Luria-Delbrück data analysis

January 29 2018

1 Protocol

A few notes before we begin.

- Text in this font `refers to R code`, be it example code or function inputs.
- There are two appendices after the main protocol.
- Appendix A has R coding advice, and if you see superscripts ([1]) in the protocol, it is to link you to advice there.
- Appendix B has information about all the inputs to the functions you will use in this lab. If you want to check the default values (and you probably should), you will want to look at the actual code.
- Parenthetical questions are not assigned, but they are pertinent to the concepts of both this and the last lab and are worth discussion.

1.1 The Directed Mutation model doesnt work, why?

Parameters control statistical models. The Directed Mutation model is mathematically simple, it has just one parameter, λ (the Greek letter lambda). This parameter gives both the mean and the variance of the distribution (where the distribution is on counts of surviving bacteria). For now we will not concern ourselves with biological interpretations of λ (it will be a function of both the number of bacteria you plated and the mutation rate).

1. Download the scripts for this lab, and the data, from the course website.
2. Open Rstudio and open a new script file (save it!).[1] This is where you will be writing all the code for the first part of the lab.
3. Load the packages ggplot2, and gridExtra (we need these for plotting).[2,3,4]
4. Read in the data from our class Luria-Delbrck experiment, make sure you assign it to a variable.[2,5,6]
5. We will start with the functions in `poisson_model.R`. In your script, source this file so we can use these functions.[2,7]
6. Were going to simulate from the Directed Mutation model (using `poissonModel()`, why is it called this?). Read the documentation for `poissonModel()` in this Appendix B, and look at the code to see what values do and do not have defaults.
7. Take a guess at a value of λ (you will be wrong, that is fine and partly the point!), and note the displays that print out. What are they telling you about how well the Directed Mutation model fits with this value of λ? Tweak values of λ until at least one of the simulated values matches the real value (try for the mean).
8. The function poissonBayes() is a fancier version of poissonModel. It uses Bayesian Inference to find the best value of lambda for you. Try it, and examine the summaries. Did it do any better than you did?

1.2 Building a more realistic model

As you have seen, reality is more complicated than the Directed Mutation model can capture. So we will now move onto more complicated models. These have many parameters that you will be able to adjust to try to match reality.

1. Open Rstudio and open a new script file (save it!).[1] This is where you will be writing all the code for the second part of the lab.
2. Source the file `growth_simulation.R`.[2,7] Read the documentation on `LuriaDelbruckInSilico()` and `simulationVisualSummary()`. While you should read the default values from the script file, you can safely ignore the actual code of the functions (but if youre interested, we commented it so you can read it!).
3. As practice, run a simulation[2] using the default values, using the function `LuriaDelbruckInSilico()`.
   (a) Set `number.of.experiments=50`
   (b) Set `number.of.tubes=<the number in the class dataset>`
   (c) Dont forget to assign the simulation to a variable![5]
4. Plot this simulation with `simulationVisualSummary()`.
5. Now youre free to explore! Using whatever parameters you like, try to find some combination of parameters that match the real data. Here are some tips:
   - Dont change more than one parameter value at a time, or you wont know which one made the simulation closer to/farther from reality.
   - Do try changing more than one parameter overall, just be careful of the above
   - If some parameter value causes the simulations to match reality, but that parameter value is unbelievable (i.e. a mutation probability of 1), trust your gut.
   - If you want to speed things up, you can try a more machine learning style approach. Divide the data into two parts (how do you do this fairly?) to create a training dataset and a test dataset, and work with the larger training dataset. If you think that looks good, see how it fits the test set.
   - Dont just change a parameter value and re-run the code, keep a record of what you did. Write a comment about how the simulations match or fail to match the observed value, and then use a new line and make the change.
6. Save a pdf of the summaries of your chosen parameter set and values[8] (you may want these for your lab report). Record the p-values somewhere (you could save them in the Rscript, by creating some variable and giving it that value). You should try to use more than 100 simulated replicates here (1000 is great if you are patient, but you should be able to get 200 in under 15 seconds). This will help your p-values and figure be more informative.
7. BEFORE YOU LEAVE check with another group. See what parameter values they are using. Are they similar to yours or different? If both your sets of values produce plausible models, which do you think is mode reasonable? Can you tell? Why or why not?

Appendix A

1. If you want a keyboard shortcut, try Command + Shift + N (Ctrl + Shift + N for non-mac)
2. Running code: from a script file, highlight the code or lines of code and press Command+Enter (on Mac) or Ctrl+Enter (on Windows or Linux), or use the run button. Alternately, if you simply put your cursor on the line you want to run and run it, Rstudio will auto-advance you to the next line for fast running of multiple lines
3. Loading packages: If you have a package installed already, in your script type and run `library(package name)`
4. Installing packages: Run `install.packages(package name)` to install a package.
5. Reading in data: R is built for data analysis, there are more ways than any one person knows to read in data. Given the format of our data, you can use `scan("path/to/data/file")`.
6. Assigning things to variables: `a <- 2` makes a variable `a` and gives it the value 2. Variables can store just about anything, including vectors like `scan()` reads in, *e.g.* `class_data <- scan("path/to/data/file")`
7. Sourcing/loading files: you can add it to your script with `source("path/to/file")` (the preferable option) or use source button in Rstudio.
8. You can print in two ways:
   - Using the Rstudio print menu
   - To code it, add two lines of code around a line with plotLuriaDelbruck(). Above the plotting line, use `pdf(file="wherever/you/want/it.pdf")`. Below that line use `dev.off()` (with nothing between the parentheses this time). You can change the size of the pdf with the arguments `width` and `height` in the call to the function `pdf()`.
9. Comments: anything after # in a line is not run by R, and is called a comment, and will not be run as code.

