A Legofit Tutorial

Alan R. Rogers

July 28, 2022

Contents

1 Introduction

This is a tutorial on Legofit, a package that uses genetic data to estimate the history of population size, subdivision, and admixture [8, 9]. I’ve designed it for use in the AGAR Workshop on Computational Evolutionary Genetics, which will meet online July 27–29, 2022. There is much more in this document than we will use in the workshop. The extra material is there for those who want to use Legofit on their own, after the workshop is over. For the workshop itself, you will need sections 2, 6 (excluding 6.3.2), and 8. Further information about Legofit is available online.

Try to install Legofit (sec. 2) on your own before the workshop starts. We'll help with installation problems during the workshop, but that will take less time if you've already made a start.

2 Installing Legofit

This section explains how to get Legofit running on your own computer.

2.1 Install git if necessary

You may have git already. To find out whether you do, type:

```
type git
```

On Mac or Linux, this will print the location of the executable git file. If there is no such file, you'll get an error message saying `type: git: not found`. I recommend using homebrew to install packages on a Mac. To install git using homebrew, type:

```
brew install git
```

Otherwise, visit git's website and look for the download instructions.

2.2 Clone legofit

Use git to clone Legofit onto your machine. This step will create a subdirectory called "legofit," which will contain the source code. Before executing the command below, use the `cd` command to move into the directory where you keep source code that other people have written. I keep such code in a directory called `distrib`, which I originally created by typing `mkdir distrib`. Having created such a directory, move into it by typing

```
cd distrib
```

Then clone legofit by typing:

```
git clone https://github.com/alanrogers/legofit.git legofit
```

This will create a directory called legofit. Use `cd` to move into it.

2.3 Install a C compiler if necessary

To see whether you have a C compiler already, type `type cc`, or `type gcc`, or `type clang`. You should get output like

```
cc is /usr/bin/cc
```

If you need to install a C compiler, there are several alternatives. On a Mac, you can install Xcode from the App Store. Or you can use Homebrew to install clang or gcc:

```
brew install clang
```

or

```
brew install gcc
```

On Linux, the command would be

```
sudo apt-get install clang
```

or

```
sudo apt-get install gcc
```

2.4 Install “make” if necessary

You will also need the program “make,” so type “type make.” If you need to install, then

```
brew install make
```

2.5 Set up a “bin” directory to hold executable files.

On Unix-like operating systems (MacOS and Linux) it is conventional for each user to maintain a directory named “bin,” just below the home directory, which contains executable files. This directory must also be added to the system’s PATH variable, which is used for finding executable files.

First create a “bin” directory, if you don’t already have one. To do so, first type the command

```
cd
```

at the shell prompt. This moves you into your home directory. Then type

```
mkdir bin
```

This creates a new directory called “bin.”

You now need to add this to your shell’s PATH variable. I’ll assume you’re using the bash shell. In your home directory, look for a file called either “`.bash_profile`” or “`.profile`”. Because the name begins with “.”, it will not show up if you type `ls`. However, it will appear if you type `ls .bash_profile` or `ls .profile`. If only one of these files exists, open it with a text editor (not a word processor). If they both exist, edit “`.bash_profile`”. If neither exists, use the editor to create a new file called “`.bash_profile`”.

Within this file, you may find existing definitions of the PATH variable. Add the following after the last line that changes this variable:

```
export PATH=$HOME/bin:$PATH
```

Save this change and exit the editor.

The contents of this file are executed by the shell each time you log into your computer. Since you have just created the file, your PATH has not yet been reset. Log out and then log back in again, and your PATH should be set. To check it, type

```
echo $PATH
```

This will print your PATH.

2.6 Compile and install legofit

The details are in the Legofit documentation.

3 Setting up your environment at the CHPC

This section isn’t needed for the AGAR workshop, because we aren’t using the CHPC.

Utah’s CHPC has several “clusters,” each with many “nodes.” Each node is a powerful computer in its own right. The slurm version of the pipeline submits legofit jobs to multiple nodes so that they can run in parallel. Within each node, legofit parallelizes across all available cores. This greatly reduces the time required for an analysis.

To log into a CHPC cluster, your own computer must support ssh. To find out whether it does, type the command

```
type ssh
```

at the bash prompt. If ssh is on your system, the response will be something like

```
ssh is /usr/bin/ssh
```

If ssh is not on your system, the response will be

```
-bash: type: ssh: not found
```

In the latter case, install openssh by typing

```
brew install openssh
```

Now that you have access to ssh, you can log into one of the CHPC clusters by typing the following at the bash prompt. Either

```
ssh -l <your unid> notchpeak.chpc.utah.edu
```

or

```
ssh -l <your unid> kingspeak.chpc.utah.edu
```

or

```
ssh -l <your unid> lonepeak.chpc.utah.edu
```

Here, `<your unid>` is your own University of Utah id. Each of these commands will log you onto one of the clusters at Utah's CHPC.

