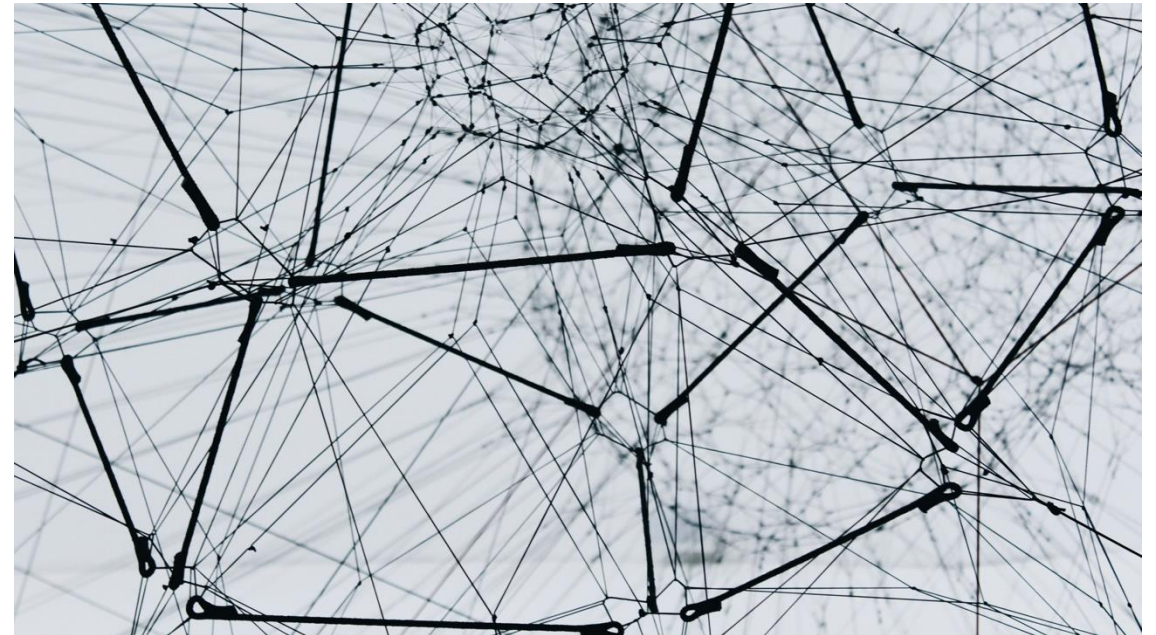
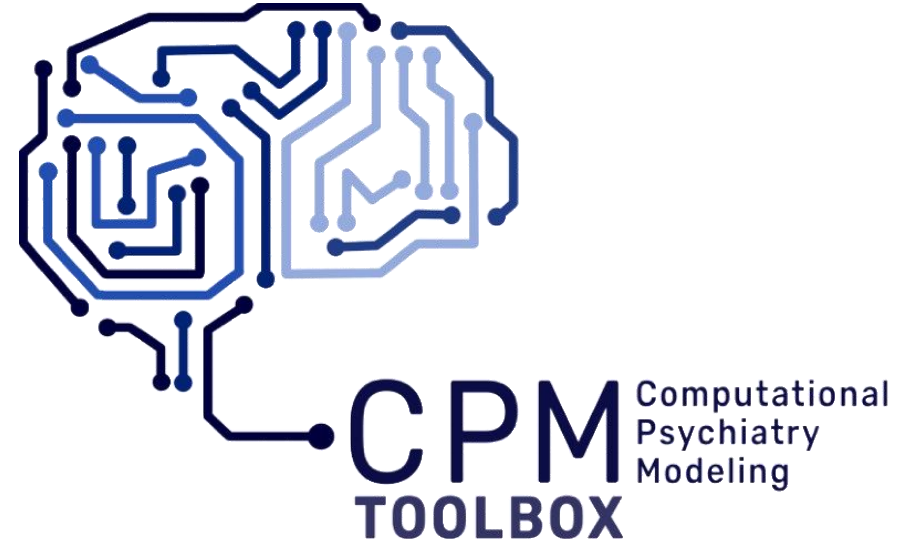


WORKSHOP: theory-driven computational modeling in computational psychiatry with cpm

Lenard Dome



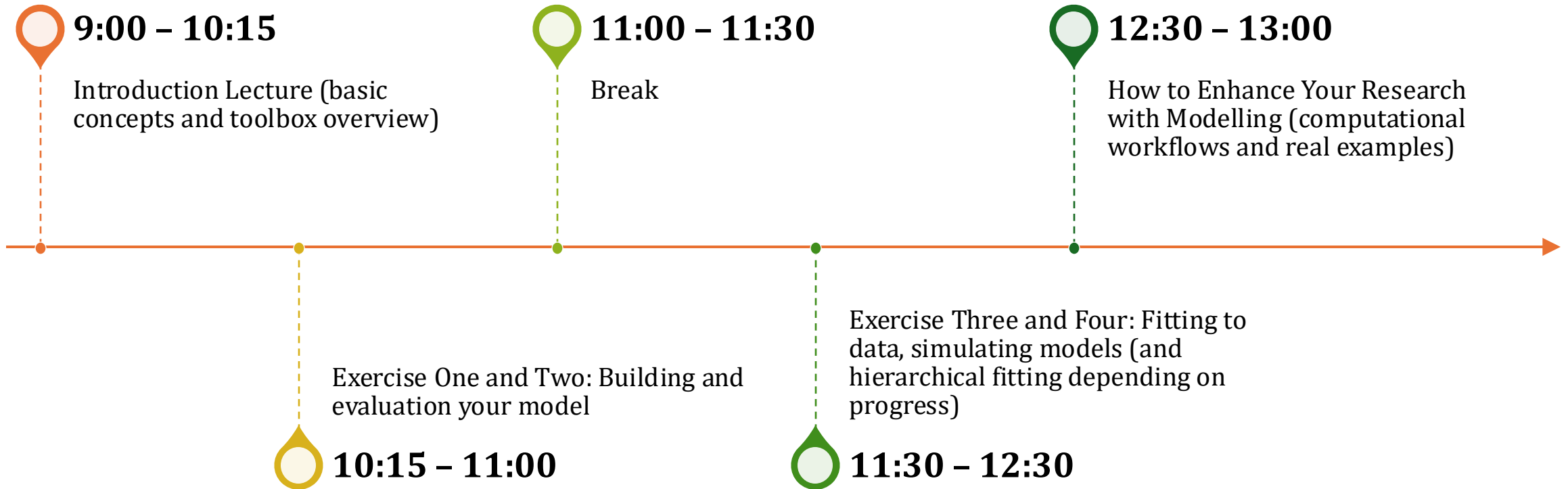
Wellcome!

Meet the team!

Workshop outcomes

- Build models
- Evaluate your models prior to data
- Being able to fit the model to data
- Understand the principles of hierarchical modeling
- Evaluate model and parameter identifiability
- Draw inferences on the group level through estimation of hyperparameters

House Keeping



Introduction

A crash-course on the basic concepts...

Which pizza has more pizza?

- 18 inch?
- 2 x 12 inch?

Pizza problem (Guest & Martin, 2021)

Amount of food per order:

$$\phi_i = N_i \pi r_i^2,$$

where i is the pizza-order option,
 N is the number of pizzas. Here is
our pairwise decision rule:

$$\omega(\phi_i, \phi_j) = \begin{cases} i, & \text{if } \phi_i > \phi_j \\ j, & \text{otherwise} \end{cases}$$



Fermat's Library
@fermatlibrary



Here's a useful counterintuitive fact:
one 18 inch pizza has more 'pizza'
than two 12 inch pizzas

Pizza problem (Guest & Martin, 2021)

Amount of food per order:

$$\phi_i = N_i \pi r_i^2,$$

where i is the pizza-order option, N is the number of pizzas. Here is our pairwise decision rule:

$$\omega(\phi_i, \phi_j) = \begin{cases} i, & \text{if } \phi_i > \phi_j \\ j, & \text{otherwise} \end{cases}$$

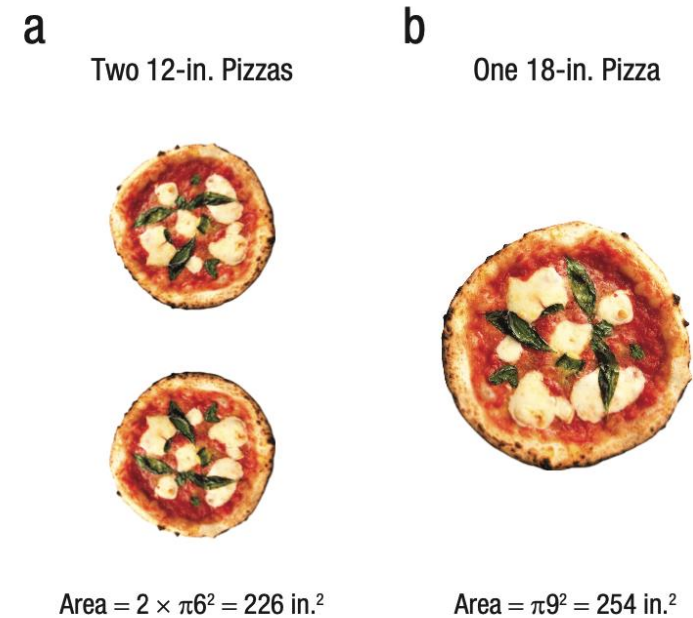


Fig. 1. The pizza problem. Something like comparing the two options presented here can appear counterintuitive, although we all learn the formula for the area of a circle in primary school. Compare (a) two 12-in. pizzas and (b) one 18-in. pizza (all three pizzas are to scale). Which order would you prefer?

What is a model?

Some Uses of Computational Model

Formal expression of theories (Wills & Pothos, 2012; Guest and Martin, 2021):

- Exemplar- and prototype-theories of categorisation (Nosofsky, 1986; Kruschke, 1992; Love, Medin, and Gureckis, 2004)
- Attentional salience of a cue is learned and adjusted on gradient descent of error (Mackintosh, 1975; Kruschke, 2001; Paskewitz and Jones, 2020)

Measuring (usually through data compression) some form of psychological constructs (Willson and Collins, 2019):

- meta-d to measure metacognitive sensitivity (Maniscalco & Lau, 2012)
- deviations from reference model via normative modeling (Rutherford et al., 2022)

Statistical modelling (Breiman, 2001):

- decision-bound strategy analysis (Edmunds, Milton, and Wills, 2018)
- hierarchical mixture models to identify peak shifts in generalization (Lee et al., 2024)

An example: Rescorla-Wagner (1972) and blocking

- What was before?
- What was the phenomenon?
- What was the model?

For its associative history, see Le Pelley (2004) Q. J. Exp. Psychol. B.

For a review on how it became so influential, see Soto et al. (2023) Neurobiol. Learn. Mem.

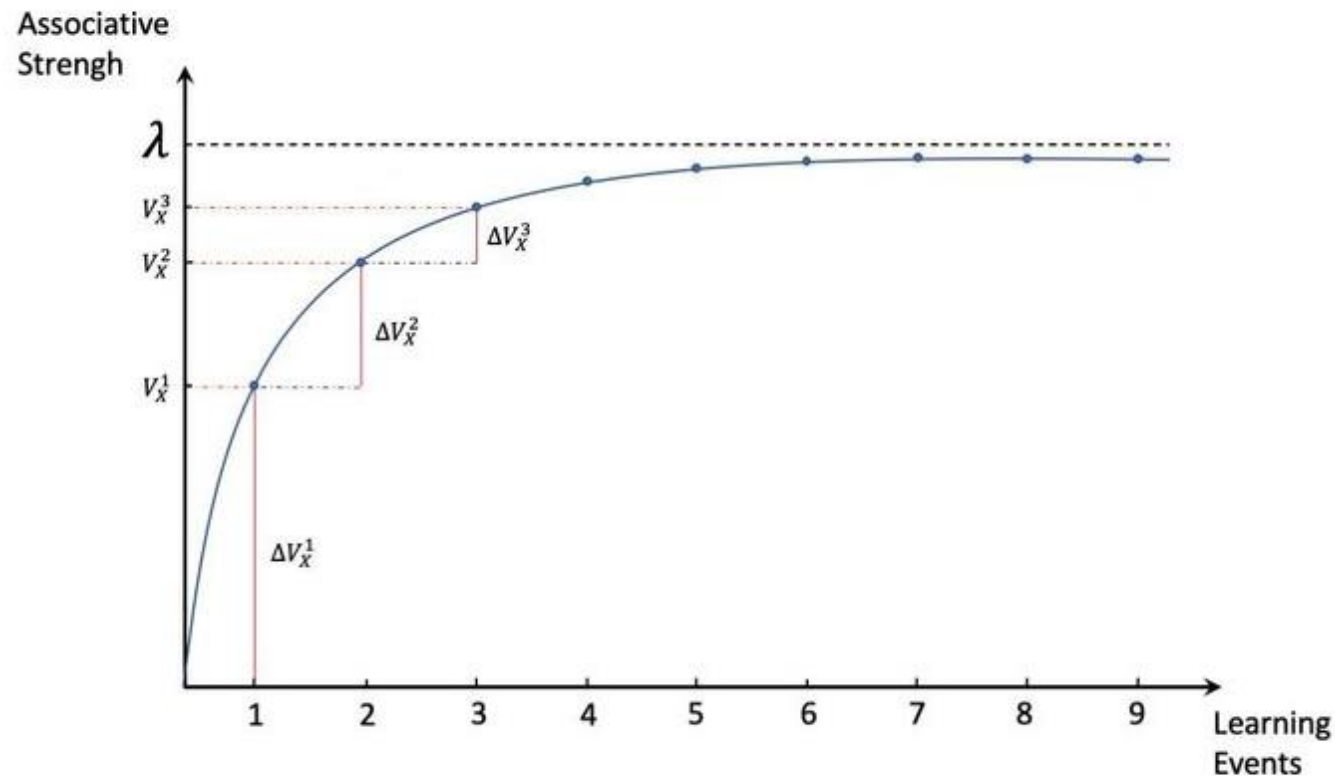


Learning as a decelerating function (Bush and Mosteller, 1951).

$$\Delta V_X = \alpha \times (\lambda - V_X)$$

In case of two stimuli:

$$\begin{aligned}\Delta V_A &= \alpha \times (\lambda - V_A) \\ \Delta V_B &= \alpha \times (\lambda - V_B)\end{aligned}$$



Blocking (Kamin, 1969)

...if a single cue is followed by an outcome (A+), and a separately encountered compound containing that cue is followed by the same outcome (AB+), then learning about B is restricted.

A simple experimental design:

Phase 1: Light -> Food (Salivation)

Phase 2: Light + Tone -> Food (Salivation)

Test Phase:

Light -> No Food (?)

Tone -> No Food (?)

