Code Example SwissRe
Julia Module: FoxesAndRabbits

Tobias Seidler

Abstract

As I'm legally not allowed to showcase code from my current work, I wrote the toy example module FoxesAndRabbits. The module is written in Julia, is strongly-typed and contains some state-of-the-art numerical techniques such as hybrid models (models mixing first principle and data driven modelling), multi-threaded Monte-Carlo simulations as well as programming techniques such as custom types and multiple dispatch alongside a very minimal CI/CD pipeline including a test-suite with demonstrating unit tests. This document gives and overview about the models contained in the module, comments on some implementation details and leads through an use case.

1 Contained Files

- *./src/FoxesAndRabbits.jl*: The actual module (149 lines). The commented code after the end of the module definition can be uncommented and used as a script to run the usage example of the module demonstrated in this document.
- *./test/test.jl*: A test file for the pipeline, only testing the generation of artificial data as an unit testing example.
- *./.gitlab-ci.yml*: Very minimal CI/CD pipeline for GitLab.
- *./Manifest.toml*: The julia manifest in order to instantiate the module dependencies

2 Usage example

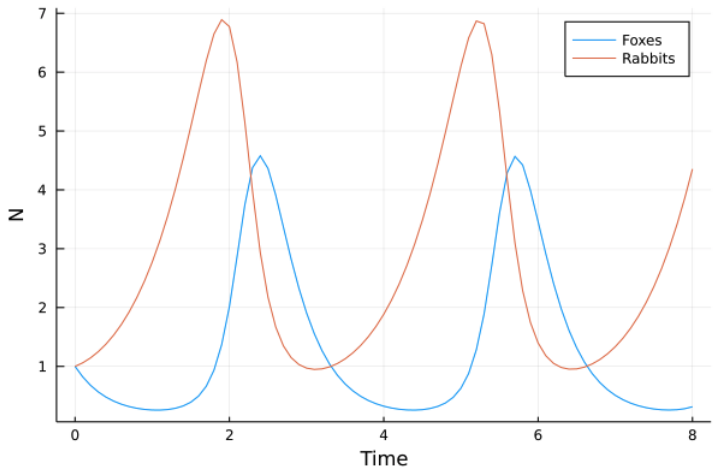
The example module is based on the Lotka-Volterra-Equations which model the dynamics of predator-prey populations. The predators in our case are foxes (F), the prey rabbits (R). The equations are:

$$\frac{\mathrm{d}R}{\mathrm{d}t} = R(a - bP) \tag{1}$$

$$\frac{\mathrm{d}F}{\mathrm{d}t} = -F(cR - d) \tag{2}$$

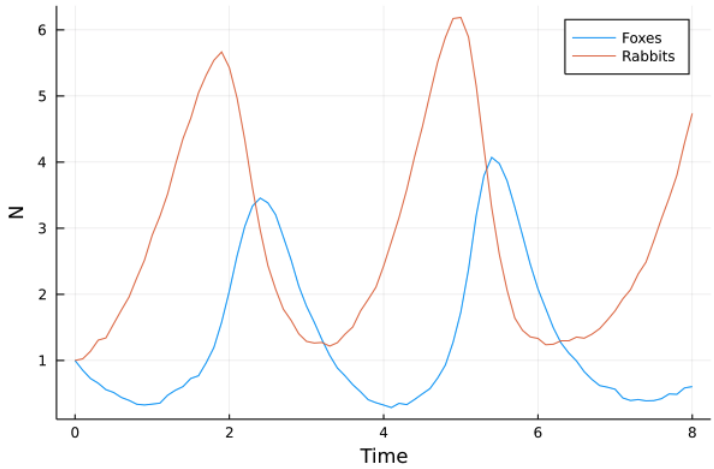
where a to d are birth and death rate parameters of the foxes and rabbits respectively. Let's assume some inital populations and generate some data over a span of 9 years and have a look at it:

```
#load module, generate some artificial data
using FoxesAndRabbits
initial_rabbits = initial_foxes = 1.0
t_end = 9.0
population = make_population_data([initial_rabbits, initial_foxes], t_end)
visualize(population)
```



We clearly see the typical regular, oscillating dynamics of a Lotka-Volterra-System. That's cool, but also a little bit boring to work with. Let's make the data more interesting & realistic:

```
#generate some artificial data using  SDEs
population = make_population_data([initial_rabbits, initial_foxes], 9.0, stochastic = true)
visualize(population)
```



What has happened in the background is that the module used stochastic differential equations (SDEs) instead of ordinary ones, hence introducing variation and noise. As we have some artificial data to work with now, let's do some data science!

