Pcm for Python

Release v.0.9

Jörn Diedrichsen

Jul 20, 2023

CONTENTS:

The Pattern Component Modelling (PCM) toolbox is designed to analyze multivariate brain activity patterns in a Bayesian approach. The theory is laid out in Diedrichsen et al. (2017) as well as in this documentation. We provide details for model specification, model estimation, visualisation, and model comparison. The documentation also refers to the empricial examples in demos folder.

The original Matlab version of the toolbox is available at https://github.com/jdiedrichsen/pcm_toolbox. The practical examples in this documentation are available as jupyter notebooks at https://github.com/DiedrichsenLab/PcmPy/tree/master/docs/demos.

Note that the toolbox does not provide functions to extract the required data from the first-level GLM or raw data, or to run search-light or ROI analyses. We have omitted these function as they strongly depend on the analysis package used for the basic imaging analysis. Some useful tools for the extraction of multivariate data from first-level GLMs can be found in the RSA-toolbox (https://github.com/rsagroup/rsatoolbox) and of course nilearn.

CHAPTER

ONE

LICENCE AND ACKNOWLEDGEMENTS

The PCMPy toolbox is being developed by members of the Diedrichsenlab including Jörn Diedrichsen, Giacomo Ariani, Spencer Arbuckle, Eva Berlot, and Atsushi Yokoi. It is distributed under MIT License, meaning that it can be freely used and re-used, as long as proper attribution in form of acknowledgments and links (for online use) or citations (in publications) are given. When using, please cite the relevant references:

- Diedrichsen, J., Yokoi, A., & Arbuckle, S. A. (2018). Pattern component modeling: A flexible approach for understanding the representational structure of brain activity patterns. Neuroimage. 180(Pt A), 119-133.
- Diedrichsen, J., Ridgway, G., Friston, K.J., Wiestler, T., (2011). Comparing the similarity and spatial structure of neural representations: A pattern-component model. Neuroimage.

CHAPTER

TWO

DOCUMENTATION

2.1 Installation

2.1.1 Dependencies

The required dependencies to use the software are:

- python >= 3.6,
- numpy >= 1.16
- pandas >= 0.24
- matplotlib >= 1.5.1
- seaborn

If you want to run the tests, you need pytest >= 3.9 and pytest-cov for coverage reporting.

2.1.2 Installation using Pip

PcmPy is available on PyPi with:

pip install PcmPy

2.1.3 Installation for developers

You can also clone or fork the whole repository from https://github.com/diedrichsenlab/PCMPy. Place the the entire repository in a folder of your choice. Then add the folder by adding the following lines to your `.bash.profile` or other shell startup file:

```
PYTHONPATH=/DIR/PcmPy:${PYTHONPATH}
export PYTHONPATH
```

You then should be able to import the entire toolbox as:

```
import PCMPy as pcm
```

2.2 Introduction

The study of brain representations aims to illuminate the relationship between brain activity patterns and "things in the world" - be it objects, actions, or abstract concepts. Understanding the internal syntax of brain representations, and how this structure changes across different brain regions, is essential in gaining insights into the way the brain processes information.

Central to the definition of representation is the concept of decoding. A feature (i.e. a variable that describes some aspect of the "things in the world") that can be decoded from the ongoing neural activity in a region is said to be represented there. For example, a feature could be the direction of a movement, the orientation and location of a visual stimulus, or the semantic meaning of a word. Of course, if we allow the decoder to be arbitrarily complex, we would use the term representation in the most general sense. For example, using a computer vision algorithm, one may be able to identify objects based on activity in primary visual cortex. However, we may not conclude necessarily that object identity is represented in V1 - at least not explicitly. Therefore, it makes sense to restrict our definition of an explicit representation to features that can be linearly decoded by a single neuron from some population activity (Kriegeskorte & Diedrichsen, 2017, 2019).

While decoding is a popular approach when analyzing multi-variate brain activity patterns, is is not the most useful tool when we aim to make inferences about the nature of brain representations. The fact that we can decode feature X well from region A does not imply that the representation in A is well characterized by feature X - there may be many other features that better determine the activity patterns in this region.

In an Encoding model, we characterize how well we can explain the activities in a specific region using a sets of features. The activity profile of each voxel (here shown as columns in the activity data matrix), is modeled as the linear combination of a set of features (Fig. 1a). Each voxel, or more generally measurement channel, has its own set of parameters ($\mathbf{W}$) that determine the weight of each feature. This can visualized by plotting the activity profile of each voxel into the space spanned by the experimental conditions (Fig. 1b). Each dot refers to the activity profile of a channel (here a voxel), indicating how strongly the voxel is activated by each condition. Estimating the weights is equivalent to a projection of each of the activity profiles onto the feature vectors. The quality of the model can then be evaluated by determining how well unseen test data can be predicted. When estimating the weights, encoding models often use some form of regularization, which essentially imposes a prior on the feature weights. This prior is an important component of the model. It determines a predicted distribution of the activity profiles (Diedrichsen & Kriegeskorte, 2017). An encoding model that matches the real distribution of activity profiles best will show the best prediction performance.

The interpretational problem for encoding models is that for each feature set that predicts the data well, there is an infinite number of other (rotated) features sets that describe the same distribution of activity profiles (and hence predict the data) equally well (Diedrichsen, 2019). The argument may be made that to understand brain representations, we should not think about specific features that are encoded, but rather about the distribution of activity profiles. This can be justified by considering a read-out neuron that receives input from a population of neurons. From the standpoint of this neuron, it does not matter which neuron has which activity profile (as long as it can adjust input weights), and which features were chosen to describe these activity profiles - all that matters is what information can read out from the code. Thus, from this perspective it may be argued that the formulation of specific feature sets and the fitting of feature weights for each voxel are unnecessary distractions.

Therefore, pattern component modeling (PCM) abstracts from specific activity patterns. This is done by summarizing the data using a suitable summary statistic (Fig. 1a), that describes the shape of the activity profile distribution (Fig. 1c). This critical characteristic of the distribution is the covariance matrix of the activity profile distribution or - more generally - the second moment. The second moment determines how well we can linearly decode any feature from the data. If, for example, activity measured for two experimental conditions is highly correlated in all voxels, then the difference between these two conditions will be very difficult to decode. If however, the activities are uncorrelated, then decoding will be very easy. Thus, the second moment is a central statistical quantity that determines the representational content of the brain activity patterns of an area.

Similarly, a representational model is formulated in PCM not by its specific feature set, but by its predicted second moment matrix. If two feature sets have the same second moment matrix, then the two models are equivalent. Thus, PCM makes hidden equivalences between encoding models explicit. To evaluate models, PCM simply compares the

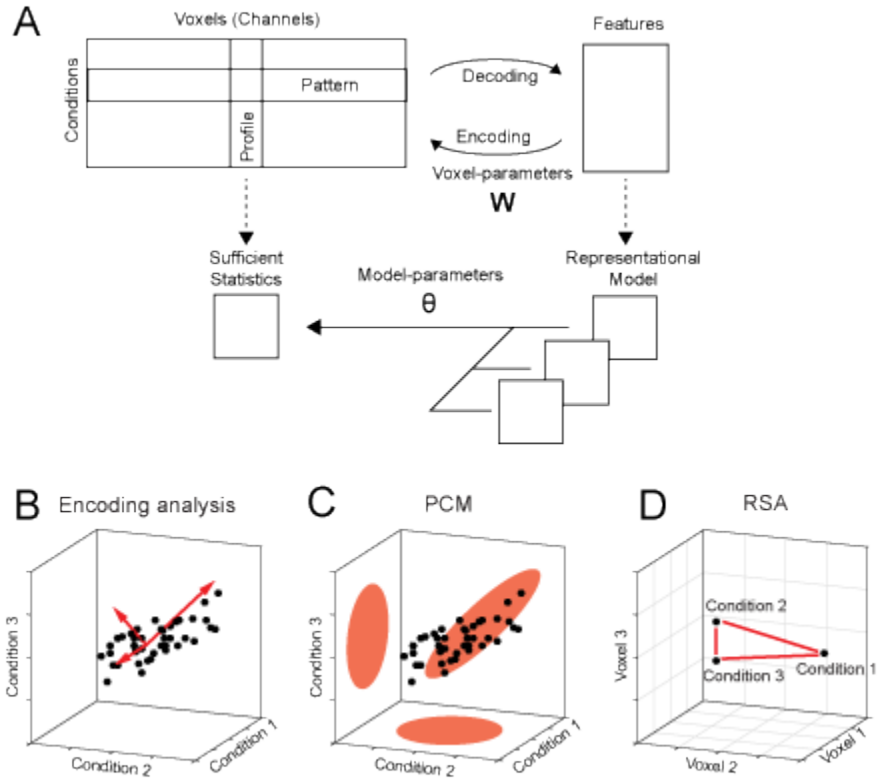


Fig. 1: **Figure 1.** *Decoding, encoding and representational models.* (**A**) *The matrix of activity data consists of rows of activity patterns for each condition or of columns of activity profiles for each voxel (or more generally, measurement channel). The data can be used to decode specific features that describe the experimental conditions (decoding). Alternatively, a set of features can be used to predict the activity data (encoding). Representational models work at the level of a sufficient statistics (the second moment) of the activity profiles. Models are formulated in this space and possibly combined and changed using higher-order model parameters.* (**B**) *Encoding analysis: The activity profiles of different voxels are points in the space of the experimental conditions. Features in encoding models are vectors that describe the overall distribution of the activity profiles.* (**C**) *PCM: The distribution of activity profiles is directly described using a multivariate normal distribution.* (**D**) *Representational similarity analysis (RSA): plots the activity patterns for different conditions in the space defined by different voxel. The distances between activity patterns serves here as the sufficient statistic, which is fully defined by the second moment matrix.*

likelihood of the data under the distribution predicted by the model. To do so, we rely on an generative model of brain activity data, which fully specifies the distribution and relationship between the random variables. Specifically, true activity profiles are assumed to have a multivariate Gaussian distribution and the noise is also assumed to be Gaussian, with known covariance structure. Having a fully-specified generative model allows us to calculate the likelihood of data under the model, averaged over all possible values of the feature weights. This results in the so-called model evidence, which can be used to compare different models directly, even if they have different numbers of features.

In summarizing the data using a sufficient statistic, PCM is closely linked to representation similarity analysis (RSA), which characterizes the second moment of the activity profiles in terms of the distances between activity patterns (Fig. 1d, also see Diedrichsen & Kriegeskorte, 2017). Thus, in many ways PCM can be considered to be intermediate approach that unifies the strength of RSA and Encoding models.

By removing the requirement to fit and cross-validate individual voxel weights, PCM enables the user to concentrate on a different kind of free parameter, namely model parameters that determine the shape of the distribution of activity profiles. From the perspective of encoding models, these would be hyper-parameters that change the form of the feature or regression matrix. For example, we can fit the distribution of activity profiles using a weighted combination of 3 different feature sets (Fig1. a). Such component models (see section Component models) are extremely useful if we hypothesize that a region cares about different groups of features (i.e.colour, size, orientation), but we do not know how strongly each feature is represented. In encoding models, this would be equivalent to providing a separate regularization factor to different parts of the feature matrix. Most encoding models, however, use a single regularization factor, making them equivalent to a fixed PCM model.

In this manual we will show how to use the PCM toolbox to estimate and compare flexible representational models. We will present the fundamentals of the generative approach taken in PCM and outline different ways in which flexible representational models with free parameters can be specified. We will then discuss methods for model fitting and for model evaluation. We will also walk in detail through three illustrative examples from our work on finger representations in primary sensory and motor cortices, also providing demo code for the examples presented in the paper (Diedrichsen, Yokoi. & Arbuckle, 2018).