A new model of blocking (Rescorla & Wagner, 1972)

Rescorla-Wagner introduced the summed error term (on trials of AB):

$$PE = \lambda - (V_A + V_B)$$

$$\Delta V_A = \alpha \beta_A \times PE$$

$$\Delta V_B = \alpha \beta_B \times PE$$



The delta rule (Rescorla & Wagner, 1972; McClelland & Rumelhart, 1985)

$$\Delta w_{i,k} = \alpha \left(\lambda - \sum_n w_{k,n} \right) s_k$$

- Discovered by several groups independently (see Sutton and Barto, 1981, Psyc. Rev.)
- This general form is expressed by Gluck and Bower (1989) J. Exp. Psychol. G.



Discrepancy functions: *comparing model output to experimental observations*

- What is the problem here?
- We have a set of values recorded in the experiment, the data: Y
- We have a set of values produced by the model, the predictions: M

So the question is... how close is M to Y ?



Discrepancy functions: non- parametric functions

- Sum of Squared Errors (SSE)
- Root Mean Squared Deviations (RMSD)
- χ^2 (chi-square)
- G^2 (log-likelihood ratio)

- $SSE = \sum_{j=1}^J (y_j - m_j)^2$
- $RMSD = \sqrt{\frac{\sum_{j=1}^J (y_j - m_j)^2}{J}}$
- $\chi^2 = \sum_{j=1}^J \frac{(y_j - Nm_j)^2}{Nm_j}$
- $G^2 = 2 \sum_{j=1}^J \log \left(y_j \log \frac{y_j}{Nm_j} \right)$

Discrepancy functions: **likelihoods**

- (most) Models will assign a probability, p_j , for each outcome, y_j
- p_j is the mean of a probability distribution
- p and y can be related through the function, f

These measures are conditional on the (M) model and its current (θ) parameter set:

$$f(y \mid \theta, M)$$



Two ways of the likelihood/probability functions

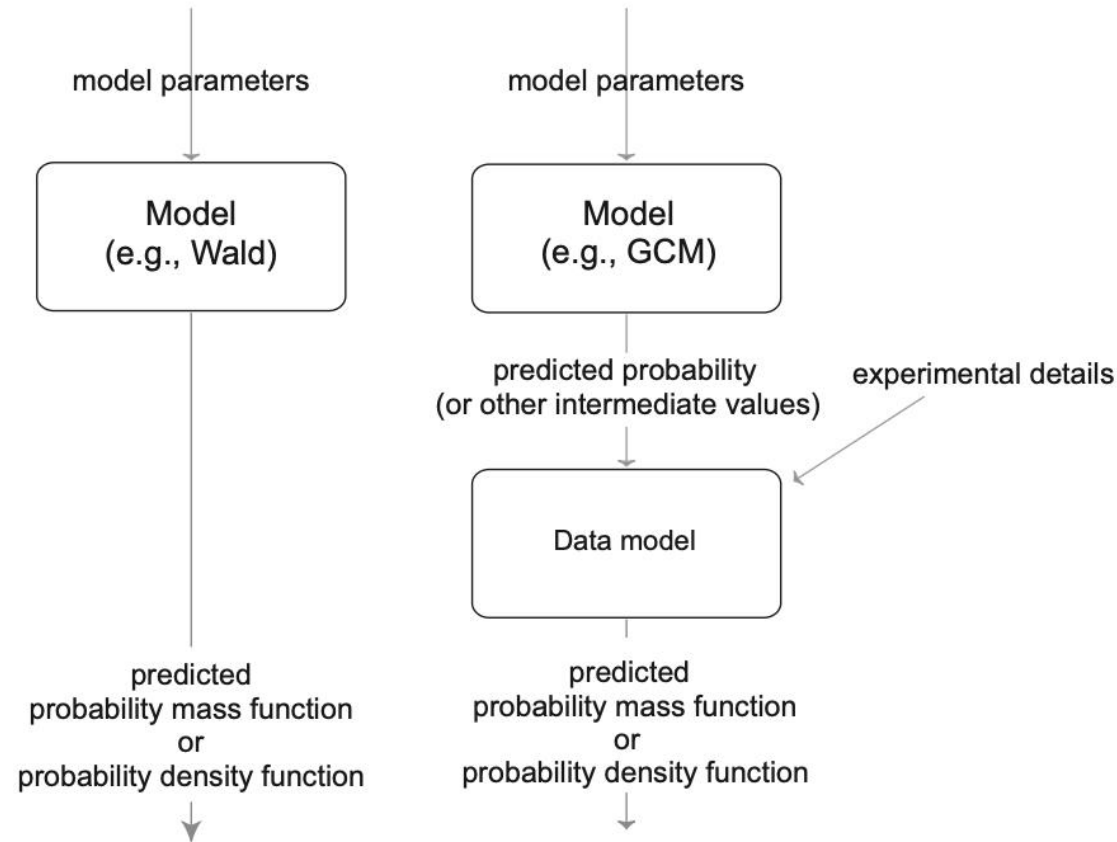
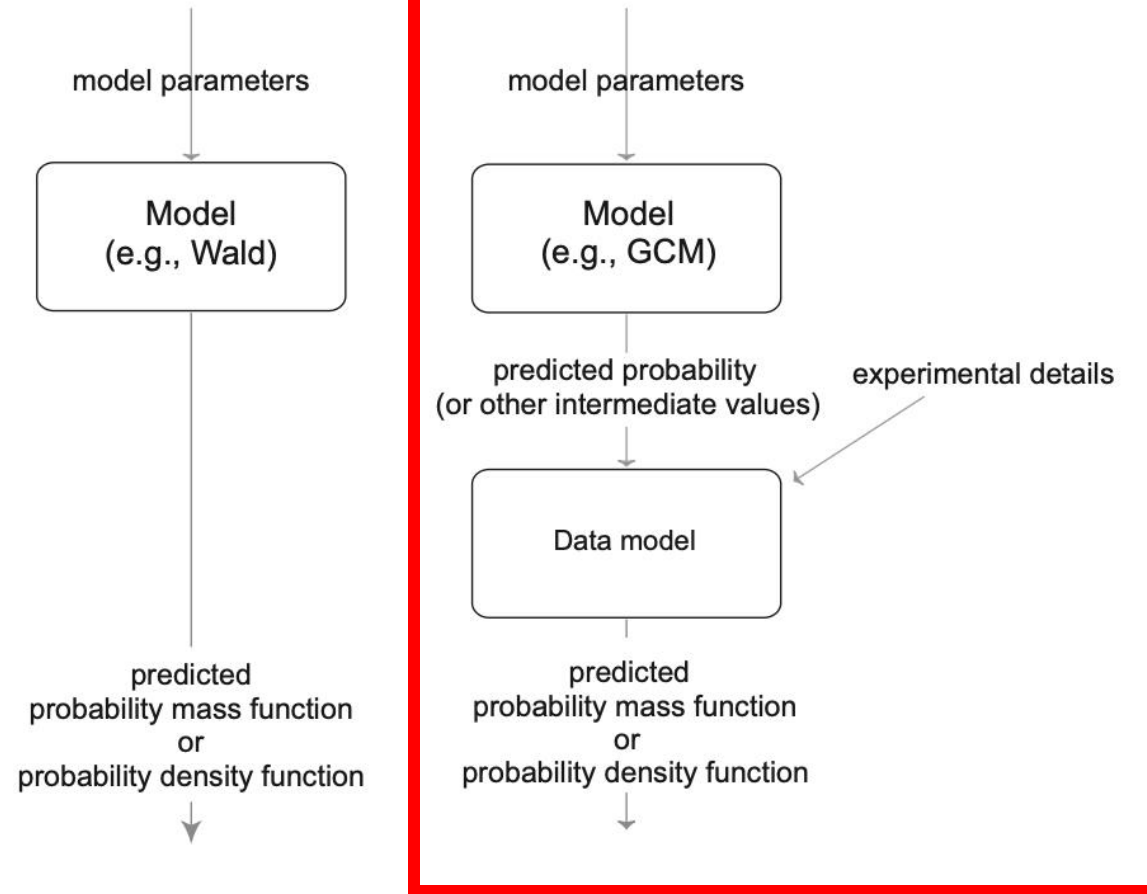


Figure 4.7 Farrell, S., & Lewandowsky, S. (2018). *Computational Modeling of Cognition and Behavior*. Cambridge University Press.

Two ways of likelihood/probability functions



Today's exercises will deal with this approach, but it is possibly to do both in the toolbox

Figure 4.7 Farrell, S., & Lewandowsky, S. (2018). *Computational Modeling of Cognition and Behavior*. Cambridge University Press.

Discrepancy functions: **likelihoods**

Let us look at a simple case, with binary responses. We will need to use:

- Bernoulli (binomial) distribution functions
- Model-assigned probabilities to two outcomes (model prediction)
- The observations we made (data)

We can relate probability of outcomes for each trial j with our observation on each trial, with N number of total trial.

$$P(y \mid \theta) = \prod_{j=1}^N P(y_j \mid \theta)$$



Discrepancy functions: **likelihoods**

Let us look at a simple case, with binary responses. We will need to use:

- Bernoulli (binomial) distribution functions
- Model-assigned probabilities to two outcomes (model prediction)
- The observations we made (data)

We can relate probability of outcomes for each trial j with our observation on each trial, with N number of total trial.

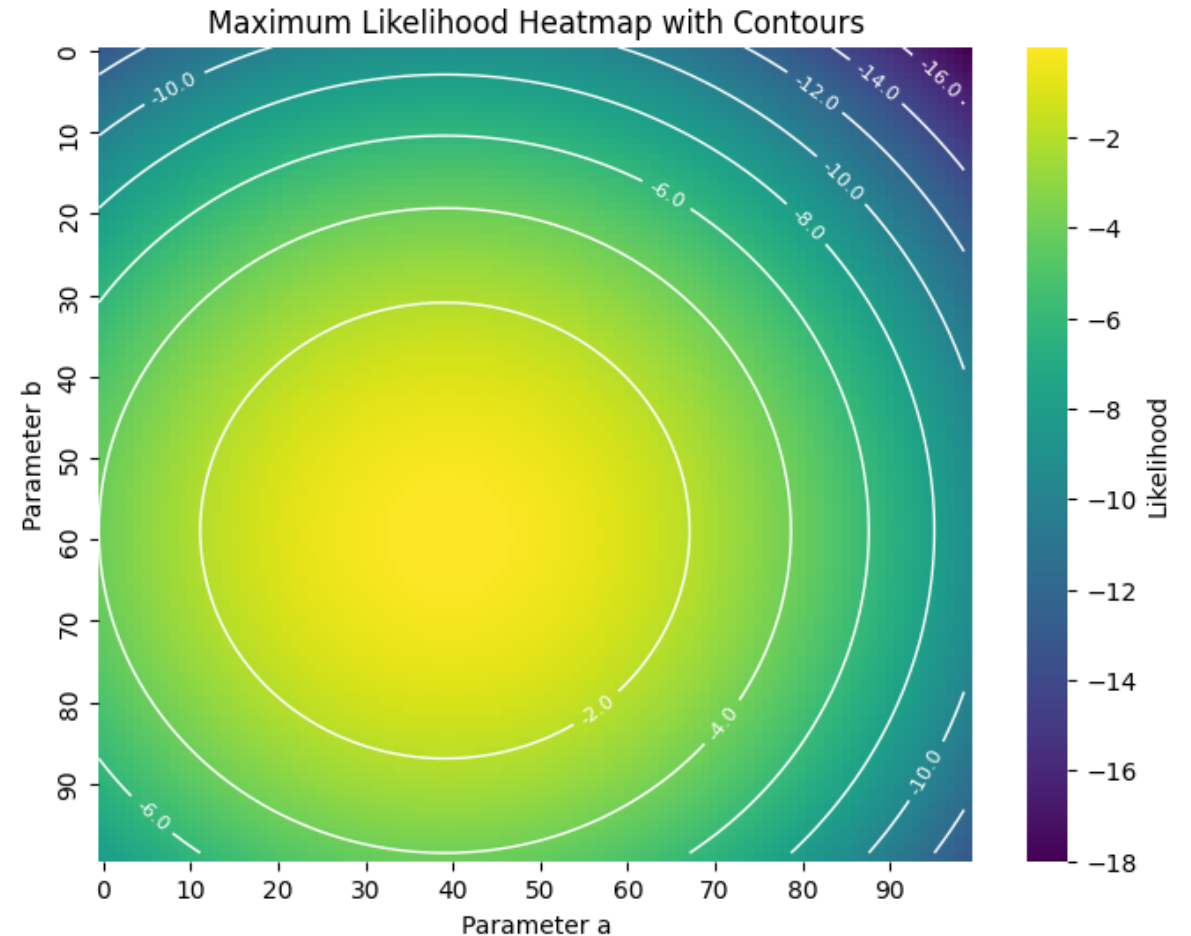
$$\text{Bernoulli}(y|\theta) = \begin{cases} \theta & | y = 1 \\ 1 - \theta & | y = 0 \end{cases}$$

$$P(y | \theta) = \prod_{j=1}^N P(y_j | \theta)$$



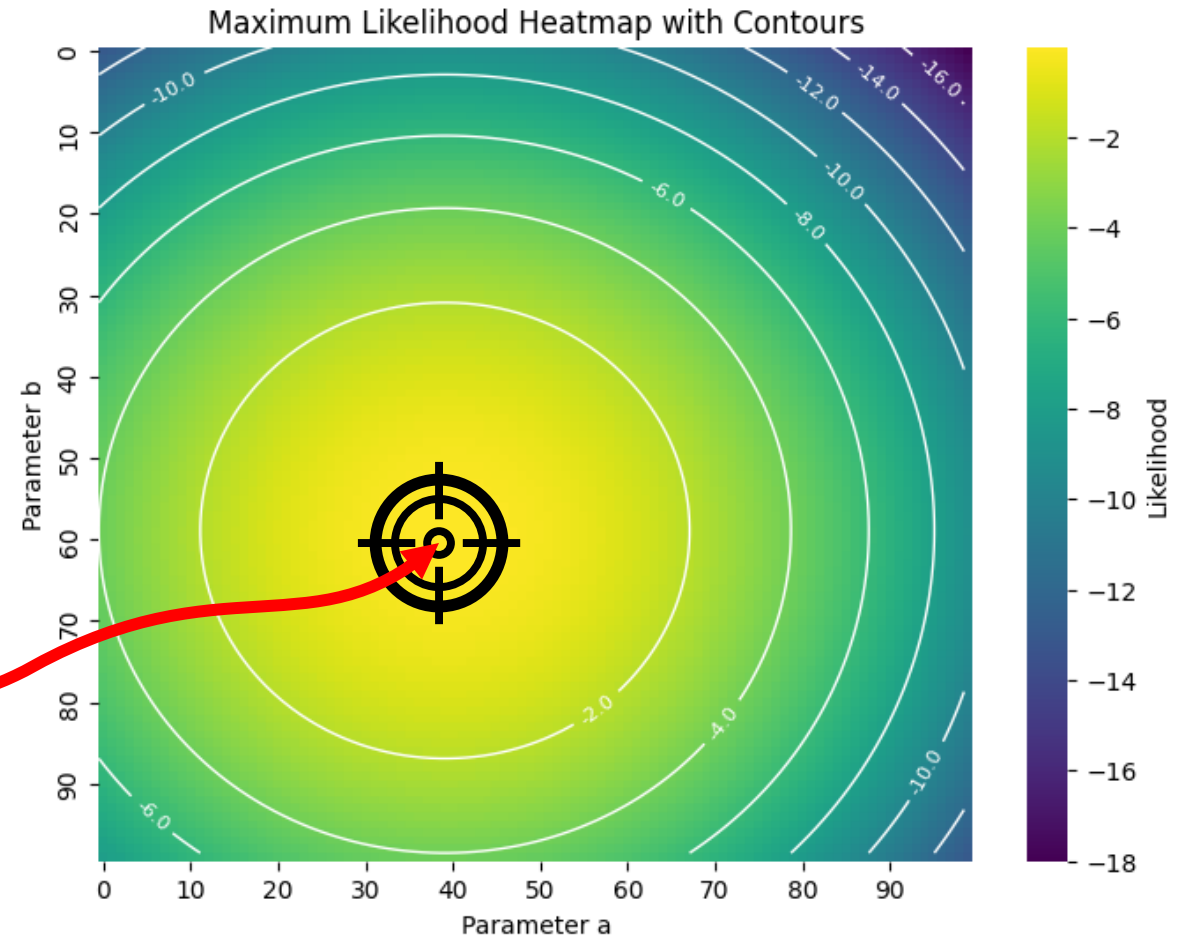
Parameter estimation: maximum likelihood estimation

We are often not concerned about all possible parameter values... only about the best-fitting parameters.



