Zelig5 Documentation

Release 5.0

The Zelig Team

July 20, 2014

Zelig is a single, easy-to-use program that can estimate, help interpret, and present the results of a large range of statistical methods. It literally is "everyone's statistical software" because Zelig uses (R) code from many researchers. We also hope it will become "everyone's statistical software" for applications, and we have designed it so that anyone can use it or add their methods to it. Zelig comes with detailed, self-contained documentation that minimizes startup costs for Zelig and R (with all methods described in exactly the same notation, syntax, and style), automates graphics and summaries for all models, and, with only three simple required commands, makes the power of R accessible for all users. Zelig also works well for teaching, and is designed so that scholars can use the same program with students that they use for their research.

For more information about the goals and direction of the project, please see the *Technical Vision*.

To get started quickly, follow the *Quickstart*.

Visit the source repository: https://github.com/IQSS/Zelig5

Be sure to follow us on Twitter @IQSS!

CHAPTER

ONE

CONTENTS

1.1 Quickstart

1.1.1 TODO

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

1.2 Installation

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

1.3 User Guide

1.3.1 Introduction

What R and Zelig 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 , 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 program for Stata (and thus the suggestions of) 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 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). For missing data, Zelig accepts multiply imputed datasets created by (see) 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). 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.

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

[Rhelp]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 . We suggest you subscribe to this mailing list while learning and using Zelig: go to . (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 of previous messages before posting your query.

How to Cite Zelig

To cite Zelig as a whole, please reference these two sources:

> Kosuke Imai, Gary King, and Olivia Lau. 2007. "Zelig: Everyone's Statistical Software," http://GKing.harvard.edu/zelig.
>
> Imai, Kosuke, Gary King, and Olivia Lau. (2008). "Toward A Common Framework for Statistical Analysis and Development." Journal of Computational and Graphical Statistics, Vol. 17, No. 4 (December), pp. 892-913.

To refer to a particular Zelig model, please refer to the "how to cite" portion at the end of each model documentation section.

1.3.2 Data Analysis Commands

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).

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.

.

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 .
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.

Data Structures

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

1. A **variable** is a one-dimensional vector of length .
2. A **data frame** is a rectangular matrix with rows and columns. Each column represents a variable and each row an observation. Each variable may have a different class. (See 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 [a:R].

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.

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.

Variables

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 for more detail on logical operators. If var1 and var2 both have observations, commands such as

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

   create TRUE/FALSE observations such that the th observation in var3 evaluates whether the logical statement is true for the th value of var1 with respect to the th 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.

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 , 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 th observation in var will no longer match the th observation in data.

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

```
> 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 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 . 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 th value in var1 with a 1 when the th value in var2 equals the th 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 to transform a vector of unique values (with observations in the complete vector) into a matrix.

Missing Data To deal with missing values in some of your variables:

1. You may generate multiply imputed datasets using (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().

1.3.3 Statistical Commands

Zelig Commands

Quick Overview

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

1. Use `zelig()` to run the chosen statistical model on a given data set, with a specific set of variables. For standard likelihood models, for example, this step estimates the coefficients, other model parameters, and a variance-covariance matrix. In addition, you may choose from a variety of options:
   - Pre-process data: Prior to calling zelig(), you may choose from a variety of data pre-processing commands (matching or multiple imputation, for example) to make your statistical inferences more accurate.
   - Summarize model: After calling zelig(), you may summarize the fitted model output using summary().
   - Validate model: After calling zelig(), you may choose to validate the fitted model. This can be done, for example, by using cross-validation procedures and diagnostics tools.
2. Use `setx()` to set each of the explanatory variables to chosen (actual or counterfactual) values in preparation for calculating quantities of interest. After calling setx(), you may use to evaluate these choices by determining whether they involve interpolation (i.e., are inside the convex hull of the observed data) or extrapolation, as well as how far these counterfactuals are from the data. Counterfactuals chosen in `setx()` that involve extrapolation far from the data can generate considerably more model dependence (see , ,).
3. Use `sim()` to draw simulations of your quantity of interest (such as a predicted value, predicted probability, risk ratio, or first difference) from the model. (These simulations may be drawn using an asymptotic normal approximation (the default), bootstrapping, or other methods when available, such as directly from a Bayesian posterior.) After calling sim(), use any of the following to summarize the simulations:
   - The summary() function gives a numerical display. For multiple setx() values, summary() lets you summarize simulations by choosing one or a subset of observations.
   - If the setx() values consist of only one observation, plot() produces density plots for each quantity of interest.

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

Examples

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

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

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

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

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

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

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

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

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

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

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

