IVIVC Tutorial

Shubham Maddhashiya

Sep 5, 2019

1 Introduction

This is an introduction and walk-through `IVIVC.jl`, a submodule under software `Pumas.jl`.

In this tutorial, we will establish In Vitro In Vivo Correlation model and predict Plasma Concentration Profile using `IVIVC.jl` for a given ivivc dataset.

The Basic workflow of IVIVC.jl is:

1. Read the data
2. InVitro Modeling
3. Unit Impulse Response (UIR)
4. Deconvolution of InVivo data
5. Establishing InVitro InVivo Correlation Model
6. Validation

1.1 Installation

This package is under `Pumas.jl`, So if you haven't installed `Pumas.jl`. Please follow the instructions here

1.2 Getting Started

Load the package

```
using Pumas.IVIVC
```

1.3 Read the data

First of all, let's read the given data set, we will have three dataset file for InVitro, Reference InVivo, and InVivo data, one for each.

`read_vitro`, `read_uir` and `read_vivo` functions are to read and parse InVitro, Reference InVivo, and InVivo data csv file, respectively.

These functions take `path` to the data file (or `DataFrame` object of csv file) and accept keyword args for columns headers.

By Default, keyword args for `read_vitro` functions are `id=:id, time=:time, conc=:conc, formulation=:form`, for `read_uir` keyword args are `time=:time, conc=:conc, formulation=:form, dose=:dose`, and for `read_vivo` function, these are `time=:time, conc=:conc, formulation=:form, dose=:dose`.

```
vivo_data = read_vivo("./vivo_data.csv");
```

if you have a different header for any column name, you can pass column name as keyword arg. For example, if we have time column header `conc` for plasma concentration column, then we can do like this.

```
vivo_data = read_vivo("./vivo_data.csv", conc=:conc);
```

These functions return array of subject or batch sorted by their subject-id of batch-id. Each entry in this array is mapping of formulation type to corresponding data for a particular subject or batch of data.

If you have your data already in DataFrame object, you can pass it instead of `path` and rest will follow the same.

```
using CSV
df = CSV.read("./vivo_data.csv")
vivo_data = read_vivo(df);
```

Let's see how the data are packed inside `vivo_data` and take the `fast` formulation (one of the formulation types in our data) data for the first subject.

```
vivo_fast_data = vivo_data[1]["fast"]
```

```
15×5 DataFrames.DataFrame
 Row  id     time     conc     formulation  dose
      Int64  Float64  Float64  String       Int64

 1    1      0.0      0.0      fast         100
 2    1      0.5      7.219    fast         100
 3    1      1.0      39.7     fast         100
 4    1      1.5      69.29    fast         100
 5    1      2.0      92.86    fast         100
 6    1      3.0      118.3    fast         100
 7    1      4.0      107.2    fast         100
 8    1      6.0      68.77    fast         100
 9    1      8.0      45.91    fast         100
 10   1      10.0     28.1     fast         100
 11   1      12.0     18.28    fast         100
 12   1      14.0     12.2     fast         100
 13   1      16.0     8.19     fast         100
 14   1      20.0     4.447    fast         100
 15   1      24.0     2.703    fast         100
```

Time array of this data

```
vivo_fast_data.time
```

```
15-element Array{Float64,1}:
  0.0
  0.5
  1.0
  1.5
  2.0
  3.0
  4.0
  6.0
  8.0
 10.0
 12.0
 14.0
 16.0
 20.0
 24.0
```

Similar syntax follows for `read_uir` and `read_vitro` functions.

1.4 InVitro Modeling

In InVitro data, We have time series data of fraction dissolved amount of drug. Using this data, we have to fit a predefined model, which, of course will be continuous. Standard models available in IVIVC.jl are Emax, Weibull, Double Weibull and Makoid banakar. You can also use custom model by passing function of the model.

`estimate_fdiss` function takes object to vitro data and `Symbol` to model to fit (which are available in the package) or function of the same.