Parameter estimation: maximum likelihood estimation

We are often not concerned about all possible parameter values... only about **the best-fitting parameters.**



Parameter Estimation

Most parameter estimation techniques requires us to minimize our likelihood (objective) function:

$$\ln L(y_j|\theta) = - \sum_{j=1}^N \log[P(y_j|\theta)]$$



Parameter Estimation

The goal of parameter estimation is to find the parameters that maximise the agreement between the model and the data.

- Models that approximate the data well can tell you something about the utility of a model.



Parameter Estimation (caveats)

So, what doesn't it tell you? (Roberts & Pashler, 2000)

- The flexibility (how much data the model cannot fit?)
- The variability of the data (e.g. diagnosticity, how firmly the data rule out what the theory cannot fit)
- The likelihood of other outcomes (perhaps the theory could have fit any possible results)

Our belief in a model should require all of these aspects

Parameter recovery

“Mathematical models often reflect real-world systems and their parameters have biological, chemical, or physical interpretations, and not identifying these parameters can result in ambiguous interpretations.”

Petrica & Popescu (2024)

Parameter recovery

Models often have to satisfy a conditions, called **identifiability**: *the parameters of a model can be uniquely determined from observed data.*

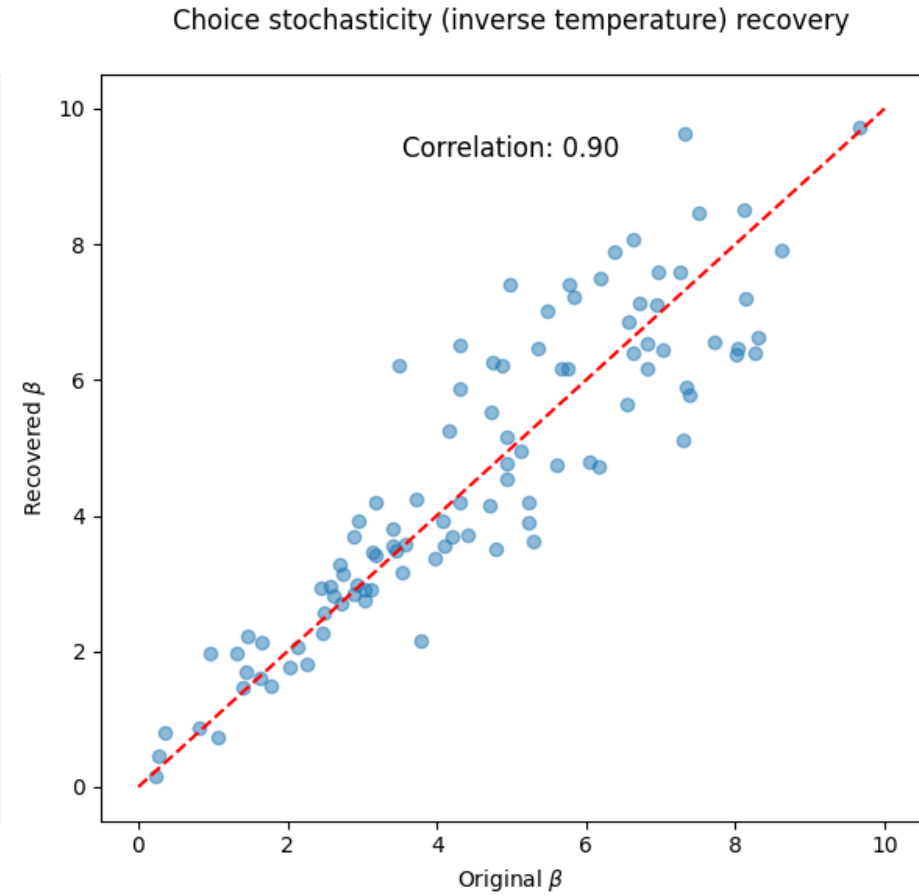
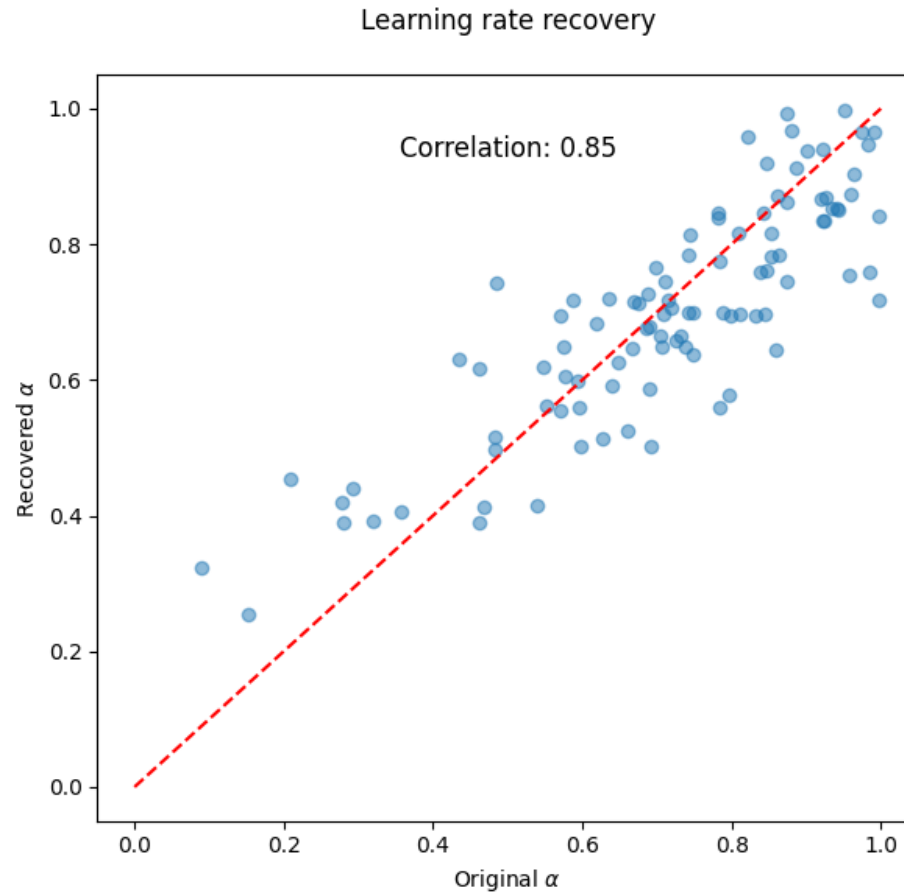
Parameter recovery

Models often have to satisfy a conditions, called **identifiability**: *the parameters of a model can be uniquely determined from observed data.*

If a parameter **IS IDENTIFIABLE**, given the model and data, there is **only one possible value** for that parameter.

If a parameter **IS NOT IDENTIFIABLE**, given the model and data, there are **multiple possible values** for that parameter.

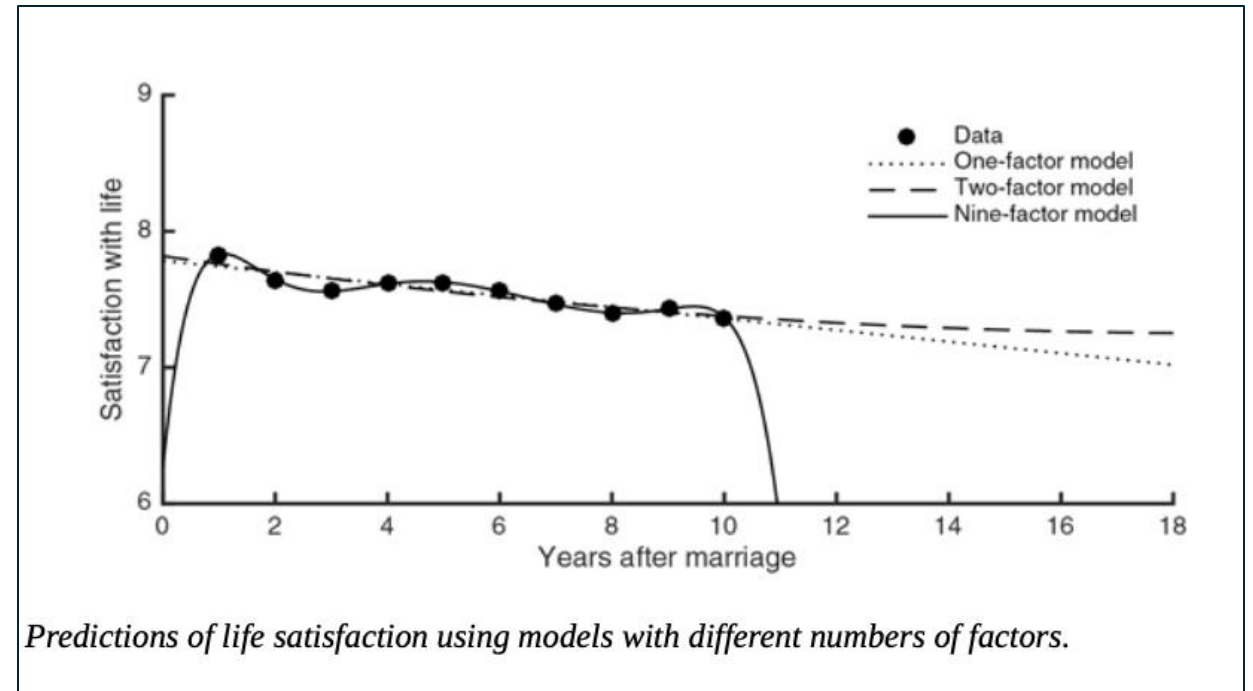
Parameter recovery



Multi-model comparisons: not all models created equal

The problem of **overfitting**: *the model fits the training data really well but may perform poorly for independent data (out-of-sample and test dataset)*

- this often becomes a problem of complexity
 - number of parameters should not approach the number of data points
 - results in a push towards parsimony



Christian, B., & Griffiths, T. (2016). *Algorithms to Live By: The Computer Science of Human Decisions*. Henry Holt and Company.

Multi-model comparisons: not all models created equal

Complexity: penalising the number of parameter

- Generalising fit across parameter space (e.g. Bayesian Information or Akaike Information Criterion)
- Incorporating functional form; variance explained by each parameter (e.g. Minimum Description Length, or Bayesian Model Selection)

Distinguishability: can the experimental design eliminate models?

- Model Recovery (how well models fit each other's generated data)

Global model analysis

- Landscaping
- Parameter Space Partitioning (also gives information about flexibility/complexity)

Multi-model comparisons: not all models created equal

Complexity: penalising the number of parameter

- Generalising fit across parameter space (e.g. Bayesian Information or Akaike Information Criterion)
- Incorporating functional form; variance explained by parameter (e.g. Minimum Description Length, or Bayesian Model Selection)

Distinguishability: can the experimental design eliminate models?

- **Model Recovery (how well models fit each other's generated data)**

Global model analysis

- Landscaping
- Parameter Space Partitioning (also gives information about flexibility/complexity)

Model recovery and confusion matrices

- Determine how identifiable the model is given the experiment (Steyvers et al., 2009)

		Predicted Model (model being fitted against the data)		
		Model 1 (M_1)	Model 2 (M_2)	Model 3 (M_3)
True Model (data generating model)	Model 1 (M_1)	$P(M_1 D_{M_1})$	$P(M_2 D_{M_1})$	$P(M_3 D_{M_1})$
	Model 2 (M_2)	$P(M_1 D_{M_2})$	$P(M_2 D_{M_2})$	$P(M_3 D_{M_2})$
	Model 3 (M_3)	$P(M_1 D_{M_3})$	$P(M_2 D_{M_3})$	$P(M_3 D_{M_3})$



Model recovery and confusion matrices

- Determine how identifiable the model is given the experiment (Steyvers et al., 2009)

		Predicted Model (model being fitted against the data)		
		Model 1 (M_1)	Model 2 (M_2)	Model 3 (M_3)
True Model (data generating model)	Model 1 (M_1)	$P(M_1 D_{M_1})$	$P(M_2 D_{M_1})$	$P(M_3 D_{M_1})$
	Model 2 (M_2)	$P(M_1 D_{M_2})$	$P(M_2 D_{M_2})$	$P(M_3 D_{M_2})$
	Model 3 (M_3)	$P(M_1 D_{M_3})$	$P(M_2 D_{M_3})$	$P(M_3 D_{M_3})$



Model recovery and confusion matrices

- Determine how identifiable the model is given the experiment (Steyvers et al., 2009)
- What is it good for?
 - Determine whether you can distinguish between models in your experiment using goodness-of-fit
 - Improve experimental design to distinguish models

Predicted Model (model being fitted against the data)

	Model 1 (M_1)	Model 2 (M_2)	Model 3 (M_3)
Model 1 (M_1)	$P(M_1 D_{M_1})$	$P(M_2 D_{M_1})$	$P(M_3 D_{M_1})$
Model 2 (M_2)	$P(M_1 D_{M_2})$	$P(M_2 D_{M_2})$	$P(M_3 D_{M_2})$
Model 3 (M_3)	$P(M_1 D_{M_3})$	$P(M_2 D_{M_3})$	$P(M_3 D_{M_3})$

True Model (data generating model)



Model recovery and confusion matrices

- Determine how identifiable the model is given the experiment (Opitz, 2009)

- What is it good for?

- Determine if two models can distinguish between two models in an experiment

- Improve experimental design to distinguish models

BEWARE!!!

Confusion matrices (sometimes also called error matrices) mean something slightly different in machine learning. See Opitz, 2024 for a brief review!

