`mlelr`
A Reference Implementation of Logistic Regression in C

Scott A. Czepiel*

Abstract

This is the accompanying documentation for `mlelr`, a basic reference implementation of maximum likelihood estimation for logistic regression models using the C programming language.

Contents

*Updated versions of this document, additional resources, contact information, and source code are available at http://czep.net/stat/mlelr.html

1 Preface

In January, 2002 I began work at a market research firm specializing in online survey data for the film industry. My first task was to launch a new tracking product. We had 3 months to begin delivering results. The basic requirement involved fitting approximately 20,000 logistic regression models to estimate half a million predicted probabilities, and delivering the results in neatly formatted Excel files. The data were compiled every night at midnight Pacific time. Results were expected by 6:00 am Eastern time.

The first prototype I built in SAS ran in just under 17 hours. I had a powerful Xeon workstation with 512 MB of RAM and screaming[1] fast 15k SCSI disks. At this point the astute reader may suggest that the problem could easily be run in parallel, perhaps using a cluster of machines that could easily allocate partitions of the problem space across multiple

[1]The disks did indeed scream. SCSI disks are very audible during read activity and I was able to tune my program by simply listening to the disks while watching the screen, to know when it was time to optimize disk access. For future reference, don't put 20,000 sql calls in a for loop. Run one giant query and store the results in memory for the duration of the program. The program will run much faster, and your ears will appreciate it!

nodes. Bear in mind however that this era predates map-reduce by about 5 years, and the only cluster I had experience with at the time was of the Beowulf variety, and most of these had been toy projects of old PCs strung together in late night hacking sessions back in our grad school data lab. Needless to say, parallelization, while an oft-dreamed panacea, simply wasn't in the realistic realm of practical design. After some weeks of tuning I managed to reduce the run time to about 6 hours. That wasn't going to win us any clients.

With 6 weeks remaining until launch day, I made the arrogant recommendation that I simply write a custom C program to handle the logistic regressions, leaving the data management and reporting in SAS. In a fit of blind trust, my boss agreed and then I had to make good on my promise. It was the most intense 6 weeks of my professional career, but come April 1, the system launched on time, with only minor issues that were quickly ironed out in a point release 3 weeks later. That system remained in production for 3+ years before a rewrite by which time we had won the business of 7 major studios and effectively obliterated the competition. Over those first 3 years this tracking program generated somewhat north of $60 million in revenue.

The average runtime of the final program was around 5 minutes. While the impressive runtime was a great source of personal achievement, and made it possible to iterate, test, and deliver the results quickly, it was not the primary source of our competitive advantage. Our competition, the industry standard for over 20 years, relied on unadjusted unweighted tabulations of the survey data. This made the results highly susceptible to large variances at small sample sizes, particularly when one was interested in a narrow target segment of the population such as unmarried Latina women 20-29 with a college degree who are fans of Nicolas Cage. Tracking such small samples daily is far too noisy to make accurate generalizations. Our innovation was to model our categorical dependent variables (e.g. Awareness of a given film or interest on a 5-point scale to see the film). This provided stability to the estimates, enabling us to accurately report on trends in very narrow population segments. The proof that our methods worked was demonstrated by our ability to predict the opening weekend box office better than anyone else in the industry. Those efforts are worth their own discussion but that is another story which will have to wait until its time has come.

Since a great deal of money was being paid for these results, our attorneys wanted some third party validation that the system was really doing what it was designed to do and not simply spitting out various random numbers into the spreadsheets. After all, the whole thing hinged on the work of a sociology grad school dropout. My self-assuredness evidently wasn't a convincing enough selling point for the studio finance departments. So we hired a very expensive consulting statistician from NYU who reviewed every line of code I had written. He had a very antagonistic style, and justifiably so, you don't become a very expensive consulting statistician by being nice. And in market research, where statistical sins are committed en masse with reckless abandon, it behooves a consulting statistician to adopt a confrontational and exceedingly condescending manner. Despite this, he found no fault in my code. Still, he wouldn't be convinced until I wrote a publication worthy research paper thoroughly describing the theory of maximum likelihood estimation as it pertains to logistic regression along with the internals of the implementation of the Newton-Raphson iterative procedure that I had used in my program. The result of that exercise is my paper *Maximum Likelihood Estimation of Logistic Regression Models: Theory and Implementation* [1] which the attorneys allowed me to publish under my own name. Unfortunately, they

were also very clear that the code was not mine and would be kept under lock and key.

In the years since I wrote this paper I have received many requests for a fully functioning program to do logistic regression rather than the small snippets I use in the paper to illustrate the basics of the implementation. Since I didn't own that code I was unable to do so and I didn't feel comfortable working on a clone since it would inevitably resemble the original very closely. Now that it has been 12 years and the original program is no longer in production (and the original company I worked for has changed hands a few times), I feel confident that I can legally produce and distribute a simple open source implementation of logistic regression in C, primarily for teaching purposes if not also for the sake of posterity. That said, this work is a complete rewrite, no parts were reused except for those already in the public domain, the sources for which are all adequately documented in the code.

Over the past few years my recommendation to anyone asking for a C program to do logistic regression has been to use R. Everything `mlelr` does could be done in R with a few libraries and a handful of code, and it would also have the benefit of using an extremely robust and tested system which will no doubt be more reliable with edge cases than my (or nearly anyone's) hand rolled routines. So treat `mlelr` like a learning tool, but if you want production quality, use R. Or use SAS, SPSS, Sudaan, or another well respected statistical package. If you do want to develop your own logistic regression program, you will be well served by looking into the boost (C++) or gsl (C) libraries. While these libraries will not automatically provide you with all the features of R's glm() function, they will help with a lot of the complicated pieces that aren't really necessary to reinvent on your own. You could also look beyond C—Python in particular has a very active community around statistical programming, and the *scikit-learn* package in particular provides an excellent environment in which to conduct logistic regression as well as many other modern machine learning techniques.

2 Introduction

The primary purpose of `mlelr`[2] is to demonstrate the estimation of baseline-category logistic regression models by maximum likelihood using Newton-Raphson iteration in the C language using only standard libraries[3]. The code should compile cleanly on any POSIX-compliant GNU/Linux system.

Logistic regression is a highly useful modeling procedure that has evolved into the *de facto* standard approach to estimating predictive models for a categorical response variable. In my paper *Maximum Likelihood Estimation of Logistic Regression Models: Theory and Implementation* [1], I provide the mathematical background behind logistic regression and a brief outline of a computer program to solve the necessary equations. In the current work, I describe `mlelr`, a C program which illustrates the logistic regression estimation process as well as a basic user interface to a dataset structure. The complete source code

[2]I've taken to pronouncing the name as "mealer". Although its pneumonic quality is debatable, it is far easier than trying to say each letter separately in rapid succession. It is an endearing, if not enduring, nickname.

[3]In the current release version, we do require one non-standard library: the GNU Scientific Library (GSL), for the implementation of the gamma functions, which is a far better alternative than attempting to hand-write these difficult functions.

is available both on my website at `http://czep.net/stat/mlelr.html` as well as at my GitHub repository at `http://github.com/czep/mlelr`. The code is released under the open-source GNU GPL license.

Since the focus of `mlelr` is the estimation of logistic regression models, there is only minimal attention devoted to data management and user interface functionality. Features like variable labels and value labels, recoding, and filtering are not provided. Statistical methods like t-tests, anova, clustering, factor analysis, are also not implemented. Included is a basic facility for importing delimited text files, printing a dataset, producing univariate frequency tables, and of course, execution of a logistic regression equation based on a model specified by the user.

Again, since the goal with `mlelr` is to demonstrate logistic regression, there are many assumptions made in the code about how the procedure is to be carried out. The code is written specifically for this purpose, so to adapt the code to use, for example, a probit link function instead of the logit link may require substantial refactoring. By comparison, fully-featured statistical packages will include a generalized linear modeling procedure for which the user can choose from a variety of link functions. In addition, `mlelr` does not provide the ability to examine fitted values, residuals, ROC curves, split-sample validation, or any of a number of additional techniques that are essential to model fitting, diagnostics, and measurement of predictive efficacy.

The approach taken here in building the design matrix and estimating the regression coefficients most closely resembles that used by PROC CATMOD in SAS. Results obtained using the glm or vglm functions in R, PROC LOGISTIC or PROC GENMOD in SAS, and the LOGISTIC REGRESSION procedure in SPSS may differ depending on design coding of the parameters as well as the underlying algorithms used in the fitting process. It is important to understand the different assumptions involved in each program's approach before comparing them to results from `mlelr`.

Finally, I wish to emphasize what is already very clearly stated in the GPL license for this code: there is absolutely NO WARRANTY expressed, implied, or otherwise that accompanies this program. It has not been certified by any third-party, and its use constitutes an acknowledgment that it is entirely your own responsibility to certify the results for your own purposes. Attempts have been made to validate the results against test cases that I have personally selected which by no means provide complete coverage of all possible use cases.

3 Program Design

3.1 The Language Question

Programming language choice is a complex question that involves a little bit of logic and a little bit of emotion. It is actually rather rare to embark on a project with a completely unencumbered choice of which language to use. There are practical limitations that often constrain the choice of language to a small subset of what would otherwise be available if the programmer had free reign. The intended deployment environment may limit the available executable file formats or runtime interpreters necessary to run the program. The target audience of users may place a similar constraint on the chosen language, particularly

if installation of ancillary programs, additional third-party libraries or operating systems cannot easily be made into prerequisites. Legacy systems with which the program must interact, or required libraries that the program must use are also important factors that can influence language choice.

In an ideal world, the programmer will be able to use a language that is both well suited to the task at hand and with which the programmer has a suitable fluency in order to be productive and efficient. One method by which a programmer can be more efficient is by taking advantage of an existing stack of technologies that are highly evolved and readily available, thus greatly speeding up development time and reducing the programmer's time spent 'reinventing the wheel' or otherwise working on pieces of code that do what someone else has already done very well in an existing stable library. Business analysts are trained to focus their organization on core competencies and a programmer's approach to programming should be no different—concentrate on the code that truly adds value rather than the scaffolding and window-dressing that are necessary to encapsulate the program in a given environment. Choice of programming language is highly important because the right one can greatly speed development by quickly "getting out of the way" and letting the programmer focus on the project rather than the tool. In general, a high-level general purpose scripting language like Python will be ideally suited for rapid prototyping and implementation of programs with a low to moderate level of complexity. In the case of web server software, typical projects lend themselves to Ruby simply because the ecosystem surrounding the platform is modern and mature.

Even today, despite—or perhaps because of—its rich and storied history, its spartan and often dangerous feature set, the C programming language remains the most versatile choice for systems programming. C is a lot like Latin: an extremely old language that is the antecedent of many, more modern, languages—a language that many people can recognize but don't necessarily speak. C is easy to read because so many languages have adopted its syntax, but it is rarely spoken outside of limited, predominantly academic circles, especially since its more modern descendents, primarily C++ and Java, have become extremely popular. C is concise to the point of lucid terseness. The C standard library is very basic, paltry if compared to the 'batteries included' style of Python, whose library was deliberately designed to take the opposite philosophy of C. However, all the important stuff you read about in Knuth's *The Art of Computer Programming* [2] are more than adequately provided for in C. The programmer has complete control of the hardware—memory, the processor registers, I/O and other system calls, the bare metal. This power makes C highly flexible, but also highly precarious, a danger well known by anyone who has ever faced the dreaded `Segmentation fault` in their console.

While one could well argue that a reference implementation of a popular statistical procedure ought to be coded in one of C's more modern derivatives, as a learning exercise I find it far more illustrative to demonstrate the procedure using the C language for three reasons. First, higher level languages and libraries with advanced features and data structures can add multiple levels of indirection making it difficult to discern how the program is actually operating. The goal of `mlelr` is to connect the theory of multinomial logistic regression with its practical application. Shielding the programmer from the details of the algorithms involved can mask what is actually being accomplished. This can hinder the interested student's learning process. Second, using ANSI C and standard libraries helps

develop good programming discipline. It also ensures the maximum level of portability and adaptability of the program. This again highlights the fact that the goal is to demonstrate the functioning of the statistical model with a minimum of external requirements. Third, I just like C. Here's that little bit of emotion coming into play. Programming in C is a very gratifying experience, reserved for my personal projects where I can choose exactly what and how I want to code: no legacy systems, no third-party libraries, no organizational coding standards to abide by to distract from the pure creative process. C is just more fun: trust me!

In the following sections, I will outline some of the general guidelines I use when approaching a C program from scratch that have influenced the layout and design of `mlelr`. I am by no means a C expert, and some of the advice proffered here may well be ill-advised in certain circumstances, particularly large-scale systems. My programming experience is generally limited to small one-man projects, developing programs that solve simple well-defined problems. If you are involved in a very large project, then the level of discipline required may necessitate a far more structured approach than what I outline here. Many of these ideas are also not primarily of my own invention, as I am indebted to three wonderful books that I urge everyone to read, think about, and use to develop your own programming style. First, Kelley and Pohl's *A Book on C* [3]. While you could naturally start with Kernighan and Ritchie's *The C Programming Language* [4], and you certainly can't go wrong with the book written by the masters themselves, I find Kelley and Pohl's book to be a great resource for a student's introduction to C. Second, Kernighan and Pike's *The Practice of Programming* [5]. I make repeated use of their sample code and will note their contributions later in this text. Finally, *Expert C Programming: Deep C Secrets* [6] by Peter van der Linden. This is an excellent book that goes beyond the introductory material and provides a thorough and highly informative understanding of how C works. It's also a wonderfully entertaining read.

Keeping in mind the above caveats, the remainder of this section is structured around various pieces of advice for developing a program in C that will help explain some of the design decisions made in `mlelr`. You will likely come across some statements with which you disagree and I wholeheartedly encourage opposing viewpoints and healthy debate. The real purpose here is to stimulate thought so you can form your own programming style that works for you.

3.2 Statistical Package vs. Utility Program

The design of `mlelr` most closely resembles the philosophy behind command-line UNIX utilities: *do one thing, and do it well.* Although there is an interactive mode, you will probably not spend much time at the `mlelr` prompt. There is no code editor, no menu system, nor even a basic Readline environment to make command editing convenient. Quite frankly, the interactive mode is really only there for demonstration purposes. `mlelr` will be most effective when treated as a UNIX utility. Generate a dataset, generate a script file that `mlelr` can read from, run the program, then collect and parse the output in an environment more suitable for scripting.

A statistical package is designed to occupy the opposite end of the user interface spectrum, where the user might be expected to spend most of their time within the program.

While most stat packages do have their own batch modes, they typically shine by providing a fully featured interactive environment designed to guide the user in the exploration of data and the model building process. Our leaning towards the utility program philosophy necessitates certain design compromises. We provide only rudimentary editing and data exploration capabilities because we expect the user to have access to far better tools with which to accomplish those tasks. At the same time, by concentrating only on what is necessary to perform a logistic regression, the code has minimal distractions and can be readily understood, adapted, and customized for your own use and learning.

3.3 C Header Files

Each header file should be wrapped in a conditional macro statement to prevent compilation errors resulting from including the same header multiple times.

```
#ifndef DATASET_H__
#define DATASET_H__
...
/* declarations */
...
#endif
```

The purpose of a header file is to declare the public functions that the module makes available for other modules to call, as well as any structs or constants that logically make sense to organize within the module's control. Functions and variables you declare in a header file should be prefaced by the `extern` keyword, indicating that the actual definitions will be contained in a different file. This different file could be the accompanying "`.c`" file or even a completely different "`.c`" file. Note that `typedef` or `struct` declarations that *do not* declare a variable (and thus, do not require the compiler to reserve storage), do not need the `extern` keyword.

