cpm: a python library for theory-driven modelling in computational psychiatry

Modular + tested workflows for simulation, fitting, and model comparison

Lenard Dome | Developmental Computational Psychiatry | University of Tübingen

Yale | MAPs Talk | Feb 2026

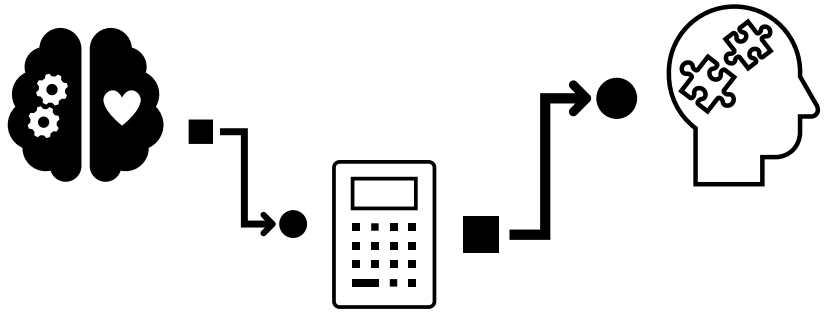
Why do we model?

Models are ways to remove the theory from the brain of its authors.

- Wills, A., 2019, personal communication

Computational psychiatry

- **Mechanistic models** link cognition → symptoms
- **Quantitative inference** from behavioral and clinical data
- **Prediction & intervention** in principled practice



It is not that simple…

Computational psychiatry

- **Fragmentation:** disparate codebases, inconsistent conventions
- **Opacity:** undocumented assumptions, weak documentation, brittle reproducibility
- **Access:** high technical barrier, slow onboarding, limited reuse

We need shared, transparent, tested workflows.

Our solution: **cpm**

An open-source general-purpose Python library for theory-driven computational psychiatry models.

- Standardized workflows: simulate/fit/compare/evaluate
- Modular building blocks: tasks, models, inference, metrics (mix-and-match)
- Designed for both novices and power users.
- Spend time on science, not glue code.

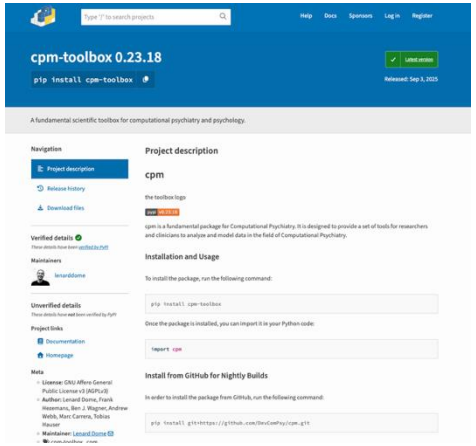
Jul, 2025 – First Release

Computational Psychiatry Modeling

https://cpm-toolbox.net

Sep, 2025 - Preprint

Sep, 2025 – Latest Stable Release



Jan, 2026 – 1.16K Downloads

downloads 1k

`DeltaRule(alpha=None, zeta=None, weights=None, feedback=None, input=None, **kwargs)`

DeltaRule class computes the prediction error for a given input and target value.

Parameters:

- `alpha` (`float` , default: `None`) – The learning rate.
- `zeta` – The constant fraction of the magnitude of the prediction error.
- `weights` (`array - like` , default: `None`) – The value matrix, where rows are outcomes and columns are stimuli or features. The values can be anything; for example belief values, association weights, connection weights, Q-values.
- `feedback` (`array - like` , default: `None`) – The target values or feedback, sometimes referred to as teaching signals. These are the values that the algorithm should learn to predict.
- `input` (`array - like` , default: `None`) – The input value. The stimulus representation in the form of a 1D array, where each element can take a value of 0 and 1.
- `**kwargs` (`dict` , default: `{}`) – Additional keyword arguments.

▾ See Also

cpm.models.learning.SeparableRule : A class representing a learning rule based on the separable error-term of Bush and Mosteller (1951).

▾ Notes

The delta-rule is a summed error term, which means that the error is defined as the difference between the target value and the summed activation of all values for a given output units target value available on the current trial/state. For separable error term, see the Bush and Mosteller (1951) rule.

The current implementation is based on the Gluck and Bower's (1988) delta rule, an extension of the Rescorla and Wagner (1972) learning rule to multi-outcome learning. Such that

$$\Delta w_{ij} = \alpha \cdot (\lambda_i - \sum_j w_{ij}) \cdot x_j$$

where Δw_{ij} is the change in weight for the j-th stimulus for the i-th outcome, λ_i is the target (feedback) value for the i-th outcome, $w_i j$ is the weights of stimulus j for the i-th outcome, x_j is the j-th stimulus input, and α is the learning rate. This is consistent with the Rescorla and Wagner (1972)'s learning rule incorporating the summed error term.

Examples:

```
>>> import numpy as np
>>> from cpm.models.learning import DeltaRule
>>> weights = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])
>>> teacher = np.array([1, 0])
>>> input = np.array([1, 1, 0])
>>> delta_rule = DeltaRule(alpha=0.1, zeta=0.1, weights=weights, feedback=teacher, input=input)
>>> delta_rule.compute()
array([[ 0.07,  0.07,  0.  ],
       [-0.09, -0.09, -0.  ]])
>>> delta_rule.noisy_learning_rule()
```

The model parameters are initialized with the following default values if not specified (values are in the form [initial, lower_bound, upper_bound]):

```
- `alpha`: [1.0, 1e-2, 5.0] (utility curvature for both gains and losses)
- `lambda_loss`: [1.0, 1e-2, 5.0] (loss sensitivity)
- `gamma`: [0.5, 1e-2, 5.0] (curvature for the weighting function for both gains and losses)
- `temperature`: [5.0, 1e-2, 15.0] (temperature parameter for softmax)
```

The priors for the parameters are set as follows:

```
- `alpha`: truncated normal with mean 1.0 and standard deviation 1.0.
- `lambda_loss`: truncated normal with mean 2.5 and standard deviation 1.0.
- `gamma`: truncated normal with mean 2.5 and standard deviation 1.0.
- `temperature`: truncated normal with mean 10.0 and standard deviation 5.0.
```

Model Specification

The model computes the subjective utility of the safe and risky options using a utility function, which can be either a power function or a user-defined utility curve. If a utility curve is not provided, the model uses the following power function with curvature parameter α after Tversky & Kahneman (1992):

$$\mathcal{U}(o) = \sum_{i=1}^{n} w(p_i) \cdot u(x_i)$$

where w is a weighting function of the probability p of a potential outcome, and u is the utility function of the magnitude x of a potential outcome. The choice options is denoted with o. The utility function u is defined as a power function for both gains and losses. It is implemented after Equation 5 in Tversky & Kahneman (1992):

$$u(x) = \begin{cases} x^{\alpha} & \text{if } x \geq 0 \\ -\lambda \cdot (-x)^{\alpha} & \text{if } x < 0 \end{cases}$$

where α is the utility curvature parameter for both gains and losses, and λ is the loss aversion parameter. The weighting function is implemented after Equation 6 in Tversky & Kahneman (1992):

$$w(p) = \frac{p^{\gamma}}{(p^{\gamma} + (1-p)^{\gamma})^{1/\gamma}}$$

where `gamma` , denoted via γ, is the discriminability parameter of the weighting function for both gains and losses. The model then applies the softmax function to compute the choice probabilities:

$$p(o_i) = \frac{e^{\beta \cdot \mathcal{U}(o_i)}}{\sum_{j=1}^{n} e^{\beta \mathcal{U}(o_j)}}$$

Model output

EXAMPLES

Two associative learning models and blocking (cpm.models, cpm.generators)

Reinforcement learning with a two-armed bandit (cpm.models, cpm.generators, cpm.optimisation)

Estimating metacognitive efficiency with meta-d (cpm.applications)

Parameter Recovery (cpm.generators, cpm.models, cpm.optimisation)

Loss (Negative Log Likelihood) and data-generating function

Specify a generative model

Revert the model to the original version and refit the data

Compare the estimated parameters with the original ones

Model Recovery and Landscaping (cpm.models, cpm.generators, cpm.optimisation)

Estimating Empirical Priors (cpm.hierarchical)

Exploring hierarchical estimation of hyperparameters (cpm.hierarchical)

Scale to hypercomputing cluster

Parameter Recovery (cpm.generators, cpm.optimisation)

Evaluating how well the model recovers the original parameters. Briefly, you generate new data, fit the model to the data, and then compare the estimat instead of using the participant's choice to select what value to update, you each option and use that to sample a choice. The technique we will use belo model: the *cpm.minimisation.LogLikelihood.bernoulli* and the *Bernoulli distribu* do here?

2. Estimate the parameter recovery of the model
    1. Generate data from the model
    2. Fit the model to the data
    3. Compare the estimated parameters to the true parameters

Loss (Negative Log Likelihood) and data-gener

Here, we define the loss function based on negative log likelihood.

$$-\log L(\theta \mid Y, M) = -\sum_{i=1}^{N} \log \left[p \right.$$

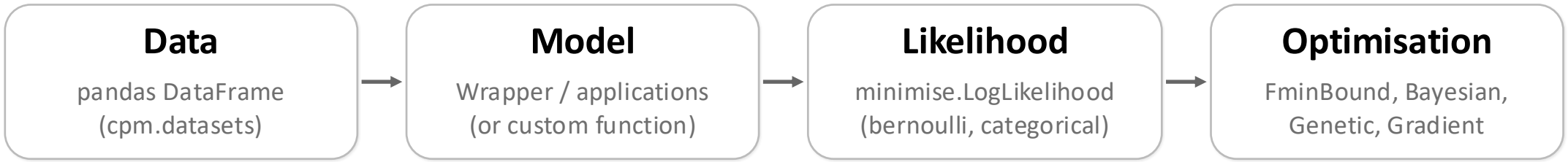
where θ are model parameters, Y is the data with N number of data points

What I’m not covering today (so we can stay sane)