2.3 Models Specification

2.3.1 Statistical Model

PCM is based on a generative model of the measured brain activity data $\mathbf{Y}$, a matrix of *N x P* activity measurements, referring to *N* time points (or trials) and *P* voxels (or channels). The data can refer to the minimally preprocessed raw activity data, or to already deconvolved activity estimates, such as those obtained as beta weights from a first-level time series model. $\mathbf{U}$ is the matrix of true activity patterns (a number of conditions x number of voxels matrix) and $\mathbf{Z}$ the design matrix. Also influencing the data are effects of no interest $\mathbf{B}$ and noise:

$$\begin{aligned}\mathbf{Y} &= \mathbf{ZU} + \mathbf{XB} + \epsilon \\ \mathbf{u}_p &\sim N(\mathbf{0}, \mathbf{G}) \\ \epsilon_p &\sim N(\mathbf{0}, \mathbf{S}\sigma^2)\end{aligned}$$

Assumption about the signal (U)

The activity profiles ($\mathbf{u}_p$ columns of $\mathbf{U}$) are considered to be a random variable. PCM models do not specify the exact activity profiles of specific voxels, but rather their probability distribution. Also, PCM is not interested in how the different activity profiles are spatially arranged. This makes sense considering that activity patterns can vary widely across different participants and do not directly impact what can be decoded from a region. For this, only the distribution of activity profiles in a region is important.

PCM assumes is that the expected mean of the activity profiles is zero. In many cases, we are not interested in how much a voxel is activated, but only how acitivity differs between conditions. In these cases, we model the mean for each voxel using the fixed effects $\mathbf{X}$.

Note that this mean pattern removal does not change in information contained in a region. In contrast, sometimes researchers also remove the mean value (Walther et al., 2016), i.e., the mean of each condition across voxels. We discourage this approach, as it would remove differences that, from the persepctive of decoding and representation, are highly meaningful.

The third assumption is that the activity profiles come from a multivariate Gaussian distribution. This is likely the most controversial assumption, but it is motivated by a few reasons: First, for fMRI data the multi-variate Gaussian is often a relatively appropriate description. Secondly, the definition causes us to focus on the mean and covariance matrix, $\mathbf{G}$, as sufficient statistics, as these completely determine the Gaussian. Thus, even if the true distribution of the activity profiles is better described by a non-Gaussian distribution, the focus on the second moment is sensible as it characterizes the linear decodability of any feature of the stimuli.

Assumptions about the Noise

We assume that the noise of each voxel is Gaussian with a temporal covariance that is known up to a constant term σ^2. Given the many additive influences of various noise sources on fMRI signals, Gaussianity of the noise is, by the central limit theorem, most likely a very reasonable assumption, commonly made in the fMRI literature. The original formulation of PCM used a model which assumed that the noise is also temporally independent and identically distributed (i.i.d.) across different trials, i.e. $\mathbf{S} = \mathbf{I}$. However, as pointed out recently (Cai et al., 2016), this assumption is often violated in non-random experimental designs with strong biasing consequences for estimates of the covariance matrix. If this is violated, we can either assume that we have a valid estimate of the true covariance structure of the noise ($\mathbf{S}$), or we can model different parts of the noise structure (see *Noise Models*).

PCM also assumes that different voxels are independent from each other. If we use fMRI data, this assumption would be clearly violated, given the strong spatial correlation of noise processes in fMRI. To reduce these dependencies we typically uses spatially pre-whitened data, which is divided by a estimate of the spatial covariance matrix (Walther et al., 2016). Recent result from our lab show that this approach is sufficient to obtain correct marginal likelihoods.

Marginal likelihood

When we fit a PCM model, we are not trying to estimate specific values of the the estimates of the true activity patterns $\mathbf{U}$. This is a difference to encoding approaches, in which we would estimate the values of $\mathbf{U}$ by estimating the feature weights $\mathbf{W}$. Rather, we want to assess how likely the data is under any possible value of $\mathbf{U}$, as specified by the prior distribution. Thus we wish to calculate the marginal likelihood

$$p\left(\mathbf{Y}|\theta\right) = \int p\left(\mathbf{Y}|\mathbf{U},\theta\right) p\left(\mathbf{U}|\theta\right) d\mathbf{U}.$$

This is the likelihood that is maximized in PCM in respect to the model parameters θ. For more details, see mathematical and algorithmic details.

2.3.2 Encoding- vs. RSA-style models

The main intellectual work when using PCM is to build the appropriate models. There are basically two complementary approaches or philosophies when it comes to specifying a representational models. Which way you feel more comfortable with will likely depend on whether you are already familiar with Encoding models or RSA. Ultimately, most problems can be formulated in both ways, and the results will be identical. Nonetheless, it is useful to become familiar with both styles of model building, as they can also be combined to find the most intuitive and computationally efficient way of writing a particular model.

Example

An empirical example for how to construct the same model as either an Encoding- or RSA-style PCM model comes from Yokoi et al. (2018). In this experiment, participants learned six motor sequences, each different permutations of pressing the finger 1, 3, and 5 (see Fig 2a). We would like to model the activity pattern in terms of two model components: In the first component, each finger contributes a specific pattern, and the pattern of the first finger has a particularly strong weight. In the second component, each transitions between 2 subsequent fingers contributes a unique pattern.

Encoding-style models

When constructing an encoding-style model, the model components are formulated as sets of features, which are encoded into the design matrix. In our example, the first feature set (first finger) has 3 columns with variables that indicate whether this first finger was digit 1, 3, or 5 (because each finger occurs exactly once in each sequence, we can ignore the subsequent presses). The second feature set has 6 features, indicating which ones of the 6 possible transitions between fingers were present in the sequence. Therefore, the design matrix **Z** has 9 columns (Fig. 2b). The encoding-style model, however, is not complete without a prior on the underlying activity patterns **U**, i.e. the feature weights. As implicit in the use of ridge regression, we assume here that all features within a feature set are independent and equally strongly encoded. Therefore, the second moment matrix for the first model component is an identity matrix of 3x3 and for the second component an identity matrix of size 6x6. Each component then is weighted by the relative weight of the component, $exp(\theta_i)$. The overall second moment matrix **G** then is the sum of the two weighted model component matrices. This model would be equivalent to an encoding model where each of the features sets has its own ridge coefficient. PCM will automatically find the optimal value of the two ridge coefficients (the importance of each feature set). If you want to use PCM to tune the regularization paramters for Tikhonov regularization, the regression module provides a simplified interface to do so (see *Regularized regression*)

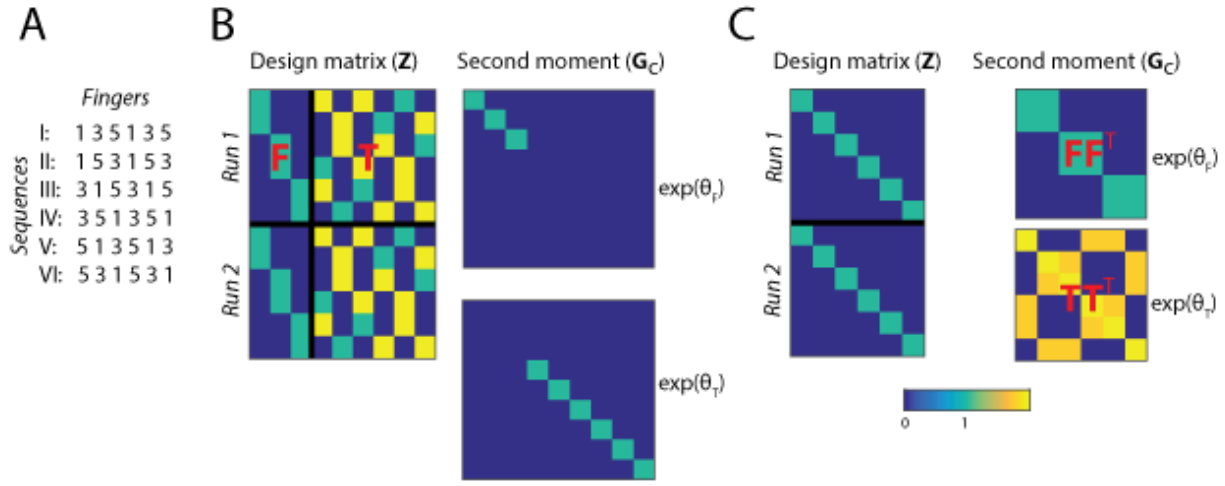


Fig. 2: **Figure2.** *(A) Set of six multi-finger sequences used in the task. (B) Encoding-style model construction.The condition matrix Z, here shown for 12 trials of 2 x 6 sequences, contains all features. Featureset* **F** *contains indicators for first fingers in each sequence, and featureset* **T** *contains the finger transitions for each sequence. The second moment matrices for the two feature sets are diagonal matrices, indicating which feature is taken into account. Each feature set is multiplied by an overall importance of the featureset (analogous to the ridge coefficient for the feature set) (C) RSA-style model construction. The design matrix indicates which of the 6 sequences was executed. The second momemnt matrix determines the hypothesized covariances between the 6 sequence patterns. In both cases, the overall second moment matrix* **G** *is the weighted sum of the two component matrices*

RSA-style models

In RSA, models are specified in terms of the predicted similarity or dissimilarity between discrete experimental conditions. Therefore, when constructing our model in RSA-style, the design matrix $\mathbf{Z}$ simply indicates which trial belonged to which sequence. The core of the model is specified in the second moment matrix, which specifies the covariances between conditions, and hence both Euclidean and correlation distances (Fig. 2C). For example, the first component predicts that sequence I and II, which both start with digit 1, share a high covariance. The predicted covariances can be calculated from an encoding-stype model by taking the inner product of the feature sets $\mathbf{F}\mathbf{F}^T$.

Which approach to use?

The Models depicted in Fig. 2B and 2C are identical. So when should we prefer one approach over the other? Some of this is up to personal taste. However, some experiments have no discrete conditions. For example, each trial may be also characterized by an action or stimulus property that varied in a continuous fashion. For these experiments, the best way is to formulate the model in an encoding style. Even if there are discrete conditions, the conditions may differ across subjects. Because the group fitting routines (see below) allow for subject-specific design matrices, but not for subject-specific second-moment matrices, encoding-style models are the way to go. In other situations, for example experiments with fewer discrete conditions and many feature sets, the RSA-style formulation can be more straightforward and faster.

Finally, the two approaches can be combined to achieve the most natural way of expressing models. For example in our example, we used the design matrix from the first finger model Fig. 2B, combined with a second moment derived from the natural statistics to capture the known covariance structure of activity patterns associated with single finger movements Ejaz et al. (2015).

2.3.3 Model types

Independently of whether you choose an Encoding- or RSA-style approach to building your model, the PCM toolbox distinguishes between a number of different model types, each of which has an own model class.

Fixed models

In fixed models, the second moment matrix $\mathbf{G}$ is exactly predicted by the model. The most common example is the Null model $\mathbf{G} = \mathbf{0}$. This is equivalent to assuming that there is no difference between any of the activity patterns. The Null-model is useful if we want to test whether there are any differences between experimental conditions. An alternative Null model would be $\mathbf{G} = \mathbf{I}$, i.e. to assume that all patterns are uncorrelated equally far away from each other.

Fixed models also occur when the representational structure can be predicted from some independent data. An example for this is shown in the following example, where we predict the structure of finger representations directly from the correlational structure of finger movements in every-day life (Ejaz et al., 2015). Importantly, fixed models only predict the the second moment matrix up to a proportional constant. The width of the distribution will vary with the overall scale or signal-to-noise-level. Thus, when evaluating fixed models we usually allow the predicted second moment matrix to be scaled by an arbitrary positive constant (see *Model Fitting*).

Example

An empirical example to for a fixed representational model comes from Ejaz et al (2015). Here the representational structure of 5 finger movements was compared to the representational structure predicted by the way the muscles are activated during finger movements (Muscle model), or by the covariance structure of natural movements of the 5 fingers. That is the predicted second moment matrix is derived from data completely independent of our imaging data.