For example, in `mlelr`, the header file in which we declare our dataset structure and the various functions we define for operating on datasets is called `dataset.h`. In this header file, we declare a `typedef` for our dataset structure which is used throughout the program. For more detail, please see our discussion of Data Management in Section 4. Remember that in C, anything declared in a header file is automatically given global scope—there are no namespaces—so be extremely judicious when making declarations here. If a variable or function is really only going to be used within the module, don't declare it in the header file. Instead, declare it in the C source file with the `static` keyword.

3.4 Module Organization

Upon uncompressing `mlelr`, the 'src' directory will contain the files listed in Table 1 below.

Kernighan & Pike's csv library is available in their wonderful book *The Practice of Programming* [5]. We use the library for two purposes: 1) to parse and import delimited data files into a dataset, and 2) to parse user command input for translation into program flow statements.

Table 1: Source files

Filename	Description
csv.c csv.h	Kernighan & Pike's csv library implementation
dataset.c dataset.h	Dataset struct and associated functions
interface.c interface.h	User interface functions
main.c	Program entry point and initialization
Makefile	Makefile for use with `make`
mlelr.c mlelr.h	Logistic regression implementation
model.c model.h	Model struct and associated functions
tabulate.c tabulate.h	Support functions for tabulation

Our dataset module is introduced in the general purpose header file `dataset.h` which is imported by most other modules for access to our dataset structure. Here we also declare several public functions for management of datasets.

```
/* forward declarations for public functions defined in dataset.c */
extern void init_dataspace (void);
extern int import_dataset (char *handle, char *filename, char delim);
extern dataset *add_dataset (char *handle, int nvars, char **varnames,
                             int is_public );
extern void add_observation (dataset *ds, double *obs);
extern void print_dataset (dataset *ds, int n, int header);
extern dataset *find_dataset (char *handle);
extern int find_varname (dataset *ds, char *varname);
extern int set_weight_variable (dataset *ds, int var);
extern int find_observation(dataset *ds, double *obs, int n_vars);
extern void sort_dataset(dataset *ds, int n_cols);
```

The first function, `init_dataspace`, is called in `main.c` at program invocation after command line option processing to allocate memory for our global `dataspace` variable, which maintains a pointer to all available datasets. We initialize space for a pointer to one dataset. The `add_dataset` function is used to add new datasets to the dataspace and will reallocate space in the dataspace as necessary. `The import_dataset` function is described below in Section 5.4. This is the primary means of reading user data into a dataset for use in the program.

The `add_observation` function is used only by `import_dataset` but could conceivably be used if additional data management functionality allowed the user to add observations to a dataset after the initial import had been completed. `print_dataset` provides a means of

writing the contents of a dataset to the output. This is useful for sanity checking whether an import procedure worked as expected. `The find_dataset` function is used for convenience so that a user can refer to a dataset by its handle. This function will return a pointer to the first dataset that it finds with the supplied handle.

The `mlelr.c` and `mlelr.h` files contain our implementation of logistic regression.

The `main.c` file is our program's entry point and contains the `main()` function. Here, command line options are parsed and some basic initialization takes place.

The `interface.c` and `interface`.h files declare and define all functions for input, output, and interacting with the user.

3.5 Makefiles

`make` is a very powerful UNIX utility designed to make program compilation and linking easier. Our Makefile is rather spartan, there are only two targets. Running `make` will compile, link, and build the executable, which on GNU/Linux systems will be named `mlelr`. Running `make clean` will remove all object files, debugging, and profiling files, and restore the src directory to its original state. The complete contents of our Makefile are shown below:

```
CC=gcc
CFLAGS=-Wall -g -pg -lm -lgsl -lgslcblas

mlelr: main.o csv.o dataset.o model.o \
       interface.o tabulate.o mlelr.o
$(CC) -o mlelr main.o csv.o dataset.o model.o \
    interface.o tabulate.o mlelr.o $(CFLAGS)

clean:
rm -f mlelr gmon.out main.o csv.o dataset.o model.o \
    interface.o tabulate.o mlelr.o
```

It may be necessary to adjust some parameters, particularly if you do not have access to the GNU compiler collection, gcc. The -g (debugging output) and -pg (profiling output) options can be removed if you have no interest in the additional features provided by these flags. If you're really cool, you can add some optimizer settings, but you do so at your own risk and are entirely responsible for knowing how these optimizer settings can affect the stability of the program. As is, the resulting executable should be able to run on an i386 architecture, so if you have an old 386 machine siting in the attic somewhere, and a version of linux old enough to run on it, there is no reason our program will not be able to compile on this rusty yet trusty architecture.

3.6 Return Values and Function Parameters

In most situations, especially in cases where abnormal situations may be encountered which we wish to alert to the calling function in the form of a specific return value, we want our functions to return an `int` and use that return code to signal the overall result of the function's performance. The convention is to return 0 for success and some non-zero integer

to signal some kind of exception. The calling function can then handle or ignore the return value at its peril. Typically, we return 0 for success because there is only one way for a function to succeed, but many many ways in which it can fail!

When a function requires numerous parameters and will typically return numerous variables, what is the cleanest way to design such a function? Consider all the variables that a logistic regression function needs to know about: the dependent variable, array of independent variables, specification of direct effects and interaction terms—and will generate a great deal of output—the parameters of the fitted model, the Wald chi-square values and statistical significance tests of the parameters, the overall log likelihood and deviance of the model, the design matrix used, etc. In short, there is a great deal of information that needs to be passed into and out of this function. To simplify the code, we wrap everything we need in a pointer variable to a structure that contains both the required input arguments as well as the output variables that will be sent back to the calling routine when processing is complete. For our `mlelr` function, we define a special struct in `model.h` called `model`. Initial allocation of the arguments needed to run the model occur in the `cmd_logreg` function after syntax checks are completed (see Section 5.9 for details. Then, additional allocations are conducted inside the function to reserve space for all the various return values. Since we are passing this structure as a pointer, it can easily be read from and written to by both the calling and called functions.

```
typedef struct {
    char    *dvname;   /* name of the dependent variable */
    int     dv;        /* dependent variable index in dataset */
    int     numiv;     /* number of independent variables */
    int     maxiv;     /* space allocated for iv array */
    char    **ivnames; /* names of the independent variables */
    int     *iv;       /* indices of the independent variables (main effects only) */
    int     *direct;   /* for each iv, stores 1 if direct effect else 0 */
    int     numints;   /* number of interactions */
    int     maxints;   /* space allocated for interactions matrix */
    int     *inttc;    /* interaction term count */
    int     **ints;    /* interactions matrix, rows=interactions, cols=index in numiv */
    char    **intnames;/* array of interaction names, eg. "var1*var2" */
    dataset *xtab;      /* cross-tabulation of all model variables */
    dataset **freqs;    /* array of frequency tables for all model variables */
} model;
```

Now, when we call `mlelr` from `cmd_logreg`, the call is very simple:

```
retval = mlelr(ds, mod);
```

where `mod` is initially allocated in `cmd_logreg` and `ds` is a pointer to the dataset for which the model will be run. The return value can be used to handle the various difficulties encountered by the logistic regression procedure.

3.7 Second-guessing the User

"You told me to do that, so I did." This is our basic philosophy when it comes to dealing with unexpected or undesirable user input. Users can and will, either by mistake, curiosity,

or just plain old dumbness, do something that stretches the boundaries of common sense and possibly bring the system to a grinding halt. If the user says they want to load a 4GB file on a system with 1GB of main memory, we're not going to try and stop them. If the user wants to print the entire contents of a dataset with a hundred thousand observations, we will happily let them watch as we output every line to the screen. If the user wants to treat a continuous variable with thousands of levels as a categorical variable (as opposed to a direct effect) in a model, again, we will let them do what they want.

For large projects, user interface design can evolve a complicated set of checks and balances, constraints, and prompts to balance what is expected and appropriate behavior against what the user has requested to do. We prefer to keep things as simple as possible for our demonstration program, and that typically means attempting exactly what the user has requested. The only stipulations are that our command syntax rules are followed. For example, if the user attempts to specify a main effect after an interaction effect, we will fail the syntax check because we need to abide by a consistent definition of our design matrix which places all interaction effects after the main effects.

Another area we will fail to allow is if the user wants to specify an interaction effect *without also including the components of the interaction as main effects.* Although it is statistically ill-advised to run such models, many programs will allow this anyway. While certainly more flexible, this places additional requirements on the construction of the design matrix that are not easily accounted for. Specifically, we tabulate each main effect in order to know how many columns are required to store it—as well as any interactions in which it is involved—in the design matrix. To handle interactions that include variables that are not specified among the main effects adds additional complexity that does not materially contribute to overall functionality. It also forces the user to reconsider their choice. If you insist on running such a model, simply create a new variable, or set of new variables, that express the interaction effect you are intending and then specify the new variable as a main effect in the model. The end result will be the same.

3.8 Generality and Compromise

A common trade-off in programming involves the level of generality vs. specificity that is required to solve a particular problem. If you make the solution too specific, it will have only one possible use—it will not be possible to reuse it or easily adapt it to be used elsewhere. This is often fine and programs like these are often thought of as "one-off" or "ad-hoc" solutions. At the other end of the spectrum, if you attempt to design a perfectly generic solution, you may find yourself descending into an absurd level of abstraction.

Reuse is a lofty goal, and laziness is a virtue because as a programmer you want to follow the DRY principle: *Don't Repeat Yourself.* Having too much one-off code that is specific to only one problem means that you will be writing far more code than you need to. Sure, it's certainly easy to copy and paste, but every time you do so you make the code more difficult to maintain. What if you want to make changes? Now you have to make the same change in multiple places. And worse, what if you find a bug? By copying and pasting, you are just adding more bugs to your program.

This is the reason design patterns were invented and have become so popular in object oriented programming methodologies. If you have multiple problems that share a common

pattern, it is best to solve the general pattern first so you can have one solution that is adaptable to many different situations. Abstraction is a necessary task at the outset of program design. Yet, there is an opposite extreme of 'analysis paralysis' where you can get so caught up in layers of abstraction that it no longer becomes a convenience. It becomes instead a hindrance to solving your problems.

Thus, the approach that we find most sensible is to seek a middle ground of generality and then weigh the various compromises which a given level of generality imposes on us. You will find examples of this approach throughout the various design decisions we have made in `mlelr`. Sometimes we opt for a simpler, less generic, more specific solution and at other times it is necessary to take a step back, refactor the design into something more abstract so that it can be hardened into a more durable solution that is easier to maintain. Evaluating the compromises we make along the way is essential in finding the optimal solution.

3.9 Wrappers

The C standard library is your best friend. But, sometimes you need to offer your friends a little help—soothe over their mistakes, play up their good aspects, and provide some cover for them when the vice-principal is interrogating them for ditching class. Help your friends and they'll help you in return. It is rarely a good idea to completely reimplement a library function, but it is often necessary to wrap the calls to handle return values and conduct appropriate error-checking on the results.

In our case, we follow the lead of Sirs Kernighan and Pike in wrapping calls to `malloc`, `realloc`, and `strdup`—three functions which we use to request memory from the operating system. These calls will return a NULL pointer if the memory allocation has failed. If this occurs, a more fully-featured program would initiate a graceful shutdown, or perhaps ask the user whether to save state, abort, retry, ignore, yadda yadda. In `mlelr` we don't really care about being graceful. If we can't allocate the memory we need in order to continue, we just throw a fatal error and exit immediately. So, every time we call one of these memory allocating functions, we will test the result for NULL, and if so, print an error message, and exit the program. These functions are declared in `dataset.h` and available to any module in the program.

```
/* safe allocation of memory */
extern void *emalloc (size_t n);
extern void *erealloc (void *vp, size_t n);
extern char *estrdup (char *s);
```

Implementation of `emalloc`:

```
void *emalloc (size_t n) {
        void *p;

        p = malloc(n);
        if (p == NULL) {
                eprintf("Fatal error!  malloc of %u bytes failed:", n);
        }
        return p;
}
```

We also use a custom `eprintf` function (again courtesy of Kernighan and Pike), which handles buffer flushing.

Wrappers are useful for our own functions as well as for library functions. For example, our logistic regression procedure is executed by the `mlelr` function in `mlelr.c`, but this is not immediately called in the user interface. Instead, we have use `cmd_logreg` defined in `interface.c` to handle syntax checking and prepare the model structure for use in the actual procedure call.

3.10 Logging

Logging is an essential feature of any moderately sized program, not only of benefit to the end user to understand what is currently going on in the program, but also to the programmer for diagnostic and debugging purposes. There are a variety of approaches to logging, with the simplest being to scatter some tactically located `printf` statements in critical sections of code. On the other end of the spectrum are far more complicated logging libraries with rich feature sets and learning curves that quickly make one realize why so many programs have such poor logging capabilities. Our solution is definitely on the light-weight side of this range, and that's the way we like it. Our demonstration program is large enough to require some logging features but not large enough to require a dedicated logging thread.

By default, log messages are written to `stderr` to distinguish them from program output which is typically written to `stdout`. In console mode, these are equivalent, but we provide command-line options to redirect logging and output to external files, if desired. The very first thing that `mlelr` does is to initialize its input/output files and the logging level with the `init` function defined in `main.c`

```
void init (void) {
    LOGFILE = stderr;
    OUTFILE = stdout;
    INPUTFILE = stdin;
    LOGLEVEL = INFO;
}
```

These are all global variables declared in `interface.h`. If you don't like global variables, that's perfectly fine. I don't really like them either, so that makes two of us. We define three `const int`s in `interface.h` that help us determine whether log messages are appropriate for the current log level.

```
int LOGLEVEL;
const int SILENT    = 0;
const int INFO      = 1;
const int VERBOSE   = 2;
```

The `LOGLEVEL` variable is set by default to `INFO` in `main`'s initialization function. Logging is done by a custom function `printlog` which is a wrapper around `vfprintf`, a variant of `printf` that allows for variable arguments to print to a file.

```
void printlog (int loglevel, char *format, ...) {
    va_list args;

    if (loglevel <= LOGLEVEL) {
        va_start(args, format);
        vfprintf(LOGFILE, format, args);
        va_end(args);
    }
}
```

The key insight here is the `if` statement which tests the loglevel argument passed in the function call against the `LOGLEVEL` global variable indicating the current level of verbosity that should be used in logging. As evident in the code above, a log message will only be printed if the loglevel argument is less than or equal to the global `LOGLEVEL`. So, when, by default, the global `LOGLEVEL` is set to `INFO`, calls to the `printlog` function with a loglevel of `VERBOSE` will not be printed. This allows us to suppress some log messages unless the user has explicitly asked for verbose mode.

3.11 Constants

Constant values are variables that are set at compile time and not expected to change throughout the life of the program. These are also referred to rather disparagingly as "magic numbers." There are four basic ways of defining constants in C, listed in order from "worst" to "best", they are:

```
while (x == 42) {
#define MAXBUFFER = 1024
enum { NOMEM = -2 };
const double PI = 3.1415926
```

Literals are so often frowned upon because they offer no guidance as to what they mean and, being hard-coded, present maintenance challenges if and when they need to change. This is why "magic numbers" are frustrating when encountered in code. What does the number mean? If we change it in one place, how many other places need to be changed? How does it interact with other variables or constants? It is impossible to tell, because they're just magic.

Macro variables used as constants are barely better than using literals directly. Essentially, the pre-processor will simply do a global search and replace of the macro variable name with its value, which means the compiler has no chance of detecting errors that may arise from bad comparisons or trying to use the value with a data type with which it is not compatible. For this reason, one of the latter two methods are preferable.