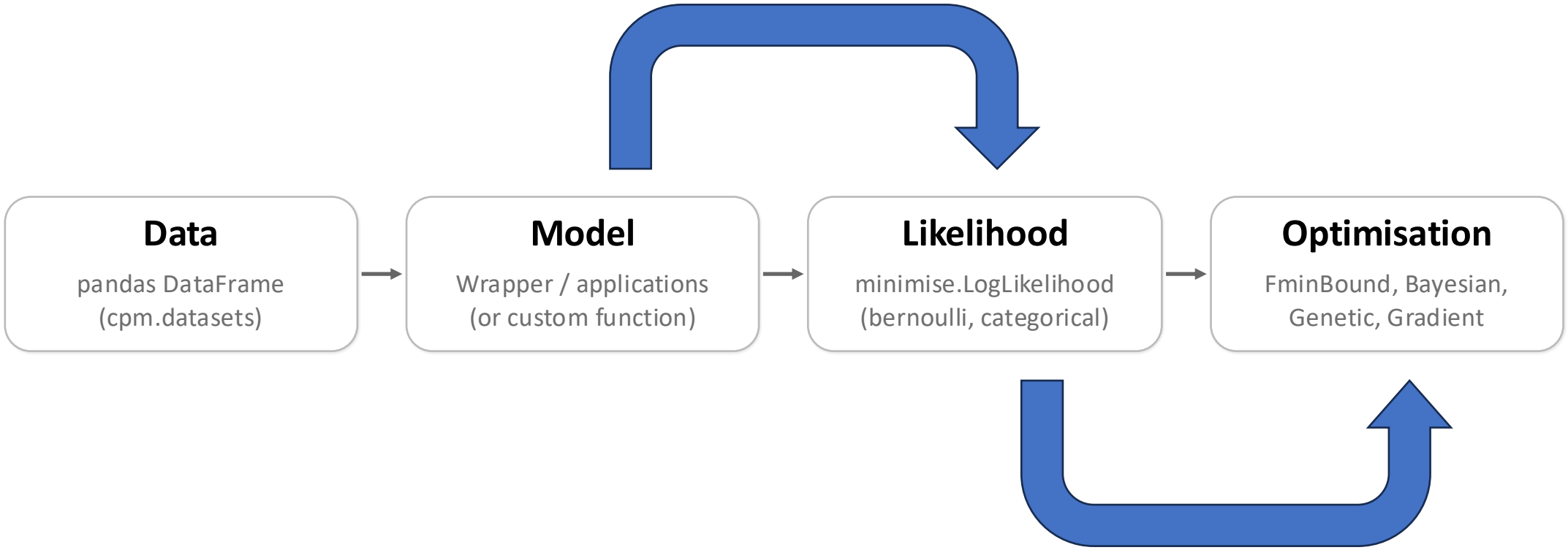
- The math behind the models
- Fitting mechanics (optimizer settings, convergence diagnostics)
- Software internals (architecture, code organization)
- Engineering plumbing (packaging, CI, performance tuning)
- Clinical claims (this is a research tool, not a diagnostic tool)

Focus: what it enables, how it works at a high level, and what you can do with it.

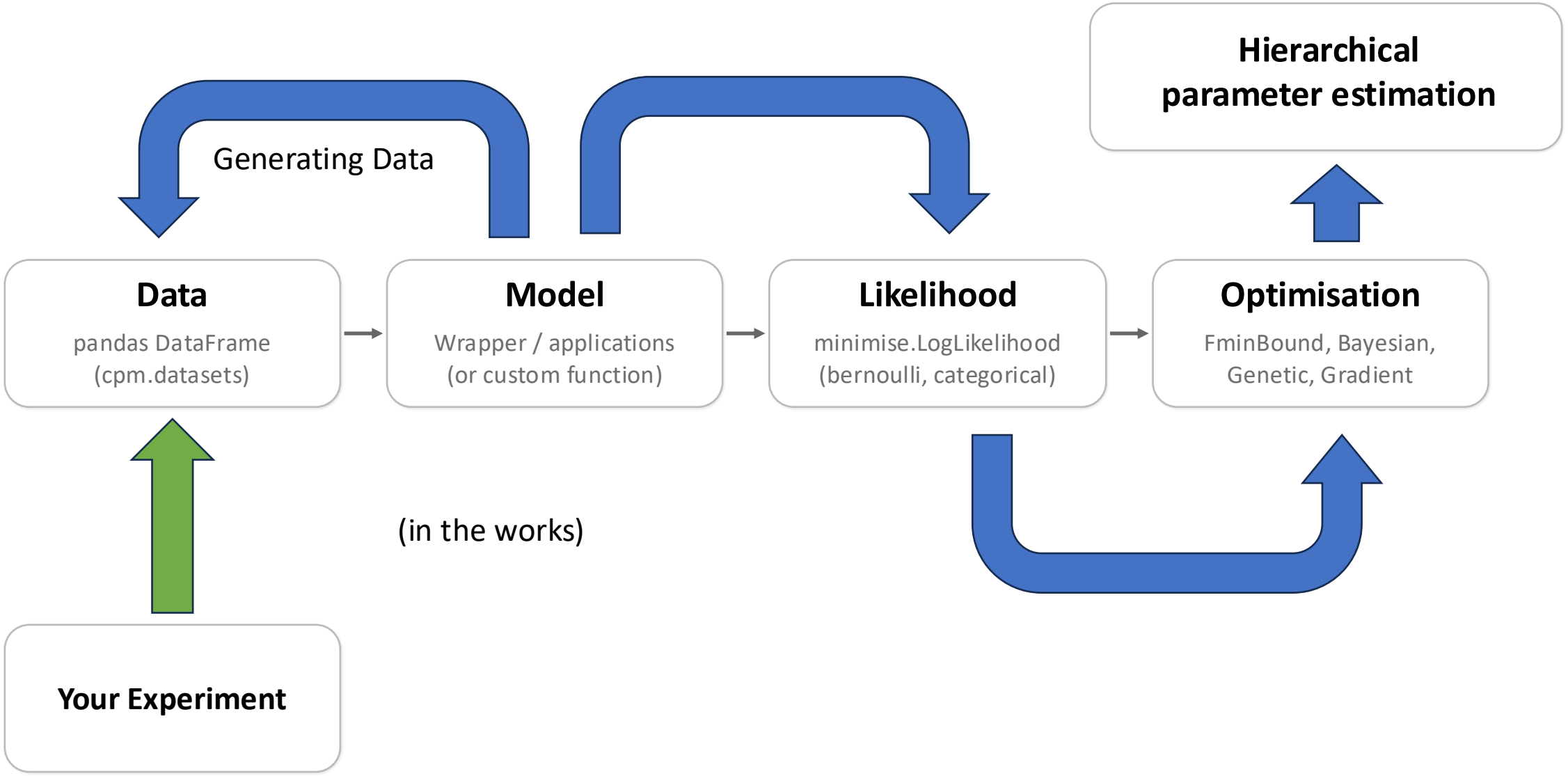
The flow



A minimal modelling loop, with many "on-ramps".



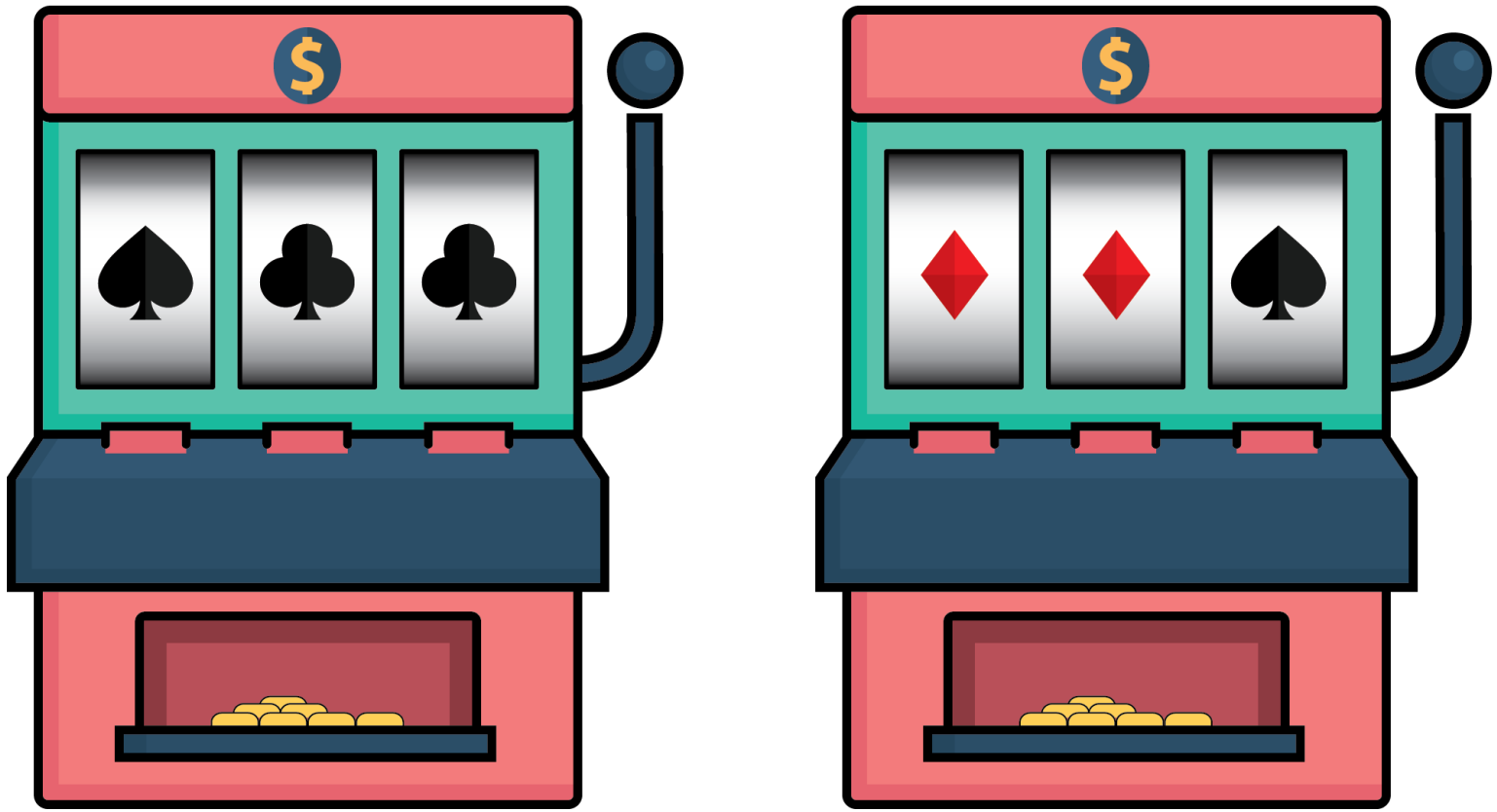
A minimal modelling loop, with many "on-ramps".



