing value), and `NaN` (not a number) are special numeric values on which most math operations will fail. (Logical operations will work, however.)

2. **Logical** operations create logical values of either `TRUE` or `FALSE`. To convert logical values to numerical values, use the `as.integer()` command:

```
> as.integer(TRUE)
   [1] 1
> as.integer(FALSE)
   [1] 0
```

3. **Character** values are text strings. For example,

```
> text <- "supercalafragilisticxpaladocious"
> text
[1] "supercalafragilisticxpaladocious"
```

   assigns the text string on the right-hand side of the `<-` to the named object in your workspace. Text strings are primarily used with data frames, described in the next section. R always returns character strings in quotes.

6.2 Data Structures

6.2.1 Arrays

Arrays are data structures that consist of only one type of scalar value (e.g., a vector of character strings, or a matrix of numeric values). The most common versions, one-dimensional and two-dimensional arrays, are known as *vectors* and *matrices*, respectively.

Ways to create arrays

1. Common ways to create **vectors** (or one-dimensional arrays) include:

```
> a <- c(3, 7, 9, 11)    # Concatenates numeric values into a vector
> a <- c("a", "b", "c")  # Concatenates character strings into a vector
> a <- 1:5               # Creates a vector of integers from 1 to 5 inclusive
> a <- rep(1, times = 5) # Creates a vector of 5 repeated '1's
```

   To manipulate a vector:

```
> a[10]                # Extracts the 10th value from the vector 'a'
> a[5] <- 3.14         # Inserts 3.14 as the 5th value in the vector 'a'
> a[5:7] <- c(2, 4, 7) # Replaces the 5th through 7th values with 2, 4, and 7
```

   Unlike larger arrays, vectors can be extended without first creating another vector of the correct length. Hence,

```
> a <- c(4, 6, 8)
> a[5] <- 9       # Inserts a 9 in the 5th position of the vector,
                  #  automatically inserting an 'NA' values position 4
```

2. A **factor vector** is a special type of vector that allows users to create j indicator variables in one vector, rather than using j dummy variables (as in Stata or SPSS). R creates this special class of vector from a pre-existing vector `x` using the `factor()` command, which separates `x` into levels based on the discrete values observed in `x`. These values may be either integer value or character strings. For example,

```
> x <- c(1, 1, 1, 1, 1, 2, 2, 2, 2, 9, 9, 9, 9)
> factor(x)
   [1] 1 1 1 1 1 2 2 2 2 9 9 9 9
   Levels: 1 2 9
```

   By default, `factor()` creates unordered factors, which are treated as discrete, rather than ordered, levels. Add the optional argument `ordered = TRUE` to order the factors in the vector:

```
> x <- c("like", "dislike", "hate", "like", "don't know", "like", "dislike")
> factor(x, levels = c("hate", "dislike", "like", "don't know"),
+        ordered = TRUE)
  [1] like    dislike    hate    like    don't know    like    dislike
Levels: hate < dislike < like < don't know
```

   The `factor()` command orders the levels according to the order in the optional argument `levels`. If you omit the levels command, R will order the values as they occur in the vector. Thus, omitting the `levels` argument sorts the levels as `like < dislike < hate < don't know` in the example above. If you omit one or more of the levels in the list of levels, R returns levels values of `NA` for the missing level(s):

```
> factor(x, levels = c("hate", "dislike", "like"), ordered = TRUE)
  [1] like    dislike hate    like    <NA>    like    dislike
Levels: hate < dislike < like
```

   Use factored vectors within data frames for plotting (see Section 5.1), to set the values of the explanatory variables using `setx` (see Section 10) and in the ordinal logit and multinomial logit models (see Section 4.2).

3. Build **matrices** (or two-dimensional arrays) from vectors (one-dimensional arrays). You can create a matrix in two ways:

   (a) From a vector: Use the command `matrix(vector, nrow =` k`, ncol =` n`)` to create a $k \times n$ matrix from the vector by filling in the columns from left to right. For example,

```
> matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)
        [,1] [,2] [,3]       # Note that when assigning a vector to a
   [1,]    1    3    5       #  matrix, none of the rows or columns
   [2,]    2    4    6       #  have names.
```

(b) From two or more vectors of length k: Use `cbind()` to combine n vectors vertically to form a $k \times n$ matrix, or `rbind()` to combine n vectors horizontally to form a $n \times k$ matrix. For example:

```
> x <- c(11, 12, 13)         # Creates a vector 'x' of 3 values.
> y <- c(55, 33, 12)         # Creates another vector 'y' of 3 values.
> rbind(x, y)                # Creates a 2 x 3 matrix.  Note that row
     [,1] [,2] [,3]          #  1 is named x and row 2 is named y,
   x   11   12   13          #  according to the order in which the
   y   55   33   12          #  arguments were passed to rbind().
> cbind(x, y)                # Creates a 3 x 2 matrix.  Note that the
         x  y                #  columns are named according to the
   [1,] 11 55                #  order in which they were passed to
   [2,] 12 33                #  cbind().
   [3,] 13 12
```

R supports a variety of matrix functions, including: `det()`, which returns the matrix's determinant; `t()`, which transposes the matrix; `solve()`, which inverts the the matrix; and `%*%`, which multiplies two matricies. In addition, the `dim()` command returns the dimensions of your matrix. As with vectors, square brackets extract specific values from a matrix and the assignment mechanism `<-` replaces values. For example:

```
> loo[,3]        # Extracts the third column of loo.
> loo[1,]         # Extracts the first row of loo.
> loo[1,3] <- 13          # Inserts 13 as the value for row 1, column 3.
> loo[1,] <- c(2,2,3)     # Replaces the first row of loo.
```

If you encounter problems replacing rows or columns, make sure that the `dims()` of the vector matches the `dims()` of the matrix you are trying to replace.

4. An **n-dimensional array** is a set of stacked matrices of identical dimensions. For example, you may create a three dimensional array with dimensions (x, y, z) by stacking z matrices each with x rows and y columns.

```
> a <- matrix(8, 2, 3)       # Creates a 2 x 3 matrix populated with 8's.
> b <- matrix(9, 2, 3)       # Creates a 2 x 3 matrix populated with 9's.
> array(c(a, b), c(2, 3, 2)) # Creates a 2 x 3 x 2 array with the first
   , , 1                     #  level [,,1] populated with matrix a (8's),
                             #  and the second level [,,2] populated
```

```
     [,1] [,2] [,3]       #  with matrix b (9's).
[1,]    8    8    8
[2,]    8    8    8       # Use square brackets to extract values.  For
                          #  example, [1, 2, 2] extracts the second
, , 2                     #  value in the first row of the second level.
                          # You may also use the <- operator to
     [,1] [,2] [,3]       #  replace values.
[1,]    9    9    9
[2,]    9    9    9
```

If an array is a one-dimensional vector or two-dimensional matrix, R will treat the array using the more specific method.

Three functions especially helpful for arrays:

- `is()` returns both the type of scalar value that populates the array, as well as the specific type of array (vector, matrix, or array more generally).
- `dims()` returns the size of an array, where

  ```
  > dims(b)
   [1]  33  5
  ```

  indicates that the array is two-dimensional (a matrix), and has 33 rows and 5 columns.
- The single bracket `[ ]` indicates specific values in the array. Use commas to indicate the index of the specific values you would like to pull out or replace:

  ```
  > dims(a)
   [1]  14
  > a[10]       # Pull out the 10th value in the vector 'a'
  > dims(b)
   [1]  33  5
  > b[1:12, ]   # Pull out the first 12 rows of 'b'
  > c[1, 2]     # Pull out the value in the first row, second column of 'c'
  > dims(d)
   [1]  1000  4  5
  > d[ , 3, 1]  # Pulls out a vector of 1,000 values
  ```

6.2.2 Lists

Unlike arrays, which contain only one type of scalar value, lists are flexible data structures that can contain heterogeneous value types and heterogeneous data structures. Lists are so flexible that one list can contain another list. For example, the list `output` can contain `coef`,

a vector of regression coefficients; `variance`, the variance-covariance matrix; and another list `terms` that describes the data using character strings. Use the `names()` function to view the named elements in a list, and to extract a named element, use

```
> names(output)
 [1] coefficients   variance   terms
> output$coefficients
```

For lists where the elements are not named, use double square brackets `[[ ]]` to extract elements:

```
> L[[4]]      # Extracts the 4th element from the list 'L'
> L[[4]] <- b # Replaces the 4th element of the list 'L' with a matrix 'b'
```

Like vectors, lists are flexible data structures that can be extended without first creating another list of with the correct number of elements:

```
> L <- list()                        # Creates an empty list
> L$coefficients <- c(1, 4, 6, 8)  # Inserts a vector into the list, and
                                     #  names that vector 'coefficients'
   #  within the list
> L[[4]] <- c(1, 4, 6, 8)          # Inserts the vector into the 4th position
                                     #  in the list.  If this list doesn't
                                     #  already have 4 elements, the empty
                                     #  elements will be 'NULL' values
```

Alternatively, you can easily create a list using objects that already exist in your workspace:

```
> L <- list(coefficients = k, variance = v) # Where 'k' is a vector and
                                            #   'v' is a matrix
```

6.2.3 Data Frames

A data frame (or data set) is a special type of list in which each variable is constrained to have the same number of observations. A data frame may contain variables of different types (numeric, integer, logical, character, and factor), so long as each variable has the same number of observations.

Thus, a data frame can use both matrix commands and list commands to manipulate variables and observations.

```
> dat[1:10,]         # Extracts observations 1-10 and all associated variables
> dat[dat$grp == 1,] # Extracts all observations that belong to group 1
> group <- dat$grp   # Saves the variable 'grp' as a vector 'group' in
                     #   the workspace, not in the data frame
> var4 <- dat[[4]]   # Saves the 4th variable as a 'var4' in the workspace
```

For a comprehensive introduction to data frames and recoding data, see Section 3.2.2.

6.2.4 Identifying Objects and Data Structures

Each data structure has several *attributes* which describe it. Although these attributes are normally invisible to users (e.g., not printed to the screen when one types the name of the object), there are several helpful functions that display particular attributes:

- For arrays, `dims()` returns the size of each dimension.
- For arrays, `is()` returns the scalar value type and specific type of array (vector, matrix, array). For more complex data structures, `is()` returns the default methods (classes) for that object.
- For lists and data frames, `names()` returns the variable names, and `str()` returns the variable names and a short description of each element.

For almost all data types, you may use `summary()` to get summary statistics.

Chapter 7

Programming Statements

This chapter introduces the main programming commands. These include functions, if-else statements, for-loops, and special procedures for managing the inputs to statistical models.

7.1 Functions

Functions are either built-in or user-defined sets of encapsulated commands which may take any number of arguments. Preface a function with the `function` statement and use the `<-` operator to assign functions to objects in your workspace.

You may use functions to run the same procedure on different objects in your workspace. For example,

```
check <- function(p, q) {
 result <- (p - q)/q
 result
 }
```

is a simple function with arguments `p` and `q` which calculates the difference between the ith elements of the vector `p` and the ith element of the vector `q` as a proportion of the ith element of `q`, and returns the resulting vector. For example, `check(p = 10, q = 2)` returns 4. You may omit the descriptors as long as you keep the arguments in the correct order: `check(10, 2)` also returns 4. You may also use other objects as inputs to the function. If `again = 10` and `really = 2`, then `check(p = again, q = really)` and `check(again, really)` also returns 4.

Because functions run commands as a set, you should make sure that each command in your function works by testing each line of the function at the R prompt.

7.2 If-Statements

Use `if` (and optionally, `else`) to control the flow of R functions. For example, let `x` and `y` be scalar numerical values:

```
if (x == y) {                # If the logical statement in the ()'s is true,
  x <- NA                    #  then 'x' is changed to 'NA' (missing value).
}
else {                       # The 'else' statement tells R what to do if
  x <- x^2                   #  the if-statement is false.
}
```

As with a function, use { and } to define the set of commands associated with each if and else statement. (If you include if statements inside functions, you may have multiple sets of nested curly braces.)

7.3 For-Loops

Use `for` to repeat (loop) operations. Avoiding loops by using matrix or vector commands is usually faster and more elegant, but loops are sometimes necessary to assign values. If you are using a loop to assign values to a data structure, you must first initialize an empty data structure to hold the values you are assigning.

Select a data structure compatible with the type of output your loop will generate. If your loop generates a scalar, store it in a vector (with the ith value in the vector corresponding to the the ith run of the loop). If your loop generates vector output, store them as rows (or columns) in a matrix, where the ith row (or column) corresponds to the ith iteration of the loop. If your output consists of matrices, stack them into an array. For list output (such as regression output) or output that changes dimensions in each iteration, use a list. To initialize these data structures, use:

```
> x <- vector()                          # An empty vector of any length.
> x <- list()                            # An empty list of any length.
```

The `vector()` and `list()` commands create a vector or list of any length, such that assigning `x[5] <- 15` automatically creates a vector with 5 elements, the first four of which are empty values (`NA`). In contrast, the `matrix()` and `array()` commands create data structures that are restricted to their original dimensions.

```
> x <- matrix(nrow = 5, ncol = 2)  # A matrix with 5 rows and 2 columns.
> x <- array(dim = c(5,2,3))       # A 3D array of 3 stacked 5 by 2 matrices.
```

If you attempt to assign a value at $(100, 200, 20)$ to either of these data structures, R will return an error message ("subscript is out of bounds"). R does not automatically extend the dimensions of either a matrix or an array to accommodate additional values.

Example 1: Creating a vector with a logical statement

```
x <- array()             # Initializes an empty data structure.
for (i in 1:10) {        # Loops through every value from 1 to 10, replacing
```

```
  if (is.integer(i/2)) { #  the even values in 'x' with i+5.
    x[i] <- i + 5
  }
}                          # Enclose multiple commands in {}.
```

You may use `for()` inside or outside of functions.

Example 2: Creating dummy variables by hand You may also use a loop to create a matrix of dummy variables to append to a data frame. For example, to generate fixed effects for each state, let's say that you have `mydata` which contains `y`, `x1`, `x2`, `x3`, and `state`, with `state` a character variable with 50 unique values. There are three ways to create dummy variables: 1) with a built-in R command; 2) with one loop; or 3) with 2 for loops.

1. R will create dummy variables on the fly from a single variable with distinct values.

   ```
   > z.out <- zelig(y ~ x1 + x2 + x3 + as.factor(state),
                    data = mydata, model = "ls")
   ```

   This method returns $k - 1$ indicators for k states.

2. Alternatively, you can use a loop to create dummy variables by hand. There are two ways to do this, but both start with the same initial commands. Using vector commands, first create an index of for the states, and initialize a matrix to hold the dummy variables:

   ```
   idx <- sort(unique(mydata$state))
   dummy <- matrix(NA, nrow = nrow(mydata), ncol = length(idx))
   ```

   Now choose between the two methods.

   (a) The first method is computationally inefficient, but more intuitive for users not accustomed to vector operations. The first loop uses `i` as in index to loop through all the rows, and the second loop uses `j` to loop through all 50 values in the vector `idx`, which correspond to columns 1 through 50 in the matrix `dummy`.

   ```
   for (i in 1:nrow(mydata)) {
     for (j in 1:length(idx)) {
       if (mydata$state[i,j] == idx[j]) {
         dummy[i,j] <- 1
       }
       else {
         dummy[i,j] <- 0
       }
     }
   }
   ```

Then add the new matrix of dummy variables to your data frame:

```
names(dummy) <- idx
mydata <- cbind(mydata, dummy)
```

(b) As you become more comfortable with vector operations, you can replace the double loop procedure above with one loop:

```
for (j in 1:length(idx)) {
  dummy[,j] <- as.integer(mydata$state == idx[j])
}
```

The single loop procedure evaluates each element in `idx` against the vector `mydata$state`. This creates a vector of n `TRUE`/`FALSE` observations, which you may transform to `1`'s and `0`'s using `as.integer()`. Assign the resulting vector to the appropriate column in `dummy`. Combine the `dummy` matrix with the data frame as above to complete the procedure.

Example 3: Weighted regression with subsets Selecting the `by` option in `zelig()` partitions the data frame and then automatically loops the specified model through each partition. Suppose that `mydata` is a data frame with variables `y`, `x1`, `x2`, `x3`, and `state`, with `state` a factor variable with 50 unique values. Let's say that you would like to run a weighted regression where each observation is weighted by the inverse of the standard error on `x1`, estimated for that observation's state. In other words, we need to first estimate the model for each of the 50 states, calculate 1 / SE($\texttt{x1}_j$) for each state $j = 1, \ldots, 50$, and then assign these weights to each observation in `mydata`.

- Estimate the model separate for each state using the `by` option in `zelig()`:

  ```
  z.out <- zelig(y ~ x1 + x2 + x3, by = "state", data = mydata, model = "ls")
  ```

  Now `z.out` is a list of 50 regression outputs.

- Extract the standard error on `x1` for each of the state level regressions.

  ```
  se <- array()                           # Initalize the empty data structure.
  for (i in 1:50) {                       # vcov() creates the variance matrix
    se[i] <- sqrt(vcov(z.out[[i]])[2,2]) # Since we have an intercept, the 2nd
  }                                       # diagonal value corresponds to x1.
  ```

- Create the vector of weights.

  ```
  wts <- 1 / se
  ```

  This vector `wts` has 50 values that correspond to the 50 sets of state-level regression output in `z.out`.

- To assign the vector of weights to each observation, we need to match each observation's state designation to the appropriate state. For simplicity, assume that the states are numbered 1 through 50.

```
mydata$w <- NA            # Initalizing the empty variable
for (i in 1:50) {
  mydata$w[mydata$state == i] <- wts[i]
}
```

We use `mydata$state` as the index (inside the square brackets) to assign values to `mydata$w`. Thus, whenever state equals 5 for an observation, the loop assigns the fifth value in the vector `wts` to the variable `w` in `mydata`. If we had 500 observations in `mydata`, we could use this method to match each of the 500 observations to the appropriate `wts`.

If the states are character strings instead of integers, we can use a slightly more complex version

```
mydata$w <- NA
idx <- sort(unique(mydata$state))
for (i in 1:length(idx) {
  mydata$w[mydata$state == idx[i]] <- wts[i]
}
```

- Now we can run our weighted regression:

```
z.wtd <- zelig(y ~ x1 + x2 + x3, weights = w, data = mydata,
               model = "ls")
```

Chapter 8

Writing New Models

With Zelig, writing a new model in R is straightforward. (If you already have a model, see Chapter 9 for how to include it in Zelig.) With tools to streamline user inputs, writing a new model does not require a lot of programming knowledge, but lets developers focus on the model's math. Generally, writing a new statistical procedure or model comes in orderly steps:

1. Write down the mathematical model. Define the parameters that you need, grouping parameters into convenient vectors or matrices whenever possible (this will make your code clearer).

2. Write the code.

3. Test the code (usually using Monte Carlo data, where you know the true values being estimated) and make sure that it works as expected.

4. Write some documentation explaining your model and the functions that run your model.

Somewhere between steps [1] and [2], you will need to translate input data into the mathematical notation that you used to write down the model. Rather than repeating whole blocks of code, use functions to streamline the number of commands that users will need to run your model.

With more steps being performed by fewer commands, the inputs to these commands become more sophisticated. The structure of those inputs actually matters quite a lot. If your function has a convoluted syntax, it will be difficult to use, difficult to explain, and difficult to document. If your function is easy to use and has an intuitive syntax, however, it will be easy to explain and document, which will make your procedure more accessible to all users.

8.1 Managing Statistical Model Inputs

Most statistical models require a matrix of explanatory variables and a matrix of dependent variables. Rather than have users create matrices themselves, R has a convenient user interface to create matrices of response and explanatory variables on the fly. Users simply specify a `formula` in the form of `dependent ~ explanatory variables`, and developers use the following functions to transform the formula into the appropriate matrices. Let `mydata` be a data frame.

```
> formula <- y ~ x1 + x2                    # User input

# Given the formula above, programmers can use the following standard commands
> D <- model.frame(formula, data = mydata) # Subset & listwise deletion
> X <- model.matrix(formula, data = D)     # Creates X matrix
> Y <- model.response(D)                   # Creates Y matrix
```

where

- `D` is a subset of `mydata` that contains only the variables specified in the formula (`y`, `x1`, and `x2`) with listwise deletion performed on the subset data frame;
- `X` is a matrix that contains a column of 1's, and the explanatory variables `x1` and `x2` from `D`; and
- `Y` is a matrix containing the dependent variable(s) from `D`.

Depending on the model, `Y` may be a column vector, matrix, or other data structure.

8.1.1 Describe the Statistical Model

After setting up the X matrix, the next step for most models will be to identify the corresponding vector of parameters. For a single response variable model with no ancillary parameters, the standard R interface is quite convenient: given X, the model's parameters are simply β.

There are very few models, however, that fall into this category. Even Normal regression, for example, has two sets of parameters β and σ^2. In order to make the R formula format more flexible, Zelig has an additional set of tools that lets you describe the inputs to your model (for multiple sets of parameters).

After you have written down the statistical model, identify the parameters in your model. With these parameters in mind, the first step is to write a `describe.*()` function for your model. If your model is called `mymodel`, then the `describe.mymodel()` function takes no arguments and returns a list with the following information:

- `category`: a character string that describes the dependent variable. See Section 13.1 for the current list of available categories.

- `parameters`: a list containing parameter sets used in your model. For each parameter (e.g., theta), you need to provide the following information:
  - `equations`: an integer number of equations for the parameter. For parameters that can take, for example, two to four equations, use `c(2, 4)`.
  - `tagsAllowed`: a logical value (`TRUE`/`FALSE`) specifying whether a given parameter allows constraints.
  - `depVar`: a logical value (`TRUE`/`FALSE`) specifying whether a parameter requires a corresponding dependent variable.
  - `expVar`: a logical value (`TRUE`/`FALSE`) specifying whether a parameter allows explanatory variables.

(See Section 13.1 for examples and additional arguments output by `describe.mymodel()`.)

8.1.2 Single Response Variable Models: Normal Regression Model

Let's say that you are trying to write a Normal regression model with stochastic component

$$\text{Normal}(y_i \mid \mu_i, \sigma^2) \;=\; \frac{1}{\sqrt{2\pi}\sigma}\,\exp\left(-\left(\frac{(y_i-\mu_i)^2}{2\sigma^2}\right)\right)$$

with scalar variance parameter $\sigma^2 > 0$, and systematic component $E(Y_i) = \mu_i = x_i\beta$. This implies two sets of parameters in your model, and the following `describe.normal.regression()` function:

```
describe.normal.regression <- function() {
  category <- "continuous"
  mu <- list(equations = 1,              # Systematic component
             tagsAllowed = FALSE,
             depVar = TRUE,
             expVar = TRUE)
  sigma2 <- list(equations = 1,          # Scalar ancillary parameter
                 tagsAllowed = FALSE,
                 depVar = FALSE,
                 expVar = FALSE)
  pars <- list(mu = mu, sigma2 = sigma2)
  list(category = category, parameters = pars)
}
```

To find the log-likelihood:

$$\begin{aligned}
\mathrm{L}\ (\beta, \sigma^2 \mid y) &= \prod_{1=1}^{n} \mathrm{Normal}(y_i \mid \mu_i, \sigma^2) \\
&= \prod_{i=1}^{n} (2\pi\sigma^2)^{-1/2} \exp\left(\frac{-(y_i - \mu_i)^2}{2\sigma^2}\right) \\
&= (2\pi\sigma^2)^{-n/2} \prod_{i=1}^{n} \exp\left(\frac{-(y_i - \mu_i)^2}{2\sigma^2}\right) \\
&= (2\pi\sigma^2)^{-n/2} \prod_{i=1}^{n} \exp\left(\frac{-(y_i - x_i\beta)^2}{2\sigma^2}\right) \\
\ln \mathrm{L}\ (\beta, \sigma^2 \mid y) &= -\frac{n}{2} \ln(2\pi\sigma^2) - \sum_{i=1}^{n} \frac{(y_i - x_i\beta)^2}{2\sigma^2} \\
&= -\frac{n}{2} \ln(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - x_i\beta)^2 \\
&\propto -\frac{1}{2} \left(n \ln \sigma^2 + \frac{\sum_{i=1}^{n} (y_i - x_i\beta)^2}{\sigma^2}\right)
\end{aligned}$$

In R code, this translates to:

```
ll.normal <- function(par, X, Y, n, terms) {
  beta <- parse.par(par, terms, eqn = "mu")             # [1]
  gamma <- parse.par(par, terms, eqn = "sigma2")        # [2]
  sigma2 <- exp(gamma)
  -0.5 * (n * log(sigma2) + sum((Y - X %*% beta)^2 / sigma2))
}
```

At Comment [1] above, we use the function `parse.par()` to pull out the vector of parameters `beta` (which relate the systematic component μ_i to the explanatory variables x_i). No matter how many covariates there are, the `parse.par()` function can use `terms` to pull out the appropriate parameters from `par`. We also use `parse.par()` at Comment [2] to pull out the scalar ancillary parameter that (after transformation) corresponds to the σ^2 parameter.

To optimize this function, simply type:

```
out <- optim(start.val, ll.normal, control = list(fnscale = -1),
             method = "BFGS", hessian = TRUE, X = X, Y = Y, terms = terms)
```

where

- `start.val` is a vector of starting values for `par`. Use `set.start()` to create starting values for all parameters, systematic and ancillary, in one step.
- `ll.normal` is the log-likelihood function derived above.

- `"BFGS"` specifies unconstrained optimization using a quasi-Newton method.
- `control = list(fnscale = -1)` specifies that R should maximize the function (omitting this causes R to minimize the function by default).
- `hessian = TRUE` instructs R to return the Hessian matrix (from which you may calculate the variance-covariance matrix).
- `X` and `Y` are the matrix of explanatory variables and vector of dependent variables, used in the `ll.normal()` function.
- `terms` are meta-data constructed from the `model.frame()` command.

Please refer to the R-help for `optim()` for more options.

To make this procedure generalizable, we can write a function that takes a user-specified data frame and formula, and optional starting values for the optimization procedure:

```
normal.regression <- function(formula, data, start.val = NULL, ...) {

  fml <- parse.formula(formula, model = "normal.regression") # [1]
  D <- model.frame(fml, data = data)
  X <- model.matrix(fml, data = D)
  Y <- model.response(D)
  terms <- attr(D, "terms")
  n <- nrow(X)

  start.val <- set.start(start.val, terms)

  res <- optim(start.val, ll.normal, method = "BFGS",
               hessian = TRUE, control = list(fnscale = -1),
               X = X, Y = Y, n = n, terms = terms, ...)      # [2]

  fit <- model.end(res, D)                                   # [3]
  fit$n <- n
  class(fit) <- "normal"                                     # [4]
  fit
}
```

The following comments correspond to the bracketed numbers above:

1. The `parse.formula()` command looks for the `describe.normal.regression()` function, which changes the user-specified formula into the following format:

   ```
   list(mu = formula,          # where 'formula' was specified by the user
        sigma = ~ 1)
   ```

2. The `...` here indicate that if the user enters any additional arguments when calling `normal.regression()`, that those arguments should go to the `optim()` function.

3. The `model.end()` function takes the optimized output and the listwise deleted data frame `D` and creates an object that will work with `setx()`.

4. Choose a class for your model output so that you will be able to write an appropriate `summary()`, `param()`, and `qi()` function for your model.

8.1.3 Multivariate models: Bivariate Normal example

Most common models have one systematic component. For n observations, the systematic component varies over observations $i = 1, \ldots, n$. In the case of the Normal regression model, the systematic component is μ_i (σ^2 is not estimated as a function of covariates).

In some cases, however, your model may have more than one systematic component. In the case of bivariate probit, we have a dependent variable $Y_i = (Y_{i1}, Y_{i2})$ observed as (0,0), (1,0), (0,1), or (1,1) for $i = 1, \ldots, n$. Similar to a single-response probit model, the stochastic component is described by two latent (unobserved) continuous variables (Y_{i1}^*, Y_{i2}^*) which follow the bivariate Normal distribution:

$$\begin{pmatrix} Y_{i1}^* \\ Y_{i2}^* \end{pmatrix} \sim \text{Normal}\left\{ \begin{pmatrix} \mu_{i1} \\ \mu_{i2} \end{pmatrix}, \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \right\},$$

where for $j = 1, 2$, μ_{ij} is the mean for Y_{ij}^* and ρ is a correlation parameter. The following observation mechanism links the observed dependent variables, Y_{ij}, with these latent variables

$$Y_{ij} = \begin{cases} 1 & \text{if } Y_{ij}^* \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

The systemic components for each observation are

$$\begin{aligned} \mu_{ij} &= x_{ij}\beta_j \quad \text{for} \quad j = 1, 2, \\ \rho &= \frac{\exp(x_{i3}\beta_3) - 1}{\exp(x_{i3}\beta_3) + 1}. \end{aligned}$$

In the default specification, ρ is a scalar (such that x_{i3} only contains an intercept term).

If so, we have two sets of parameters: $\mu_i = (\mu_{i1}, \mu_{i2})$ and ρ. This implies the following `describe.bivariate.probit()` function:

```
describe.bivariate.probit <- function() {
  category <- "dichotomous"
  package <- list(name = "mvtnorm",       # Required package and
                  version = "0.7")        #  minimum version number
  mu <- list(equations = 2,               # Systematic component has 2
             tagsAllowed = TRUE,          #  required equations
```

```
            depVar = TRUE,
            expVar = TRUE),
  rho <- list(equations = 1,              # Optional systematic component
            tagsAllowed = FALSE,          #   (estimated as an ancillary
            depVar = FALSE,               #    parameter by default)
            expVar = TRUE),
  pars <- parameters(mu = mu, rho = rho)
  list(category = category, package = package, parameters = pars)
}
```

Since users may choose different explanatory variables to parameterize μ_{i1} and μ_{i2} (and sometimes ρ), the model requires a minimum of *two* formulas. For example,

```
formulae <- list(mu1 = y1 ~ x1 + x2,                          # User input
                 mu2 = y2 ~ x2 + x3)
fml <- parse.formula(formulae, model = "bivariate.probit")   # [1]
D <- model.frame(fml, data = mydata)
X <- model.matrix(fml, data = D)
Y <- model.response(D)
```

At comment [1], `parse.formula()` finds the `describe.bivariate.probit()` function and parses the formulas accordingly.

If ρ takes covariates (and becomes a systematic component rather than an ancillary parameter), there can be three sets of explanatory variables:

```
formulae <- list(mu1 = y1 ~ x1 + x2,
                 mu2 = y2 ~ x2 + x3,
                 rho = ~ x4 + x5)
```

From the perspective of the programmer, a nearly identical framework works for both single and multiple equation models. The (`parse.formula()`) line changes the class of `fml` from `"list"` to `"multiple"` and hence ensures that `model.frame()` and `model.matrix()` go to the appropriate methods. `D`, `X` , and `Y` are analogous to their single equation counterparts above:

- `D` is the subset of `mydata` containing the variables `y1`, `y2`, `x1`, `x2`, and `x3` with listwise deletion performed on the subset;
- `X` is a matrix corresponding to the explanatory variables, in one of three forms discussed below (see Section 8.2).
- `Y` is an $n \times J$ matrix (where $J = 2$ here) with columns (`y1`, `y2`) corresponding to the outcome variables on the left-hand sides of the formulas.

Given for the bivariate probit probability density described above, the likelihood is:

$$L(\pi|Y_i) = \prod_{i=1}^{n} \pi_{00}^{\mathrm{I}\{Y_i=(0,0)\}} \pi_{10}^{\mathrm{I}\{Y_i=(1,0)\}} \pi_{01}^{\mathrm{I}\{Y_i=(0,1)\}} \pi_{11}^{\mathrm{I}\{Y_i=(1,1)\}}$$

where I is an indicator function and

- $\pi_{00} = \int_{-\infty}^{0} \int_{-\infty}^{0} \mathrm{Normal}(Y_{i1}^*, Y_{i2}^* \mid \mu_{i1}, \mu_{i2}, \rho) dY_{i2}^* dY_{i1}^*$
- $\pi_{10} = \int_{0}^{\infty} \int_{-\infty}^{0} \mathrm{Normal}(Y_{i1}^*, Y_{i2}^* \mid \mu_{i1}, \mu_{i2}, \rho) dY_{i2}^* dY_{i1}^*$
- $\pi_{01} = \int_{-\infty}^{0} \int_{0}^{\infty} \mathrm{Normal}(Y_{i1}^*, Y_{i2}^* \mid \mu_{i1}, \mu_{i2}, \rho) dY_{i2}^* dY_{i1}^*$
- $\pi_{11} = 1 - \pi_{00} - \pi_{10} - \pi_{01}$

This implies the following log-likelihood:

$$\begin{aligned} \log L(\pi|Y_i) &= \sum_{i=1}^{n} \mathrm{I}\{Y_i = (0,0)\} \log \pi_{00} + \mathrm{I}\{Y_i = (1,0)\} \log \pi_{10} \\ &\quad + \mathrm{I}\{Y_i = (0,1)\} \log \pi_{01} + \mathrm{I}\{Y_i = (1,1)\} \log \pi_{11} \end{aligned}$$

(For the corresponding R code, see Section 8.2.4 below.)

8.2 Easy Ways to Manage Matrices

Most statistical methods relate explanatory variables x_i to a dependent variable of interest y_i for each observation $i = 1, \ldots, n$. Let β be a set of parameters that correspond to each column in X, which is an $n \times k$ matrix with rows x_i. For a single equation model, the linear predictor is

$$\eta_i = x_i \beta = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_k x_{ik}$$

Thus, η is the set of η_i for $i = 1, \ldots, n$ and is usually represented as an $n \times 1$ matrix.

For a two equation model such as bivariate probit, the linear predictor becomes a matrix with columns corresponding to each dependent variable (y_{1i}, y_{2i}):

$$\eta_i = (\eta_{i1}, \eta_{i2}) = (x_{i1}\beta_1, x_{i2}\beta_2)$$

With η as an $n \times 2$ matrix, we now have a few choices as to how to create the linear predictor:

1. An **intuitive** layout, which stacks matrices of explanatory variables, provides an easy visual representation of the relationship between explanatory variables and coefficients;

2. A **computationally-efficient** layout, which takes advantage of computational vectorization; and

3. A **memory-saving** layout, which reduces the overall size of the X and β matrices.

Using the simple tools described in this section, you can pick the best matrix management method for your model.

In addition, the way in which η is created also affects the way parameters are estimated. Let's say that you want two parameters to have the same effect in different equations. By setting up X and β in a certain way, you can let users set constraints across parameters. Continuing the bivariate probit example above, let the model specification be:

```
formulae <- list(mu1 = y1 ~ x1 + x2 + tag(x3, "land"),
                 mu2 = y2 ~ x3 + tag(x4, "land"))
```

where `tag()` is a special function that constrains variables to have the same effect across equations. Thus, the coefficient for `x3` in equation `mu1` is constrained to be equal to the coefficient for `x4` in equation `mu2`, and this effect is identified as the "land" effect in both equations. In order to consider constraints across equations, the structure of both X and β matter.

8.2.1 The Intuitive Layout

A stacked matrix of X and vector β is probably the most visually intuitive configuration. Let $J = 2$ be the number of equations in the bivariate probit model, and let v_t be the total number of unique covariates in both equations. Choosing `model.matrix(..., shape = "stacked")` yields a $(Jn \times v_t) = (2n \times 6)$ matrix of explanatory variables. Again, let x_1 be an $n \times 1$ vector representing variable `x1`, x_2 `x2`, and so forth. Then

$$X = \begin{pmatrix} 1 & 0 & x_1 & x_2 & 0 & x_3 \\ 0 & 1 & 0 & 0 & x_3 & x_4 \end{pmatrix}$$

Correspondingly, β is a vector with elements

$$(\beta_0^{\mu_1} \ \beta_0^{\mu_2} \ \beta_{x_1}^{\mu_1} \ \beta_{x_2}^{\mu_1} \ \beta_{x_3}^{\mu_2} \ \beta_{\text{land}})\prime$$

where β_0^j are the intercept terms for equation $j = \{\mu_1, \mu_2\}$. Since X is $(2n \times 6)$ and β is (6×1), the resulting linear predictor η is also stacked into a $(2n \times 1)$ matrix. Although difficult to manipulate (since observations are indexed by i and $2i$ for each $i = 1, \ldots, n$ rather than just i), it is easy to see that we have turned the two equations into one big X matrix and one long vector β, which is directly analogous to the familiar single-equation η.

8.2.2 The Computationally-Efficient Layout

Choosing array X and vector β is probably the the most computationally-efficient configuration: `model.matrix(..., shape = "array")` produces an $n \times k_t \times J$ array where J is the total number of equations and k_t is the total number of parameters across all the equations. Since some parameter values may be constrained across equations, $k_t \leq \sum_{j=1}^{J} k_j$. If a

variable is not in a certain equation, it is observed as a vector of 0s. With this option, each $i = 1, \ldots, n$ x_i matrix becomes:

$$\begin{pmatrix} 1 & 0 & x_{i1} & x_{i2} & 0 & x_{i3} \\ 0 & 1 & 0 & 0 & x_{i3} & x_{i4} \end{pmatrix}$$

By stacking each of these x_i matrices along the first dimension, we get X as an array with dimensions $n \times k_t \times J$.

Correspondingly, β is a vector with elements

$$(\beta_0^{\mu_1} \ \beta_0^{\mu_2} \ \beta_{x_1}^{\mu_1} \ \beta_{x_2}^{\mu_1} \ \beta_{x_3}^{\mu_2} \ \beta_{\text{land}})\prime$$

To multiply the X array with dimensions $(n \times 6 \times 2)$ and the (6×1) β vector, we *vectorize* over equations as follows:

```
eta <- apply(X, 3, '%*%', beta)
```

The linear predictor `eta` is therefore a $(n \times 2)$ matrix.

8.2.3 The Memory-Efficient Layout

Choosing a "compact" X matrix and matrix β is probably the most memory-efficient configuration: `model.matrix(..., shape = "compact")` (the default) produces an $n \times v$ matrix, where v is the number of unique variables (5 in this case)[1] in all of the equations. Let x_1 be an $n \times 1$ vector representing variable `x1`, x_2 `x2`, and so forth.

$$X = (1 \ x_1 \ x_2 \ x_3 \ x_4) \qquad \beta = \begin{pmatrix} \beta_0^{\mu_1} & \beta_0^{\mu_2} \\ \beta_{x_1}^{\mu_1} & 0 \\ \beta_{x_2}^{\mu_1} & 0 \\ \beta_{\text{land}} & \beta_{x_3}^{\mu_2} \\ 0 & \beta_{\text{land}} \end{pmatrix}$$

The β_{land} parameter is used twice to implement the constraint, and the number of empty cells is minimized by implementing the constraints in β rather than X. Furthermore, since X is $(n \times 5)$ and β is (5×2), $X\beta = \eta$ is $n \times 2$.

8.2.4 Interchanging the Three Methods

Continuing the bivariate probit example above, we only need to modify a few lines of code to put these different schemes into effect. Using the default (memory-efficient) options, the log-likelihood is:

[1] Why 5? In addition to the intercept term (a variable which is the same in either equation, and so counts only as one variable), the *unique* variables are x_1, x_2, x_3, and x_4.

```
bivariate.probit <- function(formula, data, start.val = NULL, ...) {
  fml <- parse.formula(formula, model = "bivariate.probit")
  D <- model.frame(fml, data = data)
  X <- model.matrix(fml, data = D, eqn = c("mu1", "mu2"))       # [1]
  Xrho <- model.matrix(fml, data = D, eqn = "rho")
  Y <- model.response(D)
  terms <- attr(D, "terms")
  start.val <- set.start(start.val, terms)
  start.val <- put.start(start.val, 1, terms, eqn = "rho")

  log.lik <- function(par, X, Y, terms) {
    Beta <- parse.par(par, terms, eqn = c("mu1", "mu2"))         # [2]
    gamma <- parse.par(par, terms, eqn = "rho")
    rho <- (exp(Xrho %*% gamma) - 1) / (1 + exp(Xrho %*% gamma))
    mu <- X %*% Beta                                             # [3]
    llik <- 0
    for (i in 1:nrow(mu)){
      Sigma <- matrix(c(1, rho[i,], rho[i,], 1), 2, 2)
      if (Y[i,1]==1)
        if (Y[i,2]==1)
          llik <- llik + log(pmvnorm(lower = c(0, 0), upper = c(Inf, Inf),
                                     mean = mu[i,], corr = Sigma))
        else
          llik <- llik + log(pmvnorm(lower = c(0, -Inf), upper = c(Inf, 0),
                                     mean = mu[i,], corr = Sigma))
      else
        if (Y[i,2]==1)
          llik <- llik + log(pmvnorm(lower = c(-Inf, 0), upper = c(0, Inf),
                                     mean = mu[i,], corr = Sigma))
        else
          llik <- llik + log(pmvnorm(lower = c(-Inf, -Inf), upper = c(0, 0),
                                     mean = mu[i,], corr = Sigma))
        }
    return(llik)
  }
  res <- optim(start.val, log.lik, method = "BFGS",
               hessian = TRUE, control = list(fnscale = -1),
               X = X, Y = Y, terms = terms, ...)
  fit <- model.end(res, D)
  class(fit) <- "bivariate.probit"
  fit
}
```

If you find that the default (memory-efficient) method isn't the best way to run your model, you can use either the intuitive option or the computationally-efficient option by changing just a few lines of code as follows:

- **Intuitive option** At Comment [1]:

  ```
  X <- model.matrix(fml, data = D, shape = "stacked", eqn = c("mu1", "mu2"))
  ```

  and at Comment [2],

  ```
  Beta <- parse.par(par, terms, shape = "vector", eqn = c("mu1", "mu2"))
  ```

  The line at Comment [3] remains the same as in the original version.

- **Computationally-efficient option** Replace the line at Comment [1] with

  ```
  X <- model.matrix(fml, data = D, shape = "array", eqn = c("mu1", "mu2"))
  ```

  At Comment [2]:

  ```
  Beta <- parse.par(par, terms, shape = "vector", eqn = c("mu1", "mu2"))
  ```

  At Comment [3]:

  ```
  mu <- apply(X, 3, '%*%', Beta)
  ```

Even if your optimizer calls a C or FORTRAN routine, you can use combinations of `model.matrix()` and `parse.par()` to set up the data structures that you need to obtain the linear predictor (or your model's equivalent) before passing these data structures to your optimization routine.

Chapter 9

Adding Models and Methods to Zelig

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

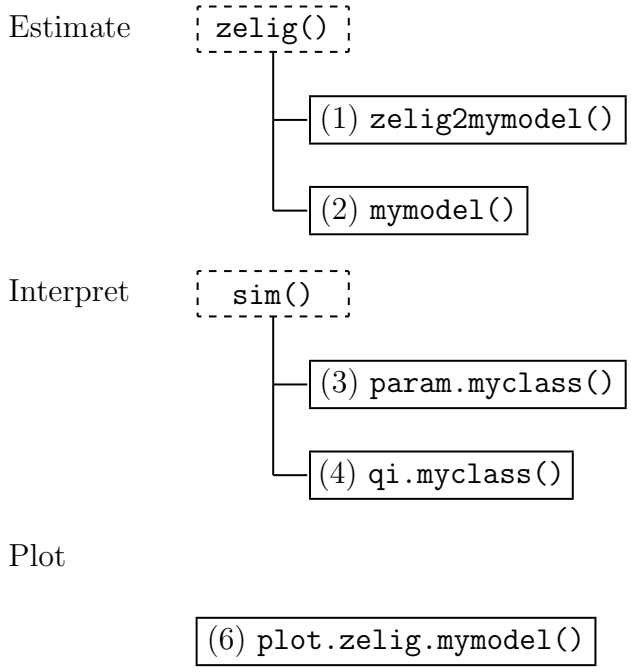
This chapter assumes that your model is written as a function that takes a user-defined formula and data set (see Chapter 8), and returns a list of output that includes (at the very least) the estimated parameters and terms that describe the data used to fit the model. You should choose a class (either S3 or S4 class) for this list of output, and provide appropriate methods for generic functions such as `summary()`, `print()`, `coef()` and `vcov()`.