Predicted Model (model being fitted against the data)

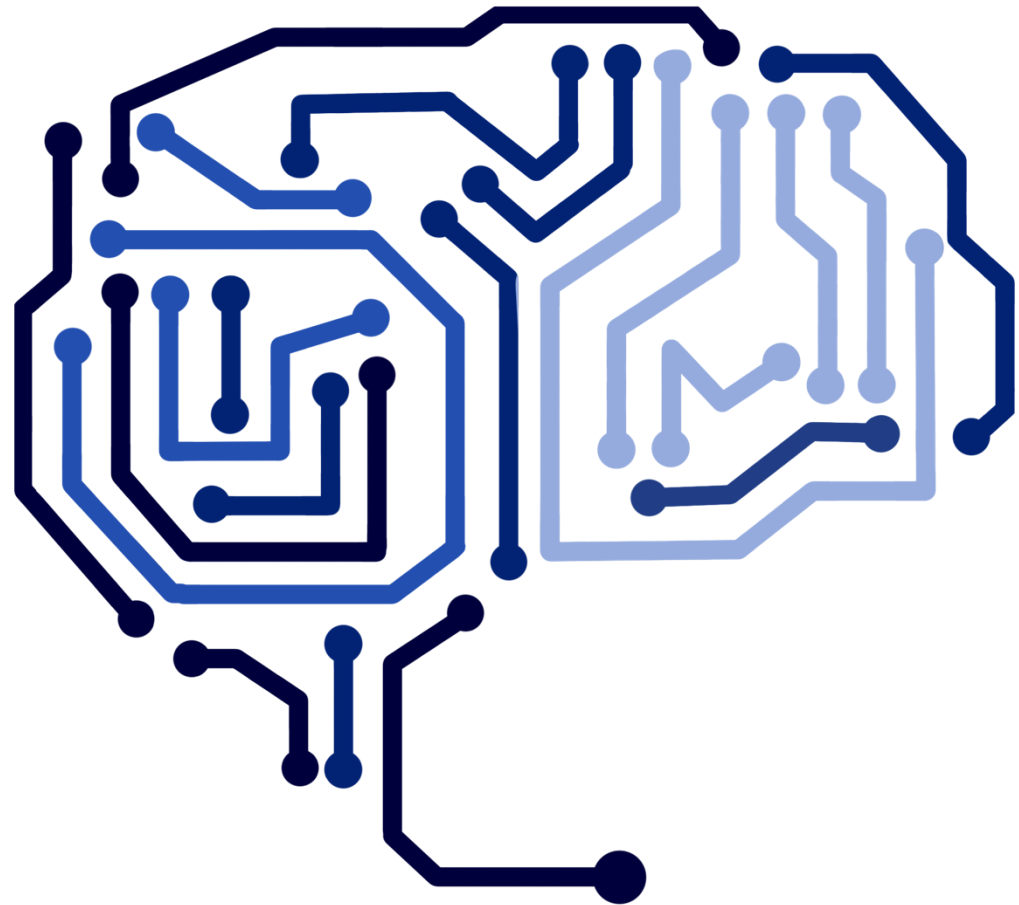
True Model (data model)				Model 3 (M_2)
				$P(M_3 D_{M_1})$
				$P(M_3 D_{M_2})$
	Model 2 (M_2)	$P(M_1 D_{M_2})$	$P(M_2 D_{M_2})$	$P(M_3 D_{M_2})$
	Model 3 (M_3)	$P(M_1 D_{M_3})$	$P(M_2 D_{M_3})$	$P(M_3 D_{M_3})$



The *cpm* toolbox

Getting you the tools to get going

Computational Psychiatry
Modeling Library in Python 🐍



Where to get the goods

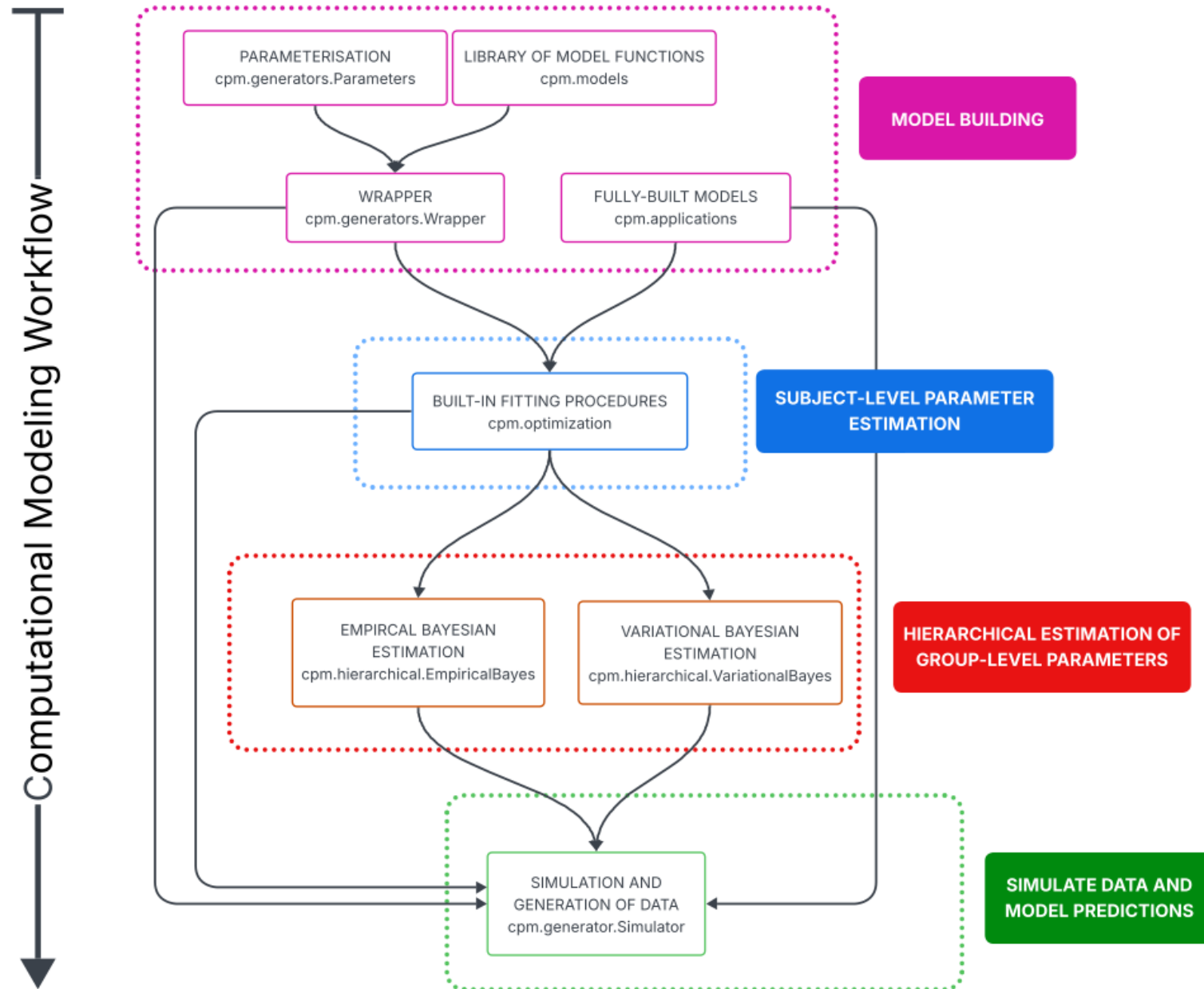
- <https://github.com/DevComPsy/cpm>

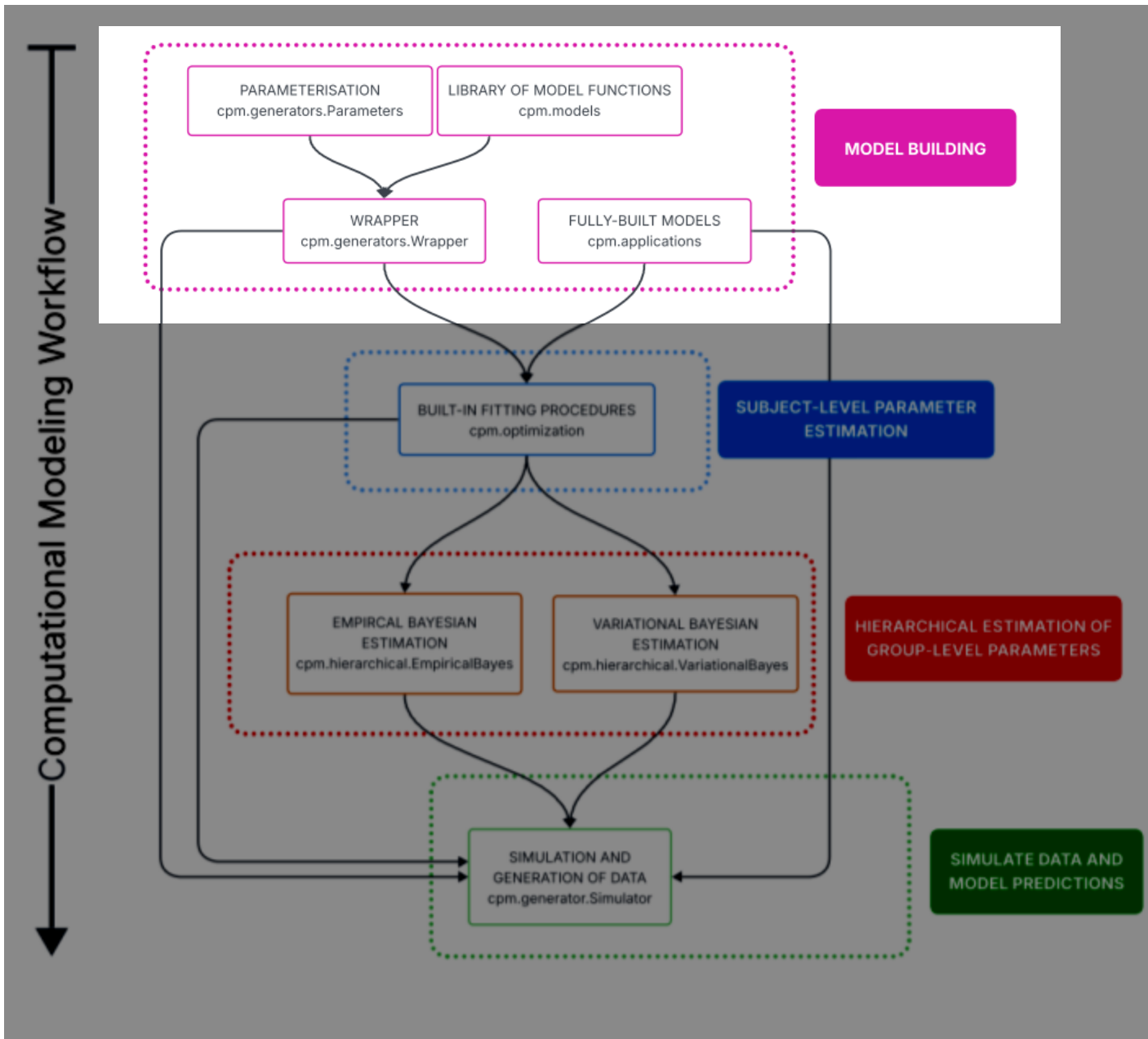
```
pip install git+https://github.com/DevComPsy/cpm.git
```