Models are a specific class, inherited from the class `Model`. To define a fixed model, we simple need to load the predicted second moment matrix and define a model structure as follows (see *Application examples*):

```
M1 = pcm.model.FixedModel('null',np.eye(5))     # Makes a Null model
M2 = pcm.model.FixedModel('muscle',modelM[0])  # Makes the muscle model
M3 = pcm.model.FixedModel('natural',modelM[1]) # Makes the natural stats model
M = [M1, M2, M3] # Join the models for fitting in list
```

When evaluating the likelihood of a data set under the prediction, the pcm toolbox still needs to estimate the scaling factor and the noise variance, so even in the case of fixed models, an iterative maximization of the likelihood is required (see below).

Component models

A more flexible model is to express the second moment matrix as a linear combination of different components. For example, the representational structure of activity patterns in the human object recognition system in inferior temporal cortex can be compared to the response of a convolutional neural network that is shown the same stimuli (Khaligh-Razavi & Kriegeskorte, 2014). Each layer of the network predicts a specific structure of the second moment matrix and therefore constitutes a fixed model. However, the representational structure may be best described by a mixture of multiple layers. In this case, the overall predicted second moment matrix is a linear sum of the weighted components matrices:

$$\mathbf{G} = \sum_h \exp(\theta_h)\mathbf{G}_h$$

The weights for each component need to be positive - allowing negative weights would not guarantee that the overall second moment matrix would be positive definite. Therefore we use the exponential of the weighing parameter here, such that we can use unconstrained optimization to estimate the parameters.

For fast optimization of the likelihood, we require the derivate of the second moment matrix in respect to each of the parameters. Thus derivative can then be used to calculate the derivative of the log-likelihood in respect to the parameters (see section 4.3. Derivative of the log-likelihood). In the case of linear component models this is easy to obtain.

$$\frac{\partial \mathbf{G}}{\partial \theta_h} = \exp(\theta_h)\mathbf{G}_h$$

Example

In the example *Finger demo*, we have two fixed models, the Muscle and the natural statistics model. One question that arises in the paper is whether the real observed structure is better fit my a linear combination of the natural statistics and the muscle activity structure. So we can define a third model, which allows any arbitrary mixture between the two type.

```
MC = pcm.ComponentModel('muscle+nat',[modelM[0],modelM[1]])
```

Feature models

A representational model can be also formulated in terms of the features that are thought to be encoded in the voxels. Features are hypothetical tuning functions, i.e.models of what activation profiles of single neurons could look like. Examples of features would be Gabor elements for lower-level vision models, elements with cosine tuning functions for different movement directions for models of motor areas, and semantic features for association areas. The actual activity profiles of each voxel are a weighted combination of the feature matrix $\mathbf{u}_p = \mathbf{M}\mathbf{w}_p$. The predicted second moment matrix of the activity profiles is then $\mathbf{G} = \mathbf{M}\mathbf{M}^T$, assuming that all features are equally strongly and independently encoded, i.e. $E\left(\mathbf{w}_p\mathbf{w}_p^T\right) = \mathbf{I}$. A feature model can now be flexibly parametrized by expressing the feature matrix as a weighted sum of linear components.

$$\mathbf{M} = \sum_h \theta_h \mathbf{M}_h$$

Each parameter θ_h determines how strong the corresponding set of features is represented across the population of voxels. Note that this parameter is different from the actual feature weights $\mathbf{W}$. Under this model, the second moment matrix becomes

$$\mathbf{G} = \mathbf{U}\mathbf{U}^T/P = \frac{1}{P}\sum_h \theta_h^2 \mathbf{M}_h\mathbf{M}_h^T + \sum_i\sum_j \theta_i\theta_j \mathbf{M}_i\mathbf{M}_j^T.$$

From the last expression we can see that, if features that belong to different components are independent of each other, i.e. $\mathbf{M}_i\mathbf{M}_j = \mathbf{0}$, then a feature model is equivalent to a component model with $\mathbf{G}_h = \mathbf{M}_h\mathbf{M}_h^T$. The only technical difference is that we use the square of the parameter θ_h, rather than its exponential, to enforce non-negativity. Thus, component models assume that the different features underlying each component are encoded independently in the population of voxels - i.e.knowing something about the tuning to feature of component A does not tell you anything about the tuning to a feature of component B. If this cannot be assumed, then the representational model is better formulated as a feature model.

By the product rule for matrix derivatives, we have

$$\frac{\partial \mathbf{G}}{\partial \theta_h} = \mathbf{M}_h\mathbf{M}(\theta)^T + \mathbf{M}\left(\theta\right)\mathbf{M}_h^T$$

Correlation model

The correlation model class is designed model correlation between specific sets of activity patterns. This problem often occurs in neuroimaging studies: For example, we may have 5 actions that are measured under two conditions (for example observation and execution), and we want to know to what degree the activtiy patterns of obseving an action related to the pattern observed when executing the same action.

Fixed correlation models: We can use a series of models that test the likelihood of the data under a fixed correlations between -1 and 1. This approach allows us to determine how much evidence we have for one specific correlation over the other. Even though the correlation is fixed for these models, the variance structure within each of the conditions is flexibly estimated. This is done using a compent model within each condition.

$$\mathbf{G}^{(1)} = \sum_h \exp(\theta_h^{(1)})\mathbf{G}_h$$
$$\mathbf{G}^{(2)} = \sum_h \exp(\theta_h^{(2)})\mathbf{G}_h$$

Usually the $mathbf{G}_{h}$ is the identity matrix (all items are equally strongly represented, or a matrix that allows individual scaling of the variances for each item. Of course you can also model any between-item covariance. The overall model is nonlinear, as the two components interact in the part of the $\mathbf{G}$ matrix that indicates the covariance

between the patterns of the two conditions ($\mathbf{C}$). Given a constant correlation *r*, the overall second moment matrix is calculated as:

$$\mathbf{G} = \begin{bmatrix} \mathbf{G}^{(1)} & r\mathbf{C} \\ r\mathbf{C}^T & \mathbf{G}^{(2)} \end{bmatrix}$$
$$\mathbf{C}_{i,j} = \sqrt{\mathbf{G}^{(1)}_{i,j}\mathbf{G}^{(2)}_{i,j}}$$

If the parameter *within_cov* is set to true, the model will also add with-condition covariance, which are not part of covariance. So the correlation that we modelling is the correlation between the pattern component related to the individual items, after the pattern component related to the overall condition (ie. observe vs. execute) has been removed.

The derivatives of that part of the matrix in respect to the parameters $\theta^{(1)}_h$ then becomes

$$\frac{\partial \mathbf{C}_{i,j}}{\partial \theta^{(1)}_h} = \frac{r}{2\mathbf{C}_{i,j}}\mathbf{G}^{(2)}_{i,j}\frac{\partial \mathbf{G}^{(1)}_{i,j}}{\partial \theta^{(1)}_h}$$

These derivatives are automatically calculated in the predict function. From the log-likelihoods for each model, we can then obtain an approximation for the posterior distribution.

Flexible correlation model: We also use a flexible correlation model, which has an additional model parameter for the correlation. To avoid bounds on the correlation, this parameter is the inverse Fisher-z transformation of the correlation, which can take values of $[-\infty, \infty]$.

$$\theta = \frac{1}{2}log\left(\frac{1+\theta}{1-\theta}\right)$$
$$r = \frac{exp(2\theta)-1}{exp(2\theta)+1}$$

The derivative of r in respect to θ can be derived using the product rule:

$$\frac{\partial r}{\partial \theta} = \frac{2exp(2\theta)}{exp(2\theta)+1} - \frac{(exp(2\theta)-1)\,(2exp(2\theta))}{(exp(2\theta)+1)^2} = \frac{4exp(2\theta)}{(exp(2\theta)+1)^2}$$

Example

For a full example, please see the Correlation models.

Free models

The most flexible representational model is the free model, in which the predicted second moment matrix is unconstrained. Thus, when we estimate this model, we would simply derive the maximum-likelihood estimate of the second-moment matrix. This model is mainly useful if we want to obtain an estimate of the maximum likelihood that could be achieved with a fully flexible model, i.e the noise ceiling (Nili et al. 20).

In estimating an unconstrained $\mathbf{G}$, it is important to ensure that the estimate will still be a positive definite matrix. For this purpose, we express the second moment as the square of an upper-triangular matrix, $\mathbf{G} = \mathbf{A}\mathbf{A}^T$ (Diedrichsen et al., 2011; Cai et al., 2016). The parameters are then simply all the upper-triangular entries of $\mathbf{A}$.

Example

To set up a free model, simple create a new model of type `FreeModel`.

```
M5 = pcm.model.FreeModel('ceil',n_cond)
```

If the number of conditions is very large, the crossvalidated estimation of the noise ceiling model can get rather slow. For a quick and approximate noise ceiling, you can also set use an unbiased estimate of the second moment matrix from `pcm.util.est_G_crossval` to determine the parameters - basically the starting values of the complete model. This will lead to slightly lower noise ceilings as compared to the full optimization, but large improvements in speed.

Custom model

In some cases, the hypotheses cannot be expressed by a model of the type mentioned above. Therefore, the PCM toolbox allows the user to define their own custom model. In general, the predicted second moment matrix is a non-linear (matrix valued) function of some parameters, $\mathbf{G} = F(\theta)$. One example is a representational model in which the width of the tuning curve (or the width of the population receptive field) is a free parameter. Such parameters would influence the features, and hence also the second-moment matrix in a non-linear way. Computationally, such non-linear models are not much more difficult to estimate than component or feature models, assuming that one can analytically derive the matrix derivatives $\partial\mathbf{G}/\partial\theta_h$.

To define a custom model, the user needs to define a new Model class, inherited from the abstract class `pcm.model.Model`. The main thing is to define the `predict` function, which takes the parameters as an input and returns $\mathbf{G}$ the partial derivatives of $\mathbf{G}$ in respect to each of these parameters. The derivates are returned as a (HxKxK) tensor, where H is the number of parameters.

```
class CustomModel(Model):
# Constructor of the class
def __init__(self,name,...):
    Model.__init__(self,name)
    ...

# Prediction function
def predict(self,theta):
    G = .... # Calculate second momement matrix
    dG_dTheta = # Calculate derivative second momement matrix
    return (G,dG_dTheta)

#  Intiialization function
def set_theta0(self,G_hat):
    """
    Sets theta0 based on the crossvalidated second-moment

    Parameters:
        G_hat (numpy.ndarray)
            Crossvalidated estimate of G
    """
    # The function can use G_hat to get good starting values,
    # or just start at fixed values
    self.theta0 = ....
```

Note that the predict function is repeatedly called by the optimization routine and needs to execute fast. That is, any computation that does not depend on the current value of θ should be performed outside the function and stored in the object.

2.3.4 Noise Models

Other than RSA and Encoding models, PCM also requires a explict model of the noise.In general, noise is assumed to come from a multivariate normal distribution with covariance matrix $\mathbf{S}\sigma^2$. In general, we also assume that the noise is independent across imaging runs (or partitions), making $\mathbf{S}$ a block-diagnonal matrix. But what do we assume about the within-run covariance?

Independent Noise: If the data comes from regression estimates from a first-level model, and if the design of the experiment is balanced, then it is usually also permissible to make the assumption that the noise is independent within each imaging run $\mathbf{S} = \mathbf{I}$. The raw regression coefficients from a single imaging run, however, are positively correlated with each other. So on solution is to remove the block-effect using a *fixed_effect = 'block'* during fitting.

Block Effect Plus Independent Noise: We can also estimate the amount of within-block correlation from the data, rather than remove it. This is especially important for models where the contrast of condition against rest is important. The *BlockPlusIndepNoise* model has two parameters - one for the shared within block covariance, one for the variance for each item. Do not use this if you removed the block effect as a fixed effect.

