Decision from Experience Behavior Modeling (DEBM): an open-source Python package for developing, evaluating, and visualizing behavioral models.

Ofir Yakobi¹ and Yefim Roth²

¹ – Department of Psychology, University of Waterloo

² – Department of Human Services, University of Haifa

Abstract

The last decade was characterized by an emphasis on enhancing reproducibility and replicability in the social sciences. To contribute to these efforts within the decision-making research field, we introduce DEBM (Decision from Experience Behavior Modeling) – an open-source Python package. The main goal of DEBM is to serve as a central collaborative pool of models and methods in the decision from experience domain. Specifically, it provides a convenient “playground” for developing models or experimenting with existing ones. DEBM includes many features such as multiprocessing, parameter estimation, visualization, and more. In this paper we cover the basic functionality of DEBM by simulating behavior using an existing model and given parameters, and recovering these parameters using grid search.

Introduction

Nowadays, researchers are required not only to master their study domain, statistics, and methodology, but also to extensively write computer code. Even with current open science best practices (e.g., Ince, Hatton, & Graham-Cumming, 2012) that encourage scholars to publish their code on open access platforms, using it might be challenging as researchers in the social sciences and related fields often lack formal programming training or other software engineering skills (such as debugging and quality assurance, Joppa et al., 2013; Merali, 2010). Furthermore, if any mistake is found in the code – correcting it in the original paper could prove a long and inefficient process (e.g., compared with code that is stored in a central repository).

The lack of formal programming training often results in a variety of software bugs for those who do try to program or modify others' code. These bugs can be roughly divided into two categories: those that stop code execution and throws an error message, and those that do not (Axelrod, 2014). The first set is less concerning as the user is aware of a problem in their code. Still, the latter type, which is the focus of this paper, may generate perfectly plausible output without alerting the user on any potential issues. A recent demonstration is reported in Eklund, Nichols, and Knutsson (2016), who found a major flaw in a popular statistics software used for analyzing functional magnetic resonance imaging (fMRI) data. Specifically, this bug inflated the false-positive rates, affecting thousands of published studies.

Software bugs are almost inevitable, and probably exist in most scientific software (Soergel, 2015). Soergel quotes an alarming estimate of 1.5%-5% errors in code that was written by *professional* programmers; it stands to reason that code written by graduate students without a formal training in programming, suffers from similar or even higher error

rates. One might argue that the availability of data and code should solve this problem; Yet, if the code is run only by few other researchers, it is unlikely that such software bugs would be discovered and fixed (Axelrod, 2014).

Moreover, those with limited training in programming will likely find using others' code effortful. If those researchers encounter a piece of code in a published (or under-review) manuscript, there must be a good reason for them to replicate or examine that code. A code repository that is actively maintained and used by a community of researchers, as we introduce here, could solve many of these issues. The rest of the paper is organized as follows: first, we present some motivating examples, illustrating how easily one can make code mistakes. Next, we introduce DEBM and demonstrate basic use cases.

How easy it is to make code mistakes

From our own experience, code mistakes are easy to make and hard to detect, regardless of one's own level of expertise. To demonstrate how easily software bugs are made and set the ground for our rationale, we included some personal examples of code errors that were made in our lab.

1. Integer uniform distribution in R: When simulating a uniform distribution of integers – for example, between 1 and 10, we used the *runif* and *round* functions. That is, we generated continuous uniform numbers, and then rounded them to the nearest number. Although initially it may sound perfectly reasonable, this is an incorrect implementation that trims values at the two extremes (Figure 1). Moreover, due to the

symmetrical nature of this mistake, the mean value of the distribution is correct, and only in later stages of our work this mistake was discovered.

2. Moving from verbal models to code: Recently, we coded a model that was verbally described in a research paper. When we tried to derive predictions from this model, our results did not make much sense. To investigate the issue, we asked a second researcher to code this model. After examining each other's' code, we noticed that we interpreted the model slightly differently. It was only after contacting the authors of the original paper that we found that both of our interpretations were different than the intentions of the authors.

Figure 1 - Histograms of the incorrect (left) and correct (right) implementation of a uniform integer distribution in R. Left: `round(runif(100000, min = 1, max = 10))`. Right: `sample(1:10, 100000, replace=TRUE)`.