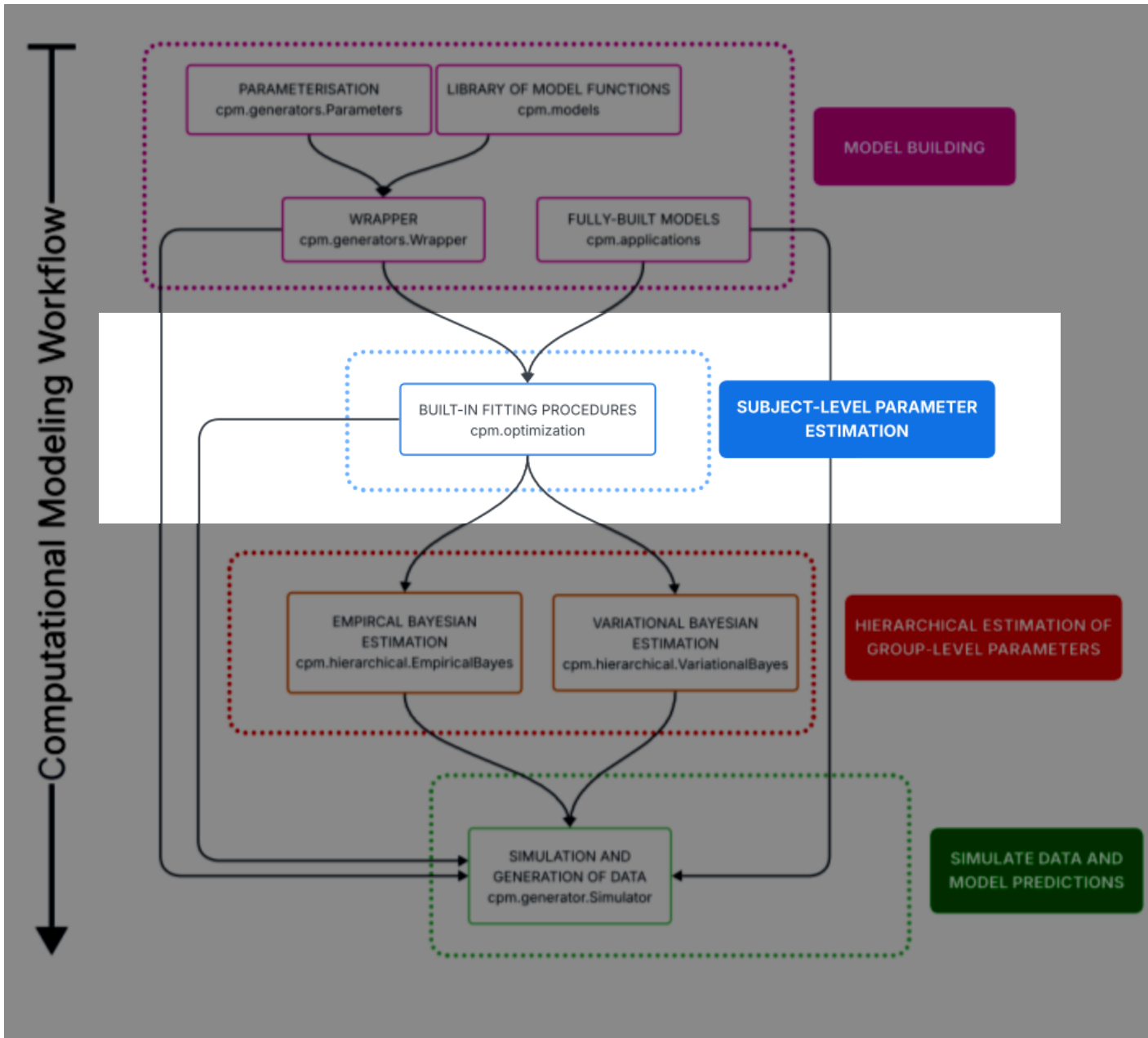

It is interoperable with widely-used libraries

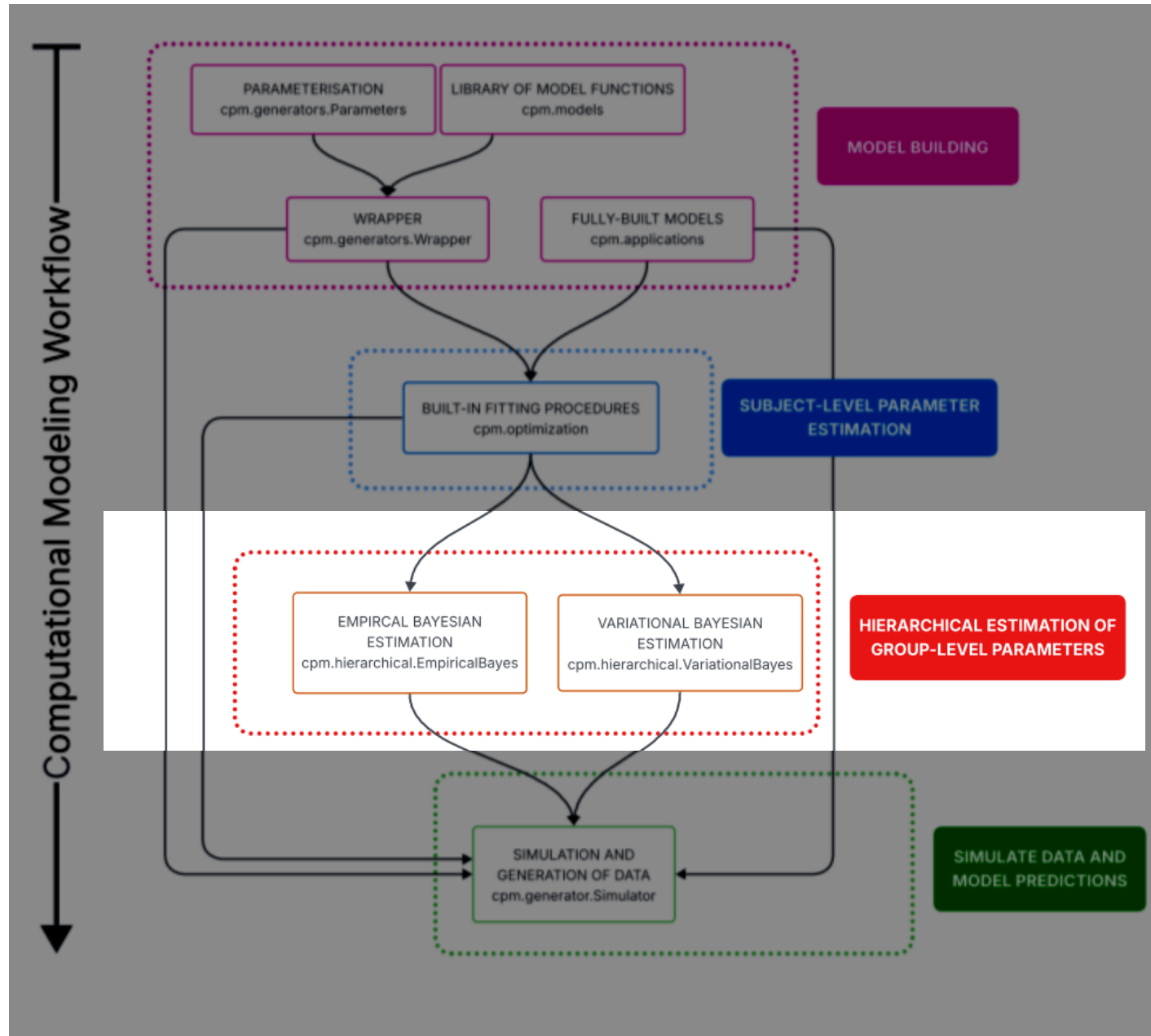
- **pandas** - a fast, powerful, flexible and easy to use open-source data analysis and manipulation tool
- **numpy** - a fundamental package for scientific computing with Python

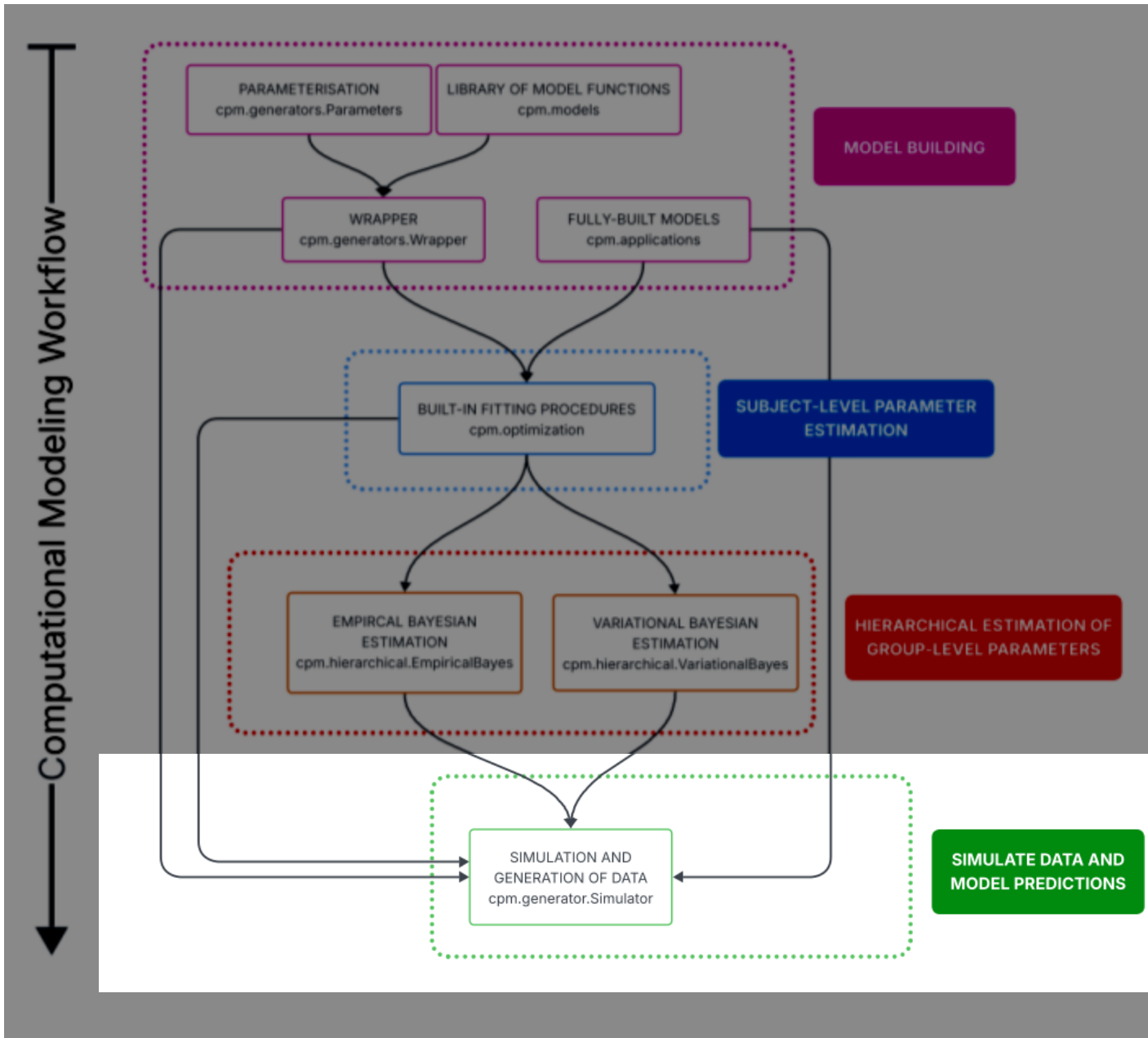
The computational workflow

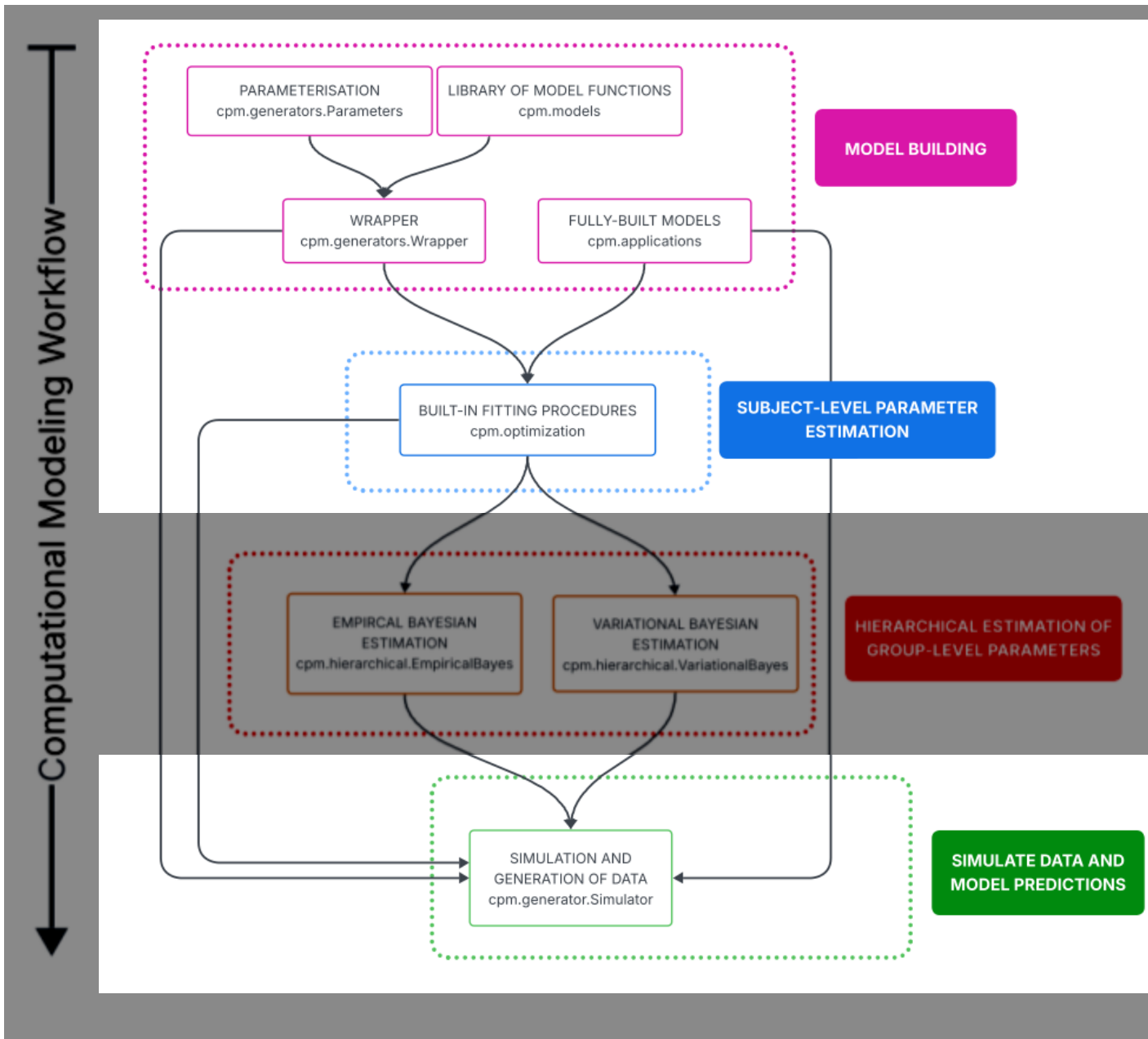












Model Construction: parameters

cpm.generators.Value

```
cpm.generators.Value(value=None, lower=0, upper=1, prior=None, args=None,
**kwargs)
```

The `Value` class is a wrapper around a float value, with additional details such as the prior distribution, lower and upper bounds. It supports all basic mathematical operations and can be used as a regular float with the parameter value as operand.

Parameters:

- `value` (float, default: `None`) – The value of the parameter.
- `lower` (float, default: `0`) – The lower bound of the parameter.
- `upper` (float, default: `1`) – The upper bound of the parameter.
- `prior` (string or object, default: `None`) – If a string, it should be one of continuous distributions from `scipy.stats`. See the [scipy documentation](#) for more details. The default is `None`. If an object, it should be or contain a callable function representing the prior distribution of the parameter with methods similar to `scipy.stats` distributions. See Notes for more details.
- `args` (dict, default: `None`) – A dictionary of arguments for the prior distribution function.

▼ Notes

We currently implement the following continuous distributions from `scipy.stats` corresponding to the `prior` argument:

- 'uniform'
- 'truncated_normal'
- 'beta'
- 'gamma'
- 'truncated_exponential'
- 'norm'

Because these distributions are inherited from `scipy.stats`, see the [scipy documentation](#) for more details on how to update variables of the distribution.

Returns:

- `Value` – A Value object, where each attribute is one of the arguments provided for the function. It support all basic mathematical operations and can be used as a regular float with the parameter value as operand.

cpm.generators.Parameters

```
cpm.generators.Parameters(**kwargs)
```

A class representing a set of parameters. It takes keyword arguments representing the parameters with their values and wraps them into a python object.

Parameters:

- `**kwargs` (dict, default: `{}`) – Keyword arguments representing the parameters.

Returns:

- `Parameters` – A Parameters object, where each attributes is one of the keyword arguments provided for the function wrapped by the Value class.

Examples:

```
>>> from cpm.generators import Parameters
>>> parameters = Parameters(a=0.5, b=0.5, c=0.5)
>>> parameters['a']
0.1
>>> parameters.a
0.1
>>> parameters()
{'a': 0.1, 'b': 0.2, 'c': 0.5}
```

The Parameters class can also provide a prior.

```
>>> x = Parameters(
>>>     a=Value(value=0.1, lower=0, upper=1, prior="normal", args={"mean": 0.5, "sd": 0.1}),
>>>     b=0.5,
>>>     weights=Value(value=[0.1, 0.2, 0.3], lower=0, upper=1, prior=None),
>>> )
```

```
>>> x.prior(log=True)
-6.5854298732499186
```

We can also sample new parameter values from the prior distributions.



Model Construction: parameters

cpm.generators.Value

```
cpm.generators.Value(value=None, lower=0, upper=1, prior=None, args=None,
**kwargs)
```

The `Value` class is a wrapper around a float value, with additional details such as the prior distribution, lower and upper bounds. It supports all basic mathematical operations and can be used as a regular float with the parameter value as operand.

Parameters:

- `value` (float, default: `None`) – The value of the parameter.
- `lower` (float, default: `0`) – The lower bound of the parameter.
- `upper` (float, default: `1`) – The upper bound of the parameter.
- `prior` (string or object, default: `None`) – If a string, it should be one of continuous distributions from `scipy.stats`. See the [scipy documentation](#) for more details. The default is `None`. If an object, it should be or contain a callable function representing the prior distribution of the parameter with methods similar to `scipy.stats` distributions. See Notes for more details.
- `args` (dict, default: `None`) – A dictionary of arguments for the prior distribution function.