To add new models to Zelig, you need to provide six R functions, illustrated in Figure 9.1. Let `mymodel` be a new model with class `"myclass"`.

These functions are as follows:

1. `zelig2mymodel()` translates `zelig()` arguments into the arguments for `mymodel()`.
2. `mymodel()` estimates your statistical procedure.
3. `param.myclass()` simulates parameters for your model. Alternatively, if your model's parameters consist of one vector with a correspondingly observed variance-covariance matrix, you may write *two* simple functions to substitute for `param.myclass()`:
   (a) `coef.myclass()` to extract the coefficients from your model output, and
   (b) `vcov.myclass()` to extract the variance-covariance matrix from your model.
4. `qi.myclass()` calculates expected values, simulates predicted values, and generates other quantities of interest for your model (applicable only to models that take explanatory variables).
5. `plot.zelig.mymodel()` to plot the simulated quantities of interest from your model.
6. A **reference manual page** to document the model. (See Section 9.3)
7. A function (`describe.mymodel()`) describing the inputs to your model, for use with a graphical user interface. (See Section 13.1).

Figure 9.1: Six functions (solid boxes) to implement a new Zelig model



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

9.1 Making the Model Compatible with Zelig

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

Zelig's modularity relies on two R programming conventions:

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

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

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

To Work with `zelig()`

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

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

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

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

If you have a pre-existing model, writing a `zelig2contrib()` function is quite easy. Let's say that your model is `contrib()`, and takes the following arguments: `formula`, `data`, `weights`, and `start`. The `zelig()` function, in contrast, only takes the `formula`, `data`, `model`, and `by` arguments. You may use the `...` to pass additional arguments from `zelig()` to `zelig2contrib()`, and `<- NULL` to omit the elements you do not need. Continuing the Normal regression example from Section 8.1.2, let `formula`, `model`, and `data` be the inputs to `zelig()`, `M` is the number of subsets, and `...` are the additional arguments not defined in the `zelig()` call, but passed to `normal.regression()`.

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

The bracketed numbers above correspond to the comments below:

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

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

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

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

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

To Work with `setx()`

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

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

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

Compatibility with `sim()`

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

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

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

The output from `param()` should be either

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

There are two ways to simulate parameters:

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

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

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

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

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

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

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

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

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

   Continuing the Normal example,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.2 Getting Ready for the GUI

Zelig can work with a variety of graphical user interfaces (GUIs). GUIs work by knowing *a priori* what a particular model accepts, and presenting only those options to the user in some sort of graphical interface. Thus, in order for your model to work with a GUI, you must describe your model in terms that the GUI can understand. For models written using the guidelines in Chapter 8, your model will be compatible with (at least) the Virtual Data Center GUI. For pre-existing models, you will need to create a `describe.*()` function for your model following the examples in Section 13.1.

9.3 Formatting Reference Manual Pages

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

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

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

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

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

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

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

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

Help File Template

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

\subsubsection{Syntax}

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

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

\subsubsection{Quantities of Interest}
\begin{itemize}
\item The expected value of your distribution, including the formula
  for the expected value as a function of the systemic component and
```

```
  ancillary paramters.
\item The predicted value drawn from the distribution defined by the
       corresponding expected value.
\item The first difference in expected values, given when x1 is specified.
\item Other quantities of interest.
\end{itemize}

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

\subsubsection{Further Information}

\subsubsection{Contributors}
```

Part III

Reference Manual

Chapter 10

Main Commands

Help for each command in Zelig and R is available through `help.zelig()`. For example, typing `help.zelig(setx)` will launch a web browser with the appropriate reference manual page for the `setx()` command. (Occasionally, you may need to use, for example, `help(print)` rather than `help.zelig(print)`, to access the R help page instead of the default Zelig help page.)

10.1 `zelig`: Estimating a Statistical Model

Description

The `zelig()` command estimates a variety of statistical models. Use `zelig()` output with `setx()` and `sim()` to compute quantities of interest, such as predicted probabilities, expected values, and first differences, along with the associated measures of uncertainty (standard errors and confidence intervals).

Syntax

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

Arguments

- `formula`: a symbolic representation of the model to be estimated, in 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. (You may include more than two explanatory variables, of course.) The `+` symbol means "inclusion" not "addition." You may also include interaction terms and main effects in the form `x1*x2` without computing them in prior steps; `I(x1*x2)` to include only the interaction term and exclude the main effects; and quadratic terms in the form `I(x1^2)`.
- `model`: the name of a statistical model, enclosed in `""`. Currently, the following models are supported:

- `"blogit"`: Bivariate logistic regression. (see Section 12.1)
- `"bprobit"`: Bivariate probit regression. (see Section 12.2)
- `"exp"`: Exponential duration regression. (see Section 12.6)
- `"factor.bayes"`: Bayesian factor analysis. (see Section 12.7)
- `"factor.mix"`: Mixed data Bayesian factor analysis. (see Section 12.8)
- `"factor.ord"`: Bayesian factor analysis for ordinal variables. (see Section 12.9)
- `"ei.dynamic"`: Quinn's dyanamic ecological inference model for 2×2 tables. (see Section 12.3)
- `"ei.hier"`: Bayesian hierarchical ecological inference model for a cross-section of 2×2 tables. (see Section 12.4)
- `"gamma"`: Gamma regression. (see Section 12.10)
- `"irt1d"`: Bayesian unidimensional item response theory model. (see Section 12.11)
- `"irtkd"`: Bayesian k-dimensional item response theory model. (see Section 12.12)
- `"logit"`: Logistic regression. (see Section 12.13)
- `"logit.bayes"`: Bayesian logistic regression. (see Section 12.14)
- `"lognorm"`: Log-normal duration regression. (see Section 12.15)
- `"ls"`: Linear least squares regression. (see Section 12.16)
- `"mlogit"`: Multinomial logistic regression. (see Section 12.17)
- `"mlogit.bayes"`: Bayesian multinomial logistic regression. (see Section 12.18)
- `"negbin"`: Negative binomial event count regression. (see Section 12.19)
- `"normal"`: Normal linear regression. (see Section 12.20)
- `"normal.bayes"`: Bayesian Normal regression (see Section 12.21)
- `"ologit"`: Ordinal logistic regression. (see Section 12.22)
- `"oprobit"`: Ordinal probit regression. (see Section 12.23)
- `"oprobit.bayes"`: Bayesian ordinal probit regression. (see Section 12.24)
- `"poisson"`: Poisson event count regression. (see Section 12.25)
- `"poisson.bayes"`: Bayesian Poisson regression. (see Section 12.26)
- `"probit"`: Probit regression. (see Section 12.27)
- `"probit.bayes"`: Bayesian Probit regression. (see Section 12.28)
- `"relogit"`: Rare events logistic regression. (see Section 12.29)
- `"tobit"`: Tobit regression. (see Section 12.30)
- `"tobit.bayes"`: Bayesian tobit regression. (see Section 12.31)
- `"weibull"`: Weibull regression. (see Section 12.32)

- `data`: the name of a data frame containing the variables referenced in the formula, or a list of multiply imputed data frames each having the same variable names and number of rows (created by `mi()`).
- `by`: A factor variable contained in `data`. Zelig will subset the data frame based on the levels in the `by` variable, and estimate a model for each subset. This a particularly powerful option which will allow you to save a considerable amount of effort. For example, to run the same model on all fifty states, you could type:

  ```
  > z.out <- zelig(y ~ x1 + x2, data = mydata, model = "ls", by = "state")
  ```

  You may also use `by` to run models by MatchIt subclass.
- `save.data`: the logical variable indicating whether the original data frame should be saved within the output of `zelig()`. If `FALSE` (default), only the name of the data frame will be stored.
- `...`: additional arguments passed to `zelig()`, depending on the model to be estimated.

Output Values

Depending on the class of model selected, `zelig()` will return an object with elements including `coefficients`, `residuals`, and `formula` which may be summarized using `summary(z.out)` or individually extracted using, for example, `z.out$coefficients`. See the specific models listed above for additional output values, or simply type `names(z.out)`.

Examples

```
> z.out <- zelig(y ~ x1 + x2, model = "logit", data = mydata)
> z.out <- zelig(log(y) ~ x1 + x2, model = "ls", data = mydata)
> z.out <- zelig(y ~ x1 + I(x2^2), model = "ologit", data = mydata)
> z.out <- zelig(y ~ x1 * x2, model = "probit", data = mydata)
> z.out <- zelig(y ~ x1 * x2, model = "probit",
                 data = mi(mydata1, mydata2, mydata3, mydata4, mydata5))
```

See Also

- Section 4.1 for an overview of the Zelig simulation procedure.
- Section 4.2 for an overview of supported models.
- Section 3.1.2 for an overview of R syntax.
- Section 2 for how to create factor variables.

Contributors

Kosuke Imai, Gary King, and Olivia Lau created the `zelig()` framework for handling a selection of the statistical models contained in the base, survreg, VGAM, and nnet packages for R. For additional details, please refer to the appropriate model reference page.

10.2 setx: Setting Explanatory Variable Values

Description

The `setx()` command uses the variables identified in the `formula` generated by `zelig()` and sets the values of the explanatory variables to the selected values. Use `setx()` after `zelig()` and before `sim()` to simulate quantities of interest.

Syntax

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

Arguments

- `z.out`: the saved output from `zelig()`.
- `fn`: a list of three functions to apply to three types of variables
  - `numeric` variables are set to their mean by default, but you may select any other mathematical function.
  - `ordered` factors are set to their median by default, and most mathematical operations will work on them. (If you select `ordered = mean`, however, `setx()` will default to median with a warning.)
  - `other` variables may consist of unordered factors, character strings, or logical variables. The `other` variables may only be set to their mode. (If you wish to set one of the other variables to a specific value, you may do so using `...` or by making them numeric and using `fn`.)

  In the special case `fn = NULL`, `setx()` will return all of the observations without applying any function to the data.
- `data`: a new data frame used to set the values of explanatory variables. If `data = NULL` (the default), the data frame called in `zelig()` is used.
- `cond`: a logical value indicating whether the usual unconditional prediction is used (the default) or whether the optional conditional prediction should be performed (choose `cond = TRUE`). If you choose `cond = TRUE`, `setx()` will coerce `fn = NULL` and ignore the arguments in `...`.
- `...`: user-defined values of specific variables that overwrite the default values set by the function. For example, adding `var1 = mean(data$var1)` sets the value of `var1` to the mean of `data$var1`; `x1 = 12` explicitly sets the value of `x1` to 12. In addition, you may specify one explanatory variable as a range of values, creating one observation for every unique value in the range of values. The other variables are set to the values given by `fn`.

Output Values

- For unconditional prediction, `x.out` is a model matrix based on the specified values for the explanatory variables. For multiple analyses (i.e., when choosing the `by` option in `zelig()`, `setx()` returns the selected values calculated over the entire data frame. If you wish to calculate values over just one subset of the data frame, the 5th subset for example, you may use:

  ```
  > x.out <- setx(z.out[[5]])
  ```

- For conditional prediction, `x.out` includes the model matrix and the dependent variable(s). For multiple analyses (when choosing the `by` option in `zelig()`), `setx()` returns the observed explanatory variables in each subset.

Example: Unconditional Prediction

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

Using all observations:

```
> x.out <- setx(z.out, fn = NULL)
```

Defining a user defined function for `fn`:

```
> quants <- function(x) {
+  quantile(x, 0.25)
+ }
> x.out <- setx(z.out, fn = list(numeric = quants))
```

Example: Conditional Prediction With MatchIt Data

Use `demo(match)` to view this example.

```
> data(lalonde)
> match.out <- matchit(treat ~ age + educ + black + hispan + married +
                       nodegree + re74 + re75, data = lalonde)
> z.out <- zelig(re78 ~ pscore, data = match.data(match, "control"),
                 model = "ls")
> x.out <- setx(z.out, data = match.data(match, "treat"), fn = NULL,
                cond = TRUE)
> s.out <- sim(z.out, x = x.out)
```

Example: Conditional Prediction With Multiple Analyses

Use `demo(conditional)` to view this example:

```
> data(turnout)
> z.out <- zelig(vote ~ age + educate + income, by = "race",
                 data = turnout, model = "probit")

# Calculate the ATE (average treatment effect) for race == "white":
> x.white <- setx(z.out$others, data = turnout[turnout$race == "white",],
                  fn = NULL, cond = TRUE)
> s.others <- sim(z.out$others, x = x.white)

# Calculate the ATE for race == "others":
> x.others <- setx(z.out$white, data = turnout[turnout$race == "others",],
                  fn = NULL, cond = TRUE)
> s.others <- sim(z.out$white, x = x.others)
```

See Also

- Section 4.1 for an overview of the Zelig simulation procedure.
- Section 4.2 for an overview of supported models.
- Section 3.1.2 for an overview of R syntax.

Contributors

Kosuke Imai, Gary King, and Olivia Lau ported `setx()` to R, following the logic of Clarify (for Stata). See King, Tomz, and Wittenberg, 2000.

Sample data are a selection of 2,000 observations from

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

10.3 `sim`: Simulating Quantities of Interest

Description

Simulate quantities of interest from the estimated model output from `zelig()` given specified values of explanatory variables established in `setx()`. For classical *maximum likelihood* models, `sim()` uses asymptotic normal approximation to the log-likelihood. 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. Alternatively, you may generate quantities of interest using bootstrapped parameters.

Syntax

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

Arguments

- `z.out`: the output object from `zelig()`.
- `x`: values of explanatory variables used for simulation, generated by `setx()`.
- `x1`: optional values of explanatory variables (generated by a second call of `setx()`), used to simulate first differences and risk ratios. (Not available for conditional prediction.)
- `num`: the number of simulations, i.e., posterior draws. If the `num` argument is omitted, `sim()` draws 1,000 simulations if `bootstrap = FALSE` (the default), or 100 simulations if `bootstrap = TRUE`. You may increase this value to improve accuracy. (Not available for conditional prediction.)
- `bootstrap:` a logical value indicating if parameters should be generated by re-fitting the model for bootstrapped data, rather than from the likelihood or posterior. (Not available for conditional prediction.)
- `bootfn:` a function which governs how the data is sampled, re-fits the model, and returns the bootstrapped model parameters. If `bootstrap = TRUE` and `bootfn = NULL`, `sim()` will sample observations from the original data (with replacement) until it creates a sampled dataset with the same number of observations as the original data. Alternative bootstrap methods (for which users have to specify their own `bootfn`) include sampling the residuals rather than the observations, weighted sampling, and parametric bootstrapping. (Not available for conditional prediction.)
- `...`: additional optional arguments passed to `boot()`.

Output Values

The output stored in `s.out` varies by model. Use the `names()` command to view the output stored in `s.out`. Common elements include:

- `x`: the `setx()` values for the explanatory variables, used to calculate the quantities of interest (expected values, predicted values, etc.).
- `x1`: the optional `setx()` object used to simulate first differences, and other model-specific quantities of interest, such as risk-ratios.
- `call`: the options selected for `sim()`, used to replicate quantities of interest.
- `zelig.call`: the original command and options for `zelig()`, used to replicate analyses.
- `num`: the number of simulations requested.
- `par`: the parameters (coefficients, and additional model-specific parameters). You may wish to use the same set of simulated parameters to calculate quantities of interest rather than simulating another set.
- `qi$ev`: simulations of the expected values given the model and `x`.
- `qi$pr`: simulations of the predicted values given by the fitted values.
- `qi$fd`: simulations of the first differences (or risk difference for binary models) for the given `x` and `x1`. The difference is calculated by subtracting the expected values given `x` from the expected values given `x1`. (If do not specify `x1`, you will not get first differences or risk ratios.)
- `qi$rr`: simulations of the risk ratios for binary and multinomial models. See specific models for details.
- `qi$ate.ev`: simulations of the average expected treatment effect for the treatment group, using conditional prediction. Let t_i be a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Then the average expected treatment effect for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \{Y_i(t_i = 1) - E[Y_i(t_i = 0)]\},$$

  where $Y_i(t_i = 1)$ is the value of the dependent variable for observation i in the treatment group. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- `qi$ate.pr`: simulations of the average predicted treatment effect for the treatment group, using conditional prediction. Let t_i be a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Then the average predicted treatment effect for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \{Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)}\},$$

  where $Y_i(t_i = 1)$ is the value of the dependent variable for observation i in the treatment group. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

In the case of censored Y in the exponential, Weibull, and lognormal models, `sim()` first imputes the uncensored values for Y before calculating the ATE.

You may use the `$` operator to extract any of the above from `s.out`. For example, `s.out$qi$ev` extracts the simulated expected values.

See Also

- Section 4.1 for an overview of the Zelig simulation procedure.
- Section 4.2 for an overview of supported models.
- Section 3.1.2 for an overview of R syntax.

Contributors

Kosuke Imai, Gary King, and Olivia Lau generalized `sim()` from a similar procedure used in Clarify (for Stata, see King, Tomz, and Wittenberg, 2000) and added procedures to bootstrap quantities of interest, and perform conditional prediction.

10.4 summary: Summarizing Zelig Output

Description

Summarize R objects using using the generic function `summary()`. R automatically recognizes the type of object (lists, data frames, variables, etc.) and selects the appropriate `summary()` function.

Syntax

```
> summary(z.out, subset = NULL, ...)
> summary(s.out, subset = NULL, CI = 95,
          stats = c("mean", "sd", "min", "max"), ...)
```

To set the desired number of digits in the summary output, use `options(digits = x)`, where `x` is the desired number of digits, prior to using the `sumamary()` command.

Arguments for multiply-imputed `zelig()` output

For `zelig()` output created with m multiply-imputed data sets, the `z.out` object contains one set of output for each imputed data set. Options for multiply-imputed `zelig()` output include:

- `subset`: specifies which of the m sets of output to summarize. Possible arguments include:
  - `NULL`: (default) which produces one summary by averaging the coefficients and standard errors from all the imputed data sets using the Rubin rules (King, Honaker, Joseph, Scheve (2000), p. 53).
  - a numeric vector: consisting of any set of numbers in $[1, m]$. For example, you may use `summary(z.out, subset = 2:4)` to view the output from the second, third, and fourth imputed data sets only.
- ...: additional options passed to `print()`.

Arguments for subsetted `zelig()` output

If you selected the `by` option, `zelig()` creates an `z.out` object with one set of regression output for each of the unique values in the `by` variable. Options for subsetted analyses are:

- `subset`: specifies which regression output to summarize. You cannot summarize multiple analyses generated using the `"by"` option in one summary, as each of the levels in the `by` variable may have a different number of associated observations. (A weighted summary would also be inappropriate, as the weighted coefficients would be identical to the coefficients generated without subsetting.)

  - numeric vector: specifying which sets of regression output to summarize. By default, `summary()` for subsetted output sequentially summarizes the regression output for each unique value in the `by` variable. You may choose to summarize just the fifteenth unique value by typing: `summary(z.out, subset = 15)`.

- `...`: Additional arguments passed to `print()`.

Arguments for `sim()` output

Summaries of `sim()` output includes these additional options:

- `subset`: Only valid for `sim()` output created using more than one observation in `x`. You may select to summarize the quantities of interest created by each observation in `x` using one of the following options:
  - `NULL`: (default) produces one summary per quantity of interest by combining the simulations from each observation into a single set and then summarizing.
  - `all`: summarizes the quantity of interest produced by each observation separately.
  - a numeric vector: specifying the observations to summarize
- `CI`: A value between 0 and 100, for the percentage confidence interval. The default value is 95, for a 95 percent confidence interval.
- `stats`: A vector specifying the statistics to calculate for each quantity of interest. The default values are `c("mean", "sd", "min", "max")`.

Using `summary()` on `sim()` output from multiple analyses will automatically weight the quantities of interest according to the proportion of observations in each strata.

Output Values

- For `zelig()` output objects, `summary()` returns a formatted summary of the regression output, including the usual table of coefficients, standard errors, and t-values. Additional output depends on the statistical model, the names of which may be viewed using the `names(summary(z.out))` command.
- For `sim()` output objects, `summary()` returns summary statistics for each quantity of interest, including a default 95% confidence interval. Use `names(summary(s.out))` to see the available output. Note that `qi.stats` contains a useful matrix of summary statistics for each of the quantities of interest. For example, `summary(s.out)$qi.stats$ev`, extracts the summary matrix of expected values.

See Also

Advanced users may wish to refer to `help.zelig("print")`, as well as `help(summary)`, `help(summary.lm)`, `help(summary.glm)`, and `help(anova)` in the base package.

Contributors

Kosuke Imai, Gary King, and Olivia Lau added summary methods for certain `zelig()` models, and for `sim()` output.

10.5 plot: Graphing Quantities of Interest

Description

The `plot()` command generates default plots for `sim()` objects with one-observation values in `x` and `x1`. Additional default plotting methods for other data structures are described in `help(plot)`.

Syntax

```
> plot(s.out, alt.col = "red", user.par = FALSE, ...)
```

Arguments

- `s.out`: stored output from `sim()`. If the `x` or `x1` values stored in the object contain more than one observation, `plot.zelig()` will return an error. For linear or generalized linear models with more than one observation in `x` and optionally `x1`, you may use `plot.ci()` (see Section 11.3).
- `alt.col`: alternative color used for contrast in the plots. The primary color is black. Type `colors()` to see a list of available alternative colors.
- `user.par`: a logical value indicating whether to use the default Zelig plotting parameters (`user.par = FALSE`) or user-defined parameters (`user.par = TRUE`), set using the `par()` function prior to plotting.
- `...`: Additional parameters passed to `plot()`. Because `plot.zelig()` primarily produces diagnostic plots, many of these parameters are hard-coded for convenience and presentation. If you wish to produce plots using different parameters, you may follow the directions in Section 5.

Output Values

Depending on the class of model selected, `plot.zelig()` will return an on-screen window with graphs of the various quantities of interest. You may save these plots using the commands described in Section 5.3.

Examples

```
> z.out <- zelig(y ~ x1 + x2, model = "logit", data = mydata)
> x.out <- setx(z.out, x1 = 10)
> x.alt <- setx(z.out, x1 = 0)
> s.out <- sim(z.out, x = x.out, x1 = x.alt)
> plot(s.out)
```

See Also

- The `help(plot)` and `help(lines)` reference pages for optional arguments passed to `...`.
- The `help(par)` reference page for user-specified plotting parameters.
- Section 5 for an overview of plotting procedures

Contributors

Kosuke Imai, Gary King, and Olivia Lau created the plots generated by `plot.zelig()`. These plots are individually available in the base, MASS, and vcd libraries.

10.6 `print`: Printing Quantities of Interest

Description

The `print` command formats summary output and allows users to specify the number of decimal places as an optional argument.

Syntax

```
> print(x, digits = 3, print.x = FALSE)
```

Arguments

- `x`: the object to be printed may be `z.out` output from `zelig()`, `x.out` output from `setx()`, `s.out` output from `sim()`, or other R data structures.
- `digits`: the minimum number of significant digits to return for all elements of $x < 0$. By default, `print()` avoids scientific notation, but setting the number of digits to 1 will frequently force output in scientific notation. The number of `digits` is not the number of significant digits for all output values, but the minimum number of significant digits for the smallest value in `x` between -1 and 1; this governs the number of significant digits in the rest of the values with decimal output.
- `print.x`: a logical value for `sim()` output, which specifies whether to print a summary (`print.x = FALSE`, the default) of the `x` and `x1` *inputs* to `sim()`, or the complete set of inputs (optionally, `print.x = TRUE`).

Examples

```
> print(summary(z.out), digits = 2)
> print(summary(s.out), digits = 3, print.x = TRUE)
```

See Also

Advanced users may wish to refer to `help(print)`.

Contributors

Kosuke Imai, Gary King, and Olivia Lau added print methods for `sim()` output, and `summary()` output for Zelig objects.

10.7 `repl`: Replicating Analyses

Description

The `repl()` command included with Zelig takes `zelig()` or `sim()` output objects and replicates (literally, re-runs) the entire analysis. The results should be an output object identical to the original input object in the case of `zelig()` output. In the case of `sim()` output, the replicated analyses may differ slightly due to stochastic randomness in the simulation procedure.

Syntax

```
> repl(z.out, data = NULL)
> repl(s.out, data = NULL, prev = NULL, x = NULL, x1 = NULL,
       bootfn = NULL)
```

Arguments

- `z.out`: Stored output from `zelig()`.
- `s.out`: Stored output from `sim()`
- `data`: You may manually input the data frame name rather than allowing replicate to draw the data frame name from the object to be replicated.
- `prev`: When replicating `sim()` output, you may optionally use the previously simulated parameters to calculate the quantities of interest rather than simulating a new set of parameters. For all models, this should produce identical quantities of interest. If the parameters were bootstrapped in the original analysis, this will save a considerable amount of time.
- `x`: When replicating `sim()` output, you may optionally use an alternative `setx()` value for the `x` input.
- `x1`: When replicating `sim()` output, you may optionally use an alternative `setx()` object for the `x1` input to replicating the `sim()` object.
- `bootfn`: When replicating `sim()` output with bootstrapped parameters, you should manually specify the `bootfn` if a non-default option was used.

Output Values

For `zelig()` output, `repl()` will create output that is in every way identical to the original input. You may check to see whether they are identical by using the `identical()` command.

For `sim()` output, `repl()` output will be will be identical to the original object if you choose not to simulate new parameters, and instead choose to calculate quantities of interest

using the previously simulated parameters (using the `prev` option. If you choose to simulate new parameters, the summary statistics for each quantity of interest should be identical, up to a random approximation error. As the number of simulations increases, this error decreases.

Examples

```
> data(turnout)
> z.out <- zelig(vote ~ race + educate, model = "logit", data = turnout[1:1000,])
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
> z.rep <- repl(z.out)
> identical(z.out$coef, z.rep$coef)
> z.alt <- repl(z.out, data = turnout[1001:2000,])
> s.rep <- repl(s.out, prev = s.out$par)
> identical(s.out$ev, s.rep$ev)
```

See Also

- Section 3.2.2 for loading data from disk.

Contributors

Kosuke Imai, Gary King, and Olivia Lau.

Chapter 11

Supplementary Commands

11.1 `matchit`: Create matched data

Description

MatchIt implements the suggestions of Ho et al. (2005) for improving parametric statistical models by preprocessing data with semi-parametric matching methods. It uses a sophisticated array of matching methods to select well-matched treated and control units from the original data set, thus reducing the dependence of causal inferences on functional form and other parametric assumptions. After pre-processing, MatchIt output can be used just like any other dataset in Zelig to estimate causal effects. In this way, MatchIt improves rather than replaces existing parametric models, reducing sensitivity to modeling assumptions. The matching methods available in MatchIt include exact matching on all covariates, nearest neighbor matching, subclassification, optimal matching, genetic matching, and full matching. An outline of all options are provided below; see the full documentation (available at `http://gking.harvard.edu/matchit/`) for more details.

Syntax

```
> m.out <- matchit(formula, data, method = "nearest", verbose = FALSE, ...)
```

Arguments

Arguments for All Matching Methods

- `formula`: formula used to calculate the distance measure for matching. It takes the usual syntax of R formulas, `treat ~ x1 + x2`, where `treat` is a binary treatment indicator, and `x1` and `x2` are the pre-treatment covariates. Both the treatment indicator and pre-treatment covariates must be contained in the same data frame, which is specified as `data` (see below). All of the usual R syntax for formulas work here. For example, `x1:x2` represents the first order interaction term between `x1` and `x2`, and `I(x1 ^ 2)` represents the square term of `x1`. See `help(formula)` for details.

- `data`: the data frame containing the variables called in `formula`.
- `method`: the matching method (default = `"nearest"`, nearest neighbor matching). Currently, `"exact"` (exact matching), `"full"` (full matching), `"nearest"` (nearest neighbor matching), `"optimal"` (optimal matching), `"subclass"` (subclassification), and `"genetic"` (genetic matching) are available. Note that within each of these matching methods, MatchIt offers a variety of options. See below for more details.
- `verbose`: a logical value indicating whether to print the status of the matching algorithm (default = `FALSE`).

Additional Arguments for Specification of Distance Measures The following arguments specify distance measures that are used for matching methods. These arguments apply to all matching methods *except exact matching*.

- `distance`: the method used to estimate the distance measure (default = `"logit"`, logistic regression) or a numerical vector of user's own distance measure. Before using any of these techniques, it is best to understand the theoretical groundings of these techniques and to evaluate the results. Most of these methods (such as logistic or probit regression) estimate the propensity score, defined as the probability of receiving treatment, conditional on the covariates. Available methods include:
  - `"mahalanobis"`: the Mahalanobis distance measure.
  - binomial generalized linear models with one of the following link functions:
    - `"logit"`: logistic link
    - `"linear.logit"`: logistic link with linear propensity score)[1]
    - `"probit"`: probit link
    - `"linear.probit"`: probit link with linear propensity score
    - `"cloglog"`: complementary log-log link
    - `"linear.cloglog"`: complementary log-log link with linear propensity score
    - `"log"`: log link
    - `"linear.log"`: log link with linear propensity score
    - `"cauchit"` Cauchy CDF link
    - `"linear.cauchit"` Cauchy CDF link with linear propensity score
  - Choose one of the following generalized additive models (see `help(gam)` for more options).
    - `"GAMlogit"`: logistic link
    - `"GAMlinear.logit"`: logistic link with linear propensity score
    - `"GAMprobit"`: probit link

[1]The linear propensity scores are obtained by transforming back onto a linear scale.

    - "GAMlinear.probit": probit link with linear propensity score
    - "GAMcloglog": complementary log-log link
    - "GAMlinear.cloglog": complementary log-log link with linear propensity score
    - "GAMlog": log link
    - "GAMlinear.log": log link with linear propensity score,
    - "GAMcauchit": Cauchy CDF link
    - "GAMlinear.cauchit": Cauchy CDF link with linear propensity score
  - "nnet": neural network model. See help(nnet) for more options.
  - "rpart": classification trees. See help(rpart) for more options.

- distance.options: optional arguments for estimating the distance measure. The input to this argument should be a list. For example, if the distance measure is estimated with a logistic regression, users can increase the maximum IWLS iterations by distance.options = list(maxit = 5000). Find additional options for general linear models using help(glm) or help(family), for general additive models using help(gam), for neutral network models help(nnet), and for classification trees help(rpart).

- discard: specifies whether to discard units that fall outside some measure of support of the distance measure (default = "none", discard no units). Discarding units may change the quantity of interest being estimated. Enter a logical vector indicating which unit should be discarded or choose from the following options:

  - "none": no units will be discarded before matching. Use this option when the units to be matched are substantially similar, such as in the case of matching treatment and control units from a field experiment that was close to (but not fully) randomized (e.g., Imai 2005), when caliper matching will restrict the donor pool, or when you do not wish to change the quantity of interest and the parametric methods to be used post-matching can be trusted to extrapolate.
  - "hull.both": all units that are not within the convex hull will be discarded. We recommend that this option be used with observational data sets.
  - "both": all units (treated and control) that are outside the support of the distance measure will be discarded.
  - "hull.control": only control units that are not within the convex hull of the treated units will be discarded.
  - "control": only control units outside the support of the distance measure of the treated units will be discarded. Use this option when the average treatment effect on the treated is of most interest and when you are unwilling to discard non-overlapping treatment units (which would change the quantity of interest).

  - `"hull.treat"`: only treated units that are not within the convex hull of the control units will be discarded.
  - `"treat"`: only treated units outside the support of the distance measure of the control units will be discarded. Use this option when the average treatment effect on the control units is of most interest and when unwilling to discard control units.

- `reestimate`: If `FALSE` (default), the model for the distance measure will not be re-estimated after units are discarded. The input must be a logical value. Re-estimation may be desirable for efficiency reasons, especially if many units were discarded and so the post-discard samples are quite different from the original samples.

Additional Arguments for Subclassification

- `sub.by`: criteria for subclassification. Choose from: `"treat"` (default), the number of treatment units; `"control"`, the number of control units; or `"all"`, the total number of units.

- `subclass`: either a scalar specifying the number of subclasses, or a vector of probabilities bounded between 0 and 1, which create quantiles of the distance measure using the units in the group specified by `sub.by` (default = `subclass = 6`).

Additional Arguments for Nearest Neighbor Matching

- `m.order`: the order in which to match treatment units with control units.
  - `"largest"` (default): matches from the largest value of the distance measure to the smallest.
  - `"smallest"`: matches from the smallest value of the distance measure to the largest.
  - `"random"`: matches in random order.

- `replace`: logical value indicating whether each control unit can be matched to more than one treated unit (default = `replace = FALSE`, each control unit is used at most once – i.e., sampling without replacement). For matching with replacement, use `replace = TRUE`.

- `ratio`: the number of control units to match to each treated unit (default = `1`). If matching is done without replacement and there are fewer control units than `ratio` times the number of eligible treated units (i.e., there are not enough control units for the specified method), then the higher ratios will have `NA` in place of the matching unit number in `match.matrix`.

- `exact`: variables on which to perform exact matching within the nearest neighbor matching (default = `NULL`, no exact matching). If `exact` is specified, only matches that exactly match on the covariates in `exact` will be allowed. Within the matches that match on the variables in `exact`, the match with the closest distance measure will be chosen. `exact` should be entered as a vector of variable names (e.g., `exact = c("X1", "X2")`).
- `caliper`: the number of standard deviations of the distance measure within which to draw control units (default = `0`, no caliper matching). If a caliper is specified, a control unit within the caliper for a treated unit is randomly selected as the match for that treated unit. If `caliper != 0`, there are two additional options:
  - `calclosest`: whether to take the nearest available match if no matches are available within the `caliper` (default = `FALSE`).
  - `mahvars`: variables on which to perform Mahalanobis-metric matching within each caliper (default = `NULL`). Variables should be entered as a vector of variable names (e.g., `mahvars = c("X1", "X2")`). If `mahvars` is specified without `caliper`, the caliper is set to 0.25.
- `subclass` and `sub.by`: See the options for subclassification for more details on these options. If a `subclass` is specified within `method = "nearest"`, the matched units will be placed into subclasses after the nearest neighbor matching is completed.

Additional Arguments for Optimal Matching

- `ratio`: the number of control units to be matched to each treatment unit (default = `1`).
- `...`: additional inputs that can be passed to the `fullmatch()` function in the `optmatch` package. See `help(fullmatch)` or http://www.stat.lsa.umich.edu/~bbh/optmatch.html for details.

Additional Arguments for Full Matching

- `...`: additional inputs that can be passed to the `fullmatch()` function in the `optmatch` package. See `help(fullmatch)` or http://www.stat.lsa.umich.edu/~bbh/optmatch.html for details.

Additional Arguments for Genetic Matching The available options are listed below.

- `ratio`: the number of control units to be matched to each treatment unit (default = `1`).
- `...`: additional minor inputs that can be passed to the `GenMatch()` function in the `Matching` package. See `help(GenMatch)` or http://sekhon.polisci.berkeley.edu/library/Matching/htm for details.

Output Values

Regardless of the type of matching performed, the `matchit` output object contains the following elements:[2]

- `call`: the original `matchit()` call.
- `formula`: the formula used to specify the model for estimating the distance measure.
- `model`: the output of the model used to estimate the distance measure. `summary(m.out$model)` will give the summary of the model where `m.out` is the output object from `matchit()`.
- `match.matrix`: an $n_1 \times$ `ratio` matrix where:
  - the row names represent the names of the treatment units (which match the row names of the data frame specified in `data`).
  - each column stores the name(s) of the control unit(s) matched to the treatment unit of that row. For example, when the `ratio` input for nearest neighbor or optimal matching is specified as 3, the three columns of `match.matrix` represent the three control units matched to one treatment unit).
  - `NA` indicates that the treatment unit was not matched.
- `discarded`: a vector of length n that displays whether the units were ineligible for matching due to common support restrictions. It equals `TRUE` if unit i was discarded, and it is set to `FALSE` otherwise.
- `distance`: a vector of length n with the estimated distance measure for each unit.
- `weights`: a vector of length n with the weights assigned to each unit in the matching process. Unmatched units have weights equal to 0. Matched treated units have weight 1. Each matched control unit has weight proportional to the number of treatment units to which it was matched, and the sum of the control weights is equal to the number of uniquely matched control units.
- `subclass`: the subclass index in an ordinal scale from 1 to the total number of subclasses as specified in `subclass` (or the total number of subclasses from full or exact matching). Unmatched units have `NA`.
- `q.cut`: the subclass cut-points that classify the distance measure.
- `treat`: the treatment indicator from `data` (the left-hand side of `formula`).
- `X`: the covariates used for estimating the distance measure (the right-hand side of `formula`). When applicable, `X` is augmented by covariates contained in `mahvars` and `exact`.

[2]When inapplicable or unnecessary, these elements may equal `NULL`. For example, when exact matching, `match.matrix = NULL`.

Contributors

If you use MatchIt, please cite

> — (2006), "Matching as Nonparametric Preprocessing for Parametric Causal Inference," Http://gking.harvard.edu/files/abs/matchp-abs.shtml and Ho, D., Imai, K., King, G., and Stuart, E. (2005), "Matching as Nonparametric Preprocessing for Parametric Causal Inference," Http://gking.harvard.edu/matchit/

The `convex.hull` discard option is implemented via the `WhatIf` package. If you use this option, please cite

> and either — (2006), "The Dangers of Extreme Counterfactuals," *Political Analysis*, 14, 131–159, http://gking.harvard.edu/files/abs/counterft-abs.shtml or

Generalized linear distance measures are implemented via the `stats` package. If you use this distance measure, please cite

> Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed

Generalized additive distance measures are implemented via the `mcgv` package. If you use this distance measure, please cite

> Hastie, T. J. and Tibshirani, R. (1990), *Generalized Additive Models*, London: Chapman Hall

The neural network distance measure is implemented via the `nnet` package. If you use this distance measure, please cite

> Ripley, B. (1996), *Pattern Recognition and Neural Networks*, Cambridge Univeristy Press

The classification trees distance measure is implemented via the `rpart` package. If you use this distance measure, please cite

> Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984), *Classification and Regression Trees*, New York, New York: Chapman & Hall

Full and optimal matching are implemented via the `optmatch` package by Ben Hansen. If you use either of these methods, please cite

> Hansen, B. B. (2004), "Full Matching in an Observational Study of Coaching for the SAT," *Journal of the American Statistical Association*, 99, 609–618

Genetic matching is implemented via the `Matching` package by Jasjeet Sekhon. If you use this method, please cite

> Diamond, A. and Sekhon, J. (2005), "Genetic Matching for Estimating Causal Effects: A New Method of Achieving Balance in Observational Studies," http://jsekhon.fas.harvard.edu/

11.2 `mi`: Create a list of multiply imputed data frames

Description

Use `mi()` to create a list of multiply imputed data frames for use with `zelig()`.

Syntax

```
mi(data1, data2, ...)
```

Arguments

- `data1`: the first multiply imputed data frame.
- `data2`: the second multiply imputed data frame.
- `...`: additional multiply imputed data frames.

Output Values

A list of data frames with class `"mi"`

See Also

- See Section 4.1.2 to use `mi()` with `zelig()`.

Contributors

Kosuke Imai, Gary King, Olivia Lau, and Ferdinand Alimadhi.

11.3 plot.ci: Plotting Vertical confidence Intervals

Description

The plot.ci() command generates vertical confidence intervals for linear or generalized linear univariate response models.

Syntax

```
> plot.ci(object, CI = 95, qi = "ev", col = c("red", "blue"), ...)
```

Arguments

- object: stored output from sim(). The x and optional x1 values used to generate the sim() output object must have more than one observation.
- CI: the selected confidence interval. Defaults to 95 percent.
- qi: the selected quantity of interest. Defaults to expected values.
- col: a vector of at most two colors for plotting the expected value given by x and the alternative set of expected values given by x1 in sim(). If the quantity of interest selected is not the expected value, or x1 = NULL, only the first color will be used.
- ...: Additional parameters, such as xlab, ylab, and main, passed to plot().

Output Values

For all univariate response models, plot.ci() returns vertical confidence intervals over a specified range of one explanatory variable. You may save this plot using the commands described in Section 5.3.

Example

This demo is available within R as demo(vertci). Load the sample data and estimate the model.

```
data(turnout)
z.out <- zelig(vote ~ race + educate + age + I(age^2) + income,
               model = "logit", data = turnout)
```

Establish a range for 'age', the key explanatory variable, and create two setx objects that hold the other explanatory variables to fixed, while allowing age to range from 18 to 95, inclusive.

```
age.range <- 18:95
x.low <- setx(z.out, educate = 12, age = age.range)
x.high <- setx(z.out, educate = 16, age = age.range)
```

Simulate quantities of interest.

```
s.out <- sim(z.out, x = x.low, x1 = x.high)
```

Plot these results using the `plot.ci()` function:

```
plot.ci(s.out, xlab = "Age in Years",
        ylab = "Predicted Probability of Voting",
        main = "Effect of Education and Age on Voting Behavior")
legend(45, 0.52, legend = c("College Education (16 years)",
       "High School Education (12 years)"), col = c("blue","red"),
       lty = c("solid"))
```

See Also

- The `help(plot)` and `help(lines)` reference pages.
- Section 5 for an overview of plotting procedures

Contributors

Kosuke Imai, Gary King, and Olivia Lau created the R procedure to generate vertical confidence intervals for simulated quantities of interest.

Sample data are a selection of 2,000 observations from

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

11.4 `rocplot`: Receiver Operator Characteristic Plots

Description

The `rocplot()` command generates a receiver operator characteristic plot to compare the in-sample (default) or out-of-sample fit for two logit or probit regressions.

Syntax

```
rocplot(y1, y2, fitted1, fitted2, cutoff = seq(from=0, to=1, length=100),
        lty1 = "solid", lty2 = "dashed", lwd1 = par("lwd"), lwd2 = par("lwd"),
        col1 = par("col"), col2 = par("col"), plot = TRUE, ...)
```

Arguments

- `y1`: Response variable for the first model.
- `y2`: Response variable for the second model.
- `fitted1`: Fitted values for the first model. These values may represent either the in-sample or out-of-sample fitted values.
- `fitted2`: Fitted values for the second model.
- `cutoff`: A vector of cut-off values between 0 and 1, at which to evaluate the proportion of 0s and 1s correctly predicted by the first and second model. By default, this is 100 increments between 0 and 1, inclusive.
- `lty1`, `lty2`: The line type for the first model (`lty1`) and the second model (`lty2`), defaulting to solid and dashed, respectively.
- `lwd1`, `lwd2`: The width of the line for the first model (`lwd1`) and the second model (`lwd2`), defaulting to 1 for both.
- `col1`, `col2`: The colors of the line for the first model (`col1`) and the second model (`col2`), defaulting to black for both.
- `plot`: By default, `plot = TRUE`, which generates a plot to the selected device. If `FALSE`, `rocplot()` returns a list of output (see below).
- `...`: Additional parameters passed to plot, including `xlab`, `ylab`, and `main`.

Output Values

If `plot = TRUE`, `rocplot()` generates an ROC plot for two logit or probit models. You may save this plot using the commands described in Section 5.3.

If `plot = FALSE`, `rocplot()` returns a list with the following elements:

- `roc1`: A matrix containing a vector of x-coordinates and y-coordinates corresponding to the number of ones and zeros correctly predicted for the first model.
- `roc2`: A matrix containing a vector of x-coordinates and y-coordinates corresponding to the number of ones and zeros correctly predicted for the second model.
- `area1`: The area under the first ROC curve, calculated using Reimann sums.
- `area2`: The area under the second ROC curve, calculated using Reimann sums.

Example

You may view this example using `demo(roc)`.