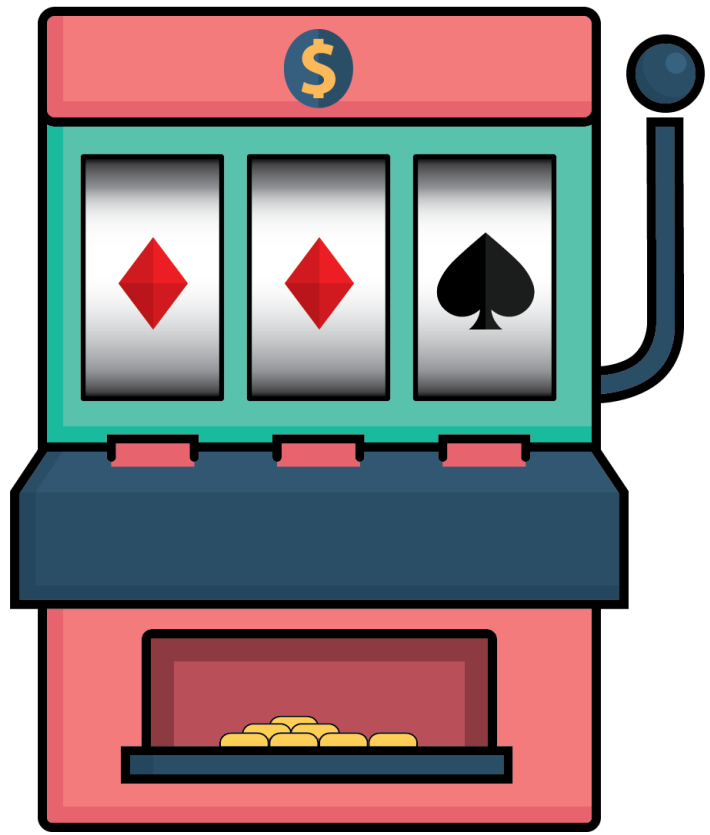
A minimal modelling loop, with many "on-ramps".

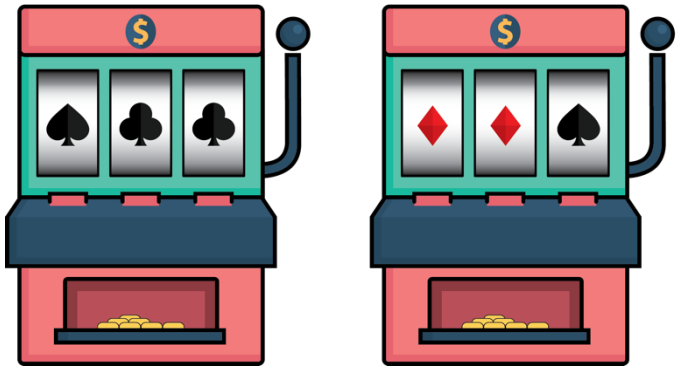
Walkthrough (two-armed bandit example)

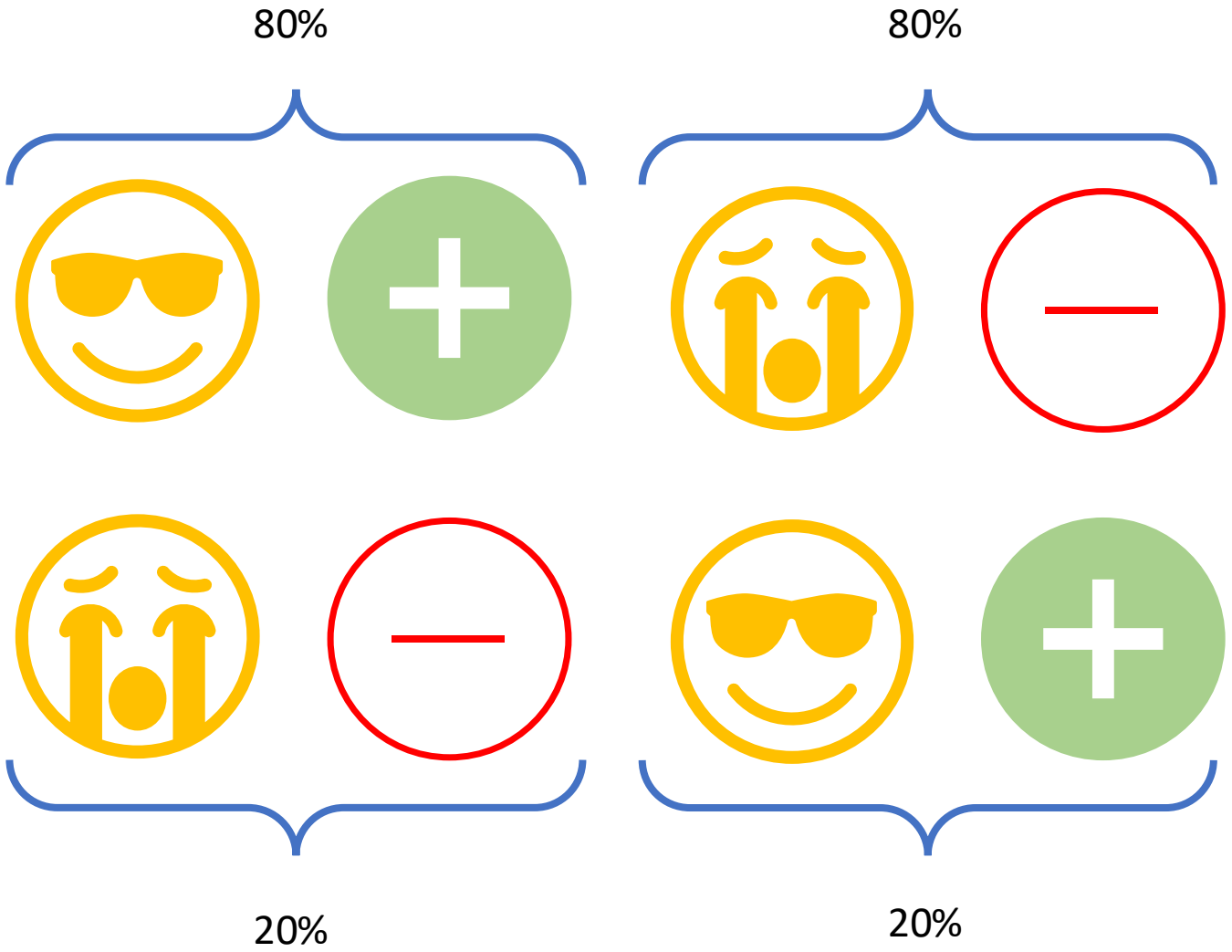
- Experiment investigating differences in learning between groups
- The comparison involves between Neurotypical and Neuroatypical (ADHD) groups
- Experiment involves some distractors throughout the learning
- The goal is to estimate some latent variables captured by model parameters (e.g. learning rates)



