Fitting Human Decision Making Models using Python

Alejandro Weinstein[‡§*], Wael El-Deredy[‡§], Stéren Chabert[‡], Myriam Fuentes[‡]

Abstract—A topic of interest in experimental psychology and cognitive neuroscience is to understand how humans make decisions. A common approach involves using computational models to represent the decision making process, and use the model parameters to analyze brain imaging data. These computational models are based on the Reinforcement Learning (RL) paradigm, where an agent learns to make decisions based on the difference between what it expects and what it gets each time it interacts with the environment. In the typical experimental setup, subjects are presented with a set of options, each one associated to different numerical rewards. The task for each subject is to learn, by taking a series of sequential actions, which option maximizes their total reward. The sequence of actions made by the subject and the obtained rewards are used to fit a parametric RL model. The model is fit by maximizing the likelihood of the parameters given the experiment data. In this work we present a Python implementation of this model fitting procedure. We extend the implementation to fit a model of the experimental setup known as the "contextual bandit", where the probabilities of the outcome change from trial to trial depending on a predictive cue. We also developed an artificial agent that can simulate the behavior of a human making decisions under the RL paradigm. We use this artificial agent to validate the model fitting by comparing the parameters estimated from the data with the known agent parameters. We also present the results of a model fitted with experimental data. We use the standard scientific Python stack (NumPy/SciPy) to compute the likelihood function and to find its maximum. The code organization allows to easily change the RL model. We also use the Seaborn library to create a visualization with the behavior of all the subjects. The simulation results validate the correctness of the implementation. The experimental results shows the usefulness and simplicity of the program when working with experimental data. The source code of the program is available at https://github.com/aweinstein/FHDMM.

Index Terms—decision making modeling, reinforcement learning

Introduction

As stated by the classic work of Rescorla and Wagner [Res72]

> *"... organisms only learn when events violate their expectations. Certain expectations are built up about the events following a stimulus complex; expectations initiated by that complex and its component stimuli are then only modified when consequent events disagree with the composite expectation."*

This paradigm allows to use the framework of Reinforcement Learning (RL) to model the process of human decision making. In the fields of experimental psychology and cognitive neuroscience these models are used to fit experimental data. Once such a model is fitted, one can use the model parameters to draw conclusions about individual difference between the participants. The model parameters can be co-varied with imaging data to make observations about the neural mechanisms underpinning the learning process. For an in-depth discussion about the connections between cognitive neuroscience and RL see chapter 16 of [Wie12].

In this work we present a Python program able to fit experimental data to a RL model. The fitting is based on a maximum likelihood approach [Cas02]. We present simulation and experimental data results.

* *Corresponding author: alejandro.weinstein@uv.cl*
‡ *Universidad de Valparaiso, Chile*
§ *Advanced Center for Electrical and Electronic Engineering*

A Decision Making Model

In this section we present the model used in this work to describe how an agent (either an artificial one or a human) learns to interact with an environment. The setup assumes that at the discrete time t the agent selects action a_t from the set $\mathscr{A} = \{1, \ldots, n\}$. After executing that action the agent gets a reward $r_t \in \mathbb{R}$, according to the properties of the environment. Typically these properties are stochastic and are defined in terms of probabilities conditioned by the action. This sequence is repeated T times. The objective of the agent is to take actions to maximize the total reward

$$R = \sum_{t=1}^{T} r_t.$$

In the RL literature, this setup is known as the "n-armed bandit problem" [Sut98].

According to the Q-learning paradigm [Sut98], the agent keeps track of its perceived value for each action through the so called action-value function $Q(a)$. When the agent selects action a_t at time t, it updates the action-value function according to

$$Q_{t+1}(a_t) = Q_t(a_t) + \alpha(r_t - Q_t(a_t)),$$

where $0 \leq \alpha \leq 1$ is a parameter of the agent known as *learning rate*. To make a decision, the agent selects an action at random from the set $\mathscr{A}$ with probabilities for each action given by the softmax rule

$$P(a_t = a) = \frac{e^{\beta Q_t(a)}}{\sum_{i=1}^{n} e^{\beta Q_t(a_i)}},$$

where $\beta > 0$ is a parameter of the agent known as *inverse temperature*.