```
> data(turnout)
> z.out1 <- zelig(vote ~ race + educate + age, model = "logit",
                  data = turnout)
> z.out2 <- zelig(vote ~ race + educate, model = "logit",
                  data = turnout)
> rocplot(z.out1$y, z.out2$y, fitted(z.out1), fitted(z.out2))
```

See Also

- The `help(plot)` and `help(lines)` reference pages.
- Section 5 for an overview of plotting procedures.

Contributors

Kosuke Imai, Gary King, and Olivia Lau.

Sample data are a selection of 2,000 observations from

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

11.5 `ternaryplot`: Ternary Diagram for 3D Data

Description

Visualizes compositional, 3-dimensional data in an equilateral triangle. A point's coordinates are found by computing the gravity center of mass points using the data entries as weights. Thus, the coordinates of a point $P(a, b, c)$ for $a + b + c = 1$ are: $P(b + \frac{c}{2}, \frac{c\sqrt{3}}{2})$.

Syntax

```
ternaryplot(x, scale = 1, dimnames = NULL,
            dimnames.position = c("corner","edge","none"),
            dimnames.color = "black", id = NULL, id.color = "black",
            coordinates = FALSE, grid = TRUE, grid.color = "gray",
            labels = c("inside", "outside", "none"),
            labels.color = "darkgray", border = "black", bg = "white",
            pch = 19, cex = 1, prop.size = FALSE, col = "red",
            main = "ternary plot", ...)
```

Arguments

- `x`: a matrix with three columns.
- `scale`: row sums scale to be used.
- `dimnames`: dimension labels (defaults to the column names of `x`).
- `dimnames.position, dimnames.color`: position and color of dimension labels.
- `id`: optional labels to be plotted below the plot symbols. `coordinates` and `id` are mutual exclusive.
- `id.color`: color of these labels.
- `coordinates`: if `TRUE`, the coordinates of the points are plotted below them. `coordinates` and `id` are mutual exclusive.
- `grid`: if `TRUE`, a grid is plotted. May optionally be a string indicating the line type (default: `"dotted"`).
- `grid.color`: grid color.
- `labels, labels.color`: position and color of the grid labels.
- `border`: color of the triangle border.
- `bg`: triangle background.

- `pch`: plotting character. Defaults to filled dots.
- `cex`: a numerical value giving the amount by which plotting text and symbols should be scaled relative to the default. Ignored for the symbol size if `prop.size` is not `FALSE`.
- `prop.size`: if `TRUE`, the symbol size is plotted proportional to the row sum of the three variables, i.e. represents the weight of the observation.
- `col`: plotting color.
- `main`: main title.
- `...`: additional graphics parameters (see `par`).

Examples

```
data(mexico)
z.out <- zelig(as.factor(vote88) ~ pristr + othcok + othsocok,
                model = "mlogit", data = mexico)
x.out <- setx(z.out)
s.out <- sim(z.out, x = x.out)

ternaryplot(s.out$qi$ev, pch = ".", col = "blue",
            main = "1988 Mexican Presidential Election")
```

See Also

- Section 11.6 for a way to add points to an existing ternary diagram.
- Section 5 for an overview of plotting procedures.

Contributors

This function was originally written by David Meyer for the vcd library, version 0.1, which uses plot graphics, and differs from the current implementation in vcd, which uses grid graphics. For additional information on vcd, please consult

Friendly, M. (2000), *Visualizing Categorical Data*, SAS Institute.

11.6 ternarypoints: Adding Points to Ternary Diagrams

Description

Use `ternarypoints()` to add points to a ternary diagram generated using the `ternaryplot()` function in the vcd library. Use ternary diagrams to plot expected values for multinomial choice models with three categories in the dependent variable.

Syntax

```
> ternarypoints(object, pch = 19, col = "blue", ...)
```

Arguments

- `object`: The input object must be a matrix with three columns.
- `pch`: The selected type of point. By default, `pch = 19`, solid disks.
- `col`: The color of the points. By default, `col = "blue"`.
- `...`: Additional parameters passed to `points()`.

Output Values

The `ternarypoints()` command adds points to a previously existing ternary diagram. Use `ternaryplot()` to generate the main ternary diagram.

Examples

```
> ternaryplot(s.out$qi$ev)
> ternarypoints((s.out$qi$ev + s.out$qi$fd))
```

See Also

- Section 11.5 to create a ternary diagram, to which you may add points.
- Section 5 for an overview of plotting procedures

Contributors

Kosuke Imai, Gary King, and Olivia Lau created the `ternarypoints()` function to work with `ternaryplot()`.

Chapter 12

Models Zelig Can Run

This section describes the mathematical components of the models supported by Zelig, using whenever possible the classification and notation of King (1989). Most models have a *stochastic component* (probability density given certain parameters) and a *systematic component* (deterministic functional form that specifies how one or more of the parameters varies over the observed values y_i as a function of the explanatory variables x_i).

Let Y_i be a random outcome variable, realized as $i = 1, \ldots, n$ observations y_i. For the probability density $f(\cdot)$ with systematic feature θ_i varying over i and a scalar ancillary parameter α (constant over i), the stochastic component is given by

$$Y_i \sim f(y_i \mid \theta_i, \alpha).$$

For a functional form $g(\cdot)$, k explanatory variables X_i, and effect parameters β, the systematic component is:

$$\theta_i = g(x_i, \beta).$$

Using the definitions of King, Tomz, and Wittenberg, 2000, Zelig generates at least two quantities of interest:

- The predicted value is a random draw from the stochastic component given random draws of β and α from their sampling (or posterior) distribution.
- The expected value is the *mean* of the stochastic component given random draws of β and α from their sampling (or posterior) distributions. For computational efficiency, Zelig deterministically calculates the expected values from the simulated parameters whenever possible.

Both the predicted values and expected values produced by Zelig can be displayed as histograms or density estimates (to summarize the full sampling or posterior density), or summarized with confidence intervals (by sorting the simulations and taking the 5th and 95th percentile values for a 90% confidence interval for example), standard errors (by taking the standard deviation of the simulations), or point estimates (by averaging the simulations). The point estimate of predicted and expected values are the same only in linear models. In

almost all situations, simulations from predicted values have more variance than expected values. As the number of simulations increases the distribution of the expected values tends toward a constant; the distribution of the predicted values does not collapse as the number of simulations increases.

12.1 blogit: Bivariate Logistic Regression for Two Dichotomous Dependent Variables

Use the bivariate logistic regression model if you have two binary dependent variables (Y_1, Y_2), and wish to model them jointly as a function of some explanatory variables. Each pair of dependent variables (Y_{i1}, Y_{i2}) has four potential outcomes, $(Y_{i1} = 1, Y_{i2} = 1)$, $(Y_{i1} = 1, Y_{i2} = 0)$, $(Y_{i1} = 0, Y_{i2} = 1)$, and $(Y_{i1} = 0, Y_{i2} = 0)$. The joint probability for each of these four outcomes is modeled with three systematic components: the marginal $\Pr(Y_{i1} = 1)$ and $\Pr(Y_{i2} = 1)$, and the odds ratio ψ, which describes the dependence of one marginal on the other. Each of these systematic components may be modeled as functions of (possibly different) sets of explanatory variables.

Syntax

```
> z.out <- zelig(list(mu1 = Y1 ~ X1 + X2 ,
                      mu2 = Y2 ~ X1 + X3),
                 model = "blogit", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Input Values

In every bivariate logit specification, there are three equations which correspond to each dependent variable (Y_1, Y_2), and ψ, the odds ratio. You should provide a list of formulas for each equation or, you may use `cbind()` if the right hand side is the same for both equations

```
formulae <- list(cbind(Y1,Y2) ~ X1 + X2)
```

which means that all the explanatory variables in equations 1 and 2 (corresponding to Y_1 and Y_2) are included, but only an intercept is estimated (all explanatory variables are omitted) for equation 3 (ψ).

You may use the function `tag()` to constrain variables across equations:

```
formulae <- list(mu1 = y1 ~ x1 + tag(x3, "x3"),
                 mu2 = y2 ~ x2 + tag(x3, "x3"))
```

where `tag()` is a special function that constrains variables to have the same effect across equations. Thus, the coefficient for `x3` in equation `mu1` is constrained to be equal to the coefficient for `x3` in equation `mu2`.

Examples

1. Basic Example

   Load the data and estimate the model:

```
> data(sanction)
> z.out1 <- zelig(cbind(import, export) ~ coop + cost + target,
                  model = "blogit", data = sanction)
```

By default, `zelig()` estimates two effect parameters for each explanatory variable in addition to the odds ratio parameter; this formulation is parametrically independent (estimating unconstrained effects for each explanatory variable), but stochastically dependent because the models share an odds ratio.

Generate baseline values for the explanatory variables (with cost set to 1, net gain to sender) and alternative values (with cost set to 4, major loss to sender):

```
> x.low <- setx(z.out1, cost = 1)
> x.high <- setx(z.out1, cost = 4)
```

Simulate fitted values and first differences:

```
> s.out1 <- sim(z.out1, x = x.low, x1 = x.high)
> summary(s.out1)
> plot(s.out1)
```

2. Joint Estimation of a Model with Different Sets of Explanatory Variables

   Load the sample data:

   ```
   > data(sanction)
   ```

   Estimate the statistical model, with `import` a function of `coop` in the first equation and `export` a function of `cost` and `target` in the second equation:

   ```
   > z.out2 <- zelig(list(import ~ coop, export ~ cost + target),
                     model = "blogit", data = sanction)
   > summary(z.out2)
   ```

   Set the explanatory variables to their means:

   ```
   > x.out2 <- setx(z.out2)
   ```

   Simulate draws from the posterior distribution:

   ```
   > s.out2 <- sim(z.out2, x = x.out2)
   > summary(s.out2)
   > plot(s.out2)
   ```

3. Joint Estimation of a Parametrically and Stochastically Dependent Model

   Load the sample data:

   ```
   > data(sanction)
   ```

   A bivariate model is parametrically dependent if Y_1 and Y_2 share some or all explanatory variables, *and* the effects of the shared explanatory variables are jointly estimated. For example,

   ```
   > z.out3 <- zelig(list(import ~ tag(coop,"coop") + tag(cost,"cost") +
                              tag(target,"target"),
                          export ~ tag(coop,"coop") + tag(cost,"cost") +
                              tag(target,"target")),
                          model = "blogit", data = sanction)
   > summary(z.out3)
   ```

   Note that this model only returns one parameter estimate for each of `coop`, `cost`, and `target`. Contrast this to Example 1 which returns two parameter estimates for each of the explanatory variables.

   Set values for the explanatory variables:

   ```
   > x.out3 <- setx(z.out3, cost = 1:4)
   ```

   Draw simulated expected values:

   ```
   > s.out3 <- sim(z.out3, x = x.out3)
   > summary(s.out3)
   ```

Model

For each observation, define two binary dependent variables, Y_1 and Y_2, each of which take the value of either 0 or 1 (in the following, we suppress the observation index). We model the joint outcome (Y_1, Y_2) using a marginal probability for each dependent variable, and the odds ratio, which parameterizes the relationship between the two dependent variables. Define Y_{rs} such that it is equal to 1 when $Y_1 = r$ and $Y_2 = s$ and is 0 otherwise, where r and s take a value of either 0 or 1. Then, the model is defined as follows,

- The *stochastic component* is

$$\begin{aligned} Y_{11} &\sim \text{Bernoulli}(y_{11} \mid \pi_{11}) \\ Y_{10} &\sim \text{Bernoulli}(y_{10} \mid \pi_{10}) \\ Y_{01} &\sim \text{Bernoulli}(y_{01} \mid \pi_{01}) \end{aligned}$$

  where $\pi_{rs} = \Pr(Y_1 = r, Y_2 = s)$ is the joint probability, and $\pi_{00} = 1 - \pi_{11} - \pi_{10} - \pi_{01}$.

- The *systematic components* model the marginal probabilities, $\pi_j = \Pr(Y_j = 1)$, as well as the odds ratio. The odds ratio is defined as $\psi = \pi_{00}\pi_{01}/\pi_{10}\pi_{11}$ and describes the relationship between the two outcomes. Thus, for each observation we have

$$\begin{aligned} \pi_j &= \frac{1}{1+\exp(-x_j\beta_j)} \quad \text{for} \quad j = 1, 2, \\ \psi &= \exp(x_3\beta_3). \end{aligned}$$

Quantities of Interest

- The expected values (`qi$ev`) for the bivariate logit model are the predicted joint probabilities. Simulations of β_1, β_2, and β_3 (drawn from their sampling distributions) are substituted into the systematic components (π_1, π_2, ψ) to find simulations of the predicted joint probabilities:

$$\begin{aligned} \pi_{11} &= \begin{cases} \frac{1}{2}(\psi-1)^{-1} - a - \sqrt{a^2+b} & \text{for } \psi \neq 1 \\ \pi_1\pi_2 & \text{for } \psi = 1 \end{cases}, \\ \pi_{10} &= \pi_1 - \pi_{11}, \\ \pi_{01} &= \pi_2 - \pi_{11}, \\ \pi_{00} &= 1 - \pi_{10} - \pi_{01} - \pi_{11}, \end{aligned}$$

  where $a = 1 + (\pi_1 + \pi_2)(\psi - 1)$, $b = -4\psi(\psi - 1)\pi_1\pi_2$, and the joint probabilities for each observation must sum to one. For n simulations, the expected values form an $n \times 4$ matrix for each observation in `x`.

- The predicted values (`qi$pr`) are draws from the multinomial distribution given the expected joint probabilities.

- The first differences (`qi$fd`) for each of the predicted joint probabilities are given by

$$\text{FD}_{rs} = \Pr(Y_1 = r, Y_2 = s \mid x_1) - \Pr(Y_1 = r, Y_2 = s \mid x).$$

- The risk ratio (`qi$rr`) for each of the predicted joint probabilities are given by

$$\text{RR}_{rs} = \frac{\Pr(Y_1 = r, Y_2 = s \mid x_1)}{\Pr(Y_1 = r, Y_2 = s \mid x)}$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \{Y_{ij}(t_i = 1) - E[Y_{ij}(t_i = 0)]\} \text{ for } j = 1, 2,$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_{ij}(t_i = 0)]$, the counterfactual expected value of Y_{ij} for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_{ij}(t_i = 1) - \widehat{Y_{ij}(t_i = 0)} \right\} \text{ for } j = 1, 2,$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_{ij}(t_i = 0)}$, the counterfactual predicted value of Y_{ij} for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "blogit", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and obtain a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: the named vector of coefficients.
  - `fitted.values`: an $n \times 4$ matrix of the in-sample fitted values.
  - `predictors`: an $n \times 3$ matrix of the linear predictors $x_j \beta_j$.
  - `residuals`: an $n \times 3$ matrix of the residuals.
  - `df.residual`: the residual degrees of freedom.
  - `df.total`: the total degrees of freedom.
  - `rss`: the residual sum of squares.
  - `y`: an $n \times 2$ matrix of the dependent variables.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
- From `summary(z.out)`, you may extract:
  - `coef3`: a table of the coefficients with their associated standard errors and t-statistics.
  - `cov.unscaled`: the variance-covariance matrix.
  - `pearson.resid`: an $n \times 3$ matrix of the Pearson residuals.
- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as arrays indexed by simulation $\times$ quantity $\times$ `x`-observation (for more than one `x`-observation; otherwise the quantities are matrices). Available quantities are:

- `qi$ev`: the simulated expected joint probabilities (or expected values) for the specified values of `x`.
- `qi$pr`: the simulated predicted outcomes drawn from a distribution defined by the expected joint probabilities.
- `qi$fd`: the simulated first difference in the expected joint probabilities for the values specified in `x` and `x1`.
- `qi$rr`: the simulated risk ratio in the predicted probabilities for given `x` and `x1`.
- `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
- `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The bivariate logit function is part of the VGAM package by Thomas Yee. Please cite the model as:

> Yee, T. W. and Hastie, T. J. (2003), "Reduced-rank vector generalized linear models," *Statistical Modelling*, 3, 15–41.

In addition, advanced users may wish to refer to `help(vglm)` in the VGAM library. Additional documentation is available at http://www.stat.auckland.ac.nz/~yee.

Sample data are from

> Martin, L. (1992), *Coercive Cooperation: Explaining Multilateral Economic Sanctions*, Princeton University Press, please inquire with Lisa Martin before publishing results from these data, as this dataset includes errors that have since been corrected.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.2 bprobit: Bivariate Logistic Regression for Two Dichotomous Dependent Variables

Use the bivariate probit regression model if you have two binary dependent variables (Y_1, Y_2), and wish to model them jointly as a function of some explanatory variables. Each pair of dependent variables (Y_{i1}, Y_{i2}) has four potential outcomes, $(Y_{i1} = 1, Y_{i2} = 1)$, $(Y_{i1} = 1, Y_{i2} = 0)$, $(Y_{i1} = 0, Y_{i2} = 1)$, and $(Y_{i1} = 0, Y_{i2} = 0)$. The joint probability for each of these four outcomes is modeled with three systematic components: the marginal $\Pr(Y_{i1} = 1)$ and $\Pr(Y_{i2} = 1)$, and the correlation parameter ρ for the two marginal distributions. Each of these systematic components may be modeled as functions of (possibly different) sets of explanatory variables.

Syntax

```
> z.out <- zelig(list(mu1 = Y1 ~ X1 + X2,
                      mu2 = Y2 ~ X1 + X3,
                      rho = ~ 1),
                 model = "bprobit", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Input Values

In every bivariate probit specification, there are three equations which correspond to each dependent variable (Y_1, Y_2), and the correlation parameter ρ. Since the correlation parameter does not correspond to one of the dependent variables, the model estimates ρ as a constant by default. Hence, only two formulas (for μ_1 and μ_2) are required. If the explanatory variables for μ_1 and μ_2 are the same and effects are estimated separately for each parameter, you may use the following short hand:

```
fml <- list(cbind(Y1,Y2) ~ X1 + X2)
```

which has the same meaning as:

```
fml <- list(mu1 = Y1 ~ X1 + X2,
            mu2 = Y2 ~ X1 + X2,
            rho = ~ 1)
```

You may use the function `tag()` to constrain variables across equations. The `tag()` function takes a variable and a label for the effect parameter. Below, the constrained effect of `x3` in both equations is called the `age` parameter:

```
fml <- list(mu1 = y1 ~ x1 + tag(x3, "age"),
            mu2 = y2 ~ x2 + tag(x3, "age"))
```

You may also constrain different variables across different equations to have the same effect.

Examples

1. Basic Example

   Load the data and estimate the model:

   ```
   > data(sanction)
   > z.out1 <- zelig(cbind(import, export) ~ coop + cost + target,
                     model = "bprobit", data = sanction)
   ```

   By default, `zelig()` estimates two effect parameters for each explanatory variable in addition to the correlation coefficient; this formulation is parametrically independent (estimating unconstrained effects for each explanatory variable), but stochastically dependent because the models share a correlation parameter.

   Generate baseline values for the explanatory variables (with cost set to 1, net gain to sender) and alternative values (with cost set to 4, major loss to sender):

   ```
   > x.low <- setx(z.out1, cost = 1)
   > x.high <- setx(z.out1, cost = 4)
   ```

   Simulate fitted values and first differences:

   ```
   > s.out1 <- sim(z.out1, x = x.low, x1 = x.high)
   > summary(s.out1)
   > plot(s.out1)
   ```

2. Joint Estimation of a Model with Different Sets of Explanatory Variables

   Load the sample data:

   ```
   > data(sanction)
   ```

   Estimate the statistical model, with `import` a function of `coop` in the first equation and `export` a function of `cost` and `target` in the second equation:

   ```
   > fml2 <- list(mu1 = import ~ coop,
                  mu2 = export ~ cost + target)
   > z.out2 <- zelig(fml2, model = "bprobit", data = sanction)
   > summary(z.out2)
   ```

   Set the explanatory variables to their means:

   ```
   > x.out2 <- setx(z.out2)
   ```

Simulate draws from the posterior distribution:

```
> s.out2 <- sim(z.out2, x = x.out2)
> summary(s.out2)
> plot(s.out2)
```

3. Joint Estimation of a Parametrically and Stochastically Dependent Model

   Load the sample data:

   ```
   > data(sanction)
   ```

   A bivariate model is parametrically dependent if Y_1 and Y_2 share some or all explanatory variables, *and* the effects of the shared explanatory variables are jointly estimated. For example,

   ```
   > fml3 <- list(mu1 = import ~ tag(coop,"coop") + tag(cost,"cost") +
                               tag(target,"target"),
                  mu2 = export ~ tag(coop,"coop") + tag(cost,"cost") +
                               tag(target,"target"))
   > z.out3 <- zelig(fml3, model = "bprobit", data = sanction)
   > summary(z.out3)
   ```

   Note that this model only returns one parameter estimate for each of `coop`, `cost`, and `target`. Contrast this to Example 1 which returns two parameter estimates for each of the explanatory variables.

   Set values for the explanatory variables:

   ```
   > x.out3 <- setx(z.out3, cost = 1:4)
   ```

   Draw simulated expected values:

   ```
   > s.out3 <- sim(z.out3, x = x.out3)
   > summary(s.out3)
   ```

Model

For each observation, define two binary dependent variables, Y_1 and Y_2, each of which take the value of either 0 or 1 (in the following, we suppress the observation index i). We model the joint outcome (Y_1, Y_2) using two marginal probabilities for each dependent variable, and the correlation parameter, which describes how the two dependent variables are related.

- The *stochastic component* is described by two latent (unobserved) continuous variables which follow the bivariate Normal distribution:

$$\begin{pmatrix} Y_1^* \\ Y_2^* \end{pmatrix} \sim N_2 \left\{ \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \right\},$$

  where μ_j is a mean for Y_j^* and ρ is a scalar correlation parameter. The following observation mechanism links the observed dependent variables, Y_j, with these latent variables

$$Y_j = \begin{cases} 1 & \text{if } Y_j^* \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

- The *systemic components* for each observation are

$$\begin{aligned} \mu_j &= x_j \beta_j \quad \text{for} \quad j = 1, 2, \\ \rho &= \frac{\exp(x_3 \beta_3) - 1}{\exp(x_3 \beta_3) + 1}. \end{aligned}$$

Quantities of Interest

For n simulations, expected values form an $n \times 4$ matrix.

- The expected values (`qi$ev`) for the binomial probit model are the predicted joint probabilities. Simulations of β_1, β_2, and β_3 (drawn form their sampling distributions) are substituted into the systematic components, to find simulations of the predicted joint probabilities $\pi_{rs} = \Pr(Y_1 = r, Y_2 = s)$:

$$\begin{aligned} \pi_{11} &= \Pr(Y_1^* \geq 0, Y_2^* \geq 0) &&= \int_0^\infty \int_0^\infty \phi_2(\mu_1, \mu_2, \rho)\, dY_2^*\, dY_1^* \\ \pi_{10} &= \Pr(Y_1^* \geq 0, Y_2^* < 0) &&= \int_0^\infty \int_{-\infty}^0 \phi_2(\mu_1, \mu_2, \rho)\, dY_2^*\, dY_1^* \\ \pi_{01} &= \Pr(Y_1^* < 0, Y_2^* \geq 0) &&= \int_{-\infty}^0 \int_0^\infty \phi_2(\mu_1, \mu_2, \rho)\, dY_2^*\, dY_1^* \\ \pi_{11} &= \Pr(Y_1^* < 0, Y_2^* < 0) &&= \int_{-\infty}^0 \int_{-\infty}^0 \phi_2(\mu_1, \mu_2, \rho)\, dY_2^*\, dY_1^* \end{aligned}$$

  where r and s may take a value of either 0 or 1, ϕ_2 is the bivariate Normal density.

- The predicted values (`qi$pr`) are draws from the multinomial distribution given the expected joint probabilities.

- The first difference (`qi$fd`) in each of the predicted joint probabilities are given by

$$\text{FD}_{rs} = \Pr(Y_1 = r, Y_2 = s \mid x_1) - \Pr(Y_1 = r, Y_2 = s \mid x).$$

- The risk ratio (`qi$rr`) for each of the predicted joint probabilities are given by

$$\text{RR}_{rs} = \frac{\Pr(Y_1 = r, Y_2 = s \mid x_1)}{\Pr(Y_1 = r, Y_2 = s \mid x)}.$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \{Y_{ij}(t_i = 1) - E[Y_{ij}(t_i = 0)]\} \text{ for } j = 1, 2,$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_{ij}(t_i = 0)]$, the counterfactual expected value of Y_{ij} for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{Y_{ij}(t_i = 1) - \widehat{Y_{ij}(t_i = 0)}\right\} \text{ for } j = 1, 2,$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_{ij}(t_i = 0)}$, the counterfactual predicted value of Y_{ij} for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "bprobit", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and obtain a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: the named vector of coefficients.
  - `fitted.values`: an $n \times 4$ matrix of the in-sample fitted values.
  - `predictors`: an $n \times 3$ matrix of the linear predictors $x_j\beta_j$.
  - `residuals`: an $n \times 3$ matrix of the residuals.

  - `df.residual`: the residual degrees of freedom.
  - `df.total`: the total degrees of freedom.
  - `rss`: the residual sum of squares.
  - `y`: an $n \times 2$ matrix of the dependent variables.
  - `zelig.data`: the input data frame if `save.data = TRUE`.

- From `summary(z.out)`, you may extract:
  - `coef3`: a table of the coefficients with their associated standard errors and t-statistics.
  - `cov.unscaled`: the variance-covariance matrix.
  - `pearson.resid`: an $n \times 3$ matrix of the Pearson residuals.

- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as arrays indexed by simulation $\times$ quantity $\times$ `x`-observation (for more than one `x`-observation; otherwise the quantities are matrices). Available quantities are:
  - `qi$ev`: the simulated expected values (joint predicted probabilities) for the specified values of `x`.
  - `qi$pr`: the simulated predicted outcomes drawn from a distribution defined by the joint predicted probabilities.
  - `qi$fd`: the simulated first difference in the predicted probabilities for the values specified in `x` and `x1`.
  - `qi$rr`: the simulated risk ratio in the predicted probabilities for given `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The bivariate probit function is part of the VGAM package by Thomas Yee. Please cite this model as:

> Yee, T. W. and Hastie, T. J. (2003), "Reduced-rank vector generalized linear models," *Statistical Modelling*, 3, 15–41.

In addition, advanced users may wish to refer to `help(vglm)` in the VGAM package. Additional documentation is available at http://www.stat.auckland.ac.nz/~yee.

Sample data are from

Martin, L. (1992), *Coercive Cooperation: Explaining Multilateral Economic Sanctions*, Princeton University Press, please inquire with Lisa Martin before publishing results from these data, as this dataset includes errors that have since been corrected.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.3 ei.dynamic: Quinn's Dynamic Ecological Inference Model

Given contingency tables with observed marginals, ecological inference (EI) models estimate each internal cell value for each table. Quinn's dynamic EI model estimates a dynamic Bayesian model for 2×2 tables with temporal dependence across tables (units). The model is implemented using a Markov Chain Monte Carlo algorithm (via a combination of slice and Gibbs sampling). For a hierarchical Bayesian implementation of EI see Quinn's dynamic EI model (Section 12.4). For contingency tables larger than 2 rows by 2 columns, see R×C EI (Section 12.5).

Syntax

```
> z.out <- zelig(cbind(t0, t1) ~ x0 + x1, N = NULL,
                 model = "MCMCei.dynamic", data = mydata)
> x.out <- setx(z.out, fn = NULL, cond = TRUE)
> s.out <- sim(z.out, x = x.out)
```

Inputs

- `t0`, `t1`: numeric vectors (either counts or proportions) containing the column margins of the units to be analyzed.
- `x0`, `x1`: numeric vectors (either counts or proportions) containing the row margins of the units to be analyzed.
- `N`: total counts in each contingency table (unit). If `t0`,`t1`, `x0` and `x1` are proportions, you must specify `N`.

Additional Inputs

In addition, `zelig()` accepts the following additional inputs for `ei.dynamic` to monitor the convergence of the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 5,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 50,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.

- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed of 12345.

The model also accepts the following additional arguments to specify priors and other parameters:

- `W`: a $p \times p$ numeric matrix describing the structure of the temporal dependence among elements of θ_0 and θ_1. The default value is 0, which constructs a weight matrix corresponding to random walk priors for θ_0 and θ_1 (assuming that the tables are equally spaced throughout time, and that the elements of `t0`, `t1`,`x0`,`x1` are temporally ordered).

- `a0`: $a_0/2$ is the shape parameter for the Inverse Gamma prior on σ_0^2. The default is 0.825.

- `b0`: $b_0/2$ is the scale parameter for the Inverse Gamma prior on σ_0^2. The default is 0.0105.

- `a1`: $a_1/2$ is the shape parameter for the Inverse Gamma prior on σ_1^2. The default is 0.825.

- `b1`: $b_1/2$ is the scale parameter for the Inverse Gamma prior on σ_1^2. The default is 0.0105.

Users may wish to refer to `help(MCMCdynamicEI)` for more options.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.

- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.

- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic examples
   Attaching the example dataset:

   ```
   > data(eidat)
   ```

   Estimating the model using `ei.dynamic`:

   ```
   > z.out <- zelig(cbind(t0, t1) ~ x0 + x1, model = "ei.dynamic",
                    data = eidat,  mcmc = 40000, thin = 10, burnin = 10000,
                    verbose = TRUE)
   > summary(z.out)
   ```

   Setting values for in-sample simulations given the marginal values of `t0`, `t1`, `x0`, and `x1`:

   ```
   > x.out <- setx(z.out, fn = NULL, cond = TRUE)
   ```

   In-sample simulations from the posterior distribution:

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

   Summarizing in-sample simulations at aggregate level weighted by the count in each unit:

   ```
   > summary(s.out)
   ```

   Summarizing in-sample simulations at unit level for the first 5 units:

   ```
   > summary(s.out, subset = 1:5)
   ```

Model

Consider the following 2×2 contingency table for the racial voting example. For each geographical unit $i = 1, \ldots, p$, the marginals t_i^0, t_i^1, x_i^0, and x_i^1 are known, and we would like to estimate n_i^{00}, n_i^{01}, n_i^{10}, and n_i^{11}.

	No Vote	Vote	
Black	n_i^{00}	n_i^{01}	x_i^0
White	n_i^{10}	n_i^{11}	x_i^1
	t_i^0	t_i^1	N_i

The marginal values x_i^0, x_i^1, t_i^0, t_i^1 are observed as either counts or fractions. If fractions, the counts can be obtained by multiplying by the total counts per table $N_i = n_i^{00}+n_i^{01}+n_i^{10}+n_i^{11}$, and rounding to the nearest integer. Although there are four internal cells, only two unknowns are modeled since $n_i^{01} = x_i^0 - n_i^{00}$ and $n_i^{11} = s_i^1 - n_i^{10}$.

The hierarchical Bayesian model for ecological inference in 2×2 is illustrated as following:

- The *stochastic component* of the model assumes that

$$\begin{aligned} n_i^{00} \mid x_i^0, \beta_i^b &\sim \text{Binomial}\left(x_i^0, \beta_i^b\right), \\ n_i^{10} \mid x_i^1, \beta_i^w &\sim \text{Binomial}\left(x_i^1, \beta_i^w\right) \end{aligned}$$

  where β_i^b is the fraction of the black voters who vote and β_i^w is the fraction of the white voters who vote. β_i^b and β_i^w as well as their aggregate summaries are the focus of inference.

- The *systematic component* of the model is

$$\begin{aligned} \beta_i^b &= \frac{\exp \theta_i^0}{1 - \exp \theta_i^0} \\ \beta_i^w &= \frac{\exp \theta_i^1}{1 - \exp \theta_i^1} \end{aligned}$$

  The logit transformations of β_i^b and β_i^w, θ_i^0, and θ_i^1 now take value on the real line. (Future versions may allow β_i^b and β_i^w to be functions of observed covariates.)

- The *priors* for θ_i^0 and θ_i^1 are given by

$$\begin{aligned} \theta_i^0 \mid \sigma_0^2 &\propto \frac{1}{\sigma_0^p} \exp\left(-\frac{1}{2\sigma_0^2}\theta_0' P \theta_0\right) \\ \theta_i^1 \mid \sigma_1^2 &\propto \frac{1}{\sigma_1^p} \exp\left(-\frac{1}{2\sigma_1^2}\theta_1' P \theta_1\right) \end{aligned}$$

where P is a $p \times p$ matrix whose off diagonal elements P_{ts} $(t \neq s)$ equal $-W_{ts}$ (the negative values of the corresponding elements of the weight matrix W), and diagonal elements $P_{tt} = \sum_{s \neq t} W_{ts}$. Scale parameters σ_0^2 and σ_1^2 have hyperprior distributions as given below.

- The *hyperpriors* for σ_0^2 and σ_1^2 are given by

$$\begin{aligned} \sigma_0^2 &\sim \text{Inverse Gamma}\left(\frac{a_0}{2}, \frac{b_0}{2}\right), \\ \sigma_1^2 &\sim \text{Inverse Gamma}\left(\frac{a_1}{2}, \frac{b_1}{2}\right), \end{aligned}$$

  where $a_0/2$ and $a_1/2$ are the shape parameters of the (independent) Gamma distributions while $b_0/2$ and $b_1/2$ are the scale parameters.

  The default hyperpriors for μ_0, μ_1, σ_0^2, and σ_1^2 are chosen such that the prior distributions for β^b and β^w are flat.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
> z.out <- (cbind(t0, t1) ~ x0 + x1, N = NULL,
           model = "ei.dynamic", data = mydata)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the quantities of interest by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the parameters.
  - `data`: the name of the input data frame.
  - `N`: the total counts when the inputs are fractions.
  - `seed`: the random seed used in the model.
- From `summary(z.out)`, you may extract:
  - `summary`: a matrix containing the summary information of the posterior estimation of β_i^b andβ_i^w for each unit and the parameters μ_0, μ_1, σ_1 and σ_2 based on the posterior distribution. The first p rows correspond to β_i^b, $i = 1, \ldots p$, the row names are in the form of `p0table`i. The $(p+1)$-th to the $2p$-th rows correspond

to β_i^w, $i = 1, \ldots, p$. The row names are in the form of `p1table`i. The last four rows contain information about μ_0, μ_1, σ_0^2 and σ_1^2, the prior means and variances of θ_0 and θ_1.

- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as arrays indexed by simulation × column × row × observation, where column and row refer to the column dimension and the row dimension of the ecological table, respectively. In this model, only 2×2 contingency tables are analyzed, hence column= 2 and row= 2 in all cases. Available quantities are:
  - `qi$ev`: the simulated expected values of each internal cell given the observed marginals.
  - `qi$pr`: the simulated expected values of each internal cell given the observed marginals.

Contributors

The dynamic EI model was developed in

> Quinn, K. (2004), "Ecological Inference in the Presence of Temporal Dependence," in *Ecological Inference: New Methodological Strategies*, eds. King, G., Rosen, O., and Tanner, M. A., New York: Cambridge University Press.

The function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*.

Sample data are adapted from

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

Ben Goodrich and Ying Lu enabled `ei.dynamic` to work with Zelig.

12.4 `ei.hier`: Hierarchical Ecological Inference Model for 2×2 Tables

Given contingency tables with observed marginals, ecological inference (EI) models estimate each internal cell value for each table. The hierarchical EI model estimates a Bayesian model for 2×2 tables. The model is implemented using a Markov Chain Monte Carlo algorithm (via a combination of slice and Gibbs sampling). For a Bayesian implementation of EI that accounts for temporal dependence, see Quinn's dynamic EI model (Section 12.3). For contingency tables larger than 2 rows by 2 columns, see R×C EI (Section 12.5).

Syntax

```
> z.out <- zelig(cbind(t0, t1) ~ x0 + x1, N = NULL,
                 model = "MCMCei.hier", data = mydata)
> x.out <- setx(z.out, fn = NULL, cond = TRUE)
> s.out <- sim(z.out, x = x.out)
```

Inputs

- `t0`, `t1`: numeric vectors (either counts or proportions) containing the column margins of the units to be analyzed.
- `x0`, `x1`: numeric vectors (either counts or proportions) containing the row margins of the units to be analyzed.
- `N`: total counts per contingency table (unit). If `t0`,`t1`, `x0` and `x1` are proportions, you must specify `N`.

Additional Inputs

In addition, `zelig()` accepts the following additional inputs for `ei.hier` to monitor the convergence of the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 5,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 50,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed of 12345.

The model also accepts the following additional arguments to specify prior parameters used in the model:

- `m0`: prior mean of μ_0 (defaults to 0).
- `M0`: prior variance of μ_0 (defaults to 2.287656).
- `m1`: prior mean of μ_1 (defaults to 0).
- `M1`: prior variance of μ_1 (defaults to 2.287656).
- `a0`: $a_0/2$ is the shape parameter for the Inverse Gamma prior on σ_0^2 (defaults to 0.825).
- `b0`: $b_0/2$ is the scale parameter for the Inverse Gamma prior on σ_0^2 (defaults to 0.0105).
- `a1`: $a_1/2$ is the shape parameter for the Inverse Gamma prior on σ_1^2 (defaults to 0.825).
- `b1`: $b_1/2$ is the scale parameter for the Inverse Gamma prior on σ_1^2 (defaults to 0.0105).

Users may wish to refer to `help(MCMChierEI)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic examples
   Attaching the example dataset:

```
 > data(eidat)
```

   Estimating the model using `ei.hier`:

```
> z.out <- zelig(cbind(t0, t1) ~ x0 + x1, model = "ei.hier",
                 data = eidat, mcmc = 40000, thin = 10, burnin = 10000,
                 verbose = TRUE)
> summary(z.out)
```

   Setting values for in-sample simulations given marginal values of `x0`, `x1`, `t0`, and `t1`:

```
> x.out <- setx(z.out, fn = NULL, cond = TRUE)
```

   In-sample simulations from the posterior distribution:

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

   Summarizing in-sample simulations at aggregate level weighted by the count in each unit:

```
> summary(s.out)
```

   Summarizing in-sample simulations at unit level for the first 5 units:

```
> summary(s.out, subset = 1:5)
```

Model

Consider the following 2×2 contingency table for the racial voting example. For each geographical unit $i = 1, \ldots, p$, the marginals t_i^0, t_i^1, x_i^0, and x_i^1 are known, and we would like to estimate n_i^{00}, n_i^{01}, n_i^{10}, and n_i^{11}.

	No Vote	Vote	
Black	n_i^{00}	n_i^{01}	x_i^0
White	n_i^{10}	n_i^{11}	x_i^1
	t_i^0	t_i^1	N_i

The marginal values x_i^0, x_i^1, t_i^0, t_i^1 are observed as either counts or fractions. If fractions, the counts can be obtained by multiplying by the total counts per table $N_i = n_i^{00} + n_i^{01} + n_i^{10} + n_i^{11}$ and rounding to the nearest integer. Although there are four internal cells, only two unknowns are modeled since $n_i^{01} = x_i^0 - n_i^{00}$ and $n_i^{11} = s_i^1 - n_i^{10}$.

The hierarchical Bayesian model for ecological inference in 2×2 is illustrated as following:

- The *stochastic component* of the model assumes that

$$\begin{aligned} n_i^{00} \mid x_i^0, \beta_i^b &\sim \text{Binomial}\left(x_i^0, \beta_i^b\right), \\ n_i^{10} \mid x_i^1, \beta_i^w &\sim \text{Binomial}\left(x_i^1, \beta_i^w\right) \end{aligned}$$

  where β_i^b is the fraction of the black voters who vote and β_i^w is the fraction of the white voters who vote. β_i^b and β_i^w as well as their aggregate level summaries are the focus of inference.

- The *systematic component* is

$$\begin{aligned} \beta_i^b &= \frac{\exp \theta_i^0}{1 - \exp \theta_i^0} \\ \beta_i^w &= \frac{\exp \theta_i^1}{1 - \exp \theta_i^1} \end{aligned}$$

  The logit transformations of β_i^b and β_i^w, θ_i^0, and θ_i^1 now take value on the real line. (Future versions may allow β_i^b and β_i^w to be functions of observed covariates.)

- The *priors* for θ_i^0 and θ_i^1 are given by

$$\begin{aligned} \theta_i^0 \mid \mu_0, \sigma_0^2 &\sim \text{Normal}\left(\mu_0, \sigma_0^2\right), \\ \theta_i^1 \mid \mu_1, \sigma_1^2 &\sim \text{Normal}\left(\mu_1, \sigma_1^2\right) \end{aligned}$$

  where μ_0 and μ_1 are the means, and σ_0^2 and σ_1^2 are the variances of the two corresponding (independent) normal distributions.

- The *hyperpriors* for μ_0 and μ_1 are given by

$$\begin{aligned} \mu_0 &\sim \text{Normal}\left(m_0, M_0\right), \\ \mu_1 &\sim \text{Normal}\left(m_1, M_1\right), \end{aligned}$$

  where m_0 and m_1 are the means of the (independent) normal distributions while M_0 and M_1 are the variances.

- The *hyperpriors* for σ_0^2 and σ_1^2 are given by

$$\begin{aligned} \sigma_0^2 &\sim \text{Inverse Gamma}\left(\frac{a_0}{2}, \frac{b_0}{2}\right), \\ \sigma_1^2 &\sim \text{Inverse Gamma}\left(\frac{a_1}{2}, \frac{b_1}{2}\right), \end{aligned}$$

  where $a_0/2$ and $a_1/2$ are the shape parameters of the (independent) Gamma distributions while $b_0/2$ and $b_1/2$ are the scale parameters.

  The default hyperpriors for μ_0, μ_1, σ_0^2, and σ_1^2 are chosen such that the prior distributions of β^b and β^w are flat.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run

```
> z.out <- (cbind(t0, t1) ~ x0 + x1, N = NULL,
          model = "ei.hier", data = mydata)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the quantities of interest by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the parameters.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
  - `N`: the total counts when the inputs are fractions.
  - `seed`: the random seed used in the model.
- From `summary(z.out)`, you may extract:

  - `summary`: a matrix containing the summary information of the posterior estimation of β_i^b andβ_i^w for each unit and the parameters μ_0, μ_1, σ_1 and σ_2 based on the posterior distribution. The first p rows correspond to β_i^b, $i = 1, \ldots p$, the row names are in the form of `p0table`i. The $(p+1)$-th to the $2p$-th rows correspond to β_i^w, $i = 1, \ldots, p$. The row names are in the form of `p1table`i. The last four rows contain information about μ_0, μ_1, σ_0^2 and σ_1^2, the prior means and variances of θ_0 and θ_1.

- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as arrays indexed by simulation × column × row × observation, where column and row refer to the column dimension and the row dimension of the contingency table, respectively. In this model, only 2×2 contingency tables are analyzed, hence column= 2 and row= 2 in all cases. Available quantities are:

  - `qi$ev`: the simulated expected values of each internal cell given the observed marginals.
  - `qi$pr`: the simulated expected values of each internal cell given the observed marginals.

Contributors

The hierarchical EI function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*.

Sample data are adapted from

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

Ben Goodrich and Ying Lu enabled `ei.hier` to work with Zelig.

12.5 ei.RxC: Hierarchical Multinomial-Dirichlet Ecological Inference Model for $R \times C$ Tables

Given n contingency tables, each with observed marginals (column and row totals), ecological inference (EI) estimates the internal cell values in each table. The hierarchical Multinomial-Dirichlet model estimates cell counts in $R \times C$ tables. The model is implemented using a nonlinear least squares approximation and, with bootstrapping for standard errors, had good frequentist properties.

Syntax

```
> z.out <- zelig(cbind(T0, T1, T2, T3) ~ X0 + X1,
                 covar = NULL,
                 model = "eiRxC", data = mydata)
> x.out <- setx(z.out, fn = NULL)
> s.out <- sim(z.out)
```

Inputs

- `T0`, `T1`, `T2`,..., `TC`: numeric vectors (either counts, or proportions that sum to one for each row) containing the column margins of the units to be analyzed.
- `X0`, `X1`, `X2`,...,`XR`: numeric vectors (either counts, or proportions that sum to one for each row) containing the row margins of the units to be analyzed.
- `covar`: (optional) a covariate that varies across tables, specified as `covar = ~ Z1`, for example. (The model only accepts one covariate.)