Custom Model: Assuming equal correlations of the activation estimates within a run is only a rough approximation to the real co-varince structure. A better estimate can be obtained by using an estimate derived from the design matrix and the estimated temporal autocorrelation of the raw signal. As pointed out recently (Cai et al.), the particular design can have substantial influence on the estimation of the second moment matrix. This is especially evident in cases where the design is such that the trial sequence is not random, but has an invariant structure (where trials of one condition are often to follow trials of another specific condition). The accuracy of our approximation hinges critically on the quality of our estimate of the temporal auto-covariance structure of the true noise. Note that it has been recently demonstrated that especially for high sampling rates, a simple autoregressive model of the noise is insufficient. In all optimisation routine, a specific noise covariance structure can be specified by passing the correct noise model to the fitting routine.

2.4 Model Fitting

Models can be either fit to individual or to group data. For group fits, some or all of the model parameters are shared across the group, while the noise and scale parameters are still fitted individually to each subject. To compare models of different complexity, we have implemented two types of crossvalidation, either within individuals across partitions, or across individuals

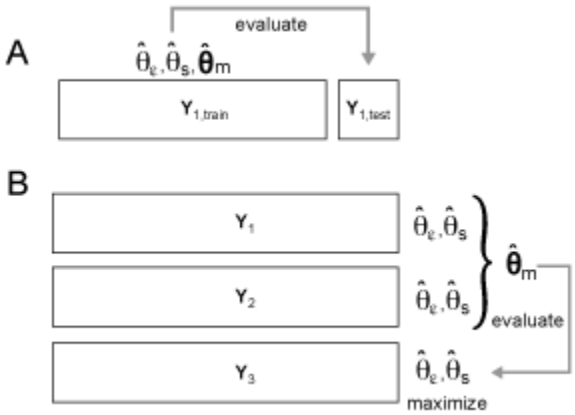


Fig. 3: **Model crossvalidation schemes.** (**A**) *Within-subject crossvalidation where the model is fit on N-1 partitions and then evaluated on the left-out partition N.* (**B**) *Group crossvalidation were the model is fit to N-1 subjects and then evaluated on a left-out subject. For group crossvalidation, individual scaling and noise parameters are fit to each subject to allow for different signal-to-noise levels.*

2.4.1 Fitting to individual data sets

Models can be fitted to each data set individually, using the function `fit_model_individ`. Individual fitting makes sense for models with a single component (fixed models), which can be evaluated without crossvalidation.

```
Y = [Dataset1, Dataset2, Dataset3, ...]
M = [Model1, Model2, ...]
T, theta = pcm.inference.fit_model_individ(Y, M, ...)
```

A number of input options can be specified as well:

- `fixed_effect`: Determines what fixed effects are included in the model. You can either pass a full design matrix, or `None`. For convenience `fixed_effect` can also be set to `block` in which case it will model the mean of each imaging run (partition). This is especially important for fMRI data, where activation estimates within a partition (imaging run) are usually correlated, as they are measured relative to the same baseline.
- `fit_scale`: Determines whether a scale parameter is fit for each subject, which determines the amount of signal variance. This should be done at least for fixed models. To ensure that the scale parameter is positive, the routine fits the log of the scale paramter, such that the resulting signal has a covariance matrix of $\mathbf{G}(\boldsymbol{\theta_m} exp(\theta_s)$. For component (and other) models that usually also model the signal strength, the additional scale parameter introduces some redundancy. For this reason we impose a log-normal prior forcing the scale parameter to be close to 1. The variance of this prior can be adjusted with through the option `scale_prior` and is set to 1000 by default.
- `noise_cov`: Here you can optionally specify a covariance structure of the noise. For fMRI data, a useful approach is to use the estimate from the first-level model, which is related to the structure of the design $(\mathbf{X}^T\mathbf{X})^{-1}$. You can also set this variable to `block`, in which case the ref{Noisemodel} will be set to `BlockPlusIndepNoise`.

The output `T` is a pandas data frame that is hierarchically organised.

```
T.likelihood['Model1'] # Log-likelihood of each data set under model 1
T.likelihood['Model2'] # Log-likelihood of each data set under model 2
...
T.noise['Model1']      # Noise parameter value
...
T.scale['Model1']      # Scale parameter values (exp(theta_scale))
...
T.iterations['Model1'] # Number of iterations until convergence
...
```

The output `theta` is a list of np-arrays, which contain the `M.n_param` model parameters, the log-scale and log-noise parameter for each data set.

The output can be used to compare the likelihoods between different models. Alternatively you can inspect the individual fits by looking at the parameters (theta). The predicted second moment matrix for any model can be obtained by

2.4.2 Fitting to individual data sets with cross-validation across partitions

Crossvalidation within subject is the standard for encoding models and can also be applied to PCM-models.

```
T, theta = pcm.inference.fit_model_individ(Y, M, ...)
```

2.4.3 Fitting to group data sets

The function `fit_model_group` fits a model to a group of subjects. By default, all parameters that change the **G** matrix, that is `theta[0:M.n_param]` are shared across all subjects. To account for the individual signal-to-noise level, by default a separate signal strength and noise parameter(s) are fitted for each subject. For each individual subject, the predicted covariance matrix of the data is:

$$\mathbf{V_i} = \theta_s \mathbf{Z}\mathbf{G}(\theta_\mathbf{m})\mathbf{Z}^\mathbf{T} + \mathbf{S}(\theta_\epsilon)$$

To finely control, which parameters are fit commonly to the group and which ones are fit individually, one can set the boolean vector `M[m].common_param`, indicating which parameters are fit to the entire group. The output *theta* for each model contains now a single vector of the common model parameters, followed by the data-set specific parameters: possibly the non-common model parameters, and then scale and noise parameters.

```
Y = [Dataset1, Dataset2, Dataset3, ...]
M = [Model1, Model2, ...]
T, theta = pcm.inference.fit_model_group(Y, M, ...)
```

2.4.4 Fitting to group data sets with cross-validation across participants

PCM allows also between-subject crossvalidation (see panel b). The common model parameters that determine the representational structure are fitted to all the subjects together, using separate noise and scale parameters for each subject. Then the model is evaluated on the left-out subjects, after maximizing scale and noise parameters (and possibly non-common model parameters). The Function `fit_model_group_crossval` implements these steps.

The demo `demo_finger.ipynb` provides a full example how to use group crossvalidation to compare different models. Three models are being tested: A muscle model, a usage model (both a fixed models) and a combination model, in which both muscle and usage can be combined in any combination. We also fit the noise-ceiling model, and a null-model. Because the combination model has one more parameter than each single model, crossvalidation is necessary for inferential tests. Note that for the simple models, the simple group fit and the cross-validated group fit are identical, as in both cases only a scale and noise parameter are optimized for each subject.

```
# Build models from the second momement matrices
M = []
M.append(pcm.FixedModel('null',np.eye(5)))
M.append(pcm.FixedModel('muscle',modelM[0]))
M.append(pcm.FixedModel('natural',modelM[1]))
M.append(pcm.ComponentModel('muscle+nat',[modelM[0],modelM[1]]))
M.append(pcm.FreeModel('ceil',5)) # Noise ceiling model

# Fit the model in to the full group, using a individual scaling parameter for each
T_gr, theta = pcm.inference.fit_model_group(Y, M, fit_scale=True)

# crossvalidated likelihood is the same as the group fit for all
# except the component and noise ceiling model
T_cv, theta_cv = pcm.inference.fit_model_group_crossval(Y, M, fit_scale=True)
```

(continues on next page)

(continued from previous page)

```
# Make a plot, using the group fit as upper, and the crossvalidated fit as a the lower␣
↪noise ceiling
ax = pcm.vis.model_plot(T_cv.likelihood,null_model = 'null',noise_ceiling= 'ceil',upper_
↪ceiling = T_gr.likelihood['ceil'])
```

2.4.5 Likelihood and Optimization

Under the hood, the main work in PCM is accomplished by the routines `likelihood_individ`, and `likelihood_group` (see *Inference*), which return the **negative log-liklihood** of the data under the model, as well as the first (and optionally) the second derivative. This enables PCM to use standard optimization routines, such a `scipy.optimize.minimize`. For many models, a Newton-Raphson algorithm, implemented in `pcm.optimize.newton` provides a fast and stable solution. A custom algorithm for models can be chosen by setting `M.fit` to be either a string with a algorithm name that is implemented in PCM, or a function that returns the fitted parameters. (**TO BE IMPLEMENTED**).

2.5 Visualisation

2.5.1 Second Moment matrices

One important way to visualize both the data and the model prediction is to plot the second moment matrix as a colormap, for example using the matplotlib command `plt.imshow`. The predicted second moment matrix for a fitted model can be obtained using `my_model.predict(theta)`. For the data we can get a cross-validated estimate obtained using the function *`util.est_G_crossval()`*. Note that if you removed the block-effect using the runEffect option 'fixed' then you need to also remove it from the data to have a fair comparison.

Note also that you can transform a second moment matrix into a representational dissimilarity matrix (RDM) using the following equivalence (see Diedrichsen & Kriegeskorte, 2016):

The only difference is that the RDM does not contain information about the baseline.

2.5.2 Multidimensional scaling

Another important way of visualizing the second moment matrix is Multi-dimensional scaling (MDS), an important technique in representational similarity analysis. When we look at the second moment of a population code, the natural way of performing this is classical multidimensional scaling. This technique plots the different conditions in a space defined by the first few eigenvectors of the second moment matrix - where each eigenvector is weighted by the $sqrt(lambda)$.

Importantly, MDS provides only one of the many possible 2- or 3-dimensional views of the high-dimensional representational structure. That means that one should never make inferences from this reduced view. It is recommended to look at as many different views of the representational structure as possible to obtain a unbiased impression. For high dimensional space, you surely will find *one* view that shows exactly what you want to show. There are a number of different statistical visualisation techniques that can be useful here, including the 'Grand tour' which provides a movie that randomly moves through different high-dimensional rotations.

Classical multidimensional scaling from the matlab version still needs to be implemented in Python.

2.5.3 Plotting model evidence

Another approach to visualize model results is to plot the model evidence (i.e. marginal likelihoods). The marginal likelihoods are returned from the modeling routines in arbitrary units, and are thus better understood after normalizing to a null model at the very least. The lower normalization bound can be a null model, and upper bound is often a noise ceiling. This technique simply plots scaled likelihoods for each model fit.

See *Application examples* for a practical example for this.

2.6 Inference

2.6.1 Inference on model parameters

First we may make inferences based on the parameters of a single fitted model. The parameter may be the weight of a specific component or another metric derived from the second moment matrix. For example, the estimated correlation coefficient between condition 1 and 2 would be $r_{1,2} = \mathbf{G}_{1,2}/\sqrt{\mathbf{G}_{1,1}\mathbf{G}_{2,2}}$. We may want to test whether the correlation between the patterns is larger than zero, or whether a parameter differs between two different subject groups, two different regions, or whether they change with experimental treatments.

The simplest way of testing parameters would be to use the point estimates from the model fit from each subject and apply frequentist statistics to test different hypotheses, for example using a t- or F-test. Alternatively, one can obtain estimates of the posterior distribution of the parameters using MCMC sampling [@RN3567] or Laplace approximation [@RN3255]. This allows the application of Bayesian inference, such as the report of credibility intervals.

One important limitation to keep in mind is that parameter estimates from PCM are not unbiased in small samples. This is caused because estimates of $\mathbf{G}$ are constrained to be positive definite. This means that the variance of each feature must be larger or equal to zero. Thus, if we want to determine whether a single activity pattern is different from baseline activity, we cannot simply test our variance estimate (i.e. elements of $\mathbf{G}$) against zero - they trivially will always be larger, even if the true variance is zero. Similarly, another important statistic that measures the pattern separability or classifiability of two activity patterns, is the Euclidean distance, which can be calculated from the second moment matrix as $d = \mathbf{G}_{1,1} + \mathbf{G}_{2,2} - 2\mathbf{G}_{1,2}$. Again, given that our estimate of $\mathbf{G}$ is positive definite, any distance estimate is constrained to be positive. To determine whether two activity patterns are reliably different, we cannot simply test these distances against zero, as the test will be trivially larger than zero. A better solution for inferences from individual parameter estimates is therefore to use a cross-validated estimate of the second moment matrix and the associated distances [@RN3565][@RN3543]. In this case the expected value of the distances will be zero, if the true value is zero. As a consequence, variance and distance estimates can become negative. These techniques, however, take us out of the domain of PCM and into the domain of representational similarity analysis [@RN2697][@RN3672].

