Ensembles and statistics

Solution | Agent-based modelling, Konstanz, 2024

Henri Kauhanen

2 July 2024

Quantifying the duration of change

> How could you quantify the duration of a change using a single number? In other words, what sort of summary statistic can you use to decide whether one trajectory goes up earlier than another one? Try to go for the *simplest* such summary statistic.

Here's a simple summary statistic that will do the job. Let $\overline{p}$ refer to the mean p across the population of our variational learners; i.e. $\overline{p}$ is the average probability that G_1 is used. We will then find out the time point at which a simulation first satisfies $\overline{p} > 0.5$, i.e. the earliest time at which G_1 has more than 50% usage. Let T refer to this time point.

The statistical test

> Once you have such a number for each trajectory, you have a set of these numbers. What kind of statistical test could you use to decide whether the set of numbers for $\beta = 0.1$ is significantly different from the set of numbers for $\beta = 0.5$? (Hint: you want a test that compares two means from two samples.)

We get a single value of T for each simulation trajectory; for example, if we repeat the simulation 100 times for each value of β, we have two sets of 100 T numbers. To test whether there is a statistically significant difference between these sets of numbers, we can use a two-sample t-test.

Implementation

> Once you have answers to the above questions, you can try and implement the following procedure:

a. Instead of 10 simulations, use `ensemblerun!` to produce simulated trajectories for 100 repetitions for each β.
b. Then figure out how to extract your summary statistic from these data.
c. Finally, carry out the statistical test in order to make a decision.

a. Running the simulations

We first load all the necessary ingredients:

```
using Random
using Agents
using Graphs
using Statistics
using DataFrames
using HypothesisTests

include("VL2.jl")
using .VL
```

Here is our function that creates one model:

```
function make_model(beta)
  G = watts_strogatz(50, 8, beta)
  space = GraphSpace(G)

  model = StandardABM(NetworkVL, space,
                      agent_step! = VL_step!)

  for i in 1:50
    add_agent_single!(model, 0.01, 0.01, 0.2, 0.1)
  end

  return model
end
```

We can set the PRNG seed for reproducibility:

```
Random.seed!(1539)
```

Create vectors of models using array comprehensions:

```julia
models1 = [make_model(0.1) for i in 1:100]
models2 = [make_model(0.5) for i in 1:100]
```

Use `ensemblerun!` to simulate and obtain the mean of `p`:

```julia
data1, _ = ensemblerun!(models1, 10_000; adata = [(:p, mean)])
data2, _ = ensemblerun!(models2, 10_000; adata = [(:p, mean)])
```

Verify that the dataframes look the way we'd expect them to:

```julia
data1
```

	time	mean_p	ensemble
	Int64	Float64	Int64
1	0	0.01	1
2	1	0.0103	1
3	2	0.010197	1
4	3	0.010095	1
5	4	0.00999408	1
6	5	0.00989414	1
7	6	0.0097952	1
8	7	0.00989725	1
9	8	0.00999827	1
10	9	0.00989829	1
11	10	0.00979931	1
12	11	0.00990131	1
13	12	0.0100023	1
14	13	0.00990228	1
15	14	0.00980326	1
16	15	0.00990522	1
17	16	0.00980617	1
18	17	0.00990811	1
19	18	0.010009	1
20	19	0.0101089	1
21	20	0.0100078	1
22	21	0.0101078	1
23	22	0.0102067	1
24	23	0.0105046	1
25	24	0.0103996	1
26	25	0.0104956	1
27	26	0.0103906	1
28	27	0.0104867	1
29	28	0.0103819	1
30	29	0.010278	1
...	...	...	...

b. Obtaining the summary statistics

Now for the tricky part: in order to determine T for a simulation run, we need to find the lowest value of `time` such that `mean_p` is at least 0.5, for each value of `ensemble` separately.

Reading the part about indexing in the DataFrames.jl documentation, we find that the following command will take a **subset** of the original dataframe, a subset which only contains rows of the original dataframe on which the value of `mean_p` is greater than 0.5:

```
data1[data1.mean_p .> 0.5, :]
```

	time	mean_p	ensemble
	Int64	Float64	Int64
1	4543	0.500058	1
2	4545	0.500459	1
3	4546	0.501054	1
4	4547	0.502444	1
5	4548	0.503219	1
6	4549	0.502587	1
7	4550	0.501361	1
8	4551	0.501548	1
9	4552	0.503332	1
10	4553	0.504099	1
11	4554	0.504658	1
12	4555	0.504611	1
13	4556	0.505565	1
14	4557	0.505309	1
15	4558	0.506456	1
16	4559	0.507992	1
17	4560	0.509312	1
18	4561	0.510019	1
19	4562	0.510519	1
20	4563	0.510613	1
21	4564	0.510707	1
22	4565	0.5102	1
23	4566	0.509498	1
24	4567	0.509603	1
25	4568	0.510307	1
26	4569	0.510804	1
27	4570	0.511296	1
28	4571	0.511183	1
29	4572	0.510271	1
30	4573	0.510369	1
...	...	...	...