Examples

1. Basic examples: No covariate
   Attaching the example dataset:

   ```
   > data(Weimar)
   ```

   Estimating the model:

   ```
   > z.out <- zelig(cbind(Nazi, Government, Communists, FarRight, Other) ~
                    shareunemployed + shareblue + sharewhite + shareself +
                    sharedomestic, model = "ei.RxC", data = Weimar)
   > summary(z.out)
   ```

   Setting values for in-sample simulations given marginal values:

```
> x.out <- setx(z.out)
```

Estimate fractions of different social groups that support political parties:

```
> s.out <- sim(z.out)
```

Summarizing fractions of different social groups that support political parties:

```
> summary(s.out)
```

2. Example of covariates being present in the model
   Attaching the example dataset:

```
 > data(Weimar)
```

Estimating the model:

```
> z.out <- zelig(cbind(Nazi, Government, Communists, FarRight, Other) ~
                 shareunemployed + shareblue + sharewhite + shareself +
                 sharedomestic,
                 covar = ~ shareprotestants,
                 model = "ei.RxC", data = Weimar)
> summary(z.out)
```

Set the covariate to its default (mean/median) value

```
> x.out <- setx(z.out)
```

Estimate fractions of different social groups that support political parties:

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

Summarizing fractions of different social groups that support political parties:

```
> summary(s.out)
```

Model

Consider the following 5×5 contingency table for the voting patterns in Weimar Germany. For each geographical unit i ($i = 1, \ldots, p$), the marginals $T_{1i}, \ldots, T_{Ci}$, $X_{1i}, \ldots, X_{Ri}$ are known for each of the p electoral precincts, and we would like to estimate $(\beta_i^{rc}, r = 1, \ldots, R, c = 1, \ldots, C-1)$ which are the fractions of people in social class r who vote for party c, for all r and c.

	Nazi	Government	Communists	Far Right	Other	
Unemployed	β_{11}^i	β_{12}^i	β_{13}^i	β_{14}^i	$1 - \sum_{c=1}^4 \beta_{1c}^i$	X_1^i
Blue	β_{21}^i	β_{22}^i	β_{23}^i	β_{24}^i	$1 - \sum_{c=1}^4 \beta_{2c}^i$	X_2^i
White	β_{31}^i	β_{32}^i	β_{33}^i	β_{34}^i	$1 - \sum_{c=1}^4 \beta_{3c}^i$	X_3^i
Self	β_{41}^i	β_{42}^i	β_{43}^i	β_{44}^i	$1 - \sum_{c=1}^4 \beta_{4c}^i$	X_4^i
Domestic	β_{51}^i	β_{52}^i	β_{53}^i	β_{54}^i	$1 - \sum_{c=1}^4 \beta_{5c}^i$	X_5^i
	T_{1i}	T_{2i}	T_{3i}	T_{4i}	$1 - \sum_{c=1}^4 \beta_{ci}$	

The marginal values $X_{1i}, \ldots, X_{Ri}$, $T_{1i}, \ldots, T_{Ci}$ may be observed as counts or fractions.

Let $T_i^{'} = (T_{1i}^{'}, T_{2i}^{'}, \ldots, T_{Ci}^{'})$ be the number of voting age persons who turn out to vote for different parties. There are three levels of hierarchy in the Multinomial-Dirichlet EI model. At the first stage, we model the data as:

- The *stochastic component* is described $T_i^{'}$ which follows a multinomial distribution:

$$T_i^{'} \sim \text{Multinomial}(\Theta_{1i}, \ldots, \Theta_{Ci})$$

- The *systematic components* are

$$\Theta_{ci} = \sum_{r=1}^{R} \beta_{rc}^i X_{ri} \quad \text{for} \quad c = 1, \ldots, C$$

At the second stage, we use an optional covariate to model Θ_{ci}'s and β_{qrc}^i:

- The *stochastic component* is described by $\beta_r^i = (\beta_{r1}, \beta_{r2}, \ldots, \beta_{r,C-1})$ for $i = 1, \ldots, , p$ and $r = 1, \ldots, R$, which follows a Dirichlet distribution:

$$\beta_r^i \sim \text{Dirichlet}(\alpha_{r1}^i, \ldots, \alpha_{rc}^i)$$

- The *systematic components* are

$$\alpha_{rc}^i = \frac{d_r exp(\gamma_{rc} + \delta_{rc} Z_i)}{d_r(1 + \sum_{j=1}^{C-1} exp(\gamma_{rj} + \delta_{rj} Z_i))} = \frac{exp(\gamma_{rc} + \delta_{rc} Z_i)}{1 + \sum_{j=1}^{C-1} exp(\gamma_{rj} + \delta_{rj} Z_i)}$$

for $i = 1, \ldots, p$, $r = 1, \ldots, R$, and $c = 1, \ldots, C-1$.

In the third stage, we assume that the regression parameters (the γ_{rc}'s and δ_{rc}'s) are *a priori* independent, and put a flat prior on these regression parameters. The parameters d_r for $r = 1, \ldots, R$ are assumed to follow exponential distributions with mean $\frac{1}{\lambda}$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run

```
> z.out <- zelig(cbind(T0, T1, T2) ~ X0 + X1 + X2,
          model = "eiRxC", data = mydata)
```

then you may examine the available information in `z.out` by using `names(z.out)`. For example,

- From the `zelig()` output object `z.out$coefficients` are the estimates of γ_{ij} (and also δ_{ij}, if covariates are present). The parameters are returned as a single vector of length $R \times (C-1)$. If there is a covariate, δ is concatenated to it.
- From the `sim()` output object, you may extract the parameters β_{ij} corresponding to the estimated fractions of different social groups that support different political parties, by using `s.out$qi$ev`. For each precinct, that will be a matrix with dimensions: simulations $\times R \times C$.

Contributors

Please cite the model as

> Rosen, O., Jiang, W., King, G., and Tanner, M. A. (2001), "Bayesian and Frequentist Inference for Ecological Inference: The $R \times C$ Case," *Statistica Neerlandica*, 55, 134–156, http://gking.harvard.edu/files/abs/rosen-abs.shtml.

Jason Wittenberg, Ferdinand Alimadhi, and Olivia Lau implemented the $R \times C$ EI model for Zelig.

12.6 exp: Exponential Regression for Duration Dependent Variables

Use the exponential duration regression model if you have a dependent variable representing a duration (time until an event). The model assumes a constant hazard rate for all events. The dependent variable may be censored (for observations have not yet been completed when data were collected).

Syntax

```
> z.out <- zelig(Surv(Y, C) ~ X, model = "exp", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Exponential models require that the dependent variable be in the form `Surv(Y, C)`, where `Y` and `C` are vectors of length n. For each observation i in $1, \ldots, n$, the value y_i is the duration (lifetime, for example), and the associated c_i is a binary variable such that $c_i = 1$ if the duration is not censored (*e.g.*, the subject dies during the study) or $c_i = 0$ if the duration is censored (*e.g.*, the subject is still alive at the end of the study and is know to live at least as long as y_i). If c_i is omitted, all Y are assumed to be completed; that is, time defaults to 1 for all observations.

Input Values

In addition to the standard inputs, `zelig()` takes the following additional options for exponential regression:

- `robust`: defaults to `FALSE`. If `TRUE`, `zelig()` computes robust standard errors based on sandwich estimators (see Huber (1981) and White (1980)) and the options selected in `cluster`.
- `cluster`: if `robust = TRUE`, you may select a variable to define groups of correlated observations. Let `x3` be a variable that consists of either discrete numeric values, character strings, or factors that define strata. Then

  ```
  > z.out <- zelig(y ~ x1 + x2, robust = TRUE, cluster = "x3",
                   model = "exp", data = mydata)
  ```

  means that the observations can be correlated within the strata defined by the variable `x3`, and that robust standard errors should be calculated according to those clusters. If `robust = TRUE` but `cluster` is not specified, `zelig()` assumes that each observation falls into its own cluster.

Example

Attach the sample data:

```
> data(coalition)
```

Estimate the model:

```
> z.out <- zelig(Surv(duration, ciep12) ~ fract + numst2, model = "exp",
                 data = coalition)
```

View the regression output:

```
> summary(z.out)
```

Set the baseline values (with the ruling coalition in the minority) and the alternative values (with the ruling coalition in the majority) for X:

```
> x.low <- setx(z.out, numst2 = 0)
> x.high <- setx(z.out, numst2 = 1)
```

Simulate expected values (`qi$ev`) and first differences (`qi$fd`):

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

Summarize quantities of interest and produce some plots:

```
> summary(s.out)
> plot(s.out)
```

Model

Let Y_i^* be the survival time for observation i. This variable might be censored for some observations at a fixed time y_c such that the fully observed dependent variable, Y_i, is defined as

$$Y_i = \begin{cases} Y_i^* & \text{if } Y_i^* \leq y_c \\ y_c & \text{if } Y_i^* > y_c \end{cases}$$

- The *stochastic component* is described by the distribution of the partially observed variable Y^*. We assume Y_i^* follows the exponential distribution whose density function is given by

  $$f(y_i^* \mid \lambda_i) = \frac{1}{\lambda_i} \exp\left(-\frac{y_i^*}{\lambda_i}\right)$$

  for $y_i^* \geq 0$ and $\lambda_i > 0$. The mean of this distribution is λ_i.

  In addition, survival models like the exponential have three additional properties. The hazard function $h(t)$ measures the probability of not surviving past time t given survival up to t. In general, the hazard function is equal to $f(t)/S(t)$ where the survival function

$S(t) = 1 - \int_0^t f(s)ds$ represents the fraction still surviving at time t. The cumulative hazard function $H(t)$ describes the probability of dying before time t. In general, $H(t) = \int_0^t h(s)ds = -\log S(t)$. In the case of the exponential model,

$$\begin{aligned} h(t) &= \frac{1}{\lambda_i} \\ S(t) &= \exp\left(-\frac{t}{\lambda_i}\right) \\ H(t) &= \frac{t}{\lambda_i} \end{aligned}$$

For the exponential model, the hazard function $h(t)$ is constant over time. The Weibull model and lognormal models allow the hazard function to vary as a function of elapsed time (see Section 12.32 and Section 12.15 respectively).

- The *systematic component* λ_i is modeled as

$$\lambda_i = \exp(x_i\beta),$$

where x_i is the vector of explanatory variables, and β is the vector of coefficients.

Quantities of Interest

- The expected values (`qi$ev`) for the exponential model are simulations of the expected duration given x_i and draws of β from its posterior,

$$E(Y) = \lambda_i = \exp(x_i\beta).$$

- The predicted values (`qi$pr`) are draws from the exponential distribution with rate equal to the expected value.

- The first difference (or difference in expected values, `qi$ev.diff`), is

$$\text{FD} \;=\; E(Y \mid x_1) - E(Y \mid x), \tag{12.1}$$

where x and x_1 are different vectors of values for the explanatory variables.

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^n t_i} \sum_{i:t_i=1}^{n} \left\{Y_i(t_i = 1) - E[Y_i(t_i = 0)]\right\},$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. When $Y_i(t_i = 1)$ is censored rather than observed, we replace it with

a simulation from the model given available knowledge of the censoring process. Variation in the simulations is due to two factors: uncertainty in the imputation process for censored y_i^* and uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. When $Y_i(t_i = 1)$ is censored rather than observed, we replace it with a simulation from the model given available knowledge of the censoring process. Variation in the simulations is due to two factors: uncertainty in the imputation process for censored y_i^* and uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(Surv(Y, C) ~ X, model = "exp", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `icoef`: parameter estimates for the intercept and scale parameter. While the scale parameter varies for the Weibull distribution, it is fixed to 1 for the exponential distribution (which is modeled as a special case of the Weibull).
  - `var`: the variance-covariance matrix for the estimates of β.
  - `loglik`: a vector containing the log-likelihood for the model and intercept only (respectively).
  - `linear.predictors`: the vector of $x_i\beta$.
  - `df.residual`: the residual degrees of freedom.
  - `df.null`: the residual degrees of freedom for the null model.
  - `zelig.data`: the input data frame if `save.data = TRUE`.

- Most of this may be conveniently summarized using `summary(z.out)`. From `summary(z.out)`, you may additionally extract:
  - `table`: the parameter estimates with their associated standard errors, p-values, and t-statistics. For example, `summary(z.out)$table`
- From the `sim()` output stored in `s.out`:
- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:
  - `qi$ev`: the simulated expected values for the specified values of `x`.
  - `qi$pr`: the simulated predicted values drawn from a distribution defined by the expected values.
  - `qi$fd`: the simulated first differences between the simulated expected values for `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The exponential function is part of the survival library by Terry Therneau, ported to R by Thomas Lumley. Advanced users may wish to refer to `help(survfit)` in the survival library and

Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

Sample data are from

King, G., Alt, J., Burns, N., and Laver, M. (1990), "A Unified Model of Cabinet Dissolution in Parliamentary Democracies," *American Journal of Political Science*, 34, 846–871, http://gking.harvard.edu/files/abs/coal-abs.shtml.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.7 `factor.bayes`: Bayesian Factor Analysis

Given some unobserved explanatory variables and observed dependent variables, the Normal theory factor analysis model estimates the latent factors. The model is implemented using a Markov Chain Monte Carlo algorithm (Gibbs sampling with data augmentation). For factor analysis with ordinal dependent variables, see ordered factor analysis (Section 12.9), and for a mix of types of dependent variables, see the mixed factor analysis model (Section 12.8).

Syntax

```
> z.out <- zelig(cbind(Y1 ,Y2, Y3) ~ NULL, factors = 2,
                 model = "factor.bayes", data = mydata)
```

Inputs

`zelig()` takes the following functions for `factor.bayes`:

- `Y1`, `Y2`, and `Y3`: variables of interest in factor analysis (manifest variables), assumed to be normally distributed. The model requires a minimum of three manifest variables.
- `factors`: number of the factors to be fitted (defaults to 2).

Additional Inputs

In addition, `zelig()` accepts the following additional arguments for model specification:

- `lambda.constraints`: list containing the equality or inequality constraints on the factor loadings. Choose from one of the following forms:
  - `varname = list()`: by default, no constraints are imposed.
  - `varname = list(d, c)`: constrains the dth loading for the variable named `varname` to be equal to `c`.
  - `varname = list(d, "+")`: constrains the dth loading for the variable named `varname` to be positive;
  - `varname = list(d, "-")`: constrains the dth loading for the variable named `varname` to be negative.
- `std.var`: defaults to `FALSE` (manifest variables are rescaled to zero mean, but retain observed variance). If `TRUE`, the manifest variables are rescaled to be mean zero and unit variance.

In addition, `zelig()` accepts the following additional inputs for `bayes.factor`:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 20,000).

- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.

- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.

- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed 12345.

- `Lambda.start`: starting values of the factor loading matrix Λ, either a scalar (all unconstrained loadings are set to that value), or a matrix with compatible dimensions. The default is `NA`, where the start value are set to be 0 for unconstrained factor loadings, and 0.5 or -0.5 for constrained factor loadings (depending on the nature of the constraints).

- `Psi.start`: starting values for the uniquenesses, either a scalar (the starting values for all diagonal elements of Ψ are set to be this value), or a vector with length equal to the number of manifest variables. In the latter case, the starting values of the diagonal elements of Ψ take the values of `Psi.start`. The default value is `NA` where the starting values of the all the uniquenesses are set to be 0.5.

- `store.lambda`: defaults to `TRUE`, which stores the posterior draws of the factor loadings.

- `store.scores`: defaults to `FALSE`. If `TRUE`, stores the posterior draws of the factor scores. (Storing factor scores may take large amount of memory for a large number of draws or observations.)

The model also accepts the following additional arguments to specify prior parameters:

- `l0`: mean of the Normal prior for the factor loadings, either a scalar or a matrix with the same dimensions as Λ. If a scalar value, that value will be the prior mean for all the factor loadings. Defaults to 0.

- `L0`: precision parameter of the Normal prior for the factor loadings, either a scalar or a matrix with the same dimensions as Λ. If `L0` takes a scalar value, then the precision matrix will be a diagonal matrix with the diagonal elements set to that value. The default value is 0, which leads to an improper prior.

- `a0`: the shape parameter of the Inverse Gamma prior for the uniquenesses is `a0/2`. It can take a scalar value or a vector. The default value is 0.001.

- `b0`: the shape parameter of the Inverse Gamma prior for the uniquenesses is `b0/2`. It can take a scalar value or a vector. The default value is 0.001.

Zelig users may wish to refer to `help(MCMCfactanal)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(swiss)
   > names(swiss) <- c("Fert","Agr","Exam","Educ","Cath","InfMort")
   ```

   Factor analysis:

   ```
   > z.out <- zelig(cbind(Agr, Exam, Educ, Cath, InfMort) ~ NULL,
                    model = "factor.bayes", data = swiss, factors = 2,
            verbose = TRUE, a0 = 1, b0 = 0.15,
                    burnin = 5000, mcmc = 50000)
   ```

   Checking for convergence before summarizing the estimates:

   ```
   > geweke.diag(z.out$coefficients)
   ```

Since the algorithm did not converge, we now add some constraints on Λ.

2. Putting Constraints on Λ
   Put constraints on Lambda to optimize the algorithm:

```
z.out <- zelig(cbind(Agr, Exam, Educ, Cath, InfMort) ~ NULL,
               model = "factor.bayes", data = swiss, factors = 2,
               lambda.constraints = list(Exam=list(1,"+"),
                                         Exam=list(2,"-"), Educ=c(2,0),
                                         InfMort=c(1,0)),
               verbose = TRUE, a0 = 1, b0 = 0.15,
               burnin = 5000, mcmc = 50000)
```

```
> geweke.diag(z.out$coefficients)
> heidel.diag(z.out$coefficients)
> raftery.diag(z.out$coefficients)
> summary(z.out)
```

Model

Suppose for observation i we observe K variables and hypothesize that there are d underlying factors such that:

$$Y_i = \Lambda\phi_i + \epsilon_i$$

where Y_i is the vector of K manifest variables for observation i. Λ is the $K \times d$ factor loading matrix and ϕ_i is the d-vector of latent factor scores. Both Λ and ϕ need to be estimated.

- The *stochastic component* is given by:

  $$\epsilon_i \sim \text{Normal}(0, \Psi).$$

  where Ψ is a diagonal, positive definite matrix. The diagonal elements of Ψ are referred to as uniquenesses.

- The *systematic component* is given by

  $$\mu_i = E(Y_i) = \Lambda\phi_i$$

- The independent conjugate *prior* for each Λ_{ij} is given by

  $$\Lambda_{ij} \sim \text{Normal}(l_{0_{ij}}, L_{0_{ij}}^{-1}) \text{ for } i = 1, \ldots, k; \quad j = 1, \ldots, d.$$

- The independent conjugate *prior* for each Ψ_{ii} is given by

  $$\Psi_{ii} \sim \text{InverseGamma}(\frac{a_0}{2}, \frac{b_0}{2}), \text{ for } i = 1, \ldots, k.$$

- The *prior* for ϕ_i is

$$\phi_i \sim \text{Normal}(0, I_d), \text{ for } i = 1, \ldots, n.$$

  where I_d is a $d \times d$ identity matrix.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(cbind(Y1, Y2, Y3), model = "factor.bayes", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated factor loadings and the uniquenesses. If `store.scores = TRUE`, the estimated factors scores are also contained in `coefficients`.
  - `data`: the name of the input data frame.
  - `seed`: the random seed used in the model.
- Since there are no explanatory variables, the `sim()` procedure is not applicable for factor analysis models.

Contributors

Bayesian factor analysis function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*.

Sample data are adapted from

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

Ben Goodrich and Ying Lu enabled `factor.bayes` to work with Zelig.

12.8 `factor.mix`: Mixed Data Factor Analysis

Mixed data factor analysis takes both continuous and ordinal dependent variables and estimates a model for a given number of latent factors. The model is estimated using a Markov Chain Monte Carlo algorithm (Gibbs sampler with data augmentation). Alternative models include Bayesian factor analysis for continuous variables (Section 12.7) and Bayesian factor analysis for ordinal variables (Section 12.9).

Syntax

```
> z.out <- zelig(cbind(Y1 ,Y2, Y3) ~ NULL, factors = 1,
                 model = "factor.mix", data = mydata)
```

Inputs

`zelig()` accepts the following arguments for `factor.mix`:

- `Y1`, `Y2`, `Y3`, `...`: The dependent variables of interest, which can be a mix of ordinal and continuous variables. You must have more dependent variables than factors.
- `factors`: The number of the factors to be fitted.

Additional Inputs

The model accepts the following additional arguments to monitor convergence:

- `lambda.constraints`: A list that contains the equality or inequality constraints on the factor loadings.
  - `varname = list()`: by default, no constraints are imposed.
  - `varname = list(d, c)`: constrains the dth loading for the variable named `varname` to be equal to `c`.
  - `varname = list(d, "+")`: constrains the dth loading for the variable named `varname` to be positive;
  - `varname = list(d, "-")`: constrains the dth loading for the variable named `varname` to be negative.

  Unlike Bayesian factor analysis for continuous variables (Section 12.7), the first column of Λ corresponds to negative item difficulty parameters and should not be constrained in general.
- `std.mean`: defaults to `TRUE`, which rescales the continuous manifest variables to have mean 0.

- `std.var`: defaults to `TRUE`. which rescales the continuous manifest variables to have unit variance.

`factor.mix` accepts the following additional arguments to monitor the sampling scheme for the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded. The default value is 1,000.
- `mcmc`: number of the MCMC iterations after burnin. The default value is 20,000.
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `tune`: tuning parameter, which can be either a scalar or a vector of length K. The value of the tuning parameter must be positive. The default value is `1.2`.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen. The default is `FALSE`.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed 12345.
- `lambda.start`: starting values of the factor loading matrix Λ for the Markov chain, either a scalar (starting values of the unconstrained loadings will be set to that value), or a matrix with compatible dimensions. The default is `NA`, where the start values for the first column of Λ are set based on the observed pattern, while for the rest of the columns of Λ, the start values are set to be 0 for unconstrained factor loadings, and 1 or -1 for constrained factor loadings (depending on the nature of the constraints).
- `psi.start`: starting values for the diagonals of the error variance (uniquenesses) matrix. Since the starting values for the ordinal variables are constrained to 1 (to identify the model), you may only specify the starting values for the continuous variables. For the continuous variables, you may specify `psi.start` as a scalar or a vector with length equal to the number of continuous variables. If a scalar, that starting value is recycled for all continuous variables. If a vector, the starting values should correspond to each of the continuous variables. The default value is `NA`, which means the starting values of all the continuous variable uniqueness are set to 0.5.
- `store.lambda`: defaults to `TRUE`, storing the posterior draws of the factor loadings.
- `store.scores`: defaults to `FALSE`. If `TRUE`, the posterior draws of the factor scores are stored. (Storing factor scores may take large amount of memory for a a large number of draws or observations.)

Use the following additional arguments to specify prior parameters used in the model:

- `l0`: mean of the Normal prior for the factor loadings, either a scalar or a matrix with the same dimensions as Λ. If a scalar value, then that value will be the prior mean for all the factor loadings. The default value is 0.

- `L0`: precision parameter of Normal prior for the factor loadings, either a scalar or a matrix with the same dimensions as Λ. If a scalar value, then the precision matrix will be a diagonal matrix with the diagonal elements set to that value. The default value is 0 which leads to an improper prior.

- `a0`: `a0/2` is the shape parameter of the Inverse Gamma priors for the uniquenesses. It can take a scalar value or a vector. The default value is 0.001.

- `b0`: `b0/2` is the shape parameter of the Inverse Gamma priors for the uniquenesses. It can take a scalar value or a vector. The default value is 0.001.

Zelig users may wish to refer to `help(MCMCmixfactanal)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.

- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.

- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(PErisk)
   ```

   Factor analysis for mixed data using `factor.mix`:

   ```
   > z.out<-zelig(cbind(courts, barb2, prsexp2, prscorr2, gdpw2) ~ NULL,
                  data = PErisk, model = "factor.mix", factors = 1,
                  burnin = 5000, mcmc = 100000, thin = 50, verbose = TRUE,
                  L0 = 0.25)
   ```

   Checking for convergence before summarizing the estimates:

   ```
   > geweke.diag(z.out$coefficients)
   > heidel.diag(z.out$coefficients)
   > summary(z.out)
   ```

Model

Let Y_i be a K-vector of observed variables for observation i, The kth variable can be either continuous or ordinal. When Y_{ik} is an ordinal variable, it takes value from 1 to J_k for $k = 1, \ldots, K$ and for $i = 1, \ldots, n$. The distribution of Y_{ik} is assumed to be governed by another K-vector of unobserved continuous variable Y_{ik}^*. There are d underlying factors. When Y_{ik} is continuous, we let $Y_{ik}^* = Y_{ik}$.

- The *stochastic component* is described in terms of Y_i^*:

  $$Y_i^* \sim \text{Normal}_K(\mu_i, I_K),$$

  where $Y_i^* = (Y_{i1}^*, \ldots, Y_{iK}^*)$, and $\mu_i = (\mu_{i1}, \ldots, \mu_{iK})$.

  For ordinal Y_{ik},

  $$Y_{ik} = j \quad \text{if} \quad \gamma_{(j-1),k} \leq Y_{ik}^* \leq \gamma_{jk} \quad \text{for} \quad j = 1, \ldots, J_k; k = 1, \ldots, K.$$

  where $\gamma_{jk}, j = 0, \ldots, J$ are the threshold parameters for the kth variable with the following constraints, $\gamma_{lk} < \gamma_{mk}$ for $l < m$, and $\gamma_{0k} = -\infty, \gamma_{J_k k} = \infty$ for any $k = 1, \ldots, K$. It follows that the probability of observing Y_{ik} belonging to category j is,

  $$\Pr(Y_{ik} = j) = \Phi(\gamma_{jk} \mid \mu_{ik}) - \Phi(\gamma_{(j-1),k} \mid \mu_{ik}) \quad \text{for } j = 1, \ldots, J_k$$

  where $\Phi(\cdot \mid \mu_{ik})$ is the cumulative distribution function of the Normal distribution with mean μ_{ik} and variance 1.

- The *systematic component* is given by,

$$\mu_i \quad = \quad \Lambda\phi_i,$$

  where Λ is a $K \times d$ matrix of factor loadings for each variable, ϕ_i is a d-vector of factor scores for observation i. Note both Λ and ϕ are estimated..

- The independent conjugate *prior* for each Λ_{ij} is given by

$$\Lambda_{ij} \quad \sim \quad \text{Normal}(l_{0_{ij}}, L_{0_{ij}}^{-1}) \text{ for } i = 1, \ldots, k; \quad j = 1, \ldots, d.$$

- The *prior* for ϕ_i is,

$$\phi_i \sim \text{Normal}(0, I_{d-1}), \quad \text{for} \quad i = 2, \ldots, n.$$

  where I_{d-1} is a $(d-1) \times (d-1)$ identity matrix. Note the first element of ϕ_i is 1.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(cbind(Y1, Y2, Y3), model = "factor.mix", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated factor loadings, the estimated cut points γ for each variable. Note the first element of γ is normalized to be 0. If `store.scores = TRUE`, the estimated factors scores are also contained in `coefficients`.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
  - `seed`: the random seed used in the model.
- Since there are no explanatory variables, the `sim()` procedure is not applicable for factor analysis models.

Contributors

Factor analysis for mixed dependent variables function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*.

Sample data are adapted from

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

Ben Goodrich and Ying Lu enabled `factor.mix` to work with Zelig.

12.9 `factor.ord`: Ordinal Data Factor Analysis

Given some unobserved explanatory variables and observed ordinal dependent variables, this model estimates latent factors using a Gibbs sampler with data augmentation. For factor analysis for continuous data, see Section 12.7. For factor analysis for mixed data (including both continuous and ordinal variables), see Section 12.8.

Syntax

```
> z.out <- zelig(cbind(Y1 ,Y2, Y3) ~ NULL, factors = 1,
                 model = "factor.ord", data = mydata)
```

Inputs

`zelig()` accepts the following arguments for `factor.ord`: :

- `Y1, Y2`, and `Y3`: variables of interest in factor analysis (manifest variables), assumed to be ordinal variables. The number of manifest variables must be greater than the number of the factors.
- `factors`: number of the factors to be fitted (defaults to 1).

Additional Inputs

In addition, `zelig()` accepts the following arguments for model specification:

- `lambda.constraints`: list that contains the equality or inequality constraints on the factor loadings. A typical entry in the list has one of the following forms:
  - `varname = list()`: by default, no constraints are imposed.
  - `varname = list(d, c)`: constrains the dth loading for the variable named `varname` to be equal to `c`;
  - `varname = list(d, "+")`: constrains the dth loading for the variable named `varname` to be positive;
  - `varname = list(d, "-")`: constrains the dth loading for the variable named `varname` to be negative.

  The first column of Λ should not be constrained in general.
- `drop.constantvars`: defaults to `TRUE`, dropping the manifest variables that have no variation before fitting the model.

The model accepts the following arguments to monitor the convergence of the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).

- `mcmc`: number of the MCMC iterations after burnin (defaults to 20,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `tune`: tuning parameter for Metropolis-Hasting sampling, either a scalar or a vector of length K. The value of the tuning parameter must be positive. The default value is 1.2.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed 12345.
- `Lambda.start`: starting values of the factor loading matrix Λ for the Markov chain, either a scalar (all unconstrained loadings are set to that value), or a matrix with compatible dimensions. The default is `NA`, such that the start values for the first column are set based on the observed pattern, while the remaining columns have start values set to 0 for unconstrained factor loadings, and -1 or 1 for constrained loadings (depending on the nature of the constraints).
- `store.lambda`: defaults to `TRUE`, which stores the posterior draws of the factor loadings.
- `store.scores`: defaults to `FALSE`. If `TRUE`, stores the posterior draws of the factor scores. (Storing factor scores may take large amount of memory for a a large number of draws or observations.)

Use the following parameters to specify the model's priors:

- `l0`: mean of the Normal prior for the factor loadings, either a scalar or a matrix with the same dimensions as Λ. If a scalar value, that value will be the prior mean for all the factor loadings. Defaults to 0.
- `L0`: precision parameter of the Normal prior for the factor loadings, either a scalar or a matrix with the same dimensions as Λ. If `L0` takes a scalar value, then the precision matrix will be a diagonal matrix with the diagonal elements set to that value. The default value is 0, which leads to an improper prior.

Zelig users may wish to refer to `help(MCMCordfactanal)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(newpainters)
   ```

   Factor analysis for ordinal data using `factor.ord`:

   ```
   > z.out <- zelig(cbind(Composition,Drawing,Colour,Expression)~NULL,
                       data=newpainters, model="factor.ord",
                       factors=1, L0=0.5,
                       burin=1000,mcmc=10000, thin=5, verbose=TRUE)
   ```

   Checking for convergence before summarizing the estimates:

   ```
   > geweke.diag(z.out$coefficients)
   > heidel.diag(z.out$coefficients)
   > raftery.diag(z.out$coefficients)
   > summary(z.out)
   ```

Model

Let Y_i be a vector of K observed ordinal variables for observation i, each ordinal variable k for $k = 1, \ldots, K$ takes integer value $j = 1, \ldots, J_k$. The distribution of Y_i is assumed to be governed by another k-vector of unobserved continuous variable Y_i^*. There are d underlying factors.

- The *stochastic component* is described in terms of the latent variable Y_i^*:

$$Y_i^* \sim \text{Normal}_K(\mu_i, I_K),$$

  where $Y_i^* = (Y_{i1}^*, \ldots, Y_{iK}^*)$, and μ_i is the mean vector for Y_i^*, and $\mu_i = (\mu_{i1}, \ldots, \mu_{iK})$. Instead of Y_{ik}^*, we observe ordinal variable Y_{ik},

$$Y_{ik} = j \text{ if } \gamma_{(j-1),k} \leq Y_{ik}^* \leq \gamma_{jk} \text{ for } \quad j = 1, \ldots, J_k, k = 1, \ldots, K.$$

  where $\gamma_{jk}, j = 0, \ldots, J$ are the threshold parameters for the kth variable with the following constraints, $\gamma_{lk} < \gamma_{mk}$ for $l < m$, and $\gamma_{0k} = -\infty, \gamma_{J_k k} = \infty$ for any $k = 1, \ldots, K$. It follows that the probability of observing Y_{ik} belonging to category j is,

$$\Pr(Y_{ik} = j) = \Phi(\gamma_{jk} \mid \mu_{ik}) - \Phi(\gamma_{(j-1),k} \mid \mu_{ik}) \text{ for } j = 1, \ldots, J_k$$

  where $\Phi(\cdot \mid \mu_{ik})$ is the cumulative distribution function of the Normal distribution with mean μ_{ik} and variance 1.

- The *systematic component* is given by,

$$\mu_i = \Lambda \phi_i,$$

  where Λ is a $K \times d$ matrix of factor loadings for each variable, ϕ_i is a d-vector of factor scores for observation i. Note both Λ and ϕ need to be estimated.

- The independent conjugate *prior* for each element of Λ, Λ_{ij} is given by

$$\Lambda_{ij} \sim \text{Normal}(l_{0_{ij}}, L_{0_{ij}}^{-1}) \text{ for } i = 1, \ldots, k; \quad j = 1, \ldots, d.$$

- The *prior* for ϕ_i is,

$$\phi_{i(2:d)} \sim \text{Normal}(0, I_{d-1}), \text{ for } \quad i = 2, \ldots, n.$$

  where I_{d-1} is a $(d-1) \times (d-1)$ identity matrix. Note the first element of ϕ_i is 1.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(cbind(Y1, Y2, Y3), model = "factor.ord", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated factor loadings, the estimated cut points γ for each variable. Note the first element of γ is normalized to be 0. If `store.scores=TRUE`, the estimated factors scores are also contained in `coefficients`.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
  - `seed`: the random seed used in the model.
- Since there are no explanatory variables, the `sim()` procedure is not applicable for factor analysis models.

Contributors

The `factor.ord` function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*.

Sample data are adapted from

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

Ben Goodrich and Ying Lu enabled `factor.ord` to work with Zelig.

12.10 gamma: Gamma Regression for Continuous, Positive Dependent Variables

Use the gamma regression model if you have a positive-valued dependent variable such as the number of years a parliamentary cabinet endures, or the seconds you can stay airborne while jumping. The gamma distribution assumes that all waiting times are complete by the end of the study (censoring is not allowed).

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "gamma", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out, x1 = NULL)
```

Additional Inputs

In addition to the standard inputs, `zelig()` takes the following additional options for gamma regression:

- `robust`: defaults to `FALSE`. If `TRUE` is selected, `zelig()` computes robust standard errors via the `sandwich` package (see Zeileis (2004)). The default type of robust standard error is heteroskedastic and autocorrelation consistent (HAC), and assumes that observations are ordered by time index.

  In addition, `robust` may be a list with the following options:

  - `method`: Choose from
    - `"vcovHAC"`: (default if `robust = TRUE`) HAC standard errors.
    - `"kernHAC"`: HAC standard errors using the weights given in Andrews (1991).
    - `"weave"`: HAC standard errors using the weights given in Lumley and Heagerty (1999).
  - `order.by`: defaults to `NULL` (the observations are chronologically ordered as in the original data). Optionally, you may specify a vector of weights (either as `order.by = z`, where `z` exists outside the data frame; or as `order.by = ~z`, where `z` is a variable in the data frame). The observations are chronologically ordered by the size of `z`.
  - `...`: additional options passed to the functions specified in `method`. See the `sandwich` library and Zeileis (2004) for more options.

Example

Attach the sample data:

```
> data(coalition)
```

Estimate the model:

```
> z.out <- zelig(duration ~ fract + numst2, model = "gamma", data = coalition)
```

View the regression output:

```
> summary(z.out)
```

Set the baseline values (with the ruling coalition in the minority) and the alternative values (with the ruling coalition in the majority) for X:

```
> x.low <- setx(z.out, numst2 = 0)
> x.high <- setx(z.out, numst2 = 1)
```

Simulate expected values (`qi$ev`) and first differences (`qi$fd`):

```
> s.out <- sim(z.out, x = x.low, x1 = x.high)
> summary(s.out)
> plot(s.out)
```

Model

- The Gamma distribution with scale parameter α has a *stochastic component*:

$$\begin{aligned} Y &\sim \text{Gamma}(y_i \mid \lambda_i, \alpha) \\ f(y) &= \frac{1}{\alpha^{\lambda_i}\,\Gamma\lambda_i}\, y_i^{\lambda_i - 1} \exp - \left\{\frac{y_i}{\alpha}\right\} \end{aligned}$$

  for $\alpha, \lambda_i, y_i > 0$.

- The *systematic component* is given by

$$\lambda_i = \frac{1}{x_i \beta}$$

Quantities of Interest

- The expected values (`qi$ev`) are simulations of the mean of the stochastic component given draws of α and β from their posteriors:

$$E(Y) = \alpha_i \lambda.$$

- The predicted values (`qi$pr`) are draws from the gamma distribution for each given set of parameters (α, λ_i).

- If `x1` is specified, `sim()` also returns the differences in the expected values (`qi$fd`),

$$E(Y \mid x_1) - E(Y \mid x)$$

.

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - E[Y_i(t_i = 0)] \right\},$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "gamma", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `residuals`: the working residuals in the final iteration of the IWLS fit.
  - `fitted.values`: the vector of fitted values.
  - `linear.predictors`: the vector of $x_i\beta$.

- `aic`: Akaike's Information Criterion (minus twice the maximized log-likelihood plus twice the number of coefficients).
- `df.residual`: the residual degrees of freedom.
- `df.null`: the residual degrees of freedom for the null model.
- `zelig.data`: the input data frame if `save.data = TRUE`.

- From `summary(z.out)`, you may extract:
  - `coefficients`: the parameter estimates with their associated standard errors, p-values, and t-statistics.
  - `cov.scaled`: a $k \times k$ matrix of scaled covariances.
  - `cov.unscaled`: a $k \times k$ matrix of unscaled covariances.

- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:
  - `qi$ev`: the simulated expected values for the specified values of `x`.
  - `qi$pr`: the simulated predicted values drawn from a distribution defined by (α_i, λ).
  - `qi$fd`: the simulated first difference in the expected values for the specified values in `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The gamma model is part of the stats package by William N. Venables and Brian D. Ripley. Users should cite this model as:

> Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

In addition, advanced users may wish to refer to `help(glm)` and `help(family)`, as well as

> McCullagh, P. and Nelder, J. A. (1989), *Generalized Linear Models*, no. 37 in Monograph on Statistics and Applied Probability, Chapman & Hall, 2nd ed.

Robust standard errors are implemented via the sandwich package by Achim Zeileis. Please cite as

Zeileis, A. (2004), "Econometric Computing with HC and HAC Covariance Matrix Estimators," *Journal of Statistical Software*, 11, 1–17

Sample data are from

King, G., Alt, J., Burns, N., and Laver, M. (1990), "A Unified Model of Cabinet Dissolution in Parliamentary Democracies," *American Journal of Political Science*, 34, 846–871, http://gking.harvard.edu/files/abs/coal-abs.shtml.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.11 `irt1d`: One Dimensional Item Response Model

Given several observed dependent variables and an unobserved explanatory variable, item response theory estimates the latent variable (ideal points). The model is estimated using the Markov Chain Monte Carlo algorithm via a Gibbs sampler and data augmentation. Use this model if you believe that the ideal points lie in one dimension, and see the k-dimensional item response model (Section 12.12) for k hypothesized latent variables.

Syntax

```
> z.out <- zelig(cbind(Y1, Y2, Y3) ~ NULL, model = "irt1d", data = mydata)
```

Inputs

`irt1d` accepts the following argument:

- `Y1, Y2`, and `Y3`: `Y1` contains the items for subject "Y1", `Y2` contains the items for subject "Y2", and so on.

Additional arguments

`irt1d` accepts the following additional arguments for model specification:

- `theta.constraints`: a list specifying possible equality or inequality constraints on the ability parameters θ. A typical entry takes one of the following forms:
  - `varname = list()`: by default, no constraints are imposed.
  - `varname = c`: constrains the ability parameter for the subject named `varname` to be equal to `c`.
  - `varname = "+"`: constrains the ability parameter for the subject named `varname` to be positive.
  - `varname = "-"`: constrains the ability parameter for the subject named `varname` to be negative.

The model also accepts the following arguments to monitor the sampling scheme for the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 20,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.

- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed 12345.
- `theta.start`: starting values for the subject abilities (ideal points), either a scalar or a vector with length equal to the number of subjects. If a scalar, that value will be the starting value for all subjects. The default is `NA`, which sets the starting values based on an eigenvalue-eigenvector decomposition of the agreement score matrix formed from the model response matrix (`cbind(Y1, Y2, ...)`).
- `alpha.start`: starting values for the difficulty parameters α, either a scalar or a vector with length equal to the number of the items. If a scalar, the value will be the starting value for all α. The default is `NA`, which sets the starting values based on a series of probit regressions that condition on `theta.start`.
- `beta.start`: starting values for the β discrimination parameters, either a scalar or a vector with length equal to the number of the items. If a scalar, the value will be the starting value for all β. The default is `NA`, which sets the starting values based on a series of probit regressions conditioning on `theta.start`.
- `store.item`: defaults to `TRUE`, storing the posterior draws of the item parameters. (For a large number of draws or a large number observations, this may take a lot of memory.)
- `drop.constant.items`: defaults to `TRUE`, dropping items with no variation before fitting the model.

`irt1d` accepts the following additional arguments to specify prior parameters used in the model:

- `t0`: prior mean of the subject abilities (ideal points). The default is 0.
- `T0`: prior precision of the subject abilities (ideal points). The default is 0.
- `ab0`: prior mean of (α, β). It can be a scalar or a vector of length 2. If it takes a scalar value, then the prior means for both α and β will be set to that value. The default is 0.
- `AB0`: prior precision of (α, β). It can be a scalar or a 2×2 matrix. If it takes a scalar value, then the prior precision will be `diag(AB0,2)`. The prior precision is assumed to be same for all the items. The default is 0.25.

Zelig users may wish to refer to `help(MCMCirt1d)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(SupremeCourt)
   > names(SupremeCourt) <- c("Rehnquist","Stevens","OConnor","Scalia",
                              "Kennedy","Souter","Thomas","Ginsburg","Breyer")
   ```

   Fitting a one-dimensional item response theory model using `irt1d`:

   ```
   > z.out <- zelig(cbind(Rehnquist, Stevens, OConnor, Scalia, Kennedy,
                          Souter, Thomas, Ginsburg, Breyer) ~ NULL,
                    data = SupremeCourt, model = "irt1d",
                    B0.alpha = 0.2, B0.beta = 0.2, burnin = 500, mcmc = 10000,
                    thin = 20, verbose = TRUE)
   ```

   Checking for convergence before summarizing the estimates:

   ```
   > geweke.diag(z.out$coefficients)
   > heidel.diag(z.out$coefficients)
   > summary(z.out)
   ```

Model

Let Y_i be a vector of choices on J items made by subject i for $i = 1, \ldots, n$. The choice Y_{ij} is assumed to be determined by an unobserved utility Z_{ij}, which is a function of the subject i's abilities (ideal points) θ_i and item parameters α_j and β_j as follows:

$$Z_{ij} \;=\; -\alpha_j + \beta_j'\theta_i + \epsilon_{ij}.$$

- The *stochastic component* is given by

$$\begin{aligned} Y_{ij} &\sim \text{Bernoulli}(\pi_{ij}) \\ &= \pi_{ij}^{Y_{ij}}(1-\pi_{ij})^{1-Y_{ij}}, \end{aligned}$$

  where $\pi_{ij} = \Pr(Y_{ij} = 1) = E(Z_{ij})$.

  The error term in the unobserved utility equation is independently and identically distributed with

$$\epsilon_{ij} \sim \text{Normal}(0, 1).$$

- The *systematic component* is given by

$$\pi_{ij} = \Phi(-\alpha_j + \beta_j'\theta_i),$$

  where $\Phi(\cdot)$ is the cumulative density function of the standard normal distribution with mean 0 and variance 1, θ_i is the subject ability (ideal point) parameter, and α_j and β_j are the item parameters. Both subject abilities and item parameters are estimated from the model, such that the model is identified by placing constraints on the subject ability parameters.