In the sections that follow, I will introduce you to several directories and files on the CHPC servers. Each one is described by its "path name" relative to your own "home directory," which is represented by the symbol `~`. (In scripts, this directory is often called `$HOME`.) Before these path names will work, you'll need to define two "soft links" within your own home directory, which point to group storage devices owned by the Rogers lab. After you define these soft links, `~/grp1` will refer to one of these group storage devices and `~/grp2` to the other. You can establish these links by typing

```
ln -s /uufs/chpc.utah.edu/common/home/rogersa-group1 ~/grp1
ln -s /uufs/chpc.utah.edu/common/home/rogersa-group2 ~/grp2
```

If everyone defines these soft links in the same way, then we can all use path names of the form `~/grp1/rogers/data/whatever`.

The Legofit executables are already available in directory `~/grp1/bin`. To make these easily available, use a text editor to edit the file `.bash_profile` in your home directory. Within that file, you will find a line that defines a variable called `PATH`, which is used by bash to find executable programs. We want to add `~/grp1/bin` to the end of `PATH`, so that bash can find the Legofit executables. Edit that line that defines `PATH` so that it looks like this

```
export PATH=$PATH:$HOME/bin:$HOME/.local/bin:$HOME/grp1/bin
```

Also add the following line at the end of the file

```
module --latest load git
```

This last line will give you access to the program "git," which you'll need to access the agar22 repository. After making these changes, save the file and exit the editor. The file `.bash_profile` is normally read only once, when you first log in. Because you've only just edited this file, bash doesn't yet know about the changes you made there. To fix this, you can either type `logout` and then `ssh` back in again, or type `. .bash_profile`, which tells bash to execute the commands in this file.

Finally, clone the agar22 repository into your own home directory on the CHPC.

```
git clone https://github.com/alanrogers/agar22.git agar22
```

This will create a subdirectory called "agar22," which contains files related to the workshop.

4 Making an input file in .raf format

Legofit uses data in ".raf" format. The following data sets are already available on the server at Utah's CHPC.

Altai Neanderthal	`~/grp1/rogers/data/altai/orig2/altai.raf.gz`
Vindija Neanderthal	`~/grp1/rogers/data/vindija/orig/vindija.raf.gz`
Western Europeans (SGDP)	`~/grp1/rogers/data/simons/raf/weur.raf.gz`

If you'd like to make a data set for another modern human population, copy the following slurm script into a directory of your own:

```
~/grp1/rogers/data/simons/raf/weur.slr
```

Change its name (because "weur" stands for "western European") and edit it to reflect the population or populations that you want to study. You'll find lots of options under

```
~/grp1/rogers/data/simons/
```

Near the top of this script, you'll find the lines

```
#SBATCH --account=rogersa-np
#SBATCH --partition=rogersa-np
```

These specify the account and partition under which the job will run. The choice "rogersa-np" refers to my (single) node on the notchpeak cluster. To run the job there, log into the notchpeak cluster, use "cd" to move to the directory that contains the script and then type

```
sbatch <your script name>
```

where `<your script name>` is the name of your version of the script.

If the notchpeak node is busy, try kingspeak. To do so, log into kingspeak and specify `--account=rogersa-kp` and `--partition=rogersa-kp`. To do this, it isn't necessary to edit the file. You can override the values in the file by specifying these options on the command line, like this:

```
sbatch -A rogersa-kp -p rogersa-kp <your script name>
```

If you're a member of my lab, you can also run under my allocation. To do this on notchpeak, log into that cluster and use

```
sbatch -A rogersa -p notchpeak <your script name>
```

On kingspeak, the command is

```
sbatch -A rogersa -p kingspeak <your script name>
```

The lonepeak cluster is available to users who have no allocation. To use it, log into lonepeak and type

```
sbatch -A rogersa -p lonepeak <your script name>
```

However you run your script, it will take hours to complete. You can check on its status at any time by typing

```
squeue -u <your user id>
```

where `<your user id>` is the id you use to log onto the CHPC cluster. As a shorthand, I often type this as

```
squeue -u ‘whoami‘
```

which uses Linux's "whoami" command to fill in your user id. (Note: the ticks in this command are "back ticks.")

5 Tabulating site patterns

Legofit works with the frequencies of "nucleotide site patterns," which are explained in this section of the Legofit website. During the workshop, you will not need to tabulate site patterns, because I'll do that for you before the workshop begins. This section is here to help you do the job on your own, after the workshop.

The Legofit package includes several programs for tabulating site patterns. I'll focus here on "sitepat," which works with data files rather than with the output of a simulation program. Sitepat reads a series of files in .raf format and writes a file describing the frequency of each site pattern. Optionally, it also writes a series of output files, each describing site pattern frequencies in a different bootstrap replicate. These replicates are used for measuring statistical uncertainty. Details are in the Legofit docs. Sitepat reads enormous data files and can take hours to run, even on a compute cluster. That is why we aren't doing this step during the workshop.