This operation returns a new dataframe, which we can now save in a new variable:

```
df1 = data1[data1.mean_p .> 0.5, :]
```

All we need to do now, to extract the T numbers we need, is to obtain the first row from this new dataframe for each separate `ensemble`. How do we do this?

The answer is something known as **split–apply–combine**. This procedure allows us to first split a dataframe based on the values in one column (in our case, `ensemble`), then carry out an operation on each of the resulting dataframes individually, and then finally combine them back into a single dataframe. The `groupby` function from DataFrames.jl is used for the splitting; here, we split on the `ensemble` column:

```
df1 = groupby(df1, :ensemble)
```

Then, we use `combine` to apply an operation to each individual dataframe in the grouping. The function we apply is `minimum`, which returns the smallest element in an array. In this case, we wish to obtain the smallest `time` for each individual dataframe:

```
df1 = combine(df1, :time => minimum)
```

	ensemble	time_minimum
	Int64	Int64
1	1	4543
2	2	4431
3	3	4648
4	4	4421
5	5	4985
6	6	5691
7	7	5601
8	8	5243
9	9	4732
10	10	4749
11	11	5069
12	12	4342
13	13	4325
14	14	4558
15	15	3906
16	16	4658
17	17	4089
18	18	5145
19	19	4727
20	20	5209
21	21	4541
22	22	4239
23	23	3999
24	24	4741
25	25	4564
26	26	4873
27	27	4930
28	28	4502
29	29	4371
30	30	4853
…	…	…

The T values we're interested in are now in the `time_minimum` column; let's store them in a new variable for ease of use:

```
T1 = df1.time_minimum
```

```
100-element Vector{Int64}:
 4543
 4431
 4648
```

```
 4421
 4985
 5691
 5601
 5243
 4732
 4749
 5069
 4342
 4325

 6073
 4126
 4386
 4041
 6995
 5015
 4455
 4101
 4528
 5090
 3895
 5525
```

We can now perform all the same steps for the second set of simulations:

```
df2 = data2[data2.mean_p .> 0.5, :]
df2 = groupby(df2, :ensemble)
df2 = combine(df2, :time => minimum)
T2 = df2.time_minimum
```

```
100-element Vector{Int64}:
 4075
 4992
 4488
 5096
 4602
 4885
 5299
 4444
 5876
 4178
```

```
4397
4873
4753

4924
4594
4410
5430
4672
4682
4807
4276
4807
4444
5099
4310
```

c. Carrying out the statistical test

The t-test can be performed using `EqualVarianceTTest` from HypothesisTests.jl (see documentation):

```julia
EqualVarianceTTest(T1, T2)
```

```
Two sample t-test (equal variance)
----------------------------------
Population details:
    parameter of interest:   Mean difference
    value under h_0:         0
    point estimate:          -41.98
    95% confidence interval: (-192.0, 108.0)

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.5817

Details:
    number of observations:   [100,100]
    t-statistic:              -0.5518436540887174
    degrees of freedom:       198
    empirical standard error: 76.07227099371633
```

The "test summary" bit tells us that the test failed to reject the null hypothesis (which in this case states that the mean T values between the two sets of simulations do not differ). Thus, **we do not have any evidence that there is actually a difference in the speed with which trajectories reach $p = 0.5$ between the two sets of simulations**.