- The *prior* for θ_i is given by

$$\theta_i \;\sim\; \text{Normal}\left(t_0, T_0^{-1}\right)$$

- The joint *prior* for α_j and β_j is given by

$$(\alpha_j, \beta_j)' \;\sim\; \text{Normal}\left(ab_0, AB_0^{-1}\right)$$

  where ab_0 is a 2-vector of prior means and AB_0 is a 2×2 prior precision matrix.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(cbind(Y1, Y2, Y3) ~ NULL, model = "irt1d", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated subject abilities(ideal points). If `store.item = TRUE`, the estimated item parameters α and β are also contained in `coefficients`.
  - `data`: the name of the input data frame.
  - `seed`: the random seed used in the model.
- Since there are no explanatory variables, the `sim()` procedure is not applicable for item response models.

Contributors

The unidimensional item-response function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*.

Sample data are adapted from

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

Ben Goodrich and Ying Lu enabled `irt1d` to work with Zelig.

12.12 `irtkd`: k-Dimensional Item Response Theory Model

Given several observed dependent variables and an unobserved explanatory variable, item response theory estimates the latent variable (ideal points). The model is estimated using the Markov Chain Monte Carlo algorithm, via a combination of Gibbs sampling and data augmentation. Use this model if you believe that the ideal points lie in k dimensions. See the unidimensional item response model (Section 12.11) for a single hypothesized latent variable.

Syntax

```
> z.out <- zelig(cbind(Y1, Y2, Y3) ~ NULL, dimensions = 1,
                 model = "irtkd", data = mydata)
```

Inputs

`irtkd` accepts the following arguments:

- `Y1`, `Y2`, and `Y3`: `Y1` contains the items for subject "Y1", `Y2` contains the items for subject "Y2", and so on.
- `dimensions`: The number of dimensions in the latent space. The default is 1.

Additional arguments

`irtkd` accepts the following additional arguments for model specification:

- `item.constraints`: a list of lists specifying possible simple equality or inequality constraints on the item parameters. A typical entry has one of the following forms:
  - `varname = list()`: by default, no constraints are imposed.
  - `varname = list(d, c)`: constrains the dth item parameter for the item named `varname` to be equal to `c`.
  - `varname = list(d, "+")`: constrains the dth item parameter for the item named `varname` to be positive;
  - `varname = list(d, "-")`: constrains the dth item parameter for the item named `varname` to be negative.

  In a k dimensional model, the first item parameter for item i is the difficulty parameter α_i, the second item parameter is the discrimination parameter on dimension 1, $(\beta_{i,1})$, the third item parameter is the discrimination parameter on dimension 2, $(\beta_{i,2}), \ldots$, and $(k+1)$th item parameter is the discrimination parameter on dimension k, $(\beta_{i,k})$. The item difficulty parameter(α) should not be constrained in general.

`irtkd` accepts the following additional arguments to monitor the sampling scheme for the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 20,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen. The default is `FALSE`.
- `zelig.data`: the input data frame if `save.data = TRUE`.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed 12345.
- `alphabeta.start`: starting values for the item parameters α and β, either a scalar or a $(k+1) \times items$ matrix. If it is a scalar, then that value will be the starting value for all the elements of `alphabeta.start`. The default is `NA` which sets the starting values for the unconstrained elements based on a series of proportional odds logistic regressions. The starting values for the inequality constrained elements are set to be either 1.0 or -1.0 depending on the nature of the constraints.
- `store.item`: defaults to `FALSE`. If `TRUE` stores the posterior draws of the item parameters. (For a large number of draws or a large number observations, this may take a lot of memory.)
- `store.ability`: defaults to `TRUE`, storing the posterior draws of the subject abilities. (For a large number of draws or a large number observations, this may take a lot of memory.)
- `drop.constant.items`: defaults to `TRUE`, dropping items with no variation before fitting the model.

`irtkd` accepts the following additional arguments to specify prior parameters used in the model:

- `b0`: prior mean of (α, β), either as a scalar or a vector of compatible length. If a scalar value, then the prior means for both α and β will be set to that value. The default is 0.
- `B0`: prior precision for (α, β), either a scalar or a $(k+1) \times items$ matrix. If a scalar value, the prior precision will be a blocked diagonal matrix with elements `diag(B0,items)`. The prior precision is assumed to be same for all the items. The default is 0.25.

Zelig users may wish to refer to `help(MCMCirtKd)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

```
> data(SupremeCourt)
> names(SupremeCourt) <- c("Rehnquist","Stevens","OConnor","Scalia",
                           "Kennedy","Souter","Thomas","Ginsburg","Breyer")
```

   Fitting a one-dimensional item response theory model using `irtkd`:

```
> z.out <- zelig(cbind(Rehnquist, Stevens, OConnor, Scalia, Kennedy, Souter,
                       Thomas, Ginsburg, Breyer) ~ NULL, dimensions = 1,
                 data = SupremeCourt, model = "irtkd", B0 = 0.25,
                 burnin = 5000, mcmc = 50000,
                 thin = 10, verbose = TRUE)
```

   Checking for convergence before summarizing the estimates:

```
> geweke.diag(z.out$coefficients)
> heidel.diag(z.out$coefficients)
> raftery.diag(z.out$coefficients)
> summary(z.out)
```

Model

Let Y_i be a vector of choices on J items made by subject i for $i = 1, \ldots, n$. The choice Y_{ij} is assumed to be determined by unobserved utility Z_{ij}, which is a function of subject abilities (ideal points) θ_i and item parameters α_j and β_j,

$$Z_{ij} \quad = \quad -\alpha_j + \beta_j' \theta_i + \epsilon_{ij}.$$

In the k-dimensional item response theory model, each subject's ability is represented by a k-vector, θ_i. Each item has a difficulty parameter α_j and a k-dimensional discrimination parameter β_j. In one-dimensional item response theory model, $k = 1$.

- The *stochastic component* is given by

  $$\begin{aligned} Y_{ij} &\sim \text{Bernoulli}(\pi_{ij}) \\ &= \pi_{ij}^{Y_{ij}} (1 - \pi_{ij})^{1 - Y_{ij}}, \end{aligned}$$

  where $\pi_{ij} = \Pr(Y_{ij} = 1) = E(Z_{ij})$.

  The error term in the unobserved utility equation has a standard normal distribution,

  $$\epsilon_{ij} \sim \text{Normal}(0, 1).$$

- The *systematic component* is given by

  $$\pi_{ij} \quad = \quad \Phi(-\alpha_j + \beta_j \theta_i),$$

  where $\Phi(\cdot)$ is the cumulative density function of the standard normal distribution with mean 0 and variance 1, while θ_i contains the k-dimensional subject abilities(ideal points), and α_j and β_j are the item parameters. Both subject abilities and item parameters need to estimated from the model. The model is identified by placing constraints on the item parameters.

- The *prior* for θ_i is given by

  $$\theta_i \quad \sim \quad \text{Normal}_k(0, I_k)$$

- The joint *prior* for α_j and β_j is given by

  $$(\alpha_j, \beta_j)' \quad \sim \quad \text{Normal}_{k+1}\left(b_{0_j}, B_{0_j}^{-1}\right)$$

  where b_{0_j} is a $(k+1)$-vector of prior mean and B_{0_j} is a $(k+1) \times (k+1)$ prior precision matrix which is assumed to be diagonal.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(cbind(Y1, Y2, Y3) ~ NULL, model = "irtkd", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated subject abilities(ideal points). If `store.item = TRUE`, the estimated item parameters α and β are also contained in `coefficients`.
  - `data`: the name of the input data frame.
  - `seed`: the random seed used in the model.
- Since there are no explanatory variables, the `sim()` procedure is not applicable for item response models.

Contributors

The k dimensional item response function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*.

Sample data are adapted from

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

Ben Goodrich and Ying Lu enabled `irtkd` to work with Zelig.

12.13 `logit`: Logistic Regression for Dichotomous Dependent Variables

Logistic regression specifies a dichotomous dependent variable as a function of a set of explanatory variables. For a Bayesian implementation, see Section 12.14.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "logit", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out, x1 = NULL)
```

Additional Inputs

In addition to the standard inputs, `zelig()` takes the following additional options for logistic regression:

- `robust`: defaults to `FALSE`. If `TRUE` is selected, `zelig()` computes robust standard errors via the `sandwich` package (see Zeileis (2004)). The default type of robust standard error is heteroskedastic and autocorrelation consistent (HAC), and assumes that observations are ordered by time index.

  In addition, `robust` may be a list with the following options:

  - `method`: Choose from
    - `"vcovHAC"`: (default if `robust = TRUE`) HAC standard errors.
    - `"kernHAC"`: HAC standard errors using the weights given in Andrews (1991).
    - `"weave"`: HAC standard errors using the weights given in Lumley and Heagerty (1999).
  - `order.by`: defaults to `NULL` (the observations are chronologically ordered as in the original data). Optionally, you may specify a vector of weights (either as `order.by = z`, where `z` exists outside the data frame; or as `order.by = ~z`, where `z` is a variable in the data frame) The observations are chronologically ordered by the size of `z`.
  - `...`: additional options passed to the functions specified in `method`. See the `sandwich` library and Zeileis (2004) for more options.

Examples

1. Basic Example

   Attaching the sample turnout dataset:

   ```
   > data(turnout)
   ```

Estimating parameter values for the logistic regression:

```
> z.out1 <- zelig(vote ~ race + educate,  model = "logit", data = turnout)
> summary(z.out1)
```

Setting values for the explanatory variables to their default values:

```
> x.out1 <- setx(z.out1)
```

Simulating quantities of interest from the posterior distribution.

```
> s.out1 <- sim(z.out1, x = x.out)
> summary(s.out1)
> plot(s.out1)
```

2. Simulating First Differences

   Estimating the risk difference (and risk ratio) between low education (25th percentile) and high education (75th percentile) while all the other variables held at their default values.

```
> x.high <- setx(z.out, educate = quantile(turnout$educate, prob = 0.75))
> x.low <- setx(z.out, educate = quantile(turnout$educate, prob = 0.25))
```

```
> s.out2 <- sim(z.out1, x = x.high, x1 = x.low)
> summary(s.out2)
> plot(s.out2)
```

3. Presenting Results: An ROC Plot

   One can use an ROC plot to evaluate the fit of alternative model specifications. (Use `demo(roc)` to view this example, or see King and Zeng (2002).)

```
> z.out1 <- zelig(vote ~ race + educate + age, model = "logit",
                  data = turnout)
> z.out2 <- zelig(vote ~ race + educate, model = "logit", data = turnout)
> rocplot(z.out1, z.out2)
```

Model

Let Y_i be the binary dependent variable for observation i which takes the value of either 0 or 1.

- The *stochastic component* is given by

$$\begin{aligned} Y_i &\sim \text{Bernoulli}(y_i \mid \pi_i) \\ &= \pi_i^{y_i}(1-\pi_i)^{1-y_i} \end{aligned}$$

  where $\pi_i = \Pr(Y_i = 1)$.

- The *systematic component* is given by:

$$\pi_i = \frac{1}{1+\exp(-x_i\beta)}.$$

  where x_i is the vector of k explanatory variables for observation i and β is the vector of coefficients.

Quantities of Interest

- The expected values (`qi$ev`) for the logit model are simulations of the predicted probability of a success:

$$E(Y) = \pi_i = \frac{1}{1+\exp(-x_i\beta)},$$

  given draws of β from its sampling distribution.

- The predicted values (`qi$pr`) are draws from the Binomial distribution with mean equal to the simulated expected value π_i.

- The first difference (`qi$fd`) for the logit model is defined as

$$\text{FD} = \Pr(Y = 1 \mid x_1) - \Pr(Y = 1 \mid x).$$

- The risk ratio (`qi$rr`) is defined as

$$\text{RR} = \Pr(Y = 1 \mid x_1) \;/\; \Pr(Y = 1 \mid x).$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - E[Y_i(t_i = 0)] \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "logit", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `residuals`: the working residuals in the final iteration of the IWLS fit.
  - `fitted.values`: the vector of fitted values for the systemic component, π_i.
  - `linear.predictors`: the vector of $x_i\beta$
  - `aic`: Akaike's Information Criterion (minus twice the maximized log-likelihood plus twice the number of coefficients).
  - `df.residual`: the residual degrees of freedom.
  - `df.null`: the residual degrees of freedom for the null model.
  - `data`: the name of the input data frame.
- From `summary(z.out)`, you may extract:
  - `coefficients`: the parameter estimates with their associated standard errors, p-values, and t-statistics.
  - `cov.scaled`: a $k \times k$ matrix of scaled covariances.
  - `cov.unscaled`: a $k \times k$ matrix of unscaled covariances.
- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:

- `qi$ev`: the simulated expected probabilities for the specified values of `x`.
- `qi$pr`: the simulated predicted values for the specified values of `x`.
- `qi$fd`: the simulated first difference in the expected probabilities for the values specified in `x` and `x1`.
- `qi$rr`: the simulated risk ratio for the expected probabilities simulated from `x` and `x1`.
- `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
- `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The logit function is part of the base package by William N. Venables and Brian D. Ripley. Please cite this model as:

> Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

Advanced users may wish to refer to `help(glm)` and `help(family)`, as well as

> McCullagh, P. and Nelder, J. A. (1989), *Generalized Linear Models*, no. 37 in Monograph on Statistics and Applied Probability, Chapman & Hall, 2nd ed.

Robust standard errors are implemented via the sandwich package by Achim Zeileis. Please cite as

> Zeileis, A. (2004), "Econometric Computing with HC and HAC Covariance Matrix Estimators," *Journal of Statistical Software*, 11, 1–17

Sample data are a selection of 2,000 observations from

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

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.14 `logit.bayes`: Bayesian Logistic Regression

Logistic regression specifies a dichotomous dependent variable as a function of a set of explanatory variables using a random walk Metropolis algorithm. For a maximum likelihood implementation, see Section 12.13.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "logit.bayes", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

Use the following arguments to monitor the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 10,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `tune`: Metropolis tuning parameter, either a positive scalar or a vector of length k, where k is the number of coefficients. The tuning parameter should be set such that the acceptance rate of the Metropolis algorithm is satisfactory (typically between 0.20 and 0.5) before using the posterior density for inference. The default value is 1.1.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed of 12345.
- `beta.start`: starting values for the Markov chain, either a scalar or vector with length equal to the number of estimated coefficients. The default is `NA`, such that the maximum likelihood estimates are used as the starting values.

Use the following parameters to specify the model's priors:

- `b0`: prior mean for the coefficients, either a numeric vector or a scalar. If a scalar value, that value will be the prior mean for all the coefficients. The default is 0.
- `B0`: prior precision parameter for the coefficients, either a square matrix (with the dimensions equal to the number of coefficients) or a scalar. If a scalar value, that value times an identity matrix will be the prior precision parameter. The default is 0, which leads to an improper prior.

Zelig users may wish to refer to `help(logit.bayes)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(turnout)
   ```

   Estimating the logistic regression using `logit.bayes`:

   ```
   > z.out < - zelig(vote ~ race + educate, model = "logit.bayes",
                     data = turnout, verbose = TRUE)
   ```

   Convergence diagnostics before summarizing the estimates:

   ```
   > geweke.diag(z.out$coefficients)
   > heidel.diag(z.out$coefficients)
   > raftery.diag(z.out$coefficients)
   > summary(z.out)
   ```

Setting values for the explanatory variables to their sample averages:

```
> x.out <- setx(z.out)
```

Simulating quantities of interest from the posterior distribution given `x.out`.

```
> s.out1 <- sim(z.out, x = x.out)
> summary(s.out1)
```

2. Simulating First Differences
   Estimating the first difference (and risk ratio) in individual's probability of voting when education is set to be low (25th percentile) versus high (75th percentile) while all the other variables held at their default values.

```
> x.high <- setx(z.out, educate = quantile(turnout$educate, prob = 0.75))
> x.low <- setx(z.out, educate = quantile(turnout$educate, prob = 0.25))
> s.out2 <- sim(z.out, x = x.high, x1 = x.low)
> summary(s.out2)
```

Model

Let Y_i be the binary dependent variable for observation i which takes the value of either 0 or 1.

- The *stochastic component* is given by

$$\begin{aligned} Y_i &\sim \text{Bernoulli}(\pi_\text{i}) \\ &= \pi_i^{Y_i}(1-\pi_i)^{1-Y_i}, \end{aligned}$$

  where $\pi_i = \Pr(Y_i = 1)$.

- The *systematic component* is given by

$$\pi_i = \frac{1}{1+\exp(-x_i\beta)},$$

  where x_i is the vector of k explanatory variables for observation i and β is the vector of coefficients.

- The *prior* for β is given by

$$\beta \sim \text{Normal}_k\left(b_0, B_0^{-1}\right)$$

  where b_0 is the vector of means for the k explanatory variables and B_0 is the $k \times k$ precision matrix (the inverse of a variance-covariance matrix).

Quantities of Interest

- The expected values (`qi$ev`) for the logit model are simulations of the predicted probability of a success:

$$E(Y) = \pi_i = \frac{1}{1 + \exp(-x_i\beta)},$$

  given the posterior draws of β from the MCMC iterations.

- The predicted values (`qi$pr`) are draws from the Bernoulli distribution with mean equal to the simulated expected value π_i.

- The first difference (`qi$fd`) for the logit model is defined as

$$\text{FD} = \Pr(Y = 1 \mid X_1) - \Pr(Y = 1 \mid X).$$

- The risk ratio (`qi$rr`)is defined as

$$\text{RR} = \Pr(Y = 1 \mid X_1) \ / \ \Pr(Y = 1 \mid X).$$

- In conditional prediction models, the average expected treatment effect (`qi$att.ev`) for the treatment group is

$$\frac{1}{\sum t_i} \sum_{i:t_i=1} [Y_i(t_i = 1) - E[Y_i(t_i = 0)]],$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

- In conditional prediction models, the average predicted treatment effect (`qi$att.pr`) for the treatment group is

$$\frac{1}{\sum t_i} \sum_{i:t_i=1} [Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)}],$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run

```
z.out <- zelig(y ~ x, model = "logit.bayes", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated parameters.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
  - `seed`: the random seed used in the model.
- From the `sim()` output object `s.out`:
  - `qi$ev`: the simulated expected values(probabilities) for the specified values of `x`.
  - `qi$pr`: the simulated predicted values for the specified values of `x`.
  - `qi$fd`: the simulated first difference in the expected values for the values specified in `x` and `x1`.
  - `qi$rr`: the simulated risk ratio for the expected values simulated from `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

Bayesian logistic regression function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*

Ben Goodrich and Ying Lu enabled `logit.bayes` to work with Zelig.

12.15 `lognorm`: Log-Normal Regression for Duration Dependent Variables

The log-normal model describes an event's duration, the dependent variable, as a function of a set of explanatory variables. The log-normal model may take time censored dependent variables, and allows the hazard rate to increase and decrease.

Syntax

```
> z.out <- zelig(Surv(Y, C) ~ X, model = "lognorm", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Log-normal models require that the dependent variable be in the form `Surv(Y, C)`, where `Y` and `C` are vectors of length n. For each observation i in $1, \ldots, n$, the value y_i is the duration (lifetime, for example) of each subject, and the associated c_i is a binary variable such that $c_i = 1$ if the duration is not censored (*e.g.*, the subject dies during the study) or $c_i = 0$ if the duration is censored (*e.g.*, the subject is still alive at the end of the study). If c_i is omitted, all Y are assumed to be completed; that is, time defaults to 1 for all observations.

Input Values

In addition to the standard inputs, `zelig()` takes the following additional options for log-normal regression:

- `robust`: defaults to `FALSE`. If `TRUE`, `zelig()` computes robust standard errors based on sandwich estimators (see Huber (1981) and White (1980)) based on the options in `cluster`.
- `cluster`: if `robust = TRUE`, you may select a variable to define groups of correlated observations. Let `x3` be a variable that consists of either discrete numeric values, character strings, or factors that define strata. Then

  ```
  > z.out <- zelig(y ~ x1 + x2, robust = TRUE, cluster = "x3",
                   model = "exp", data = mydata)
  ```

  means that the observations can be correlated within the strata defined by the variable `x3`, and that robust standard errors should be calculated according to those clusters. If `robust = TRUE` but `cluster` is not specified, `zelig()` assumes that each observation falls into its own cluster.

Example

Attach the sample data:

```
> data(coalition)
```

Estimate the model:

```
> z.out <- zelig(Surv(duration, ciep12) ~ fract + numst2, model = "lognorm",
                 data = coalition)
```

View the regression output:

```
> summary(z.out)
```

Set the baseline values (with the ruling coalition in the minority) and the alternative values (with the ruling coalition in the majority) for X:

```
> x.low <- setx(z.out, numst2 = 0)
> x.high <- setx(z.out, numst2 = 1)
```

Simulate expected values (`qi$ev`) and first differences (`qi$fd`):

```
> s.out <- sim(z.out, x = x.low, x1 = x.high)
> summary(s.out)
> plot(s.out)
```

Model

Let Y_i^* be the survival time for observation i with the density function $f(y)$ and the corresponding distribution function $F(t) = \int_0^t f(y)dy$. This variable might be censored for some observations at a fixed time y_c such that the fully observed dependent variable, Y_i, is defined as

$$Y_i = \begin{cases} Y_i^* & \text{if } Y_i^* \leq y_c \\ y_c & \text{if } Y_i^* > y_c \end{cases}$$

- The *stochastic component* is described by the distribution of the partially observed variable, Y^*. For the lognormal model, there are two equivalent representations:

  $$Y_i^* \sim \text{LogNormal}(\mu_i, \sigma^2) \quad \text{or} \quad \log(Y_i^*) \sim \text{Normal}(\mu_i, \sigma^2)$$

  where the parameters μ_i and σ^2 are the mean and variance of the Normal distribution. (Note that the output from `zelig()` parameterizes `scale`$= \sigma$.)

  In addition, survival models like the lognormal have three additional properties. The hazard function $h(t)$ measures the probability of not surviving past time t given survival up to t. In general, the hazard function is equal to $f(t)/S(t)$ where the survival function $S(t) = 1 - \int_0^t f(s)ds$ represents the fraction still surviving at time t. The cumulative

hazard function $H(t)$ describes the probability of dying before time t. In general, $H(t) = \int_0^t h(s)ds = -\log S(t)$. In the case of the lognormal model,

$$\begin{aligned} h(t) &= \frac{1}{\sqrt{2\pi}\,\sigma t\, S(t)} \exp\left\{-\frac{1}{2\sigma^2}(\log \lambda t)^2\right\} \\ S(t) &= 1 - \Phi\left(\frac{1}{\sigma}\log \lambda t\right) \\ H(t) &= -\log\left\{1 - \Phi\left(\frac{1}{\sigma}\log \lambda t\right)\right\} \end{aligned}$$

where $\Phi(\cdot)$ is the cumulative density function for the Normal distribution.

- The *systematic component* is described as:

$$\mu_i = x_i\beta.$$

Quantities of Interest

- The expected values (`qi$ev`) for the lognormal model are simulations of the expected duration:

$$E(Y) = \exp\left(\mu_i + \frac{1}{2}\sigma^2\right),$$

given draws of β and σ from their sampling distributions.

- The predicted value is a draw from the log-normal distribution given simulations of the parameters (λ_i, σ).

- The first difference (`qi$fd`) is

$$\text{FD} = E(Y \mid x_1) - E(Y \mid x).$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^n t_i} \sum_{i:t_i=1}^{n} \{Y_i(t_i = 1) - E[Y_i(t_i = 0)]\},$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. When $Y_i(t_i = 1)$ is censored rather than observed, we replace it with a simulation from the model given available knowledge of the censoring process. Variation in the simulations is due to two factors: uncertainty in the imputation process for censored y_i^* and uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \{Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)}\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. When $Y_i(t_i = 1)$ is censored rather than observed, we replace it with a simulation from the model given available knowledge of the censoring process. Variation in the simulations are due to two factors: uncertainty in the imputation process for censored y_i^* and uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(Surv(Y, C) ~ X, model = "lognorm", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `icoef`: parameter estimates for the intercept and σ.
  - `var`: Variance-covariance matrix.
  - `loglik`: Vector containing the log-likelihood for the model and intercept only (respectively).
  - `linear.predictors`: the vector of $x_i\beta$.
  - `df.residual`: the residual degrees of freedom.
  - `df.null`: the residual degrees of freedom for the null model.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
- Most of this may be conveniently summarized using `summary(z.out)`. From `summary(z.out)`, you may additionally extract:
  - `table`: the parameter estimates with their associated standard errors, p-values, and t-statistics.

- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation × `x`-observation (for more than one `x`-observation). Available quantities are:
  - `qi$ev`: the simulated expected values for the specified values of `x`.
  - `qi$pr`: the simulated predicted values drawn from the distribution defined by (λ_i, σ).
  - `qi$fd`: the simulated first differences between the simulated expected values for `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The exponential function is part of the survival library by by Terry Therneau, ported to R by Thomas Lumley. Advanced users may wish to refer to `help(survfit)` in the survival library, and

Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

Sample data are from

King, G., Alt, J., Burns, N., and Laver, M. (1990), "A Unified Model of Cabinet Dissolution in Parliamentary Democracies," *American Journal of Political Science*, 34, 846–871, http://gking.harvard.edu/files/abs/coal-abs.shtml.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.16 `ls`: Least Squares Regression for Continuous Dependent Variables

Use least squares regression analysis to estimate the best linear predictor for the specified dependent variables.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "ls", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

In addition to the standard inputs, `zelig()` takes the following additional options for least squares regression:

- `robust`: defaults to `FALSE`. If `TRUE` is selected, `zelig()` computes robust standard errors based on sandwich estimators (see Zeileis (2004), Huber (1981), and White (1980)). The default type of robust standard error is heteroskedastic consistent (HC), *not* heteroskedastic and autocorrelation consistent (HAC).

  In addition, `robust` may be a list with the following options:

  - `method`: choose from
    - `"vcovHC"`: (the default if `robust = TRUE`), HC standard errors.
    - `"vcovHAC"`: HAC standard errors without weights.
    - `"kernHAC"`: HAC standard errors using the weights given in Andrews (1991).
    - `"weave"`: HAC standard errors using the weights given in Lumley and Heagerty (1999).
  - `order.by`: only applies to the HAC methods above. Defaults to `NULL` (the observations are chronologically ordered as in the original data). Optionally, you may specify a time index (either as `order.by = z`, where `z` exists outside the data frame; or as `order.by = ~z`, where `z` is a variable in the data frame). The observations are chronologically ordered by the size of `z`.
  - `...`: additional options passed to the functions specified in `method`. See the `sandwich` library and Zeileis (2004) for more options.

Examples

1. Basic Example with First Differences

   Attach sample data:

```
> data(macro)
```

Estimate model:

```
> z.out1 <- zelig(unem ~ gdp + capmob + trade, model = "ls", data = macro)
```

Summarize regression coefficients:

```
> summary(z.out1)
```

Set explanatory variables to their default (mean/mode) values, with high (80th percentile) and low (20th percentile) values for the trade variable:

```
> x.high <- setx(z.out1, trade = quantile(trade, 0.8))
> x.low <- setx(z.out1, trade = quantile(trade, 0.2))
```

Generate first differences for the effect of high versus low trade on GDP:

```
> s.out1 <- sim(z.out1, x = x.high, x1 = x.low)
> summary(s.out1)
> plot(s.out1)
```

2. Using Dummy Variables

   Estimate a model with fixed effects for each country (see Section 2 for help with dummy variables). Note that you do not need to create dummy variables, as the program will automatically parse the unique values in the selected variable into discrete levels.

```
> z.out2 <- zelig(unem ~ gdp + trade + capmob + as.factor(country),
                  model = "ls", data = macro)
```

   Set values for the explanatory variables, using the default mean/mode values, with country set to the United States and Japan, respectively:

```
> x.US <- setx(z.out2, country = "United States")
> x.Japan <- setx(z.out2, country = "Japan")
```

   Simulate quantities of interest:

```
> s.out2 <- sim(z.out2, x = x.US, x1 = x.Japan)
> plot(s.out2)
```

Model

- The *stochastic component* is described by a density with mean μ_i and the common variance σ^2

$$Y_i \sim f(y_i \mid \mu_i, \sigma^2).$$

- The *systematic component* models the conditional mean as

$$\mu_i = x_i\beta$$

where x_i is the vector of covariates, and β is the vector of coefficients.

The least squares estimator is the best linear predictor of a dependent variable given x_i, and minimizes the sum of squared residuals, $\sum_{i=1}^{n}(Y_i - x_i\beta)^2$.

Quantities of Interest

- The expected value (`qi$ev`) is the mean of simulations from the stochastic component,

$$E(Y) = x_i\beta,$$

given a draw of β from its sampling distribution.

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{Y_i(t_i = 1) - E[Y_i(t_i = 0)]\right\},$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "ls", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.

- `residuals`: the working residuals in the final iteration of the IWLS fit.
- `fitted.values`: fitted values.
- `df.residual`: the residual degrees of freedom.
- `zelig.data`: the input data frame if `save.data = TRUE`.

- From `summary(z.out)`, you may extract:
  - `coefficients`: the parameter estimates with their associated standard errors, p-values, and t-statistics.

$$\hat{\beta} = \left(\sum_{i=1}^{n} x_i' x_i \right)^{-1} \sum x_i y_i$$

  - `sigma`: the square root of the estimate variance of the random error e:

$$\hat{\sigma} = \frac{\sum (Y_i - x_i \hat{\beta})^2}{n - k}$$

  - `r.squared`: the fraction of the variance explained by the model.

$$R^2 = 1 - \frac{\sum (Y_i - x_i \hat{\beta})^2}{\sum (y_i - \bar{y})^2}$$

  - `adj.r.squared`: the above R^2 statistic, penalizing for an increased number of explanatory variables.
  - `cov.unscaled`: a $k \times k$ matrix of unscaled covariances.

- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:
  - `qi$ev`: the simulated expected values for the specified values of `x`.
  - `qi$fd`: the simulated first differences (or differences in expected values) for the specified values of `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.

Contributors

The least squares regression is part of the stats package by William N. Venables and Brian D. Ripley. Please cite the model as:

> Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

In addition, advanced users may wish to refer to `help(lm)` and `help(lm.fit)`.

Robust standard errors are implemented via the sandwich package by Achim Zeileis. Please cite as

> Zeileis, A. (2004), "Econometric Computing with HC and HAC Covariance Matrix Estimators," *Journal of Statistical Software*, 11, 1–17

Sample data are from

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

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.17 `mlogit`: Multinomial Logistic Regression for Dependent Variables with Unordered Categorical Values

Use the multinomial logit distribution to model unordered categorical variables. The dependent variable may be in the format of either character strings or integer values. See Section 12.18 for a Bayesian version of this model.

Syntax

```
> z.out <- zelig(as.factor(Y) ~ X1 + X2, model = "mlogit", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Input Values

If the user wishes to use the same formula across all levels, then `formula <- as.factor(Y) ~ X1 + X2` may be used. If the user wants to use different formula for each level then the following syntax should be used:

```
formulae <- list(list(id(Y, "apples")~ X1,
                      id(Y, "bananas")~ X1 + X2)
```

where Y above is supposed to be a factor variable with levels apples,bananas,oranges. By default, oranges is the last level and omitted. (You cannot specify a different base level at this time.) For J equations, there must be $J + 1$ levels.

Examples

1. The same formula for each level

   Load the sample data:

   ```
   > data(mexico)
   ```

   Estimate the empirical model:

   ```
   > z.out1 <- zelig(as.factor(vote88) ~ pristr + othcok + othsocok,
                     model = "mlogit", data = mexico)
   ```

   Set the explanatory variables to their default values, with `pristr` (for the strength of the PRI) equal to 1 (weak) in the baseline values, and equal to 3 (strong) in the alternative values:

```
> x.weak <- setx(z.out1, pristr = 1)
> x.strong <- setx(z.out1, pristr = 3)
```

Generate simulated predicted probabilities `qi$ev` and differences in the predicted probabilities `qi$fd`:

```
> s.out1 <- sim(z.out1, x = x.strong, x1 = x.weak)
> summary(s.out1)
```

Generate simulated predicted probabilities `qi$ev` for the alternative values:

```
> ev.weak <- s.out1$qi$ev + s.out1$qi$fd
```

Plot the differences in the predicted probabilities.

```
> library(vcd)
> ternaryplot(s.out1$qi$ev, pch = ".", col = "blue",
              main = "1988 Mexican Presidential Election")
> ternarypoints(ev.weak, pch = ".", col = "red")
```

2. Different formula for each level

   Estimate the empirical model:

```
> z.out2 <- zelig(list(id(vote88, "1") ~ pristr + othcok,
                       id(vote88, "2") ~ othsocok),
                  model = "mlogit", data = mexico)
```

   Set the explanatory variables to their default values, with `pristr` (for the strength of the PRI) equal to 1 (weak) in the baseline values, and equal to 3 (strong) in the alternative values:

```
> x.weak <- setx(z.out2, pristr = 1)
> x.strong <- setx(z.out2, pristr = 3)
```

   Generate simulated predicted probabilities `qi$ev` and differences in the predicted probabilities `qi$fd`:

```
> s.out1 <- sim(z.out2, x = x.strong, x1 = x.weak)
> summary(s.out1)
```

   Generate simulated predicted probabilities `qi$ev` for the alternative values:

```
> ev.weak <- s.out2$qi$ev + s.out2$qi$fd
```

Plot the differences in the predicted probabilities.

```
> library(vcd)
> ternaryplot(s.out2$qi$ev, pch = ".", col = "blue",
              main = "1988 Mexican Presidential Election")
> ternarypoints(ev.weak, pch = ".", col = "red")
```

Model

Let Y_i be the unordered categorical dependent variable that takes one of the values from 1 to J, where J is the total number of categories.

- The stochastic component is given by

$$Y_i \sim \text{Multinomial}(y_i \mid \pi_{ij}),$$

  where $\pi_{ij} = \Pr(Y_i = j)$ for $j = 1, \ldots, J$.

- The systemic component is given by:

$$\pi_{ij} = \frac{\exp(x_i\beta_j)}{\sum_{k=1}^{J}\exp(x_i\beta_k)},$$

  where x_i is the vector of explanatory variables for observation i, and β_j is the vector of coefficients for category j.

Quantities of Interest

- The expected value (`qi$ev`) is the predicted probability for each category:

$$E(Y) = \pi_{ij} = \frac{\exp(x_i\beta_j)}{\sum_{k=1}^{J}\exp(x_i\beta_k)}.$$

- The predicted value (`qi$pr`) is a draw from the multinomial distribution defined by the predicted probabilities.

- The first difference in predicted probabilities (`qi$fd`), for each category is given by:

$$\text{FD}_j = \Pr(Y = j \mid x_1) - \Pr(Y = j \mid x) \quad \text{for} \quad j = 1, \ldots, J.$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{n_j}\sum_{i:t_i=1}^{n_j}\left\{Y_i(t_i = 1) - E[Y_i(t_i = 0)]\right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of treated observations in category j.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{n_j}\sum_{i:t_i=1}^{n_j}\left\{Y_i(t_i=1)-\widehat{Y_i(t_i=0)}\right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of treated observations in category j.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "mlogit", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: the named vector of coefficients.
  - `fitted.values`: an $n \times J$ matrix of the in-sample fitted values.
  - `predictors`: an $n \times (J-1)$ matrix of the linear predictors $x_i\beta_j$.
  - `residuals`: an $n \times (J-1)$ matrix of the residuals.
  - `df.residual`: the residual degrees of freedom.
  - `df.total`: the total degrees of freedom.
  - `rss`: the residual sum of squares.
  - `y`: an $n \times J$ matrix of the dependent variables.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
- From `summary(z.out)`, you may extract:
  - `coef3`: a table of the coefficients with their associated standard errors and t-statistics.
  - `cov.unscaled`: the variance-covariance matrix.
  - `pearson.resid`: an $n \times (m-1)$ matrix of the Pearson residuals.
- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as arrays. Available quantities are:
  - `qi$ev`: the simulated expected probabilities for the specified values of `x`, indexed by simulation $\times$ quantity $\times$ `x`-observation (for more than one `x`-observation).

- `qi$pr`: the simulated predicted values drawn from the distribution defined by the expected probabilities, indexed by simulation × `x`-observation.
- `qi$fd`: the simulated first difference in the predicted probabilities for the values specified in `x` and `x1`, indexed by simulation × quantity × `x`-observation (for more than one `x`-observation).
- `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models,
- `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Further Information

The multinomial logit model is part of the VGAM package by Thomas Yee. Please cite the model as:

> Yee, T. W. and Hastie, T. J. (2003), "Reduced-rank vector generalized linear models," *Statistical Modelling*, 3, 15–41.

In addition, advanced users may wish to refer to `help(vglm)` in the VGAM library. Additional documentation is available at http://www.stat.auckland.ac.nz/~ yee.

Sample data are from

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

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.18 mlogit.bayes: Bayesian Multinomial Logistic Regression

Use Bayesian multinomial logistic regression to model unordered categorical variables. The dependent variable may be in the format of either character strings or integer values. The model is estimated via a random walk Metropolis algorithm or a slice sampler. See Section 12.17 for the maximum-likelihood estimation of this model.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "mlogit.bayes", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

`zelig()` accepts the following arguments for `mlogit.bayes`:

- `baseline`: either a character string or numeric value (equal to one of the observed values in the dependent variable) specifying a baseline category. The default value is `NA` which sets the baseline to the first alphabetical or numerical unique value of the dependent variable.

The model accepts the following additional arguments to monitor the Markov chains:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 10,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `mcmc.method`: either `"MH"` or `"slice"`, specifying whether to use Metropolis Algorithm or slice sampler. The default value is `"MH"`.
- `tune`: tuning parameter for the Metropolis-Hasting step, either a scalar or a numeric vector (for k coefficients, enter a k vector). The tuning parameter should be set such that the acceptance rate is satisfactory (between 0.2 and 0.5). The default value is 1.1.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed of 12345.

- `beta.start`: starting values for the Markov chain, either a scalar or a vector (for k coefficients, enter a k vector). The default is `NA` where the maximum likelihood estimates are used as the starting values.

Use the following arguments to specify the priors for the model:

- `b0`: prior mean for the coefficients, either a scalar or vector. If a scalar, that value will be the prior mean for all the coefficients. The default is 0.

- `B0`: prior precision parameter for the coefficients, either a square matrix with the dimensions equal to the number of coefficients or a scalar. If a scalar, that value times an identity matrix will be the prior precision parameter. The default is 0 which leads to an improper prior.

Zelig users may wish to refer to `help(MCMCmnl)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.

- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.

- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(mexico)
   ```

   Estimating multinomial logistics regression using `mlogit.bayes`:

   ```
   > z.out <- zelig(vote88 ~ pristr + othcok + othsocok, model = "mlogit.bayes",
                 data = mexico)
   ```

   Checking for convergence before summarizing the estimates:

   ```
   > heidel.diag(z.out$coefficients)
   > raftery.diag(z.out$coefficients)
   > summary(z.out)
   ```

   Setting values for the explanatory variables to their sample averages:

   ```
   > x.out <- setx(z.out)
   ```

   Simulating quantities of interest from the posterior distribution given `x.out`.

   ```
   > s.out1 <- sim(z.out, x = x.out)
   > summary(s.out1)
   ```

2. Simulating First Differences
   Estimating the first difference (and risk ratio) in the probabilities of voting different candidates when `pristr` (the strength of the PRI) is set to be weak (equal to 1) versus strong (equal to 3) while all the other variables held at their default values.

   ```
   > x.weak <- setx(z.out, pristr = 1)
   > x.strong <- setx(z.out, pristr = 3)
   > s.out2 <- sim(z.out, x = x.strong, x1 = x.weak)
   > summary(s.out2)
   ```

Model

Let Y_i be the (unordered) categorical dependent variable for observation i which takes an integer values $j = 1, \ldots, J$.

- The *stochastic component* is given by:

$$Y_i \sim \text{Multinomial}(Y_i \mid \pi_{ij}).$$

  where $\pi_{ij} = \Pr(Y_i = j)$ for $j = 1, \ldots, J$.

- The *systematic component* is given by

$$\pi_{ij} = \frac{\exp(x_i \beta_j)}{\sum_{k=1}^{J} \exp(x_i \beta_k)}, \text{ for } j = 1, \ldots, J-1,$$

  where x_i is the vector of k explanatory variables for observation i and β_j is the vector of coefficient for category j. Category J is assumed to be the baseline category.

- The *prior* for β is given by

$$\beta_j \sim \text{Normal}_k \left(b_0, B_0^{-1}\right) \text{ for } j = 1, \ldots, J-1,$$

  where b_0 is the vector of means for the k explanatory variables and B_0 is the $k \times k$ precision matrix (the inverse of a variance-covariance matrix).

Quantities of Interest

- The expected values (`qi$ev`) for the multinomial logistics regression model are the predicted probability of belonging to each category:

$$\Pr(Y_i = j) = \pi_{ij} = \frac{\exp(x_i \beta_j)}{\sum_{k=1}^{J} \exp(x_J \beta_k)}, \quad \text{for } j = 1, \ldots, J-1,$$

  and

$$\Pr(Y_i = J) = 1 - \sum_{j=1}^{J-1} \Pr(Y_i = j)$$

  given the posterior draws of β_j for all categories from the MCMC iterations.

- The predicted values (`qi$pr`) are the draws of Y_i from a multinomial distribution whose parameters are the expected values(`qi$ev`) computed based on the posterior draws of β from the MCMC iterations.

- The first difference (`qi$fd`) in category j for the multinomial logistic model is defined as

$$\text{FD}_j = \Pr(Y_i = j \mid X_1) - \Pr(Y_i = j \mid X).$$

- The risk ratio (`qi$rr`) in category j is defined as

$$\text{RR}_j = \Pr(Y_i = j \mid X_1) \;/\; \Pr(Y_i = j \mid X).$$

- In conditional prediction models, the average expected treatment effect (`qi$att.ev`) for the treatment group in category j is

$$\frac{1}{n_j} \sum_{i:t_i=1}^{n_j} [Y_i(t_i = 1) - E[Y_i(t_i = 0)]],$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of treated observations in category j.

- In conditional prediction models, the average predicted treatment effect (`qi$att.pr`) for the treatment group in category j is

$$\frac{1}{n_j} \sum_{i:t_i=1}^{n_j} [Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)}],$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of treated observations in category j.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(y ~ x, model = "mlogit.bayes", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated coefficients β for each category except the baseline category.
  - `zelig.data`: the input data frame if `save.data = TRUE`.

  - `seed`: the random seed used in the model.

- From the `sim()` output object `s.out`:

  - `qi$ev`: the simulated expected values(probabilities) of each of the J categories given the specified values of `x`.
  - `qi$pr`: the simulated predicted values drawn from the multinomial distribution defined by the expected values(`qi$ev`) given the specified values of `x`.
  - `qi$fd`: the simulated first difference in the expected values of each of the J categories for the values specified in `x` and `x1`.
  - `qi$rr`: the simulated risk ratio for the expected values of each of the J categories simulated from `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

Bayesian multinomial logistic regression function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*

Ben Goodrich and Ying Lu enabled `mlogit.bayes` to work with Zelig.

12.19 negbin: Negative Binomial Regression for Event Count Dependent Variables

Use the negative binomial regression if you have a count of events for each observation of your dependent variable. The negative binomial model is frequently used to estimate over-dispersed event count models.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "negbin", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

In addition to the standard inputs, `zelig()` takes the following additional options for negative binomial regression:

- `robust`: defaults to `FALSE`. If `TRUE` is selected, `zelig()` computes robust standard errors via the `sandwich` package (see Zeileis (2004)). The default type of robust standard error is heteroskedastic and autocorrelation consistent (HAC), and assumes that observations are ordered by time index.

