Models of language change

Agent-based modelling, Konstanz, 2024

Henri Kauhanen

7 May 2024

Plan

- Up to now, we have implemented an agent that
  - uses G_1 with prob. p and G_2 with prob. $1-p$
  - can produce strings from both grammars
  - can receive such strings produced by other agents and update the value of p correspondingly
- It is now time to look more carefully what happens at the population level when multiple such agents interact
- In last week's homework, we encapsulated all our variational learning code in a module (download here)
- To use this module, we call:[1]

```
include("VariationalLearning.jl")
using .VariationalLearning
```

Population of agents

- Last time, we also saw how an array comprehension can be used to create a whole population of agents:

```
pop = [VariationalLearner(0.1, 0.01, 0.4, 0.1) for i in 1:1000]
```

[1] The first line makes Julia aware of the code, i.e. of the module definition. The second one then instructs Julia to use that module. The dot before the module name is required for complex reasons (simple answer: this is a module of our own making, not an "official" one).

```
1000-element Vector{VariationalLearner}:
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)

 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
 VariationalLearner(0.1, 0.01, 0.4, 0.1)
```

- We also saw that the `rand` function can be used to pick random agents from the population:

```
rand(pop)
```

```
VariationalLearner(0.1, 0.01, 0.4, 0.1)
```

- With an array comprehension, this allows us to undergo random interactions:

```
[interact!(rand(pop), rand(pop)) for t in 1:100]
```

```
100-element Vector{Float64}:
 0.099
 0.099
```

```
0.099
0.099
0.099
0.099
0.099
0.099
0.099
0.099
0.099
0.099
0.099

0.099
0.099
0.099
0.099
0.099
0.099
0.099
0.10900000000000001
0.099
0.099
0.09801
0.10900000000000001
0.099
```

Exercise

What gets returned is an array of 100 numbers (`100-element Vector{Float64}`).

What are these numbers, and where do they come from?

> Answer
>
> They come from `learn!` via `interact!`.
> Recall that the last line of the `learn!` function returns the current (i.e. new, after learning) value of p of the learner:
>
> ```
> function learn!(x::VariationalLearner, s::String)
>   ...
>   return x.p
> end
> ```

Summary statistics

- But this just gives us the current state of a random speaker
- This is rarely the sort of information we wish to gather
- More useful would be: average p over all agents in the population
- A quantity like this is known as a **summary statistic** – it summarizes the state of the entire population

Getting the average

- The average, or mean, can be obtained using Julia's `mean` function. This is part of the `Statistics` module:

```
using Statistics
my_vector = [1, 2, 3, 4, 5]
mean(my_vector)
```

```
3.0
```

- Note that you could also compute this "by hand"!

```
sum(my_vector)/length(my_vector)
```

```
3.0
```

Average p

- With our population, we can't just do:

```
mean(pop)
```

- Why? Well, `mean` takes the average over an array of numbers. But `pop` is **not** an array of numbers – it is an array of `VariationalLearner` objects.

Exercise

How can we obtain the average p over our `pop` object?

Answer

Once again, the answer is an array comprehension!

```
mean([speaker.p for speaker in pop])
```

```
0.10006002999999991
```

Average p

- Let's wrap this up as a function:

```
function average_p(x::Array{VariationalLearner})
  mean([speaker.p for speaker in x])
end
```

```
average_p (generic function with 1 method)
```

- Note: the type of the argument is `Array{VariationalLearner}`, which means an array of elements all of which are `VariationalLearner`s
- We can now simply call:

```
average_p(pop)
```

```
0.10006002999999991
```

Interacting and summarizing

- Earlier, we used this to evolve the population:

```
[interact!(rand(pop), rand(pop)) for t in 1:100]
```

- What if we also want to summarize, so that the resulting array stores the average p rather than the p of a random agent?
- Problem: array comprehension takes only a single command to the left of the `for` block
- Solution: a `begin ... end` block:

```
history = [begin
               interact!(rand(pop), rand(pop))
               average_p(pop)
           end for t in 1:100]
```

```
100-element Vector{Float64}:
 0.10002888999999995
 0.10002788999999994
 0.10002688999999994
 0.10002588999999994
 0.10003488999999995
 0.10003388999999994
 0.10003288999999996
 0.10003188999999994
 0.10003088999999994
 0.10002989999999994
 0.10002889999999993
 0.10002790999999994
 0.10002690999999994

 0.10002051079999993
 0.10001951079999995
 0.10001851079999996
 0.10001751079999996
 0.10001651079999996
 0.10002551079999997
 0.10002451079999995
 0.10002353069999995
 0.10002253069999996
 0.10002154069999997
 0.10002054069999995
 0.10001955069999996
```