2.6.2 Inference on model evidence

As an alternative to parameter-based inference, we can fit multiple models and compare them according to their model evidence; the likelihood of the data given the models (integrated over all parameters). In encoding models, the weights $\mathbf{W}$ are directly fitted to the data, and hence it is important to use cross-validation to compare models with different numbers of features. The marginal likelihood already integrates all over all likely values of $\mathbf{U}$, and hence $\mathbf{W}$, thereby removing the bulk of free parameters. Thus, in practice the marginal likelihood will be already close to the true model evidence.

Our marginal likelihood, however, still depends on the free parameters $\boldsymbol{\theta}$. So, when comparing models, we need to still account for the risk of overfitting the model to the data. For fixed models, there are only two free parameters: one relating to the strength of the noise (θ_ϵ) and one relating to the strength of the signal (θ_s). This compares very favorably to the vast number of free parameters one would have in an encoding model, which is the size of $\mathbf{W}$, the number of features x number of voxels. However, even the fewer model parameters still need to be accounted for. We consider here four ways of doing so.

The first option is to use empirical Bayes or Type-II maximal likelihood. This means that we simply replace the unknown parameters with the point estimates that maximize the marginal likelihood. This is in general a feasible strategy if the number of free parameters is low and all models have the same numbers of free parameters, which is for example the case when we are comparing different fixed models. The two free parameters here determine the signal-to-noise ratio. For models with different numbers of parameters we can penalize the likelihood by $\frac{1}{2} d_\theta \log(n)$, yielding the Bayes information criterion (BIC) as the approximation to model evidence.

As an alternative option, we can use cross-validation within the individual (hyperref[fig2]{Fig. 2a}) to prevent overfitting for more complex flexible models, as is also currently common practice for encoding models [@RN3096]. Taking one imaging run of the data as test set, we can fit the parameters to data from the remaining runs. We then evaluate the likelihood of the left-out run under the distribution specified by the estimated parameters. By using each imaging run as a test set in turn, and adding the log-likelihoods (assuming independence across runs), we thus can obtain an approximation to the model evidence. Note, however, that for a single (fixed) encoding model, cross-validation is not necessary under PCM, as the activation parameters for each voxel ($\mathbf{W}$ or $\mathbf{U}$) are integrated out in the likelihood. Therefore, it can be handled with the first option we described above.

For the third option, if we want to test the hypothesis that the representational structure in the same region is similar across subjects, we can perform cross-validation across participants (hyperref[fig2]{Fig. 2b}). We can estimate the parameters that determine the representational structure using the data from all but one participant and then evaluate the likelihood of data from the left-out subject under this distribution. When performing cross-validation within individuals, a flexible model can fit the representational structure of individual subjects in different ways, making the results hard to interpret. When using the group cross-validation strategy, the model can only fit a structure that is common across participants. Different from encoding models, representational models can be generalized across participants, as we do not fit the actual activity patterns, but rather the representational structure. In a sense, this method is performing "hyper alignment" [@RN3572] without explicitly calculating the exact mapping into voxel space. When using this approach, we still allow each participant to have its own signal and noise parameters, because the signal-to-noise ratio is idiosyncratic to each participant's data. When evaluating the likelihood of left-out data under the estimated model parameters, we therefore plug in the ML-estimates for these two parameters for each subject.

Finally, a last option is to implement a full Bayesian approach and to impose priors on all parameters, and then use a Laplace approximation to estimate the model evidence[@RN3654][@RN3255]. While it certainly can be argued that this is the most elegant approach, we find that cross-validation at the level of model parameters provides us with a practical, straightforward, and transparent way of achieving a good approximation.

Each of the inference strategies supplies us with an estimate of the model evidence. To compare models, we then calculate the log Bayes factor, which is the difference between the log model evidences.

$$\begin{aligned} \log B_{12} &= \log \tfrac{p(\mathbf{Y}|M_1)}{p(\mathbf{Y}|M_2)} \\ &= \log p(\mathbf{Y}|M_1) - \log p(\mathbf{Y}|M_2) \end{aligned}$$

Log Bayes factors of over 1 are usually considered positive evidence and above 3 strong evidence for one model over the other [@RN3654].

2.6.3 Group inference

How to perform group inference in the context of Bayesian model comparison is a topic of ongoing debate in the context of neuroimaging. A simple approach is to assume that the data of each subject is independent (a very reasonable assumption) and that the true model is the same for each subject (a maybe less reasonable assumption). This motivates the use of log Group Bayes Factors (GBF), which is simple sum of all individual log Bayes factor across all subjects n

$$\log GBF = \sum_n log B_n.$$

Performing inference on the GBF is basically equivalent to a fixed-effects analysis in neuroimaging, in which we combine all time series across subjects into a single data set, assuming they all were generated by the same underlying model. A large GBF therefore could be potentially driven by one or few outliers. We believe that the GBF therefore

does not provide a desirable way of inferring on representational models - even though it has been widely used in the comparison of DCM models [@RN2029].

At least the distribution of individual log Bayes factors should be reported for each model. When evaluating model evidences against a Bayesian criterion, it can be useful to use the average log Bayes factor, rather than the sum. This stricter criterion is independent of sample size, and therefore provides a useful estimate or effect size. It expresses how much the favored model is expected to perform better on a new, unseen subject. We can also use the individual log Bayes factors as independent observations that are then submitted to a frequentist test, using either a t-, F-, or nonparametric test. This provides a simple, practical approach that we will use in our examples here. Note, however, that in the context of group cross-validation, the log-Bayes factors across participants are not strictly independent.

Finally, it is also possible to build a full Bayesian model on the group level, assuming that the winning model is different for each subject and comes from a multinomial distribution with unknown parameters [@RN3653].

2.6.4 Noise ceilings

Showing that a model provides a better explanation of the data as compared to a simpler Null-model is an important step. Equally important, however, is to determine how much of the data the model does not explain. Noise ceilings[@RN3300] provide us with an estimate of how much systematic structure (either within or across participants) is present in the data, and what proportion is truly random. In the context of PCM, this can be achieved by fitting a fully flexible model, i.e. a free model in which the second moment matrix can take any form. The non-cross-validated fit of this model provides an absolute upper bound - no simpler model will achieve a higher average likelihood. As this estimate is clearly inflated (as it does not account for the parameter fit) we can also evaluate the free model using cross-validation. Importantly, we need to employ the same cross-validation strategy (within slash between subjects) as used with the models of interest. If the free model performs better than our model of interest even when cross-validated, then we know that there are definitely aspects of the representational structure that the model did not capture. If the free model performs worse, it is overfitting the data, and our currently best model provides a more concise description of the data. In this sense, the performance of the free model in the cross-validated setting provides a lower bound to the noise ceiling. It still may be the case that there is a better model that will beat the currently best model, but at least the current model already provides an adequate description of the data. Because they are so useful, noise ceilings should become a standard reporting requirement when fitting representational models to fMRI data, as they are in other fields of neuroscientific inquiry already. The Null-model and the upper noise ceiling also allow us to normalize the log model evidence to be between 0 (Null-model) and 1 (noise ceiling), effectively obtaining a Pseudo-R^2.

2.6.5 Inference on model components: Model families

Often, we have multiple non-exclusive explanations for the observed activity patterns, and would like to know which model components (or combinations of model components) are required to explain the data. For example, for sequence representations, we may consider as model components the representation of single fingers, finger transitions, or whole sequences (see Yokoi et al., 2019). To assess the importance of each of the components, we could fit each components seperately and test how much the marginal likelihood increases relative to the Null-model (*knock-in*). We can also fit the full model containing all components and then assess how much the marginal likelihood decreases when we leave a single model component out (*knock-out*). The most comprehensive approach, however, is to fit all combinations of components separately (Shen and Ma, 2017).

To do this, we can construct a model family containing all possible combination models by switching the individual components either on or off. If we have k components of interest, we will end up with 2^k models.

After fitting all possible model combinations, one could simply select the model combination with the highest marginal likelihood. The problem, however, is that often there are a number of combinations, which all achieve a relatively high likelihood - such that the winning model changes from data set to data set. Because the inference on individual components can depend very strongly on the winning model, this approach is inherently unstable.

To address this issue we can use *Bayesian model averaging*. We can posterior likelihood for each model component, averaged across all possible model combinations (Clyde 1999). In the context of a model family, we can calculate the

posterior probability of a model component being present ($F = 1$) from the summed posteriors of all models that contained that component ($M : F = 1$)

$$p(F = 1|data) = \frac{\sum_{M:F=1} p(data|M)p(M)}{\sum_{M} p(data|M)p(M)}$$

Finally, we can also obtain a Bayes factor as a measure of the evidence that the component is present

$$BF_{F=1} = \frac{\sum_{M:F=1} p(data|M)}{\sum_{M:F=0} p(data|M)}$$

See the *Component inference and model families* example to see how to construct and fit a model family, and how to then make inference on the individual model components.

2.7 Regularized regression

PCM can be used to tune the regularization parameter for ridge regression. Specifically, ridge regression is a special case of the PCM model

$$\mathbf{y_i} = \mathbf{Z}\mathbf{u}_i + \mathbf{X}\boldsymbol{\beta}_i + \boldsymbol{\epsilon}_p$$

where $\mathbf{u} \sim N(0, \mathbf{I}s)$ are the vectors of random effects, and $\boldsymbol{\epsilon} \sim N(0, \mathbf{I}\sigma^2_\epsilon)$ the measurement error.

$\boldsymbol{\beta}$ are the fixed effects - in the case of standard ridge regression, is the intercept. In this case $\mathbf{X}$ would be a vector of 1s. The more general implementation allows arbitrary fixed effects, which may also be correlated with the random effects.

Assuming that the intercept is already removed, the random effect estimates are:

$$\hat{\mathbf{u}} = (\mathbf{Z}^T\mathbf{Z} + \mathbf{I}\lambda)^{-1}\mathbf{Z}^T\mathbf{y}_i$$
$$\lambda = \frac{\sigma^2_\epsilon}{s} = \frac{exp(\theta_s)}{exp(\theta_\epsilon)}$$

This makes the random effects estiamtes in PCM indentical to Ridge regression with an optimal regularization coefficient λ.

The PCM regularized regression model is design to work with multivariate data, i.e. many variables $\mathbf{y}_i, \ddots, \mathbf{y}_P$ that all share the same generative model ($\mathbf{X}$, $\mathbf{Z}$), but have different random and fixed effects. Of course, the regression model works on univariate regression models with only one data vector.

Most importantly, the pcm regression model allows you to estimate a different ridge coefficients for different columns of the design matrix. In general, we can can set the covariance matrix of $\mathbf{u}$ to

$$G = \begin{bmatrix} exp(\theta_1) & & & \\ & exp(\theta_1) & & \\ & & \ddots & \\ & & & exp(\theta_Q) \end{bmatrix}$$

where Q groups of effects share the same variance (and therefore the same Ridge coefficient). In the extreme, every column in the design matrix would have its own regularization parameter to be estimated. The use of the Restricted Maxmimum Likelihood (ReML) makes the estimation of such more complex regularisation both stable and computationally feasible.