Appendix B

There are many functions you will use in this assignment. Here is a list with some explanation of all the inputs. The list is roughly in the order you will use the functions.

`poissonModel()`
This is a function that simulates from the Directed Mutation (Poisson) model.

- `lambda`: the sole parameter of the model, called the rate, equal to the mean and variance
- `real.data` the real data, as a vector, for visualization
- `num.draws`: number of simulated experiments (each of equal size to the real one)

The function plots a set of summary statistics of the model. Those include:

- `mean`: the mean number of surviving colonies of each experiment
- `variance` the variance of the number of surviving colonies of each experiment
- `relative.variance`: the variance of the number of surviving bacteria divided by the mean (should be 1 for a Poisson model)
- `kurtosis`: a measure of the tendency of the model to display particularly large values (numbers of surviving colonies)

`poissonBayes()`
This is a function that first fits the Directed Mutation model (using Bayesian Inference), and then simulates from this model. The inputs and outputs are the same as in `poissonModel()`.

`LuriaDelbruckInSilico()`
There are two inputs that you probably will not adjust as much, and determine the scale of the simulations. These should match the real data.

- `number.of.experiments`: determines how many total experiments are simulated
- `number.of.tubes`: determines how many tubes were reared in each experiment

The inputs that you should consider adjusting fall into four groups.

1. The termination conditions
    - `tube.volume.microliters`: how much liquid is in the test tube the bacteria are grown in?
    - `saturation.concentration.per.mL`: the concentration (bacteria per microliter) at which growth stops
    - `simulate.for.fixed.time`: the default is to simulate until saturation, this allows you to stop early, but will never allow more bacteria than the saturation constant
    - `end.time`: with `simulate.for.fixed.time` allows you to stop growth at some number of hours
2. Factors affecting the appearence and growth of mutants
    - `sensitive.doubling.time.hours`: how many hours to double the size of the wildtype population?
    - `resistant.doubling.time.hours`: how many hours to double the size of the resistant population?

- `mutation.probability`: the probability that, when a wiltype divides, it produces a mutant offspring
- `intial.sensitive.population.size.per.test.tube`: how many sensitive bacteria are in the innoculant (assume 0 resistant)?

3. Random fluctuations during plating process
   - `enable.tube.to.plate.variability`: allows us to model variability induced by pipetting technique (for example, not quite getting the entire pipetted volume)
4. Induced mutation (this allows us to include the Poisson/Directed Mutation model in our simulations)
   - `induced.mutation.probability`: the probability that a sensitive bacterium mutates to become resistant, post-plating, because of addition of virus (different from `poissonModel()`, where we used `induced.mutation.probability` $\times$ the number of plated bacteria)
   - `induced.mutation.probability.standard.deviation`: this allows the rate of mutation to vary between bacteria, allowing us to move beyond the Poisson model. This is the standard deviation of the mutation rate in the population that the innoculants were taken from, so be careful with large values (is a standard deviation of 0.1 and a mean of 10^{-8} resonable?). The curious can use `visualizeInducedMutationVariance()` to see what the chosen value implies about the variability in the mutation rate

`simulationVisualSummary()`
This is a function that will plot useful information about simulations you have performed. It is also called by the functions `poissonModel()` and `poissonBayes()`

- `simulated.data`: the simulations (of experiments) under a model
- `real data`: the real values from the experiment

`visualizeInducedMutationVariance()` This function gives some information on what the values of `induced.mutation.probability.standard.deviation` imply in `luriaDelbruckInSilico`. It plots the distribution of mutation rates in the founding population (the one the innoculants came from) as a black line, and the mean rate (which is given by `induced.mutation.probability`) as a dashed green line. The function also reports some information about how many bacteria fall in certain rate ranges, by default how many bacteria have a rate greater than 10^{-10}, 10^{-9}, 10^{-8}, 10^{-7}, and 10^{-6}.

- `induced.mutation.probability`: the probability that a sensitive bacterium mutates to become resistant, post-plating, because of addition of virus (different from `poissonModel()`, where we used `induced.mutation.probability` * the number of plated bacteria)
- `induced.mutation.probability.standard.deviation`: this allows the rate of mutation to vary between bacteria, allowing us to move beyond the Poisson model. This is the standard deviation of the mutation rate in the population that the innoculants were taken from, so be careful with large values (is a standard deviation of 0.1 and a mean of 10^{-8} resonable?). The curious can use `visualizeInducedMutationVariance()` to see what the chosen value implies about the variability in the mutation rate
- `evaluate.probabilities.at`: the function will report the probability that a given bacterium has a mutation rate greater than the specified value (or values)