Have a look at file data.opf. After the header, the first few lines look like

```
#       SitePat          E[count]          loBnd          hiBnd
              x    312922.8110121 311974.2398439 313732.1909228
              y    306301.1413693 305178.5610866 307121.6833708
              v    102039.5699405 101293.0904018 102887.7097098
              a     78653.9538690  77779.2808408  79467.4795387
              d    339489.6889881 337728.6566221 341531.9807664
            x:y    189532.2514885 187871.4715776 191321.2242190
```

The left column lists the site patterns. "E[count]" is (more or less) the number of sites exhibiting each site pattern. (For a full explanation, see the Legofit docs.) The values under "loBnd" and "hiBnd" enclose a 95% confidence interval. The frequency of site pattern "x" is its value of "E[count]" divided by the sum of all the values under "E[count]."

It's best to run sitepat within a script, both because it takes awhile to run, and also because the script will document what you did. Here is the slurm script that generated the output above.

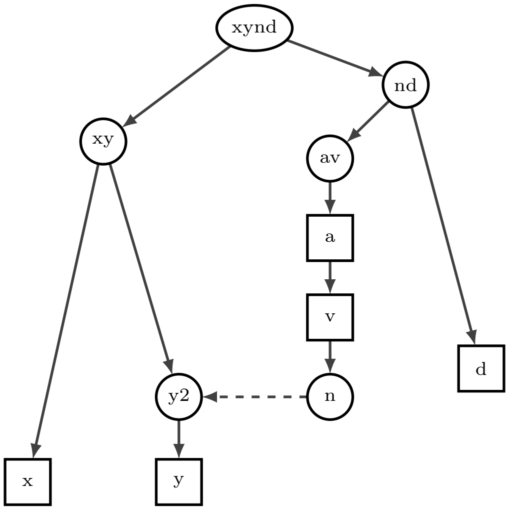

Figure 1: Model of history described in file "a/a.lgo." Squares represent segments from which we have data. Solid arrows indicate ancestor-descendant relationships, and dashed arrows represent gene flow. Vertical axis is not to scale.

6 Fitting models of history

6.1 Organizing the data directory

For each set of site pattern data, I create a directory tree with the same format, which is illustrated in the github repository at legofit/europe. Within this directory, you'll find (1) a README.md file, which describes all the others, (2) a file called "data.opf," which contains observed site pattern frequencies, (3) sitepat.slr, a slurm script that runs "sitepat" and generates "data.opf," and "boot," a directory containing bootstrap replicates. There will eventually be other files with names like "all.bootci" and "all.bma," which contain the results of analyses. Within this directory, you will also find subdirectories with names like "a," "ab," and so on. Each of these refers to a different model of history, which we wish to fit to the data in "data.opf." Name these as you wish, but it's a good idea to keep the names short. In my naming scheme, directory "a" contains a model in which there is only one episode of admixture, labeled α; "ab" is for a model that has episodes α and β; and so on.

6.2 Preliminary exercises with legosim and legofit

Within the subdirectory for each model, assumptions about population history are described in file "a.lgo." These assumptions are described using syntax that you can read about here. Use `cd` to move into directory "a," and have a look at a.lgo, which describes the model illustrated in Fig. 1

We can use this file to calculate the expected frequency of each site patter. Type:

```
legosim a.lgo
```

This runs the program "legosim," which reads file a.lgo, estimates the probability of each site pattern, and prints out the result. By default, legosim doesn't calculate probabilities for singleton site patterns (those in which the derived allele is present only once), and it uses a stochastic algorithm to estimate probabilities. To include singleton site patterns and use the (faster and more accurate) deterministic algorithm, the command would be

```
legosim -1 -d 0 a.lgo
```

Because it is fast, legosim is a useful way to check for errors in your .lgo file. For further details about the program and its command-line arguments, see its documentation.

Having ascertained that the sotware can read “a.lgo” without errors, let’s use the “legofit” program to estimate parameters. The parameters to be estimated are those defined as “free” in file “a.lgo.” Type the following at the bash command line:

```
legofit -1 -d 0 --tol 1e-3 a.lgo ../data.opf
```

After a few seconds, legofit should print a page of output. But before examining that, let’s unpack the command that generated it. The 1st argument on the command line is “`-1`.” This tells legofit to use singleton site patterns—those in which the derived allele is present only once. Next, the arguments “`-d 0`” tells legofit to use its deterministic algorithm, which is faster and more accurate than the stochastic one it uses by default. (The stochastic algorithm is needed for complex models.) The arguments “`--tol 1e-3`” tell legofit to be satisfied with a fairly loose fit between model and data. In research, you would want a smaller number here. The next argument, `a.lgo`, is the file describing the model of history, and the final argument, `../data.opf`, is the data file.