2.7.1 Practical guide

See Jupyter notebook `demos/demo_regression.ipynb` for a working example, which also shows a direct comparison to ridge regression. In this work book, we generate a example with $N = 100$ observations, $P = 10$ variables, and $Q = 10$ regressors:

```
# Make the training data:
N = 100 # Number of observations
Q = 10  # Number of random effects regressors
P = 10  # Number of variables
Z = np.random.normal(0,1,(N,Q)) # Make random design matrix
U = np.random.normal(0,1,(Q,P))*0.5 # Make random effects
Y = Z @ U + np.random.normal(0,1,(N,P)) # Generate training data
# Make testing data:
Zt = np.random.normal(0,1,(N,Q))
Yt = Zt @ U + np.random.normal(0,1,(N,P))
```

Given this data, we can now define Ridge regression model, where all regressors are sharing the same ridge coefficient. `comp` is a index matrix for each column of Z.

```
# Vector indicates that all columns are scaled by the same parameter
comp = np.array([0,0,0,0,0,0,0,0,0,0])
# Make the model
M1 = pcm.regression.RidgeDiag(comp, fit_intercept = True)
# Estimate optimal regularization parameters from training data
M1.optimize_regularization(Z,Y)
```

After estimation, the two theta parameters (for signal and noise) can be retrieved from `M1.theta_`. The Regularization parameter for ridge regression is then `exp(M1.theta_[1])/exp(M1.theta_[0])`.

The model then can be fitted to the training (or other) data to determine the coefficients.

```
# Addition fixed effects can be passed in X
M1.fit(Z, Y, X = None)
```

The random effect coefficients are stored in `M1.coefs_` and the fixed effects in `M1.beta_s`.

Finally we can predict the data for he indepenent test set and evaluate this predicton.

```
Yp = M1.predict(Zt)
R2 = 1- np.sum((Yt-Yp)**2)/np.sum((Yt)**2)
```

Finally, if we want to estimate the importance of different groups of columns, we can define different ridge coefficients for different groups of columns:

```
comp = np.array([0,0,1,1,1,1,1,2,2,2])
M2 = pcm.regression.RidgeDiag(comp, fit_intercept = True)
```

In this example, the first 2, the next 5, and the last 3 columns share one Ridge coefficient. The call to `M1.optimize_regularization(Z,Y)` causes 4 theta parameters and hence 3 regularization coefficients to be estimated. If the importance of different columns of the design matrix is truely different, this will provide better predictions.

2.8 Application examples

Here some full application examples, using Jupyter notebooks. The life notebooks and underlying data can be found at https://github.com/DiedrichsenLab/PcmPy/tree/master/demos.

2.8.1 Finger representations

Example of a fit of fixed and component pcm-models to data from M1. The models and data is taken from Ejaz et al. (2015). *Hand usage predicts the structure of representations in sensorimotor cortex*, Nature Neuroscience.

We will fit the following 5 models

- null: G=np.eye, all finger patterns are equally far away from each other, Note that in many situations the no-information null model, G = np.zeros, maybe more appropriate
- Muscle: Fixed model with G = covariance of muscle activities
- Natural: Fixed model with G = covariance of natural movements
- Muscle+nat: Combination model of muscle and natural covariance
- Noiseceil: Noise ceiling model

[1]:
```
# Import necessary libraries
import PcmPy as pcm
import numpy as np
import pickle
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Read in the activity Data (`Data`), condition vector (`cond_vec`), partition vector (`part_vec`), and model matrices for Muscle and Natural stats Models (`M`):

[2]:
```
f = open('data_demo_finger7T.p','rb')
Data,cond_vec,part_vec,modelM = pickle.load(f)
f.close()
```

Now we are build a list of datasets (one per subject) from the Data and condition vectors

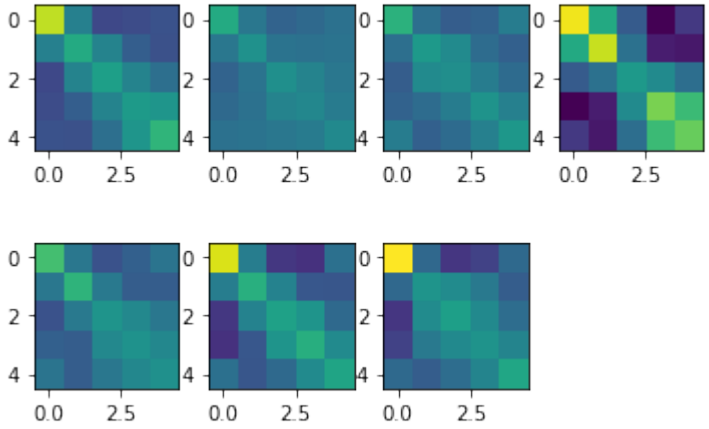
[3]:
```
Y = list()
for i in range(len(Data)):
    obs_des = {'cond_vec': cond_vec[i],
               'part_vec': part_vec[i]}
    Y.append(pcm.dataset.Dataset(Data[i],obs_descriptors = obs_des))
```

Inspect the data

Before fitting the models, it is very useful to first visualize the different data sets to see if there are outliers. One powerful way is to estimate a cross-validated estimate of the second moment matrix. This matrix is just another form of representing a cross-validated representational dissimilarity matrix (RDM).

[16]:
```
# Estimate and plot the second moment matrices across all data sets
N=len(Y)
G_hat = np.zeros((N,5,5))
for i in range(N):
    G_hat[i,:,:],_ = pcm.est_G_crossval(Y[i].measurements,
                                Y[i].obs_descriptors['cond_vec'],
                                Y[i].obs_descriptors['part_vec'],
                                X=pcm.matrix.indicator(Y[i].obs_descriptors['part_vec']))
```

[5]:
```
# show all second moment matrices
vmin = G_hat.min()
vmax = G_hat.max()
for i in range(N):
    plt.subplot(2,4,i+1)
    plt.imshow(G_hat[i,:,:],vmin=vmin,vmax=vmax)
```

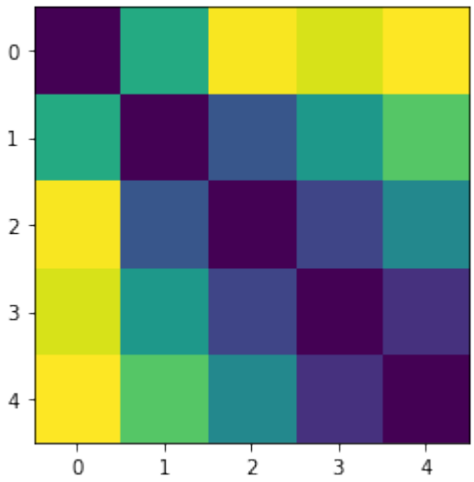


Nice - up to a scaling factor (Subject 4 has especially high signal-to-noise) all seven subjects have a very similar structure of the reresentation of fingers in M1.

If you are more used to looking in representational dissimilarity matrices (RDMs), you can also transform the second momement matrix into this (see Diedrichsen & Kriegeskorte,2017)

[17]:
```
from scipy.spatial.distance import squareform
C = pcm.pairwise_contrast(np.arange(5))
RDM = squareform(np.diag(C @ G_hat[0,:,:]@C.T))
plt.imshow(RDM)
```

[17]:
```
<matplotlib.image.AxesImage at 0x1269684f0>
```



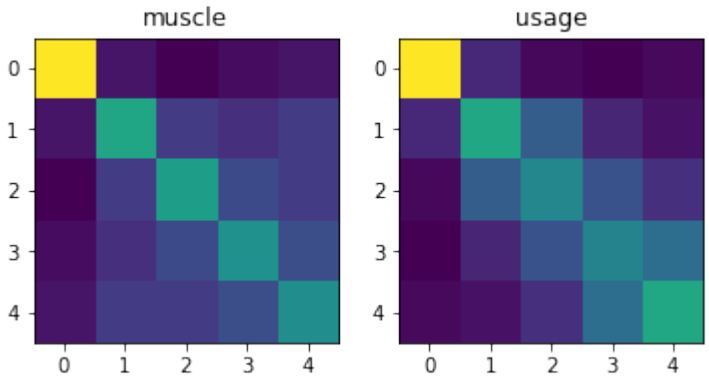
Build the models

Now we are building a list of models, using a list of second moment matrices

[18]:
```
# Make an empty list
M = []
# Null model: All fingers are represented independently.
# For RSA model that would mean that all distances are equivalent
M.append(pcm.FixedModel('null',np.eye(5)))
# Muscle model: Structure is given by covariance structure of EMG signals
M.append(pcm.FixedModel('muscle',modelM[0]))
# Usage model: Structure is given by covariance structure of EMG signals
M.append(pcm.FixedModel('usage',modelM[1]))
# Component model: Linear combination of the muscle and usage model
M.append(pcm.ComponentModel('muscle+usage',[modelM[0],modelM[1]]))
# Free noise ceiling model
M.append(pcm.FreeModel('ceil',5)) # Noise ceiling model
```

Now let's look at two underlying second moment matrices - these are pretty similar

[19]:
```
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(M[i+1].G)
    plt.title(M[i+1].name)
```



Model fitting

Now let's fit the models to individual data set. There are three ways to do this. We can fit the models

- to each individual participant with it's own parameters θ
- to each all participants together with shared parameters, but with an individual parameter for the signal strength and for group.
- in a cross-participant crossvalidated fashion. The models are fit to N-1 subjects and evaluated on the Nth subject.

[20]:
```
# Do the individual fits - suppress verbose printout
T_in, theta_in = pcm.fit_model_individ(Y,M,fit_scale = True, verbose = False)
```

[22]:
```
# Fit the model in to the full group, using a individual scaling parameter for each
T_gr, theta_gr = pcm.fit_model_group(Y, M, fit_scale=True)
```

```
Fitting model 0
Fitting model 1
Fitting model 2
Fitting model 3
Fitting model 4
```

[23]:
```
# crossvalidated likelihood
T_cv, theta_cv = pcm.fit_model_group_crossval(Y, M, fit_scale=True)
```

```
Fitting model 0
Fitting model 1
Fitting model 2
Fitting model 3
Fitting model 4
```

Inspecting and interpreting the results

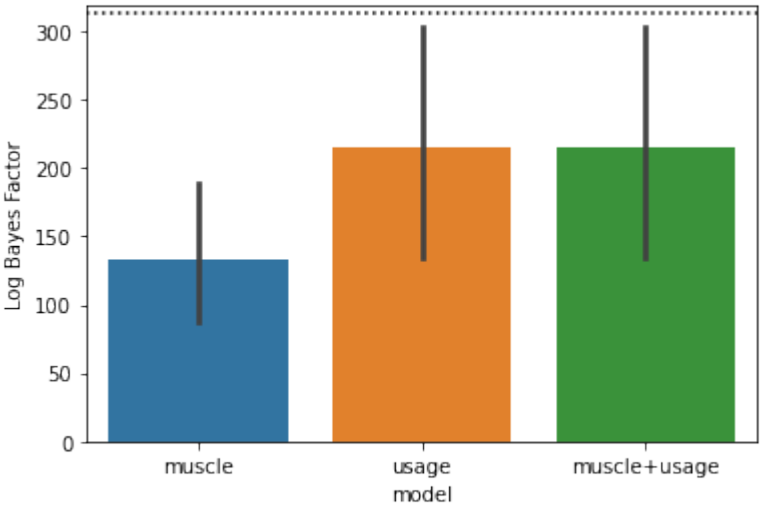
The results are returned as a nested data frame with the likelihood, noise and scale parameter for each individuals

[24]:
```
T_in
```

[24]:
```
variable    likelihood                                          \
model             null        muscle         usage  muscle+usage
0       -42231.412711 -41966.470799 -41786.672956 -41786.672927
1       -34965.171104 -34923.791342 -34915.406608 -34914.959612
2       -34767.538097 -34679.107626 -34632.643241 -34632.642946
3       -45697.970627 -45609.052395 -45448.518276 -45448.518254
4       -31993.363827 -31866.288313 -31806.982719 -31806.982521
5       -41817.234010 -41632.061473 -41543.438786 -41543.438769
6       -50336.142592 -50201.799362 -50173.300358 -50173.300306