+100






80%
80%
20%
20%

DATA

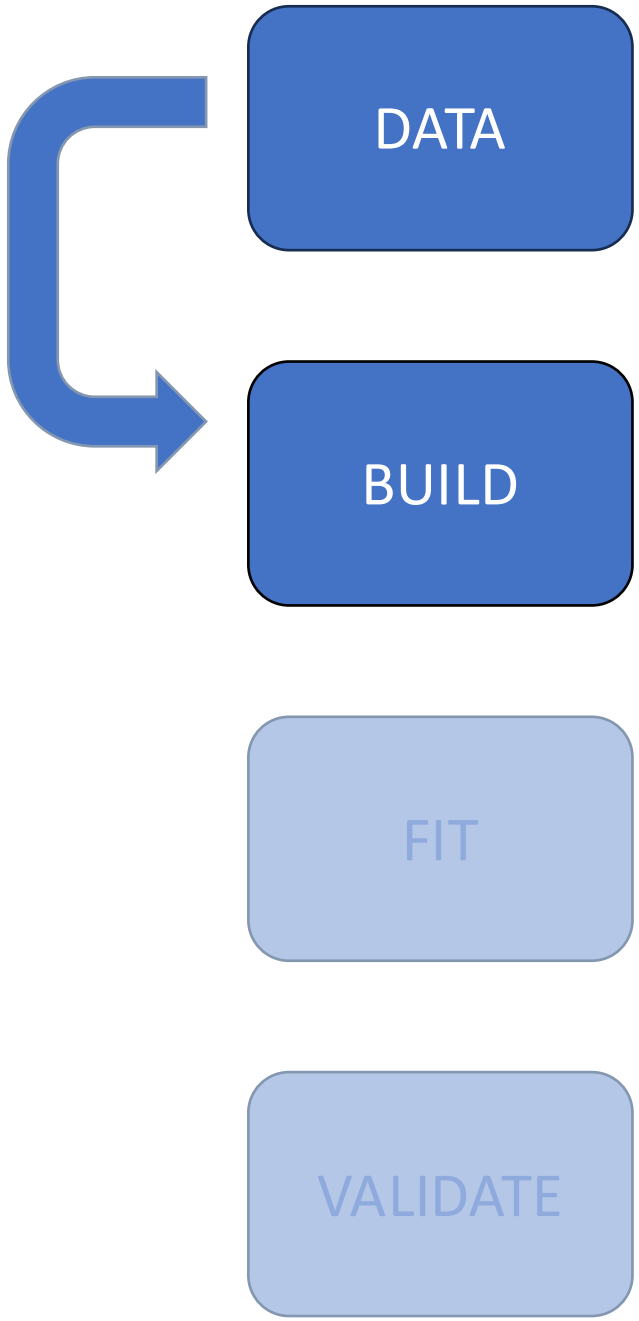
BUILD

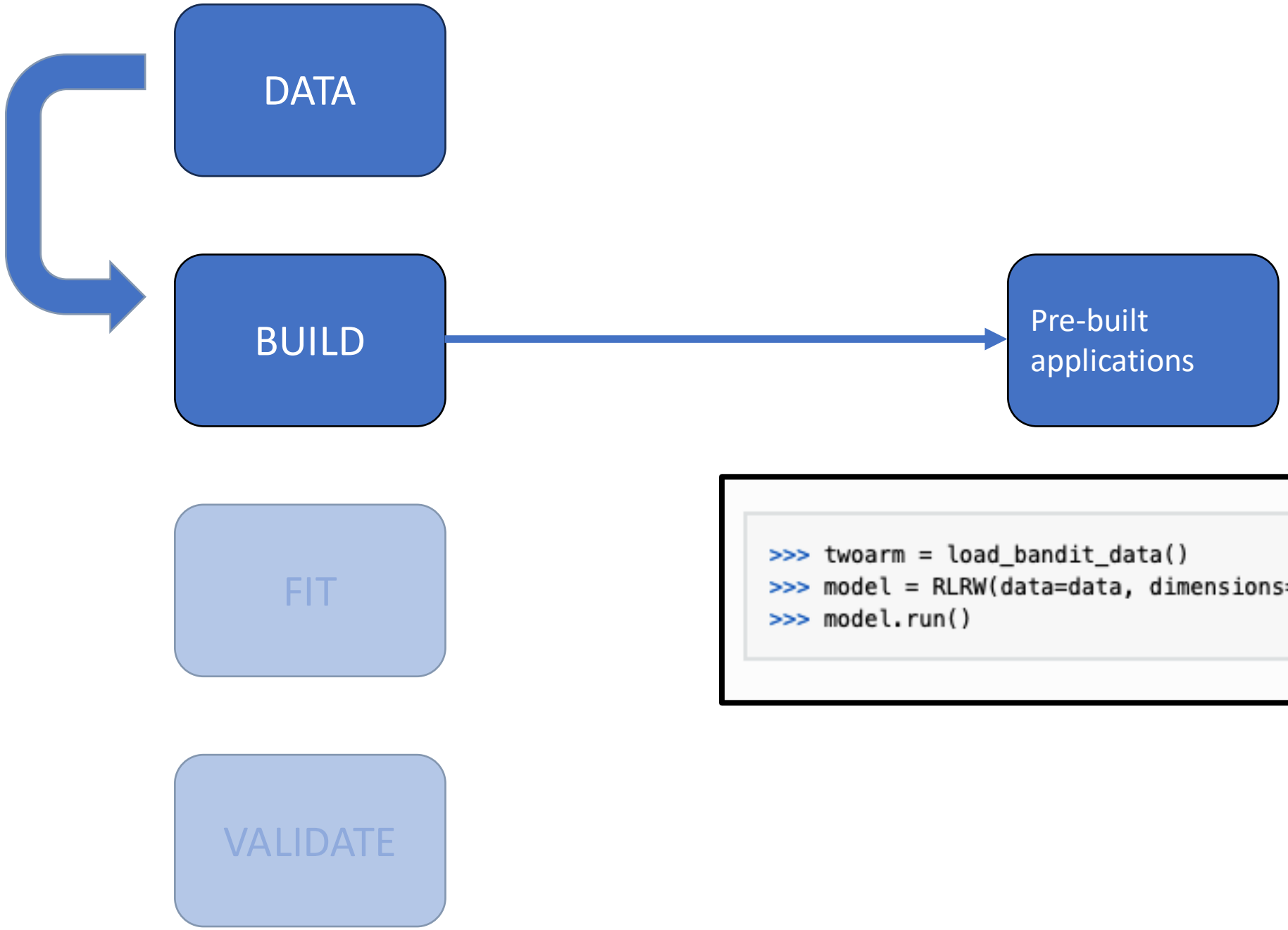
FIT

VALIDATE

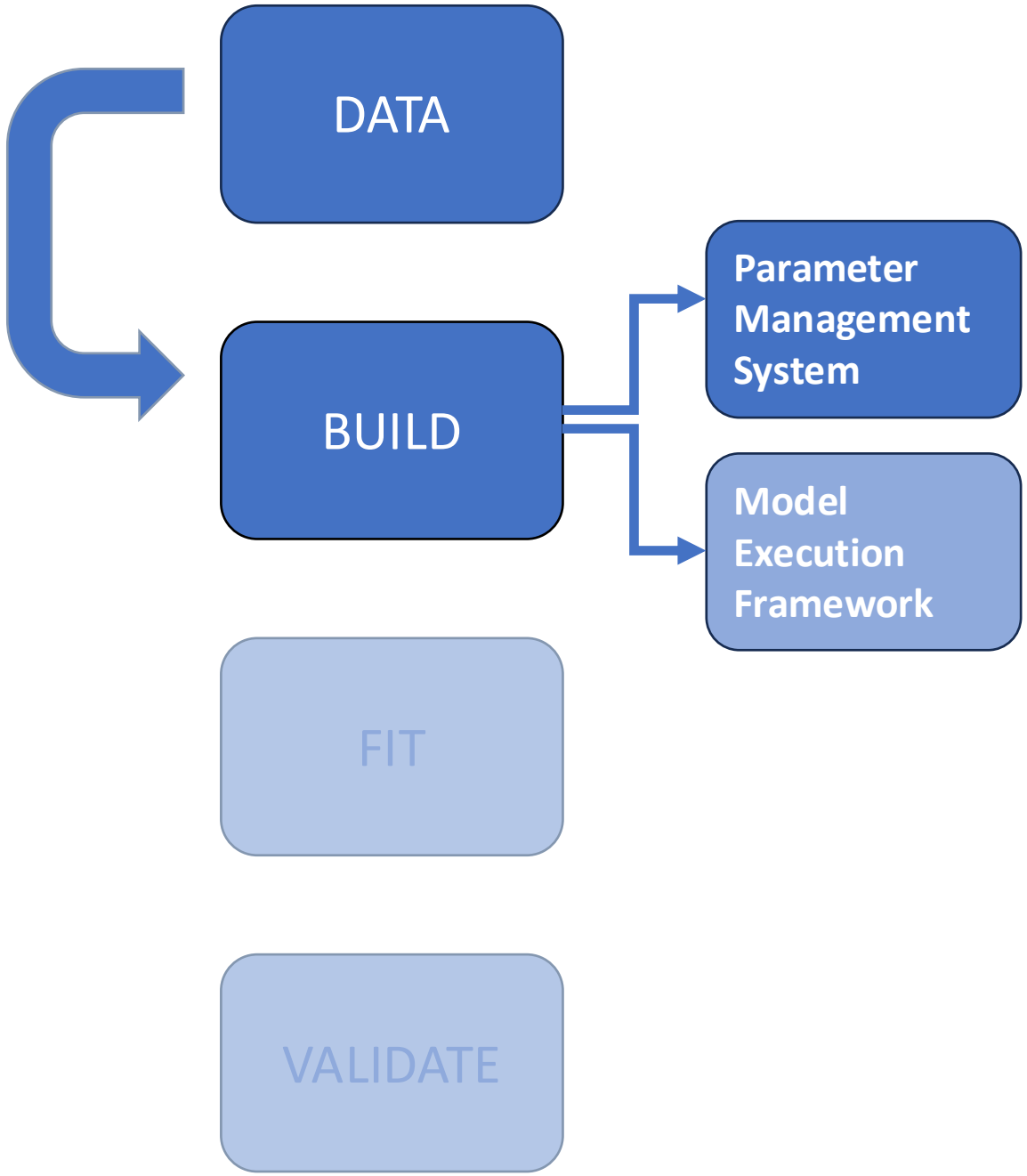
```
## toolbox and libraries
import cpm
import ipyparallel as ipp ## for parallel computing with ipython (specific for Jupyter Notebook)
import pandas as pd
import numpy

import matplotlib.cm as cmx
import matplotlib.pyplot as plt
from prettyformatter import pprint

experiment = cpm.datasets.load_bandit_data()
experiment.head(10)
```

	ppt	trial	arm_left	arm_right	reward_left	reward_right	response	feedback
0	1	1	2	1	0.0	1.0	1	1.0
1	1	2	3	2	1.0	0.0	0	1.0
2	1	3	1	3	1.0	0.0	0	1.0
3	1	4	1	3	0.0	0.0	0	0.0
4	1	5	2	4	1.0	0.0	0	1.0
5	1	6	3	1	1.0	1.0	0	1.0
6	1	7	4	2	1.0	1.0	1	1.0
7	1	8	1	4	0.0	0.0	0	0.0
8	1	9	3	4	0.0	0.0	0	0.0
9	1	10	1	4	1.0	0.0	0	1.0