Now lets examine the output of that last command. The first section of output echoes the the legofit command and the input parameters, along with default values of parameters that we didn’t set on the command line.

```
########################################
# legofit: estimate population history #
#      version 2.3.8-12-gf1f00c9       #
########################################
#
# Program was compiled: Jun 21 2022 11:28:40
# Program was run: Thu Jul  7 20:38:55 2022
#
# cmd: legofit -1 -d 0 --tol 1e-3 a.lgo ../data.opf
# curr dir: /Users/rogers/txt/agar22/legofit/europe/a
# Stage nOptItr nSimReps
#     0    1000        1
# algorithm          : deterministic
# ignoring probs <=  : 0
# Branch length floor: 0
# DE strategy        : 2
#    F               : 0.3
#    CR              : 0.8
#    tolerance       : 0.001
# nthreads           : 2
# lgo input file     : a.lgo
# site pat input file: ../data.opf
# free parameters    : 12
# points in DE swarm : 120
# Including singleton site patterns.
# cost function      : KL
```

In the output above, “tolerance” is the value we specified as `--tol 1e-3` on the command line. The next section prints the initial values of all parameters, as read from the .lgo file.

```
Initial parameter values
Fixed:
      zero = 0
       one = 1
       TmN = 1
     Txynd = 25920
Free:
       Tnd = 24000
       Tav = 14000
       Txy = 10000
        Td = 2500
        Ta = 4000
        Tv = 1000
    twoNav = 2000
     twoNn = 2000
    twoNnd = 2000
    twoNxy = 20000
  twoNxynd = 20000
        mN = 0.01
```

Then comes a line that tells us the result of the optimization procedure.

```
DiffEv reached_goal. cost=7.22802e-04 spread=8.77863e-04
```

To understand this line, you need to know how legofit fits parameters to data. It uses something called "KL divergence" [5] to measure the difference between observed site pattern frequencies and those predicted by the model of history. Legofit searches for parameter values that minimize KL divergence, which is called "cost" in the output above. The value printed there (7.22802e-04) is the lowest KL divergence legofit was able to find. This search is conducted by an algorithm called "differential evolution" (DE) [7]. DE maintains a swarm of points, each representing a guess about parameter values. In each iteration of the algorithm, the best points (those with lowest KL) mutate, recombine, and reproduce to produce a new generation of points. The algorithm stops when the difference between the best point and the worst (called "spread" in the output above) falls to a specified tolerance. As you can see in this output, spread is smaller than 0.001, the tolerance we specified using "`--tol 1e-3`" on the command line. Thus, the algorithm has reached its goal, and this is why it printed `DiffEv reached_goal` in the output above. This goal is arbitrary, and the output can be useful even when the goal is not reached.

The next section of output lists the fitted values of all parameters.

```
Fitted parameter values
Free:
       Tnd = 24871.7
       Tav = 16727.1
       Txy = 13090.4
        Td = 1494.04
        Ta = 5346.73
        Tv = 2660.19
    twoNav = 14309.4
     twoNn = 8788.21
    twoNnd = 2998.19
    twoNxy = 23935.7
  twoNxynd = 46699.2
        mN = 0.0198353
```

Finally, legofit prints the expected branch length associated with each site pattern in the fitted model.

```
#       SitePat  BranchLen
              x 38204.0073508
              y 37200.1770458
              v 12628.3146863
              a 10003.0899715
              d 41976.1547361
            x:y 22591.0958761
            x:v 274.3861485
            x:a 278.8300856
            x:d 3121.4558596
            y:v 383.1800171
            y:a 321.8630482
            y:d 2936.3292837
            v:a 28571.5881777
            v:d 1060.8437350
            a:d 1067.6185093
          x:y:v 576.2188433
          x:y:a 571.7749063
          x:y:d 7001.3957667
          x:v:a 2884.5519560
          x:v:d 225.8439349
          x:a:d 230.2878720
          y:v:a 3460.0030731
          y:v:d 235.9020933
          y:a:d 229.1273190
          v:a:d 17599.3335385
        x:y:v:a 7238.2996703
        x:y:v:d 624.7610570
        x:y:a:d 620.3171199
        x:v:a:d 5567.0327885
        y:v:a:d 6019.8141157
```

For example, the expected branch length of site pattern x is 38204.0073508 generations. This is the average length, within the gene genealogy, of the branch that, should a mutation strike it, would generate site pattern x. Under the model of infinite sites [4], the expected frequency of a site pattern among polymorphic sites is proportional to its expected length and can be calculated as the ratio of this length to the sum of all the lengths.

6.3 A legofit pipeline

A minimal legofit analysis would consist of just one run of the program, such as the one we did above. This provides a point estimate of each parameter but does not estimate the uncertainties of those estimates. To estimate uncertainties, you also need to run legofit on each bootstrap estimate. Having done all that, you have completed what I like to call "stage 1" of a complete analysis.

This first stage is all one really needs to do. I usually do several additional stages (described below), which do seem to improve precision. They are time consuming, however. If one is fitting numerous models of high complexity, I advise fitting all models with only the first stage of analysis. Then choose a few of the best models for detailed study, using the full pipeline.

The second stage of analysis is designed to guard against a problem that often arises when one tries to find the minimum of a complex function. The cost function may have many local minima (low spots), some of which are deeper than others. We are looking for the "global minimum," which is the deepest of these local minima. The DE algorithm is pretty good at finding global minima, but it isn't perfect. It's possible that the legofit runs from stage 1 ended up at several local minima. Stage 2 is designed to choose among these local minima.