Consider the following use case: The rabbits are very aware of the foxes in the forest and some of them have a life insurance with *RabbitLife*. *RabbitLife* on their hand would like to hedge there risk, so they approach *ForestRe* to get a reinsurance contract. *ForestRe* is very experienced in modelling fox populations, so they now the parameters ¨c¨ and ¨d¨. Concerning the population dynamics of rabbits they are very unsure about the underlying equation however, so they decide to fit a hybrid model - making this part of the ODE-system completely data driven. We will do so by using a shallow neural network NN, which uses the current population

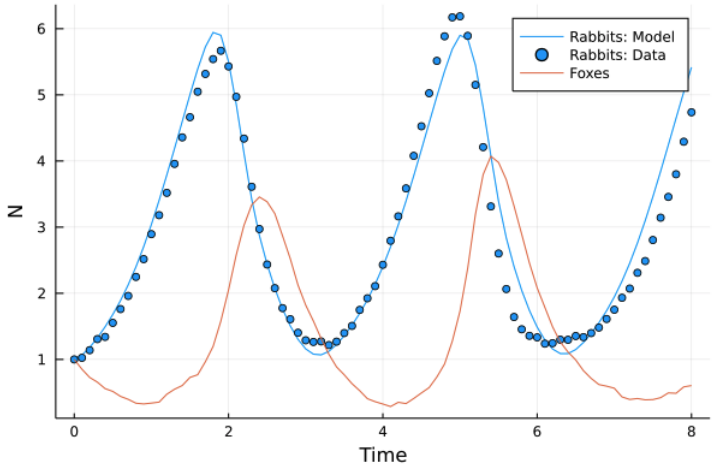
values of foxes and rabbits for the input layer as expression for the change in rabbits :

$$\frac{\mathrm{d}R}{\mathrm{d}t} = NN(R(t), F(t)) \tag{3}$$

$$\frac{\mathrm{d}F}{\mathrm{d}t} = -F(cR - d) \tag{4}$$

This hybrid model is implemented in the module, so let's train it:

```
#fit hybrid model
predicted_population = learn!(population)
visualize(predicted_population)
```



Note that

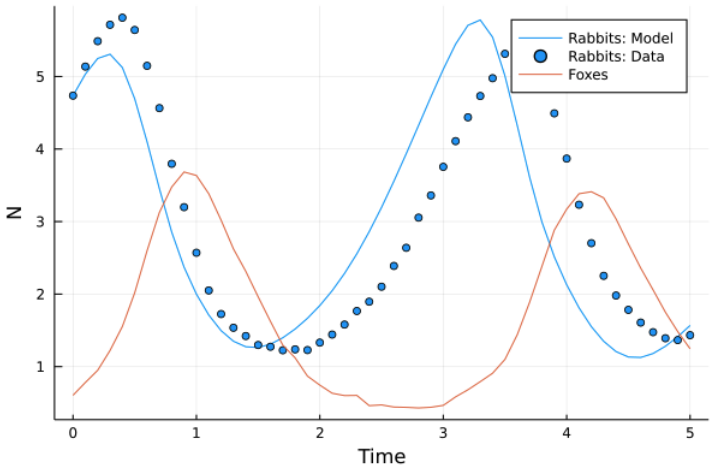
```
@time learn!(population)
```

yields less than 2 minutes (on an average laptop) and takes less than 500 iterations (default value) to fit a neural network living in a system of ODEs. This is due to the autodifferentation algorithms utilized in the code.

Of course *ForestRe* would like to validate this model, so they wait some time to have data that was not seen by the model in order to estimate the test performance.

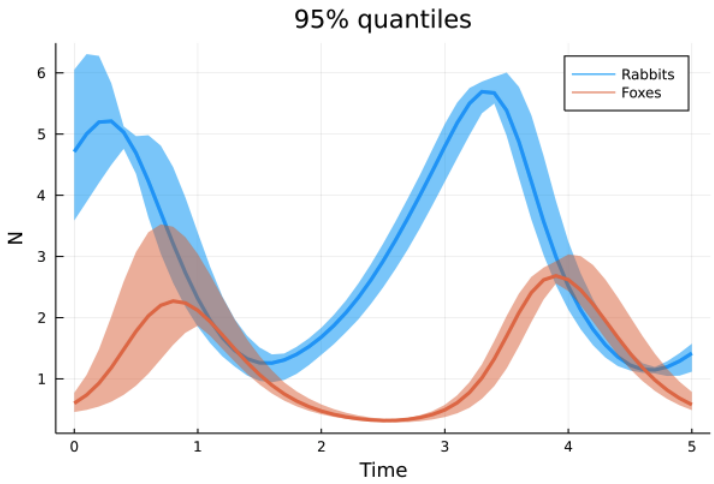
```
#make test set, continuing in time
initial_rabbits = population.rabbits[end]
initial_foxes = population.foxes[end]
t_end = 5.0
new_population = make_population_data([initial_rabbits, initial_foxes], t_end, stochastic = true)

#predict test set
predicted_rabbits = predict(predicted_population.parameters, new_population)
#make composed data struct so we can visualize as before
modelled_data = PopulationData(new_population.time, new_population.foxes, predicted_rabbits)
test_population = PredictedPopulationData(new_population, modelled_data, [])
visualize(test_population)
```



Which is a quite good extrapolation performance. The team at *ForrestRe* however is aware that models are not perfect. Also they know that the population data they use has some maximal (normalized) uncertainty. Based on that, they would like to run a Monte-Carlo simulation, varying the initial conditions according to the estimated maximum uncertainty for a total of 50 times in order to obtain the mean and 95%-quantiles of the model prediction.

```
#run ensemble prediction
max_uncertainty = 0.3
n = 50
sol = predict_ensemble(new_population,predicted_population.parameters, max_uncertainty,
                       trajectories = n)
visualize(sol)
```



This automatically used all available threads on the system to run the single trajectories and therefore is very fast. Also note that along this use case, we always used the *visualize* method to plot, independent of the underlying data types. This is possible due to a type-dispatched code layout, invoking the correct overloaded method in any case.