Enumerated types will only work with integral values, and can be useful in certain cases like when you are evaluating various return values from a function and want to have some conditional logic using an if-else or switch statement without having to hard-code the return values in the code. Still, this bypasses any compiler type checking and is really not much cleaner than using macro variables.

By defining a proper variable and using the `const` keyword, you are fully enabling the compiler to do its job. Using `const`, the compiler understands your data and can ensure

two things: first, you will be guarded against type mismatch operations, and second, the compiler will not allow you to change the value of the variable. This is usually the best method of defining a constant value for use throughout the program.

To distinguish constants from regular variables, we like to use all caps. This way the variable name is SCREAMING at you that there is something a little different about it to which you should pay some careful attention. Note, however, that we also use all caps for special global variables like the `LOGLEVEL` and input/output file handles. This inconsistency may be too much for small minds to bear, in which case, we do suggest you fix it to your own liking. Just don't get too religious about it—to my knowledge nobody has been sent to Hell for being slightly inconsistent in their naming of variables.

3.12 Security

Since I have no special skills in security engineering nor did I consciously set out to design and code this program using defensive techniques, and given the fact that this program eagerly laps up user data with only the barest forms of validation, it is safe to assume, in light of how readily security flaws are found in systems intentionally designed to be hardened against them, that there are numerous exploitable holes in this program that would enable a hacker of the 1337 variety to handily gain control of the system and bring the world financial system to its knees (again).

There are actually two issues involved here. First, this is a young program that accepts user input. By young, I mean that testing has thus far primarily been concerned with benign well-formed datasets designed to work within the system, not destroy it in a raging fit of teen angst. We're not treating our datasets with suspicion, we're simply trying to get them to work. Testing has revealed that loading an improperly formatted dataset could result in a segfault. This is because we allow commands to be run even if an import has failed. What happens if you run an import command on top of an existing dataset? Such questions have never been a concern because we are expecting the user to try to make things work, not break them. I'm almost certain that someone could craft a dataset that would smash the stack and allow an attacker to run arbitrary code. This is the nature of user input. The fact that we're not even superficially attempting to guard against attacks nearly guarantees that a dedicated attack will succeed.

The second issue is that security engineering is a highly specialized domain with institutional knowledge and an arsenal of techniques employed to detect and guard against potential exploits. The most I've thought about security while writing this program has been "try to use `strncat` instead of `strcat`." Any time there is a risk that bad guys would have an incentive to break your program—if your system is dealing with PII for example—then having a professional security engineer on the team is absolutely critical. You really want someone to tell you that you shouldn't even run a "Hello, World!" program without a proper security audit.

Despite the fact that I completely fail to follow my own advice and thus contribute to the world yet another insecure program, I point this out because it nevertheless is important enough that everyone should at least be aware of security implications even in the simplest of programs and even where risk is considered to be minimal. In the spirit of a good old-fashioned crowd-sourced challenge, I offer the following bug bounty: one hand-written

personally addressed postcard will be sent to the first person who can demonstrate an exploit involving `mlelr` which will allow the user to run arbitrary code. I realize this is not much of a bounty, but I am very cheap. As an extra bonus, I will also throw in my acknowledgment and admiration of your elite skillz. Surely that's enough incentive!

To close this section I will make one safe, and serious, recommendation: please do not allow `mlelr` to run in an untrusted environment or on any system that contains sensitive data.

3.13 Freedom

One thing you may note about `mlelr` that is bound to draw the ire of the particularly anally-retentive, is that there are frighteningly few `free` statements in the program. As a stark example, we provide no facility to "delete" a dataset or otherwise release the memory that has been allocated to it. This means our program will leak like a sieve. It's possible to import multiple datasets into the dataspace, but not to remove any of them. ZOMG! But I was taught never to write a `malloc` statement without also writing the corresponding `free` statement! And this is the beauty of `free` software: if you don't like it, you can just go and do it yourself.

The truth of the matter is that this functionality is not entirely necessary for our reference implementation. We expect the user to import a dataset, specify and run a logistic regression model, and then exit the program. We don't anticipate long-running sessions with multiple datasets in use simultaneously.

And the real truth of the matter is that the previous paragraph is a complete rationalization and the bottom line is in addition to being cheap, I am also lazy. And being a lazy programmer means knowing that the `return` statement is the easiest and most reliable way to handle memory management. All jokes aside, in a production system, it would be imperative to properly free memory from datasets or model structs that are no longer pertinent.

4 Data Management

Popular statistical packages offer their own complete programming environments in order to provide users with the flexibility needed to manage data and prepare it for use in statistical procedures. Such systems can include their own scripting language, possibly with macro pre-processing facilities, a lexer and parser, and other features that enable users to extend and adapt the system to solve complex problems in their own domain. To be as generic as possible requires a great deal of abstraction in the design of essential data structures. Browsing the R source code will reveal macro functions for `car` and `cdr` operations. These concepts are borrowed from Lisp and enable the R parser to manage an object-oriented environment with powerful features like function closures. This underscores the simplicity and extensibility of the R language (arguably, also its insanity-inducing frustration). Unfortunately, it can also be difficult to understand program code when it is wrapped up in these various layers of abstraction.

Since `mlelr` is primarily a learning tool, many of the facilities provided in a full statistical programming environment would be unnecessary overkill. Additional layers of complexity

often obfuscate the intent of the code and how it works. For `mlelr` we would like the data model and associated data management facilities to be as simple as possible—but no simpler. In other words, we want data access functionality that is easy to use, but also generic enough to handle arbitrary datasets loaded by the user at run time. This implies that we should be able to accept any number of observations and variables, and that we should provide the user with the ability to refer to datasets and variables by name. We will want some basic scripting functionality, without which the only alternative would be for the user to recompile the program source code every time they wanted to do something different. As noted above in the section on Generality and Compromise (Section 3.8), we cannot be entirely generic and will need to make some compromises by enforcing some restrictions that we consider reasonable for our demonstration program.

The first compromise we are going to make is to require that all values be stored as double-precision floating point variables. This means we will not be able to use datasets that contain string variables. This would be unacceptable in a fully-featured statistical package, but ours is not, and the extra boiler-plating that is required in order to support string variables really doesn't belong in a reference implementation which will make no direct use of them. In SAS, SPSS, and R, string variables that are used as categorical variables in statistical procedures are converted to numeric types in the actual processing of the routines. SAS and SPSS will use formats or value labels to map string values to underlying numeric values, and R will use factors to do the same thing. Since the underlying data needs to be numeric anyway, and since we are not designing a general-purpose data management tool, we find it not too inconvenient a requirement that all input data must be numeric. Thus, when preparing a dataset for use with `mlelr`, use whatever environment is convenient to recode any string variables (e.g. {Yes, No}) to nice numeric variables (e.g. {1, 0}). Integers are automatically promoted to doubles, and exponential notation is acceptable—basically anything that looks like a number can probably be successfully parsed as a number.

Among the numerous ways of parsing input data, we have defined a special function for this purpose:

```
static int string_to_double(char *str, double *d);
```

This is defined in `dataset.c`, and is used exclusively by the `import_dataset` function to read in data from an external file. This custom function is essentially a wrapper around `strtod` defined in `<string.h>` in the C standard library. The manpage for this function has more detail on its return codes. In our use case, we will not pay too much attention to the return codes, except to set the value equal to our `SYSMIS` constant if the conversion to double was unsuccessful.

The function `fscanf`, derived from `scanf` defined in `<stdio.h>` is an alternative method of reading input from a file and, using format specifiers, attempt to convert the resulting string into a floating point number. However, `scanf` and its derivatives handle unexpected input poorly and since the return value is the number of characters successfully read, it is not always possible to know if there was a problem with the input. In addition, it is not possible to vary between exponential notation and standard decimal notation since you must hard-code the format specifier into the function call. For these reasons, we use `strtod` as the safer, more flexible alternative.

Now that we have defined a way to read data from a file, what do we do with it after reading it in? We expect most datasets that we will encounter to be standard flat tabular files with rows and columns—observations and variables—stored in *row-major order*, meaning that all the variables for one observation are stored first, followed by all the variables for the second observation, etc. Since we're only concerned with a single data type, double-precision floating point, what we're essentially looking for is a data structure to store a matrix of doubles. However, C has no built-in concept of a matrix. So the fun and exciting task falls upon us to invent one. What C does have a built-in concept of is an array. Arrays can have multiple dimensions. So, in the simplest implementation, we could define our dataset as a two-dimensional array of doubles:

```
int NROWS = 100;
int NCOLS = 5;
double dataset[NROWS][NCOLS];
```

Unfortunately this is not very flexible. With a two-dimensional array, only one of the indices can be supplied dynamically at run-time. The first index, however, must be known at compile time[4]. Thus, in order to use this data structure to store our dataset, we'd have to know, or if not, guess, either the number of observations or number of variables. This is not very practical. We really need a data structure that can adapt to hold an arbitrary number of observations and an arbitrary number of variables per observation—memory permitting, of course.

Fortunately, C has another well-known, flexible, powerful, and dangerous method of mimicking a multi-dimensional array or matrix: pointers. If we declare our dataset as a pointer to a pointer to double:

```
double **dataset;
```

Now, we can allow for arbitrary dimensions specified at run time by using `malloc` and `realloc` to manage the memory pointed to by our dataset variable. In order to make use of this data structure, you will need at least two calls to the `malloc` function. First, you need to allocate space for some number of pointers to double:

```
dataset = (double **) malloc(NROWS * sizeof(double *);
```

Now what we've done is requested enough memory from the operating system to hold `NROWS` instances of pointers to double. In a row-major design, these would be pointers to the first variable in each observation. Next we need to allocate space for each observation to handle the number of columns we need:

```
for (i = 0; i < NROWS; i++) {
    dataset[i] = (double *) malloc(NCOLS * sizeof(double));
}
```

[4]This was true until the C99 standard which provides for variable-length arrays. However, these are still less than ideal for use with a general purpose array or matrix representing a dataset for two reasons: first, functions that expect arrays cannot always handle arrays defined as pointers, and vice versa. (See C FAQ question 6.18 [7]). Second, arrays are initialized on the stack which means they disappear once out of scope. Not only is it dangerous to instantiate large objects on the stack, it also makes it impossible to pass them around from one module to another.

An important design consideration is whether to use a column-major or a row-major organization of our dataset. In other words, should our first index represent the row or the column? Row-major ordering is, in my opinion, more natural. As noted above, most datasets we expect to encounter will be stored in this way; this is the standard way that spreadsheets, csv files, and general purpose data files are oriented. Thus, a row-major design for our dataset will make data import easier. When reading data from an external file, we will read one line at a time, where each line is a new observation. In addition, matrix notation is typically written in row-major order, with the row subscript appearing first, followed by the column subscript. Since we want to easily compare our code to the formulas on which they are based, a row-major ordering to our dataset structure will maintain consistency and be easier to read.

Column-major designs can be more efficient in certain situations. Analytic database systems often use column-major ordering to make data aggregation more efficient. HP's Vertica, for example, uses column-major ordering combined with run-length encoding to compress observations. Consider a variable with 100 million rows, but only two unique values. Instead of storing 800 million bytes of data, you could sort them and store 4 numbers: the first value and then a count of the number of rows containing that value, followed by the second value and the count of the remaining number of rows with that value. Such optimizations, however, are unlikely to be made use of in our reference implementation, but it helps to be aware of the possible implications of choosing a row-major versus a column-major ordering.

Another concern with the row-major ordering is the number of `malloc` calls required to reserve space for the dataset as it grows in size. The datasets we expect to use in logistic regression will have several hundred or even several thousand observations but a much smaller number of variables, perhaps 10 or 20. Thus, in a simple row major ordering, we could end up calling `malloc` once per observation, which could amount to thousands of system calls. Even with relatively small datasets we may notice the performance penalty of this approach. Fortunately, there are two quick optimizations we can take advantage of that will mitigate this problem.

The first optimization is to grow the array by powers of 2, maintaining separate counts of the number of observations and the maximum number of observations allowed given the current amount of allocated memory. This technique is very well documented by Kernighan and Pike [5]. There will be some amount of wasted space—in our example of 1,000 rows we will have allocated space for 1,024—but we judge this to be an acceptable compromise[5].

The astute reader will have recognized that there is another problem with our row-major design. Even though we grow the array by powers of 2, we will still need one `malloc` call per observation. Each time the number of observations reaches the maximum number of observations for which we have reserved memory space, we will need to `realloc` the dataset to hold twice as many observations. Recall that `dataset` is a pointer to a pointer to double. Each "row", `dataset[i]`, is an array of doubles with size equal to the number of variables in the dataset. So now that we've allocated space to hold additional observations, for each

[5]You could conceivably improve on this by using a realloc after all your observations are read, to shrink the allocated space back to only what you need. However, if you are really this memory-constrained, then your dataset is probably too big for the system and you will likely run out of memory at some point down the road.

observation we subsequently add to the dataset, we would still need to `malloc` enough space to hold all the variables in the observation.

Note that if we had a column-major ordering, the number of `malloc` calls would not be as large. If the first index in the array represents a variable, then every time we change size by a power of 2, we only need one `realloc` per variable. Thus, for a dataset with 1,000 observations and 10 variables, using a column-major ordering we would have the following:

- 11 initial `malloc` calls: one for `dataset`, and one for each column, `dataset[i]`
- 100 `realloc` calls: $\log_2 1000 = 10$ cycles of growing the arrays * 10 variables each

Compare this to our row-major design:

- 2 initial `malloc` calls: one for `dataset`, and one for the first observation, `dataset[i]`
- 10 `realloc` calls: $\log_2 1000 = 10$ cycles of growing the array
- 999 additional `malloc` calls: for each observation before we read it into the array

The difference is due to the fact that in the column-major design, we can make fewer requests for larger amounts of memory, since we are storing one entire variable as the first dimension in the array. In the row-major design, the first dimension in the array is an observation, so the most we can ask for at any one time is the number of variables that the observation will hold.

Note that even if you know before hand how many observations are in your dataset[6], you still need to call `malloc` once per observation. There's no way around it. Or is there?

Of course there is, otherwise I never would have bothered to write the preceding paragraphs and this section would have been a whole lot shorter! The solution is to create a singly-dimensioned array of contiguous space for the entire dataset, and then use another matrix of pointers to facilitate the convenient indexing. At this point, let's introduce our dataset structure, defined in `dataset.h`:

```
typedef struct {
    char    *handle;        /* a short label for this dataset */
    int      n;             /* number of observations */
    int      size;          /* amount of space allocated for values */
    int      nvars;         /* number of variables */
    char   **varnames;      /* array of variable names */
    double  *values;        /* array of contiguous space to store all data */
    double **obs;           /* matrix of pointers to access each obs[i][j] */
    int      weight;        /* index of the weight variable, or -1 if none */
} dataset;
```

Note that we have wrapped some additional metadata along with the actual data values so that everything we need to describe a dataset can be contained in one place. (This is as close as we ever need to get to C++ objects). All of the actual data will be stored in the

[6]You could do this by reading the file twice, but this is such a wastefully sub-optimal solution that we really should not even bother considering it.

`values` pointer, which will be incremented as the number of observations grows and will grow in multiples of the number of variables needed for each observation. We could stop here and use `values` directly, but then we would have to do annoying pointer offset arithmetic every time we wanted to access a single observation, e.g. observation 10 of variable 5, in a dataset with 20 variables would be:

```
values[4 + (9 * 20)]
```

Instead, we build a matrix of pointers, `obs`, that all point to `values` in such a way that we can simplify data access and then our ability to access the values in the dataset becomes far more natural:

```
obs[9][4]
```

This introduces some additional memory overhead, since `obs` will consume one pointer for each "cell" in the dataset. Since a pointer is typically 4 bytes vs the 8 already allocated for the `values` array, we're introducing a 50% increase in the amount of memory we need to store the dataset. Ah, such is the price we pay for convenience!