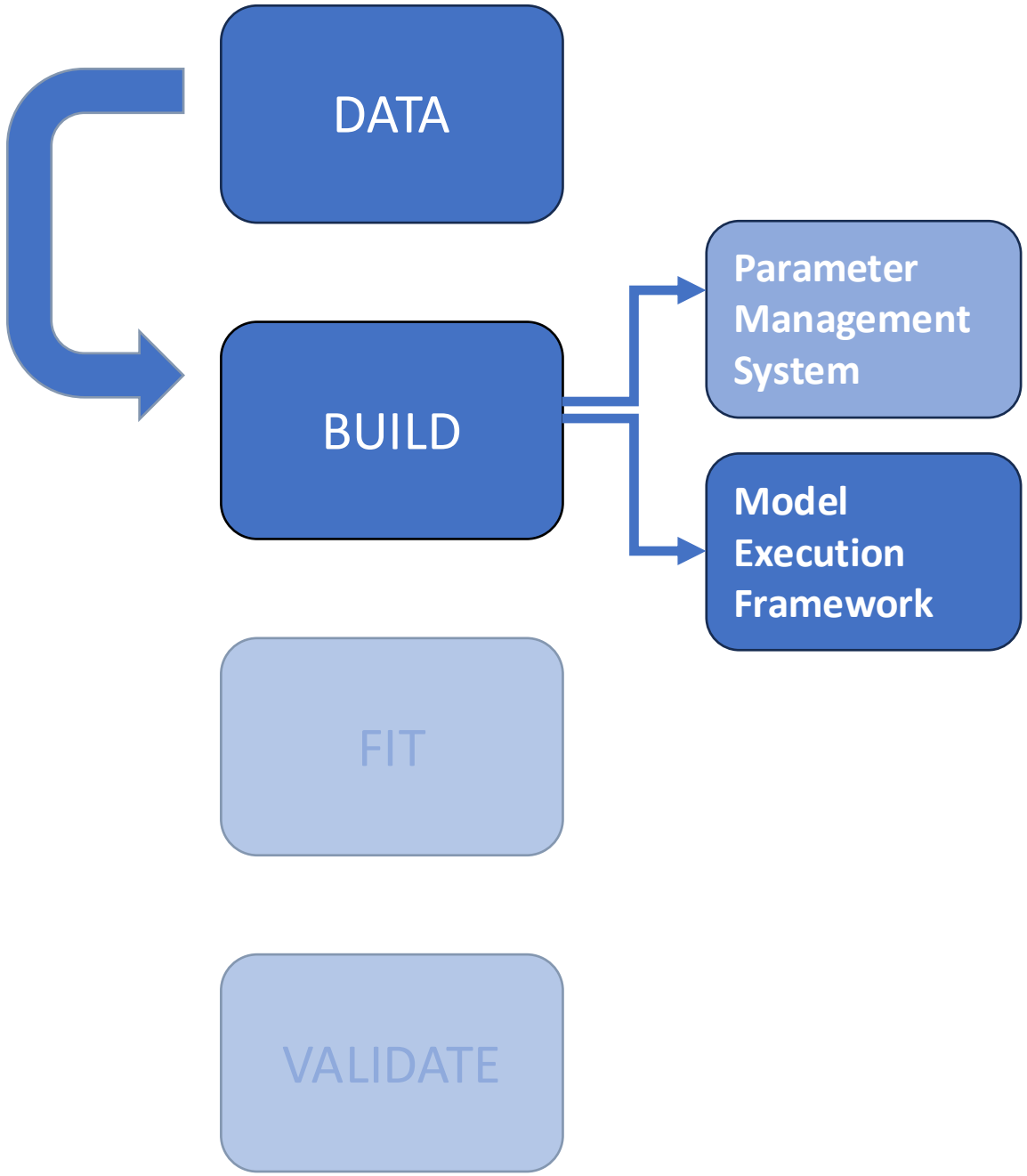

DATA
BUILD
FIT
VALIDATE



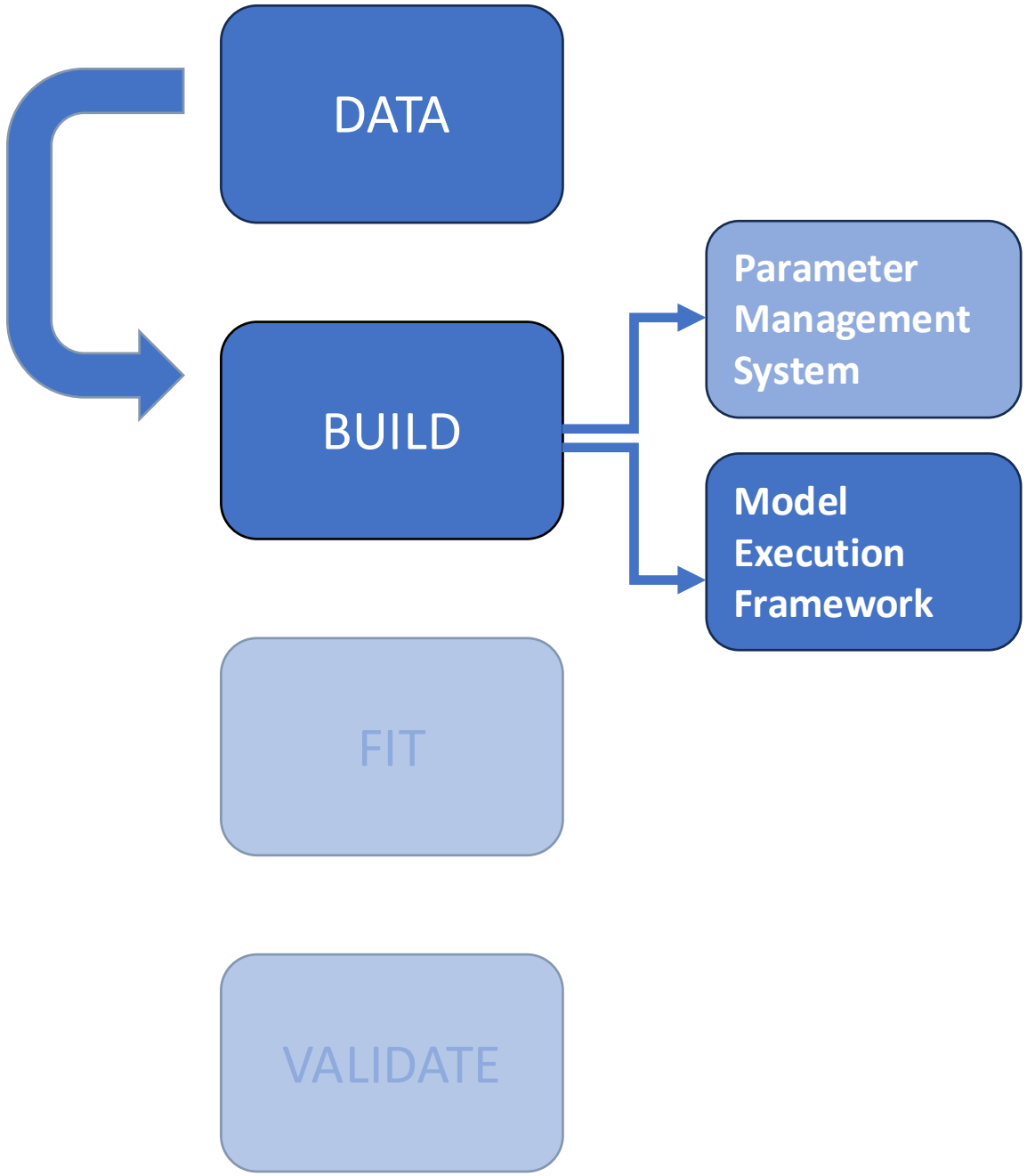

```
>>> twoarm = load_bandit_data()
>>> model = RLRW(data=data, dimensions=4)
>>> model.run()
```




```python
from cpm.generators import Parameters, Value

parameters = Parameters(
    # freely varying parameters are indicated by specifying priors
    alpha=Value(
        value=0.5,
        lower=1e-10,
        upper=1,
        prior="truncated_normal",
        args={"mean": 0.5, "sd": 0.25},
    ),
    temperature=Value(
        value=1,
        lower=0,
        upper=20,
        prior="truncated_normal",
        args={"mean": 10, "sd": 5},
    ),
    # everything without a prior is part of the initial state of the
    values = numpy.array([0.0, 0.0, 0.0, 0.0]))
```



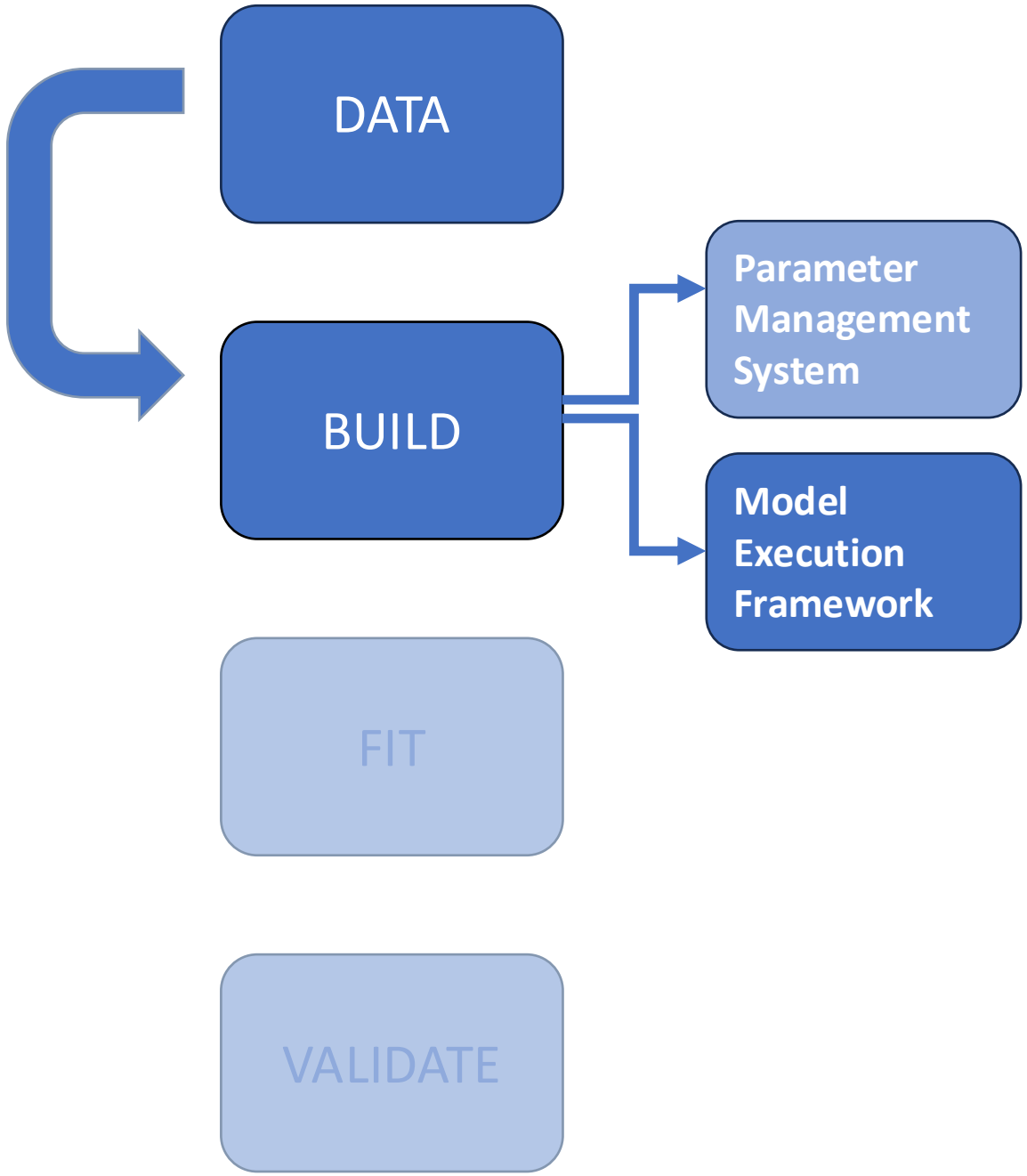

```python
import cpm
import ipyparallel as ipp  ## for parallel computing with ipython (specific fo

@ipp.require("numpy")
def model(parameters, trial):
    # pull out the parameters
    alpha = parameters.alpha
    temperature = parameters.temperature
    values = numpy.asarray(parameters.values)

    # pull out the trial information
    stimulus = numpy.array([trial.arm_left, trial.arm_right]).astype(int)
    feedback = numpy.array([trial.reward_left, trial.reward_right])
    human_choice = trial.response.astype(int)
```




```python
import cpm
import ipyparallel as ipp  ## for parallel computing with ipython (specific fo

@ipp.require("numpy")
def model(parameters, trial):
    # pull out the parameters
    alpha = parameters.alpha
    temperature = parameters.temperature
```

```python
    # Equation 2.
    # calculate a policy based on the activations
    choice_rule = cpm.models.decision.Softmax(
        activations=expected_rewards,
        temperature=temperature
        )
    choice_rule.compute() # compute the policy
```

```python
    # Equation 4.
    update = cpm.models.learning.SeparableRule(
                    weights=values,
                    feedback=teacher,
                    input=what_to_update,
                    alpha=alpha
                    )
    update.compute()
    # Equation 5.
    values += update.weights.flatten()
```




```
# compile output
output = {
    "trial"    : trial.trial.astype(int), # trial num
    "activation" : expected_rewards.flatten(), # expe
    "policy"   : choice_rule.policies,       # polici
    "response" : model_choice,               # model's c
    "reward"   : reward,                     # received
    "error"    : update.weights,             # predictio
    "values"   : values,                     # updated v
    # dependent variable
    "dependent"  : numpy.array([choice_rule.policies
}
return output
```