variable                  noise                                          \
model             ceil      null    muscle     usage muscle+usage      ceil
0       -41689.860467  0.875853  0.871286  0.868482     0.868483  0.872297
1       -34889.042762  1.070401  1.067480  1.069075     1.068119  1.066987
2       -34571.750931  1.026408  1.021219  1.019122     1.019123  1.023299
3       -45225.784824  1.480699  1.479592  1.474026     1.474025  1.478701
4       -31707.184233  0.808482  0.805621  0.805774     0.805774  0.807319
5       -41439.111953  1.035696  1.031827  1.031649     1.031648  1.034879
6       -50099.140706  1.479001  1.472401  1.474430     1.474428  1.476145

variable iterations                                      scale           \
model          null muscle usage muscle+usage  ceil      null    muscle
0               4.0    4.0   4.0          7.0  30.0  0.109319  0.750145
1               4.0    4.0   4.0         16.0  21.0  0.045008  0.324407
2               4.0    4.0   4.0          7.0  32.0  0.059863  0.435483
3               4.0    4.0   4.0          6.0  18.0  0.173031  1.193770
4               4.0    4.0   4.0          6.0  32.0  0.073935  0.516338
5               4.0    4.0   4.0          5.0  19.0  0.116696  0.801114
6               4.0    4.0   4.0         12.0  29.0  0.101477  0.714043

variable
model        usage muscle+usage      ceil
0         0.786771     1.000000  0.996839
1         0.322917     0.963006  0.998003
2         0.463987     1.000000  0.992453
3         1.235628     1.000000  0.998176
4         0.532421     1.000000  0.999360
5         0.828773     1.000000  0.999325
6         0.723969     1.000000  0.981997
```

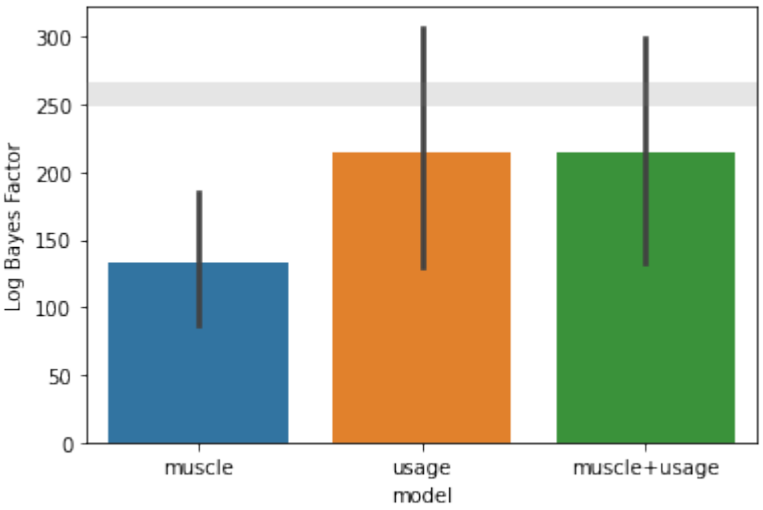
The likelihoods are very negative and quite different across participants, which is expected (see documentation). What we need to interpret are the difference is the likelihood relative to a null model. We can visualized these using the model_plot

[25]:
```
ax = pcm.model_plot(T_in.likelihood,
                    null_model = 'null',
                    noise_ceiling= 'ceil')
```



The problem with the noise ceiling is that it is individually fit to each subject. It has much more parameters than the models it is competing against, so it is overfitting. To compare models with different numbers of parameters directly, we need to look at our cross-validated group fits. The group fits can be used as an upper noise ceiling.

[26]:
```
ax = pcm.model_plot(T_cv.likelihood,
                    null_model = 'null',
                    noise_ceiling= 'ceil',
                    upper_ceiling = T_gr.likelihood['ceil'])
```



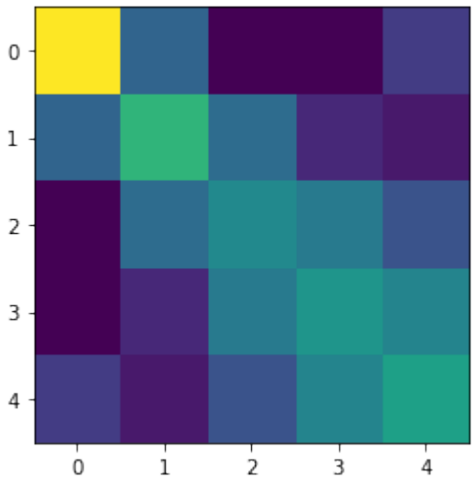
As you can see, the likelihood for individual, group, and crossvalidated group fits for the fixed models (null, muscle + usage) are all identical, because these models do not have common group parameters - in all cases we are fitting an individual scale and noise parameter.

Visualizing the model fit

Finally, it is very useful to visualize the model prediction in comparision to the fitted data. The model parameters are stored in the return argument `theta`. We can pass these to the `Model.predict()` function to get the predicted second moment matrix.

[27]:
```
G,_ = M[4].predict(theta_gr[4][:M[4].n_param])
plt.imshow(G)
```

[27]:
```
<matplotlib.image.AxesImage at 0x12633f6a0>
```



2.8.2 PCM correlation models

This demo shows two ways to use PCM models to test hypotheses about the correlation between activity patterns.

- In the first part of this jupyter notebook, we'll focus on how to assess the **true** correlation between two activity patterns.
- In the second part, we will consider a slightly more complex situation in which we want to estimate the true correlation between two **sets** of activity patterns measured under two different conditions.

  For example, we might want to know how the activity patterns related to the observation of 3 hand gestures correlate (at a gesture-specific level) with the activity patterns related to the execution of the same 3 hand gestures.

[31]:
```
# First import necessary libraries
#(make sure PcmPy is in your python path)
import PcmPy as pcm
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from numpy import exp, sqrt
import scipy.stats as sps
```

1. Estimating the true correlation between two patterns

How similar/correlated are two activity patterns? It is easy to test whether 2 activity patterns are more correlated than chance (i.e., zero correlation). However, even if the two conditions elicit exactly the same pattern, the correlation will not be 1, simply because of measurement noise. Thus, it is very hard to estimate the **true** correlation between condition **A** and **B**. As explained in our blog Brain, Data, and Science, cross-validation does not result in unbiased estimates.

To solve this problem, PCM turns the problem around: rather than asking which correlation is the best estimate given the data, let's instead ask how likely the data is given different levels of correlations. Thus, we will calculate the likelihood of the data given a range of specific correlations, $p(Y|r)$, and then compare this likelihood across a range of correlation models.

In estimating the likelihood, we also need to estimate two additional model parameters: - The strength (variance across voxels) of the activity patterns associated with condition **A**. - The strength (variance across voxels) of the activity patterns associated with condition **B**.

And one additional noise parameter: - The variance of the measurement noise across repeated measures of the same pattern.

These hyper-parameters are collectively referred to by θ (thetas). Here we will compare different models by using the (Type II) maximum likelihood to approximate the model evidence:

$p(Y|r) \approx \max_{\theta} p(Y|r, \theta)$.

> This may seem like a bit of a cheat, but works in this case quite well. Estimating parameters on the same data that you use to evaluate the model of course leads to an overestimation of the likelihood.
>
> However, as the number of hyper-parameters is low and all correlation models have the same number of parameters, this bias will be approximately stable across all models. Since we are interested in the *difference* in log-likelihood across models, this small bias simply disappears.
>
> If you want to compare models with different numbers of parameters, a more sophisticated approach (such as group-cross-validation) is required.

1.1 Simulating the data

We will use the PCM toolbox to simulate data from a given model of the underlying true (noiseless) correlation between two activity patterns (`Mtrue`). In this example, we set that our two activity patterns are positively correlated with a true correlation of 0.7.

> Note that in `pcm.CorrelationModel` we set `num_items` to 1, as we have only 1 activity pattern per condition, and we set `cond_effect` to `False`, as we do not want to model the overall effect between different conditions (each with multiple items, see section 2 below).

The true model (`Mtrue`) has 2 hyper-parameters reflecting the signal strength (or true pattern variance) for each activity pattern (`item`). In addition to the 2 model parameters, PCM also fits one parameter for the variance of the measurement noise.

[32]:
```
Mtrue = pcm.CorrelationModel('corr', num_items=1, corr=0.7, cond_effect=False)
print(f"The true model has {Mtrue.n_param} hyper parameters")
```

```
The true model has 2 hyper parameters
```

We can now use the simulation module to create 20 datasets (e.g., one per simulated participant) with a relatively low signal-to-noise level (0.2:1). We will use a design with 2 conditions and 8 partitions/runs per dataset. >Note that the thetas are specified as log(variance).

[33]:
```
# Create the design. In this case it's 2 conditions, across 8 runs (partitions)
cond_vec, part_vec = pcm.sim.make_design(n_cond=2, n_part=8)

# Starting from the true model above, generate 20 datasets/participants
# with relatively low signal-to-noise ratio (0.2:1)
# Note that signal variance is 0.2 - noise variance is by default set to 1
# For replicability, we are using a fixed random seed (100)
rng = np.random.default_rng(seed=100)
D = pcm.sim.make_dataset(model=Mtrue, \
    theta=[0,0], \
    cond_vec=cond_vec, \
    part_vec=part_vec, \
    n_sim=20, \
    signal=0.2,\
    rng=rng)
```

First let's look at the correlation that we get when we calculate the *naive* correlation—i.e. the correlation between the two estimated activity patterns.

[34]:
```
def get_corr(X,cond_vec):
    """
        Get normal correlation
    """
    p1 = X[cond_vec==0,:].mean(axis=0)
    p2 = X[cond_vec==1,:].mean(axis=0)
    return np.corrcoef(p1,p2)[0,1]

r = np.empty((20,))
for i in range(20):
    data = D[i].measurements
    r[i] = get_corr(data, cond_vec)
print(f'Estimated mean correlation: {r.mean():.4f}')
```

```
Estimated mean correlation: 0.4127
```

As we can see, due to measurement noise, the estimated mean correlation is much lower than the true value of 0.7.

This is not a problem if we just want to test the hypothesis that the true correlation is larger than zero. Then we can just calculate the individual correlations per subject and test them against zero using a t-test.

However, if we want to test whether the true correlation has a specific value (for example `true_corr=1`, indicating that the activity patterns are the same), or if we want to test whether the correlations are higher in one brain area than another, then this becomes an issue.

Different brain regions measured with fMRI often differ dramatically in their signal-to-noise ratio. Thus, we need to take into account our level of measurement noise. PCM can do that.

1.2 Fitting the data

We can solve this problem by making a series of PCM correlation models in the range e.g., [0 1] (or -1 to 1 if you want). We also generare a flexible model (`Mflex`) that has the correlation as a parameter that is being optimized.

[35]:
```
nsteps = 20 # how many correlation models?
M = [] # initialize the output list M

# Generate the models equally spaced between 0 and 1
for r in np.linspace(0, 1, nsteps):
    M.append(pcm.CorrelationModel(f"{r:0.2f}", num_items=1, corr=r, cond_effect=False))

# Now make the flexible model
Mflex = pcm.CorrelationModel("flex", num_items=1, corr=None, cond_effect=False)
M.append(Mflex)
```

We can now fit the models to the datasets in one go. The resulting dataframe `T` has the log-likelihoods for each model (columns) and dataset (rows). The second return argument `theta` contains the parameters for each model fit.

[36]:
```
# In this case, we model the block effect as a mixed effect
# We don't need to include a scale parameter given that we don't have a fixed model
T, theta = pcm.fit_model_individ(D, M, fixed_effect=None, fit_scale=False, verbose=False)
T.head()
```

[36]:
```
variable  likelihood                                                    \
model           0.00        0.05        0.11        0.16        0.21
0       -234.937150 -234.793421 -234.679377 -234.594949 -234.540553
1       -249.115550 -248.445834 -247.806381 -247.194513 -246.608029
2       -278.140577 -277.923649 -277.723567 -277.539976 -277.372656
3       -262.453736 -261.949207 -261.472521 -261.021806 -260.595609
4       -254.133454 -253.676709 -253.248643 -252.847649 -252.472559

variable                                                          ...  \
model           0.26        0.32        0.37        0.42        0.47  ...
0       -234.517117 -234.526138 -234.569756 -234.650865 -234.773248  ...
1       -246.045157 -245.504529 -244.985186 -244.486599 -244.008726  ...
2       -277.221540 -277.086717 -276.968453 -276.867193 -276.783574  ...
3       -260.192870 -259.812921 -259.455499 -259.120782 -258.809456  ...
4       -252.122629 -251.797543 -251.497447 -251.222995 -250.975441  ...