```
> library(boot)
> data(turnout)
> z.out <- zelig(vote ~ race + educate, model = "logit", data = turnout, cite=F)
> cv.out <- cv.glm(z.out, data = turnout, k=11)
> print(cv.out$delta)
```

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 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 , 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) 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 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().)

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

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

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

   - You may perform the usual *unconditional* prediction (by default, cond = FALSE), by explicitly choosing the values of each explanatory variable yourself or letting setx() compute them, either from the data used to create z.out or from a new data set specified in the optional data argument. You may also compute predictions for all observed values of your explanatory variables using fn = NULL.

- Alternatively, for advanced uses, you may perform *conditional* prediction (cond = TRUE), which predicts certain quantities of interest by conditioning on the observed value of the dependent variable. In a simple linear regression model, this procedure is not particularly interesting, since the conditional prediction is merely the observed value of the dependent variable for that observation. However, conditional prediction is extremely useful for other models and methods, including the following:
  - In a matched sampling design, the sample average treatment effect for the treated can be estimated by computing the difference between the observed dependent variable for the treated group and their expected or predicted values of the dependent variable under no treatment .
  - With censored data, conditional prediction will ensure that all predicted values are greater than the censored observed values .
  - In ecological inference models, conditional prediction guarantees that the predicted values are on the tomography line and thus restricted to the known bounds .
  - The conditional prediction in many linear random effects (or Bayesian hierarchical) models is a weighted average of the unconditional prediction and the value of the dependent variable for that observation, with the weight being an estimable function of the accuracy of the unconditional prediction . 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 .

  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
```

(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 . 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. [1]

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), 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:

```
> 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.outqipr (for predicted values), s.outqiev (for expected values), and s.outqifd (for first differences in expected values). For the logit, probit, multinomial logit, ordinal logit, and ordinal probit models, quantities of interest also include s.outqirr (the risk ratio).

Describe a model's systematic and stochastic parameters

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.

```
describe.mymodel()
```

The describe.mymodel() function takes no arguments.

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).

[1] 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.

  - "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.,). 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.

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 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 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.)

- for an overview of how the describe.*() function works with parse.formula().

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

Supported Models

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

1. **Continuous Unbounded** dependent variables can take any real value in the range . 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) 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 and (conditional on the specified model).
   (b) "normal": The *Normal* (see) 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 is consistent but biased.
   (c) "normal.bayes": The *Bayesian Normal* regression model () is similar to maximum likelihood Gaussian regression, but makes valid small sample inferences via draws from the exact posterior and also allows for priors.
   (d) "netls": The *network least squares* regression () is similar to least squares regression for continuous-valued proximity matrix dependent variables. Proximity matrices are also known as sociomatrices, adjacency matrices, and matrix representations of directed graphs.
   (e) "tobit": The *tobit* regression model (see) is a Normal distribution with left-censored observations.
   (f) "tobit.bayes": The *Bayesian tobit* distribution (see) is a Normal distribution that has either left and/or right censored observations.
   (g) "arima": Use *auto-regressive, integrated, moving-average* (ARIMA) models for time series data (see .
   (h) "factor.bayes": The *Bayesian factor analysis* model (see) estimates multiple observed continuous dependent variables as a function of latent explanatory variables.
2. **Dichotomous** dependent variables consist of two discrete values, usually .
   (a) "logit": *Logistic regression* (see) specifies 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) 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) 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) Specifies 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) is similar to maximum likelihood probit regression, but makes valid small sample inferences via draws from the exact posterior and also allows for priors.
   (f) "netlogit": The *network logistic* regression () is similar to logistic regression for binary-valued proximity matrix dependent variables. Proximity matrices are also known as sociomatrices, adjacency matrices, and matrix representations of directed graphs.
   (g) "blogit": The *bivariate logistic* model (see) models for according to a bivariate logistic density.
   (h) "bprobit": The *bivariate probit* model (see) models for according to a bivariate normal density.
   (i) "irt1d": The *one-dimensional item response* model (see) takes multiple dichotomous dependent variables and models them as a function of *one* latent (unobserved) explanatory variable.

(j) "irtkd": The *k-dimensional item response* model (see) takes multiple dichotomous dependent variables and models them as a function of 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) specifies the stochastic component of the unobserved variable to be a standard logistic distribution.

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

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

   (b) "mlogit.bayes": *Bayesian multinomial logistic* regression (see) 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) 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, :math:'lambda = textrm{E}(Y_i|X_i)

   = textrm{V}(Y_i|X_i)'.

   (b) "poisson.bayes": *Bayesian Poisson* regression (see) 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) has the same systematic component as the Poisson, but allows event counts to be over-dispersed, such that .

6. **Continuous Bounded**[duration] dependent variables that are continuous only over a certain range, usually . 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) for positively-valued, continuous dependent variables that are fully observed (no censoring).

   (b) "exp": The *exponential* model (see) 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) for right-censored dependent variables relaxes the assumption of constant hazard by including an additional scale parameter : If , the risk of failure increases the longer the subject has survived; if , the risk of failure decreases the longer the subject has survived. While zelig() estimates by default, you may optionally fix at any value greater than 0. Fixing results in an exponential model.

(d) "lognorm": The *log-normal* model (see) 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) produces estimates for a cross-section of tables.

   (b) ei.dynamic: *Quinn's dynamic Bayesian ei* model (see) estimates a dynamic Bayesian model for tables with temporal dependence across tables.

   (c) ei.RxC: The ei model (see) estimates a hierarchical Multinomial-Dirichlet ei model for contingency tables with more than 2 rows or columns.

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). 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.

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.

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 .

Finally, you may use the identical() command to ensure that the replicated regression output is in every way identical to the original zelig() output. [2] For example:

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

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

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

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

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

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:

[2] 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.

1. If `X` is a variable of length , `plot(X)` returns a scatter plot of for . If X is unsorted, this procedure produces a messy graph. Use `plot(sort(X))` to arrange the plotted values of from smallest to largest.
2. With two numeric vectors `X` and `Y`, both of length , `plot(X, Y)` plots a scatter plot of each point for . 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:

  [!h]

"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:

  [!h]

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

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

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 , and a y-axis with values between , you may add a point at .

- **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 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 to the next. Alternatively, `lines` also takes command output which specifies coordinates. For example, `density(Z)` creates a vector of and a vector of , and plot(density(Z)) draws the kernel density function.
- **text()** adds a character string at the specified set of coordinates. For example, text(5, 5, labels = "Key Point") adds the label "Key Point" at the plot location . 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 coordinates. Type demo(vertci) to see an example for legend().

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.

Examples

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")
```