DATA

BUILD

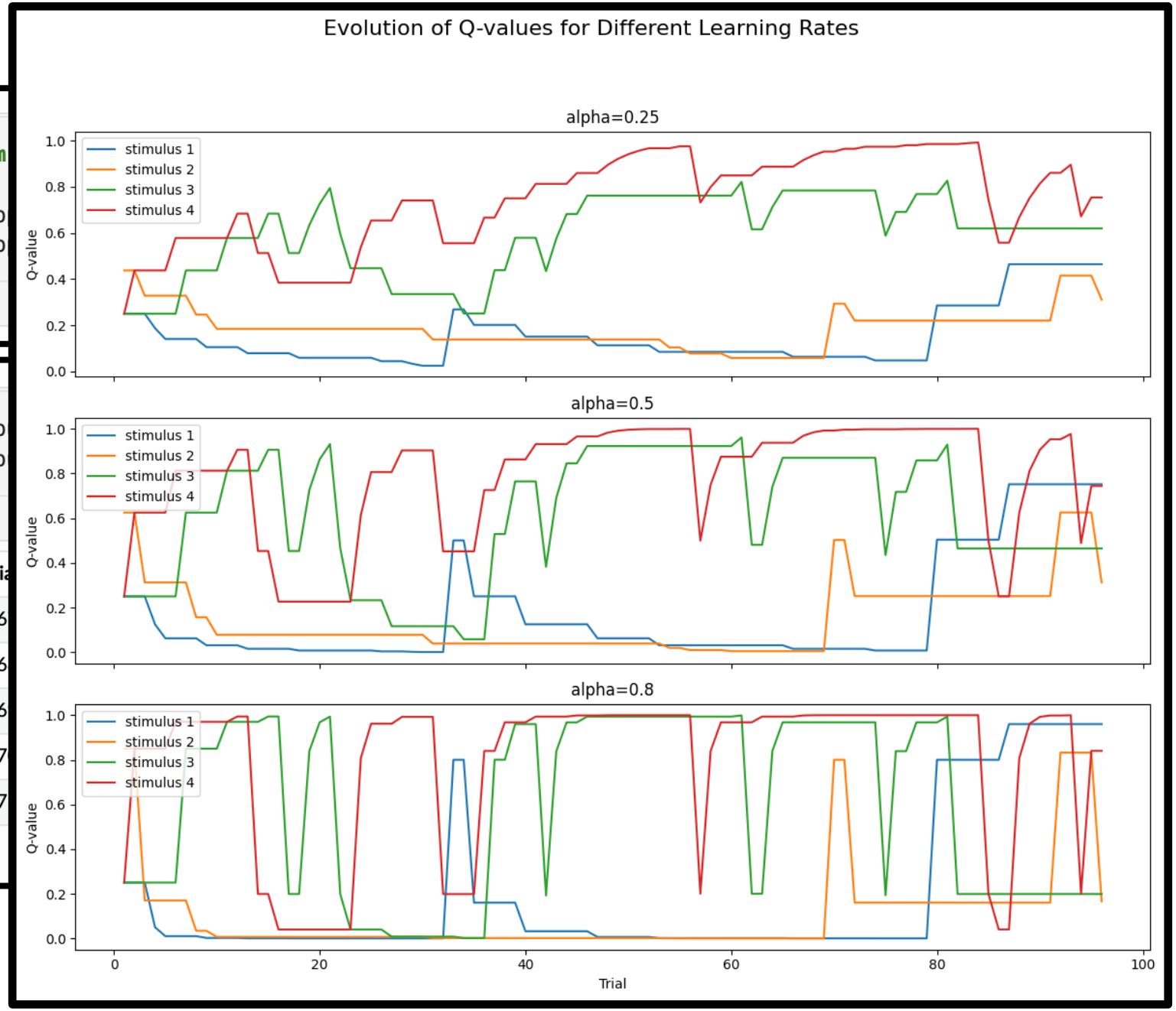
FIT

VALIDATE

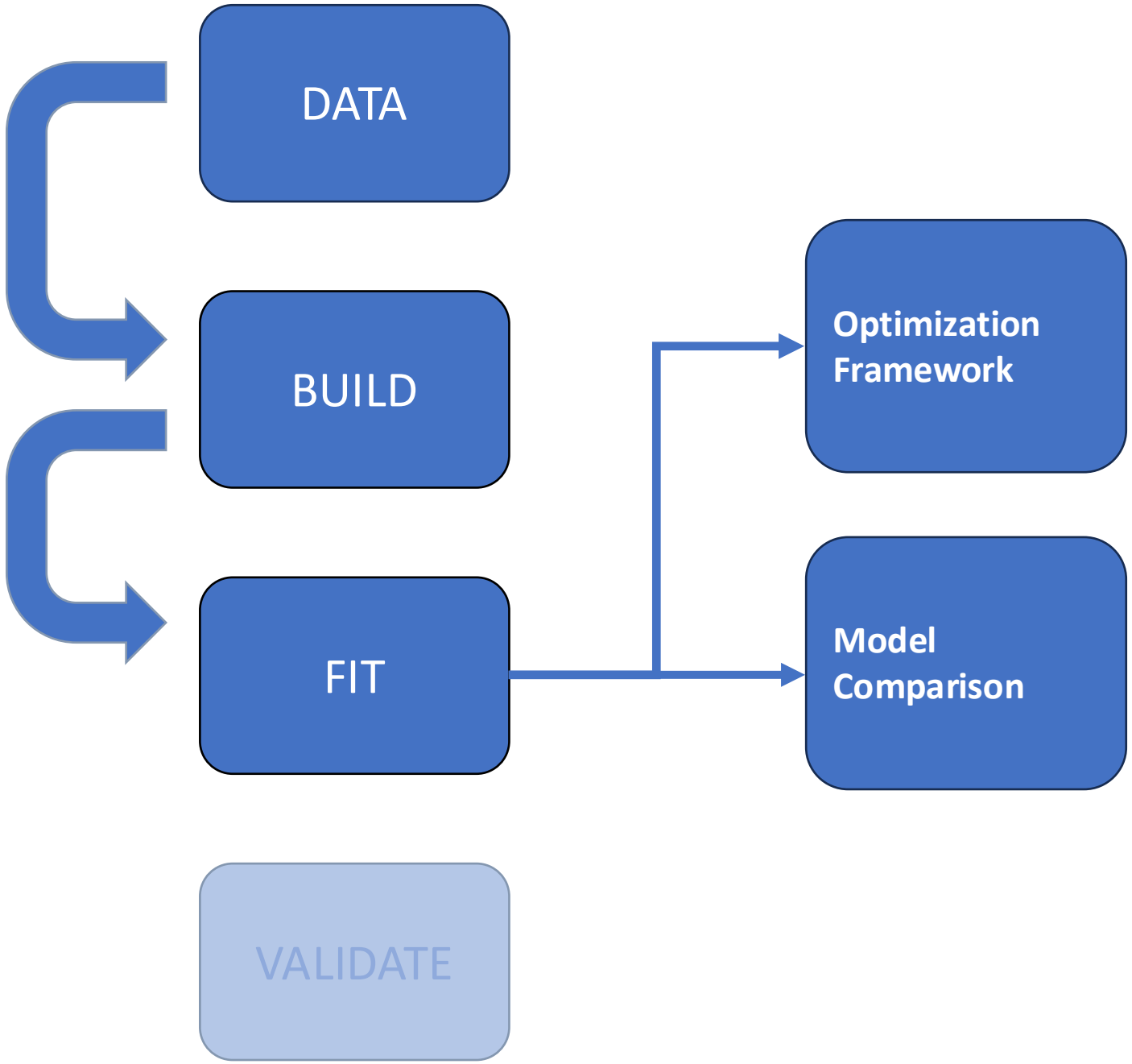
```
from

wrap
wrap
```

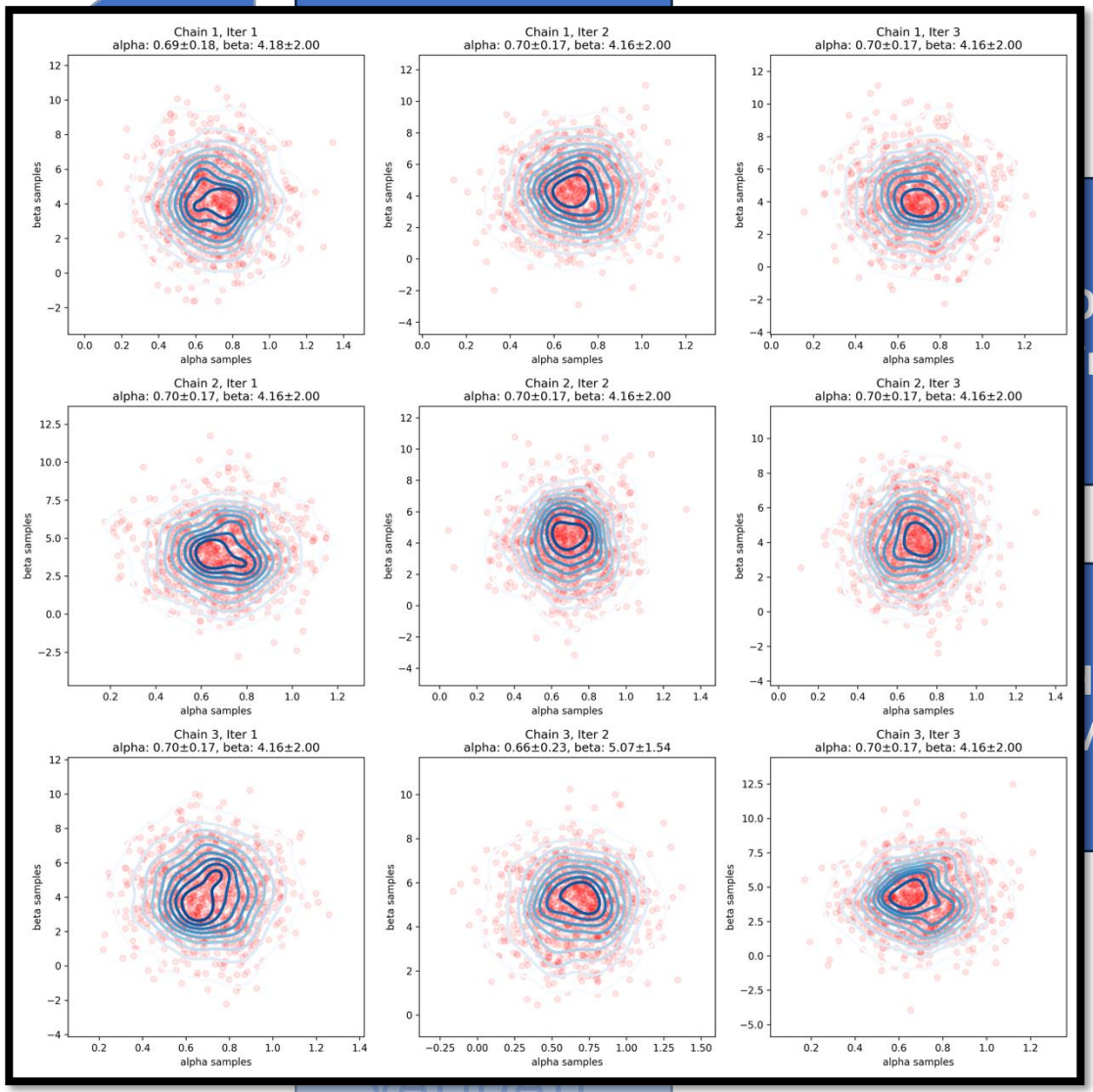
```
outp
outp
```

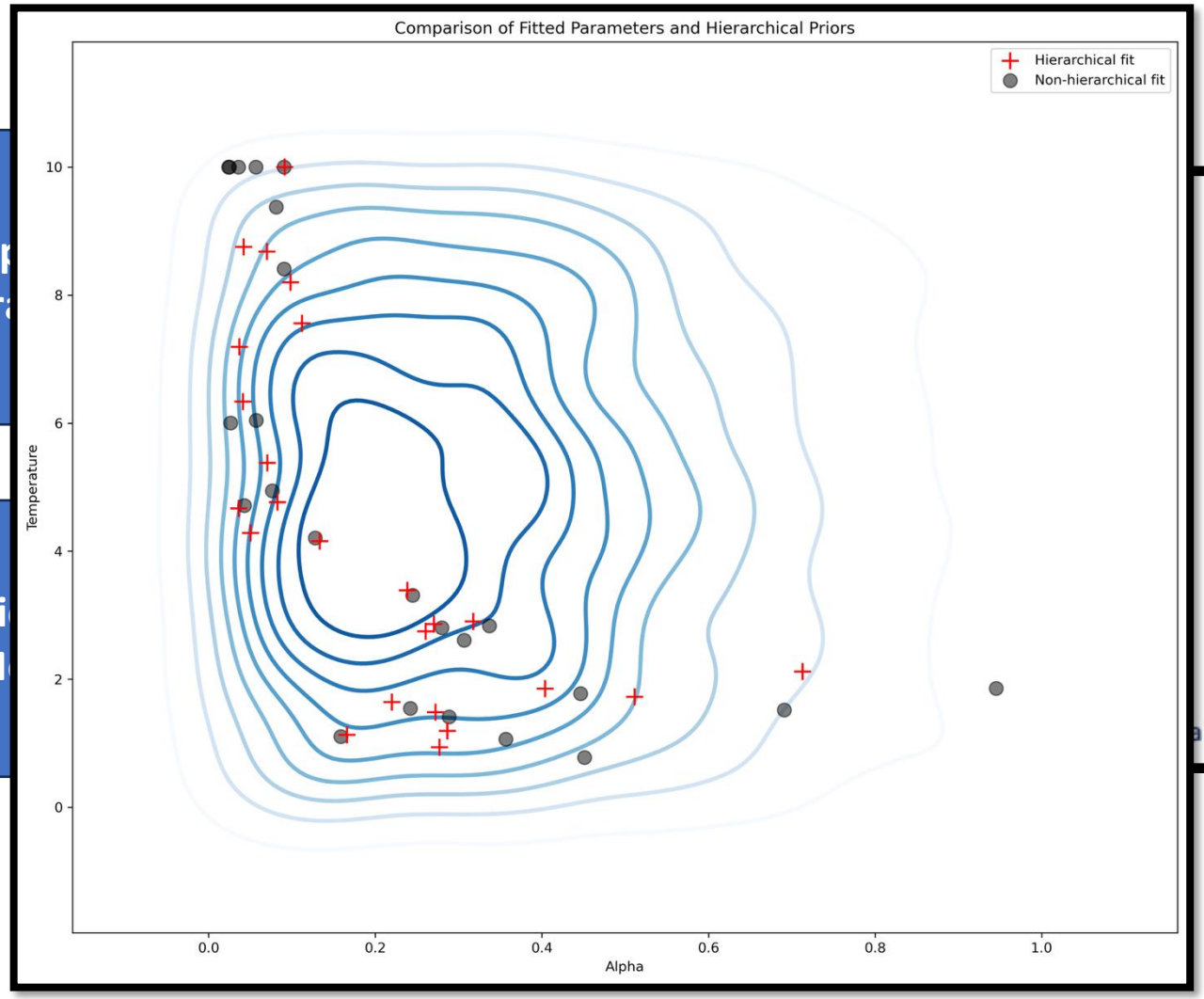
	tria
66	6
67	6
68	6
69	7
70	7



	x_0	x_1	grad_0	grad_1	task	funcalls	nit	warnflag	hessian_0	hessian_1	hessian_2	hes
0	0.560707	14.653588	7.105427e-07	1.776357e-07	CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL	45	12	0	161.658910	-1.030274	-1.030274	0.0
1	0.014001	9.980408	-7.105427e-04	-3.552713e-06	CONVERGENCE: RELATIVE REDUCTION OF F <= FACTR*...	42	10	0	20851.926301	37.455916	37.455916	0.0
2	0.477092	1.966134	7.887024e-05	6.394885e-06	CONVERGENCE: RELATIVE REDUCTION OF F <= FACTR*...	30	7	0	57.077150	5.443754	5.443754	2.4
3	0.796371	10.322221	3.552714e-06	7.105427e-07	CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL	72	18	0	197.869939	-2.948346	-2.948346	0.1
4	0.437350	4.920923	7.105427e-07	7.105427e-07	CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL	36	8	0	117.979596	2.236371	2.236371	0.7