In this work we consider the case were the probabilities associated to the reward, in addition to being conditioned by the action, are also conditioned by a context of the environment. This context change at each time step and is observed by the agent. This means that the action-value function, the softmax rule, α,

and β also depend on the current context of the environment. In this scenario, the update action-value and softmax rules become

$$Q_{t+1}(a_t, c_t) = Q_t(a_t, c_t) + \alpha_{c_t}(r_t - Q_t(a_t, c_t)) \quad (1)$$

$$P(a_t = a, c_t) = \frac{e^{\beta_{c_t} Q_t(a, c_t)}}{\sum_{i=1}^{n} e^{\beta_{c_t} Q_t(a_i, c_t)}}, \quad (2)$$

where c_t is the cue observed at time t. In the literature, this setup is known as *associative search* [Sut98] or *contextual bandit* [Lan08].

In summary, each interaction, or trial, between the agent and the environment starts by the agent observing the environment context, or cue. Based on that observed cue and on what the agent has learned so far from previous interactions, the agent makes a decision about what action to execute next. It then gets a reward (or penalty), and based on the value of that reward (or penalty) it updates the action-value function accordingly.

Fitting the Model Using Maximum Likelihood

In cognitive neuroscience and experimental psychology one is interested in fitting a decision making model, as the one described in the previous section, to experimental data [Daw11].

In our case, this means to find, given the sequences of cues, actions and rewards

$$(c_1, a_1, r_1), (c_2, a_2, r_2) \ldots, (c_T, a_T, r_T)$$

the corresponding α_c and β_c. The model is fit by maximizing the likelihood of the parameters α_c and β_c given the experiment data. The likelihood function of the parameters is given by

$$\mathscr{L}(\alpha_c, \beta_c) = \prod_{t=1}^{T} P(a_t, c_t), \quad (3)$$

where the probability $P(a_t, c_t)$ is calculated using equations (1) and (2).

Once one has access to the likelihood function, the parameters are found by determining the α_c and β_c that maximize the function. In practice, this is done by minimizing the negative of the logarithm of the likelihood (NLL) function [Daw11]. In other words, the estimate of the model parameters are given by

$$\widehat{\alpha}_c, \widehat{\beta}_c = \underset{0 \le \alpha \le 1, \beta \ge 0}{\operatorname{argmin}} \; -\log(\mathscr{L}(\alpha_c, \beta_c)). \quad (4)$$

The quality of this estimate can be estimated through the inverse of the Hessian matrix of the NLL function evaluated at the optimum. In particular, the diagonal elements of this matrix correspond to the standard error associated to α_c and β_c [Daw11].

Details about the calculation of the likelihood function and its optimization are given in the *Implementation and Results* section.

Experimental Data

The data used in this work consists on the record of a computerized card game played by 46 participants of the experiment. The game consists of 360 trials. Each trial begins with the presentation of a cue during one second. This cue can be a circle, a square or a triangle. The cue indicates the probability of winning on that trial. These probabilities are 20%, 50% and 80%, and are unknown to the participants. The trial continues with the presentation of four cards with values 23, 14, 8 and 3. The participant select one of these cards and wins or loses the amount of points indicated on the selected card, according to the probabilities defined by the cue. The outcome of the trial is indicated by a stimulus that lasts one second (an arrow pointing up for winning and down for losing). The trial ends with a blank inter-trial stimulus that also last one second. Figure 1 shows a schematic of the stimulus presentation. Participants were instructed to maximize their winnings and minimize their losses. See [Mas12] for more details about the experimental design.