Additional metadata includes a short handle which the user can use to refer to the dataset. This is useful in either a script or interactive environment. Next, a few `int`s to store the number of observations (`n`), the total number of observations for which space has been allocated (`size`), and the number of variables (`nvars`). Finally, we include an array of variable names so the user can conveniently refer to names as opposed to column indices.

Now that we have introduced a decent abstraction for a dataset to organize the data and its corresponding metadata in a convenient C `struct`, the following section will describe the program flow of `mlelr`. This section will describe how to interact with the program, what commands are available, how to read data from external files into a dataset, and how to setup and run a logistic regression.

5 Program Flow

This section will describe how `mlelr` works by detailing the program flow of the code and describing the commands available to the user.

5.1 Installation

`mlelr` can be downloaded in source form as a compressed archive. Run the following command in a compatible shell to unpack the archive:

```
$ tar xzf mlelr-1.0.tgz
```

Enter the 'src' directory and run the command `make`. In order for the program to link properly, you will need to make sure the gsl library is installed. See Section 8.5 on how to setup a development environment in which to build and run `mlelr`.

5.2 Program Use

`mlelr` is compiled and linked to a standard console application consistent with the executable formats available on the target platform. Running the program with no arguments will enter an interactive mode and the program will wait for user input:

```
$ ./mlelr
mlelr - a reference implementation of logistic regression in C
version: 1.0
mlelr->
```

There are several available command line arguments. The "-h" or "-help" argument will bring up a brief help message. This will also be printed if an unrecognized command line argument is passed to the program.

```
$ ./mlelr -h
mlelr - a reference implementation of logistic regression in C
version: 1.0
Available command line arguments:
     -f or -file:    read and execute commands from the named file
     -o or -out:     redirect output to a file
     -l or -log:     redirect log messages to a file
     -v or -verbose: log extra detail to the log file
     -s or -silent:  suppress all logging information
     -h or -help:    print this message
mlelr->
```

There are three options that can be used to redirect input and output. The code behind the processing of startup options is found in the `main` function in the file `main.c`. To skip interactive mode and redirect input to read from a script file, use the "-f" or "-file" option. This option, when immediately followed by a valid pathname, will attempt to open and read the file and then process commands from the input file instead of from the console directly. When all commands from the input file have been processed, the program will exit.

We use a rather basic convention for output and logging of errors and warnings. Ordinarily, the results output from the normal operation of a command will be printed to `stdout` while error messages, warnings, or informational diagnostics will be printed to `stderr`. In a standard console, however, these pipes will likely resolve to the same place, i.e. the screen. To assist with scripting, we provide two options to redirect either or both of these targets to a file. The "-o" or "-out" option will redirect output to the file named immediately after the option. For example:

```
$ ./mlelr -o output.txt
```

Similarly, the "-l" or "-log" option will redirect logging to a file. Note that by convention, we treat the output file as a single-session only file while the logging file can persist across sessions. This is achieved by opening the output file in "w" (write) mode, which will overwrite it if the file already exists, but we open the logging file in "a" (append) mode,

which will append to the file if it exists. This is beneficial when you have a script that will call `mlelr` in a loop, estimating a new model each time, and parsing each output file independently, while maintaining a single log file that will collect diagnostics across all calls to the program.

There are two additional switches that control how little or how much information is written to the log. The "-v" or "-verbose" option will provide more detail, this is mostly of benefit while debugging. The "-s" or "-silent" option will attempt to silence all logging information. Note, however, that output messages will still be written to the output file, and some fatal error messages, such as running out of memory, will still be printed to the log even if the silent option is on.

Once command line options have been successfully processed, the dataspace is initialized—memory is allocated for the structure to hold pointers to datasets—initial options are set, and a welcome message is printed. At this point, control is passed to the input handler function. This function is continually called until a non-zero value is returned, which signals the program to clean-up and exit.

```
while (!retval) {
    retval = input_handler();
}
```

The purpose of the input handler, defined in `interface.c`, is to accept commands either from a specified input file or from direct input by the user to `stdin`. The input handler will call Kernighan and Pike's csv library, splitting on spaces to delimit each line into words. This is a poor-man's parser—although it is not purposely built for parsing, with two small adaptations it became completely adequate for our purposes. It forces on us a rigid and terse command syntax, but lexing and parsing are extremely non-trivial areas of program design that should not be entered lightly and which we desperately seek to avoid in a demonstration program of limited means. Besides, we're not in the business of designing flowery scripting languages here. All we need is the minimum syntax necessary to tell the program what we want to do. This is another compromise we are willing to make in order to reuse functionality that is already present in importing data files, and adapting it for parsing command strings from user input.

One limitation is that each command must consist of only one line. Fortunately, our commands typically require only a few arguments so this is not terribly inconvenient. However, for the specification of a complex model, it may become cumbersome to keep everything on one line. It is also important to keep arguments delimited by spaces. When an argument itself includes a space character, wrapping the entire argument in double quotes will allow the csv parser to treat it as a single field.

The input handler begins by printing a prompt if in console mode, i.e. when reading from `stdin`. Next, the csv library is called, splitting on spaces and compressing multiple consecutive spaces into a single delimiter. If nothing is returned by the csv library, the program will exit. Otherwise, we will scan the array `cmds` for an entry that matches the first word that was read by the parser. If a match is found, the `cmds` array contains a function pointer which is then executed, and control is passed to that function where the remainder of the command string is checked.

To make this happen, we first define a special typedef for a pointer to a function with no arguments and returning int. This will be the basic entry point for all functions accessible to the user. Then we create a `COMMAND` typedef with the name of the command and a pointer to the function to call if that command is found by the parser:

```
typedef int int_fp_v (void);

typedef struct {
    char      *name;      /* the name of the command */
    int_fp_v  *f;         /* pointer to the function to call for this command */
    char      *desc;      /* brief description of command */
} COMMAND;

COMMAND cmds[] = {
    {"import", cmd_import, "Import a delimited text file."},
    {"print",  cmd_print,  "Print a dataset."},
    {"table",  cmd_table,  "Univariate frequency tabulation."},
    {"logreg", cmd_logreg, "Estimate a logistic regression model."},
    {"weight", cmd_weight, "Assign a weight variable to the dataset."},
    {"option", cmd_option, "Set a global option."},
    {"help",   cmd_help,   "Print some help on command syntax."},
    {"q",      cmd_quit,   "Exit the program."},
    {"quit",   cmd_quit,   "Exit the program."},
    {"#",      cmd_comment, "This line is a comment."},
    {(char *) NULL, (int_fp_v *) NULL, (char *) NULL}
};
```

To differentiate them as special functions, we give all the functions in the `cmds` array a prefix of "`cmd_`". They must all operate the same way: accepting no arguments and returning 0 unless we want the program to exit. The `cmd_quit` function is the only one that ordinarily returns a non-zero value, since exiting is its intended purpose. The idea behind these "`cmd_`" functions is to validate the syntax of the commands before passing control to the actual procedure to perform the required work.

5.3 Command Syntax

As one can deduce from the `cmds` array above, there are only a small number of commands to which our program will respond. The input handler will parse the first word of a command string and perform a simple linear search through the `cmds[]` array. Some may chafe at the inefficiency of a linear search, but for a use case such as this, where there is a very small number of available commands, binary search or a hash table approach would be unnecessary overkill. As a general rule of thumb, linear search will be the most efficient solution for arrays with less than roughly 50 items[7]

As noted above, the general syntax rules to which all commands must conform are to follow the command name with its arguments delimited by spaces, with everything on a

[7] If the array is sorted based on expected match frequency—putting the most used commands at the top—linear search can be efficient up to several hundred array items before its performance can be topped by other methods.

single line. Any newline will be treated as the end of the command and parsing of arguments will cease. When a command name is matched, control is passed to the special `cmd_` function corresponding to the command, where additional syntax checking will occur.

5.4 Import a new dataset

The `import` command is used to read a delimited text file and generate a dataset which can then be used in subsequent commands by referring to its handle. This is generally the first command you will want to use since there is very little to do in `mlelr` without having a dataset available in the dataspace. The text file containing the data that we want to import needs to reside on the local filesystem. A delimiter can be any single character, although it is conventional to use either a comma, tab, pipe, or space. Since a tab character is not easily represented in its actual form, we can refer to it using the escape sequence `\t`. Handling escape sequences is not a pretty task and for the sake of simplicity, with a dash of laziness, we have hard-coded the parsing of `\t` to be a tab character, but have not done the same with other control sequences. Thus, if you have data for which someone decided to use an exotic non-standard delimiter character such as the BEL character, 0x07, represented as `\a`, I'm afraid you will have to convert this to one of the afore-mentioned standard delimiter characters in order to work with `mlelr`. Let this be a lesson: don't use exotic delimiter characters. Use a comma, tab, pipe, or space.

The import command expects three required arguments that will be parsed by the `cmd_import` function: a "handle" which is merely a short name for the dataset by which subsequent commands may refer to it, the filename where the raw data file is located, and the delimiter character that should be used to separate fields in each line. Here is an example of an import command:

```
import ingots ../data/ingots.dat "\t"
```

If a successfully parsed import command is read, control is passed to the `import_dataset` function declared in `dataset.h` and defined in `dataset.c`. The above command will import the ingots dataset from the 'data' directory, specifying that the file is tab-delimited. Our `import` function assumes that the first line of the data file contains variable names. If you produce your own dataset for use in `mlelr`, be sure to include variable names as the first line in the file, delimited by the same delimiter used for the actual data.

As noted earlier, we use Kernighan and Pike's csv library to read each line of the data file and transform it for use in our dataset structure. The interface to the library is declared in `csv.h` and the source is in `csv.c`. We made two changes to the original implementation in order to add two small bits of extra functionality. The first change we made was to allow for delimiters other than the comma. The original implementation hard-codes the comma as the delimiter character, and for good reason—the name of the library itself calls out the assumption that the values are comma-separated. The original code contains this line which defines the field separator to be a comma:

```
static char fieldsep[] = ","; /* field separator chars */
```

We abstract this by giving the `csvgetline` function a new argument `char delim` which is then used throughout the library in place of the `fieldsep` char. The other change we

made is the new `int compress` argument to `csvgetline`. This is called with a 1 or 0 where a value of 1 indicates that multiple delimiter characters in succession are to be treated as a single delimiter. This is useful when in command strings we typically want to delimit words by any combination of spaces. If a user command includes two spaces after a command word, we don't want the parser to think of this as an extra blank field. Similarly, in the preparation of datasets, tabs and spaces are often mixed, or extra spaces are used as padding to line up the columns for convenient reading. Note, however, that for this version of `mlelr`, we have hard-coded the `import_dataset` function to call `csvgetline` with `compress = 0`. However, if you wanted to allow for this functionality, it would be possible to allow a user the option of specifying whether to treat all delimiters individually or to collapse them. We may consider this for a future version.

After reading the variable names from the first line of the specified data file, we now know how many variables for which to allocate space in our dataset.

```
ds = add_dataset(handle, nvars, varnames, 1);
```

The `add_dataset` function, also defined in `dataset.c`, takes 4 arguments and returns a pointer to a `dataset` struct. See Section 4 for a description of our dataset structure. The first argument is the handle providing a convenient name for the dataset. Next is the number of variables in each observation of the dataset. It is necessary to know the number of variables in order to allocate space for the `values` and `obs` arrays. The final argument, `int is_public`, determines whether to add the dataset to the user's dataspace. Throughout `mlelr` we take advantage of the dataset structure for a variety of purposes. It is used in frequency tabulation, for example, to store the table of values and counts. The datasets that are used by `mlelr` behind the scenes are not added to the dataspace, so the user does not have direct access to them—and they don't unnecessarily pollute the dataspace with a lot of temporary datasets that the program is creating but for which the user doesn't really have any use. All datasets called via the `import_dataset` command will be called with `is_public = 1`.

Since our dataspace structure is dynamically allocated, we first check whether we need to grow the array of datasets, as we would if we were adding a new observation to a dataset. We grow the array in powers of 2 using the same method identified by Kernighan and Pike that we described earlier.

After the `add_dataset` function returns, we then allocate space for an array of doubles to store each observation as we read each line from the file. Note that we cannot handle jagged arrays in the data file: if we encounter a line with a different field count than what we expect, the import function will fail and print an error message. We parse each field using the `string_to_double` function which provides a wrapper around the `strtod` library function. There are some edge cases which we are probably being overzealous in attempting to account for, but in general it is a good idea to check the return values for library functions, especially those that deal with user input.

For each line, we read each field into the `obs` variable. After the entire observation has been parsed, we call the `add_observation` function. This function sets up all the ugly pointer arithmetic that make the `obs` and `values` arrays work properly as described above in Section 4.

Once all lines are read from the input file, the number of observations imported is logged, the input file pointer is closed, and the `import_dataset` function returns 0, indicating success. This returns control back to the `cmd_import` function which cleans up and then returns control back to the `input_handler` function which will then return 0 back to the while loop in the `main` function. Since we returned a 0, the loop will continue, the `input_handler` function will be called again, and we will then await input from the user or begin to process the next command from the `INPUTFILE`.

Reviewing the call stack we have the following flow of function calls:

```
main() -> input_handler() -> cmd_import() -> import_dataset() -> add_dataset()
                                                                 add_observation()
```

5.5 Print a dataset

The `print` command is used, predictably, to print a dataset to the output file (or `stdout`). It takes two required arguments: the first is the handle for the dataset you wish to print, and the second is an integer specifying the number of lines to print. If you want to print the entire dataset, enter 0 for the number of lines. The following example command will print the first 10 lines of the dataset with handle 'gator':

```
print gator 10
```

The function that actually prints the dataset is called `print_dataset` and is defined in `dataset.c`. This function takes 3 arguments: a pointer to a dataset structure, the number of lines to print, and a third integer argument indicating whether a header should be printed with some summary information to the output. In order to translate the handle provided in the call to `cmd_print` to a pointer to the dataset with that handle, the `find_dataset` function is used. This function walks through the dataspace looking for the first dataset that matches the supplied handle. It is crucial that handles remain unique, although `mlelr` does not enforce any rules around handle uniqueness. If you use `add_dataset` with a handle that already exists, your new dataset will never be found anytime the `find_dataset` function is used to identify the dataset by its handle.

The header option prints metadata about the dataset including the handle, number of variables, and number of observations. When printing datasets initiated by the `cmd_print` function—thus originating from a `print` command by a user—the header is always included, since it conveniently identifies and summarizes the dataset that is being printed to the output. However, there are some situations where we call `print_dataset` when we do not want this header to be included: for example, when printing datasets that are frequency tabulations.

5.6 Univariate frequency tabulation

The `table` command will tabulate a given variable and print the results to the output window. Although not strictly necessary in a program designed to demonstrate logistic regression, the functionality behind the table command is internally required for some of the preliminary steps in building the design matrix. Thus, it was actually not a big hurdle to include this as a user-facing feature. The `table` command has only two arguments: a

dataset handle and a variable name to tabulate. This example command will generate the frequency counts for the *agegrp* variable in the *census* dataset:

```
table census agegrp
```

The `cmd_table` function will use the `find_dataset` function to get a pointer to the dataset identified by the supplied handle. Next, `find_varname` is called which is a similar function to find the index of a variable based on its name. This function simply walks the array of varnames in the dataset until it finds a match, returning the index. If the variable name was found in the dataset, the function `frequency_table` declared in `tabulate.h` and defined in `tabulate.c` is called.