- We can finally evolve the population, recording its average state at every iteration, for as many iterations as we wish:[2]

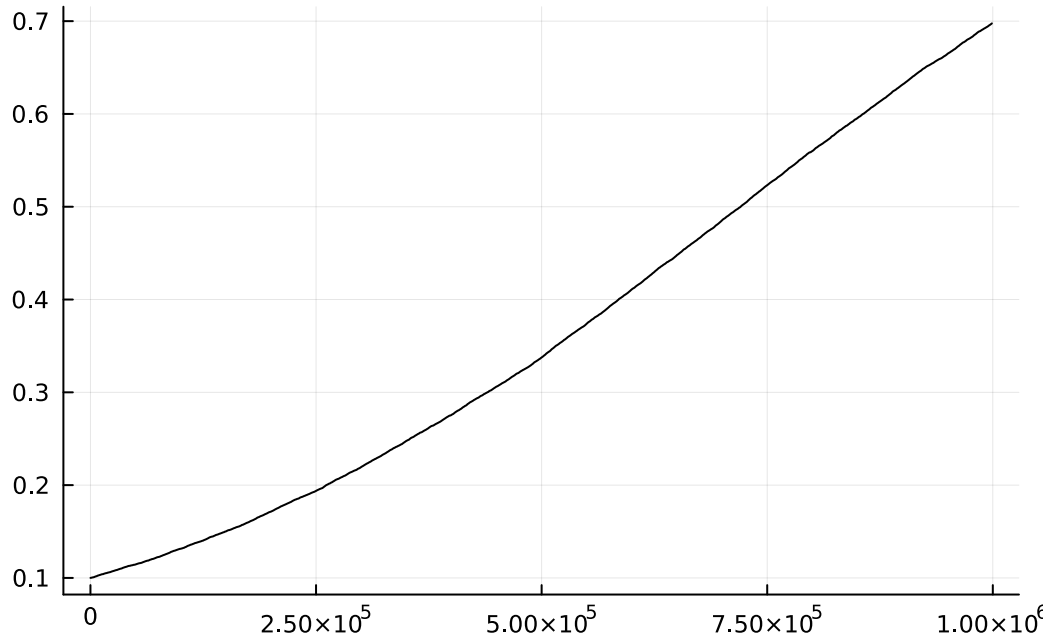
```
history2 = [begin
               interact!(rand(pop), rand(pop))
```

[2]Here, the `_` in the upper limit for `t` is a number separator. It just helps humans read the large number (here, a million); the compiler ignores it. You could also write `1000000` if you wish (but then I, for one, won't be able to tell how many zeroes you have there!).

```
            average_p(pop)
        end for t in 1:1_000_000]
```

- Let's plot the simulation history (i.e. the evolution of average p over time):

```
using Plots
plot(1:1_000_000, history2, color=:black, legend=false)
```



Free tip

- We have plotted a million points here (and `Plots` also connects them with veeeeery tiny lines). This is a lot, and may slow your computer down.
- To plot, say, every 1000th point, try:

```
plot(1:1000:1_100_000, history2[begin:1000:end], color=:black, legend=false)
```

Exercise

In our simulation, we see the average value of p steadily going up with time. What do you predict will happen in the future, i.e. if we continued the simulation for, say, another million time steps?

Answer

We would expect the average to keep increasing, as the p of every speaker tends to increase over time. Why does it tend to keep increasing? Because of the way we initialized the model: we set the `P1` and `P2` values for each learner to `0.4` and `0.1`, meaning that there is always more evidence for grammar G_1 than for grammar G_2.
Of course, the average value of p, just like each individual p, cannot increase forever. They have a hard maximum at $p = 1$, since probabilities cannot be greater than 1. In fact, the average p plateaus at 1, if we continue the simulation. (Try it!)

Looking at individual learners

- What if, instead of summarizing the population, we **want** to look at the histories of individual learners?
- This is also very easy, using **two-dimensional array comprehensions**.
- I will be using a much smaller population, for a much shorter simulation, for clarity:

```
pop = [VariationalLearner(0.1, 0.01, 0.4, 0.1) for i in 1:20]
history = [interact!(rand(pop), l) for t in 1:100, l in pop]
```

- Read this as: for every time step `t`, for every learner `l` in the population, make a random speaker speak to `l`.