To make this possible, each job in stage 1 uses the `--stateOut` option, which tells legofit to write a file summarizing the final state of the DE swarm. Then, each job in stage 2 uses the `--stateIn` option, which tells legofit to initialize its swarm of DE points by reading all the state files produced by stage 1. Thus, each legofit job in stage 2 begins with a DE swarm that includes points from all the legofit jobs of stage 1. If the initial swarm includes points from several local minima, then legofit can choose among these to find the global minimum.

In addition to multiple local minima, legofit also faces another problem: a small increase in the value of one parameter may have nearly the same effect as a small decrease in another. This problem can be seen in Fig. 2, where some parameter estimates are tightly correlated with others. These correlations make it hard for the optimizer to zero in on optimal parameter values.

To ameliorate this problem, I use Legofit's "pclgo" program. It uses principal components analysis to reexpress all free parameters as functions of a set of uncorrelated "principal components." Optionally, you can tell pclgo to ignore principal components that account only for a tiny fraction of the variance. This step rewrites a.lgo to produce a new file called "b.lgo." In this new file, all free variables are principal components, and all the real variables are functions of the principal components. The third and fourth stages of analysis are just like the first and second, except that they use b.lgo instead of a.lgo.

There are several ways to organize this four-stage analysis into a pipeline. On the CHPC cluster, I use slurm scripts, which are discussed below. First, however, I introduce a version of the pipeline written entirely in bash.

6.3.1 Bash version of pipeline

Within the agar22 directory tree, `cd` into directory `legofit/europe/a`. You'll find several bash scripts there, whose names all end with `.sh`. All but one of these scripts is involved in the bash version of the pipeline. The bash pipeline is a scaled-down version of the one described below in section 6.3.2. In this smaller version, legofit fits only 10 of the bootstrap replicates, rather than all 50, and each run of legofit is allowed to do only 100 iterations of the optimization algorithm, rather than 5000. Because we are using so few replicates, it isn't possible to do a principal components analysis, so the bash pipeline does only the first two stages of analysis.

The bash pipeline consists of the following files:

`a.lgo`	defines model of history
`a1.sh`	stage 1 of analysis on real data
`a1boot.sh`	stage 1 on a bootstrap replicate
`a2.sh`	stage 2 on real data