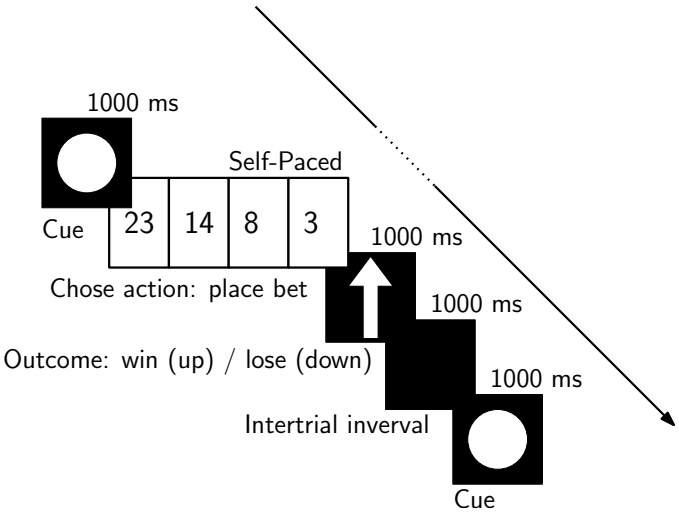

Fig. 1: *Schematic of the stimulus presentation. A trial begins with the presentation of a cue. This cue can be a circle, a square or a triangle and is associated with the probability of winning in that trial. These probabilities are 20%, 50% and 80%, and are unknown to the participants. The trial continues with the presentation of four cards with values 23, 14, 8 and 3. After selecting a card, the participant wins or lose the amount of points indicated on the card, according to the probabilities associated with the cue. The outcome of the trial is indicated by a stimulus, where the win or lose outcome is indicated by an arrow up or down, respectively [Mas12].*

Note that in the context with probability of winning 50% any strategy followed by the subject will produce an expected reward of 0. Thus, there is nothing to learn for this context. For this reason, we do not consider this context in the following analysis.[1]

The study was approved by the University of Manchester research ethics committee. Informed written consent was obtained from all participants.

Implementation and Results

Before testing the experimental data, we present an implementation of an artificial agent that makes decisions according to the decision model presented above. This artificial agent allows us to generate simulated data for different parameters, and then use the data to evaluate the estimation algorithm.

The code for the artificial agent is organized around two classes. The class `ContextualBandit` provides a simulation of the environment. The key two methods of the class are `get_context` and `reward`. The `get_context` method sets the context, or cue, for the trial uniformly at random and returns its value. The `reward` method returns the reward, given the selected action. The value of the reward is selected at random with the probability of winning determined by the current context. The following code snippet shows the class implementation.

1. This condition was included in the original work to do a behavioral study not related to decision making.

```
class ContextualBandit(object):
    def __init__(self):
        # Contexts and their probabilities of
        # winning
        self.contexts = {'punishment': 0.2,
                         'neutral': 0.5,
                         'reward': 0.8}
        self.actions = (23, 14, 8, 3)
        self.n = len(self.actions)
        self.get_context()

    def get_context_list(self):
        return list(self.contexts.keys())

    def get_context(self):
        k = list(self.contexts.keys())
        self.context = np.random.choice(k)
        return self.context

    def reward(self, action):
        p = self.contexts[self.context]
        if np.random.rand() < p:
            r = action
        else:
            r = -action
        return r
```

The behavior of the artificial agent is implemented in the `ContextualAgent` class. The class is initialized with parameters learning rate `alpha` and inverse temperature `beta`. Then, the `run` method is called for each trial, which in turn calls the `choose_action` and `update_action_value` methods. These methods implement equations (2) and (1), respectively. The action-value function is stored in a dictionary of NumPy arrays, where the key is the context of the environment. The following code snippet shows the class implementation.

```
class ContextualAgent(object):
    def __init__(self, bandit, beta, alpha):
        # ...

    def run(self):
        context = self.bandit.get_context()
        action = self.choose_action(context)
        action_i = self.actions[action]
        reward = self.bandit.reward(action_i)
        # Update action-value
        self.update_action_value(context, action,
                                 reward)

    def choose_action(self, context):
        p = softmax(self.Q[context], self.beta)
        actions = range(self.n)
        action = np.random.choice(actions, p=p)
        return action

    def update_action_value(self, context, action,
                            reward):
        error = reward - self.Q[context][action]
        self.Q[context][action] += self.alpha * error
```

The function `run_single_softmax_experiment` shows how these two classes interact:

```
def run_single_softmax_experiment(beta, alpha):
    cb = ContextualBandit()
    ca = ContextualAgent(cb, beta=beta, alpha=alpha)
    trials = 360
    for _ in range(steps):
        ca.run()
```

In this function, after the classes are initialized, the `run` method is run once per trial. The results of the simulation are stored in a pandas dataframe (code not shown). Figure 2 shows an example of a simulation for $\alpha = 0.1$ and $\beta = 0.5$ (same value for all contexts). The top and bottom plots show the actions made by the agent when it observes the context with a probability of winning of 80% and 20%, respectively. The plots also show a blue and red vertical bar for each trial where the agent won or lost, respectively. We observe that the agent learned to made actions close to the optimal ones.