Notes

We currently implement the following continuous distributions from `scipy.stats` corresponding to the `prior` argument:

- 'uniform'
- 'truncated_normal'
- 'beta'
- 'gamma'
- 'truncated_exponential'
- 'norm'

Because these distributions are inherited from `scipy.stats`, see the [scipy documentation](#) for more details on how to update variables of the distribution.

Returns:

- `Value` – A Value object, where each attribute is one of the arguments provided for the function. It support all basic mathematical operations and can be used as a regular float with the parameter value as operand.

cpm.generators.Parameters

```
cpm.generators.Parameters(**kwargs)
```

A class representing a set of parameters. It takes keyword arguments representing the parameters with their values and wraps them into a python object.

Parameters:

- `**kwargs` (dict, default: `{}`) – Keyword arguments representing the parameters.

Returns:

- `Parameters` – A Parameters object, where each attributes is one of the keyword arguments provided for the function wrapped by the Value class.

Examples:

```
>>> from cpm.generators import Parameters
>>> parameters = Parameters(a=0.5, b=0.5, c=0.5)
>>> parameters['a']
0.1
>>> parameters.a
0.1
>>> parameters()
{'a': 0.1, 'b': 0.2, 'c': 0.5}
```

complete parameterization

The Parameters class can also provide a prior.

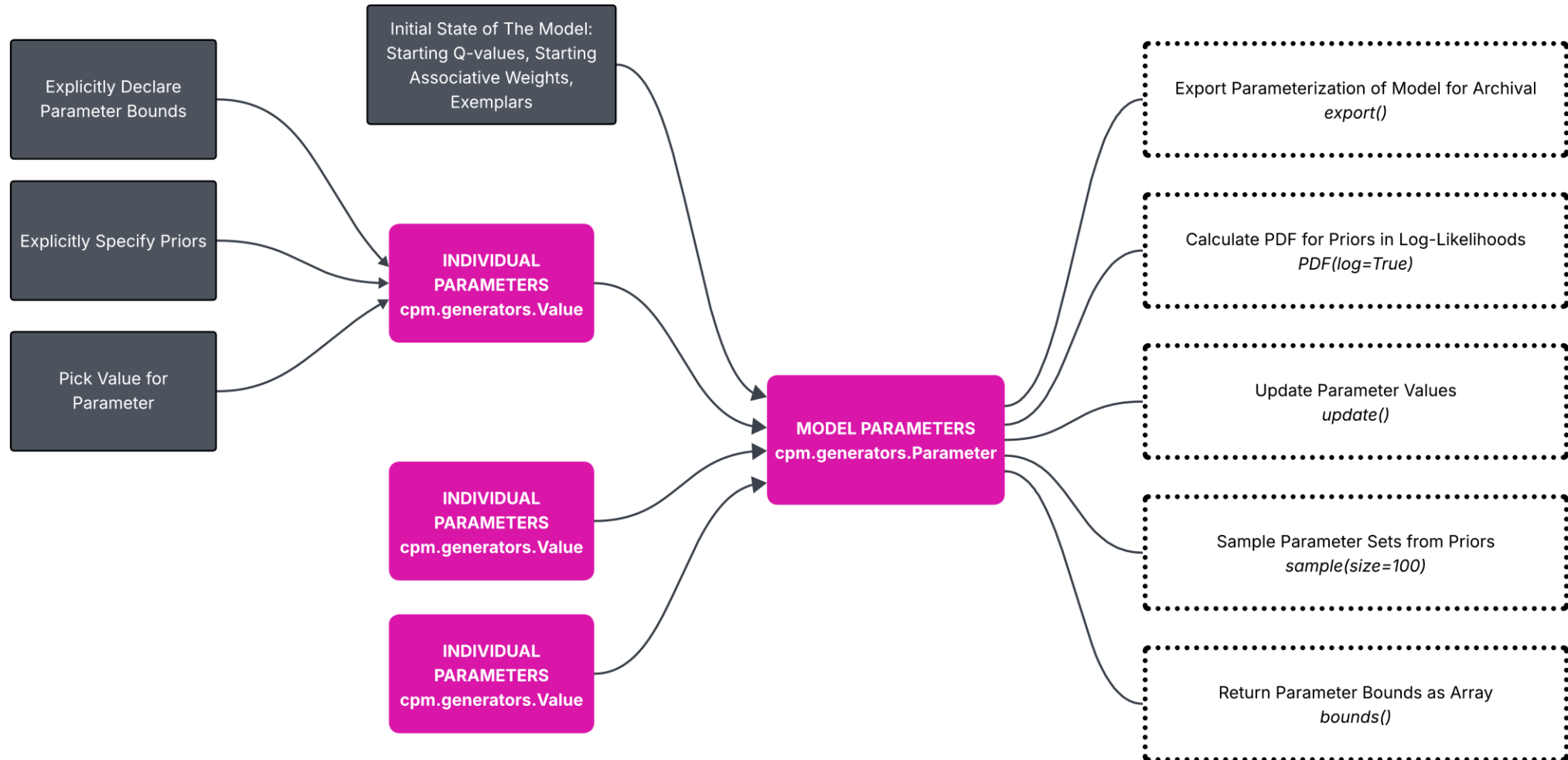
```
>>> x = Parameters(
>>>     a=Value(value=0.1, lower=0, upper=1, prior="normal", args={"mean": 0.5, "sd": 0.1}),
>>>     b=0.5,
>>>     weights=Value(value=[0.1, 0.2, 0.3], lower=0, upper=1, prior=None),
>>> )
```

```
>>> x.prior(log=True)
-6.5854298732499186
```

We can also sample new parameter values from the prior distributions.

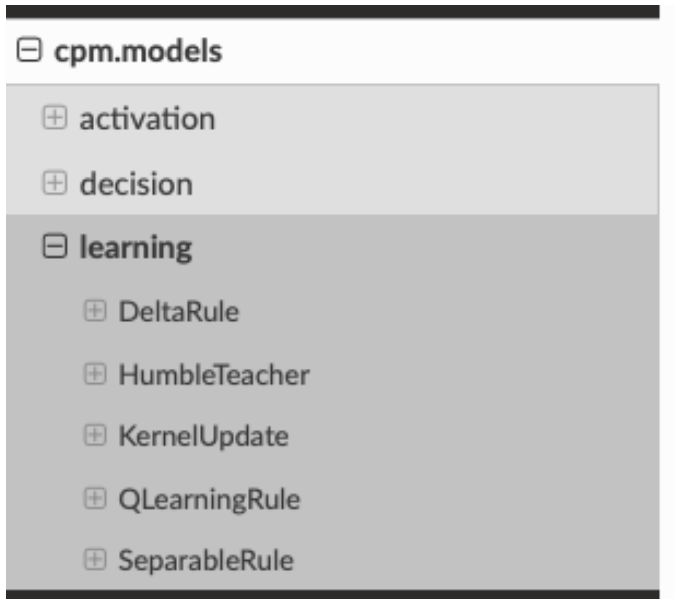


Model Construction: parameters



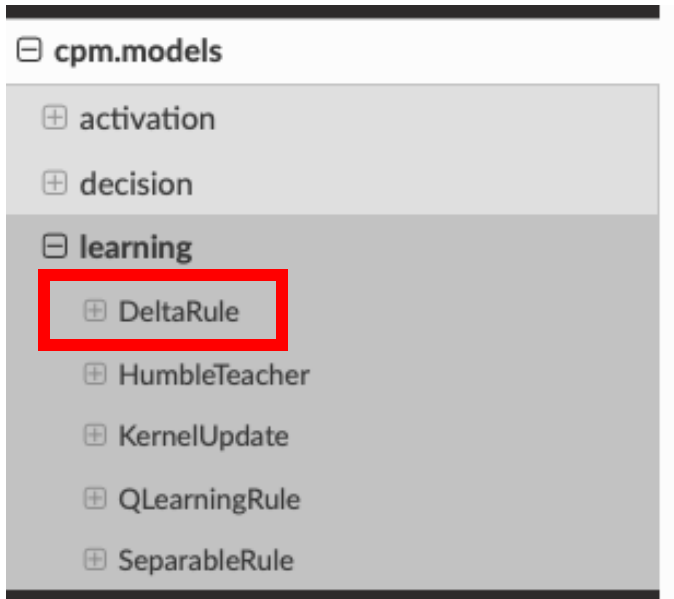
Model Construction: building your model

- **cpm.models** contain a library of algorithmic components.



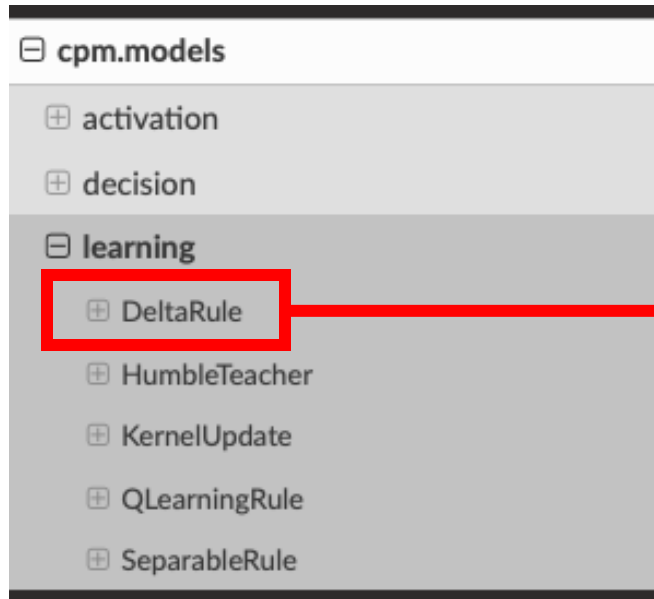
Model Construction: building your model

- **cpm.models** contain a library of algorithmic components.



Model Construction: building your model

- **cpm.models** contain a library of algorithmic components.



cpm.models.learning

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.

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()
array([[ 0.0575793,  0.09214091,  0. ],
       [-0.08837513, -0.1304325 ,  0.]])
```



```

from cpm.models import learning, decision, utils
import copy

def model(parameters, trial):
    # pull out the parameters
    alpha = parameters.alpha
    temperature = parameters.temperature
    values = np.array(parameters.values)
    # pull out the trial information
    stimulus = trial.get('trials')
    feedback = trial.get("feedback")
    mute = np.zeros(4) # mute learning for all cues not presented

    # activate the value of each available action
    # here there are two possible actions, that can take up on 4 different values
    # so we subset the values to only include the ones that are activated...
    # ...according to which stimuli was presented
    activation = values[stimulus - 1]
    # convert the activations to a 2x1 matrix, where rows are actions/outcomes
    activations = activation.reshape(2, 1)
    # calculate a policy based on the activations
    response = decision.Softmax(activations=activations, temperature=epsilon)
    response.compute() # compute the policy
    choice = response.choice() # get the choice based on the policy
    reward = feedback[choice] # get the reward of the chosen action

    # update the value of the chosen action
    mute[stimulus[choice] - 1] = 1 # unmute the learning for the chosen action
    teacher = np.array([reward])
    update = learning.SeparableRule(weights=values, feedback=teacher, input=mute, alpha=alpha)
    update.compute()
    values += update.weights.flatten()
    ## compile output
    output = {
        "policy" : response.policies,          # policies
        "response" : choice,                   # choice based on the policy
        "reward" : reward,                     # reward of the chosen action
        "values" : values,                     # updated values
        "change" : update.weights,             # change in the values
        "activation" : activations.flatten(),   # activation of the values
        "dependent" : response.policies,       # dependent variable
    }
    return output

```

```
from cpm.models import learning, decision, utils
import copy
```

```
def model(parameters, trial):
```

```
# pull out the parameters
```

```
alpha = parameters.alpha
temperature = parameters.temperature
values = np.array(parameters.values)
# pull out the trial information
stimulus = trial.get('trials')
feedback = trial.get("feedback")
mute = np.zeros(4) # mute learning for all cues not presented
```

```
# activate the value of each available action
```

```
# here there are two possible actions, that can take up on 4 different values
# so we subset the values to only include the ones that are activated...
# ...according to which stimuli was presented
activation = values[stimulus - 1]
# convert the activations to a 2x1 matrix, where rows are actions/outcomes
activations = activation.reshape(2, 1)
# calculate a policy based on the activations
response = decision.Softmax(activations=activations, temperature=epsilon)
response.compute() # compute the policy
choice = response.choice() # get the choice based on the policy
reward = feedback[choice] # get the reward of the chosen action

# update the value of the chosen action
mute[stimulus[choice] - 1] = 1 # unmute the learning for the chosen action
teacher = np.array([reward])
update = learning.SeparableRule(weights=values, feedback=teacher, input=mute, alpha=alpha)
update.compute()
values += update.weights.flatten()
```

```
output = {
    "policy" : response.policies, # policies
    "response" : choice, # choice based on the policy
    "reward" : reward, # reward of the chosen action
    "values" : values, # updated values
    "change" : update.weights, # change in the values
    "activation" : activations.flatten(), # activation of the values
    "dependent" : response.policies, # dependent variable
}
return output
```

```
from cpm.models import learning, decision, utils
import copy
```

```
def model(parameters, trial):
```

```
# pull out the parameters
alpha = parameters.alpha
temperature = parameters.temperature
values = np.array(parameters.values)
# pull out the trial information
stimulus = trial.get('trials')
feedback = trial.get("feedback")
mute = np.zeros(4) # mute learning for all cues not presented
```

```
# activate the value of each available action
# here there are two possible actions, that can take up on 4 different values
# so we subset the values to only include the ones that are activated...
# ...according to which stimuli was presented
activation = values[stimulus - 1]
# convert the activations to a 2x1 matrix, where rows are actions/outcomes
activations = activation.reshape(2, 1)
# calculate a policy based on the activations
response = decision.Softmax(activations=activations, temperature=epsilon)
response.compute() # compute the policy
choice = response.choice() # get the choice based on the policy
reward = feedback[choice] # get the reward of the chosen action

# update the value of the chosen action
mute[stimulus[choice] - 1] = 1 # unmute the learning for the chosen action
teacher = np.array([reward])
update = learning.SeparableRule(weights=values, feedback=teacher, input=mute, alpha=alpha)
update.compute()
values += update.weights.flatten()
```

```
output = {
    "policy" : response.policies, # policies
    "response" : choice, # choice based on the policy
    "reward" : reward, # reward of the chosen action
    "values" : values, # updated values
    "change" : update.weights, # change in the values
    "activation" : activations.flatten(), # activation of the values
    "dependent" : response.policies, # dependent variable
}
return output
```

parameters: cpm.generators.Parameters
trial: any information the model needs from the environment, pandas.Series


```
from cpm.models import learning, decision, utils
import copy
```

```
def model(parameters, trial):
```

```
# pull out the parameters
alpha = parameters.alpha
temperature = parameters.temperature
values = np.array(parameters.values)
# pull out the trial information
stimulus = trial.get('trials')
feedback = trial.get("feedback")
mute = np.zeros(4) # mute learning for all cues not presented
```

pulling out your parameters