...	...	...	...	...	...	...	...	...	...	...	...	
195	0.353022	4.384126	-7.105427e-07	0.000000e+00	CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL	63	14	0	97.938458	5.468136	5.468136	1.0
196	0.327815	10.236654	-7.815970e-06	0.000000e+00	CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL	45	12	0	346.098032	3.042619	3.042619	0.2


DATA
BUILD
FIT
VALIDATE
Optimization Framework
Model Comparison


Chain 1, Iter 1
alpha: 0.69±0.18, beta: 4.18±2.00
Chain 1, Iter 2
alpha: 0.70±0.17, beta: 4.16±2.00
Chain 1, Iter 3
alpha: 0.70±0.17, beta: 4.16±2.00
Chain 2, Iter 1
alpha: 0.70±0.17, beta: 4.16±2.00
Chain 2, Iter 2
alpha: 0.70±0.17, beta: 4.16±2.00
Chain 2, Iter 3
alpha: 0.70±0.17, beta: 4.16±2.00
Chain 3, Iter 1
alpha: 0.70±0.17, beta: 4.16±2.00
Chain 3, Iter 2
alpha: 0.66±0.23, beta: 5.07±1.54
Chain 3, Iter 3
alpha: 0.70±0.17, beta: 4.16±2.00
beta samples
alpha samples


Comparison of Fitted Parameters and Hierarchical Priors
Hierarchical fit
Non-hierarchical fit
Temperature
Alpha