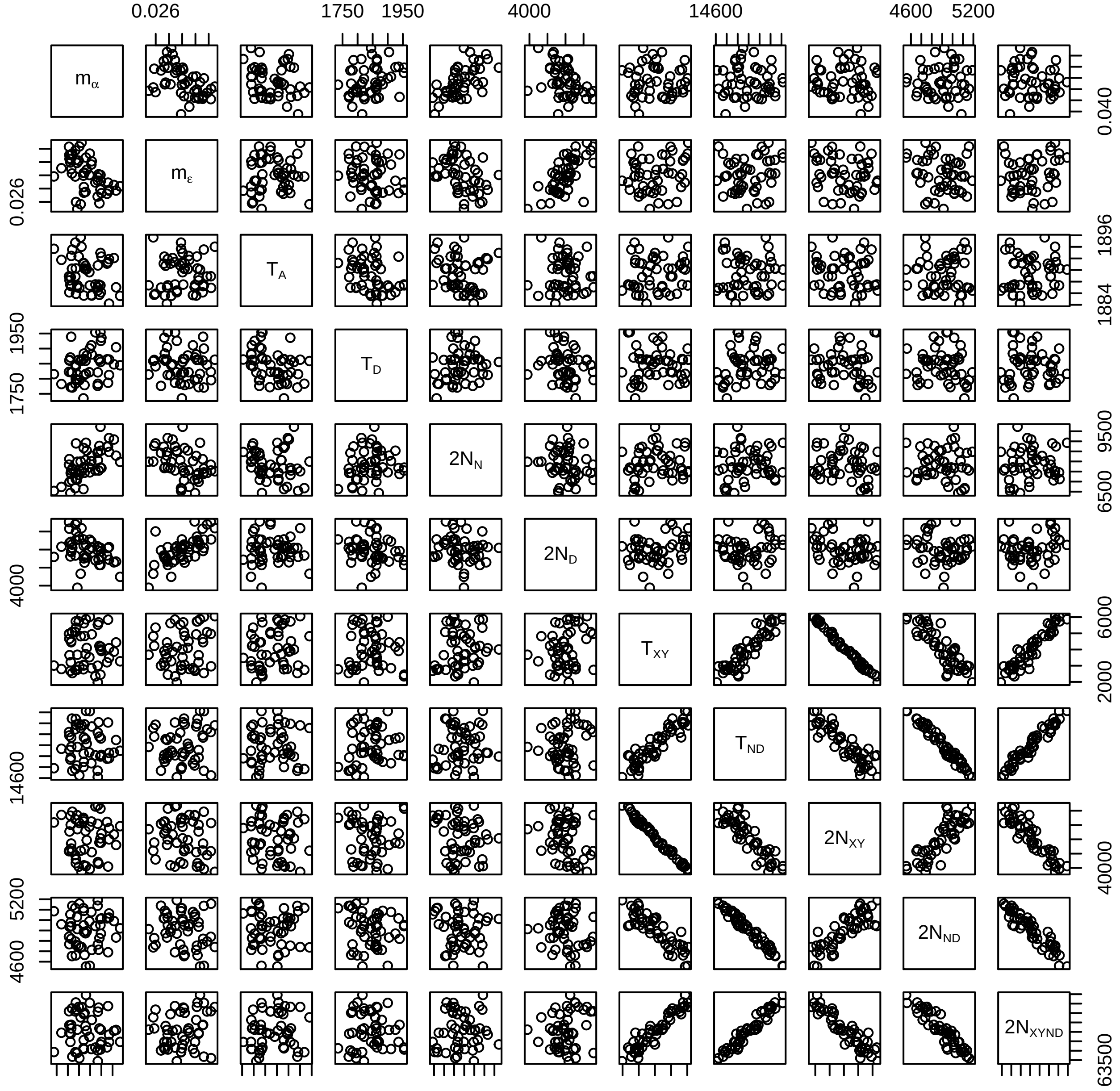

Figure 2: Scatter plot of each parameter against each other, based on 50 simulated data sets. After Rogers [9, Fig. 3].

`a2boot.sh`	stage 2 on a bootstrap replicate
`bash_pipeline.sh`	runs all the other scripts

To run the first stage of analysis on the real data, enter the following command:

```
bash a1.sh
```

That command takes 2.7 seconds on my 2018 Macbook Air. After it runs, there should be three new files in your directory:

a1.legofit	main output from legofit
a1.err	error output from legofit
a1.state	state of DE swarm at end of legofit run

To run the first stage on the first 10 bootstrap replicates, type:

```
seq 0 9 | xargs -n 1 bash a1boot.sh
```

In real research, you'd want to use all 50 bootstrap replicates. The abbreviated command above will run 10 jobs, each of which might have been launched by typing on the command line something like

```
bash a1boot.sh <i>
```

where `<i>` is 0 for the first job, 1 for the second, and so on up to 9. Each job will analyze a single bootstrap replicate and will write three files that are analogous to the three discussed above. It takes a minute or two to do all 10 bootstrap replicates.

We could, if we wished, proceed in this fashion and execute the remaining stages of analysis by typing bash commands at the terminal. But there is an easier way. Examine the file `bash_pipeline.sh` by typing

```
less bash_pipeline.sh
```

After a few lines at the top, which establish rules for dealing with errors, you will find

```
# Stage 1 of analysis.
bash a1.sh
seq 0 9 | xargs -n 1 bash a1boot.sh
```

These lines are identical to the commands we used above to execute the first stage of analysis. To do the entire analysis, type

```
bash bash_pipeline.sh
```

You should end up with 22 .legofit files (11 for each stage), the same number of .err files, and half that many .state files. Before talking about what to do with these files, let us consider other ways to implement a pipeline.

This implementation, using bash, has the advantages of simplicity and low overhead. It's probably the easiest version to get working on your laptop. However, there are better ways to implement a pipeline. One problem the current approach is that when something goes wrong half way through the analysis, it's hard to start the process up again in the middle. Another is that we are doing only one legofit run at a time. It would be better to do many runs in parallel.

In the bash pipeline, we used the xargs command. This has a "`-P`" argument, which makes it possible to run multiple jobs in parallel. This would not be useful in the current pipeline, however, because legofit is multithreaded and each run uses all the cores available on your computer. To run multiple legofit jobs in parallel, you need to run each one on a different node within a high-performance computing cluster. This is the rationale for the slurm version of the pipeline.

We won't be using slurm in the AGAR workshop. So if you're in the workshop, skip ahead to section 8.

6.3.2 Slurm version of pipeline

Use ssh as described above to log into one of the CHPC clusters, and then `cd` into directory `agar22/legofit/europe/a`. You'll find several slurm scripts there, whose names all end with `.slr`. There is also a shell script called "pipeline.sh," which runs the entire pipeline. The pipeline comprises the following files:

`a.lgo`	defines model of history
`a1.slr`	stage 1 of analysis on real data
`a1boot.slr`	stage 1 on a bootstrap replicate
`a2.slr`	stage 2 on real data
`a2boot.slr`	stage 2 on a bootstrap replicate
`pclgo.slr`	run pclgo to create b.lgo
`b1.slr`	stage 3 of analysis on real data
`b1boot.slr`	stage 3 on a bootstrap replicate
`b2.slr`	stage 4 on real data
`b2boot.slr`	stage 4 on a bootstrap replicate
`pipeline.sh`	runs all the slurm scripts

To run the first stage of analysis on the real data, enter the following command

```
sbatch -A rogersa -p <your cluster> a1.slr
```

where `<your cluster>` is "notchpeak," "kingspeak," or "lonepeak," depending on which cluster you logged into. This queues your job for execution but does not necessarily run it immediately. If the system is busy, it may not run for hours. To check on the status of this job, type

```
squeue -u <your unid>
```

The output will summarize the status of all jobs you've submitted to slurm.