```
# activate the value of each available action
# here there are two possible actions, that can take up on 4 different values
# so we subset the values to only include the ones that are activated...
# ...according to which stimuli was presented
activation = values[stimulus - 1]
# convert the activations to a 2x1 matrix, where rows are actions/outcomes
activations = activation.reshape(2, 1)
# calculate a policy based on the activations
response = decision.Softmax(activations=activations, temperature=epsilon)
response.compute() # compute the policy
choice = response.choice() # get the choice based on the policy
reward = feedback[choice] # get the reward of the chosen action

# update the value of the chosen action
mute[stimulus[choice] - 1] = 1 # unmute the learning for the chosen action
teacher = np.array([reward])
update = learning.SeparableRule(weights=values, feedback=teacher, input=mute, alpha=alpha)
update.compute()
values += update.weights.flatten()
```

```
output = {
    "policy" : response.policies, # policies
    "response" : choice, # choice based on the policy
    "reward" : reward, # reward of the chosen action
    "values" : values, # updated values
    "change" : update.weights, # change in the values
    "activation" : activations.flatten(), # activation of the values
    "dependent" : response.policies, # dependent variable
}
return output
```

parameters: cpm.generators.Parameters
trial: any information the model needs from the environment, pandas.Series

```
from cpm.models import learning, decision, utils
import copy
```

```
def model(parameters, trial):
```

```
# pull out the parameters
alpha = parameters.alpha
temperature = parameters.temperature
values = np.array(parameters.values)
# pull out the trial information
stimulus = trial.get('trials')
feedback = trial.get("feedback")
mute = np.zeros(4) # mute learning for all cues not presented
```

pulling out your parameters

```
# activate the value of each available action
# here there are two possible actions, that can take up on 4 different values
# so we subset the values to only include the ones that are activated...
# ...according to which stimuli was presented
activation = values[stimulus - 1]
# convert the activations to a 2x1 matrix, where rows are actions/outcomes
activations = activation.reshape(2, 1)
# calculate a policy based on the activations
response = decision.Softmax(activations=activations, temperature=epsilon)
response.compute() # compute the policy
choice = response.choice() # get the choice based on the policy
reward = feedback[choice] # get the reward of the chosen action

# update the value of the chosen action
mute[stimulus[choice] - 1] = 1 # unmute the learning for the chosen action
teacher = np.array([reward])
update = learning.SeparableRule(weights=values, feedback=teacher, input=mute, alpha=alpha)
update.compute()
values += update.weights.flatten()
```

your model code goes here

```
output = {
    "policy" : response.policies, # policies
    "response" : choice, # choice based on the policy
    "reward" : reward, # reward of the chosen action
    "values" : values, # updated values
    "change" : update.weights, # change in the values
    "activation" : activations.flatten(), # activation of the values
    "dependent" : response.policies, # dependent variable
}
return output
```

parameters: cpm.generators.Parameters
trial: any information the model needs from the environment, pandas.Series

```
from cpm.models import learning, decision, utils
import copy
```

```
def model(parameters, trial):
```

```
# pull out the parameters
alpha = parameters.alpha
temperature = parameters.temperature
values = np.array(parameters.values)
# pull out the trial information
stimulus = trial.get('trials')
feedback = trial.get("feedback")
mute = np.zeros(4) # mute learning for all cues not presented
```

pulling out your parameters

```
# activate the value of each available action
# here there are two possible actions, that can take up on 4 different values
# so we subset the values to only include the ones that are activated...
# ...according to which stimuli was presented
activation = values[stimulus - 1]
# convert the activations to a 2x1 matrix, where rows are actions/outcomes
activations = activation.reshape(2, 1)
# calculate a policy based on the activations
response = decision.Softmax(activations=activations, temperature=epsilon)
response.compute() # compute the policy
choice = response.choice() # get the choice based on the policy
reward = feedback[choice] # get the reward of the chosen action

# update the value of the chosen action
mute[stimulus[choice] - 1] = 1 # unmute the learning for the chosen action
teacher = np.array([reward])
update = learning.SeparableRule(weights=values, feedback=teacher, input=mute, alpha=alpha)
update.compute()
values += update.weights.flatten()
```

your model code goes here

```
output = {
    "policy" : response.policies, # policies
    "response" : choice, # choice based on the policy
    "reward" : reward, # reward of the chosen action
    "values" : values, # updated values
    "change" : update.weights, # change in the values
    "activation" : activations.flatten(), # activation of the values
    "dependent" : response.policies, # dependent variable
}
return output
```

list all things you want to save

parameters: cpm.generators.Parameters
trial: any information the model needs from the environment, pandas.Series

Model Construction: taking in your specification and applying it to trial-level data

cpm.generators

- **cpm.generators.Parameters**,
cpm.generators.Value
- **cpm.generators.Wrapper** (turns model parameterization and model function in a fully-fledged model)
- **cpm.generators.Simulator** (simulates data for multiple participants)

Model Construction: taking in your specification and applying it to trial-level data

cpm.generators

- `cpm.generators.Parameters`, `cpm.generators.Value`
- **`cpm.generators Wrapper`** (turns model parameterization and model function in a fully-fledged model)
- **`cpm.generators.Simulator`** (simulates data for multiple participants)

`cpm.generators Wrapper(model=None, data=None, parameters=None)`

A `Wrapper` class for a model function in the CPM toolbox. It is designed to run a model for a **single** experiment (participant) and store the output in a format that can be used for further analysis.

Parameters:

- `model` (`function` , default: `None`) – The model function that calculates the output(s) of the model for a single trial. See Notes for more information. See Notes for more information.
- `data` (`DataFrame or dict` , default: `None`) – If a `pandas.DataFrame` , it must contain information about each trial in the experiment that serves as an input to the model. Each trial is a complete row. If a `dictionary` , it must contain information about the each state in the environment or each trial in the experiment - all input to the model that are not parameters.
- `parameters` (`Parameters object` , default: `None`) – The parameters object for the model that contains all parameters (and initial states) for the model. See Notes on how it is updated internally.

Returns:

- `Wrapper` (`object`) – A Wrapper object.

▼ Notes

The model function should take two arguments: `parameters` and `trial` . The `parameters` argument should be a `Parameter` object specifying the model parameters. The `trial` argument should be a `dictionary` or `pd.Series` containing all input to the model on a single trial. The model function should return a `dictionary` containing the model output for the trial. If the model is intended to be fitted to data, its output should contain the following keys:

- 'dependent': Any dependent variables calculated by the model that will be used for the loss function.

If a model output contains any keys that are also present in parameters, it updates those in the parameters based on the model output.

Information on how to compile the model can be found in the [\[Tutorials - Build your model\]\[tutorials/defining-model\]](#) module.



Model Construction: taking in your specification and applying it to trial-level data

cpm.generators

- `cpm.generators.Parameters`, `cpm.generators.Value`
- **`cpm.generators.Wrapper`** (turns model parameterization and model function in a fully-fledged model)
- **`cpm.generators.Simulator`** (simulates data for multiple participants)

`export()`

Export the trial-level simulation details.

Returns: • `DataFrame` – A dataframe containing the model output for each participant and trial. If the output variable is organised as an array with more than one dimension, the output will be flattened.

`reset(parameters=None, data=None)`

Reset the model.

Parameters: • `parameters` (dict, array_like, Series or Parameters), default: `None` – The parameters to reset the model with.

Notes

When resetting the model, and `parameters` is `None`, reset model to initial state. If parameter is `array_like`, it resets the only the parameters in the order they are provided, where the last parameter updated is the element in `parameters` corresponding to `len(parameters)`.

Examples:

```
>>> x = Wrapper(model = mine, data = data, parameters = params)
>>> x.run()
>>> x.reset(parameters = [0.1, 1])
>>> x.run()
>>> x.reset(parameters = {'alpha': 0.1, 'temperature': 1})
>>> x.run()
>>> x.reset(parameters = np.array([0.1, 1, 0.5]))
>>> x.run()
```

Returns: • `None` –

`run()`

Run the model.

Returns: • `None` –



Model Construction: simulating data

cpm.generators

- **cpm.generators.Parameters**, **cpm.generators.Value**
- **cpm.generators Wrapper** (turns model parameterization and model function in a fully-fledged model)
- **cpm.generators.Simulator** (simulates data for multiple participants)

cpm.generators.Simulator(wrapper=None, data=None, parameters=None)

A **Simulator** class for a model in the CPM toolbox. It is designed to run a model for **multiple** participants and store the output in a format that can be used for further analysis.

Parameters:

- **wrapper** (**Wrapper** , default: **None**) – An initialised Wrapper object for the model.
- **data** (**pandas.core.groupby.generic.DataFrameGroupBy** or **list of dictionaries** , default: **None**) – The data required for the simulation. If it is a **pandas.core.groupby.generic.DataFrameGroupBy**, as returned by **pandas.DataFrame.groupby()**, each group must contain the data (or environment) for a single participant. If it is a list of dictionaries, each dictionary must contain the data (or environment) for a single participant.
- **parameters** ((**Parameters**, **DataFrame**, **Series** or **list**) , default: **None**) – The parameters required for the simulation. It can be a **Parameters** object or a list of dictionaries whose length is equal to data. If it is a **Parameters** object, Simulator will use the same parameters for all simulations. If it is a list of dictionaries, it will use match the parameters with data, so that for example parameters[6] will be used for the simulation of data[6].

Returns:

- **simulator** (**Simulator**) – A Simulator object.

export(save=False, path=None)

Return the trial- and participant-level information about the simulation.

Parameters:

- **save** (**bool** , default: **False**) – If True, the output will be saved to a file.
- **path** (**str** , default: **None**) – The path to save the output to.

Returns:

- **DataFrame** – A dataframe containing the the model output for each participant and trial. If the output variable is organised as an array with more than one dimension, the output will be flattened.

generate(variable='dependent')

Generate data for parameter recovery, etc.

Parameters:

- **variable** – Name of the variable to pull out from model output.



Fitting models to data

- **cpm.optimisation** (interface to parameter estimation techniques from scipy and other libraries)
 - Fmin
 - FminBound
 - BADS (Bayesian Adaptive Direct Search)
 - DifferentialEvolution
 - Minimse
- **cpm.optimisation.minimise** (a battery of discrepancy functions)
 - e.g. LogLikelihoods
 - e.g. Non-parametric functions



Fitting models to data

```
cpm.optimisation.FminBound(model=None, data=None, initial_guess=None,
number_of_starts=1, minimisation=None, cl=None, parallel=False, libraries=
['numpy', 'pandas'], prior=False, ppt_identifier=None, display=False, **kwargs)
```

Class representing the Fmin search (bounded) optimization algorithm using the L-BFGS-B method.

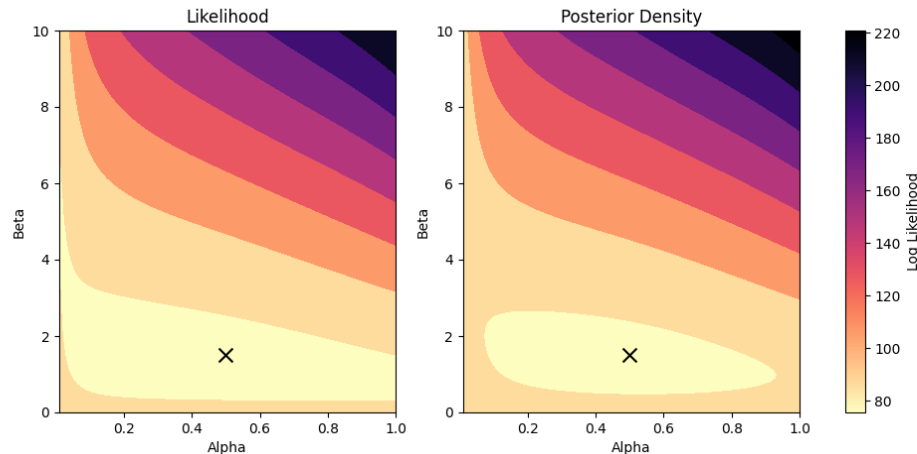
Parameters:

- `model` (`Wrapper` , default: `None`) – The model to be optimized.
- `data` ((`DataFrame` , `DataFrameGroupBy` , `list`) , default: `None`) – The data used for optimization. If a `pd.DataFrame`, it is grouped by the `ppt_identifier`. If it is a `pd.DataFrameGroupby`, groups are assumed to be participants. An array of dictionaries, where each dictionary contains the data for a single participant, including information about the experiment and the results too. See Notes for more information.
- `minimisation` (`function` , default: `None`) – The loss function for the objective minimization function. See the `minimise` module for more information. User-defined loss functions are also supported.
- `prior` – Whether to include the prior in the optimization. Default is `False`.
- `number_of_starts` (`int` , default: `1`) – The number of random initialisations for the optimization. Default is `1`.
- `initial_guess` (`list or array - like` , default: `None`) – The initial guess for the optimization. Default is `None`. If `number_of_starts` is set, and the `initial_guess` parameter is 'None', the initial guesses are randomly generated from a uniform distribution.
- `parallel` (`bool` , default: `False`) – Whether to use parallel processing. Default is `False`.
- `cl` (`int` , default: `None`) – The number of cores to use for parallel processing. Default is `None`. If `None`, the number of cores is set to 2. If `cl` is set to `None` and `parallel` is set to `True`, the number of cores is set to the number of cores available on the machine.
- `libraries` (`list` , default: `['numpy', 'pandas']`) – The libraries to import for parallel processing for `ipyparallel` with the IPython kernel. Default is `["numpy", "pandas"]`.
- `ppt_identifier` (`str` , default: `None`) – The key in the participant data dictionary that contains the participant identifier. Default is `None`. Returned in the optimization details.
- `**kwargs` (`dict` , default: `{}`) – Additional keyword arguments. See the `scipy.optimize.fmin_l_bfgs_b` documentation for what is supported.

built-in functionalities not
available elsewhere

Hierarchical fitting: model building

- priors are the natural part of **cpm.generators.Parameters**
- priors are defined in the parameterisation of the model
- priors are required



Hierarchical fitting: model building

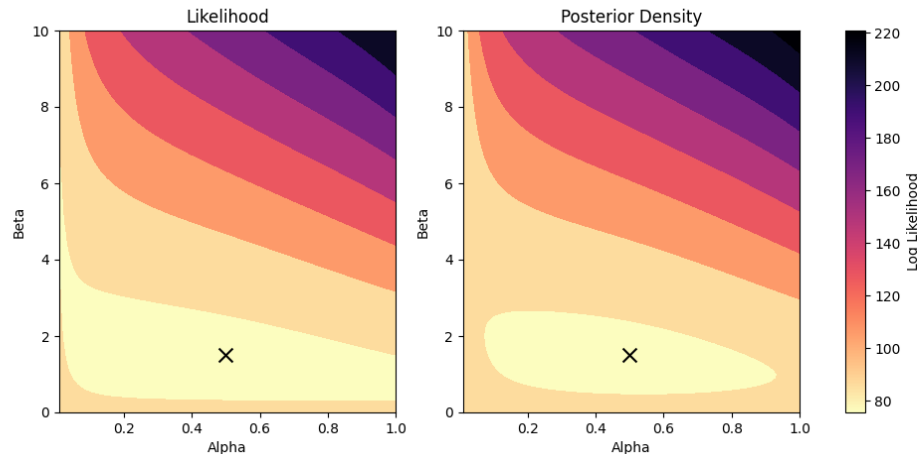
- priors are the natural part of **cpm.generators.Parameters**
- priors are defined in the parameterisation of the model
- priors are required

cpm.generators.Parameters(kwargs)**

A class representing a set of parameters. It takes keyword arguments representing the parameters with their values and wraps them into a python object.

Parameters: • ****kwargs** (dict, default: {}) – Keyword arguments representing the parameters.

Returns: • **Parameters** – A Parameters object, where each attributes is one of the keyword arguments provided for the function wrapped by the Value class.



The Parameters class can also provide a prior.