Modeling human decisions from experience (see a review in Erev & Haruvy, 2016) involves the following four stages: (a) simulating environments and behavior, (b) making predictions, (c) estimating the models and (d) comparing them. These steps usually translate

into tens or hundreds of lines of code which deter many from trying to replicate others or writing their own models. To simplify modeling, we developed an open-source Python package with numerous built-in tools and models, including support for multiprocessing to significantly reduce processing time. We designed DEBM to be both flexible and easy to use for people with basic programming skills. To do so, DEBM consists of complimenting building blocks allowing researchers to run various models with minor code modifications. We chose Python because of its popularity for data modeling, while its readability and friendly syntax are aligned with our goal to make behavioral modeling approachable. In the next section, we introduce DEBM and demonstrate basic use cases with increasing complexity. Full documentation , more tutorials, as well as the list of models that are already integrated in DEBM can be found online at:

<https://git.io/JwHEu>.

Using DEBM

First steps

You should have Python installed on your machine, with the following packages: NumPy, Pandas, and Matplotlib. We recommend installing Anaconda (www.anaconda.com) which already includes Python and most of the popular packages. DEBM can be installed ¹using the command line by typing:

```
pip install debm
```

In the next sections, we will replicate the predictions made in Erev and Roth (2014; Problems 1 and 2, Table 2). In these problems, participants have to choose between two buttons, in a repeated 100 trials experiment (i.e., the “clicking paradigm”, Barron and Erev 2003; see Roth and Yakobi, 2022 for a review), as can be seen in Figure 2. Each button produced outcomes according to a pre-defined distribution (unknown to the participants), and the participants are presented with the outcomes of the two buttons. They gain (or lose, if the outcome is negative) points according to the prospect they chose and get paid accordingly.

¹ If you are using Anaconda, you should install packages using the Anaconda Prompt. From the start menu, open Anaconda Prompt and type *pip install debm*.

Figure 2 - The Clicking paradigm

Defining prospects

Open your favorite IDE (e.g., Spyder, PyCharm) and start with importing the required packages.

```
from debm Import * # Import all modules from debm
```

```
import numpy as np
```

import pandas as pdNext, we will define the prospects. Problem 1 describes a 100-trial repeated choice task between a status-quo option (always zero) and a risky option (1 with 90%, -10 otherwise). The syntax in DEBM for creating a prospect follows:

Prospect(trials, fun, oneOutcomePerRun, *args, **kwargs)

Trials defines the number of trials, *fun* the function that generates the outcomes or an array of outcomes, *oneOutcomePerRun* is a Boolean variable defining whether the functions *fun* returns a single value per run, or the whole set of outcomes.

Args, kwargs – Arguments for the function *fun*.

The following code creates the status-quo prospect:

```
StatQuo=Prospect(100,[0]*100,False)
```

We defined a prospect with 100 trials, passed an array of one hundred zeros (`'[0]*100'`) representing the outcomes array, and passed *False*, stating that we passed the whole set of outcomes (*oneOutcomePerRun=False*). We named this prospect *StatQuo*.

For the risky option (1 with 90%, -10 otherwise), we cannot simply pass an array of outcomes because the outcomes are stochastic. Thus, we need to pass a function that generates random outcomes, such as NumPy's random choice module. The syntax is:

```
np.random.choice(a, size=None, replace=True, p=None)
```

The following command returns an array of 100 numbers drawn randomly from the distribution described above (1, 90%; -10 otherwise):

```
np.random.choice([-10,1],100,True,[0.1,0.9])
```

The first argument is the array of possible outcomes, the second set the number of outcomes to return, the third one states that numbers are drawn with replacement, and the last one defines the respective probabilities of each outcome.

In the following line of code, we define a prospect of 100 outcomes named B1, based on the function `np.random.choice`. The third argument (*False*) states that this function will return an array of the full length of the outcomes (100 in our case). The next four arguments will be

passed directly to the function we specified (`np.random.choice`) each time new outcomes are generated.

```
B1=Prospect(100,np.random.choice,False,[-10,1],100,True,[0.1,0.9])
```

At this point, we should have two prospects in the memory, saved as *StatQuo* and *B1*, representing the two choices participants faced in each round of the decision making task.

Setting up models and making predictions

Erev and Roth (2014) suggest that participants' behavior can be captured by a Sample of K model. This model suggests that in each decision, people use some of their past experience. Specifically, they sample with replacement K previous trials, and consider this sample's mean of each prospect in their decision. Thus, this model has one free parameter – K (Kappa), which determines the size of the sample. This model is part of the DEBM package and can be instantiated with the following syntax:

Sample_of_K(parameters, prospect_list, number_of_simulations)

The first argument is a dictionary of parameters, the second is an array of prospects, and the last is the number of simulations that will be used to make predictions. The following line of code defines a sample of K model with the arbitrary name *sok1*, with a Kappa of 5 and the two prospects we defined earlier. We chose to simulate choices 10000 times every time we generate predictions.

```
sok1=Sample_of_K({'Kappa':5},[StatQuo,B1],10000)
```

The full code up to this point should look like the following:

```
from debm import *

import numpy as np

StatQuo=Prospect(100,[0]*100,False)

B1=Prospect(100,np.random.choice,False,[-10,1],100,True,[0.1,0.9])

sok1=Sample_of_K({'Kappa':5},[StatQuo,B1],10000)
```

We can now use sok1 to generate predictions. The following code generates predictions and saves them as an array in the new variable *choices1*.

```
choices1=sok1.Predict()
```

To print the mean over all trials, type `print(choices1.mean(axis=0))`. The results should be approximately 38% and 62% as in Erev and Roth (Table 2).