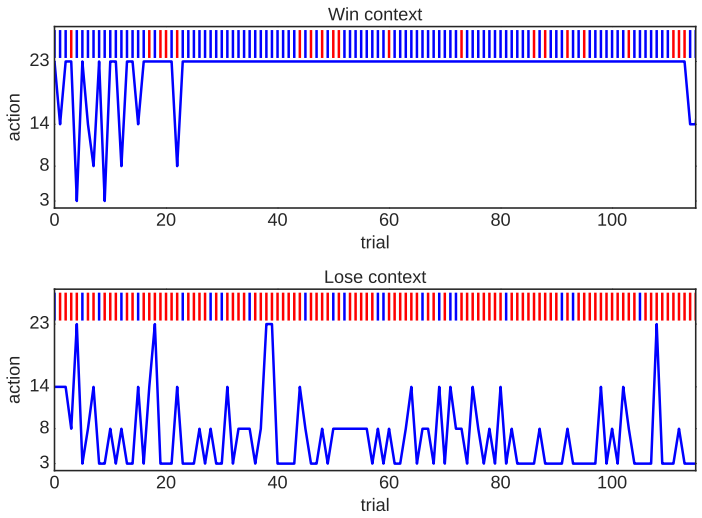

Fig. 2: *Simulation results for an experiment with $\alpha = 0.1$ and $\beta = 0.5$. Actions made by the agent when the context has a probability of winning of 80% (top) and 20% (bottom). The plots also show a vertical bar for each trial indicating if the agent won (blue) or lose (red).*

The key step in the estimation of the parameters is the computation of the likelihood function described by equation (3). As explained before, for numerical reasons one works with the negative of the likelihood function of the parameters $-\log(\mathscr{L}(\alpha_c, \beta_c))$. The following code snippet describes the steps used to compute the negative log likelihood function.

```
prob_log = 0
Q = dict([[cue, np.zeros(self.n_actions)]
          for cue in self.cues])
for action, reward, cue in zip(actions, rewards, cues):
    Q[cue][action] += alpha * (reward - Q[cue][action])
    prob_log += np.log(softmax(Q[cue], beta)[action])
prob_log *= -1
```

After applying the logarithmic function to the likelihood function, the product of probabilities becomes a sum of probabilities. We initialize the variable `prob_log` to zero, and then we iterate over the sequence (c_t, a_t, r_t) of cues, actions, and rewards. These values are stored as lists in the variables `actions`, `rewards`, and `cues`, respectively. The action value function $Q(a_t, c_t)$ is represented as a dictionary of NumPy arrays, where the cues are the keys of the dictionary. The arrays in this dictionary are initialized to zero. To compute each term of the sum of logarithms, we first compute the corresponding value of the action-value function according to equation (1). After updating the action-value function, we can compute the probability of choosing the action according to equation (2). Finally we multiply the sum of probabilities by negative one.

Once we are able to compute the negative log-likelihood function, to find the model parameter we just need to minimize this function, according to equation (3). Since this is a con-