This is a nice meaty computer science problem because it requires use of the two staples of Intro to Algorithms texts: searching and sorting. Thus, we could approach the solution with a variety of strategies. First, we need a description of the problem: having identified a variable in a dataset, we want to produce a frequency table of its discrete values. The output should be a dataset containing two variables: *Value* and *Count*, where *Count* is the sum of the weights of each occurrence of *Value* in the dataset.

Among the variety of possible solutions, the basic paradigm difference among them is whether to maintain a sorted data structure while iterating through the problem space, or to do all the searching up front and leave the sorting until the end. Having discovered a clever method of using the library function `qsort` on our dataset structure (see below), it was far easier to opt for the latter approach.

We begin by calling `add_dataset` with the `is_public` argument equal to 0. The resulting dataset will not be added to the dataspace. We provide a custom handle which is really just a title for the frequency table. There are two variables named `Value` and `Freq`. Next we loop through each observation in the dataset and attempt to find the target value in the frequency dataset. If it has not been encountered before, we will call `add_observation`. If it has been encountered, when we find it we will increment the `Freq` variable by 1.0 or, the value of the current observation's weight if a weight variable has been set for this dataset. When this is done, we sort the frequency dataset and print it to the output.

The `sort_dataset` function involves a clever bit of trickery that enables the use of `qsort` on our dataset structure. The library function `qsort` can be used to sort a variable of any type provided that a special comparison function is provided to it which basically tells `qsort` how to determine which of two given values is less than the other. This comparison function must follow a specific function signature: it needs to take two pointers as arguments and return a negative number if argument 1 is less than argument 2, 0 if they are equal, and a positive number if argument 1 is greater than argument 2. See the documentation for the comparison function in glibc [8] for more details[8].

After struggling with various attempts to code a sort function for our `dataset` structure, I happened upon a discussion of a clever trick which would enable using `qsort`[9]. Basically, we overload the arguments to our sort function. Our sort function will set a `static` variable if it is called with the first argument null. This variable will determine how many columns

[8]link: `http://www.gnu.org/software/libc/manual/html_node/Comparison-Functions.html#Comparison-Functions`

[9]link: `http://cboard.cprogramming.com/c-programming/114867-qsort-question.html`

of the dataset are to be sorted in subsequent calls. Using this trick, we can rely on the well-tested functionality of `qsort` in the standard library instead of having to code our own sort routine:

```
void sort_dataset(dataset *ds, int n_cols) {

    compare_obs(NULL, &n_cols);
    qsort(ds->values, ds->n, ds->nvars * sizeof(double), compare_obs);

    return;
}

int compare_obs (const void *v1, const void *v2) {

    static int sort_columns = 1;
    const double *a = (const double *) v1;
    const double *b = (const double *) v2;
    int i;

    /* trickery */
    if (v1 == NULL) {
        sort_columns = *(int *) v2;
        return 0;
    }

    for (i = 0; i < sort_columns && a[i] == b[i]; i++);

    return (a[i] > b[i]) - (a[i] < b[i]);
}
```

5.7 Assign a weight variable to a dataset

The weight command is used to identify a variable used as a weight, i.e. the count of the number of observations, instead of 1, that each observation represents. This occurs frequently enough that it makes good sense to include this as a feature to support. Many datasets are distributed essentially in contingency table form, with a count variable which is basically an explicit weight. In other situations, datasets that are samples with complex sampling methods may require a weight variable to be properly representative of the underlying population. The syntax is simple as there are only two required arguments:

```
weight ingots n
```

The first argument is a dataset handle and the second is the variable name of the variable which should be treated as the weight variable. In all tabulations, including those that are used in the logistic regression procedure, the weight variable will then be treated as the logical count of observations. The function `set_weight_variable` defined in `dataset.c` changes the `weight` attribute of the associated `dataset` by setting the value equal to the index of the variable. If `weight` equals -1, then the dataset will be treated as unweighted.

5.8 Set a global option

Global options are generally not a great design pattern, but can be used to simplify other commands by setting some variable which then can be accessed globally throughout the program. Our option functionality is extremely basic. In `main`, we call an `init_options` function defined in `interface.c` which allocates space for an `options` struct, declared in `interface.h`. This allows us to set arbitrary key-value pairs which can be accessed by any module that includes `interface.h`.

Currently, we only use this for one feature: the ability to change the parameterization from the default of full-rank center-point to dummy coding. To do so, issue this command:

```
option params dummy
```

To go back to the default:

```
option params centerpoint
```

The `get_option(k)` and `set_option(k, v)` functions defined in `interface.c` enable access to any existing options set with `option` commands. The `get_option` function takes a key as its argument which will be searched in the array of keys in the `options` struct. If a match is found, the corresponding value will be returned. For `set_option`, two strings are passed as arguments, key and value. The key is looked up in the `options` struct and if found, the corresponding value will be updated. If not found, a new key-value pair will be added.

5.9 Estimate a logistic regression model

At long last we finally come to a description of the command for logistic regression, the primary purpose of our program! The `logreg` command will begin the logistic regression procedure. Since we need to pack a fair amount of information into the command string, the syntax is a bit stilted, but still functional. Below are example calls illustrating different types of model specifications:

```
logreg ucla admit = gre gpa rank
logreg ucla admit = direct.gpa direct.gre rank gre*gpa gre*rank
```

The first argument to the `logreg` command is the dataset handle and the second is the dependent variable to be modelled. Following the dependent variable is an equals sign (which owing to the limitations of our parsing method must be separated by spaces on both sides) after which we have a specification for the independent variables. There are three types of independent variables: main effects, direct effects, and interaction effects. Main effects are names of variables that must already exist in the dataset. They will be treated as categorical effects in the design matrix—the total number of parameters for each response function equals one less than the number of discrete values of the variable. For continuous variables, or variables with a large number of discrete levels, or variables that we simply want to treat as continuous, prefacing the variable name with "`direct.`" will enter the values directly into the design matrix, thus occupying only one column. Interaction effects

are specified with an asterisk between two (or more) variable names. There is no limit to the number of variables that can be included in an interaction effect. The type of interaction here is sometimes referred to as a crossed effect. In the design matrix, the values of each variable are multiplied together to form all possible permutations of the resulting design columns. Other interaction types such as nested interactions are not supported.

The `cmd_logreg` function does a fair amount of validation of the model syntax before passing control to the `mlelr` function which will actually perform the logistic regression procedure. In addition to checking the syntax, here is where a model structure will be initialized which translates the human friendly model specification into something more convenient on which the logistic regression procedure can operate. The header file `model.h` declares a `model` struct containing all the metadata about a model. In Section 3.6 above, we show the complete source of our `model` struct.

The parsing that takes place in the `cmd_logreg` function makes use of the `add_model_variable` function defined in `model.c`. This function is long and complicated, owing to the fact that we have allowed a lot of functionality in the model specification that then needs to be carefully translated into the model structure. The parsing logic begins by finding the dataset with the supplied handle. Next we initialize a model structure and add the dependent variable. Be aware that we only do rudimentary validation of the model specification. If a confusingly specified model is entered—for instance adding interactions involving variables that have not been entered as main effects or adding duplicate interactions—the results may be nonsensical.

Once parsing of the model specification is complete, we now have all the information needed to run the model in the `model` structure itself. Thus, instead of a long series of arguments, we now only need to pass two arguments: a pointer to the dataset, and a pointer to our custom model structure.

```
mlelr(ds, mod);
```

This function is declared in `mlelr.h` and defined in `mlelr.c`. The `mlelr` function is a long one, occupying over 500 lines including comments. In a general purpose statistical package it would make sense to break this functionality out into multiple different modules. Constructing the design matrix, for example, could be handled in its own code module to decouple it from the execution of any particular model. Printing the results is also something that should exist as a helper module, rather than being hard coded into this one big function. However, as we have made clear by now, this is not a general purpose statistical package, and we have been more than happy to make certain compromises in order for the program to stand alone as an easily understood and purpose built utility. Thus, even though a further layer of abstraction here may make the various components more extensible, we find it perfectly acceptable to leave this as a single monster function given that it is the central feature of the program.

Another reason the code was written in this monolithic manner is to parallel the outline provided in the companion article [1]. The `mlelr` function should appear very similar to the implementation sketch provided in the prior work. However, do be advised that there may be some minor notational differences between the code and the equations found in the original article.

In addition to the already complex model structure, `mlelr` declares a bewildering multitude of temporary variables on the stack. Some of them are commented, others' purpose will be obvious by their names, others you may need to hunt in the code to find out what they actually do, and some of them may actually not even be used at all and are remnants of functionality that was not included in the current version. At one point I considered factoring out a large chunk of these variables and putting them in a structure. Similar to the way the `model` structure encapsulates all the data required to run the model, we could have another structure that would encapsulate all the data generated by the running of the model. However, given that the requirements for this program do not include doing anything with the results of the model except printing them to the output window, I ultimately opted to leave these as temporary variables within the `mlelr` function. The function itself is organized around the following steps:

- Step 1. Build the crosstab as a precursor to the design matrix.
- Step 2. Count the number of populations and set a population index for each row in the xtab.
- Step 3. Count number of columns needed in X and Y.
- Step 4. Allocate space for model vectors and matrices.
- Step 5. Build X, Y, and n.
- Step 6. Build the interactions portion of X.
- Step 7. The Newton-Raphson loop.
- Denouement: Print the results.

Step 1. Build the crosstab as a precursor to the design matrix.

When the `mlelr` function is entered, we have a dataset and a model structure which tells us the dependent variable and the various types of independent variables—main effects, direct effects, and interactions. The first six steps of this function are devoted to setting up the design matrix, the most important structure in carrying out the actual modeling procedure. Although we can code the design matrix with as many rows as there are actual rows in the dataset, it is easier computationally to collapse common rows into a set of unique "populations", where each population represents a unique combination of the values of the independent variables. Each population will have a count equal to the sum of the weights of each observation that belongs to the population. For example, if we have a single independent variable with three unique values, the design matrix will only need to include three rows, regardless of the total sample size, because each observation will belong to one of those three populations.

See Section 2.1.1 of [1] for a description of the model specification for binary logistic regression, and Section 2.2.1 for a description of the model generalized to the multinomial case. In our `mlelr` function, we will build X as our design matrix having N rows and K

columns, where N is the number of populations and K is the number of columns needed in the design matrix to parameterize all the independent variables and their interactions.

Before we can begin building the design matrix, we will need several intermediate datasets which provide tabulations of all the variables to be used in the model. We can use our *tabulate* framework to generate these datasets and we have defined a special function `tabulate` which accesses the model structure directly and generates the tabulations we require. We accomplish this with the following simple function call from Step 1 in the `mlelr` function:

```
tabulate(ds, mod);
```

Note from the definition of the model structure in `model.h` that there are two special dataset pointers defined for each model which up to this point have not been allocated or set:

```
dataset *xtab;      /* cross-tabulation of all model variables */
dataset **freqs;    /* array of frequency tables for all model variables */
```

It is the tabulate function's job to build these datasets. The `xtab` dataset is a tabulation of all the independent variables plus the dependent variable. Thus, it will have one column for each independent variable, one for the dependent variable, and one for the count of observations at each level. Next, `freqs` is an array of datasets, one for each independent variable and one for the dependent variable. These datasets will have two columns: `Value` and `Freq`. We could tabulate each of these variables individually by reusing the `frequency_table` function already defined, but this would involve traversing the input dataset multiple times. We want to do this in one pass over the dataset, so we loop through each observation and then loop through each variable—all independent variables plus the dependent variable—building the individual variable tabulations in the `freq` dataset array. As we loop over each variable, we also store the current observation's value in an array called `obs`. When we have completed the loop over each variable, and our `obs` array now represents the full set of variables for the current row in the dataset, we use the `find_observation` function defined in `dataset.c` to search for this observation in the `xtab` dataset. If the observation is found in `xtab`, then we increment the count (by the value of `weight`), otherwise we add a new observation to `xtab`.

Note that we do not provide a way to handle negative or zero weights, so we will ignore the entire observation unless there is a positive weight. When `tabulate` is finished, the `xtab` and `freqs` datasets are now ready for `mlelr` to begin counting the number of populations on our way to building the design matrix.

To simplify the code and reduce the level of pointer indirection, we set the following temporary variables to refer to various aspects of the `xtab` dataset:

```
/* for convenient reference */
xtab = mod->xtab->obs;
xtabrows = mod->xtab->n;
xtabcols = mod->xtab->nvars;
```

Step 2. Count the number of populations and set a population index for each row in the xtab.

The `xtab` dataset that was generated by the `tabulate` function does not directly provide us with the design matrix, but provides us all the information we need to build it. Note that the `xtab` dataset also contains the dependent variable, so the populations as we have defined them to be unique combinations of the independent variable values will appear multiple times in `xtab`, potentially once for each response function (level of the dependent variable). We want to count and identify the unique populations in `xtab`, so to do this we use an integer array called `popindex`:

```
popindex = (int *) emalloc(xtabrows * sizeof(int));
```

In Step 2, we loop through `xtab` and maintain a running count of the number of unique populations represented therein. For each row in `xtab`, we store its population index in the `popindex` array. When we are done with Step 2, we now have a mapping to the population index of each row in `xtab`. We also know `N`, the number of populations, and `M`, the total sample size—the sum of the observation weights in each row of `xtab`.

Step 3. Count number of columns needed in X and Y.

Now we can count the number of columns needed in `X` and `Y`. `Y` will have `J` columns, which is the number of values of the dependent variable, also referred to as the number of response functions.

```
/* number of cols in Y */
J = mod->freqs[mod->numiv]->n;
```

If there are 3 independent variables in the model, `freqs` will be an array of 4 datasets, the last one, which can be indexed `mod->numiv`, will refer to the dependent variable. `X` will have `K` columns. The algorithm for determining `K` can be summarized like this:

```
K <- 1
For each independent variable do:
    If the variable is specified as a direct effect:
        K <- K + 1
    Else:
        K <- K + (the number of values minus 1)
For each interaction do:
    k <- 1
    For each variable in the interaction do:
        If the variable is not specified as a direct effect:
            k <- k * (the number of values minus 1)
    K <- K + k
```

We begin with one column for the intercept. Each independent variable will contribute one column if it is specified as a direct effect, otherwise it will contribute one fewer than the

number of distinct values of the variable, which we can obtain as the number of observations in the `freq` dataset for the variable. Interaction terms are multiplicative, so if we have two terms in an interaction both having three distinct values, the interaction will contribute 4 columns $(3-1)*(3-1)$.

Step 4. Allocate space for model vectors and matrices.

In this step we allocate space for `X`, `Y`, and `n`, the primary structures needed for the Newton-Raphson iterative procedure. `X` and `Y` are matrices and `n` is an array, all of which have `N` rows. We also allocate space for several convenience arrays that will be useful in the next step.

Step 5. Build X, Y, and n.

Here we loop through each row in `xtab`. For each new population encountered, we build the design matrix columns for all main effects (interactions are saved for the next step). The first column is always the intercept and it will always have a value of 1. Next, we parameterize each main effect. If the effect is specified as a direct effect, we take the actual value from `xtab` and use it in the design matrix. Remember this is used for continuous variables which will only have one parameter estimate for each response function. For categorical effects, we have two options for parameterization: the default is full-rank center-point and as an option (see Section 5.8) we can use dummy coding instead. In both schemes, we look at the number of levels of the variable and allocate one fewer columns for it in the design matrix. Full-rank center-point parameterization codes a 1 for the target column and 0 for other columns except that the reference value is coded with -1. In dummy coding, all columns besides the target are coded 0. To visualize this, consider a variable with 4 levels, ranging from 1 to 4. Table 2 illustrates the two ways each value would be coded.