The predictions could be also plotted over trials, or over blocks with the following lines of code:

```
sok1.plot_predicted() # Plot over all trials

sok1.plot_predicted(4) # Plot over 4 block
```

Estimating models

In order to quantify how good a model fits actual behavior, we can calculate the mean squared deviation (MSD)² between the predictions and observed data. For that, we need a set

² DEBM can also calculate Log-Likelihood by replacing *MSD* with *LL*, see the online tutorial for more details.

of observed data to compare against our model's predictions. For our purposes, instead of using actual behavior data³, we will generate synthetic data using another model. In the following lines of code, we define a new model named *simulated_agents*, which is also based on the sample of K model. However, here we set Kappa to 10 (sample size of 10), generate predictions and save them in *simulated_outcomes*.

```
simulated_agents = Sample_of_K({'Kappa':10},[StatQuo,B1],10000)

simulated_outcomes= simulated_agents.Predict()
```

We will save the outcomes stored in *simulated_outcomes* in our original model (*sok1*) as the observed behavior:

```
sok1.set_obs_choices(simulated_outcomes)
```

In order to calculate the MSD for *sok1*, we will use the *loss* method. We specify the loss function and the scope of the loss calculation as follows:

Model.loss(method, scope) where *method* could be MSD or LL (for log-likelihood), and the *scope* could be pw (prospect-wise – the loss function is calculated using the mean predictions of each prospect over all trials), or bw (bit-wise – the loss function is calculated over each trial and prospect). The following line of code outputs the prospect-wise MSD for the *sok1* model by comparing the predictions of the model, and the stored “observed” outcomes (generated earlier using a second model):

³ Data can be loaded from a CSV file or any other source, with the end goal of having a numerical array of the size n by p , where n is the number of trials, and p is the number of prospects. See online example for such cases: <https://github.com/ofiryakobi/debm>

```
sok1.loss('MSD','pw')4
```

The relatively high MSD (~0.04) reflects the difference in Kappa between the original model (K=5) and the simulated behavior (K=10). Next, we will try to find an optimal Kappa that best captures the observed outcomes. Because in this case we generated the observed outcomes, we can expect to find the optimal Kappa to be 10.

Model fitting

DEBM supports multiprocessing (i.e., utilizing more than one CPU in parallel for reducing computation time). The following line of code will set sok1 model to use four CPUs (or less, if your computer has less than four CPUs) in the fitting process:

```
sok1.mp=4
```

For fitting, we will use the *OptimizeBF* method for finding the optimal K for our data.

OptimizeBF is using brute-force (grid-search) to find the best set of parameters, with the following syntax:

OptimizeBF(*pars_dicts*, *progress_bar*, *loss_function*, *scope*)

The first argument, *pars_dicts*, is a list of dictionaries. Each dictionary defines all the parameters of the model (in this example, only K), and all the dictionaries in the list are examined. For example, if we want to examine all Kappas from 1 to 50, *pars_dicts* should be:

⁴ Note that the *loss* method uses the predictions stored in the model after running `sok1.Predict()`.

```
[{'Kappa':1}, {'Kappa':2}, {'Kappa':3},.... {'Kappa':50}]
```

We can use the helper function *makeGrid* to create this long list without manually setting the values:

```
pspace=makeGrid({'Kappa':range(1,51)})
```

The second argument of *OptimizeBF* is *Progress_bar* – a Boolean variable stating if we want to see a progress bar during the fitting process. *Loss_function* and *scope* are the two arguments we introduced before that are passed to the loss function (MSD or Log-likelihood, and bit-wise or prospect-wise, respectively).

The following line of code will find the best Kappa for our data, and store the fitting results as a dictionary in *res1*:

```
res1=sok1.OptimizeBF(pspace,True,'MSD','pw')
```

After the fitting is completed, *res1* will contain the best K and the corresponding MSD, as well as the list of Kappas that were examined (i.e., 1-50) and their corresponding MSDs. The following line will print the optimal Kappa and its corresponding MSD:

```
print('The best Kappa is:',res1['bestp']['Kappa'], 'with an MSD of',res1['minloss'])
```

The next line will plot the loss function over the different sets of parameters:

```
resPlot(res1)
```

Figure 3 - The loss function for different K values.