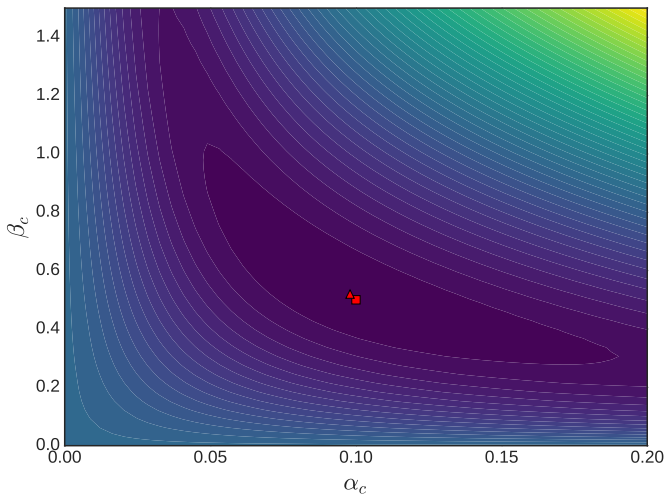

Fig. 3: *Likelihood function of the parameters given the data of the artificial agent for the win context. The data correspond to an agent operating with $\alpha_c = 0.1$ and $\beta_c = 0.5$ (red square). The model parameters estimated using the maximum likelihood are $\widehat{\alpha}_c = 0.098$ and $\widehat{\beta}_c = 0.508$ (red triangle).*

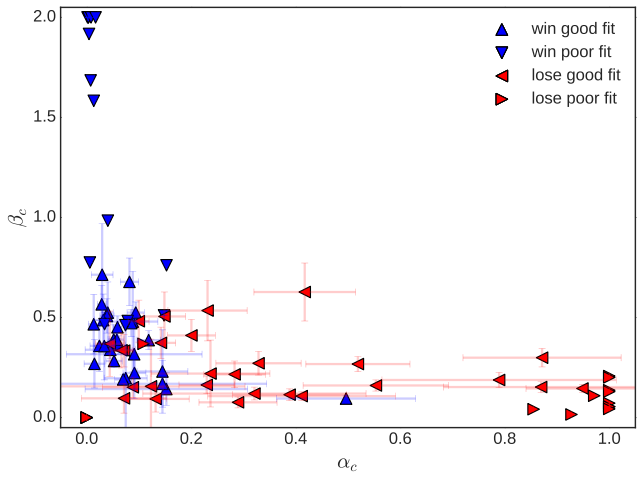

Fig. 4: *Estimated model parameters. Each point shows the estimated $\widehat{\alpha}_c$ and $\widehat{\beta}_c$ for each subject and and context. Blue upside/down triangles are the estimates for the "win context" (probability of winning 80%). Red left/right triangles are the estimates for the "lose context" (probability of winning 20%). We show the standard error for the estimates that are a good fit.*

strained minimization problem, we use the L-BFGS-B algorithm [Byr95], available as an option of the `minimize` function of the `scipy.optimize` module. The following code snippet shows the details.

```
r = minimize(self.neg_log_likelihood, [0.1,0.1],
             method='L-BFGS-B',
             bounds=(0,1), (0,2))
```

This function also computes an approximation of the inverse Hessian matrix evaluated at the optimum. We use this matrix to compute the standard error associated to the estimated parameter.

Before using our implementation of the model estimation method with real data, it is important, as a sanity check, to test the code with the data generated by the artificial agent. Since in this case we know the actual values of the parameters, we can compare the estimated values with the real ones. To run this test we generate 360 trials (same number of trials as in the experimental data) with an agent using parameters $\alpha_c = 0.1$ and $\beta_c = 0.5$. Figure 3 shows the likelihood function of the parameters. Using the maximum likelihood criteria we find the estimated parameters $\widehat{\alpha}_c = 0.098$ and $\widehat{\beta}_c = 0.508$. The actual values of the agent parameters are shown with a red square and the estimated parameters with a red plus sign. This result shows that our implementation is calculating the parameter estimation as expected. The NLL function and the quality of the estimation is similar for other parameter settings.

It is good practice to visualize the raw experimental data before doing any further analysis. In this case, this means showing the actions taken by each subject for each trial. Ideally, we wish to show the behaviors of all the subject for a given context in a single figure, to get an overview of the whole experiment. Fortunately, the Seaborn library [Was16] allows us to do this with little effort. Figure 5 shows the result for the context with a probability of winning of 80%. We also add vertical lines (blue for winning and red for losing) for each trial.