  In addition, `robust` may be a list with the following options:

  - `method`: Choose from
    - `"vcovHAC"`: (default if `robust = TRUE`) HAC standard errors.
    - `"kernHAC"`: HAC standard errors using the weights given in Andrews (1991).
    - `"weave"`: HAC standard errors using the weights given in Lumley and Heagerty (1999).
  - `order.by`: defaults to `NULL` (the observations are chronologically ordered as in the original data). Optionally, you may specify a vector of weights (either as `order.by = z`, where `z` exists outside the data frame; or as `order.by = ~z`, where `z` is a variable in the data frame). The observations are chronologically ordered by the size of `z`.
  - `...`: additional options passed to the functions specified in `method`. See the `sandwich` library and Zeileis (2004) for more options.

Example

Load sample data:

```
> data(sanction)
```

Estimate the model:

```
> z.out <- zelig(num ~ target + coop, model = "negbin", data = sanction)
> summary(z.out)
```

Set values for the explanatory variables to their default mean values:

```
> x.out <- setx(z.out)
```

Simulate fitted values:

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

Model

Let Y_i be the number of independent events that occur during a fixed time period. This variable can take any non-negative integer value.

- The negative binomial distribution is derived by letting the mean of the Poisson distribution vary according to a fixed parameter ζ given by the Gamma distribution. The *stochastic component* is given by

$$\begin{aligned} Y_i \mid \zeta_i &\sim \text{Poisson}(\zeta_i \mu_i), \\ \zeta_i &\sim \frac{1}{\theta}\text{Gamma}(\theta). \end{aligned}$$

  The marginal distribution of Y_i is then the negative binomial with mean μ_i and variance $\mu_i + \mu_i^2/\theta$:

$$\begin{aligned} Y_i &\sim \text{NegBin}(\mu_i, \theta), \\ &= \frac{\Gamma(\theta + y_i)}{y!\,\Gamma(\theta)} \frac{\mu_i^{y_i}\,\theta^{\theta}}{(\mu_i + \theta)^{\theta + y_i}}, \end{aligned}$$

  where θ is the systematic parameter of the Gamma distribution modeling ζ_i.

- The *systematic component* is given by

$$\mu_i = \exp(x_i \beta)$$

  where x_i is the vector of k explanatory variables and β is the vector of coefficients.

Quantities of Interest

- The expected values (`qi$ev`) are simulations of the mean of the stochastic component. Thus,
$$E(Y) = \mu_i = \exp(x_i\beta),$$
given simulations of β.

- The predicted value (`qi$pr`) drawn from the distribution defined by the set of parameters (μ_i, θ).

- The first difference (`qi$fd`) is
$$\text{FD} \;=\; E(Y|x_1) - E(Y \mid x)$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is
$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - E[Y_i(t_i = 0)] \right\},$$
where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is
$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$
where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "negbin", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `theta`: the maximum likelihood estimate for the stochastic parameter θ.
  - `SE.theta`: the standard error for `theta`.
  - `residuals`: the working residuals in the final iteration of the IWLS fit.
  - `fitted.values`: a vector of the fitted values for the systemic component λ.
  - `linear.predictors`: a vector of $x_i\beta$.
  - `aic`: Akaike's Information Criterion (minus twice the maximized log-likelihood plus twice the number of coefficients).
  - `df.residual`: the residual degrees of freedom.
  - `df.null`: the residual degrees of freedom for the null model.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
- From `summary(z.out)`, you may extract:
  - `coefficients`: the parameter estimates with their associated standard errors, p-values, and t-statistics.
  - `cov.scaled`: a $k \times k$ matrix of scaled covariances.
  - `cov.unscaled`: a $k \times k$ matrix of unscaled covariances.
- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:
  - `qi$ev`: the simulated expected values given the specified values of `x`.
  - `qi$pr`: the simulated predicted values drawn from the distribution defined by (μ_i, θ).
  - `qi$fd`: the simulated first differences in the simulated expected values given the specified values of `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The negative binomial model is part of the MASS library by William N. Venables and Brian D. Ripley. Please cite the model as:

> Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

In addition, advanced users may wish to refer to `help(glm.nb)` in the MASS library and McCullagh and Nelder (1989).

Robust standard errors are implemented via the sandwich package by Achim Zeileis. Please cite as

> Zeileis, A. (2004), "Econometric Computing with HC and HAC Covariance Matrix Estimators," *Journal of Statistical Software*, 11, 1–17

Sample data are from

> Martin, L. (1992), *Coercive Cooperation: Explaining Multilateral Economic Sanctions*, Princeton University Press, please inquire with Lisa Martin before publishing results from these data, as this dataset includes errors that have since been corrected.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.20 normal: Normal Regression for Continuous Dependent Variables

The Normal regression model is a close variant of the more standard least squares regression model (see Section 12.16). Both models specify a continuous dependent variable as a linear function of a set of explanatory variables. The Normal model reports maximum likelihood (rather than least squares) estimates. The two models differ only in their estimate for the stochastic parameter σ.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "normal", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

In addition to the standard inputs, `zelig()` takes the following additional options for normal regression:

- `robust`: defaults to `FALSE`. If `TRUE` is selected, `zelig()` computes robust standard errors via the `sandwich` package (see Zeileis (2004)). The default type of robust standard error is heteroskedastic and autocorrelation consistent (HAC), and assumes that observations are ordered by time index.

  In addition, `robust` may be a list with the following options:

  - `method`: Choose from
    - `"vcovHAC"`: (default if `robust = TRUE`) HAC standard errors.
    - `"kernHAC"`: HAC standard errors using the weights given in Andrews (1991).
    - `"weave"`: HAC standard errors using the weights given in Lumley and Heagerty (1999).
  - `order.by`: defaults to `NULL` (the observations are chronologically ordered as in the original data). Optionally, you may specify a vector of weights (either as `order.by = z`, where `z` exists outside the data frame; or as `order.by = ~z`, where `z` is a variable in the data frame). The observations are chronologically ordered by the size of `z`.
  - `...`: additional options passed to the functions specified in `method`. See the `sandwich` library and Zeileis (2004) for more options.

Examples

1. Basic Example with First Differences

   Attach sample data:

```
> data(macro)
```

Estimate model:

```
> z.out1 <- zelig(unem ~ gdp + capmob + trade, model = "normal",
                  data = macro)
```

Summarize of regression coefficients:

```
> summary(z.out1)
```

Set explanatory variables to their default (mean/mode) values, with high (80th percentile) and low (20th percentile) values for trade:

```
> x.high <- setx(z.out, trade = quantile(macro$trade, 0.8))
> x.low <- setx(z.out, trade = quantile(macro$trade, 0.2))
```

Generate first differences for the effect of high versus low trade on GDP:

```
> s.out1 <- sim(z.out, x = x.high, x1 = x.low)
> summary(s.out1)
```

A visual summary of quantities of interest:

```
> plot(s.out1)
```

2. Using Dummy Variables

   Estimate a model with a dummy variable for each year and country (see 2 for help with dummy variables). Note that you do not need to create dummy variables, as the program will automatically parse the unique values in the selected variables into dummy variables.

   ```
   > z.out2 <- zelig(unem ~ gdp + trade + capmob + as.factor(year)
                     + as.factor(country), model = "normal", data = macro)
   ```

   Set values for the explanatory variables, using the default mean/mode variables, with country set to the United States and Japan, respectively:

   ```
   > x.US <- setx(z.out2, country = "United States")
   > x.Japan <- setx(z.out2, country = "Japan")
   ```

   Simulate quantities of interest:

   ```
   > s.out2 <- sim(z.out2, x = x.US, x1 = x.Japan)
   > summary(s.out2)
   > plot(s.out2)
   ```

Model

Let Y_i be the continuous dependent variable for observation i.

- The *stochastic component* is described by a univariate normal model with a vector of means μ_i and scalar variance σ^2:

$$Y_i \sim \text{Normal}(\mu_i, \sigma^2).$$

- The *systematic component* is

$$\mu_i = x_i\beta,$$

where x_i is the vector of k explanatory variables and β is the vector of coefficients.

Quantities of Interest

- The expected value (`qi$ev`) is the mean of simulations from the the stochastic component,

$$E(Y) = \mu_i = x_i\beta,$$

given a draw of β from its posterior.

- The predicted value (`qi$pr`) is drawn from the distribution defined by the set of parameters (μ_i, σ).

- The first difference (`qi$fd`) is:

$$\text{FD} = E(Y \mid x_1) - E(Y \mid x)$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - E[Y_i(t_i = 0)] \right\},$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "normal", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `residuals`: the working residuals in the final iteration of the IWLS fit.
  - `fitted.values`: fitted values. For the normal model, these are identical to the `linear predictors`.
  - `linear.predictors`: fitted values. For the normal model, these are identical to `fitted.values`.
  - `aic`: Akaike's Information Criterion (minus twice the maximized log-likelihood plus twice the number of coefficients).
  - `df.residual`: the residual degrees of freedom.
  - `df.null`: the residual degrees of freedom for the null model.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
- From `summary(z.out)`, you may extract:
  - `coefficients`: the parameter estimates with their associated standard errors, p-values, and t-statistics.
  - `cov.scaled`: a $k \times k$ matrix of scaled covariances.
  - `cov.unscaled`: a $k \times k$ matrix of unscaled covariances.
- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:
  - `qi$ev`: the simulated expected values for the specified values of `x`.
  - `qi$pr`: the simulated predicted values drawn from the distribution defined by (μ_i, σ).

- `qi$fd`: the simulated first difference in the simulated expected values for the values specified in `x` and `x1`.
- `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
- `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The Normal model is part of the stats package by William N. Venables and Brian D. Ripley. Please cite the model as:

> Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

Advanced users may wish to refer to `help(glm)`, `help(family)`, and McCullagh and Nelder (1989).

Robust standard errors are implemented via the sandwich package by Achim Zeileis. Please cite as

> Zeileis, A. (2004), "Econometric Computing with HC and HAC Covariance Matrix Estimators," *Journal of Statistical Software*, 11, 1–17.

Sample data are from

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

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.21 normal.bayes: Bayesian Normal Linear Regression

Use Bayesian regression to specify a continuous dependent variable as a linear function of specified explanatory variables. The model is implemented using a Gibbs sampler. See Section 12.20 for the maximum-likelihood implementation or Section 12.16 for the ordinary least squares variation.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "normal.bayes", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

Use the following arguments to monitor the convergence of the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 10,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA`, which corresponds to a random seed of 12345.
- `beta.start`: starting values for the Markov chain, either a scalar or vector with length equal to the number of estimated coefficients. The default is `NA`, which uses the least squares estimates as the starting values.

Use the following arguments to specify the model's priors:

- `b0`: prior mean for the coefficients, either a numeric vector or a scalar. If a scalar, that value will be the prior mean for all the coefficients. The default is 0.
- `B0`: prior precision parameter for the coefficients, either a square matrix (with the dimensions equal to the number of the coefficients) or a scalar. If a scalar, that value times an identity matrix will be the prior precision parameter. The default is 0, which leads to an improper prior.

- `c0`: `c0/2` is the shape parameter for the Inverse Gamma prior on the variance of the disturbance terms.
- `d0`: `d0/2` is the scale parameter for the Inverse Gamma prior on the variance of the disturbance terms.

Zelig users may wish to refer to `help(MCMCregress)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(macro)
   ```

   Estimating linear regression using `normal.bayes`:

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

Checking for convergence before summarizing the estimates:

```
> geweke.diag(z.out$coefficients)
> heidel.diag(z.out$coefficients)
> raftery.diag(z.out$coefficients)
> summary(z.out)
```

Setting values for the explanatory variables to their sample averages:

```
> x.out <- setx(z.out)
```

Simulating quantities of interest from the posterior distribution given `x.out`:

```
> s.out1 <- sim(z.out, x = x.out)
> summary(s.out1)
```

2. Simulating First Differences
   Set explanatory variables to their default(mean/mode) values, with high (80th percentile) and low (20th percentile) trade on GDP:

```
> x.high <- setx(z.out, trade = quantile(macro$trade, prob = 0.8))
> x.low <- setx(z.out, trade = quantile(macro$trade, prob = 0.2))
```

   Estimating the first difference for the effect of high versus low trade on unemployment rate:

```
> s.out2 <- sim(z.out, x = x.high, x1 = x.low)
> summary(s.out2)
```

Model

- The *stochastic component* is given by

$$\epsilon_i \sim \text{Normal}(0, \sigma^2)$$

  where $\epsilon_i = Y_i - \mu_i$.

- The *systematic component* is given by

$$\mu_i = x_i \beta,$$

  where x_i is the vector of k explanatory variables for observation i and β is the vector of coefficients.

- The *semi-conjugate priors* for β and σ^2 are given by

$$\begin{aligned} \beta &\sim \text{Normal}_k\left(b_0, B_0^{-1}\right) \\ \sigma^2 &\sim \text{InverseGamma}\left(\frac{c_0}{2}, \frac{d_0}{2}\right) \end{aligned}$$

  where b_0 is the vector of means for the k explanatory variables, B_0 is the $k \times k$ precision matrix (the inverse of a variance-covariance matrix), and $c_0/2$ and $d_0/2$ are the shape and scale parameters for σ^2. Note that β and σ^2 are assumed to be *a priori* independent.

Quantities of Interest

- The expected values (`qi$ev`) for the linear regression model are calculated as following:

$$E(Y) = x_i\beta,$$

  given posterior draws of β based on the MCMC iterations.

- The first difference (`qi$fd`) for the linear regression model is defined as

$$\text{FD} = E(Y \mid X_1) - E(Y \mid X).$$

- In conditional prediction models, the average expected treatment effect (`qi$att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^n t_i} \sum_{i:t_i=1} \{Y_i(t_i = 1) - E[Y_i(t_i = 0)]\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^n t_i} \sum_{i:t_i=1}^{n} \left\{Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)}\right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(y ~ x, model = "normal.bayes", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated parameters. The first k columns contain the posterior draws of the coefficients β, and the last column contains the posterior draws of the variance σ^2.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
  - `seed`: the random seed used in the model.
- From the `sim()` output object `s.out`:
  - `qi$ev`: the simulated expected values for the specified values of `x`.
  - `qi$fd`: the simulated first difference in the expected values for the values specified in `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.

Contributors

The Bayesian regression function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*

Ben Goodrich and Ying Lu enabled `normal.bayes` to work with Zelig.

12.22 `ologit`: Ordinal Logistic Regression for Ordered Categorical Dependent Variables

Use the ordinal logit regression model if your dependent variable is ordered and categorical, either in the form of integer values or character strings.

Syntax

```
> z.out <- zelig(as.factor(Y) ~ X1 + X2, model = "ologit", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

If `Y` takes discrete integer values, the `as.factor()` command will order automatically order the values. If `Y` takes on values composed of character strings, such as "strongly agree", "agree", and "disagree", `as.factor()` will order the values in the order in which they appear in `Y`. You will need to replace your dependent variable with a factored variable prior to estimating the model through `zelig()`. See Section 2 for more information on creating ordered factors and Example 1 below.

Example

1. Creating An Ordered Dependent Variable

   Load the sample data:

   ```
   > data(sanction)
   ```

   Create an ordered dependent variable:

   ```
   > sanction$ncost <- factor(sanction$ncost, ordered = TRUE,
                              levels = c("net gain", "little effect",
                              "modest loss", "major loss"))
   ```

   Estimate the model:

   ```
   > z.out <- zelig(ncost ~ mil + coop, model = "ologit", data = sanction)
   ```

   Set the explanatory variables to their observed values:

   ```
   > x.out <- setx(z.out, fn = NULL)
   ```

   Simulate fitted values given `x.out` and view the results:

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

2. First Differences

   Load the sample data:

   ```
   > data(sanction)
   ```

   Estimate the empirical model and returning the coefficients:

   ```
   > z.out <- zelig(as.factor(cost) ~ mil + coop, model = "ologit",
                    data = sanction)
   summary(z.out)
   ```

   Set the explanatory variables to their means, with `mil` set to 0 (no military action in addition to sanctions) in the baseline case and set to 1 (military action in addition to sanctions) in the alternative case:

   ```
   > x.low <- setx(z.out, mil = 0)
   > x.high <- setx(z.out, mil = 1)
   ```

   Generate simulated fitted values and first differences, and view the results:

   ```
   > s.out <- sim(z.out, x = x.low, x1 = x.high)
   > summary(s.out)
   ```

Model

Let Y_i be the ordered categorical dependent variable for observation i that takes one of the integer values from 1 to J where J is the total number of categories.

- The *stochastic component* begins with an unobserved continuous variable, Y_i^*, which follows the standard logistic distribution with a parameter μ_i,

  $$Y_i^* \sim \text{Logit}(y_i^* \mid \mu_i),$$

  to which we add an observation mechanism

  $$Y_i = j \quad \text{if} \quad \tau_{j-1} \leq Y_i^* \leq \tau_j \quad \text{for} \quad j = 1, \ldots, J.$$

  where τ_l (for $l = 0, \ldots, J$) are the threshold parameters with $\tau_l < \tau_m$ for all $l < m$ and $\tau_0 = -\infty$ and $\tau_J = \infty$.

- The *systematic component* has the following form, given the parameters τ_j and β, and the explanatory variables x_i:

  $$\Pr(Y \leq j) = \Pr(Y^* \leq \tau_j) = \frac{\exp(\tau_j - x_i\beta)}{1 + \exp(\tau_j - x_i\beta)},$$

  which implies:

  $$\pi_j = \frac{\exp(\tau_j - x_i\beta)}{1 + \exp(\tau_j - x_i\beta)} - \frac{\exp(\tau_{j-1} - x_i\beta)}{1 + \exp(\tau_{j-1} - x_i\beta)}.$$

Quantities of Interest

- The expected values (`qi$ev`) for the ordinal logit model are simulations of the predicted probabilities for each category:

$$E(Y = j) \;=\; \pi_j \;=\; \frac{\exp(\tau_j - x_i\beta)}{1 + \exp(\tau_j - x_i\beta)} - \frac{\exp(\tau_{j-1} - x_i\beta)}{1 + \exp(\tau_{j-1} - x_i\beta)},$$

  given a draw of β from its sampling distribution.

- The predicted value (`qi$pr`) is drawn from the logit distribution described by μ_i, and observed as one of J discrete outcomes.

- The difference in each of the predicted probabilities (`qi$fd`) is given by

$$\Pr(Y = j \mid x_1) \;-\; \Pr(Y = j \mid x) \quad \text{for} \quad j = 1, \ldots, J.$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{n_j} \sum_{i:t_i=1}^{n_j} \left\{ Y_i(t_i = 1) - E[Y_i(t_i = 0)] \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of treated observations in category j.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{n_j} \sum_{i:t_i=1}^{n_j} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of treated observations in category j.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "ologit", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.

  - `zeta`: a vector containing the estimated class boundaries τ_j.
  - `deviance`: the residual deviance.
  - `fitted.values`: the $n \times J$ matrix of in-sample fitted values.
  - `df.residual`: the residual degrees of freedom.
  - `edf`: the effective degrees of freedom.
  - `Hessian`: the Hessian matrix.
  - `zelig.data`: the input data frame if `save.data = TRUE`.

- From `summary(z.out)`, you may extract:
  - `coefficients`: the parameter estimates with their associated standard errors, and t-statistics.

- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as arrays. Available quantities are:
  - `qi$ev`: the simulated expected probabilities for the specified values of `x`, indexed by simulation $\times$ quantity $\times$ `x`-observation (for more than one `x`-observation).
  - `qi$pr`: the simulated predicted values drawn from the distribution defined by the expected probabilities, indexed by simulation $\times$ `x`-observation.
  - `qi$fd`: the simulated first difference in the predicted probabilities for the values specified in `x` and `x1`, indexed by simulation $\times$ quantity $\times$ `x`-observation (for more than one `x`-observation).
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The ordinal logit model is part of the MASS library by William N. Venables and Brian D. Ripley. Please cite the model as:

> Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

In addition, advanced users may wish to refer to `help(polr)` in the MASS libary and McCullagh and Nelder (1989).

Sample data are from

Martin, L. (1992), *Coercive Cooperation: Explaining Multilateral Economic Sanctions*, Princeton University Press, please inquire with Lisa Martin before publishing results from these data, as this dataset includes errors that have since been corrected.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.23 oprobit: Ordinal Probit Regression for Ordered Categorical Dependent Variables

Use the ordinal probit regression model if your dependent variables are ordered and categorical. They may take on either integer values or character strings. For a Bayesian implementation of this model, see Section 12.24.

Syntax

```
> z.out <- zelig(as.factor(Y) ~ X1 + X2, model = "oprobit", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

If `Y` takes discrete integer values, the `as.factor()` command will order it automatically. If `Y` takes on values composed of character strings, such as "strongly agree", "agree", and "disagree", `as.factor()` will order the values in the order in which they appear in `Y`. You will need to replace your dependent variable with a factored variable prior to estimating the model through `zelig()`. See Section 2 for more information on creating ordered factors and Example 1 below.

Example

1. Creating An Ordered Dependent Variable

   Load the sample data:

   ```
   > data(sanction)
   ```

   Create an ordered dependent variable:

   ```
   > sanction$ncost <- factor(sanction$ncost, ordered = TRUE,
                              levels = c("net gain", "little effect",
                              "modest loss", "major loss"))
   ```

   Estimate the model:

   ```
   > z.out <- zelig(ncost ~ mil + coop, model = "oprobit", data = sanction)
   ```

   Set the explanatory variables to their observed values:

   ```
   > x.out <- setx(z.out, fn = NULL)
   ```

   Simulate fitted values given `x.out` and view the results:

```
> s.out <- sim(z.out, x = x.out)
> summary(s.out)
> plot(s.out)
```

2. First Differences

   Load the sample data:

   ```
   > data(sanction)
   ```

   Estimate the empirical model and returning the coefficients:

   ```
   > z.out <- zelig(as.factor(cost) ~ mil + coop, model = "oprobit",
                    data = sanction)
   > summary(z.out)
   ```

   Set the explanatory variables to their means, with `mil` set to 0 (no military action in addition to sanctions) in the baseline case and set to 1 (military action in addition to sanctions) in the alternative case:

   ```
   > x.low <- setx(z.out, mil = 0)
   > x.high <- setx(z.out, mil = 1)
   ```

   Generate simulated fitted values and first differences, and view the results:

   ```
   > s.out <- sim(z.out, x = x.low, x1 = x.high)
   > summary(s.out)
   > plot(s.out)
   ```

Model

Let Y_i be the ordered categorical dependent variable for observation i that takes one of the integer values from 1 to J where J is the total number of categories.

- The *stochastic component* is described by an unobserved continuous variable, Y_i^*, which follows the normal distribution with mean μ_i and unit variance

  $$Y_i^* \sim N(\mu_i, 1).$$

  The observation mechanism is

  $$Y_i = j \quad \text{if} \quad \tau_{j-1} \leq Y_i^* \leq \tau_j \quad \text{for} \quad j = 1, \ldots, J.$$

  where τ_k for $k = 0, \ldots, J$ is the threshold parameter with the following constraints; $\tau_l < \tau_m$ for all $l < m$ and $\tau_0 = -\infty$ and $\tau_J = \infty$.

Given this observation mechanism, the probability for each category, is given by

$$\Pr(Y_i = j) \;=\; \Phi(\tau_j \mid \mu_i) - \Phi(\tau_{j-1} \mid \mu_i) \quad \text{for} \quad j = 1, \ldots, J$$

where $\Phi(\mu_i)$ is the cumulative distribution function for the Normal distribution with mean μ_i and unit variance.

- The *systematic component* is given by

$$\mu_i \;=\; x_i\beta$$

where x_i is the vector of explanatory variables and β is the vector of coefficients.

Quantities of Interest

- The expected values (`qi$ev`) for the ordinal probit model are simulations of the predicted probabilities for each category:

$$E(Y_i = j) \;=\; \Pr(Y_i = j) \;=\; \Phi(\tau_j \mid \mu_i) - \Phi(\tau_{j-1} \mid \mu_i) \quad \text{for} \quad j = 1, \ldots, J,$$

given draws of β from its posterior.

- The predicted value (`qi$pr`) is the observed value of Y_i given the underlying standard normal distribution described by μ_i.

- The difference in each of the predicted probabilities (`qi$fd`) is given by

$$\Pr(Y = j \mid x_1) \;-\; \Pr(Y = j \mid x) \quad \text{for} \quad j = 1, \ldots, J.$$

- In conditional prediction models, the average expected treatment effect (`qi$att.ev`) for the treatment group in category j is

$$\frac{1}{n_j} \sum_{i:t_i=1}^{n_j} [Y_i(t_i = 1) - E[Y_i(t_i = 0)]],$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of treated observations in category j.

- In conditional prediction models, the average predicted treatment effect (`qi$att.pr`) for the treatment group in category j is

$$\frac{1}{n_j} \sum_{i:t_i=1}^{n_j} [Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)}],$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of treated observations in category j.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "oprobit", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: the named vector of coefficients.
  - `fitted.values`: an $n \times J$ matrix of the in-sample fitted values.
  - `predictors`: an $n \times (J-1)$ matrix of the linear predictors $x_i\beta_j$.
  - `residuals`: an $n \times (J-1)$ matrix of the residuals.
  - `df.residual`: the residual degrees of freedom.
  - `df.total`: the total degrees of freedom.
  - `rss`: the residual sum of squares.
  - `y`: an $n \times J$ matrix of the dependent variables.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
- From `summary(z.out)`, you may extract:
  - `coef3`: a table of the coefficients with their associated standard errors and t-statistics.
  - `cov.unscaled`: the variance-covariance matrix.
  - `pearson.resid`: an $n \times (m-1)$ matrix of the Pearson residuals.
- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as arrays. Available quantities are:
  - `qi$ev`: the simulated expected probabilities for the specified values of `x`, indexed by simulation $\times$ quantity $\times$ `x`-observation (for more than one `x`-observation).
  - `qi$pr`: the simulated predicted values drawn from the distribution defined by the expected probabilities, indexed by simulation $\times$ `x`-observation.
  - `qi$fd`: the simulated first difference in the predicted probabilities for the values specified in `x` and `x1`, indexed by simulation $\times$ quantity $\times$ `x`-observation (for more than one `x`-observation).
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The ordinal probit function is part of the VGAM package by Thomas Yee. Please cite the model as:

> Yee, T. W. and Hastie, T. J. (2003), "Reduced-rank vector generalized linear models," *Statistical Modelling*, 3, 15–41.

In addition, advanced users may wish to refer to `help(vglm)` in the VGAM library. Additional documentation is available at http://www.stat.auckland.ac.nz/~ yee.

Sample data are from

> Martin, L. (1992), *Coercive Cooperation: Explaining Multilateral Economic Sanctions*, Princeton University Press, please inquire with Lisa Martin before publishing results from these data, as this dataset includes errors that have since been corrected.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.24 `oprobit.bayes`: Bayesian Ordered Probit Regression

Use the ordinal probit regression model if your dependent variables are ordered and categorical. They may take either integer values or character strings. The model is estimated using a Gibbs sampler with data augmentation. For a maximum-likelihood implementation of this models, see Section 12.23.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "oprobit.bayes", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

`zelig()` accepts the following arguments to monitor the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults 10,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `tune`: tuning parameter for the Metropolis-Hasting step. The default value is `NA` which corresponds to 0.05 divided by the number of categories in the response variable.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed 12345.
- `beta.start`: starting values for the Markov chain, either a scalar or vector with length equal to the number of estimated coefficients. The default is `NA`, which uses the maximum likelihood estimates as the starting values.

Use the following parameters to specify the model's priors:

- `b0`: prior mean for the coefficients, either a numeric vector or a scalar. If a scalar value, that value will be the prior mean for all the coefficients. The default is 0.

- `B0`: prior precision parameter for the coefficients, either a square matrix (with dimensions equal to the number of coefficients) or a scalar. If a scalar value, that value times an identity matrix will be the prior precision parameter. The default is 0 which leads to an improper prior.

Zelig users may wish to refer to `help(MCMCoprobit)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(sanction)
   ```

   Estimating ordered probit regression using `oprobit.bayes`:

   ```
   > z.out <- zelig(ncost ~ mil + coop, model = "oprobit.bayes",
                    data = sanction, verbose=TRUE)
   ```

Creating an ordered dependent variable:

```
sanction$ncost <- factor(sanction$ncost, ordered = TRUE,
                         levels = c("net gain", "little effect",
                         "modest loss", "major loss"))
```

Checking for convergence before summarizing the estimates:

```
> heidel.diag(z.out$coefficients)
> raftery.diag(z.out$coefficients)
> summary(z.out)
```

Setting values for the explanatory variables to their sample averages:

```
> x.out <- setx(z.out)
```

Simulating quantities of interest from the posterior distribution given: `x.out`.

```
> s.out1 <- sim(z.out, x = x.out)
> summary(s.out1)
```

2. Simulating First Differences
   Estimating the first difference (and risk ratio) in the probabilities of incurring different level of cost when there is no military action versus military action while all the other variables held at their default values.

```
> x.high <- setx(z.out, mil=0)
> x.low <- setx(z.out, mil=1)
> s.out2 <- sim(z.out, x = x.high, x1 = x.low)
> summary(s.out2)
```

Model

Let Y_i be the ordered categorical dependent variable for observation i which takes an integer value $j = 1, \ldots, J$.

- The *stochastic component* is described by an unobserved continuous variable, Y_i^*,

$$Y_i^* \sim \text{Normal}(\mu_i, 1).$$

  Instead of Y_i^*, we observe categorical variable Y_i,

$$Y_i = j \quad \text{if } \tau_{j-1} \leq Y_i^* \leq \tau_j \text{ for } j = 1, \ldots, J.$$

where τ_j for $j = 0, \ldots, J$ are the threshold parameters with the following constraints, $\tau_l < \tau_m$ for $l < m$, and $\tau_0 = -\infty, \tau_J = \infty$.

The probability of observing Y_i equal to category j is,

$$\Pr(Y_i = j) \quad = \quad \Phi(\tau_j \mid \mu_i) - \Phi(\tau_{j-1} \mid \mu_i) \text{ for } j = 1, \ldots, J$$

where $\Phi(\cdot \mid \mu_i)$ is the cumulative distribution function of the Normal distribution with mean μ_i and variance 1.

- The *systematic component* is given by

$$\mu_i = x_i \beta,$$

where x_i is the vector of k explanatory variables for observation i and β is the vector of coefficients.

- The *prior* for β is given by

$$\beta \sim \text{Normal}_k \left(b_0, B_0^{-1} \right)$$

where b_0 is the vector of means for the k explanatory variables and B_0 is the $k \times k$ precision matrix (the inverse of a variance-covariance matrix).

Quantities of Interest

- The expected values (`qi$ev`) for the ordered probit model are the predicted probability of belonging to each category:

$$\Pr(Y_i = j) = \Phi(\tau_j \mid x_i \beta) - \Phi(\tau_{j-1} \mid x_i \beta),$$

given the posterior draws of β and threshold parameters τ from the MCMC iterations.

- The predicted values (`qi$pr`) are the observed values of Y_i given the observation scheme and the posterior draws of β and cut points τ from the MCMC iterations.

- The first difference (`qi$fd`) in category j for the ordered probit model is defined as

$$\text{FD}_j = \Pr(Y_i = j \mid X_1) - \Pr(Y_i = j \mid X).$$

- The risk ratio (`qi$rr`) in category j is defined as

$$\text{RR}_j = \Pr(Y_i = j \mid X_1) \ / \ \Pr(Y_i = j \mid X).$$

- In conditional prediction models, the average expected treatment effect (`qi$att.ev`) for the treatment group in category j is

$$\frac{1}{n_j}\sum_{i:t_i=1}^{n_j}\{Y_i(t_i=1)-E[Y_i(t_i=0)]\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of observations in the treatment group that belong to category j.

- In conditional prediction models, the average predicted treatment effect (`qi$att.pr`) for the treatment group in category j is

$$\frac{1}{n_j}\sum_{i:t_i=1}^{n_j}[Y_i(t_i=1)-\widehat{Y_i(t_i=0)}],$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups, and n_j is the number of observations in the treatment group that belong to category j.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(y ~ x, model = "oprobit.bayes", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated coefficients β and threshold parameters τ. Note, element τ_1 is normalized to 0 and is not returned in the `coefficients` object.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
  - `seed`: the random seed used in the model.

- From the `sim()` output object `s.out`:
  - `qi$ev`: the simulated expected values (probabilities) of each of the J categories for the specified values of `x`.

- `qi$pr`: the simulated predicted values (observed values) for the specified values of `x`.
- `qi$fd`: the simulated first difference in the expected values of each of the J categories for the values specified in `x` and `x1`.
- `qi$rr`: the simulated risk ratio for the expected values of each of the J categories simulated from `x` and `x1`.
- `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
- `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

Bayesian ordinal probit regression function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*

Ben Goodrich and Ying Lu enabled `oprobit.bayes` to work with Zelig.

12.25 poisson: Poisson Regression for Event Count Dependent Variables

Use the Poisson regression model if the observations of your dependent variable represents the number of independent events that occur during a fixed period of time (see the negative binomial model, Section 12.19, for over-dispersed event counts.) For a Bayesian implementation of this model, see Section 12.26.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "poisson", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

In addition to the standard inputs, `zelig()` takes the following additional options for poisson regression:

- `robust`: defaults to `FALSE`. If `TRUE` is selected, `zelig()` computes robust standard errors via the `sandwich` package (see Zeileis (2004)). The default type of robust standard error is heteroskedastic and autocorrelation consistent (HAC), and assumes that observations are ordered by time index.

  In addition, `robust` may be a list with the following options:

  - `method`: Choose from
    - `"vcovHAC"`: (default if `robust = TRUE`) HAC standard errors.
    - `"kernHAC"`: HAC standard errors using the weights given in Andrews (1991).
    - `"weave"`: HAC standard errors using the weights given in Lumley and Heagerty (1999).
  - `order.by`: defaults to `NULL` (the observations are chronologically ordered as in the original data). Optionally, you may specify a vector of weights (either as `order.by = z`, where `z` exists outside the data frame; or as `order.by = ~z`, where `z` is a variable in the data frame). The observations are chronologically ordered by the size of `z`.
  - `...`: additional options passed to the functions specified in `method`. See the `sandwich` library and Zeileis (2004) for more options.

Example

Load sample data:

```
> data(sanction)
```

Estimate Poisson model:

```
> z.out <- zelig(num ~ target + coop, model = "poisson", data = sanction)
> summary(z.out)
```

Set values for the explanatory variables to their default mean values:

```
> x.out <- setx(z.out)
```

Simulate fitted values:

```
> s.out <- sim(z.out, x = x.out
> summary(s.out)
> plot(s.out)
```

Model

Let Y_i be the number of independent events that occur during a fixed time period. This variable can take any non-negative integer.

- The Poisson distribution has *stochastic component*

$$Y_i \sim \text{Poisson}(\lambda_i),$$

  where λ_i is the mean and variance parameter.

- The *systematic component* is

$$\lambda_i = \exp(x_i\beta),$$

  where x_i is the vector of explanatory variables, and β is the vector of coefficients.

Quantities of Interest

- The expected value (`qi$ev`) is the mean of simulations from the stochastic component,

$$E(Y) = \lambda_i = \exp(x_i\beta),$$

  given draws of β from its sampling distribution.

- The predicted value (`qi$pr`) is a random draw from the poisson distribution defined by mean λ_i.

- The first difference in the expected values (`qi$fd`) is given by:

$$\text{FD} = E(Y|x_1) - E(Y \mid x)$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - E[Y_i(t_i = 0)] \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "poisson", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `residuals`: the working residuals in the final iteration of the IWLS fit.
  - `fitted.values`: a vector of the fitted values for the systemic component λ.
  - `linear.predictors`: a vector of $x_i\beta$.
  - `aic`: Akaike's Information Criterion (minus twice the maximized log-likelihood plus twice the number of coefficients).
  - `df.residual`: the residual degrees of freedom.
  - `df.null`: the residual degrees of freedom for the null model.

- `zelig.data`: the input data frame if `save.data = TRUE`.

- From `summary(z.out)`, you may extract:
  - `coefficients`: the parameter estimates with their associated standard errors, p-values, and t-statistics.
  - `cov.scaled`: a $k \times k$ matrix of scaled covariances.
  - `cov.unscaled`: a $k \times k$ matrix of unscaled covariances.

- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:
  - `qi$ev`: the simulated expected values given the specified values of `x`.
  - `qi$pr`: the simulated predicted values drawn from the distributions defined by λ_i.
  - `qi$fd`: the simulated first differences in the expected values given the specified values of `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The Poisson model is part of the stats package by William N. Venables and Brian D. Ripley. Please cite the model as:

> Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

Advanced users may wish to refer to `help(glm)` and `help(family)`, as well as McCullagh and Nelder (1989).

Robust standard errors are implemented via the sandwich package by Achim Zeileis. Please cite as

> Zeileis, A. (2004), "Econometric Computing with HC and HAC Covariance Matrix Estimators," *Journal of Statistical Software*, 11, 1–17.

Sample data are from

> Martin, L. (1992), *Coercive Cooperation: Explaining Multilateral Economic Sanctions*, Princeton University Press, please inquire with Lisa Martin before publishing results from these data, as this dataset includes errors that have since been corrected.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.26 `poisson.bayes`: Bayesian Poisson Regression

Use the Poisson regression model if the observations of your dependent variable represents the number of independent events that occur during a fixed period of time. The model is fit using a random walk Metropolis algorithm. For a maximum-likelihood estimation of this model see Section 12.25.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "poisson.bayes", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

Use the following argument to monitor the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 10,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `tune`: Metropolis tuning parameter, either a positive scalar or a vector of length k, where k is the number of coefficients. The tuning parameter should be set such that the acceptance rate of the Metropolis algorithm is satisfactory (typically between 0.20 and 0.5). The default value is 1.1.
- `verbose`: default to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed of 12345.
- `beta.start`: starting values for the Markov chain, either a scalar or vector with length equal to the number of estimated coefficients. The default is `NA`, such that the maximum likelihood estimates are used as the starting values.

Use the following parameters to specify the model's priors:

- `b0`: prior mean for the coefficients, either a numeric vector or a scalar. If a scalar, that value will be the prior mean for all the coefficients. The default is 0.

- `B0`: prior precision parameter for the coefficients, either a square matrix (with the dimensions equal to the number of the coefficients) or a scalar. If a scalar, that value times an identity matrix will be the prior precision parameter. The default is 0, which leads to an improper prior.

Zelig users may wish to refer to `help(MCMCpoisson)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(sanction)
   ```

   Estimating the Poisson regression using `poisson.bayes`:

   ```
   > z.out <- zelig(num ~ target + coop, model = "poisson.bayes",
                    data = sanction, verbose = TRUE)
   ```

Checking convergence diagnostics before summarizing the estimates:

```
> geweke.diag(z.out$coefficients)
> heidel.diag(z.out$coefficients)
> raftery.diag(z.out$coefficients)
> summary(z.out)
```

Setting values for the explanatory variables to their sample averages:

```
> x.out <- setx(z.out)
```

Simulating quantities of interest from the posterior distribution given `x.out`.

```
> s.out1 <- sim(z.out, x = x.out)
> summary(s.out1)
```

2. Simulating First Differences
   Estimating the first difference in the number of countries imposing sanctions when the number of targets is set to be its maximum versus its minimum :

```
> x.max <- setx(z.out, target = max(sanction$target))
> x.min <- setx(z.out, target = min(sanction$target))
> s.out2 <- sim(z.out, x = x.max, x1 = x.min)
> summary(s.out2)
```

Model

Let Y_i be the number of independent events that occur during a fixed time period.

- The *stochastic component* is given by

$$Y_i \sim \text{Poisson}(\lambda_i)$$

  where λ_i is the mean and variance parameter.

- The *systematic component* is given by

$$\lambda_i = \exp(x_i\beta)$$

  where x_i is the vector of k explanatory variables for observation i and β is the vector of coefficients.

- The *prior* for β is given by

$$\beta \sim \text{Normal}_k\left(b_0, B_0^{-1}\right)$$

  where b_0 is the vector of means for the k explanatory variables and B_0 is the $k \times k$ precision matrix (the inverse of a variance-covariance matrix).

Quantities of Interest

- The expected values (`qi$ev`) for the Poisson model are calculated as following:

$$E(Y \mid X) = \lambda_i = \exp(x_i\beta),$$

  given the posterior draws of β based on the MCMC iterations.

- The predicted values (`qi$pr`) are draws from the Poisson distribution with parameter λ_i.

- The first difference (`qi$fd`) for the Poisson model is defined as

$$\text{FD} = E(Y \mid X_1) - E(Y \mid X).$$

- In conditional prediction models, the average expected treatment effect (`qi$att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1} \{Y_i(t_i = 1) - E[Y_i(t_i = 0)]\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

- In conditional prediction models, the average predicted treatment effect (`qi$att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1} [Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)}],$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(y ~ x, model = "poisson.bayes", data)
```

you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:

- `coefficients`: draws from the posterior distributions of the estimated parameters.
- `zelig.data`: the input data frame if `save.data = TRUE`.
- `seed`: the random seed used in the model.

- From the `sim()` output object `s.out`:
  - `qi$ev`: the simulated expected values for the specified values of `x`.
  - `qi$pr`: the simulated predicted values for the specified values of `x`.
  - `qi$fd`: the simulated first difference in the expected values for the values specified in `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

Bayesian Poisson regression function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*

Ben Goodrich and Ying Lu enabled `poisson.bayes` to work with Zelig.

12.27 probit: Probit Regression for Dichotomous Dependent Variables

Use probit regression to model binary dependent variables specified as a function of a set of explanatory variables. For a Bayesian implementation of this model, see Section 12.28.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "probit", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out, x1 = NULL)
```

Additional Inputs

In addition to the standard inputs, `zelig()` takes the following additional options for probit regression:

- `robust`: defaults to `FALSE`. If `TRUE` is selected, `zelig()` computes robust standard errors via the `sandwich` package (see Zeileis (2004)). The default type of robust standard error is heteroskedastic and autocorrelation consistent (HAC), and assumes that observations are ordered by time index.

  In addition, `robust` may be a list with the following options:

  - `method`: Choose from
    - `"vcovHAC"`: (default if `robust = TRUE`) HAC standard errors.
    - `"kernHAC"`: HAC standard errors using the weights given in Andrews (1991).
    - `"weave"`: HAC standard errors using the weights given in Lumley and Heagerty (1999).
  - `order.by`: defaults to `NULL` (the observations are chronologically ordered as in the original data). Optionally, you may specify a vector of weights (either as `order.by = z`, where `z` exists outside the data frame; or as `order.by = ~z`, where `z` is a variable in the data frame). The observations are chronologically ordered by the size of `z`.
  - `...`: additional options passed to the functions specified in `method`. See the `sandwich` library and Zeileis (2004) for more options.

Examples

Attach the sample turnout dataset:

```
> data(turnout)
```

Estimate parameter values for the probit regression:

```
> z.out <- zelig(vote ~ race + educate,  model = "probit", data = turnout)
> summary(z.out)
```

Set values for the explanatory variables to their default values.