Thus, we successfully recovered the behavior of the synthetic agents (which were simulated with a Kappa of 10).

Conclusions

In order to enhance reproducibility and replicability (Munafò et al., 2017) in the decisions from experience research field, we introduced DEBM – an open-source Python package. DEBM allows evaluating models and creating predictions and visualizations with a few lines of code, and embedding this code in research papers for an easy reproduction of results by other researchers.

We demonstrated a basic use-case of DEBM: using an existing model for finding a best set of parameters for a given dataset (see Figure 4 for an overview of the process and code). By working with synthetic data, rather than real human behavior, we also showed how to generate expected behavior for a given model and a given set of parameters. This can be used to assess parameter recoverability of a model (e.g., by generating many synthetic datasets with known parameters, and correlate these parameters with the fitted parameters).

Other DEBM features that are not covered here include creating correlated or dynamic prospects, creating your own model and embedding it in DEBM, individual differences, multi-game fitting (i.e., when participant complete more than one decision task), exporting predictions to a CSV file, using other optimization methods, and more. These features – which are fully embedded or compatible with DEBM, are described and demonstrated in <https://git.io/JwHEu>.

(1) Define the choice problem

```
Safe=Prospect(100,np.random.normal,False,0,1,100)  
Risky=Prospect(100,np.random.normal,False,2,4,100)
```

(2) Define the decision-making process

```
sok1=Sample_of_K({'Kappa':5},[Safe,Risky],1000)
```

Model: Sample of K
1. If trial-1: make random choice;
2. If trial-1: Sample K past trials with replacement;
Calculate mean of each prospect;
Choose the prospect with the highest mean;
....

(3) Store agents' behavior

```
data_320ss=np.genfromtxt('experiment_results.csv',delimiter=',')  
sok1.set_obs_choices(data_320ss)
```

(4) Fit the model

```
pspace=makeGrid({'Kappa':range(1,101)})  
res1=sok1.OptimizeBF(pspace,True,'MSD','pw')
```

(5) Examine fitting results

```
print('The best Kappa is',res1['bestp']]['Kappa'], 'with an MSD  
of',res1['minloss'])  
resPlot(res1)  
sok1.plot_fit(4) #Plot predictions against behavior
```

The best Kappa is 8 with an MSD of 0.0003

Figure 4 - The process of model fitting using DEBM

REFERENCES

- Axelrod, V. (2014). Minimizing bugs in cognitive neuroscience programming. *Frontiers in Psychology*. <https://doi.org/10.3389/fpsyg.2014.01435>
- Barron, G., & Erev, I. (2003). Small Feedback-based Decisions and Their Limited Correspondence to Description-based Decisions. *Journal of Behavioral Decision Making*. <https://doi.org/10.1002/bdm.443>
- Eklund, A., Nichols, T. E., & Knutsson, H. (2016). Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates. *Proceedings of the National Academy of Sciences of the United States of America*, 113(28). <https://doi.org/10.1073/pnas.1602413113>
- Erev, I., & Haruvy, E. (2016). Learning and the Economics of Small Decisions. In *The Handbook of Experimental Economics, Volume Two*. <https://doi.org/10.1515/9781400883172-011>
- Erev, I., & Roth, A. E. (2014). Maximization, learning, and economic behavior. *Proceedings of the National Academy of Sciences of the United States of America*. <https://doi.org/10.1073/pnas.1402846111>
- Ince, D. C., Hatton, L., & Graham-Cumming, J. (2012). The case for open computer programs. *Nature*, 482(7386). <https://doi.org/10.1038/nature10836>
- Joppa, L. N., McInerny, G., Harper, R., Salido, L., Takeda, K., O'Hara, K., ... Emmott, S. (2013). Troubling trends in scientific software use. *Science*. <https://doi.org/10.1126/science.1231535>
- Merali, Z. (2010). Error: Why scientific programming does not compute. *Nature*, 467(7317).
- Munafò, M. R., Nosek, B. A., Bishop, D. V. M., Button, K. S., Chambers, C. D., Percie Du Sert, N., ... Ioannidis, J. P. A. (2017). A manifesto for reproducible science. *Nature Human Behaviour*. <https://doi.org/10.1038/s41562-016-0021>
- Roth, Y., & Yakobi, O. (2022). On Attention to Information: The Checking Paradox. In *Handbook of Experimental Finance*.

Soergel, D. A. W. (2015). Rampant software errors may undermine scientific results.
F1000Research, 3. <https://doi.org/10.12688/f1000research.5930.2>