To run the first stage of analysis on all bootstrap replicates, the command is

```
sbatch -A rogersa -p <your cluster> --array=0-49 a1boot.slr
```

To launch the entire pipeline, first use a text editor (e.g., vim or emacs) to edit the file pipeline.sh. Near the top of that file, you'll find the lines

```
export SBATCH_ACCOUNT=rogersa
export SBATCH_PARTITION=notchpeak
```

Edit these lines to reflect the account and partition you are using, then save the file and exit the editor. Finally, type

```
. pipeline.sh
```

This will queue the entire pipeline for execution.

7 Model selection, model averaging, and confidence intervals

As discussed above, legofit uses KL divergence to measure the difference between observed and expected site pattern frequences, and it labels this value "cost" in its output. One might therefore assume that the best model would be the one with the lowest value of cost. This is not necessarily so, because of the problem of overfitting. Very complex models fit not only the signal in the data, but also the noise, and the noise in one data set does not predict that in another. Consequently, the best-fitting model may not be the most predictive.

To address this problem, Legofit uses the "bootstrap estimate of predictive error" (bepe) [8, 2, 3]. This method uses variation among data sets (the real data plus 50 replicates generated by a moving-blocks bootstrap [6]) to approximate variation in repeated sampling. It fits the model to one data set and then tests this fit against all the others. Let us apply Legofit's "bepe" program to the output from the last stage of analysis. Use `cd` to move into the directory with all your .legofit output files, and type

```
bepe ../data.opf ../boot/boot*.opf -L b2.legofit b2boot*.legofit > b2.bepe
```

On the bepe command line, we first list all the input .opf files, beginning with the real data. Then, after `-L` we list the corresponding .legofit output files in the same order. The first few lines of output look like this

```
################################################
# bepe: bootstrap estimate of predictive error #
#          version 2.3.8-12-gf1f00c9          #
################################################
#
# Program was compiled: Mar 22 2022 17:40:38
# Program was run: Sat Jul  9 13:41:24 2022
# curr dir: /Users/rogers/txt/agar22/legofit/europe/a
#
#          bepe        DataFile     LegofitFile
1.130271806e-06        data.opf      b2.legofit
1.089578158e-06       boot0.opf b2boot0.legofit
```

The first number in the left column is the bepe value of the real data. This is the number to use in choosing among models. The numbers below this one refer to bootstrap replicates and are use by booma, to which we now turn.

If one model has a bepe value much lower than the others, then it's a pretty obvious winner. But what if the best bepe is only slightly lower than the second best? When several models provide reasonable descriptions of the data, it is better to average across models than to choose just one. This allows uncertainty about the model itself to be incorporated into confidence intervals. For this purpose, Legofit uses bootstrap model averaging, "booma" [1, 8]. The booma weight of the ith model is the fraction of data sets (including the real data and 50 bootstrap replicates) in which that model "wins," i.e. has the lowest value of bepe.

In the agar22 repository, there are several subdirectories just below `legofit/europe`, each of which contains the analysis of a single model. Each of these contains a file names `b2.bepe`, which was calculated as described above. Each directory also contains a file named `b2.flat`, which was calculated like this:

```
flatfile.py b2.legofit b2boot*.legofit > b2.flat
```

From the `legofit/europe` directory, type

```
booma */b2.bepe -F */b2.flat > all.bma
```

This will create a file called “all.bma” that gives the booma weight of each model and also the model-averaged parameter estimates. The first few lines look like this

```
##################################
# booma: bootstrap model average #
#    version 2.3.9-7-g8bbcef2    #
##################################
#
# Program was compiled: Jul  9 2022 14:41:56
# Program was run: Sat Jul  9 14:50:30 2022
# cmd: booma a/b2.bepe ab/b2.bepe abc/b2.bepe abcd/b2.bepe abd/b2.bepe ac/b2.be>
# curr dir: /Users/rogers/txt/agar22/legofit/europe
#         Weight        MSC_file       Flat_file
#              0       a/b2.bepe       a/b2.flat
#              0      ab/b2.bepe      ab/b2.flat
#              0     abc/b2.bepe     abc/b2.flat
#   0.9803921569    abcd/b2.bepe    abcd/b2.flat
#  0.01960784314     abd/b2.bepe     abd/b2.flat
#              0      ac/b2.bepe      ac/b2.flat
#              0     acd/b2.bepe     acd/b2.flat
#              0      ad/b2.bepe      ad/b2.flat
```

Only two of the models (abc and abcd) have non-zero weights. These weights are used in calculating the model-averaged parameter estimates, which are listed below the weights.

To calculate the bootstrap confidence interval of each parameter, type

```
bootci.py all.bma > all.bootci
```

These confidence intervals are slightly wider than they would be, had we based them on results from a single model, rather than on an average across models. This is because they include uncertainty about the model itself in addition to uncertainty about parameter values under a given model.