You can see available models using `get_avail_models` function. It returns dictionary of `Symbol` to model.

```
get_avail_models()
```

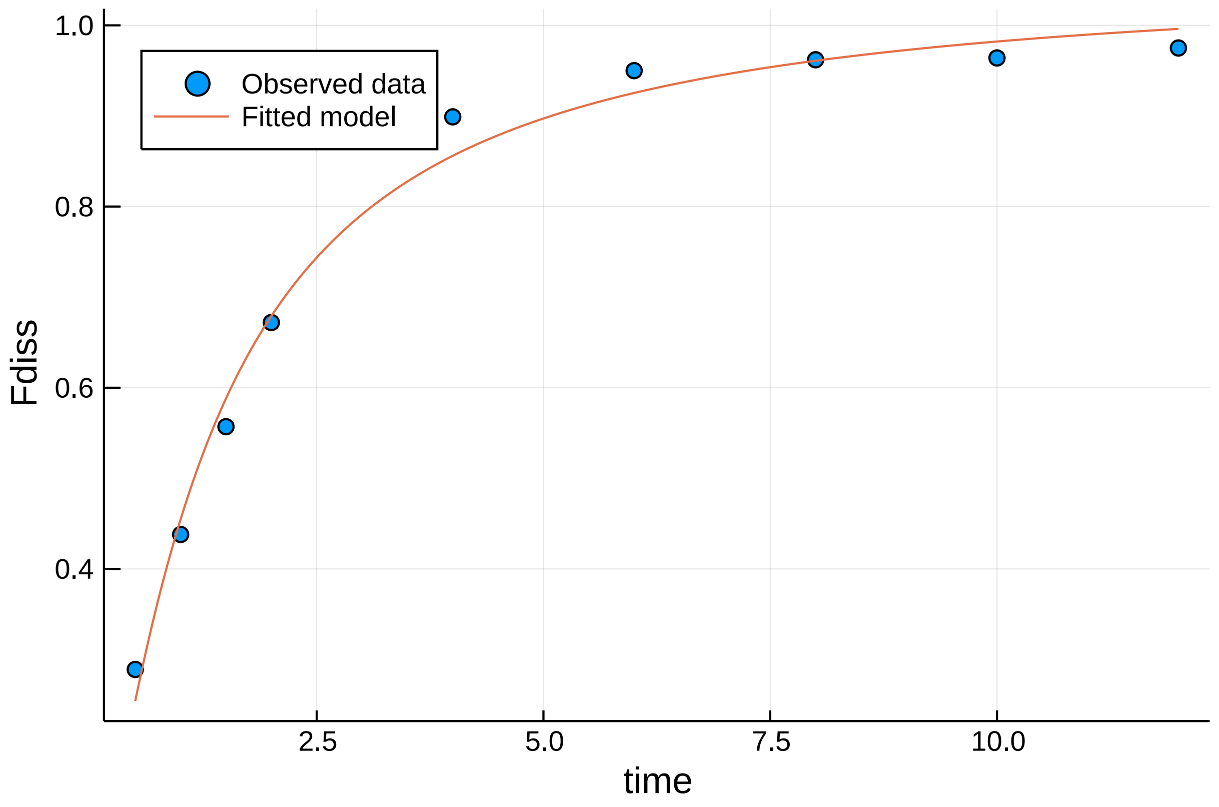
```
Dict{Symbol,Function} with 10 entries:
  :eng      => emax_ng
  :w        => weibull
  :m        => makoid
  :e        => emax
  :d_weibll => double_weibull
  :emax     => emax
  :makoid   => makoid
  :weibull  => weibull
  :emax_ng  => emax_ng
  :dw       => double_weibull
```

Let's fit Emax model to our InVitro data.

```
vitro_data = read_vitro("./vitro_data.csv");
vitro_fast_data = vitro_data[1]["fast"]
estimate_fdiss(vitro_fast_data, :emax);
```

We can call `plot` function on this instance of data to plot the fitted model.

```
using Plots
scatter(vitro_fast_data.time, vitro_fast_data.conc, label="Observed
    data", xlabel="time", ylabel="Fdiss")
plot!(vitro_fast_data, label="Fitted model", legend=:topleft)
```



Note: If you are passing function then you will also have to provide initial estimates of parameters, lower bounds and upper bounds. For example, let's fit Emax model again but passing our own function.

```
Emax_model(t, p) = @. (p[1] * (t ^ p[2])) / (p[3]^p[2] + t^p[2])
p0 = [vitro_fast_data.conc[end], 1.2, vitro_fast_data.time[2]]
lb = [0.0, 1.0, 0.0]
ub = [1.25, Inf, vitro_fast_data.time[end]]
estimate_fdiss(vitro_fast_data, Emax_model, p0=p0, lower_bound=lb, upper_bound=ub);
```

Optimized parameters are stored in `pmin` vector.

```
vitro_fast_data.pmin
```

```
3-element Array{Float64,1}:
 1.0541385268387788
 1.2547575044440769
 1.2461970228752515
```