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.
```

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 for the sample code, type demo(mlogit) at the R prompt to run the example, and refer to Section [ternary] 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 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 [ROC] 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.

   [plot.vertci]

1.3.5 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 for loading data, or for a description of Zelig commands.

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.

Data Structures

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. [factors] A **factor vector** is a special type of vector that allows users to create indicator variables in one vector, rather than using 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), to set the values of the explanatory variables using setx (see) and in the ordinal logit and multinomial logit models (see).

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 = , ncol = )` to create a 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 : Use `cbind()` to combine vectors vertically to form a matrix, or `rbind()` to combine vectors horizontally to form a 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 by stacking matrices each with rows and 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
```

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
```

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. . . sourcecode:: r

> 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 .

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.

1.3.6 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.

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 th elements of the vector p and the th element of the vector q as a proportion of the th 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.

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.)

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 th value in the vector corresponding to the the th run of the loop). If your loop generates vector output, store them as rows (or columns) in a matrix, where the th row (or column) corresponds to the th 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 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.

```
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 indicators for 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 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.

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(x1) for each state , 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")
```

1.4 Developer Guide

1.4.1 Writing New Models

With Zelig, writing a new model in R is straightforward. (If you already have a model, see Chapter [c:addingmodels] 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.

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.

Describe the Statistical Model

After setting up the 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 , the model's parameters are simply .

There are very few models, however, that fall into this category. Even Normal regression, for example, has two sets of parameters and . 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 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 for examples and additional arguments output by describe.mymodel().)

Single Response Variable Models: Normal Regression Model

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

with scalar variance parameter , and systematic component . 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:

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 to the explanatory variables). 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 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.

Multivariate models: Bivariate Normal example

Most common models have one systematic component. For observations, the systematic component varies over observations . In the case of the Normal regression model, the systematic component is (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 observed as (0,0), (1,0), (0,1), or (1,1) for . Similar to a single-response probit model, the stochastic component is described by two latent (unobserved) continuous variables (,) which follow the bivariate Normal distribution:

where for , is the mean for and is a correlation parameter. The following observation mechanism links the observed dependent variables, , with these latent variables

The systemic components for each observation are

In the default specification, is a scalar (such that only contains an intercept term).

If so, we have two sets of parameters: and . 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 and (and sometimes), 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 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).
- Y is an matrix (where 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:

where I is an indicator function and

-
-
-
-

This implies the following log-likelihood:

(For the corresponding R code, see below.)

Easy Ways to Manage Matrices

Most statistical methods relate explanatory variables to a dependent variable of interest for each observation . Let be a set of parameters that correspond to each column in , which is an matrix with rows . For a single equation model, the linear predictor is

Thus, is the set of for and is usually represented as an matrix.

For a two equation model such as bivariate probit, the linear predictor becomes a matrix with columns corresponding to each dependent variable :

With as an 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 and 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 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 and 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 and matter.

The Intuitive Layout

A stacked matrix of and vector is probably the most visually intuitive configuration. Let be the number of equations in the bivariate probit model, and let be the total number of unique covariates in both equations. Choosing model.matrix(. . . , shape = "stacked") yields a matrix of explanatory variables. Again, let be an vector representing variable x1, x2, and so forth. Then

Correspondingly, is a vector with elements

where are the intercept terms for equation . Since is and is , the resulting linear predictor is also stacked into a matrix. Although difficult to manipulate (since observations are indexed by and for each rather than just), it is easy to see that we have turned the two equations into one big matrix and one long vector , which is directly analogous to the familiar single-equation .

The Computationally-Efficient Layout

Choosing array and vector is probably the the most computationally-efficient configuration: model.matrix(. . . , shape = "array") produces an array where is the total number of equations and is the total number of parameters across all the equations. Since some parameter values may be constrained across equations, . If a variable is not in a certain equation, it is observed as a vector of 0s. With this option, each matrix becomes:

By stacking each of these matrices along the first dimension, we get as an array with dimensions .

Correspondingly, is a vector with elements

To multiply the array with dimensions and the vector, we *vectorize* over equations as follows:

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

The linear predictor eta is therefore a matrix.

The Memory-Efficient Layout

Choosing a "compact" matrix and matrix is probably the most memory-efficient configuration: model.matrix(..., shape = "compact") (the default) produces an matrix, where is the number of unique variables (5 in this case) [3] in all of the equations. Let be an vector representing variable x1, x2, and so forth.

The parameter is used twice to implement the constraint, and the number of empty cells is minimized by implementing the constraints in rather than . Furthermore, since is and is , is .

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:

```
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)
```

[3] 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 , , , and .

```
  }
  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.

1.4.2 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 [s:new]), 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 [add]. Let mymodel be a new model with class "myclass".

[h!] [add]

(160,170)(0,0)

(0,166)Estimate

(70,162)(0,-1)42 (50,162)(40,12)zelig() (70,144)(0,-24)2(1,0)9 (80,138)(83,12)(1) zelig2mymodel() (80,114)(57,12)(2) mymodel()

(0,96)Interpret

(70,92)(0,-1)42 (50,92)(40,12)sim() (70,74)(0,-24)2(1,0)9 (80,68)(83,12)(3) param.myclass() (80,44)(69,12)(4) qi.myclass()

(0,26)Plot

(50,0)(105,12)(6) plot.zelig.mymodel()

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)
7. A function (describe.mymodel()) describing the inputs to your model, for use with a graphical user interface. (See).
8. An optional **demo script** mymodel.R which contains commented code for the models contained in the example section of your reference manual page.

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().

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 , 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.

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 preprocesses the zelig() output object and passes the appropriate arguments to setx.default().

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).

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 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
}
```

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 . (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.

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 [s:new], your model will be compatible with (at least) the GUI. For pre-existing models, you will need to create a describe.*() function for your model following the examples in .

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 .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 subsubsection 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.

```
\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}
```

1.5 Contributing

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

1.6 Frequently Asked Questions

1.6.1 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 () 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**: If the latest version of Zelig is not yet available at CRAN but you would like to install it on your Mac, try typing the following at your R prompt:

  ```
  install.packages("Zelig", repos = "http://gking.harvard.edu", type = "source")
  ```

- **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). For Intel Macs, download the 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 for your platform. If you still have problems, you can search the for the R help mailing list, or email the list directly at .

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 FilesR, but you would type C:/Program Files/R// 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 . (2) 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 . (5) to find a discussion of your issue on the zelig listserv.

If none of these work, then if you haven't already, please (6) and (7) send your question to the listserv at `zelig@lists.gking.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 () 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.

1.6.2 For Zelig Contributors

Where can I find the source code for Zelig?

Zelig is distributed under the . 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 , this is by default C:Program FilesRlibraryZelig. 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 . (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).

1.7 Release Notes

1.7.1 v 5.0 (Jul 2014)

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

CHAPTER

TWO

TECHNICAL VISION

Zelig adds considerable infrastructure to improve the use of existing methods. It generalizes the program Clarify (for Stata), which translates hard-to-interpret coefficients into quantities of interest; combines multiply imputed data sets (such as output from Amelia) to deal with missing data; automates bootstrapping for all models; uses sophisticated nonparametric matching commands which improve parametric procedures (via MatchIt); allows one-line commands to run analyses in all designated strata; automates the creation of replication data files so that you (or, if you wish, anyone else) can replicate the results of your analyses (hence satisfying the replication standard); makes it easy to evaluate counterfactuals (via WhatIf); and allows conditional population and superpopulation inferences. Zelig includes many specific methods, based on likelihood, frequentist, Bayesian, robust Bayesian, and nonparametric theories of inference. Developers make their R packages usable from Zelig by writing a few simple bridge functions.

CHAPTER

THREE

CONTACT

For questions, please join the Zelig mailing list: https://groups.google.com/forum/#!forum/zelig-statistical-software

CHAPTER

FOUR

LICENSE

GPL-2 | GPL-3 [expanded from: GPL (2)]