Table 2: Parameterization options

Value	Full-rank center-point	Dummy coding
1	1, 0, 0	1, 0, 0
2	0, 1, 0	0, 1, 0
3	0, 0, 1	0, 0, 1
4	-1, -1, -1	0, 0, 0

Choice of parameterization will not affect the resulting predicted probabilities, it is mainly used to assist in interpreting the coefficients. In Step 5, we also store the counts for each response function in `Y` as well as the overall count of observations per population in `n`.

Step 6. Build the interactions portion of X.

We are almost there, but not quite ready. We have coded the main effects portion of the design matrix but still need to handle the interactions. This is highly tricky and we need to make use of several temporary helper variables in order to build the interactions. Earlier we defined several arrays to assist us here:

```
int     *startcol;  /* starting location of each iv in X */
int     *colspan;   /* number of columns occupied by each iv in X */
int     *intcolidx; /* interaction column index */
```

These are used in coordination with the interaction specific variables from the model structure:

```
int     numints;   /* number of interactions */
int     maxints;   /* space allocated for interactions matrix */
int     *inttc;    /* interaction term count */
int     **ints;    /* interactions matrix, rows=interactions, cols=index in numiv */
```

The `startcol` and `colspan` arrays were previously allocated in Step 4 to store an `int` for each independent variable: `startcol` identifies the column location in `X` for the starting location of the variable and `colspan` is a count of the number of columns in `X` that are spanned by the variable. For example, if the first variable after the intercept has 4 levels, it will have a `startcol` = 1 and `colspan` = 3. The third array `intcolidx` (interaction column index) will be used to store a counter for the current column being used for each interaction term. It is allocated to store enough space for an `int` for the interaction with the largest interaction term count.

At the start of step 4, we loop through the independent variables and set `startcol` and `colspan` for each. We also increment `xc` which, at the end of the `for` loop, will point to the first column in the design matrix where interactions begin.

```
xc = 1;
for (i = 0; i < xtabcols - 2; i++) {
    startcol[i] = xc;
    if (mod->direct[i])
        xc += 1;
    else
        xc += mod->freqs[i]->n - 1;
    colspan[i] = xc - startcol[i];
}
```

Next we begin a loop for each interaction. We first initialize the `intcolidx` array to 1 for each interaction term in the current interaction.

```
/* do for each set of interactions */
for (i = 0; i < mod->numints; i++) {

    /* setup a counter for each term */
    for (j = 0; j < mod->inttc[i]; j++)
        intcolidx[j] = 1;
```

Next we will begin a loop which accomplishes the following. For each row in the design matrix, get the values corresponding to the current interaction column index for each variable (initially equal to `startcol`) involved in this interaction. Multiply those values together and store the result in the current column, indexed by `xc`, of the design matrix corresponding to this interaction term.

```
q = 1;

/* construct each column of the interaction */
while (q) {

    /* multiply and write the current column */
    for (j = 0; j < N; j++) {
        tgt = 1.0;
        for (k = 0; k < mod->inttc[i]; k++)
            tgt *= X[j][startcol[mod->ints[i][k]] + intcolidx[k] - 1];
        X[j][xc] = tgt;
    }
```

Next we will test for another available column for this interaction by incrementing the `intcolidx` of the last term, until we have iterated through all columns for the last term at which point we start over and increment the `intcolidx` of the next to last term. In this way, we construct each interaction term by exhausting the available columns of each interaction term from right to left.

```
    /* move to next column in X */
    xc += 1;

    /* test for another column, starting with the last variable */
    q = 0;
    for (j = mod->inttc[i] - 1; j >= 0; j--) {
        if (!q) {
            intcolidx[j] += 1;
            if (intcolidx[j] > colspan[mod->ints[i][j]])
                intcolidx[j] = 1;
            else
                q = 1;
        }
    }
```

To visualize what is happening here, consider an interaction with three terms: the first two terms have 3 levels each and the last term has 4 levels. Let's say they are also the first three variables in the list of independent variables. Then, we will have the following data:

```
inttc[i] = 3
startcol[0] = 1
startcol[1] = 3
startcol[2] = 5
colspan[0] = 2
colspan[1] = 2
colspan[2] = 3
```

The first time through the loop, each `intcolidx` is set to 1. So, we obtain `tgt` by multiplying the values in the `startcol`s of each variable; 1, 3, and 5. Next, in our test for another column, we increment `intcolidx[2]` to 2. Since this is less than the value of `colspan[2]`, the loop continues and we obtain `tgt` for the next `xc` column by multiplying

the values in columns 1, 3, and 6. Next, we will get columns 1, 3, and 7 at which point the last term has reached its `colspan`, so `intcolidx[2]` will be reset to 1, the `for` loop will decrement `j`, and we will then increment `intcolidx[1]`. In this way we will get columns 1, 4, and 5 and continue the same way until we again get to the point where we exhaust the `colspan` for the final term and then increment `intcolidx[0]`. When the loop is complete for this interaction, we will have constructed 12 columns for this interaction term ($2 * 2 * 3$).

Step 7. The Newton-Raphson loop.

Finally we have built our design matrix, `X`, and are ready to prepare for Newton-Raphson iteration to solve the equations necessary to estimate the logistic regression model. At this point you are strongly urged to review the material in the companion article and be comfortable with the derivations of the equations, especially Eq. 20, Eq. 23, and Eq. 40, which express the matrix form of each step of the Newton-Raphson method.

To prepare for Newton-Raphson, we allocate space for several arrays that we will pass to the function. These include arrays to store the coefficients (betas), the matrix `xtwx` of second partial derivatives, and several additional arrays to store data used for significance testing of the parameters and the overall model log likelihood. We initialize the starting betas to 0 and call the `newton_raphson` function until we reach convergence (where new betas do not significantly differ from one iteration to the next) or until we reach a hard-coded limit for the maximum number of iterations we will allow before assuming the model has run away to a local maximum. We define `MAX_ITER` to be 30. In my experience, most models that do converge require fewer than 10 iterations to do so.

When the Newton-Raphson loop is complete, we conduct some significance tests and then print the results to the output. For more details on the output, see Section 6 in which we will walk through several examples provided with the source code. For more details on how the Newton-Raphson procedure works, see Section 7.

5.10 Other commands recognized by the parser

The parser will recognize a few more commands. `help` will print some help text, including a list of available commands, `q` or `quit` will tell the input handler to stop processing input and thus exit the program. The `#` symbol indicates a comment line that the parser should ignore. It will skip over the line and continue with the next line. Any other word or token at the beginning of a line that does not match a known command will print a warning but allow the user to continue.

6 Examples

In this section we will provide several examples of the functionality of `mlelr`. Scripts for these examples are provided in the distribution of `mlelr` and can be found in the 'tests' directory. They reference datasets that are saved in the 'data' directory. To run one of the example scripts, you can either enter each command at the `mlelr` interactive prompt, or run them in batch mode:

```
$ ./mlelr -f ../tests/alligator.txt
```

This assumes your current working directory is the 'src' directory which is where the `mlelr` executable file will be built by the `make` command.

6.1 Alligator dataset

The dataset of alligator food choice is one of the canonical examples used to test baseline category logistic regression. It is available in many locations online, the version we are using is from Agresti [9]. Note that if you find a version of this dataset elsewhere, there may be differences in the ordering of the levels of the categorical variables. This will result in the parameter estimates being mixed up, but will not affect the overall stability of the results.

This dataset has 80 observations including a count variable, as it is essentially a contingency table. It is delimited by spaces. There are 5 variables: lake, sex, size, food, and count. Let's import this dataset and print the first few observations:

```
mlelr-> import gator ../data/alligator.dat " "
Importing dataset from file: ../data/alligator.dat
Number of variables found: 5
Variable names: lake sex size food count
New dataset created with handle: gator
Number of observations read: 80
Import complete.
```

Since the dataset needs to be weighted by the count variable, we will set the weight:

```
mlelr-> weight gator count
Setting weight variable to: 'count' (4)
```

Let's print the first ten lines of the dataset to see what it looks like:

```
mlelr-> print gator 10
Dataset: gator
Number of observations: 80
Number of variables: 5
            lake             sex            size            food           count
            1.00            1.00            1.00            1.00            1.00
            1.00            1.00            1.00            2.00            0.00
            1.00            1.00            1.00            3.00            0.00
            1.00            1.00            1.00            4.00            5.00
            1.00            1.00            1.00            5.00            7.00
            1.00            1.00            2.00            1.00            0.00
            1.00            1.00            2.00            2.00            0.00
            1.00            1.00            2.00            3.00            1.00
            1.00            1.00            2.00            4.00            2.00
            1.00            1.00            2.00            5.00            4.00
```

The first three are independent variables. Lake is one of 4 lakes where these alligators were found, sex is male or female, and size is a two category variable for small vs. large. The dependent variable, food, indicates one of 5 food choices of the alligator. Next, let's run a tabulation of our dependent variable, food.

```
mlelr-> table gator food
Dataset: Frequency table for: food
Number of observations: 5
Number of variables: 2
           Value            Freq
            1.00           61.00
            2.00           19.00
            3.00           13.00
            4.00           32.00
            5.00           94.00
```

Now, we would like to predict the food choice of the alligators based on lake and size. Here is our logistic regression model:

```
mlelr-> logreg gator food = lake size
```

This is a simple model and a small dataset so it will execute very quickly. Considering all the time you may have spent digesting the material in this document, it is a rather awesome feat to see how readily the computer provides the answer. There are certain tasks where humans are better than computers, but maximum likelihood estimation of logistic regression models is not one of them! At the top of the output is some basic information about the model we just specified:

```
============================================================
  Maximum Likelihood Estimation of Logistic Regression Model
============================================================

Model Summary
==============
Dependent variable: food
Number of independent variables: 2
Effect 1: lake
Effect 2: size
Number of interactions: 0
Number of populations: 8
Total frequency: 219.000000
Response Levels: 5
Number of columns in X: 5
```

Following this is a frequency table of the dependent variable and a cross-tabulation of all the variables in the model (this is actually the `xtab` dataset from the model structure that was generated by our `tabulate` function). We also print out the design matrix, in very simplified form:

```
Design Matrix (all values rounded)
==================================
   1     1     0     0     1
   1     1     0     0    -1
   1     0     1     0     1
   1     0     1     0    -1
```

```
1     0     0     1     1
1     0     0     1    -1
1    -1    -1    -1     1
1    -1    -1    -1    -1
```

Here it is easy to see that the first column represents the intercept, and that we are using full-rank center-point parameterization. Since lake has 4 levels, the second, third, and fourth columns specify the effects for lake, and the fifth column specifies the effect for size. Next we get a report of the model convergence and model fit criteria:

```
Model Results
==============
Number of Newton-Raphson iterations: 7
Convergence: YES

Model Fit Results
==================
Test 1:  Fitted model vs. intercept-only model
Initial log likelihood: -129.940567
Final log likelihood:   -47.513803
Chisq value:   164.8535, df:    14, Pr(ChiSq):   0.0000

Test 2:  Fitted model vs. saturated model
Deviance: 17.079831
Chisq value:    17.0798, df:    12, Pr(ChiSq):   0.1466
```

Finally, we have the parameter estimates along with their standard error, the Wald chi-squared value, and the significance results of the Wald chi-squared test.

```
Maximum Likelihood Parameter Estimates
======================================
           Parameter  DV    Estimate   Std Err  Wald Chisq  Pr > Chisq
           Intercept   0 -0.71970490    0.2109     11.6420      0.0006
           Intercept   1 -1.83093861    0.3398     29.0275      0.0000
           Intercept   2 -2.12598750    0.3654     33.8562      0.0000
           Intercept   3 -1.15144200    0.2343     24.1445      0.0000
                lake   0 -1.75856999    0.4371     16.1897      0.0001
                lake   1 -0.41644885    0.5589      0.5552      0.4562
                lake   2  0.41269843    0.5115      0.6509      0.4198
                lake   3  0.23914171    0.3458      0.4784      0.4892
                lake   0  0.83700793    0.3260      6.5919      0.0102
                lake   1  0.79964649    0.4710      2.8822      0.0896
                lake   2 -0.93562692    0.8149      1.3183      0.2509
                lake   3 -0.58140144    0.5061      1.3198      0.2506
                lake   0  1.02177344    0.3385      9.1111      0.0025
                lake   1  1.27602784    0.4677      7.4446      0.0064
                lake   2  0.80534763    0.5424      2.2044      0.1376
                lake   3  0.92931423    0.3836      5.8702      0.0154
                size   0  0.72910231    0.1980     13.5634      0.0002
                size   1 -0.17563142    0.2900      0.3667      0.5448
                size   2 -0.31532987    0.3212      0.9635      0.3263
                size   3  0.16577513    0.2241      0.5471      0.4595
```

You can verify that these estimates are identical to those found in Agresti's Table 9.4 on p. 311 of [9].

6.2 Ingots dataset

The ingots dataset is originally from an article by Cox and Snell and is available from the SAS website at [10][10]. This dataset has 4 variables and is tab-delimited. We being by importing the dataset:

```
mlelr-> import ingots ../data/ingots.dat "\t"
Importing dataset from file: ../data/ingots.dat
Number of variables found: 4
Variable names: heat soak r n
New dataset created with handle: ingots
Number of observations read: 38
Import complete.
```

This dataset also has a count variable, so we will weight by the variable n:

```
mlelr-> weight ingots n
Setting weight variable to: 'n' (3)
```

The first two variables in the dataset, heat and soak, are continuous variables while the dependent variable, r, is binary.

[10] link: `http://support.sas.com/documentation/cdl/en/statug/63962/HTML/default/viewer.htm#statug_catmod_sect042.htm`

```
mlelr-> table ingots r
Dataset: Frequency table for: r
Number of observations: 2
Number of variables: 2
           Value            Freq
            0.00          375.00
            1.00           12.00
```

This dataset is used as an example test for the CATMOD procedure in SAS. Since we want to treat heat and soak as direct effects, we will specify the model like this:

```
mlelr-> logreg ingots r = direct.heat direct.soak
```

First we can review the model summary at the top of the output:

```
============================================================
  Maximum Likelihood Estimation of Logistic Regression Model
============================================================

Model Summary
==============
Dependent variable: r
Number of independent variables: 2
Effect 1: heat (DIRECT)
Effect 2: soak (DIRECT)
Number of interactions: 0
Number of populations: 19
Total frequency: 387.000000
Response Levels: 2
Number of columns in X: 3
```

The full output contains the frequency table for the dependent variable, the cross-tabulation of all model variables, and the simplified design matrix. We see from the model results that the model converged in 8 iterations.

```
Model Results
==============
Number of Newton-Raphson iterations: 8
Convergence: YES

Model Fit Results
==================
Test 1:  Fitted model vs. intercept-only model
Initial log likelihood: -234.615310
Final log likelihood:   -14.040158
Chisq value:   441.1503, df:     0, Pr(ChiSq):   0.0000

Test 2:  Fitted model vs. saturated model
Deviance: 13.752628
Chisq value:    13.7526, df:    16, Pr(ChiSq):   0.6171
```

```
Maximum Likelihood Parameter Estimates
======================================
          Parameter  DV    Estimate   Std Err  Wald Chisq  Pr > Chisq
          Intercept   0  5.55916646    1.1197     24.6502      0.0000
               heat   0 -0.08203080    0.0237     11.9452      0.0005
               soak   0 -0.05677131    0.3312      0.0294      0.8639
```

Note that the parameter estimates match those reported in the output from PROC CATMOD in the SAS example.