Finally, we can fit a model for each subject. To do this we perform the maximum likelihood estimation of the parameters using the experimental data. Figure 4 shows the estimated $\widehat{\alpha}_c$ and $\widehat{\beta}_c$ for each subject and context. Blue upside/down triangles are the estimates for the "win context" (probability of winning 80%). Red left/right triangles are the estimates for the "lose context" (probability of winning 20%). We show the standard error for the estimates that are a good fit, declared when the standard error is below 0.3 for both $\widehat{\alpha}_c$ and $\widehat{\beta}_c$.

We notice from this result that not all behaviors can be properly fitted with the RL model. This is a known limitation of this model [Daw11]. We also observe that in general the parameters associated with the "lose context" exhibit larger values of learning rate α and smaller values of inverse temperature β. Although at this point of our research it is not clear the reason for this difference, we conjecture that this phenomenon can be explained by two factors. First, in the lose context people bet smaller amounts after learning that the probability of wining is low in this context. This means that the term $(r_t - Q_t(a_t, c_t))$ in equation (1) is smaller compared to the win context. Thus, a larger learning rate is needed to get an update on the action value function of a magnitude similar to the win context.[2] Secondly, it is known that humans commonly exhibit a loss aversion behavior [Kah84]. This can explain, at least in part, the larger learning rates observed for the lose context, since it could be argued that people penalized more their violation of their expectations, as reflected by the term $(r_t - Q_t(a_t, c_t))$ of equation (1), when they were experiencing the losing situation.

In terms of execution time, running a simulation of the artificial agent consisting of 360 steps takes 34 milliseconds; minimizing the NLL function for a single subject takes 21 milliseconds; and fitting the model for all 43 subjects, including loading the experimental data from the hard disk, takes 14 seconds. All these measurements were made using the IPython `%timeit` magic function in a standard laptop (Intel Core i5 processor with 8 gigabytes of RAM).

2. This difference suggests that the experimental design should be modified to equalize this effect between the contexts.