```
>>> x = Parameters(  
>>>     a=Value(value=0.1, lower=0, upper=1, prior="normal", args={"mean": 0.5, "sd": 0.1}),  
>>>     b=0.5,  
>>>     weights=Value(value=[0.1, 0.2, 0.3], lower=0, upper=1, prior=None),  
>>> )
```

```
>>> x.prior(log=True)  
-6.5854290732499186
```

Hierarchical fitting: model building

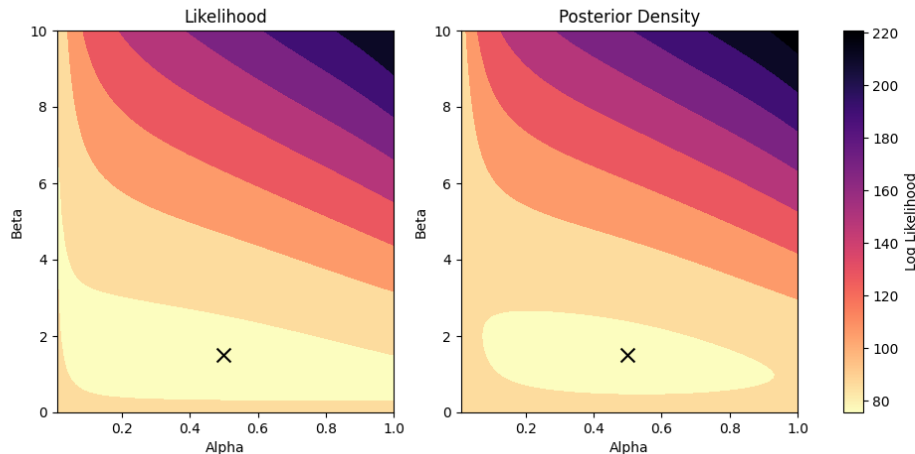
- priors are the natural part of **cpm.generators.Parameters**
- priors are defined in the parameterisation of the model
- priors are required

cpm.generators.Parameters(kwargs)**

A class representing a set of parameters. It takes keyword arguments representing the parameters with their values and wraps them into a python object.

Parameters: • **kwargs** (dict, default: {}) – Keyword arguments representing the parameters.

Returns: • **Parameters** – A Parameters object, where each attributes is one of the keyword arguments provided for the function wrapped by the Value class.



The Parameters class can also provide a prior.

```
>>> x = Parameters(  
>>>     a=Value(value=0.1, lower=0, upper=1, prior="normal", args={"mean": 0.5, "sd": 0.1}),  
>>>     b=0.5,  
>>>     weights=Value(value=[0.1, 0.2, 0.3], lower=0, upper=1, prior=None),  
>>> )
```

```
>>> x.prior(log=True)  
-6.5854290732499186
```

Hierarchical fitting: model building

- priors are the natural part of **cpm.generators.Parameters**
- priors are defined in the parameterisation of the model
- priors are required

cpm.generators.Parameters(kwargs)**

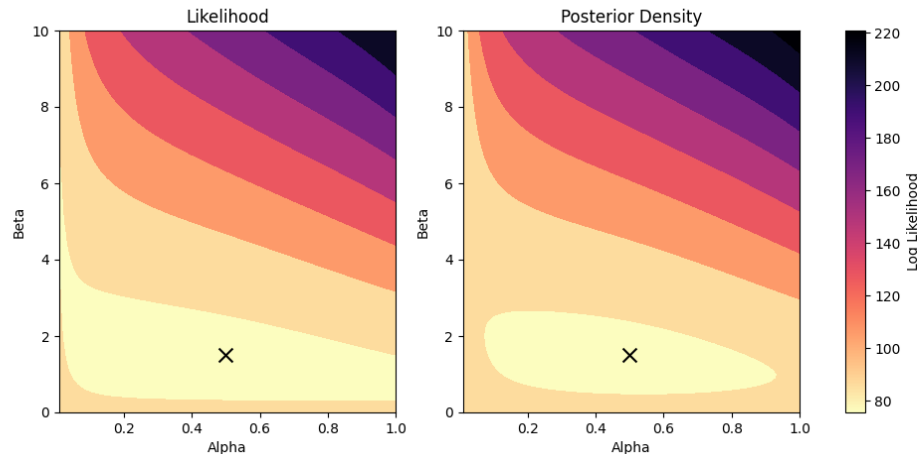
A class representing a set of parameters. It takes keyword arguments representing the parameters with their values and wraps them into a python object.

Parameters:

- ****kwargs** (dict, default: {}) – Keyword arguments representing the parameters.

Returns:

- **Parameters** – A Parameters object, where each attributes is one of the keyword arguments provided for the function wrapped by the Value class.



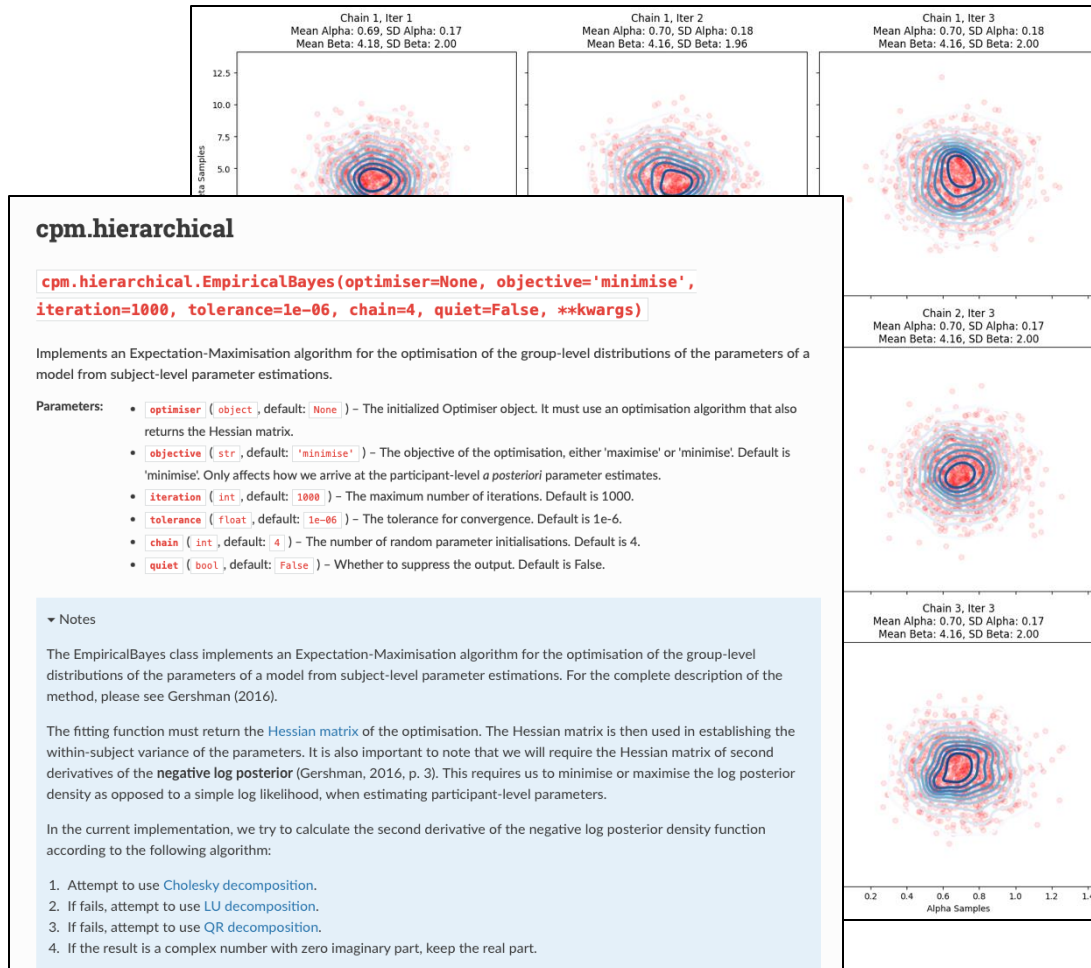
The Parameters class can also provide a prior.

```
>>> x = Parameters(  
>>>     a=Value(value=0.1, lower=0, upper=1, prior="normal", args={"mean": 0.5, "sd": 0.1}),  
>>>     b=0.5,  
>>>     weights=Value(value=[0.1, 0.2, 0.3], lower=0, upper=1, prior=None),  
>>> )
```

```
>>> x.prior(log=True)  
-6.5854290732499186
```

Hierarchical fitting: hyperparameters

- using priors to constrain parameter estimates
- estimate hyperparameters (means and standard deviations) to draw group-level inferences



cpm.hierarchical

Applications

- the toolbox implements existing models and existing computational methods as a growing number of applications
- `cpm.applications.reinforcement_learning.RLRW`
- `cpm.applications.signal_detection.EstimateMetaD`
- (in-development) `cpm.applications.reinforcement_learning.MBMF`
- (in-development) `cpm.applications.decision_making.PTSM`
- (in-development) `cpm.applications.decision_making.PTSEAM`

Other convenient features: modularity

- Functions within their family are interchangeable
- Interaction with third-party libraries

Even more convenience: documentation,
examples and troubleshooting!



EXERCISES

One and Two: Building and evaluating models

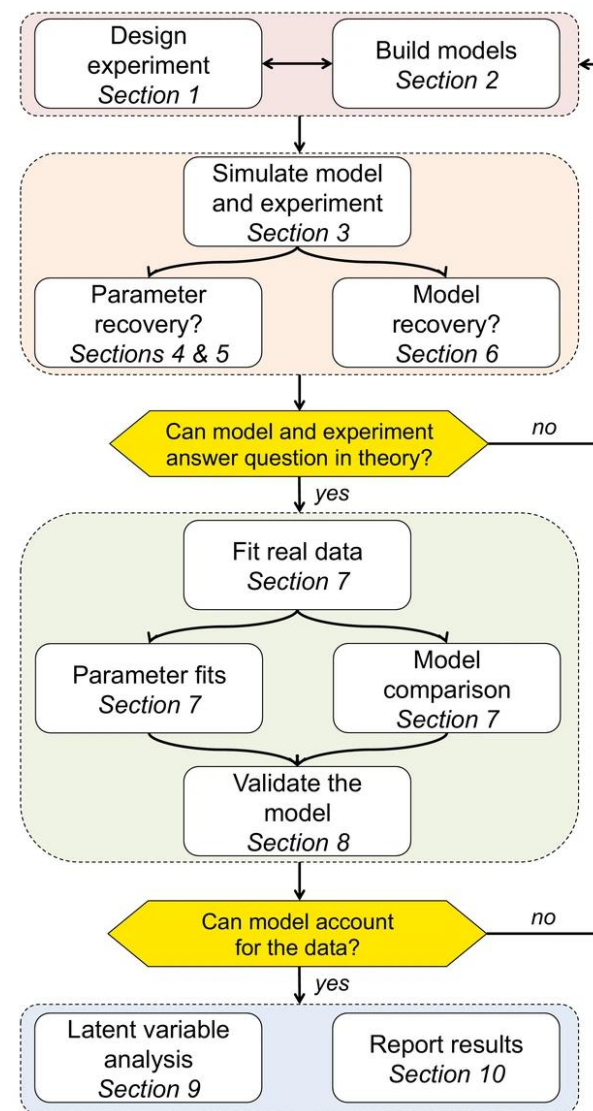
Activity Format

- Complete worksheets
 - read through the Jupyter Notebooks
 - complete each task in chronological order (they tend to build on each other)
- Self-directed activity
- Help each other when you can!

If you need help or have questions, we are here to assist you!

What do the exercises cover?

1. Build models
2. Simulate with models
3. Recover parameters
4. Recover models (optional homework)
5. Fit real data
6. Parameter fits (+ hierarchical parameter estimation)
7. Exploring parameter estimates

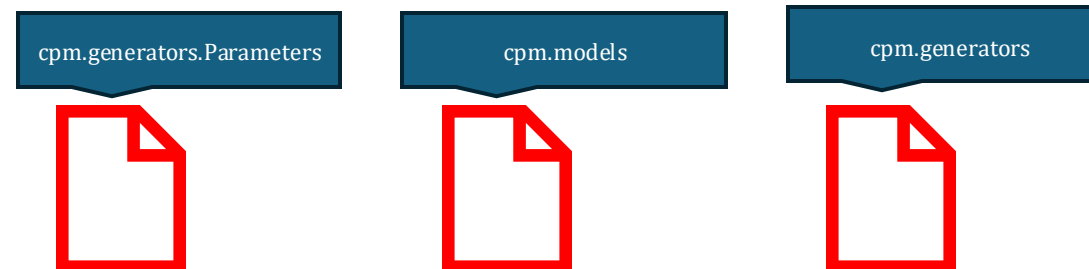


Wilson, R. C., & Collins, A. G. (2019). Ten simple rules for the computational modeling of behavioral data. *eLife*, 8, e49547. <https://doi.org/10.7554/eLife.49547>

Exercise 1. and 2.

- <https://github.com/lenarddome/cpm-workshop-2025.git>
- **exercises** folder -> [01-model-building.ipynb](#) and [02-model-parameter-recovery.ipynb](#)

Read through the explanations, and complete the exercises by filling in the missing parts of the code.





BREAK

11:00 – 11:15



EXERCISES

Three and Four: Fitting models and exploring estimated parameters

Exercise 3. and 4.

- <https://github.com/lenarddome/cpm-workshop-2025.git>
- **exercises** folder -> [03-fitting-to-data.ipynb](#) and [04-fitting-explore-model-predictions.ipynb](#)

Read through the explanations, and complete the exercises by filling in the missing parts of the code.

documentation:

<https://devcompsy.github.io/cpm/>



EXERCISES

Five (optional): model recovery

Exercise 5. (optional homework)

- Solutions are available: [05-model-recovery.ipynb](#) and [05-model-recovery.py](#)

documentation:








<https://devcompsy.github.io/cpm/>



Wrapping up

Some take-aways and complex topics

What did you do?

- ✓  Build models
(cpm.models, cpm.generators)
- ✓  Evaluate your models prior to data
(cpm.generators, cpm.optimisation)
- ✓   Evaluate model and parameter identifiability
(cpm.generators, cpm.optimisation)
- ✓  Being able to fit the model to data
(cpm.optimisation)
- ✓  Understand the principles of hierarchical modeling
(cpm.generators.Parameters, cpm.hierarchical)
- ✓  Draw inferences on the group level through estimation of hyperparameters
(cpm.hierarchical)

Workshop evaluation form