6.3 UCLA dataset

The UCLA dataset contains several variables for predicting admittance to graduate school and is available from [11]. There are 4 variables: admit, gre, gpa, and rank, and the dataset is tab-delimited. First we will import the dataset and print the first ten observations.

```
mlelr - a reference implementation of logistic regression in C
version: 1.0
mlelr-> import ucla ../data/ucla.dat \t
Importing dataset from file: ../data/ucla.dat
Number of variables found: 4
Variable names: admit gre gpa rank
New dataset created with handle: ucla
Number of observations read: 400
Import complete.
mlelr-> print ucla 10
Dataset: ucla
Number of observations: 400
Number of variables: 4
           admit             gre             gpa            rank
            0.00          380.00            3.61            3.00
            1.00          660.00            3.67            3.00
            1.00          800.00            4.00            1.00
            1.00          640.00            3.19            4.00
            0.00          520.00            2.93            4.00
            1.00          760.00            3.00            2.00
            1.00          560.00            2.98            1.00
            0.00          400.00            3.08            2.00
            1.00          540.00            3.39            3.00
            0.00          700.00            3.92            2.00
```

From this output, the variables gre and gpa appear to be continuous, while admit looks binary and rank ranges from 1 to 4. We can confirm this by tabulating the two categorical variables.

```
mlelr-> table ucla admit
Dataset: Frequency table for: admit
Number of observations: 2
Number of variables: 2
```

```
           Value            Freq
            0.00          273.00
            1.00          127.00
mlelr-> table ucla rank
Dataset: Frequency table for: rank
Number of observations: 4
Number of variables: 2
           Value            Freq
            1.00           61.00
            2.00          151.00
            3.00          121.00
            4.00           67.00
```

Next, we want to run the same model used in the SAS example at [11]. Both gre and gpa are continuous and we don't want to treat them as categorical so we will specify them as direct effects in the model. For rank, however, we will treat this as a categorical variable. Since the SAS example is using PROC LOGISTIC which uses dummy coding by default (as opposed to full-rank center-point parameterization used by PROC CATMOD), in order for the parameter estimates for our model to match, we need to turn on our dummy coding option before running the model.

```
mlelr-> option params dummy
mlelr-> logreg ucla admit = direct.gre direct.gpa rank
```

The model runs and we see the summary at the top of the output:

```
============================================================
  Maximum Likelihood Estimation of Logistic Regression Model
============================================================

Model Summary
==============
Dependent variable: admit
Number of independent variables: 3
Effect 1: gre (DIRECT)
Effect 2: gpa (DIRECT)
Effect 3: rank
Number of interactions: 0
Number of populations: 391
Total frequency: 400.000000
Response Levels: 2
Number of columns in X: 6

Frequency Table for Dependent Variable
======================================
           Value            Freq
            0.00          273.00
            1.00          127.00
```

Note that the number of populations, 391, is very nearly equal to the total sample size of 400. This is a consequence of working with continuous variables, and also illustrates how

foolish it would be to treat gre and gpa as categorical variables—the resulting design matrix would be far too sparse and would very likely be unable to result in a stable model.

Note also our coding on the dependent variable is 0 and 1. In `mlelr` we have hard-coded the baseline category to be the highest value, in this case 1. For binary dependent variables, the intent is typically to model the probability of success, usually coded as 1. However, if 1 is treated as the reference category, the parameter estimates will actually be reversed in sign vs. what one would expect. One way to get around this annoyance is to code binary dependent variables using a 1 vs. 2 scheme. This way, success = 1 will be modeled vs. the omitted category of not success = 2. Note that in the PROC LOGISTIC example, they have used the 'descending' option. This will effectively reverse the choice of reference category, so instead of modeling the probability of 0 vs. 1, the results will be reflecting the probability of 1 vs. 0.

Because of the continuous variables, the crosstab of all model variables and the design matrix occupy a lot of space in the output. In a future version of `mlelr` it would be nice to specify an option to suppress this output if it isn't wanted. In any case, here are the results of our logistic regression model along with the parameter estimates:

```
Model Results
==============
Number of Newton-Raphson iterations: 6
Convergence: YES

Model Fit Results
==================
Test 1:  Fitted model vs. intercept-only model
Initial log likelihood: -274.080818
Final log likelihood:   -226.080692
Chisq value:    96.0003, df:     3, Pr(ChiSq):   0.0000

Test 2:  Fitted model vs. saturated model
Deviance: 446.380641
Chisq value:   446.3806, df:   385, Pr(ChiSq):   0.0167


Maximum Likelihood Parameter Estimates
======================================
           Parameter  DV    Estimate   Std Err  Wald Chisq  Pr > Chisq
           Intercept   0  5.54144275    1.1381     23.7086      0.0000
                 gre   0 -0.00226443    0.0011      4.2843      0.0385
                 gpa   0 -0.80403755    0.3318      5.8715      0.0154
                rank   0 -1.55146368    0.4178     13.7873      0.0002
                rank   0 -0.87602075    0.3667      5.7059      0.0169
                rank   0 -0.21125976    0.3929      0.2892      0.5907
```

Comparing these to the results shown in [11], you'll notice that the signs on the estimates are reversed, but otherwise identical. This is due to the 'descending' option used in PROC LOGISTIC to reverse the order in which the baseline category would be selected. The estimates are the same in magnitude because the dependent variable is binary, all we are doing is reversing the interpretation of the resulting model. When the dependent variable

has more than two levels, the parameter estimates would differ in magnitude as well because we would be using a completely different reference category. Note that this is only a matter of interpretation using the parameter estimates. In either case, if you were to look at odds ratios or the predicted probabilities by solving the regression equation, the results would be the same whether you use the lowest or the highest category as the baseline.

7 The Newton-Raphson Method

In a logistic regression model, we equate the logit transform, the log-odds of the probability of some event, to a linear combination of predictor variables weighted by parameter coefficients. In maximum likelihood estimation, we seek to find the values for the coefficients that maximize the likelihood of the observed data occurring. The likelihood function is equivalent to the joint probability density function of the multinomial distribution, except that while the pdf expresses the probability of the observed data as a function of the parameter coefficients, the likelihood function expresses the probability of the parameter coefficients as a function of the observed data.

To find the set of values for the coefficients which maximize the likelihood function, we need to evaluate the first and second derivatives of the likelihood function. A maximum occurs where the first derivative is equal to zero and the second derivative at that point is negative. The likelihood function itself is difficult to work with, however, the log likelihood function is a valid substitute since the log transform will not affect the location of critical points.

The Newton-Raphson method is a root-finding method, it is used to find the values x for which $f(x) = 0$. In our case, the function whose roots we want to find is the first derivative of the log likelihood function, and the values x are the parameter coefficients. Recall from [1] that each iteration of Newton-Raphson will apply Eq. 23. In solving this system of equations, we will make use of Eq. 32 for the first derivative, Eq. 37 for the second derivative, and will use the identity in Eq. 40 which is equivalent to Eq. 23 but is a more computationally stable approach.

To be strictly precise, our `newton_raphson` function is really only responsible for one iteration of the method. The complete Newton-Raphson method is more accurately represented by the complete loop in which we iterate, retrieve the new values for the coefficients, and test for convergence. Note that while $\boldsymbol{Y}$ is our response matrix, and $\boldsymbol{\beta}$ is also technically a matrix of the same dimensions, we will treat $\boldsymbol{\beta}$ as a column vector with each set of response functions stacked on top of one another. This makes it easier to construct the matrix of second derivatives which would otherwise need to be formulated in three dimensions if we consider $\boldsymbol{\beta}$ as a matrix.

```
static int newton_raphson (
    double **X,     /* design matrix, N rows by K cols */
    double **Y,     /* response matrix, N rows by J-1 cols */
    double  *n,     /* vector of population counts, N rows */
    int      J,     /* number of discrete values of Y */
    int      N,     /* number of populations */
    int      K,     /* number of columns in X */
    double  *beta0, /* starting parameters, K * J-1 rows */
```

```
double  *beta1, /* parameters after this iteration */
double **xtwx,
double  *loglike,
double  *deviance  );
```

In the loop from `mlelr`, we call each iteration like this:

```
nrret = newton_raphson(X, Y, n, J, N, K, beta0, beta, xtwx, loglike, deviance);
```

For each iteration of `newton_raphson`, we pass as argument `beta0` the starting values of the coefficients, and we return the values from the new iteration in `beta1`. We will also update the matrix of second partial derivatives, `xtwx`, which will be used in the calling function to obtain the standard errors of the betas. Our function begins by allocating memory for some arrays we will use for temporary purposes or intermediate results. This includes a matrix `pi` having the same dimensions as the response matrix `Y`. `pi` holds the predicted probabilities for each population, and each response function. We then initialize our temporary variables to 0. This also includes two important temporary arrays that we will rely upon to do most of the work of the function:

```
double  *g;     /* gradient vector: first derivative of ll */
double **H;     /* Hessian matrix: second derivative of ll */
```

Our intent is to solve for `beta1` using the derivation in Eq. 40. To do so we will compute the second term by calculating $\boldsymbol{X}^T(\boldsymbol{y}-\boldsymbol{\mu})$ and saving the results in our gradient vector, `g`. Then we will matrix multiply `H` by `beta0` to get the first part of the second [bracketed] term in Eq. 40. Finally, we will invert `H` and solve for the new betas by matrix multiplication of the newly inverted `xtwx` with the second term.

We begin by iterating through each population in the design matrix. Recall that our system of equations involves one equation for each design effect K. Thus, in our loop, we are calculating and accumulating the results for one equation at a time[11].

```
/* main loop for each row (population) in the design matrix */
for (i = 0; i < N; i++) {
```

Our first task is to calculate the predicted probabilities π_{ij} for the current population i and each response function j. Refer to Eq. 25 for the appropriate formula. Note that we first need to calculate the the sum of $\boldsymbol{X\beta}$, using the design matrix values for $\boldsymbol{X}$ and the initial values of `beta0` for $\boldsymbol{\beta}$.

```
    /* matrix multiplication of one row of X * Beta */

    denom = 1.0;
    jj = 0;

    for (j = 0; j < J - 1; j++) {
```

[11] If you are used to thinking about these equations in a convenient vectorized environment like that provided by R, Python, or Mathematica, you'll have to forget about that now. In C, we do linear algebra the old fashioned way!

```
        sum1 = 0;
        for (k = 0; k < K; k++)
            sum1 += X[i][k] * beta0[jj++];
        numer[j] = exp(sum1);
        denom += numer[j];
    }

    /* calculate predicted probabilities */
    for (j = 0; j < J - 1; j++)
        pi[i][j] = numer[j] / denom;

    /* omitted category */
    pi[i][j] = 1.0 / denom;
```

We use `jj` as a secondary counter to keep track of the index in our `beta0` array which as noted above is formulated as a stacked array instead of a matrix. In the loop over `j`, we only need to accumulate the sums of $\boldsymbol{X\beta}$ for the first $J-1$ response functions, because the probabilities for the omitted category can be determined once these are known. We store the numerator for each response function in Eq. 25 in `numer[j]`. The denominator starts with 1 and is then incremented by each numerator as we iterate through the levels of `j`.

Next, we increment the contribution to the log likelihood function from the current population. See Eq. 38 and Eq. 39. We do the same to accumulate the deviance from this population. These values are used in goodness of fit tests reported in the output if the model converges. Note that this involves our dependency on the gsl library for the implementation of the *log gamma* function. Although this is not strictly necessary to complete the logistic regression, it is a rather essential feature to be able to measure these goodness of fit tests, so we consider it important enough to include this functionality even if it means reliance on an external library[12].

Next we will want to increment the current population's contribution to the first and second derivatives. This loop will be performed once for each response function, except for the baseline category.

```
    /* increment first and second derivatives */
    for (j = 0, jj = 0; j < J - 1; j++) {
```

The following code will store in `q1` the parts of Eq. 32 that will later be multiplied by `X`.

```
        /* terms for first derivative, see Eq. 32 */
        q1 = Y[i][j] - n[i] * pi[i][j];
```

Similarly, we store in `w1` some terms that will be used in Eq. 37.

```
        /* terms for second derivative, see Eq. 37 */
        w1 = n[i] * pi[i][j] * (1 - pi[i][j]);
```

[12]The gsl is an excellent free resource and should be very easy to install on your development system. See Section 8.5 for details

This next loop is very similar to what appears in the sample code from [1] on p. 21. We increment `g` by multiplying `q1` with `X`, thus completing Eq. 23. Then, using the dual formulation for matrix W as shown in Eq. 37, we complete the multiplications required to build `H`.

```
for (k = 0; k < K; k++) {

    /* first derivative term in Eq. 23 */
    g[jj] += q1 * X[i][k];

    /* increment the current pop's contribution to the 2nd derivative */

    /* jprime = j (see Eq. 37) */

    kk = jj - 1;
    for (kprime = k; kprime < K; kprime++) {
        kk += 1;
        H[jj][kk] += w1 * X[i][k] * X[i][kprime];
        H[kk][jj] = H[jj][kk];
    }

    /* jprime != j (see Eq. 37) */

    for (jprime = j + 1; jprime < J - 1; jprime++) {
        w2 = -n[i] * pi[i][j] * pi[i][jprime];
        for (kprime = 0; kprime < K; kprime++) {
            kk += 1;
            H[jj][kk] += w2 * X[i][k] * X[i][kprime];
            H[kk][jj] = H[jj][kk];
        }
    }
    jj++;
}
```

This aspect of the algorithm is definitely the most difficult to grasp; however, once you master it all the remaining pieces will fall neatly into place. I highly recommend sketching out an example on paper or in a spreadsheet. The alligator example is a great case study because the dataset is small enough to manage, and you won't need to run through all iterations—but studying the loops a few times will help you get a feel for how we go about solving Eq. 40.

The alligator dataset has 8 populations N, and the design matrix with the example model using lake and size has 5 columns K. The dependent variable, food, has 5 levels, so there are 5 response functions J. The array of coefficients $\boldsymbol{\beta}$ has 20 elements, and $\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}$ is a 20×20 square matrix. The `beta0`, `beta1`, and `g` arrays in our code are organized such that all the design effects for the first response function appear first, followed by the effects for each subsequent response function (up to $J - 1$). So the first 5 elements are for the 5 design columns K for the first response function, where food = 1. Then the next 5 are for food = 2, etc. The matrices `xtwx` and `H` are organized the same way. Each element is

the second partial derivative of each beta coefficient taken with respect to each other beta coefficient.

When we begin the first iteration of Newton-Raphson, all the values for `g` and `H` are initialized to 0. The main loop progresses once for each of the 8 populations, indexed by i. The matrix `pi` has the same dimensions as `Y`, 8 rows (one for each population) and 5 columns (one for each response function of the dependent variable). Each element expresses the probability that an observation in population i will take on the value j of the dependent variable. Starting with the first population where i = 0, since all the betas are initialized to 0, all `pi[0][j]` will be the same, in this case 0.2.

Within each population, we enter a loop over j and another over k. The first loop is for the first $J-1$ response functions, the second is for the K columns in the design matrix. In the alligator dataset, this means we will loop a total of 20 times, once for each of the 20 elements of $\boldsymbol{\beta}$. This allows us to increment the value of `g`, which will store the terms needed to express the first derivative of the log likelihood function as formulated in Eq. 32. Within the loop for k, we will then setup another loop over k' in which we progressively build the values in the `H` matrix.

The 'Hessian matrix' $\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}$ is symmetric, so once we know the value for row 1 column 2, we also know the value for row 2 column 1. This matrix is also constructed differently to correspond to the two formulations in Eq. 37. Since each element of this matrix is the second partial derivative of the log likelihood function, each β_{kj} is differentiated with respect to all other coefficients, denoted $\beta_{k'j'}$.