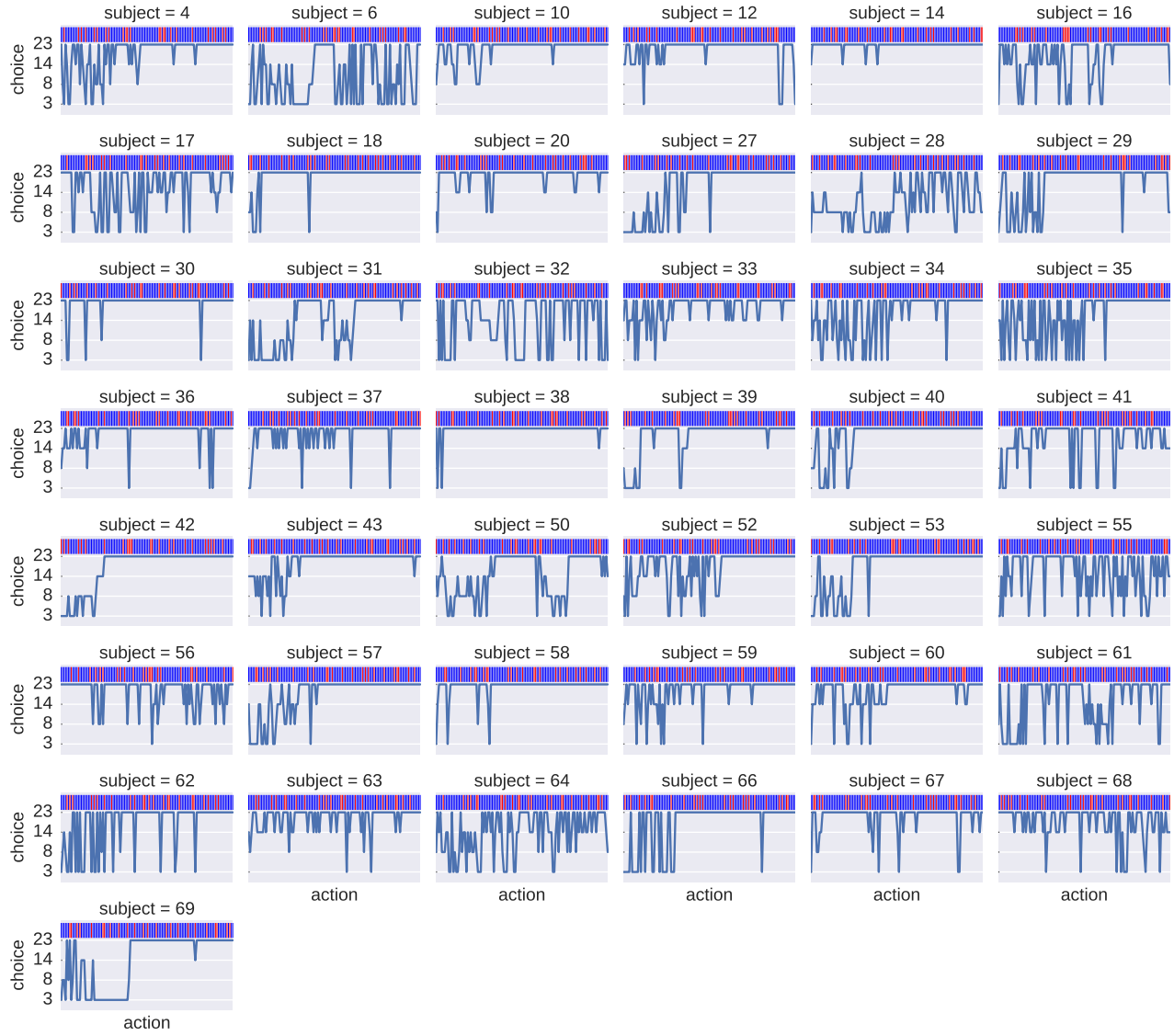

Fig. 5: *Actions taken by all the subjects for trials with context associated to the 80% probability of winning. The vertical bars show if the subject won (blue) or lost (red) in that particular trial.*

Discussion

We have shown a Python program able to fit a decision making model from experimental data, using the maximum likelihood principle. Thanks to Python and the SciPy stack, it was possible to implement this program in a way that we believe is easy to understand and that has a clear correspondence to the theoretical development of the model. We think that the structure of the code allows to easily extend the implementation to test variations in the decision making model presented in this work.

Acknowledgments

We thanks Liam Mason for sharing the experimental data used in this work. This work was supported by the Advanced Center for Electrical and Electronic Engineering, AC3E, Basal Project FB0008, CONICYT.

References

[Byr95] R.Byrd, P. Lu and J. Nocedal. *A Limited Memory Algorithm for Bound Constrained Optimization*, SIAM Journal on Scientific and Statistical Computing 16 (5): 1190-1208, 1995.

[Cas02] G. Casella and R. L. Berger, Statistical Inference. Thomson Learning, 2002.

[Daw11] N. D. Daw, *Trial-by-trial data analysis using computational models*, Decision making, affect, and learning: Attention and performance XXIII, vol. 23, p. 1, 2011.

[Kah84] D. Kahneman and A. Tversky. *Choices, values, and frames.*, American psychologist 39.4, 1984.

[Lan08] J. Langford, and T. Zhang, *The epoch-greedy algorithm for multi-armed bandits with side information*, Advances in neural information processing systems, 2008.

[Mas12] L. Mason, N. O'Sullivan, R. P. Bentall, and W. El-Deredy, *Better Than I Thought: Positive Evaluation Bias in Hypomania*, PLoS ONE, vol. 7, no. 10, p. e47754, Oct. 2012.

[Res72] R. A. Rescorla and A. R. Wagner, *A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement*, Classical conditioning II: Current research and theory, vol. 2, pp. 64–99, 1972.

[Sut98] R. Sutton and A. Barto, Reinforcement Learning. Cambridge, Massachusetts: The MIT press, 1998.

[Wie12] M. Wiering and M. van Otterlo, Eds., Reinforcement Learning, vol. 12. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[Was16] M. Waskom et al. seaborn: v0.7.0 (January 2016). ; DOI: 10.5281/zenodo.45133. Available at: http://dx.doi.org/10.5281/zenodo.45133.