- The result is a **matrix** (two-dimensional array), here of 100 rows and 20 columns:

```
pop = [VariationalLearner(0.1, 0.01, 0.4, 0.1) for i in 1:20]
history = [interact!(rand(pop), l) for t in 1:100, l in pop]
```

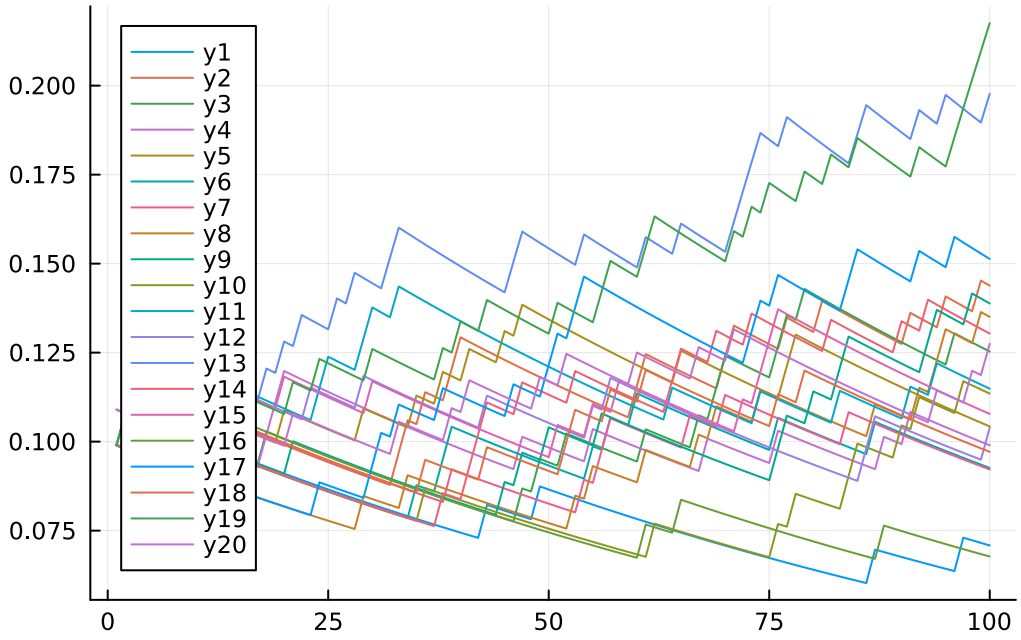
```
100×20 Matrix{Float64}:
 0.099      0.109      0.099      0.099      …  0.099      0.099      0.109
 0.09801    0.10791    0.09801    0.10801       0.09801    0.10801    0.10791
 0.0970299  0.106831   0.0970299  0.10693       0.0970299  0.10693    0.106831
 0.0960596  0.105763   0.0960596  0.105861      0.0960596  0.105861   0.105763
 0.095099   0.104705   0.095099   0.104802      0.095099   0.114802   0.104705
 0.094148   0.103658   0.094148   0.103754   …  0.094148   0.113654   0.103658
 0.0932065  0.102621   0.0932065  0.102716      0.103207   0.112517   0.112621
 0.0922745  0.101595   0.0922745  0.101689      0.102174   0.111392   0.111495
 0.0913517  0.100579   0.0913517  0.100672      0.101153   0.110278   0.11038
 0.0904382  0.109573   0.0904382  0.0996657     0.100141   0.119176   0.109276
 0.0895338  0.108478   0.0895338  0.098669   …  0.10914    0.117984   0.108184
 0.0986385  0.107393   0.0986385  0.0976823     0.108048   0.116804   0.107102
```

```
0.0976521  0.106319  0.0976521  0.0967055     0.106968  0.115636  0.106031

0.0682832  0.12873   0.129221   0.100333      0.108478  0.177992  0.109783
0.0676004  0.127443  0.127929   0.0993302     0.107393  0.176212  0.108685
0.0669244  0.136168  0.12665    0.108337   …  0.106319  0.174449  0.107598
0.0662551  0.134807  0.125383   0.107253      0.105256  0.182705  0.106522
0.0655926  0.133459  0.124129   0.106181      0.104203  0.180878  0.105457
0.0649366  0.132124  0.122888   0.105119      0.103161  0.179069  0.114402
0.0642873  0.140803  0.121659   0.104068      0.102129  0.177278  0.113258
0.0636444  0.139395  0.130442   0.103027   …  0.101108  0.185506  0.112126
0.073008   0.138001  0.129138   0.101997      0.100097  0.193651  0.111005
0.0722779  0.136621  0.127847   0.100977      0.0990961 0.201714  0.119895
0.0715551  0.145255  0.126568   0.0999673     0.0981051 0.209697  0.118696
0.0708395  0.143802  0.125303   0.0989676     0.0971241 0.2176    0.127509
```

- Each column of the matrix represents the history of one learner
- `plot()` is rather clever and accepts the matrix as argument:

```
plot(history)
```



Homework

- In the homework, you get to replicate the above simulations (with the summary statistic of average p), exploring how variation in **model parameters** such as population size and learning rate affects the population's evolution
- Next week:
  - Structured populations (with the help of *Agents.jl*)
  - Info about the final projects