DATA

BUILD

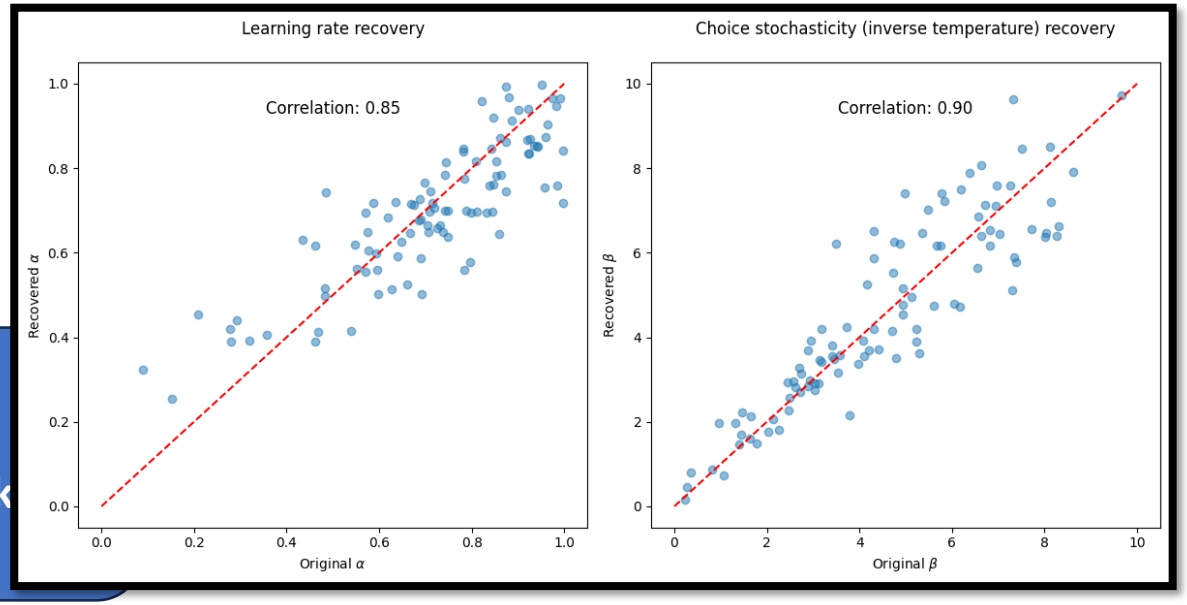
FIT

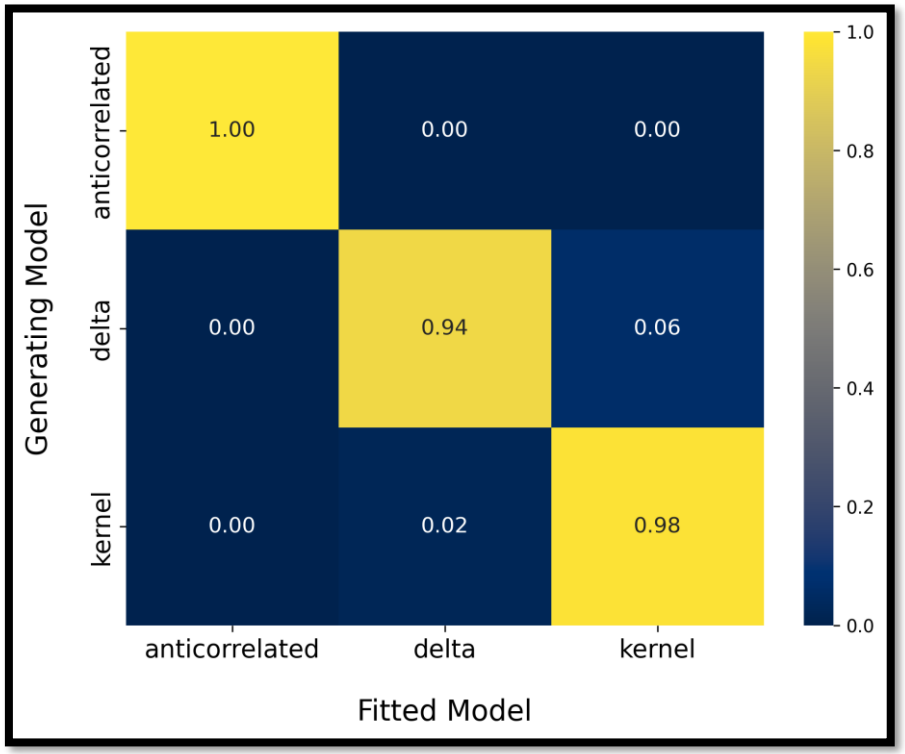
VALIDATE

Evaluation Framework

Data Generation Framework

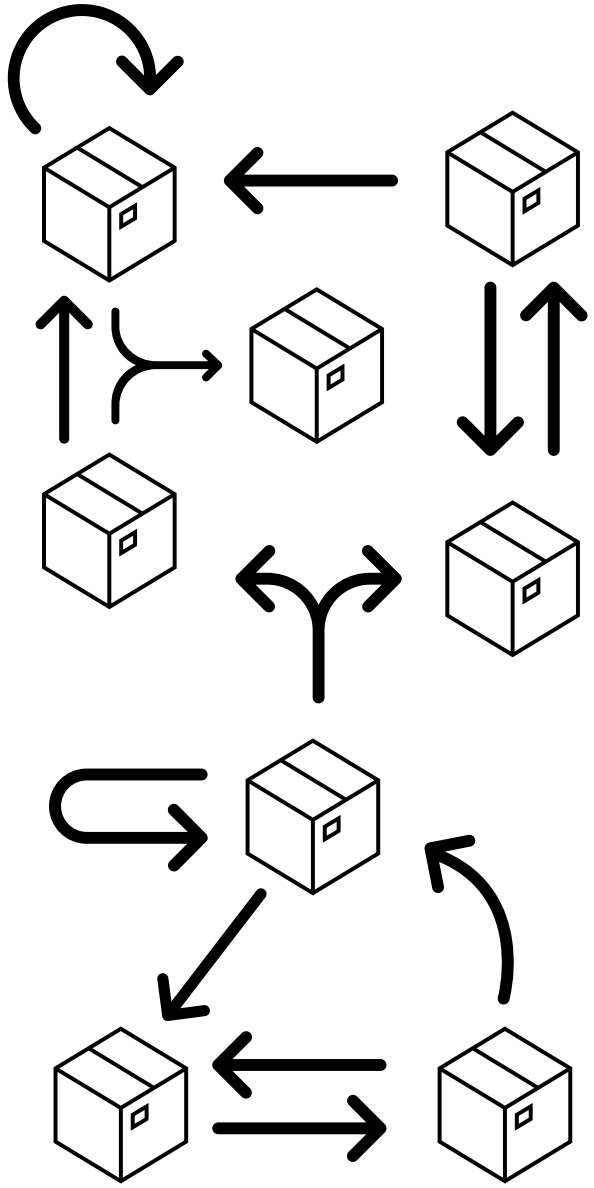
Simulation Framework





Accessibility and Modularity

- **Start anywhere:** multiple entry-points
- **Few lines to run** a canonical model and export trial-level predictions
- **Built-in** datasets + tutorials lower the barrier to “doing it right”



Reproducible science needs *reproducible code* (tests catch silent breakage).

What we test (130+):

- deterministic utilities (parsing, transforms, likelihoods)
- **parameter constraints & edge cases** (bounds, NaNs, empty inputs)
- **invariants** (e.g., probabilities sum to 1, shapes match, monotonic properties)

Reproducible science needs *reproducible code* (tests catch silent breakage).

What we test:

- deterministic utilities (parsing, transforms, likelihoods)
- **parameter constraints & edge cases** (bounds, NaNs, empty inputs)
- **invariants** (e.g., probabilities sum to 1, shapes match, monotonic properties)

- **Continuous integration:** tests run automatically across OS + Python versions.
- **Payoff**: easier refactoring, safer extensions, more reliable runs

How to sum up the toolbox?

Open-source by default

Codebase on GitHub

Documented, inspectable components

Each model component is explicit and tied to a theoretical reference.

Clean outputs for reproducible analysis

Exports tidy pandas DataFrames for fitting, simulation, and diagnostics.

Plays nicely with other tooling

Export objective functions to third-party optimisers when needed.

Planned directions (near-to-mid term)

- Broaden model categories + add more preconfigured applications
- Richer model comparison (Bayes factors, exceedance probabilities, MDL)
- More efficient hierarchical fitting (incl. MCMC / VBMC approaches)
- More simulation-based model evaluation tools (recovery, landscaping, etc.)
- Integration with data collection platforms – Brain Explorer

Get involved: pip install cpm • GitHub: DevComPsy/cpm • Contributions welcome

Thank you for your kind attention!

🙏 The DCP LAB:

Meng Gao
Susan Fischer
Vanessa Teckentrup
Jasper Bischofberger
Amzar Muzani Maarof
Taiki Oka
Floor Burghoorn
Ben J. Wagner
Maciej Szul
Aleya Marzuki
George Tertikas
Kenza Kadri
Alisa Loosen
Tricia Seow
Xin Sui
Claudius Gruner
Larisa Dinu

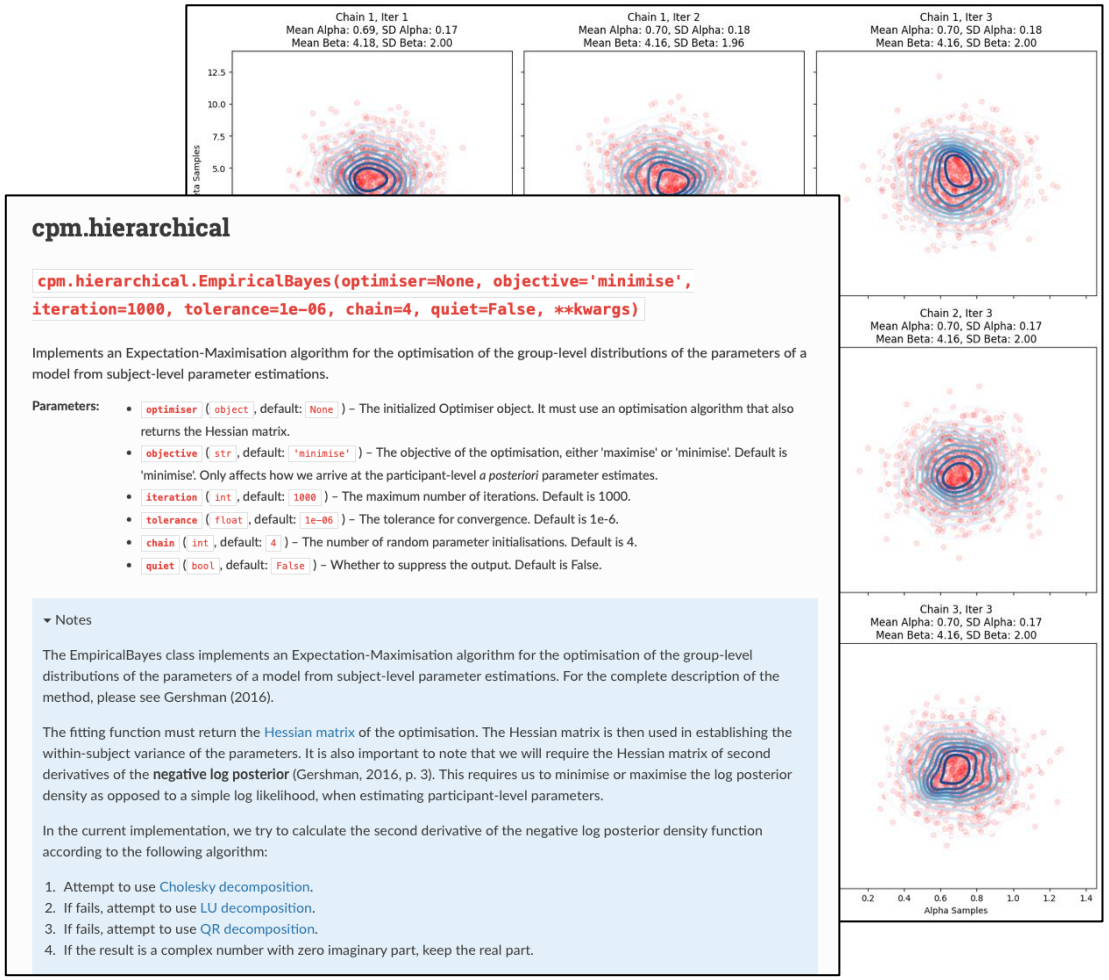
https://lenarddome.com/

https://osf.io/preprints/psyarxiv/6jc3t_v1c

https://devcompsy.org/

Alumni: Layla Alsalih, Frank Hezemans, Ehsan Kakaei

Appendix A: Hierarchical fitting: model building



- **cpm.hierarchical.EmpiricalBayes** (Gershman, 2016)
  an Expectation-Maximisation algorithm for the optimisation of the group-level distributions of the parameters
- **cpm.hierarchical.VariationalBayes** (Piray, 2019)
  a reduced version of the Hierarchical Bayesian Inference (HBI) algorithm proposed by Piray et al. (2019)

Appendix B: Optimisation Procedures

- **cpm.optimisation** (interface to parameter estimation techniques from scipy and other libraries; optimised for subject-level estimates)
  - Fmin
  - FminBound
  - BADS (Bayesian Adaptiove Direct Search)
  - DifferentialEvolultion
  - Minimse
- **cpm.optimisation.minimise** (a battery of discrepency functions)
  - e.g. LogLikelihoods
  - e.g. Non-parametric functions

Appendix C: The scope

Covers common tasks, models, and fitting strategies in one framework:

- Models:
  - Reinforcement learning
  - Decision-making under risk
  - Signal Detection / Metacognition (meta-d')
- Tasks:
  - Trial-level behavioural datasets in pandas-compatible formats
  - Supporting analysis with flexible column naming for stimuli, rewards, and responses
- Methods:
  - Hierarchical Estimation Frameworks
  - Modular learning/decision/activation components
  - Reusable Wrappers for simulation and fitting
  - Optimizer Integration
  - Diagnostic tools for parameter inference
- Extensibility:
  - Infrastructure for adding new models, datasets, loss functions
  - Consistent APIs and documentations