Bonus: pipes

To obtain the vector of T values from a simulation history, we did this:

```
df1 = data1[data1.mean_p .> 0.5, :]
df1 = groupby(df1, :ensemble)
df1 = combine(df1, :time => minimum)
T1 = df1.time_minimum
```

Notice that what we're doing here is to take the contents of a variable (`df1`), carry out some operation, and put the result back in the same variable. Julia, like many modern programming languages, support an operation known as the **pipe** which makes this kind of process simpler. The idea is that the result of an operation is piped into the following operation, whose result is then piped into the following operation, and so on. In Julia, the pipe operator is `|>`, and the following does exactly the same as the above code snippet:

```
using Pipe
@pipe data1[data1.mean_p .> 0.5, :] |> groupby(_, :ensemble) |> combine(_, :time => minimum)
```

```
100-element Vector{Int64}:
 4543
 4431
 4648
 4421
 4985
 5691
 5601
 5243
 4732
 4749
 5069
 4342
 4325

 6073
 4126
```

```
4386
4041
6995
5015
4455
4101
4528
5090
3895
5525
```

Notice that the underscore (`_`) symbol takes the place of the "anonymous" variable. The `@pipe` macro does the magic of populating this temporary variable for you, so you don't need to do it yourself.

What this means is that, to create the `T1` and `T2` arrays, all we need are the following two lines of code:

```
T1 = @pipe data1 |> _[_.mean_p .> 0.5, :] |> groupby(_, :ensemble) |> combine(_, :time => mi
T2 = @pipe data2 |> _[_.mean_p .> 0.5, :] |> groupby(_, :ensemble) |> combine(_, :time => mi
```

> **Tip**
>
> Whether you find using pipes more natural than explicitly creating temporary variables (such as `df1` above) boils down to personal preference and experience. If you're like me, you will initially find pipes confusing, but the more programming experience you gather, the more natural pipes become. Having said this, it's good to point out that using a pipe is never *necessary*; whatever you can do with a pipe you can also do without.

Bonus 2: plotting the distributions of T

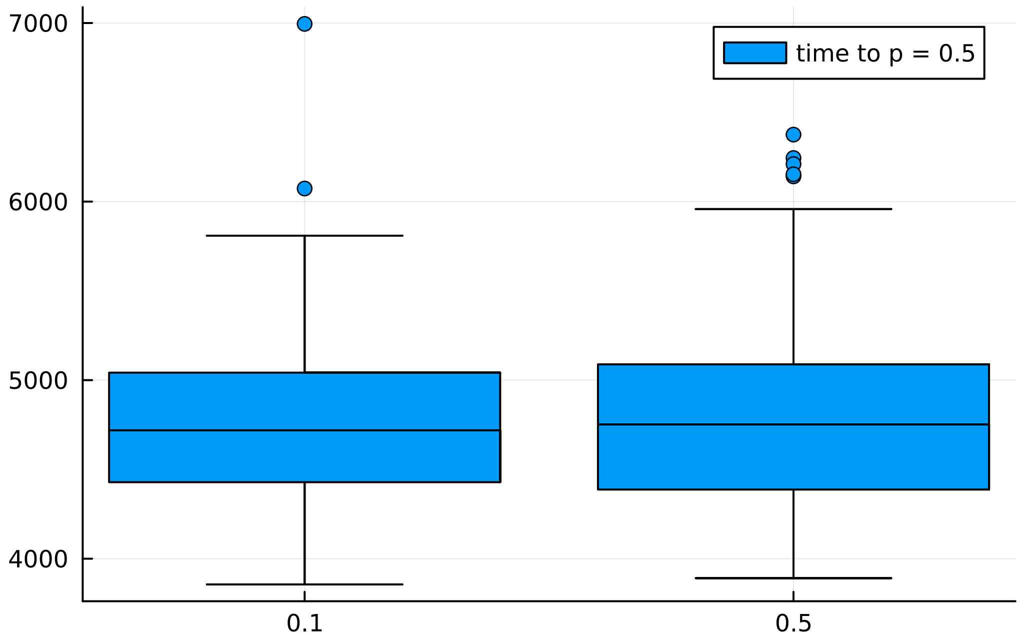
The statistical test suggests that there is no difference between the two sets of T numbers. Can we visualize this somehow? A usual way of doing this is by way of a boxplot. Here's how we can do it in Julia.

```
# load the StatsPlots package
using StatsPlots

# create dataframes: first column is the beta value, second column is the T values
set1 = DataFrame(beta="0.1", T=T1)
set2 = DataFrame(beta="0.5", T=T2)
```

```
# join these dataframes (literally, put one on top of the other, "vertical catenation")
sets = vcat(set1, set2)

# plot
@df sets boxplot(:beta, :T, label="time to p = 0.5")
```



We see that the distributions of T numbers overlap to a large extent; this is a visual representation of the fact that there is no difference between the two sets.

> **Tip**
>
> Peruse the StatsPlots.jl documentation to learn more about the `@df` macro and all the visualization functions you can use with dataframes.