8 Graphing results

There are several R scripts in the legofit section of the agar22 repository. Two of them are in `legofit/europe`:

patfrq.r	plots site pattern frequencies as in Rogers [10, Fig. 3]
bmadot.r	plots confidence intervals as Rogers [9, Fig. 7]

You can experiment with those on your own. During the workshop, we’ll be working with the three R scripts in `legofit/europe/a`:

a2pairs.r	makes a pairwise scatterplot like Fig. 2
resid.r	plots residual error as in Fig. 5 of [9]
dotplot.r	plots confidence intervals as in Fig. 7 of [9]

To use these, first make sure that R is installed on your system.

The next step is to summarize legofit's output files, so that R can read them. Type

```
flatfile.py a2.legofit a2boot*.legofit > a2.flat
```

This creates "a2.flat," a file with a row for each data set (the real data and 10 bootstrap replicates) and a column for each free variable. To examine it, type `less -S a2.flat`. The final summary file is created like this:

```
bootci.py a2.flat > a2.bootci
```

This creates "a2.bootci," which gives the bootstrap confidence interval of each parameter.

To make a pairwise scatterplot, type

```
Rscript a2pairs.r
```

This creates a file called "a2pairs.pdf," which you can examine with any pdf viewer. It makes a scatter plot of each parameter against each other parameter. One of the plotted circles in each panel is the real data and the rest are bootstrap replicates. You will probably find that some parameters are tightly correlated with others, as in Fig. 2.

To plot the residuals, type

```
Rscript resid.r
```

This will make "a-a2-resid.pdf," which will show you the residual errors, i.e. the difference between the observed site pattern frequencies and those predicted by the model. The red asterisks are the real data, and the blue circles are bootstrap replicates. The blue scatters measure the uncertainty in the estimate, and ideally these scatters would all overlap zero. But you'll probably find that this isn't so. You've fit a very simple model, and the lack of fit is a clue that something is missing from the model.

To plot the confidence interval of each parameter, type

```
Rscript dotplot.r
```

This will create "a-det-dot.pdf." In this graph, the open circles are point estimates, and the horizontal lines are 95% bootstrap confidence intervals. Don't worry if some of your point estimates are outside of the confidence intervals. This can happen when confidence intervals are generated by bootstrap. Look at the confidence intervals of the parameters that are tightly correlated with other parameters in a2pairs.pdf. You'll probably find that these parameters have wide confidence intervals. This is because tight correlations make it hard to get precise estimates, and this results in broad confidence intervals.

9 Conclusions

If you followed all the steps above, you should have all the tools you need to do a complete legofit analysis on your own computer. It is feasible to fit models of moderate complexity on a laptop. More complex models demand faster computers with multiple cores. Highly complex models are best studied on a high-performance computing cluster. The instructions above are specific to the CHPC at my own institution. You may need to modify them if you use a different center for high-performance computing.

Acknowledgements

This work was supported by NSF BCS 1638840, NSF BCS 1945782, and the Center for High Performance Computing at the University of Utah.

References

[1] Steven T Buckland, Kenneth P Burnham, and Nicole H Augustin. "Model Selection: an Integral Part of Inference". In: *Biometrics* 53.2 (1997), pp. 603–618. DOI: 10.2307/2533961.

[2] Bradley Efron. "Estimating the Error Rate of a Prediction Rule: Improvement on Cross-Validation". In: *Journal of the American Statistical Association* 78.382 (1983), pp. 316–331. DOI: 10.1080/01621459.1983.10477973.

[3] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. New York: Chapman and Hall, 1993. DOI: 10.1007/978-1-4899-4541-9.

[4] Motoo Kimura. "The Number of Heterozygous Nucleotide Sites Maintained in a Finite Population Due to Steady Flux of Mutation". In: *Genetics* 61 (1969), pp. 893–903. DOI: 10.1093/genetics/61.4.893.

[5] Solomon Kullback and Richard A Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (Mar. 1951), pp. 79–86. DOI: 10.1214/aoms/1177729694.

[6] Regina Y. Liu and Kesar Singh. "Moving Blocks Jacknife and Bootstrap Capture Weak Dependence". In: *Exploring the "Limits" of the Bootstrap*. Ed. by Raoul LePage and Lynne Billard. New York: Wiley, 1992, pp. 225–248.

[7] Kenneth Price, Rainer M Storn, and Jouni A Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Berlin: Springer Science and Business Media, 2006. ISBN: 978-3-540-20950-8.

[8] Alan R. Rogers. "Legofit: Estimating Population History from Genetic Data". In: *BMC Bioinformatics* 20 (2019), p. 526. DOI: 10.1186/s12859-019-3154-1.

[9] Alan R. Rogers. "An Efficient Algorithm for Estimating Population History from Genetic Data". In: *Peer Community Journal* 2 (2022), e32. DOI: 10.24072/pcjournal.132.

[10] Alan R. Rogers. "Using Genetic Data to Build Intuition about Population History". In: *arXiv* 2201.02668 (2022). DOI: 10.48550/arXiv.2201.02668.