As noted above, the array of coefficients $\boldsymbol{\beta}$ is organized with the K coefficients for the first response function, then the K coefficients for the second response function, and so on. Thus, expressed as β_{kj}, it is indexed using the following pattern:

$$\begin{bmatrix}\beta_{00} & \beta_{10} & \beta_{20}\cdots\beta_{K0} & \beta_{01} & \beta_{11} & \beta_{21}\cdots\beta_{K1}\cdots\beta_{0J-1} & \beta_{1J-1} & \beta_{2J-1}\cdots\beta_{KJ-1}\end{bmatrix}^T$$

We include the ^{T} there because $\boldsymbol{\beta}$ is technically a column vector. Now visualize how $\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}$ is constructed, when we take the derivative of each of these elements against each other. Recall that each element can be expressed as: $\frac{\partial^2 l(\beta)}{\partial\beta_{kj}\partial\beta_{k'j'}}$. Here is what the matrix looks like:

$$\begin{bmatrix}
\frac{\partial^2 l(\beta)}{\partial\beta_{00}\partial\beta_{00}} & \frac{\partial^2 l(\beta)}{\partial\beta_{00}\partial\beta_{10}} & \cdots & \frac{\partial^2 l(\beta)}{\partial\beta_{00}\partial\beta_{KJ-1}} \\
\frac{\partial^2 l(\beta)}{\partial\beta_{10}\partial\beta_{00}} & \frac{\partial^2 l(\beta)}{\partial\beta_{10}\partial\beta_{10}} & \cdots & \frac{\partial^2 l(\beta)}{\partial\beta_{10}\partial\beta_{KJ-1}} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 l(\beta)}{\partial\beta_{KJ-1}\partial\beta_{00}} & \frac{\partial^2 l(\beta)}{\partial\beta_{KJ-1}\partial\beta_{10}} & \cdots & \frac{\partial^2 l(\beta)}{\partial\beta_{KJ-1}\partial\beta_{KJ-1}}
\end{bmatrix}$$

If you sketch out the full matrix, you will notice that $j' = j$ for a set of $J-1$ square sub-matrices along the diagonal, each of which has K dimensions. In these cells of `H`, we use the formulation of Eq. 37 where $j' = j$, and for all other cells, we use the alternate formulation where $j' \neq j$. In each pass of the loop over j and k—this will be a total of 20 loops in the alligator example—we update the jjth row of `H` and also, since the matrix is symmetric, the jjth column.

The counter `jj` is used to keep track of the current row in `H` on which we are operating, and the counter `kk` is used to keep track of the column. When we enter the loop, we set `kk` equal to `jj` because we only need to update the cells to the right of and below the current index. Continuing with the alligator example, when we first enter the loop, we start with `jj` and `kk` equal to 0. For the first 5 cells, $j' = j$, and for the remaining 15 cells, $j' \neq j$. Just before we loop again, we increment `jj`. At this point, we have updated the first row and column of `H`.

Next, when `jj` = 1, we will start at the second row and second column of `H`. There are 4 cells where $j' = j$ and the remaining 15 where $j' \neq j$. Now we have updated the second row and column of `H`. Each time we increment `jj`, we offset where in `H` we begin, so we have one less element to update. When we reach the last row, when `jj` = 19, there is only one cell left to update because the entire matrix up and to the left has already been updated.

After the loop over each population is complete, we have enough information to solve for Eq. 40. At this point, the 'gradient' or 'score' vector stored in `g` is equivalent to $\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{\mu})$. In matrix terms, this has K rows and $J - 1$ columns, but recall that we are "stacking" our $J - 1$ columns together to form one column vector represented by `g`. Our matrix `H`, equivalent to $\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}$, is a square matrix of order $K * (J - 1)$, and $\boldsymbol{\beta}^{(0)}$ is a column vector of length $K * (J - 1)$. When we multiply them together, the result is also a column vector of length $K * (J - 1)$. Here, we multiply `H` by `beta0` and add the result to `g`.

```
/* compute xtwx * beta0 + x(y-mu) (see Eq. 40) */
for (i = 0; i < K * (J - 1); i++) {
    sum1 = 0;
    for (j = 0; j < K * (J - 1); j++)
        sum1 += H[i][j] * beta0[j];
    g[i] += sum1;
}
```

Now, what we have in `g` is actually the complete second term in the first line of Eq. 40. Next, we need to invert $\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}$, which we do using the following steps:

```
/* invert xtwx */
if (cholesky(H, K * (J - 1))) return 11;
if (backsub(H, K * (J - 1))) return 12;
if (trimult(H, xtwx, K * (J - 1))) return 13;
```

These are standard algorithms for inversion of a square symmetric positive definite matrix taken from [12]. It is possible for the inversion process to fail and if any of these functions returns a non-zero value, it most likely means the model is lost. We do not handle this situation in `mlelr` but in a production environment it would be advisable to handle these return values appropriately in the calling function. If all goes well then our matrix `xtwx` now contains $[\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}]^{-1}$, so we can matrix multiply this with `g` to finish Eq. 40:

```
/* solve for new betas */
for (i = 0; i < K * (J - 1); i++) {
    sum1 = 0;
    for (j = 0; j < K * (J - 1); j++) {
        sum1 += xtwx[i][j] * g[j];
```

```
        }
        beta1[i] = sum1;
    }
```

Congratulations! Now anytime you run a logistic regression model, you know exactly what is happening.

8 Concluding Remarks

8.1 Comments

The code is generously littered with comments. Writing comments is really an integral part of the programming process—it greatly helps compose your thoughts and remind you what you are actually trying to do. The fact that it also serves to help others know what you did is a nice added bonus.

In the spirit of using this program as a teaching and learning exercise, the level of detail provided in the comments may be excessive at times. Our goal with commenting is to point out anything unusual, potentially controversial, or idiomatic that may require more than a quick glance to decipher the intent of the code. At the same time, some areas are sparsely commented because comments would only get in the way. In the Newton-Raphson routine for example, the reader is expected to be following along with the formulas and code outline described in the companion article, *Maximum Likelihood Estimation of Logistic Regression Models: Theory and Implementation* [1], and adding excessive levels of comments here would only detract from the flow of reading and understanding the code. Hopefully we have struck a good balance.

As well, we hope that comments add incremental understanding, rather than more confusion. It is often the case that comments fall out of synchronization with the code itself and end up being wrong or out of date.

8.2 Lines of code

Counting lines of code is not as straightforward a metric as it may sound and it is often employed as a flawed attempt at measuring the size or scope of a project. Different programming styles can lead to very different counts. Lines are after all a human convenience—the compiler doesn't care. If you strip all newlines out of your source code, the object files will still be the same size!

Still, the count of lines of code has its use and knowing that a program is one thousand lines versus one hundred thousand is a clear indication of the size of the respective projects. `mlelr` weighs in at about 3,000 lines including comments. This surprises many people and interestingly, some have suspected the number to be much greater while others have assumed it is a lot smaller. In reading this document you will clearly be able to appreciate that a large bulk of this code is devoted to the "scaffolding" while the actual implementation of logistic regression is really a page or two of actual code. Our `mlelr` function defined in `mlelr.c` runs to over 500 lines, but the first 300 or so are dedicated to all the precursory work involved in creating the design matrix, and a good part of the end of the function is

all about printing the output. The `newton_raphson` function is about 170 lines including comments.

So, all told, stripping out all the comments, window-dressing, memory handling, and data structure setup, the true kernel of code necessary to carry out a logistic regression would boil down to something around 100 or 200 lines.

8.3 Version numbering

The first public release version is numbered "1.0". Since this is a small project, the standard tripartite version numbering structure including major, minor, and point release numbers is not necessary. Any point releases that address bugs or minor maintenance issues found in this version will increment the secondary number. Development versions were numbered "0.0.1" with a postfix indicating alpha or beta stages and a letter sequence for unique reference. Any future version that adds substantial features will advance to "2.0".

Version numbers are a bit of programming vanity and one shouldn't get too worked up about them. They're not even the icing on the cake, more like gratuitous swirls of frosting on top of the icing on the cake.

8.4 Future work

What we have implemented here is a basic reference implementation of logistic regression using the Newton-Raphson iterative algorithm for solving the maximum likelihood equations. There are various alternatives to Newton-Raphson which may be explored in future work. These include Fisher scoring, iteratively re-weighted least squares, and gradient descent methods.

One feature mentioned in my companion article but not implemented here is the idea of using step-halving to correct for the possibility that the Newton-Raphson algorithm may begin to track and converge to a local maximum. We have omitted this feature here because it is not really a part of the core algorithm, but is rather an enhancement that can be employed in certain edge cases to "rescue" the procedure from a scenario in which it would result in a sub-optimal solution.

The test for parameters tending toward infinity is another area that is subject to potential improvement. We have not generated any test cases specifically to explore how this works and thus cannot really provide guidance on how best to handle such a situation.

Robust handling of interactions, the ability to handle different types of interactions (e.g. nested), would be another welcome enhancement.

The output format and precision is currently hard-coded. Parameter estimates are printed with 8 decimal places. This is an arbitrary choice. It should be sufficient in most applications, but in some cases having the full limit of double precision (roughly 15 decimal places) may be necessary to produce estimates that are not unduly perturbed by rounding error. Allowing the user to specify output precision is a feature worth considering.

Additional output details, or suppressing of output details, is another feature area which could be improved. As suggested earlier, it would be best if printing of the model output could be removed from the `mlelr` function and abstracted into a helper module which could allow the user to specify what to do with the output—what parts and whether to print it, an option to save results to a file, etc.—in a more flexible manner. Architecturally, it may

be more useful to have the `mlelr` function return an object that can then be referred to with user-accessible commands.

Exponentiating the results, i.e. solving the regression equation to generate predicted probabilities, or providing odds ratios would also make the output more useful.

Goodness of fit test results are currently difficult to interpret, and additional detail on what the output of this section means would be another good area for improvement.

8.5 Development environment

Development of `mlelr` has been principally conducted using a virtual machine running CentOS 6.5 with VirtualBox. This has allowed me to maintain a clean environment with only the minimal resources necessary to generate the code and to ensure the maximum level of portability by working in a sandbox of sorts which is unconstrained by and protected from any custom configurations or libraries running on my client machines.

To recreate the environment used in the development of `mlelr`, begin by downloading VirtualBox and the "Minimal" ISO image of CentOS. Create a new virtual machine of type "Linux" and Version "Red Hat (64 bit)". Under the System tab, allow 1024 MB for Base Memory, this will be sufficient since we will not be using X windows. Under Storage, configure a new virtual disk with 8 GB, and choose the option to allow the OS to allocate space as needed rather than reserving the entire 8 GB all at once[13].

Under Network, configure two adapters. Attach Adapter 1 to NAT. This will allow your virtualbox to use your host operating system's network connection using network address translation. This is necessary because you will want to download some packages, as well as the source for `mlelr`, and will need an internet connection to do so. Configure Adapter 2 as a Host-only adapter. This will create a new network vboxnet0 on your machine and will allow you to ssh from your host OS to your virtual box.

Next, in the Storage tab, under Controller: IDE, add a disk image and browse to the ISO image file of the CentOS Minimal ISO that you downloaded. When you start up the virtual machine, it will then boot into the CentOS installation as if you had inserted the install CD in your machine. When installing CentOS, provide a hostname, and be sure to configure the network interfaces to connect automatically. When partitioning, create a small 512 MB swap partition and allocate the remainder of the space to the '/' root partition, configured as ext4, or another filesystem of your own liking. When the installation is done, you can ssh into your virtualbox from a terminal on your host operating system, using the appropriate IP address assigned on your virtual network, e.g.:

```
ssh root@192.168.56.101
```

Next, you will want to run an update and add the packages necessary to compile and link mlelr:

```
yum update
yum install vim wget
```

[13] If you do not plan to add more packages and are running low on space, you can reduce this to 3 GB. In my virtualbox, df -h only reports usage of 1.3 GB, but you will need to allow some extra space for the swap partition and other file system overhead.

```
yum groupinstall development
yum install gsl gsl-devel
```

That last line will install the gsl library which is required for `mlelr` to run. Also it will be a good idea to use the `adduser` command to add a non-root user to login and do your work. At this point you can download the source for `mlelr`, untar, and make. Enjoy!

8.6 Thank you

Finally, I wish to thank everyone who has written to me in the past 12 years with comments on my original article and expressing interest in the program that has become `mlelr`. I am extremely pleased to release this to the community and in doing so hope that the time I have spent studying logistic regression can benefit others. I welcome any comments or questions.

If the degree of detail to which I descend in describing both the math and the code implementation appears too pedantic at times, rest assured that this was my foremost intention. I have always found texts and articles on the subject to gloss over the crucial details that my puny mind requires to reach a basic level of understanding. The dreaded "to be left as an exercise to the reader" accompanying high-level equation-laden summaries of statistical methods is to me a careless and arrogant abrogation of the responsibility of those who know to share that knowledge. Consider my work to be filling in those blanks that other treatments of the topic have left empty. If, on the other hand, you find that I am not being careful enough in detailing and explaining how this works, then please let me know so that I may correct the text and code to make it as universally useful as possible.

Lastly, do not be intimidated (or fooled) by the fact that I can produce such a lovely and long document as this. I assure you I barely understand this stuff myself! That is precisely why this needs to be so long-winded. I am an amateur, and we amateurs have a duty to stick together and help each other out. My expectation is that at this point one of my readers will now go to GitHub, fork `mlelr`, make it really shine, and then do something really amazing—cure cancer, abolish war, end poverty, forecast the opening weekend box office of Hollywood movies. But my principal aim here has been to help the reader understand how logistic regression works. If I have succeeded in accomplishing this humble aim, then I will consider this work to be a success.

And now I can turn my attention to another method that has captured my imagination ever since I first read Leo Breiman's work back in grad school: *random forests*. With a little time and luck perhaps I can give it the same treatment as I have here with logistic regression. Check back with me in, say another 12 years. In 2027, I'm sure I will still be an amateur, still fascinated by and eager to help explain the magic and mysteries of statistics, and yes, I will still be coding in C.

This document was typeset using LaTeX on February 2, 2015.

9 References

[1] Scott A. Czepiel. 2002. *Maximum Likelihood Estimation of Logistic Regression Models: Theory and Implementation.* `http://czep.net/stat/mlelr.html`.

[2] Donald E. Knuth. 1997. *The Art of Computer Programming*, vols. 1-3. Addison-Wesley.

[3] Al Kelley and Ira Pohl. 1998. *A Book on C*. 4th ed. Addison-Wesley.

[4] Brain W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language*. 2nd ed. Prentice Hall.

[5] Brian W. Kernighan and Rob Pike. 1999. *The Practice of Programming*. 1st ed. Addison-Wesley.

[6] Peter van der Linden. 1994. *Expert C Programming: Deep C Secrets*. 1st ed. Prentice Hall.

[7] *comp.lang.c Frequently Asked Questions*. `http://c-faq.com/index.html`.

[8] Free Software Foundation, Inc. 2014. *The GNU C Library*. Available from: `http://www.gnu.org/software/libc/manual/`.

[9] Agresti, A. 1990. *Categorical Data Analysis*. 1st ed. New York: John Wiley.

[10] SAS Institute Inc. 2015. *SAS/STAT User's Guide, Version 9.3*. Cary, NC: SAS Institute Inc.

[11] UCLA Statistical Consulting Group. *Introduction to SAS. UCLA: Statistical Consulting Group*. Available from: `http://www.ats.ucla.edu/stat/sas/dae/logit.htm`

[12] Golub, G.H. and Van Loan, C.F. 1996. *Matrix Computations*. 3rd ed. Baltimore: Johns Hopkins.