```
> x.out <- setx(z.out)
```

Simulate quantities of interest from the posterior distribution.

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

Model

Let Y_i be the observed binary dependent variable for observation i which takes the value of either 0 or 1.

- The *stochastic component* is given by

$$Y_i \sim \text{Bernoulli}(\pi_i),$$

  where $\pi_i = \Pr(Y_i = 1)$.

- The *systematic component* is

$$\pi_i = \Phi(x_i\beta)$$

  where $\Phi(\mu)$ is the cumulative distribution function of the Normal distribution with mean 0 and unit variance.

Quantities of Interest

- The expected value (`qi$ev`) is a simulation of predicted probability of success

$$E(Y) = \pi_i = \Phi(x_i\beta),$$

  given a draw of β from its sampling distribution.

- The predicted value (`qi$pr`) is a draw from a Bernoulli distribution with mean π_i.

- The first difference (`qi$fd`) in expected values is defined as

$$\text{FD} = \Pr(Y = 1 \mid x_1) - \Pr(Y = 1 \mid x).$$

- The risk ratio (`qi$rr`) is defined as

$$\text{RR} = \Pr(Y = 1 \mid x_1) / \Pr(Y = 1 \mid x).$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - E[Y_i(t_i = 0)] \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "probit", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `residuals`: the working residuals in the final iteration of the IWLS fit.
  - `fitted.values`: a vector of the in-sample fitted values.
  - `linear.predictors`: a vector of $x_i\beta$.
  - `aic`: Akaike's Information Criterion (minus twice the maximized log-likelihood plus twice the number of coefficients).
  - `df.residual`: the residual degrees of freedom.
  - `df.null`: the residual degrees of freedom for the null model.

  - `data`: the name of the input data frame.

- From `summary(z.out)`, you may extract:
  - `coefficients`: the parameter estimates with their associated standard errors, p-values, and t-statistics.
  - `cov.scaled`: a $k \times k$ matrix of scaled covariances.
  - `cov.unscaled`: a $k \times k$ matrix of unscaled covariances.

- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:
  - `qi$ev`: the simulated expected values, or predicted probabilities, for the specified values of `x`.
  - `qi$pr`: the simulated predicted values drawn from the distributions defined by the predicted probabilities.
  - `qi$fd`: the simulated first differences in the predicted probabilities for the values specified in `x` and `x1`.
  - `qi$rr`: the simulated risk ratio for the predicted probabilities simulated from `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The probit model is part of the base package by William N. Venables and Brian D. Ripley. Please cite the model as:

> Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

In addition, advanced users may wish to refer to `help(glm)` and `help(family)`, as well as McCullagh and Nelder (1989).

Robust standard errors are implemented via the sandwich package by Achim Zeileis. Please cite as

> Zeileis, A. (2004), "Econometric Computing with HC and HAC Covariance Matrix Estimators," *Journal of Statistical Software*, 11, 1–17.

Sample data are a selection of 2,000 observations from

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

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.28 probit.bayes: Bayesian Probit Regression

Use the probit regression model for model binary dependent variables specified as a function of a set of explanatory variables. The model is estimated using a Gibbs sampler. For other models suitable for binary response variables, see Bayesian logistic regression(Section 12.14), maximum likelihood logit regression (Section 12.13), and maximum likelihood probit regression (Section 12.27).

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "probit.bayes", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Additional Inputs

Using the following arguments to monitor the Markov chains:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 10,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed of 12345.
- `beta.start`: starting values for the Markov chain, either a scalar or vector with length equal to the number of estimated coefficients. The default is `NA`, such that the maximum likelihood estimates are used as the starting values.

Use the following parameters to specify the model's priors:

- `b0`: prior mean for the coefficients, either a numeric vector or a scalar. If a scalar value, that value will be the prior mean for all the coefficients. The default is 0.
- `B0`: prior precision parameter for the coefficients, either a square matrix (with the dimensions equal to the number of the coefficients) or a scalar. If a scalar value, that value times an identity matrix will be the prior precision parameter. The default is 0, which leads to an improper prior.

Use the following arguments to specify optional output for the model:

- `bayes.resid`: defaults to `FALSE`. If `TRUE`, the latent Bayesian residuals for all observations are returned. Alternatively, users can specify a vector of observations for which the latent residuals should be returned.

Zelig users may wish to refer to `help(MCMCprobit)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(turnout)
   ```

   Estimating the probit regression using `probit.bayes`:

   ```
   > z.out < - zelig(vote ~ race + educate, model = "probit.bayes",
                     data = turnout, verbose = TRUE)
   ```

   Checking for convergence before summarizing the estimates:

```
> geweke.diag(z.out$coefficients)
> heidel.diag(z.out$coefficients)
> raftery.diag(z.out$coefficients)
> summary(z.out)
```

Setting values for the explanatory variables to their sample averages:

```
> x.out <- setx(z.out)
```

Simulating quantities of interest from the posterior distribution given: `x.out`.

```
> s.out1 <- sim(z.out, x = x.out)
> summary(s.out1)
```

2. Simulating First Differences
   Estimating the first difference (and risk ratio) in individual's probability of voting when education is set to be low (25th percentile) versus high (75th percentile) while all the other variables are held at their default values:

```
> x.high <- setx(z.out, educate = quantile(turnout$educate, prob = 0.75))
> x.low <- setx(z.out, educate = quantile(turnout$educate, prob = 0.25))
> s.out2 <- sim(z.out, x = x.high, x1 = x.low)
> summary(s.out2)
```

Model

Let Y_i be the binary dependent variable for observation i which takes the value of either 0 or 1.

- The *stochastic component* is given by

$$\begin{aligned} Y_i &\sim \text{Bernoulli}(\pi_i) \\ &= \pi_i^{Y_i}(1-\pi_i)^{1-Y_i}, \end{aligned}$$

  where $\pi_i = \Pr(Y_i = 1)$.

- The *systematic component* is given by

$$\pi_i = \Phi(x_i\beta),$$

  where $\Phi(\cdot)$ is the cumulative density function of the standard Normal distribution with mean 0 and variance 1, x_i is the vector of k explanatory variables for observation i, and β is the vector of coefficients.

- The *prior* for β is given by

$$\beta \sim \text{Normal}_k\left(b_0, B_0^{-1}\right)$$

  where b_0 is the vector of means for the k explanatory variables and B_0 is the $k \times k$ precision matrix (the inverse of a variance-covariance matrix).

Quantities of Interest

- The expected values (`qi$ev`) for the probit model are the predicted probability of a success:

$$E(Y \mid X) = \pi_i = \Phi(x_i\beta),$$

  given the posterior draws of β from the MCMC iterations.

- The predicted values (`qi$pr`) are draws from the Bernoulli distribution with mean equal to the simulated expected value π_i.

- The first difference (`qi$fd`) for the probit model is defined as

$$\text{FD} = \Pr(Y = 1 \mid X_1) - \Pr(Y = 1 \mid X).$$

- The risk ratio (`qi$rr`)is defined as

$$\text{RR} = \Pr(Y = 1 \mid X_1) \ / \ \Pr(Y = 1 \mid X).$$

- In conditional prediction models, the average expected treatment effect (`qi$att.ev`) for the treatment group is

$$\frac{1}{\sum t_i} \sum_{i:t_i=1} [Y_i(t_i = 1) - E[Y_i(t_i = 0)]],$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

- In conditional prediction models, the average predicted treatment effect (`qi$att.pr`) for the treatment group is

$$\frac{1}{\sum t_i} \sum_{i:t_i=1} [Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)}],$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(y ~ x, model = "probit.bayes", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated parameters.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
  - `bayes.residuals`: When `bayes.residual` is `TRUE` or a set of observation numbers is given, this object contains the posterior draws of the latent Bayesian residuals of all the observations or the observations specified by the user.
  - `seed`: the random seed used in the model.
- From the `sim()` output object `s.out`:
  - `qi$ev`: the simulated expected values (probabilities) for the specified values of `x`.
  - `qi$pr`: the simulated predicted values for the specified values of `x`.
  - `qi$fd`: the simulated first difference in the expected values for the values specified in `x` and `x1`.
  - `qi$rr`: the simulated risk ratio for the expected values simulated from `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

Bayesian probit regression function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*

Ben Goodrich and Ying Lu enabled `probit.bayes` to work with Zelig.

12.29 relogit: Rare Events Logistic Regression for Dichotomous Dependent Variables

The relogit procedure estimates the same model as standard logistic regression (appropriate when you have a dichotomous dependent variable and a set of explanatory variables; see Section 12.13), but the estimates are corrected for the bias that occurs when the sample is small or the observed events are rare (i.e., if the dependent variable has many more 1s than 0s or the reverse). The relogit procedure also optionally uses prior correction for case-control sampling designs.

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, model = "relogit", tau = NULL,
                 case.correct = c("prior", "weighting"),
                 bias.correct = TRUE, robust = FALSE,
                 data = mydata, ...)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Arguments

The relogit procedure supports four optional arguments in addition to the standard arguments for zelig(). You may additionally use:

- tau: a vector containing either one or two values for τ, the true population fraction of ones. Use, for example, tau = c(0.05, 0.1) to specify that the lower bound on tau is 0.05 and the upper bound is 0.1. If left unspecified, only finite-sample bias correction is performed, not case-control correction.

- case.correct: if tau is specified, choose a method to correct for case-control sampling design: "prior" (default) or "weighting".

- bias.correct: a logical value of TRUE (default) or FALSE indicating whether the intercept should be corrected for finite sample (rare events) bias.

- robust: defaults to FALSE (except when case.control = "weighting"; the default in this case becomes robust = TRUE). If TRUE is selected, zelig() computes robust standard errors via the sandwich package (see Zeileis (2004)). The default type of robust standard error is heteroskedastic and autocorrelation consistent (HAC), and assumes that observations are ordered by time index.

  In addition, robust may be a list with the following options:

  - method: Choose from

    * "vcovHAC": (default if robust = TRUE) HAC standard errors.

    - "kernHAC": HAC standard errors using the weights given in Andrews (1991).
    - "weave": HAC standard errors using the weights given in Lumley and Heagerty (1999).
  - order.by: defaults to NULL (the observations are chronologically ordered as in the original data). Optionally, you may specify a vector of weights (either as order.by = z, where z exists outside the data frame; or as order.by = ~z, where z is a variable in the data frame) The observations are chronologically ordered by the size of z.
  - ...: additional options passed to the functions specified in method. See the sandwich library and Zeileis (2004) for more options.

Note that if tau = NULL, bias.correct = FALSE, robust = FALSE, the relogit procedure performs a standard logistic regression without any correction.

Example 1: One Tau with Prior Correction and Bias Correction

Due to memory and space considerations, the data used here are a sample drawn from the full data set used in King and Zeng, 2001, The proportion of militarized interstate conflicts to the absence of disputes is $\tau = 1,042/303,772 \approx 0.00343$. To estimate the model,

```
> data(mid)
> z.out1 <- zelig(conflict ~ major + contig + power + maxdem + mindem + years,
                  data = mid, model = "relogit", tau = 1042/303772)
```

Summarize the model output:

```
> summary(z.out1)
```

Set the explanatory variables to their means:

```
> x.out1 <- setx(z.out1)
```

Simulate quantities of interest:

```
> s.out1 <- sim(z.out1, x = x.out1)
> summary(s.out1)
> plot(s.out1)
```

Example 2: One Tau with Weighting, Robust Standard Errors, and Bias Correction

Suppose that we wish to perform case control correction using weighting (rather than the default prior correction). To estimate the model:

```
> data(mid)
> z.out2 <- zelig(conflict ~ major + contig + power + maxdem + mindem + years,
                  data = mid, model = "relogit", tau = 1042/303772,
                  case.control = "weighting", robust = TRUE)
```

Summarize the model output:

```
> summary(z.out2)
```

Set the explanatory variables to their means:

```
> x.out2 <- setx(z.out2)
```

Simulate quantities of interest:

```
> s.out2 <- sim(z.out2, x = x.out2)
> summary(s.out2)
```

Example 3: Two Taus with Bias Correction and Prior Correction

Suppose that we did not know that $\tau \approx 0.00343$, but only that it was somewhere between $(0.002, 0.005)$. To estimate a model with a range of feasible estimates for τ (using the default prior correction method for case control correction):

```
> data(mid)
> z.out2 <- zelig(conflict ~ major + contig + power + maxdem + mindem
                  + years, data = mid, model = "relogit",
                  tau = c(0.002, 0.005))
```

Summarize the model output:

```
> summary(z.out2)
```

Set the explanatory variables to their means:

```
> x.out2 <- setx(z.out2)
```

Simulate quantities of interest:

```
> s.out <- sim(z.out2, x = x.out2)
> summary(s.out2)
> plot(s.out2)
```

The cost of giving a range of values for τ is that point estimates are not available for quantities of interest. Instead, quantities are presented as confidence intervals with significance less than or equal to a specified level (e.g., at least 95% of the simulations are contained in the nominal 95% confidence interval).

Model

- Like the standard logistic regression, the *stochastic component* for the rare events logistic regression is:

$$Y_i \sim \text{Bernoulli}(\pi_i),$$

where Y_i is the binary dependent variable, and takes a value of either 0 or 1.

- The *systematic component* is:

$$\pi_i = \frac{1}{1+\exp(-x_i\beta)}.$$

- If the sample is generated via a case-control (or choice-based) design, such as when drawing all events (or "cases") and a sample from the non-events (or "controls") and going backwards to collect the explanatory variables, you must correct for selecting on the dependent variable. While the slope coefficients are approximately unbiased, the constant term may be significantly biased. Zelig has two methods for case control correction:

  1. The "prior correction" method adjusts the intercept term. Let τ be the true population fraction of events, $\bar{y}$ the fraction of events in the sample, and $\hat{\beta}_0$ the uncorrected intercept term. The corrected intercept β_0 is:

$$\beta = \hat{\beta}_0 - \ln\left[\left(\frac{1-\tau}{\tau}\right)\left(\frac{\bar{y}}{1-\bar{y}}\right)\right].$$

  2. The "weighting" method performs a weighted logistic regression to correct for a case-control sampling design. Let the 1 subscript denote observations for which the dependent variable is observed as a 1, and the 0 subscript denote observations for which the dependent variable is observed as a 0. Then the vector of weights w_i

$$\begin{aligned} w_1 &= \frac{\tau}{\bar{y}} \\ w_0 &= \frac{(1-\tau)}{(1-\bar{y})} \\ w_i &= w_1 Y_i + w_0(1-Y_i) \end{aligned}$$

  If τ is unknown, you may alternatively specify an upper and lower bound for the possible range of τ. In this case, the `relogit` procedure uses "robust Bayesian" methods to generate a confidence interval (rather than a point estimate) for each quantity of interest. The nominal coverage of the confidence interval is at least as great as the actual coverage.

- By default, estimates of the the coefficients β are bias-corrected to account for finite sample or rare events bias. In addition, quantities of interest, such as predicted probabilities, are also corrected of rare-events bias. If $\widehat{\beta}$ are the uncorrected logit coefficients and bias$(\widehat{\beta})$ is the bias term, the corrected coefficients $\tilde{\beta}$ are

$$\widehat{\beta} - \text{bias}(\widehat{\beta}) = \tilde{\beta}$$

The bias term is

$$\text{bias}(\widehat{\beta}) = (X'WX)^{-1}X'W\xi$$

where

$$\begin{aligned} \xi_i &= 0.5Q_{ii}\Big((1+w-1)\widehat{\pi}_i - w_1\Big) \\ Q &= X(X'WX)^{-1}X' \\ W &= \text{diag}\{\widehat{\pi}_i(1-\widehat{\pi}_i)w_i\} \end{aligned}$$

where w_i and w_1 are given in the "weighting" section above.

Quantities of Interest

- For either one or no τ:
  - The expected values (`qi$ev`) for the rare events logit are simulations of the predicted probability

$$E(Y) = \pi_i = \frac{1}{1+\exp(-x_i\beta)},$$

    given draws of β from its posterior.
  - The predicted value (`qi$pr`) is a draw from a binomial distribution with mean equal to the simulated π_i.
  - The first difference (`qi$fd`) is defined as

$$\text{FD} = \Pr(Y=1 \mid x_1, \tau) - \Pr(Y=1 \mid x, \tau).$$

  - The risk ratio (`qi$rr`) is defined as

$$\text{RR} = \Pr(Y=1 \mid x_1, \tau) \;/\; \Pr(Y=1 \mid x, \tau).$$

- For a range of τ defined by $[\tau_1, \tau_2]$, each of the quantities of interest are $n \times 2$ matrices, which report the lower and upper bounds, respectively, for a confidence interval with nominal coverage at least as great as the actual coverage. At worst, these bounds are conservative estimates for the likely range for each quantity of interest. Please refer to King and Zeng (2002) for the specific method of calculating bounded quantities of interest.

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - E[Y_i(t_i = 0)] \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "relogit", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `bias.correct`: `TRUE` if bias correction was selected, else `FALSE`.
  - `prior.correct`: `TRUE` if prior correction was selected, else `FALSE`.
  - `weighting`: `TRUE` if weighting was selected, else `FALSE`.
  - `tau`: the value of `tau` for which case control correction was implemented.
  - `residuals`: the working residuals in the final iteration of the IWLS fit.
  - `fitted.values`: the vector of fitted values for the systemic component, π_i.
  - `linear.predictors`: the vector of $x_i\beta$

- `aic`: Akaike's Information Criterion (minus twice the maximized log-likelihood plus twice the number of coefficients).
- `df.residual`: the residual degrees of freedom.
- `df.null`: the residual degrees of freedom for the null model.
- `zelig.data`: the input data frame if `save.data = TRUE`.

Note that for a range of τ, each of the above items may be extracted from the `"lower.estimate"` and `"upper.estimate"` objects in your `zelig` output. Use `lower <- z.out$lower.estimate`, and then `lower$coefficients` to extract the coefficients for the empirical estimate generated for the smaller of the two τ.

- From `summary(z.out)`, you may extract:
  - `coefficients`: the parameter estimates with their associated standard errors, p-values, and t-statistics.
  - `cov.scaled`: a $k \times k$ matrix of scaled covariances.
  - `cov.unscaled`: a $k \times k$ matrix of unscaled covariances.
- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:
  - `qi$ev`: the simulated expected values, or predicted probabilities, for the specified values of `x`.
  - `qi$pr`: the simulated predicted values drawn from Binomial distributions given the predicted probabilities.
  - `qi$fd`: the simulated first difference in the predicted probabilities for the values specified in `x` and `x1`.
  - `qi$rr`: the simulated risk ratio for the predicted probabilities simulated from `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
  - `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Differences with Stata Version

The Stata version of ReLogit and the R implementation differ slightly in their coefficient estimates due to differences in the matrix inversion routines implemented in R and Stata. Zelig uses orthogonal-triangular decomposition (through `lm.influence()`) to compute the bias term, which is more numerically stable than standard matrix calculations.

Contributors

Please cite the rare events logit model as:

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

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

— (2002a), "Estimating Risk and Rate Levels, Ratios, and Differences in Case-Control Studies," *Statistics in Medicine*, 21, 1409–1427, http://gking.harvard.edu/files/abs/1s-abs.shtml.

Sample data are a selection of observations from

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

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.30 `tobit`: Linear Regression for a Left-Censored Dependent Variable

Tobit regression estimates a linear regression model for a left-censored dependent variable, where the dependent variable is censored from below. While the classical tobit model has values censored at 0, you may select another censoring point. For other linear regression models with fully observed dependent variables, see Bayesian regression (Section 12.21), maximum likelihood normal regression (Section 12.20), or least squares (Section 12.16).

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, below = 0, above = Inf,
                  model = "tobit", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Inputs

`zelig()` accepts the following arguments to specify how the dependent variable is censored.

- `below`: (defaults to 0) The point at which the dependent variable is censored from below. If any values in the dependent variable are observed to be less than the censoring point, it is assumed that that particular observation is censored from below at the observed value. (See Section 12.31 for a Bayesian implementation that supports both left and right censoring.)

- `robust`: defaults to `FALSE`. If `TRUE`, `zelig()` computes robust standard errors based on sandwich estimators (see Huber (1981) and White (1980)) and the options selected in `cluster`.

- `cluster`: if `robust = TRUE`, you may select a variable to define groups of correlated observations. Let `x3` be a variable that consists of either discrete numeric values, character strings, or factors that define strata. Then

  ```
  > z.out <- zelig(y ~ x1 + x2, robust = TRUE, cluster = "x3",
                   model = "tobit", data = mydata)
  ```

  means that the observations can be correlated within the strata defined by the variable `x3`, and that robust standard errors should be calculated according to those clusters. If `robust = TRUE` but `cluster` is not specified, `zelig()` assumes that each observation falls into its own cluster.

Zelig users may wish to refer to `help(survreg)` for more information.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(tobin)
   ```

   Estimating linear regression using `tobit`:

   ```
   > z.out < - zelig(durable ~ age + quant, model = "tobit", data = tobin)
   ```

   Setting values for the explanatory variables to their sample averages:

   ```
   > x.out <- setx(z.out)
   ```

   Simulating quantities of interest from the posterior distribution given `x.out`.

   ```
   > s.out1 <- sim(z.out, x = x.out)
   > summary(s.out1)
   ```

2. Simulating First Differences
   Set explanatory variables to their default(mean/mode) values, with high (80th percentile) and low (20th percentile) liquidity ratio (`quant`):

   ```
   > x.high <- setx(z.out, quant = quantile(tobin$quant, prob = 0.8))
   > x.low <- setx(z.out, quant = quantile(tobin$quant, prob = 0.2))
   ```

   Estimating the first difference for the effect of high versus low liquidity ratio on duration(`durable`):

   ```
   > s.out2 <- sim(z.out, x = x.high, x1 = x.low)
   > summary(s.out2)
   ```

Model

- Let Y_i^* be a latent dependent variable which is distributed with *stochastic* component

$$Y_i^* \sim \text{Normal}(\mu_i, \sigma^2)$$

  where μ_i is a vector means and σ^2 is a scalar variance parameter. Y_i^* is not directly observed, however. Rather we observed Y_i which is defined as:

$$Y_i = \begin{cases} Y_i^* & \text{if} \quad c < Y_i^* \\ c & \text{if} \quad c \geq Y_i^* \end{cases}$$

  where c is the lower bound below which Y_i^* is censored.

- The *systematic component* is given by

$$\mu_i \quad = \quad x_i\beta,$$

where x_i is the vector of k explanatory variables for observation i and β is the vector of coefficients.

Quantities of Interest

- The expected values (`qi$ev`) for the tobit regression model are the same as the expected value of $Y*$:

$$E(Y^*|X) = \mu_i = x_i\beta$$

- The first difference (`qi$fd`) for the tobit regression model is defined as

$$\text{FD} = E(Y^* \mid x_1) - E(Y^* \mid x).$$

- In conditional prediction models, the average expected treatment effect (`qi$att.ev`) for the treatment group is

$$\frac{1}{\sum t_i} \sum_{i:t_i=1} [E[Y_i^*(t_i = 1)] - E[Y_i^*(t_i = 0)]],$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(y ~ x, model = "tobit.bayes", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated parameters. The first k columns contain the posterior draws of the coefficients β, and the last column contains the posterior draws of the variance σ^2.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
  - `seed`: the random seed used in the model.

- From the `sim()` output object `s.out`:
  - `qi$ev`: the simulated expected value for the specified values of `x`.
  - `qi$fd`: the simulated first difference in the expected values given the values specified in `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.

Contributors

The tobit function is part of the survival library by Terry Therneau, ported to R by Thomas Lumley. Advanced users may wish to refer to `help(survfit)` in the survival library and

Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

Sample data are from

King, G., Alt, J., Burns, N., and Laver, M. (1990), "A Unified Model of Cabinet Dissolution in Parliamentary Democracies," *American Journal of Political Science*, 34, 846–871, http://gking.harvard.edu/files/abs/coal-abs.shtml.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

12.31 tobit.bayes: Bayesian Linear Regression for a Censored Dependent Variable

Bayesian tobit regression estimates a linear regression model with a censored dependent variable using a Gibbs sampler. The dependent variable may be censored from below and/or from above. For other linear regression models with fully observed dependent variables, see Bayesian regression (Section 12.21), maximum likelihood normal regression (Section 12.20), or least squares (Section 12.16).

Syntax

```
> z.out <- zelig(Y ~ X1 + X2, below = 0, above = Inf,
                  model = "tobit.bayes", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Inputs

`zelig()` accepts the following arguments to specify how the dependent variable is censored.

- `below`: point at which the dependent variable is censored from below. If the dependent variable is only censored from above, set `below = -Inf`. The default value is 0.
- `above`: point at which the dependent variable is censored from above. If the dependent variable is only censored from below, set `above = Inf`. The default value is `Inf`.

Additional Inputs

Use the following arguments to monitor the convergence of the Markov chain:

- `burnin`: number of the initial MCMC iterations to be discarded (defaults to 1,000).
- `mcmc`: number of the MCMC iterations after burnin (defaults to 10,000).
- `thin`: thinning interval for the Markov chain. Only every `thin`-th draw from the Markov chain is kept. The value of `mcmc` must be divisible by this value. The default value is 1.
- `verbose`: defaults to `FALSE`. If `TRUE`, the progress of the sampler (every 10%) is printed to the screen.
- `seed`: seed for the random number generator. The default is `NA` which corresponds to a random seed of 12345.
- `beta.start`: starting values for the Markov chain, either a scalar or vector with length equal to the number of estimated coefficients. The default is `NA`, such that the least squares estimates are used as the starting values.

Use the following parameters to specify the model's priors:

- `b0`: prior mean for the coefficients, either a numeric vector or a scalar. If a scalar, that value will be the prior mean for all coefficients. The default is 0.
- `B0`: prior precision parameter for the coefficients, either a square matrix (with the dimensions equal to the number of the coefficients) or a scalar. If a scalar, that value times an identity matrix will be the prior precision parameter. The default is 0, which leads to an improper prior.
- `c0`: `c0/2` is the shape parameter for the Inverse Gamma prior on the variance of the disturbance terms.
- `d0`: `d0/2` is the scale parameter for the Inverse Gamma prior on the variance of the disturbance terms.

Zelig users may wish to refer to `help(MCMCtobit)` for more information.

Convergence

Users should verify that the Markov Chain converges to its stationary distribution. After running the `zelig()` function but before performing `setx()`, users may conduct the following convergence diagnostics tests:

- `geweke.diag(z.out$coefficients)`: The Geweke diagnostic tests the null hypothesis that the Markov chain is in the stationary distribution and produces z-statistics for each estimated parameter.
- `heidel.diag(z.out$coefficients)`: The Heidelberger-Welch diagnostic first tests the null hypothesis that the Markov Chain is in the stationary distribution and produces p-values for each estimated parameter. Calling `heidel.diag()` also produces output that indicates whether the mean of a marginal posterior distribution can be estimated with sufficient precision, assuming that the Markov Chain is in the stationary distribution.
- `raftery.diag(z.out$coefficients)`: The Raftery diagnostic indicates how long the Markov Chain should run before considering draws from the marginal posterior distributions sufficiently representative of the stationary distribution.

If there is evidence of non-convergence, adjust the values for `burnin` and `mcmc` and rerun `zelig()`.

Advanced users may wish to refer to `help(geweke.diag)`, `help(heidel.diag)`, and `help(raftery.diag)` for more information about these diagnostics.

Examples

1. Basic Example
   Attaching the sample dataset:

   ```
   > data(tobin)
   ```

   Estimating linear regression using `tobit.bayes`:

   ```
   > z.out < - zelig(durable ~ age + quant, model = "tobit.bayes",
                     data = tobin, verbose=TRUE)
   ```

   Checking for convergence before summarizing the estimates:

   ```
   > geweke.diag(z.out$coefficients)
   > heidel.diag(z.out$coefficients)
   > raftery.diag(z.out$coefficients)
   > summary(z.out)
   ```

   Setting values for the explanatory variables to their sample averages:

   ```
   > x.out <- setx(z.out)
   ```

   Simulating quantities of interest from the posterior distribution given `x.out`.

   ```
   > s.out1 <- sim(z.out, x = x.out)
   > summary(s.out1)
   ```

2. Simulating First Differences
   Set explanatory variables to their default(mean/mode) values, with high (80th percentile) and low (20th percentile) liquidity ratio (`quant`):

   ```
   > x.high <- setx(z.out, quant = quantile(tobin$quant, prob = 0.8))
   > x.low <- setx(z.out, quant = quantile(tobin$quant, prob = 0.2))
   ```

   Estimating the first difference for the effect of high versus low liquidity ratio on duration(`durable`):

   ```
   > s.out2 <- sim(z.out, x = x.high, x1 = x.low)
   > summary(s.out2)
   ```

Model

Let Y_i^* be the dependent variable which is not directly observed. Instead, we observe Y_i which is defined as following:

$$Y_i = \begin{cases} Y_i^* & \text{if} \quad c_1 < Y_i^* < c_2 \\ c_1 & \text{if} \quad c_1 \geq Y_i^* \\ c_2 & \text{if} \quad c_2 \leq Y_i^* \end{cases}$$

where c_1 is the lower bound below which Y_i^* is censored, and c_2 is the upper bound above which Y_i^* is censored.

- The *stochastic component* is given by

$$\epsilon_i \sim \text{Normal}(0, \sigma^2)$$

  where $\epsilon_i = Y_i^* - \mu_i$.

- The *systematic component* is given by

$$\mu_i = x_i \beta,$$

  where x_i is the vector of k explanatory variables for observation i and β is the vector of coefficients.

- The *semi-conjugate priors* for β and σ^2 are given by

$$\begin{aligned} \beta &\sim \text{Normal}_k\left(b_0, B_0^{-1}\right) \\ \sigma^2 &\sim \text{InverseGamma}\left(\frac{c_0}{2}, \frac{d_0}{2}\right) \end{aligned}$$

  where b_0 is the vector of means for the k explanatory variables, B_0 is the $k \times k$ precision matrix (the inverse of a variance-covariance matrix), and $c_0/2$ and $d_0/2$ are the shape and scale parameters for σ^2. Note that β and σ^2 are assumed *a priori* independent.

Quantities of Interest

- The expected values (`qi$ev`) for the tobit regression model is calculated as following. Let

$$\begin{aligned} \Phi_1 &= \Phi\left(\frac{(c_1 - x\beta)}{\sigma}\right) \\ \Phi_2 &= \Phi\left(\frac{(c_2 - x\beta)}{\sigma}\right) \\ \phi_1 &= \phi\left(\frac{(c_1 - x\beta)}{\sigma}\right) \\ \phi_2 &= \phi\left(\frac{(c_2 - x\beta)}{\sigma}\right) \end{aligned}$$

where $\Phi(\cdot)$ is the (cumulative) Normal density function and $\phi(\cdot)$ is the Normal probability density function of the standard normal distribution. Then the expected values are

$$\begin{aligned} E(Y|x) &= P(Y^* \leq c_1|x)c_1 + P(c_1 < Y^* < c_2|x)E(Y^* \mid c_1 < Y^* < c_2, x) + P(Y^* \geq c_2)c_2 \\ &= \Phi_1 c_1 + x\beta(\Phi_2 - \Phi_1) + \sigma(\phi_1 - \phi_2) + (1 - \Phi_2)c_2, \end{aligned}$$

- The first difference (`qi$fd`) for the tobit regression model is defined as

$$\text{FD} = E(Y \mid x_1) - E(Y \mid x).$$

- In conditional prediction models, the average expected treatment effect (`qi$att.ev`) for the treatment group is

$$\frac{1}{\sum t_i} \sum_{i:t_i=1} [Y_i(t_i = 1) - E[Y_i(t_i = 0)]],$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run:

```
z.out <- zelig(y ~ x, model = "tobit.bayes", data)
```

then you may examine the available information in `z.out` by using `names(z.out)`, see the draws from the posterior distribution of the `coefficients` by using `z.out$coefficients`, and view a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: draws from the posterior distributions of the estimated parameters. The first k columns contain the posterior draws of the coefficients β, and the last column contains the posterior draws of the variance σ^2.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
  - `seed`: the random seed used in the model.
- From the `sim()` output object `s.out`:
  - `qi$ev`: the simulated expected value for the specified values of `x`.
  - `qi$fd`: the simulated first difference in the expected values given the values specified in `x` and `x1`.
  - `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.

Contributors

Bayesian tobit regression function is part of the MCMCpack library by Andrew D. Martin and Kevin M. Quinn. If you use this model, please cite:

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

The convergence diagnostics are part of the CODA library by Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. These diagnostics should be cited as:

> Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*.

Sample data are adapted from

> Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

Ben Goodrich and Ying Lu enabled `tobit.bayes` to work with Zelig.

12.32 `weibull`: Weibull Regression for Duration Dependent Variables

Choose the Weibull regression model if the values in your dependent variable are duration observations. The Weibull model relaxes the exponential model's (see Section 12.6) assumption of constant hazard, and allows the hazard rate to increase or decrease monotonically with respect to elapsed time.

Syntax

```
> z.out <- zelig(Surv(Y, C) ~ X1 + X2, model = "weibull", data = mydata)
> x.out <- setx(z.out)
> s.out <- sim(z.out, x = x.out)
```

Weibull models require that the dependent variable be in the form `Surv(Y, C)`, where `Y` and `C` are vectors of length n. For each observation i in 1, ..., n, the value y_i is the duration (lifetime, for example), and the associated c_i is a binary variable such that $c_i = 1$ if the duration is not censored (*e.g.*, the subject dies during the study) or $c_i = 0$ if the duration is censored (*e.g.*, the subject is still alive at the end of the study). If c_i is omitted, all Y are assumed to be completed; that is, time defaults to 1 for all observations.

Input Values

In addition to the standard inputs, `zelig()` takes the following additional options for weibull regression:

- `robust`: defaults to `FALSE`. If `TRUE`, `zelig()` computes robust standard errors based on sandwich estimators (see Huber (1981) and White (1980)) based on the options in `cluster`.
- `cluster`: if `robust = TRUE`, you may select a variable to define groups of correlated observations. Let `x3` be a variable that consists of either discrete numeric values, character strings, or factors that define strata. Then

  ```
  > z.out <- zelig(y ~ x1 + x2, robust = TRUE, cluster = "x3",
                   model = "exp", data = mydata)
  ```

  means that the observations can be correlated within the strata defined by the variable `x3`, and that robust standard errors should be calculated according to those clusters. If `robust = TRUE` but `cluster` is not specified, `zelig()` assumes that each observation falls into its own cluster.

Example

Attach the sample data:

```
> data(coalition)
```

Estimate the model:

```
> z.out <- zelig(Surv(duration, ciep12) ~ fract + numst2, model = "weibull",
                 data = coalition)
```

View the regression output:

```
> summary(z.out)
```

Set the baseline values (with the ruling coalition in the minority) and the alternative values (with the ruling coalition in the majority) for X:

```
> x.low <- setx(z.out, numst2 = 0)
> x.high <- setx(z.out, numst2 = 1)
```

Simulate expected values (`qi$ev`) and first differences (`qi$fd`):

```
> s.out <- sim(z.out, x = x.low, x1 = x.high)
> summary(s.out)
> plot(s.out)
```

Model

Let Y_i^* be the survival time for observation i. This variable might be censored for some observations at a fixed time y_c such that the fully observed dependent variable, Y_i, is defined as

$$Y_i = \begin{cases} Y_i^* & \text{if } Y_i^* \leq y_c \\ y_c & \text{if } Y_i^* > y_c \end{cases}$$

- The *stochastic component* is described by the distribution of the partially observed variable Y^*. We assume Y_i^* follows the Weibull distribution whose density function is given by

  $$f(y_i^* \mid \lambda_i, \alpha) = \frac{\alpha}{\lambda_i^\alpha} y_i^{*\alpha-1} \exp\left\{-\left(\frac{y_i^*}{\lambda_i}\right)^\alpha\right\}$$

  for $y_i^* \geq 0$, the scale parameter $\lambda_i > 0$, and the shape parameter $\alpha > 0$. The mean of this distribution is $\lambda_i\Gamma(1+1/\alpha)$. When $\alpha = 1$, the distribution reduces to the exponential distribution (see Section 12.6). (Note that the output from `zelig()` parameterizes `scale`$= 1/\alpha$.)

  In addition, survival models like the Weibull have three additional properties. The hazard function $h(t)$ measures the probability of not surviving past time t given survival up to t. In general, the hazard function is equal to $f(t)/S(t)$ where the survival function

$S(t) = 1 - \int_0^t f(s)ds$ represents the fraction still surviving at time t. The cumulative hazard function $H(t)$ describes the probability of dying before time t. In general, $H(t) = \int_0^t h(s)ds = -\log S(t)$. In the case of the Weibull model,

$$\begin{aligned} h(t) &= \frac{\alpha}{\lambda_i^\alpha} t^{\alpha-1} \\ S(t) &= \exp\left\{-\left(\frac{t}{\lambda_i}\right)^\alpha\right\} \\ H(t) &= \left(\frac{t}{\lambda_i}\right)^\alpha \end{aligned}$$

For the Weibull model, the hazard function $h(t)$ can increase or decrease monotonically over time.

- The *systematic component* λ_i is modeled as

$$\lambda_i = \exp(x_i\beta),$$

where x_i is the vector of explanatory variables, and β is the vector of coefficients.

Quantities of Interest

- The expected values (`qi$ev`) for the Weibull model are simulations of the expected duration:

$$E(Y) = \lambda_i \, \Gamma(1 + \alpha^{-1}),$$

given draws of β and α from their sampling distributions.

- The predicted value (`qi$pr`) is drawn from a distribution defined by (λ_i, α).

- The first difference (`qi$fd`) in expected value is

$$\text{FD} = E(Y \mid x_1) - E(Y \mid x).$$

- In conditional prediction models, the average expected treatment effect (`att.ev`) for the treatment group is

$$\frac{1}{\sum_{i=1}^n t_i} \sum_{i:t_i=1}^n \left\{Y_i(t_i = 1) - E[Y_i(t_i = 0)]\right\},$$

where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. When $Y_i(t_i = 1)$ is censored rather than observed, we replace it with a simulation from the model given available knowledge of the censoring process. Variation in the simulations are due to uncertainty in simulating $E[Y_i(t_i = 0)]$, the counterfactual expected value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

- In conditional prediction models, the average predicted treatment effect (`att.pr`) for the treatment group is

$$\frac{1}{\sum_{i=1}^{n} t_i} \sum_{i:t_i=1}^{n} \left\{ Y_i(t_i = 1) - \widehat{Y_i(t_i = 0)} \right\},$$

  where t_i is a binary explanatory variable defining the treatment ($t_i = 1$) and control ($t_i = 0$) groups. When $Y_i(t_i = 1)$ is censored rather than observed, we replace it with a simulation from the model given available knowledge of the censoring process. Variation in the simulations are due to uncertainty in simulating $\widehat{Y_i(t_i = 0)}$, the counterfactual predicted value of Y_i for observations in the treatment group, under the assumption that everything stays the same except that the treatment indicator is switched to $t_i = 0$.

Output Values

The output of each Zelig command contains useful information which you may view. For example, if you run `z.out <- zelig(y ~ x, model = "weibull", data)`, then you may examine the available information in `z.out` by using `names(z.out)`, see the `coefficients` by using `z.out$coefficients`, and a default summary of information through `summary(z.out)`. Other elements available through the `$` operator are listed below.

- From the `zelig()` output object `z.out`, you may extract:
  - `coefficients`: parameter estimates for the explanatory variables.
  - `icoef`: parameter estimates for the intercept and "scale" parameter $1/\alpha$.
  - `var`: the variance-covariance matrix.
  - `loglik`: a vector containing the log-likelihood for the model and intercept only (respectively).
  - `linear.predictors`: a vector of the $x_i\beta$.
  - `df.residual`: the residual degrees of freedom.
  - `df.null`: the residual degrees of freedom for the null model.
  - `zelig.data`: the input data frame if `save.data = TRUE`.
- Most of this may be conveniently summarized using `summary(z.out)`. From `summary(z.out)`, you may additionally extract:
  - `table`: the parameter estimates with their associated standard errors, p-values, and t-statistics.
- From the `sim()` output object `s.out`, you may extract quantities of interest arranged as matrices indexed by simulation $\times$ `x`-observation (for more than one `x`-observation). Available quantities are:

- `qi$ev`: the simulated expected values for the specified values of `x`.
- `qi$pr`: the simulated predicted values drawn from a distribution defined by (λ_i, α).
- `qi$fd`: the simulated first differences between the simulated expected values for `x` and `x1`.
- `qi$att.ev`: the simulated average expected treatment effect for the treated from conditional prediction models.
- `qi$att.pr`: the simulated average predicted treatment effect for the treated from conditional prediction models.

Contributors

The Weibull model is part of the survival library by Terry Therneau, ported to R by Thomas Lumley. Advanced users may wish to refer to `help(survfit)` in the survival library, and

Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

Sample data are from

King, G., Alt, J., Burns, N., and Laver, M. (1990), "A Unified Model of Cabinet Dissolution in Parliamentary Democracies," *American Journal of Political Science*, 34, 846–871, http://gking.harvard.edu/files/abs/coal-abs.shtml.

Kosuke Imai, Gary King, and Olivia Lau added Zelig functionality.

Chapter 13

Commands for Programmers and Contributors

13.1 `describe`: Describe a model's systematic and stochastic parameters

Description

In order to use `parse.formula()`, `parse.par()`, and the `model.*.multiple()` commands, you must write a `describe.mymodel()` function where `mymodel` is the name of your modeling function. (Hence, if your function is called `normal.regression()`, you need to write a `describe.normal.regression()` function.) Note that `describe()` is *not* a generic function, but is called by `parse.formula(..., model = "mymodel")` using a combination of `paste()` and `exists()`. You will never need to call `describe.mymodel()` directly, since it will be called from `parse.formula()` as that function checks the user-input formula or list of formulas.

Syntax

```
describe.mymodel()
```

Arguments

The `describe.mymodel()` function takes no arguments.

Output Values

The `describe.mymodel()` function returns a list with the following information:

- `category`: a character string, consisting of one of the following:

- `"continuous"`: the dependent variable is continuous, numeric, and unbounded (e.g., normal regression), but may be censored with an associated censoring indicator (e.g., tobit regression).
- `"dichotomous"`: the dependent variable takes two discrete integer values, usually 0 and 1 (e.g., logistic regression).
- `"ordinal"`: the dependent variable is an ordered factor response, taking 3 or more discrete values which are arranged in an ascending or descending manner (e.g., ordered logistic regression).
- `"multinomial"`: the dependent variable is an unordered factor response, taking 3 or more discrete values which are arranged in no particular order (e.g., multinomial logistic regression).
- `"count"`: the dependent variable takes integer values greater than or equal to 0 (e.g., Poisson regression).
- `"bounded"`: the dependent variable is a continuous numeric variable, that is restricted (bounded within) some range (e.g., $(0, \infty)$). The variable may also be censored either on the left and/or right side, with an associated censoring indicator (e.g., Weibull regression).
- `"mixed"`: the dependent variables are a mix of the above categories (usually applies to multiple equation models).

Selecting the category is particularly important since it sets certain interface parameters for the entire GUI.

- `package`: (optional) a list with the following elements
  - `name`: a characters string with the name of the package containing the `mymodel()` function.
  - `version`: the minimum version number that works with Zelig.
  - `CRAN`: if the package is not hosted on CRAN mirrors, provide the URL here as a character string. You should be able to install your package from this URL using `name`, `version`, and `CRAN`:

    ```
    install.packages(name, repos = CRAN, installWithVers = TRUE)
    ```

    By default, `CRAN = "http://cran.us.r-project.org/"`.

- `parameters`: For each systematic and stochastic parameter (or set of parameters) in your model, you should create a list (named after the parameters as given in your model's notation, e.g., `mu`, `sigma`, `theta`, etc.; not literally `myparameter`) with the following information:

- `equations`: an integer number of equations for the parameter. For parameters that can take an undefined number of equations (for example in seemingly unrelated regression), use `c(2, Inf)` or `c(2, 999)` to indicate that the parameter can take a minimum of two equations up to a theoretically infinite number of equations.
- `tagsAllowed`: a logical value (`TRUE`/`FALSE`) specifying whether a given parameter allows constraints. If there is only one equation for a parameter (for example, `mu` for the normal regression model has `equations = 1`), then `tagsAllowed = FALSE` by default. If there are two or more equations for the parameter (for example, `mu` for the bivariate probit model has `equations = 2`), then `tagsAllowed = TRUE` by default.
- `depVar`: a logical value (`TRUE`/`FALSE`) specifying whether a parameter requires a corresponding dependent variable.
- `expVar`: a logical value (`TRUE`/`FALSE`) specifying whether a parameter allows explanatory variables. If `depVar = TRUE` and `expVar = TRUE`, we call the parameter a "systematic component" and `parse.formula()` will fail if formula(s) are not specified for this parameter. If `depVar = FALSE` and `expVar = TRUE`, the parameter is estimated as a scalar ancillary parameter, with default formula `~ 1`, if the user does not specify a formula explicitly. If `depVar = FALSE` and `expVar = FALSE`, the parameter can only be estimated as a scalar ancillary parameter.
- `specialFunction`: (optional) a character string giving the name of a function that appears on the left-hand side of the formula. Options include `"Surv"`, `"cbind"`, and `"as.factor"`.
- `varInSpecial`: (optional) a scalar or vector giving the number of variables taken by the `specialFunction`. For example, `Surv()` takes a minimum of 2 arguments, and a maximum of 4 arguments, which is represented as `c(2, 4)`.

If you have more than one parameter (or set of parameters) in your model, you will need to produce a `myparameter` list for each one. See examples below for details.

Examples

For a Normal regression model with mean `mu` and scalar variance parameter `sigma2`, the minimal `describe.*()` function is as follows:

```
describe.normal.regression <- function() {
  category <- "continuous"
  mu <- list(equations = 1,              # Systematic component
             tagsAllowed = FALSE,
             depVar = TRUE,
             expVar = TRUE)
  sigma2 <- list(equations = 1,          # Scalar ancillary parameter
```

```
                  tagsAllowed = FALSE,
                  depVar = FALSE,
                  expVar = FALSE)
  pars <- list(mu = mu, sigma2 = sigma2)
  model <- list(category = category, parameters = pars)
}
```

See Section 8.1.2 for full code to execute this model from scratch in R with Zelig.

Now consider a bivariate probit model with parameter vector `mu` and correlation parameter `rho` (which may or may not take explanatory variables). Since the bivariate probit function uses the `pmvnorm()` function from the mvtnorm library, we list this under `package`.

```
describe.bivariate.probit <- function() {
  category <- "dichotomous"
  package <- list(name = "mvtnorm",
                  version = "0.7")
  mu <- list(equations = 2,                # Systematic component
             tagsAllowed = TRUE,
             depVar = TRUE,
             expVar = TRUE)
  rho <- list(equations = 1,               # Optional systematic component
             tagsAllowed = FALSE,          #   Estimated as an ancillary
             depVar = FALSE,               #   parameter by default
             expVar = TRUE)
  pars <- list(mu = mu, rho = rho)
  list(category = category, package = package, parameters = pars)
}
```

See Section 8.1.3 for the full code to write this model from scratch in R with Zelig.

For a multinomial logit model, which takes an undefined number of equations (corresponding to each level in the response variable):

```
describe.multinomial.logit <- function() {
  category <- "multinomial"
  mu <- list(equations = c(1, Inf),
             tagsAllowed = TRUE,
             depVAR = TRUE,
             expVar = TRUE,
             specialFunction <- "as.factor",
             varInSpecial <- c(1, 1))
  list(category = category, parameters = list(mu = mu))
}
```

(This example does not have corresponding executable sample code.)

See Also

- Section 8 for an overview of how the `describe.*()` function works with `parse.formula()`.
- Section 13.5 for information on `parse.formula()`.

Contributors

Kosuke Imai, Gary King, Olivia Lau, and Ferdinand Alimadhi.

13.2 `model.end`: Cleaning up after optimization

Description

The `model.end()` function creates a list of regression output from `optim()` output. The list includes coefficients (from the `optim() par` output), a variance-covariance matrix (from the `optim()` Hessian output), and any terms, contrasts, or xlevels (from the model frame). Use `model.end()` after calling `optim()`, but before assigning a class to the regression output.

Syntax

```
model.end(res, mf)
```

Arguments

- `res`: the output from `optim()` or another fitting-algorithm.
- `mf`: the model frame output by `model.frame()`.

Output Values

A list of regression output, including:

- `coefficients`: the optimized parameters.
- `variance`: the variance-covariance matrix (the negative inverse of the Hessian matrix returned from the optimization procedure).
- `terms`: the terms object. See `help(terms.object)` for more information.
- `...`: additional elements passed from `res`.

See Also

- Section 8 for an overview of how to write a new model.

Contributors

Kosuke Imai, Gary King, Olivia Lau, and Ferdinand Alimadhi.

13.3 model.frame.multiple: Extracting the "environment" of a model formula

Description

Use model.frame.multiple() after parse.par() to create a data frame of the unique variables identified in the formula (or list of formulas).

Syntax

```
model.frame.multiple(formula, data, eqn = NULL, ...)
```

Arguments

- formula: a list of formulas of class "multiple", returned from parse.par().
- data: a data frame containing all the variables used in formula.
- eqn: an optional character string or vector of character strings specifying the equations for which you would like to extract variables. Defaults to NULL, which pulls out all the variables for all equations in formula.
- ...: additional arguments passed to model.frame.default().

Output Values

The output is a data frame (with a terms attribute) containing all the unique explanatory and response variables identified in the list of formulas. By default, missing (NA) values are listwise deleted.

If as.factor() appears on the left-hand side, the response variables will be returned as an indicator (0/1) matrix with columns corresponding to the unique levels in the factor variable.

If any formula contains a tag(), model.frame.multiple() will return the original variable in the data frame and use the tag() information in the terms attribute only.

Examples

```
formulae <- list(import ~ coop + cost + target,
                 export ~ coop + cost + target)
fml <- parse.formula(formulae, model = "bivariate.logit")
D <- model.frame(fml, data = mydata)
```

Since the output from parse.formula() is of class "multiple", you do not need to call model.frame.multiple() explicitly, but can use the generic model.frame() instead.

See Also

- Section 13.5 for `parse.formula()`
- Section 8.1 for an overview of the user-interface.

Contributors

Kosuke Imai, Gary King, Olivia Lau, and Ferdinand Alimadhi.

13.4 `model.matrix.multiple`: Design matrix for multivariate models

Description

Use `model.matrix.multiple()` after `parse.formula()` to create a design matrix for multiple-equation models.

Syntax

```
model.matrix(object, data, shape = "compact", eqn = NULL, ...)
```

Arguments

- `object`: the list of formulas output from `parse.formula()`.
- `data`: a data frame created with `model.frame.multiple()`.
- `shape`: a character string specifying the shape of the outputed matrix. Available options are:
  - `"compact"`: (default) the output matrix will be an $n \times v$, where v is the number of unique variables in all of the equations (including the intercept term).
  - `"array"`: the output is an $n \times K \times J$ array where J is the total number of equations and K is the total number of parameters across all the equations. If a variable is not in a certain equation, it is observed as a vector of 0s.
  - `"stacked"`: the output will be a $2n \times K$ matrix where K is the total number of parameters across all the equations.
- `eqn`: a character string or a vector of character strings identifying the equations from which to construct the design matrix. The defaults to `NULL`, which only uses the systematic parameters (for which `DepVar = TRUE` in the appropriate `describe.model()`) .
- `...`: additional arguments passed to `model.matrix.default()`.

Output Values

A design matrix or array, depending on the options chosen in `shape`, with appropriate terms attributes.

Examples

Let's say that the name of the model is `"bivariate.probit"`, and the corresponding describe function is `describe.bivariate.probit()`, which identifies `mu1` and `mu2` as systematic components, and an ancillary parameter `rho`, which may be parameterized, but is estimated as a scalar by default. Let `par` be the parameter vector (including parameters for `rho`), `formulae` a user-specified formula given in one of the formats in Table 13.1, and `mydata` the user specified data frame.

Acceptable combinations of `parse.par()` and `model.matrix()` are as follows:

```
## Setting up
fml <- parse.formula(formulae, model = "bivariate.probit")
D <- model.frame(fml, data = mydata)
terms <- terms(D)

## Intuitive option
Beta <- parse.par(par, terms, shape = "vector", eqn = c("mu1", "mu2"))
X <- model.matrix(fml, data = D, shape = "stacked", eqn = c("mu1", "mu2")
eta <- X %*% Beta

## Memory-efficient (compact) option (default)
Beta <- parse.par(par, terms, eqn = c("mu1", "mu2"))
X <- model.matrix(fml, data = D, eqn = c("mu1", "mu2"))
eta <- X %*% Beta

## Computationally-efficient (array) option
Beta <- parse.par(par, terms, shape = "vector", eqn = c("mu1", "mu2"))
X <- model.matrix(fml, data = D, shape = "array", eqn = c("mu1", "mu2"))
eta <- apply(X, 3, '%*%', Beta)
```

In each case, `eta` is an $n \times 2$ matrix with columns corresponding to the linear predictors for `mu1` and `mu2`, respectively.

See Also

- Section 13.6 for selecting and shaping parameter vectors.
- Section 8 for examples of how to write new models.

Contributors

Kosuke Imai, Gary King, Olivia Lau, and Ferdinand Alimadhi.

13.5 parse.formula: Parsing the inputs

Description

Parse the input formula (or list of formulas) into the standard format described below. Since labels for this format will vary by model, `parse.formula()` will evaluate a function `describe.model()`, where `model` is given as an input to `parse.formula()`.

If the `check.model()` function has more than one parameter for which `ExpVar = TRUE` and `DepVar = TRUE`, then the user-specified equations must have labels to match those parameters, else `parse.formula()` should return an error. In addition, if the formula entries are not unambiguous, then `parse.formula()` should return an error.

Syntax

```
> fml <- parse.formula(formula, model, data = NULL)
```

Arguments

- `formula`: either a single formula or a list of formula objects.
- `model`: a character string specifying the name of the model.
- `data`: an optional data frame for models that require a factor response variable.

Output Values

The output is a list of formula objects with class `("multiple", "list")`. Let's say that the name of the model is `"bivariate.probit"`, and the corresponding describe function is `describe.bivariate.probit()`, which identifies `mu1` and `mu2` as systematic components, and an ancillary parameter `rho`, which may be parameterized, but is estimated as a scalar by default. Given this model, Table 13.1 gives acceptable user inputs.

Examples

```
formulae <- list(cbind(import, export) ~ coop + cost + target)
fml <- parse.formula(formulae, model = "bivariate.probit")
D <- model.frame(fml, data = mydata)
```

See Also

- Section 8.1 for commented examples of how `parse.formula()` and `describe.model()` work together.
- Section 13.9 for constraints between coefficients in a multiple equation context.

Contributors

Kosuke Imai, Gary King, Olivia Lau, and Ferdinand Alimadhi.

Table 13.1: Examples of acceptable short-hand for user-specified formulas, using bivariate probit as an example

	User Input	Output from `parse.formula()`

Same covariates, separate effects	`cbind(y1, y2) ~ x1 + x2 * x3`	`list(mu1 = y1 ~ x1 + x2 * x3,`
		`mu2 = y2 ~ x1 + x2 * x3,`
		`rho = ~ 1)`

With ρ as a systematic equation	`list(cbind(y1, y2) ~ x1 + x2,`
	`rho = ~ x4 + x5)`
		`list(mu1 = y1 ~ x1 + x2,`
		`mu2 = y2 ~ x1 + x2,`
		`rho = ~ x4 + x5)`

With constraints (same variable)	`list(mu1 = y1 ~ x1 + tag(x2, "x2"),`
	`mu2 = y2 ~ x3 + tag(x2, "x2"))`
		`list(mu1 = y1 ~ x1 + tag(x2, "x2"),`
		`mu2 = y2 ~ x3 + tag(x2, "x2"),`
		`rho = ~ 1)`

With constraints (different variables)	`list(mu1 = y1 ~ x1 + tag(x2, "z1"),`
	`mu2 = y2 ~ x3 + tag(x4, "z1"))`
		`list(mu1 = y1 ~ x1 + tag(x2, "z1"),`
		`mu2 = y2 ~ x3 + tag(x4, "z1"),`
		`rho = ~ 1)`

13.6 parse.par: Select and reshape parameter vectors

Description

The parse.par() function reshapes parameter vectors for compatability with the output matrix from model.matrix.multiple(). (SeeSection 13.4.) Use parse.par() to identify sets of parameters; for example, within optimization functions that require vector input, or within qi() functions that take matrix input of all parameters as a lump.

Syntax

```
parse.par(par, terms, shape = "matrix", eqn = NULL)
```

Arguments

- par: the vector (or matrix) of parameters.
- terms: the terms from either model.frame.*() or model.matrix.*().
- shape: a character string (either "matrix" or "vector") that identifies the type of output structure.
- eqn: a character string (or strings) that identify the parameters that you would like to subset from the larger par structure.

Output Values

A matrix or vector of the sub-setted (and reshaped) parameters for the specified parameters given in eqn. By default, eqn = NULL, such that all systematic components are selected. (Systematic components have ExpVar = TRUE in the appropriate describe.model() function.)

If an ancillary parameter (for which ExpVar = FALSE in describe.model()) is specified in eqn, it is always returned as a vector (ignoring shape). (Ancillary parameters are all parameters that have intercept only formulas.)

Examples

Let's say that the name of the model is "bivariate.probit", and the corresponding describe function is describe.bivariate.probit(), which identifies mu1 and mu2 as systematic components, and an ancillary parameter rho, which may be parameterized, but is estimated as a scalar by default. Let par be the parameter vector (including parameters for rho), formulae a user-specified formula given in one of the formats in Table 13.1, and mydata the user specified data frame.

Acceptable combinations of parse.par() and model.matrix() are as follows:

```
## Setting up
fml <- parse.formula(formulae, model = "bivariate.probit")
D <- model.frame(fml, data = mydata)
terms <- terms(D)

## Intuitive option
Beta <- parse.par(par, terms, shape = "vector", eqn = c("mu1", "mu2"))
X <- model.matrix(fml, data = D, shape = "stacked", eqn = c("mu1", "mu2")
eta <- X %*% Beta

## Memory-efficient (compact) option (default)
Beta <- parse.par(par, terms, eqn = c("mu1", "mu2"))
X <- model.matrix(fml, data = D, eqn = c("mu1", "mu2"))
eta <- X %*% Beta

## Computationally-efficient (array) option
Beta <- parse.par(par, terms, shape = "vector", eqn = c("mu1", "mu2"))
X <- model.matrix(fml, data = D, shape = "array", eqn = c("mu1", "mu2"))
eta <- apply(X, 3, '%*%', Beta)
```

In each case, `eta` is an $n \times 2$ matrix with columns corresponding to the linear predictors for `mu1` and `mu2`, respectively.

See Also

- Section 13.4 for a description of how multiple equation models work with `model.matrix()`.
- Section 8.2 for more detail on the three combinations (intuitive, memory-efficient, and computationally-efficient) methods of multiplying matrices of parameters and variables.

Contributors

Kosuke Imai, Gary King, Olivia Lau, and Ferdinand Alimadhi.

13.7 put.start: Set specific starting values for certain parameters

Description

After calling `set.start()` to create default starting values, use `put.start()` to change starting values for specific parameters or parameter sets.

Syntax

```
put.start(start.val, value, terms, eqn)
```

Arguments

- `start.val`: the vector of starting values created by `set.start()`.
- `value`: the scalar or vector of replacement starting values.
- `terms`: the terms output from `model.frame.multiple()`.
- `eqn`: the parameters for which you would like to replace the default values with `value`.

Output Values

A vector of starting values (of the same length as `start.val`).

See Also

- Section 13.8 to set default starting values.
- Section 8 for an overview of the procedure to add models to Zelig.

Contributors

Kosuke Imai, Gary King, Olivia Lau, and Ferdinand Alimadhi.

13.8 `set.start`: Set starting values for all parameters

Description

After using `parse.par()` and `model.matrix()`, use `set.start()` to set starting values for all parameters. By default, starting values are set to 0. If you wish to select alternative starting values for certain parameters, use `put.start()` after `set.start()`.

Syntax

```
set.start(start.val = NULL, terms)
```

Arguments

- `start.val`: user-specified starting values. If `NULL` (default), the default starting values for all parameters are set to 0.
- `terms`: the terms output from `model.frame.multiple()`.

Output Values

A named vector of starting values for all parameters specified in `terms`, defaulting to 0.

Example

```
fml <- parse.formula(formula, model = "bivariate.probit")
D <- model.frame(fml, data = data)
terms <- terms(D)
start.val <- set.start(start.val = NULL, terms)
```

See Also

- Section 13.7 to change starting values for specific parameter sets.
- Section 8 for detailed examples of writing new models.

Contributors

Kosuke Imai, Gary King, Olivia Lau and Ferdinand Alimadhi.

13.9 `tag`: Constrain parameter effects across equations

Description

Use `tag()` to identify parameters and constrain their effects across equations in multiple-equation models.

Syntax

```
tag(x, label)
```

Arguments

- `x`: the variable to be constrained.
- `label`: the name that the constrained variable takes.

Output Values

While there is no specific output from `tag()` itself, `parse.formula()` uses `tag()` to identify parameter constraints across equations, when a model takes more than one systematic component.

Examples

See Also

- Section 8.1 for an overview of the multiple-equation user-interface.
- Section 13.5 for more examples of acceptable uses for `tag()` in formulas.

Contributors

Kosuke Imai, Gary King, Olivia Lau, and Ferdinand Alimadhi.

Part IV

Appendices

Appendix A

Frequently Asked Questions

A.1 For All Zelig Users

How do I cite Zelig?

We would appreciate if you would cite Zelig as:

> Imai, Kosuke, Gary King and Olivia Lau. 2006. "Zelig: Everyone's Statistical Software," `http://GKing.Harvard.Edu/zelig`.

Please also cite the contributors for the models or methods you are using. These citations can be found in the contributors section of each model or command page.

Why can't I install Zelig?

You must be connected to the internet to install packages from web depositories. In addition, there are a few platform-specific reasons why you may have installation problems:

- **On Windows**: If you are using the very latest version of R, you may not be able to install Zelig until we update Zelig to work on the latest release of R. If you wish to install Zelig in the interim, check the Zelig release notes (Section B.1) and download the appropriate version of R to work with the last release of Zelig. You may have to manually download and install Zelig.

- **On Mac or Linux systems**: If you get the following warning message at the end of your installation:

  ```
  Installation of package VGAM had non-zero exit status in ...
  ```

  this means that you were not able to install VGAM properly. Make sure that you have the g77 Fortran compiler. For PowerPC Macs, download g77 from http://hpc.sourceforge.net). For Intel Macs, download the xcode Apple developer tools. After installation, try to install Zelig again.

Why can't I install R?

If you have problems installing R (rather than Zelig), you should check the R FAQs for your platform. If you still have problems, you can search the archives for the R help mailing list, or email the list directly at r-help@stat.math.ethz.ch.

Why can't I load data?

When you start R, you need to specify your working directory. In linux R, this is done pretty much automatically when you start R, whether within ESS or in a terminal window. In Windows R, you may wish to specify a working directory so that you may load data without typing in long directory paths to your data files, and it is important to remember that *Windows* R uses the *Linux* directory delimiter. That is, if you right click and select the "Properties" link on a Windows file, the slashes are backslashes (\), but Windows R uses forward slashes (/) in directory paths. Thus, the Windows link may be `C:\Program Files\R\R-2.3.1\`, but you would type `C:/Program Files/R/R-2.3.1/` in Windows R.

When you start R in Windows, the working directory is by default the directory in which the R executible is located.

```
# Print your current working directory.
> getwd()

# To read data not located in your working directory.
> data <- read.table("C:/Program Files/R/newwork/mydata.tab")

# To change your working directory.
> setwd("C:/Program Files/R/newwork")

# Reading data in your working directory.
> data <- read.data("mydata.tab")
```

Once you have set the working directory, you no longer need to type the entire directory path.

Where can I find old versions of Zelig?

For some replications, you may require older versions of Zelig.

- **Windows** users may find old binaries at `http://gking.harvard.edu/bin/windows/contrib/` and selecting the appropriate version of R.
- **Linux** and **MacOSX** users may find source files at `http://gking.harvard.edu/src/contrib/`

If you want an older version of Zelig because you are using an older version of R, we strongly suggest that you update R and install the latest version of Zelig.

Some Zelig functions don't work for me!

If this is a new phenomenon, there may be functions in your namespace that are overwriting Zelig functions. In particular, if you have a function called `zelig`, `setx`, or `sim` in your workspace, the corresponding functions in Zelig will not work. Rather than deleting things that you need, R will tell you the following when you load the Zelig library:

```
Attaching package: 'Zelig'
        The following object(s) are masked _by_ .GlobalEnv :
         sim
```

In this case, simply rename your `sim` function to something else and load Zelig again:

```
> mysim <- sim
> detach(package:Zelig)
> library(Zelig)
```

Who can I ask for help? How do I report bugs?

If you find a bug, or cannot figure something out, please follow these steps: (1) Reread the relevant section of the documentation. (2) Update Zelig if you don't have the current version. (3) Rerun the same code and see if the bug has been fixed. (4) Check our list of frequently asked questions. (5) Search or browse messages to find a discussion of your issue on the zelig listserv.

If none of these work, then if you haven't already, please (6) subscribe to the Zelig listserv and (7) send your question to the listserv at `zelig@latte.harvard.edu`. Please explain exactly what you did and include the full error message, including the `traceback()`. You should get an answer from the developers or another user in short order.

How do I increase the memory for R?

Windows users may get the error that R has run out of memory.

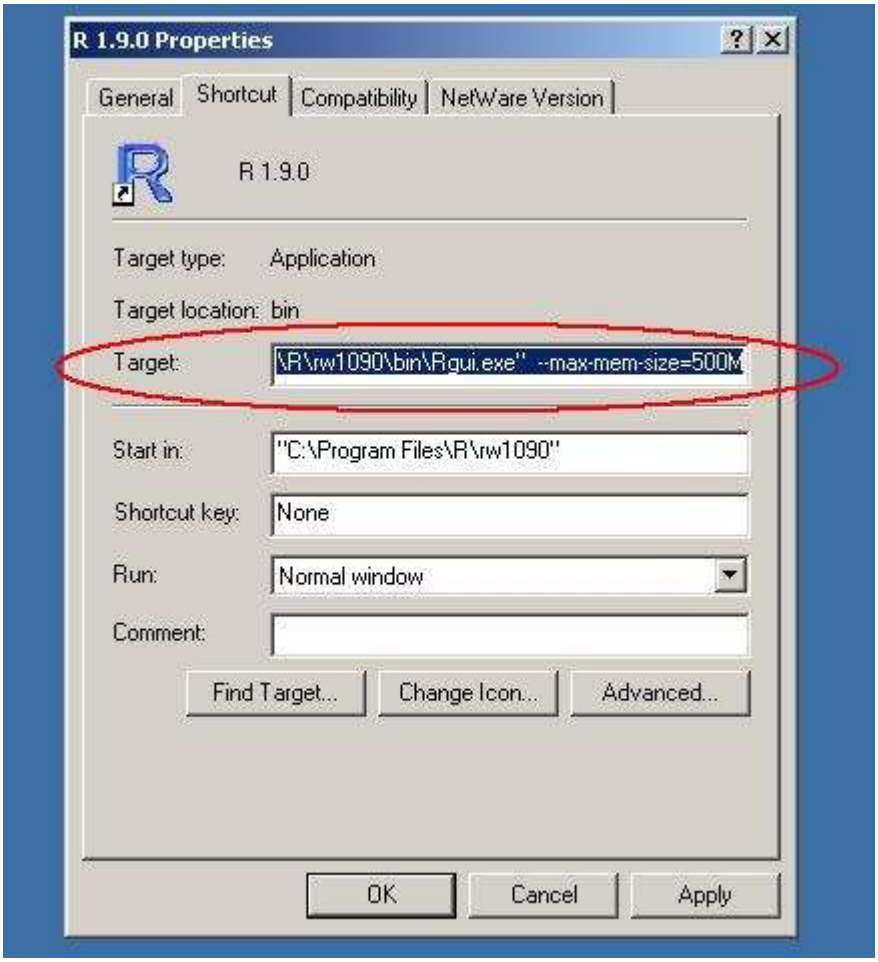
If you have R already installed and subsequently install more RAM, you may have to reinstall R in order to take advantage of the additional capacity.

You may also set the amount of available memory manually. Close R, then right-click on your R program icon (the icon on your desktop or in your programs directory). Select "Properties", and then select the "Shortcut" tab. Look for the "Target" field and after the closing quotes around the location of the R executible, add

```
--max-mem-size=500M
```

as shown in the figure below. You may increase this value up to 2GB or the maximum amount of physical RAM you have installed.

If you get the error that R cannot allocate a vector of length x, close out of R and add the following line to the "Target" field:




```
--max-vsize=500M
```

or as appropriate.

You can always check to see how much memory R has available by typing at the R prompt

```
> round(memory.limit()/2^20, 2)
```

which gives you the amount of available memory in MB.

Why doesn't the pdf print properly?

Zelig uses several special LaTeX environments. If the pdf looks right on the screen, there are two possible reasons why it's not printing properly:

- Adobe Acrobat isn't cleaning up the document. Updating to Acrobat Reader 6.0.1 or higher should solve this problem.

- Your printer doesn't support PostScript Type 3 fonts. Updating your print driver should take care of this problem.

R is neat. How can I find out more?

R is a collective project with contributors from all over the world. Their website (`http://www.r-project.o`
has more information on the R project, R packages, conferences, and other learning material.

In addition, there are several canonical references which you may wish to peruse:

Venables, W.N. and B.D. Ripley. 2002. *Modern Applied Statistics with S.* 4th Ed. Springer-Verlag.

Venables, W.N. and B.D. Ripley. 2000. *S Programming.* Springer-Verlag.

A.2 For Zelig Contributors

Where can I find the source code for Zelig?

Zelig is distributed under the GNU General Public License, Version 2. After installation, the source code is located in your R library directory. For Linux users who have followed our installation example, this is `~/.R/library/Zelig/`. For Windows users under R 2.3.1, this is by default `C:\Program Files\R\R-2.3.1\library\Zelig\`. For Macintosh users, this is `~/Library/R/library/Zelig/`.

In addition, you may download the latest Zelig source code as a tarball'ed directory from `http://gking.harvard.edu/src/contrib/`. (This makes it easier to distinguish functions which are run together during installation.)

How can I make my R programs run faster?

Unlike most commercial statistics programs which rely on precompiled and pre-packaged routines, R allows users to program functions and run them in the same environment. If you notice a perceptible lag when running your R code, you may improve the performance of your programs by taking the following steps:

- Reduce the number of loops. If it is absolutely necessary to run loops in loops, the inside loop should have the most number of cycles because it runs faster than the outside loop. Frequently, you can eliminate loops by using vectors rather than scalars. Most R functions deal with vectors in an efficient and mathematically intuitive manner.

- Do away with loops altogether. You can vectorize functions using the `apply`, `mapply()`, `sapply()`, `lapply()`, and `replicate()` functions. If you specify the function passed to the above `*apply()` functions properly, the R consensus is that they should run significantly faster than loops in general.

- You can compile your code using C or Fortran. R is not compiled, but can use bits of precompiled code in C or Fortran, and calls that code seamlessly from within R wrapper functions (which pass input from the R function to the C code and back to R). Thus, almost every regression package includes C or Fortran algorithms, which are locally compiled in the case of Linux systems or precompiled in the case of Windows distributions. The recommended Linux compilers are gcc for C and g77 for Fortran, so you should make sure that your code is compatible with those standards to achieve the widest possible distribution.

Which compilers can I use with R and Zelig?

In general, the C or Fortran algorithms in your package should compile for any platform. While Windows R packages are distributed as compiled binaries, Linux R compiles packages locally during installation. Thus, to ensure the widest possible audience for your package, you should make sure that your code will compile on gcc (for C and C++), or on g77 (for Fortran).

Appendix B

What's New? What's Next?

B.1 What's New: Zelig Release Notes

Releases listed as "stable releases" have been tested against prior versions of Zelig for consistency and accuracy. Testing distributions may contain bugs, but are usually replaced by stable releases within a few days.

- **2.7-4** (November 10, 2006): Stable release for R 2.4.0. Fixed bugs related to R check.
- **2.7-3** (November 9, 2006): Stable release for R 2.4.0. Fixed bugs related to R check.
- **2.7-2** (November 5, 2006): Stable release for R 2.4.0. Temporarily removed ARIMA models.
- **2.7-1** (November 3, 2006): Stable release for R 2.4.0. Made changes regarding the S4 classes in VGAM. The ARIMA (`arima`) model for time series data added by Justin Grimmer. First level dependencies are as follows:

MASS	7.2-29
boot	1.2-26
VGAM	0.7-1
MCMCpack	0.7-4
mvtnorm	0.7-5
survival	2.29
sandwich	2.0-0
zoo	1.2-1
coda	0.10-7

- **2.6-5** (September 14, 2006): Stable release for R 2.3.0-2.3.1. Fixed bugs in bivariate logit, bivariate probit, multinomial logit, and model.matrix.multiple (related to changes in version 2.6-4, but not previous versions, thanks to Chris Lawrence). First level

dependencies are as follows:

MASS	7.2-27.1
boot	1.2-26
VGAM	0.6-9
MCMCpack	0.7-1
mvtnorm	0.7-2
survival	2.28
sandwich	1.1-1
zoo	1.0-6
coda	0.10-5

- **2.6-4** (September 8, 2006): Stable release for R 2.3.0-2.3.1. Fixed bugs in `setx()`, and bugs related to `multiple` and the multinomial logit model. Added instructions for installing Fortran tools for Intel macs. Added the R×C ecological inference model. (thanks to Kurt Hornik, Luke Keele, Joerg Mueller-Scheessel, and B. Dan Wood)

- **2.6-3** (June 19, 2006): Stable release for R 2.0.0-2.3.1. Fixed bug in VDC interface functions, and `parse.formula()`. (thanks to Micah Altman, Christopher N. Lawrence, and Eric Kostello)

- **2.6-2** (June 7, 2006): Stable release for R 2.0.0-2.3.1. Removed R × C EI. Changed `data = list()` to `data = mi()` for multiply-imputed data frames. First level version compatibilities are as for version 2.6-1.

- **2.6-1** (April 29, 2006): Stable release for R 2.0.0-2.2.1. Fixed major bug in ordinal logit and ordinal probit expected value simulation procedure (does not affect Bayesian ordinal probit). (reported by Ian Yohai) Added the following ecological inference EI models: Bayesian hierarchical EI, Bayesian dynamic EI, and R × C EI. First level version compatibilities (at time of release) are as follows:

MASS	7.2-24
boot	1.2-24
VGAM	0.6-8
MCMCpack	0.7-1
mvtnorm	0.7-2
survival	2.24
sandwich	1.1-1
zoo	1.0-6
coda	0.10-5

- **2.5-4** (March 16, 2006): Stable release for R 2.0.0-2.2.1. Fixed bug related to windows build. First-level dependencies are the same as in version 2.5-1.

- **2.5-3** (March 9, 2006): Stable release for R 2.0.0-2.2.1. Fixed bugs related to VDC GUI. First level dependencies are the same as in version 2.5-1.

- **2.5-2** (February 3, 2006): Stable release for R 2.0.0-2.2.1. Fixed bugs related to VDC GUI. First level dependencies are the same as in version 2.5-1.

- **2.5-1** (January 31, 2006): Stable release for R 2.0.0-2.2.1. Added methods for multiple equation models. Added tobit regression. Fixed bugs related to robust estimation and upgrade of sandwich and zoo packages. Revised `setx()` to use environments. Added `current.packages()` to retrieve version of packages upon which Zelig depends. First level version compatibilities (at time of release) are as follows:

MASS	7.2-24
boot	1.2-24
VGAM	0.6-7
mvtnorm	0.7-2
survival	2.20
sandwich	1.1-0
zoo	1.0-4
MCMCpack	0.6-6
coda	0.10-3

- **2.4-7** (December 10, 2005): Stable release for R 2.0.0-2.2.2. Fixed the environment of `eval()` called within `setx.default()` (thanks to Micah Altman).

- **2.4-6** (October 27, 2005): Stable release for R 2.0.0-2.2.2. Fixed bug related to simulation for Bayesian Normal regression.

- **2.4-5** (October 18, 2005): Stable release for R 2.0.0-2.2.0. Fixed installation instructions.

- **2.4-4** (September 29, 2005): Stable release for R 2.0.0-2.2.0. Fixed `help.zelig()` links.

- **2.4-3** (September 29, 2005): Stable release for R 2.0.0-2.2.0. Revised `matchit()` documentation.

- **2.4-2** (August 30, 2005): Stable release for R 2.0.0-2.1.1. Fixed bug in `setx()` related to `as.factor()` and `I()`. Streamlined `qi.survreg()`.

- **2.4-1** (August 15, 2005): Stable release for R 2.0.0-2.1.1. Added the following Bayesian models: factor analysis, mixed factor analysis, ordinal factor analysis, unidimensional item response theory, k-dimensional item response theory, logit, multinomial logit, normal, ordinal probit, Poisson, and tobit. Also fixed minor bug in formula (long variable names coerced to list).

- **2.3-2** (August 5, 2005): Stable release for R 2.0.0-2.1.1. Fixed bug in simulation procedure for lognormal model.

- **2.3-1** (August 4, 2005): Stable release for R 2.0.0-2.1.1. Fixed documentation errors related to model parameterization and code bugs related to first differences and conditional prediction for exponential, lognormal, and Weibull models. (reported by Alison Post)

- **2.2-4** (July 30, 2005): Stable release for R 2.0.0-2.1.1. Revised relogit, adding option for weighting in addition to prior correction. (reported by Martin Plöderl)

- **2.2-3** (July 24, 2005): Stable release for R 2.0.0-2.1.1. Fixed bug associated with robust standard errors for negative binomial.

- **2.2-2** (July 13, 2005): Stable release for R 2.0.0-2.1.1. Fixed bug in setx(). (reported by Ying Lu)

- **2.2-1** (July 11, 2005): Stable release for R 2.0.0-2.1.0. Revised ordinal probit to use MASS library. Added robust standard errors for the following regression models: exponential, gamma, logit, lognormal, least squares, negative binomial, normal (Gaussian), poisson, probit, and weibull.

- **2.1-4** (May 22, 2005): Stable release for R 1.9.1-2.1.0. Revised help.zelig() to deal with CRAN build of Windows version. Added recode of slots to lists in NAMESPACE. Revised `install.R` script to deal with changes to `install.packages()`. (reported by Dan Powers and Ying Lu)

- **2.1-3** (May 9, 2005): Stable release for R 1.9.1-2.1.0. Revised param.lm() function to work with bootstrap simulation. (reported by Jens Hainmueller)

- **2.1-2** (April 14, 2005): Stable release for R 1.9.1-2.1.0. Revised summary.zelig().

- **2.1-1** (April 7, 2005): Stable release for R 1.9.1-2.1.0. Fixed bugs in NAMESPACE and summary.vglm().

- **2.0-14** (April 5, 2005): Stable release for R 1.9.1-2.0.1. Added `summary.vglm()` to ensure the compatibility with VGAM 0.6-2.

- **2.0-13** (March 11, 2005): Stable release for R 1.9.1-2.0.1. Fixed bugs in `NAMESPACE` and R-help file for `rocplot()`.

- **2.0-12** (February 20, 2005): Stable release for R 1.9.1-2.0.1. Added `plot = TRUE` option to `rocplot()`.

- **2.0-11** (January 14, 2005): Stable release for R 1.9.1-2.0.1. Changed class name for subsettted models from `"multiple"` to `"strata"`, and modified affected functions.

- **2.0-10** (January 5, 2005): Stable release for R 1.9.1 and R 2.0.0. Fixed bug in ordinal logit simulation procedure. (reported by Ian Yohai)

- **2.0-9** (October 21, 2004): Stable release for R 1.9.1 *and* R 2.0.0 (Linux and Windows). Fixed bug in NAMESPACE file.

- **2.0-8** (October 18, 2004): Stable release for R 1.9.1 *and* R 2.0.0 (Linux only). Revised for submission to CRAN.

- **2.0-7** (October 14, 2004): Stable release for R 1.9.1 *and* R 2.0.0 (Linux only). Fixed bugs in `summary.zelig()`, NAMESPACE, and assorted bugs related to new R release. Revised syntax for multiple equation models.

- **2.0-6** (October 4, 2004): Stable release for R 1.9.1. Fixed problem with NAMESPACE.

- **2.0-5** (September 25, 2004): Stable release for R 1.9.1. Changed installation procedure to source `install.R` from Zelig website.

- **2.0-4** (September 22, 2004): Stable release for R 1.9.1. Fixed typo in installation directions, implemented NAMESPACE, rationalized `summary.zelig()`, and tweaked documentation for least squares.

- **2.0-3** (September 1, 2004): Stable release for R 1.9.1. Fixed bug in conditional prediction for survival models.

- **2.0-2** (August 25, 2004): Stable release for R 1.9.1. Removed predicted values from `ls`.

- **2.0-1b** (July 16, 2004): Stable release for R 1.9.1. MD5 checksum problem fixed. Revised `plot.zelig()` command to be a generic function with methods assigned by the model. Revised entire architecture to accept multiply imputed data sets with strata. Added functions to simplify adding models. Completely restructured reference manual. Fixed bugs related to conditional prediction in setx and summarizing strata in summary.zelig.

- **1.1-2** (June 24, 2004): Stable release for R 1.9.1 (MD5 checksum problem not fixed, but does not seem to cause problems). Fixed bug in `help.zelig()`. (reported by Michael L. Levitan)

- **1.1-1** (June 14, 2004): Stable release for R 1.9.0. Revised `zelig()` procedure to use `zelig2model()` wrappers, revised `help.zelig()` to use a data file with extension `.url.tab`, and revised `setx()` procedure to take a list of `fn` to apply to variables, and such that `fn = NULL` returns the entire `model.matrix()`.

- **1.0-8** (May 27, 2004): Stable release for R 1.9.0. Fixed bug in simulation procedure for survival models. (reported by Elizabeth Stuart)

- **1.0-7** (May 26, 2004): Stable release for R 1.9.0. Fixed bug in relogit simulation procedure. (reported by Tom Vanwellingham)

- **1.0-6** (May 11, 2004): Stable release for R 1.9.0. Fixed bug in setx.default, which had previously failed to ignore extraneous variables in data frame. (reported by Steve Purpura)
- **1.0-5** (May 7, 2004): Replaced relogit procedure with memory-efficient version. (reported by Tom Vanwellingham)
- **1.0-4** (April 19, 2004): Stable release for R 1.9.0. Added vcov.lm method; changed print for summary.relogit.
- **1.0-2** (April 16, 2004): Testing distribution for R 1.9.0.
- **1.0-1** (March, 23, 2004): Stable release for R 1.8.1.

B.2 What's Next?

We have several plans for expanding and improving Zelig. Major changes slated for Version 3.0 (and beyond) include:

- Hierarchical and multi-level models
- Ecological inference models
- GEE models
- Neural network models
- Average treatment effects for everyone (treated and control units)
- Cross-validation
- Time-series cross-sectional models (via `nlme`)
- Generalized boosted regression model (via `gbm`)
- Saving random seeds to ensure exact replication

If you have suggestions, or packages that you would like to contribute to Zelig, please email our listserv at zelig@latte.harvard.edu

Bibliography

Adolph, C., Gary King, w. M. C. H., and Shotts, K. W. (2003), "A Consensus on Second Stage Analyses in Ecological Inference Models," *Political Analysis*, 11, 86–94, http://gking.harvard.edu/files/abs/akhs-abs.shtml.

Andrews, D. W. (1991), "Heteroskedasticity and Autocorrelation Consistent Covariance Matrix Estimation," *Econometrica*, 59, 817–858.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984), *Classification and Regression Trees*, New York, New York: Chapman & Hall.

Diamond, A. and Sekhon, J. (2005), "Genetic Matching for Estimating Causal Effects: A New Method of Achieving Balance in Observational Studies," http://jsekhon.fas.harvard.edu/.

Friendly, M. (2000), *Visualizing Categorical Data*, SAS Institute.

Gelman, A. and King, G. (1994), "A Unified Method of Evaluating Electoral Systems and Redistricting Plans," *American Journal of Political Science*, 38, 514–554, http://gking.harvard.edu/files/abs/writeit-abs.shtml.

Hansen, B. B. (2004), "Full Matching in an Observational Study of Coaching for the SAT," *Journal of the American Statistical Association*, 99, 609–618.

Hastie, T. J. and Tibshirani, R. (1990), *Generalized Additive Models*, London: Chapman Hall.

Ho, D., Imai, K., King, G., and Stuart, E. (2005), "Matching as Nonparametric Preprocessing for Parametric Causal Inference," Http://gking.harvard.edu/matchit/.

— (2006), "Matching as Nonparametric Preprocessing for Parametric Causal Inference," Http://gking.harvard.edu/files/abs/matchp-abs.shtml.

Huber, P. J. (1981), *Robust Statistics*, Wiley.

Imai, K. (2005), "Do Get-Out-The-Vote Calls Reduce Turnout? The Importance of Statistical Methods for Field Experiments," *American Political Science Review*, 99, 283–300.

Katz, J. and King, G. (1999), "A Statistical Model for Multiparty Electoral Data," *American Political Science Review*, 93, 15–32, http://gking.harvard.edu/files/abs/multiparty-abs.shtml.

King, G. (1989), *Unifying Political Methodology: The Likelihood Theory of Statistical Inference*, Michigan University Press.

— (1995), "Replication, Replication," *PS: Political Science and Politics*, 28, 443–499, http://gking.harvard.edu/files/abs/replication-abs.shtml.

— (1997), *A Solution to the Ecological Inference Problem: Reconstructing Individual Behavior from Aggregate Data*, Princeton: Princeton University Press.

King, G., Alt, J., Burns, N., and Laver, M. (1990), "A Unified Model of Cabinet Dissolution in Parliamentary Democracies," *American Journal of Political Science*, 34, 846–871, http://gking.harvard.edu/files/abs/coal-abs.shtml.

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

King, G., Murray, C. J., Salomon, J. A., and Tandon, A. (2004), "Enhancing the Validity and Cross-cultural Comparability of Measurement in Survey Research," *American Political Science Review*, 94, 191–205, http://gking.harvard.edu/files/abs/vign-abs.shtml.

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

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

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

— (2002a), "Estimating Risk and Rate Levels, Ratios, and Differences in Case-Control Studies," *Statistics in Medicine*, 21, 1409–1427, http://gking.harvard.edu/files/abs/1s-abs.shtml.

— (2002b), "Improving Forecasts of State Failure," *World Politics*, 53, 623–658, http://gking.harvard.edu/files/abs/civil-abs.shtml.

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

Lumley, T. and Heagerty, P. (1999), "Weighted Empirical Adaptive Variance Estimators for Correlated Data Regression," *jrssb*, 61, 459–477.

Martin, A. D. and Quinn, K. M. (2005), *MCMCpack: Markov chain Monte Carlo (MCMC) Package*.

Martin, L. (1992), *Coercive Cooperation: Explaining Multilateral Economic Sanctions*, Princeton University Press, please inquire with Lisa Martin before publishing results from these data, as this dataset includes errors that have since been corrected.

McCullagh, P. and Nelder, J. A. (1989), *Generalized Linear Models*, no. 37 in Monograph on Statistics and Applied Probability, Chapman & Hall, 2nd ed.

Plummer, M., Best, N., Cowles, K., and Vines, K. (2005), *coda: Output analysis and diagnostics for MCMC*.

Quinn, K. (2004), "Ecological Inference in the Presence of Temporal Dependence," in *Ecological Inference: New Methodological Strategies*, eds. King, G., Rosen, O., and Tanner, M. A., New York: Cambridge University Press.

Ripley, B. (1996), *Pattern Recognition and Neural Networks*, Cambridge Univeristy Press.

Rosen, O., Jiang, W., King, G., and Tanner, M. A. (2001), "Bayesian and Frequentist Inference for Ecological Inference: The $R \times C$ Case," *Statistica Neerlandica*, 55, 134–156, http://gking.harvard.edu/files/abs/rosen-abs.shtml.

Scheve, K. and Slaughter, M. (2001), "Labor Market Competition and Individual Preferences over Immigration Policy," *Review of Economics and Statistics*, 83, 133–145, sample data include only the first five of ten multiply imputed data sets.

Stoll, H., King, G., and Zeng, L. (2006), "WhatIf: Software for Evaluating Counterfactuals," *Journal of Statistical Software*, 15, http://gking.harvard.edu/whatif/.

Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Springer-Verlag, 4th ed.

White, H. (1980), "A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity," *Econometrica*, 48, 817–838.

Yee, T. W. and Hastie, T. J. (2003), "Reduced-rank vector generalized linear models," *Statistical Modelling*, 3, 15–41.

Zeileis, A. (2004), "Econometric Computing with HC and HAC Covariance Matrix Estimators," *Journal of Statistical Software*, 11, 1–17.