variable iterations
model          0.58 0.63 0.68 0.74 0.79 0.84 0.89 0.95 1.00 flex
0               4.0  4.0  5.0  5.0  5.0  6.0  7.0  8.0  9.0  4.0
1               4.0  4.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  6.0
2               3.0  3.0  3.0  3.0  3.0  3.0  4.0  4.0  4.0  5.0
3               3.0  3.0  3.0  3.0  3.0  2.0  3.0  3.0  4.0  5.0
4               3.0  3.0  3.0  3.0  3.0  3.0  4.0  4.0  4.0  5.0

[5 rows x 63 columns]
```

1.3 Interpreting the fit

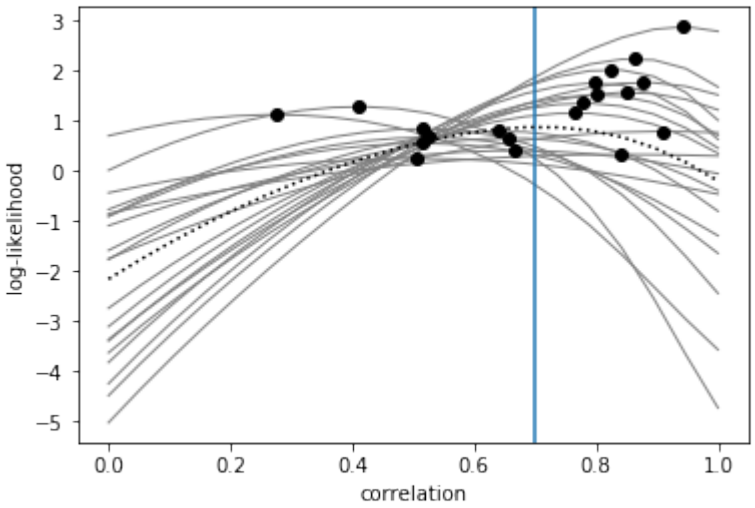
Note that the log-likelihood values are negative and differ substantially across datasets. This is normal—the only thing we can interpret are differences between log-likelihoods across the same data set for different models.

Therefore, first, we remove the mean log-likelihood for each dataset across correlation models, expressing each log-likelihood as the difference from the mean.

Next, we plot the full log-likelihood curves (solid lines) and the maximum likelihood estimate (filled circles) of the correlation for each participant. We can also add the mean log-likelihood curve (dotted line) and the mean of the maximum log-likelihood estimates (vertical blue line) across participants.

[37]:
```
L = T.likelihood.to_numpy()

# express log-likelihoods relative to mean
L = L - L.mean(axis=1).reshape(-1,1)
maxL = L[:,-1] # Last model is the flexible
L = L[:,0:-1] # Remove it

# Get the correlation for each of the models
r = np.empty((nsteps,))
for i in range(nsteps):
    r[i] = M[i].corr

# Get the maximum likelihood estimate of each parameter
maxr = M[-1].get_correlation(theta[-1])

# Now we make the plot:
# Plot the individual log-liklihood curves and their peak in gray lines:
plt.plot(r, L.T, color='gray', marker=None, linewidth=1)
plt.plot(maxr, maxL, 'k.', markersize=12)

# Plot the mean of the maximum-liklihood estimates in a vertical line
plt.axvline(maxr.mean())

# Plot the mean of the likelihood curves as a dashed line
plt.plot(r,L.mean(axis=0), 'k:')
plt.xlabel('correlation')
plt.ylabel('log-likelihood')
```

[37]:
```
Text(0,0.5,'log-likelihood')
```



As we can see, the maximum-likelihood estimates (filled circles) behave quite well, as they are at least around the true correlation value (0.7).

However, the mean of the maximum-likelihood estimates (vertical line) is unfortunately not unbiased but slightly biased towards zero (see Brain, Data, and Science blog). Therefore, best way to use the log-likelihoods is to do a paired-samples t-test between the log-likelihoods for two correlation values.

For example, 0.7 vs. 1, or 0.7 vs. 0:

[38]:
```
print(f"Testing correlation of {r[13]:0.2f} against {r[19]:0.2f}")
t, p = sps.ttest_rel(L[:,13],L[:,19])
print(f'Paired-samples t-test: t({len(L)-1})={t:1.3f}; p({len(L)-1})={p:1.6f}\n')

print(f"Testing correlation of {r[13]:0.2f} against {r[0]:0.2f}")
t, p = sps.ttest_rel(L[:,13],L[:,0])
print(f'Paired-samples t-test: t({len(L)-1})={t:1.3f}; p({len(L)-1})={p:1.6f}')
```

```
Testing correlation of 0.68 against 1.00
Paired-samples t-test: t(19)=3.334; p(19)=0.003491

Testing correlation of 0.68 against 0.00
Paired-samples t-test: t(19)=6.101; p(19)=0.000007
```

Thus, we have clear evidence that the correlation of 0.68 is more likely than a correlation of zero (i.e., that the patterns are unrelated), and more likely than a correlation of one (i.e., that the patterns are identical).

2. Testing for specific correlations between activity patterns across two conditions

In the second part of this notebook, we will simulate data from a hypothetical experiment, in which participants observed 3 hand gestures or executed the same 3 hand gestures. Thus, we have 3 items (i.e., the hand gestures) in each of 2 conditions (i.e., either observe or execute).

We are interested in the average correlation between the patterns associated with observing and executing action A, observing and executing action B, and observing and executing action C, while accounting for overall differences in the average patterns of observing and executing. To solve this problem, we again calculate the likelihood of the data given a range of specific correlations $p(Y|r)$.

In this case we have a few more additional hyper-parameters to estimate: - The variance of the movement-specific activity patterns associated with observing actions. - The variance of the movement-specific activity patterns associated with executing actions.

> These hyper-parameters express the strength of encoding and are directly related to the average inter-item distance in an RSA analysis.

- The variance of the pattern component that is common to all observed actions.
- The variance of the pattern component that is common to all executed actions.
- Finally, we again have a hyper-parameter for the noise variance.

2.1 Data simulation

First, we create our true model (`Mtrue`): one where the all actions are equally strongly encoded in each condition, but where the strength of encoding can differ between conditions (i.e., between observation or execution).

> For example, we could expect the difference between actions to be smaller during observation than during execution (simply due to overall levels of brain activation).

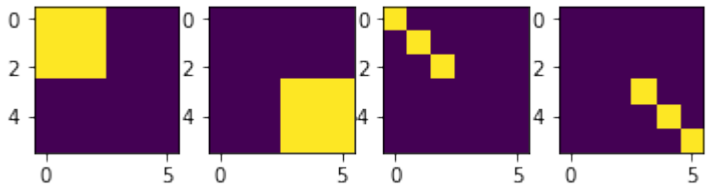
Next, we also model the covariance between items within each condition with a condition effect (i.e., by setting `condEffect` to `True`). Finally, we set the ground-truth correlation to be 0.7.

[39]:
```
Mtrue = pcm.CorrelationModel('corr', num_items=3, corr=0.7, cond_effect=True, within_
↪cov=None)
print(f"The true model has {Mtrue.n_param} hyper parameters")
```

```
The true model has 4 hyper parameters
```

These four parameters are concerned with the condition effect and item effect for observation and execution, respectively. Visualizing the components of the second moment matrix (also known as variance-covariance, or simply covariance matrix) helps to understand this:

[40]:
```
H = Mtrue.n_param
for i in range(H):
    plt.subplot(1, H, i+1)
    plt.imshow(Mtrue.Gc[i,:,:])
```



> The first two components plotted above reflect the condition effect and model the covariance between items within each condition (observation, execution). The second two components reflect the item effect and model the item-specific variance for each item (3 hand gestures) in each condition.

To Simulate a dataset, we need to simulate an experimental design. Let's assume we measure the 6 trial types (3 items x 2 conditions) in 8 imaging runs and submit the beta-values from each run to the model as $\mathbf{Y}$.

We then generate a dataset where there is a strong overall effect for both observation (exp(0)) and execution (exp(1)). In comparison, the item-specific effects for observation (exp(-1.5)) and execution (exp(-1)) are pretty weak (this is a rather typical finding).

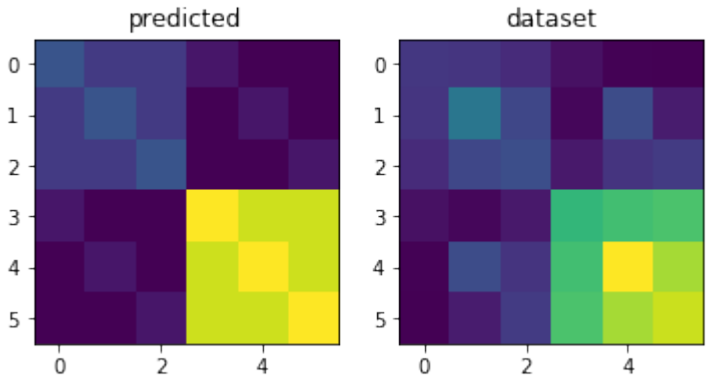
Note that all hyper parameters are log(variances)—this helps us to keep variances positive and the math easy.

[41]:
```
# Create the design. In this case it's 8 runs, 6 trial types
cond_vec, part_vec = pcm.sim.make_design(n_cond=6, n_part=8)
#print(cond_vec)
#print(part_vec)

# Starting from the true model above, generate 20 datasets/participants
# with relatively low signal-to-noise ratio (0.2:1)
D = pcm.sim.make_dataset(model=Mtrue, \
    theta=[1,2.7,0.4,0.2], \
    cond_vec=cond_vec, \
    part_vec=part_vec, \
    n_sim=20, \
    signal=0.2)
```

As a quick check, let's plot the predicted second moment matrix of our true model (using the simulation parameters) and the crossvalidated estimate from the first dataset.

[42]:
```
# Get the predicted G-matrix  from the true model
G,_ = Mtrue.predict([1,2.7,0.4,0.2])

# The estimated G-matrix from the first dataset
trial_type = D[1].obs_descriptors['cond_vec']
G_hat,_ = pcm.est_G_crossval(D[1].measurements, trial_type, part_vec)

# Visualize the second moment (G) matrices
plt.subplot(1,2,1)
plt.imshow(G)
plt.title('predicted')

plt.subplot(1,2,2)
plt.imshow(G_hat)
plt.title('dataset')
```

[42]:
```
Text(0.5,1,'dataset')
```



2.2 Fitting the data

Now we are fitting these datasets with a range of models, each assuming a correlation value between 0 and 1. The other parameters will still be included, as we did for the true model.

For comparison, we also include a flexible correlation model (`Mflex`), which has a additional free parameter that models the correlation.

[44]:
```
nsteps = 100 # how many correlation models?
M = [] # initialize the output list M

# Generate the models equally spaced between 0 and 1
for r in np.linspace(0, 1, nsteps):
    M.append(pcm.CorrelationModel(f"{r:0.2f}", num_items=3, corr=r, cond_effect=True))

# Now make the flexible model
Mflex = pcm.CorrelationModel("flex", num_items=3, corr=None, cond_effect=True)
M.append(Mflex)
```

We can now fit the model to the datasets in one go. The resulting dataframe `T` has the log-likelihoods for each model (columns) / dataset (rows). The second return argument `theta` contains the parameters for each model fit.

[45]:
```
# In this case, we model the block effect as a mixed effect
# We don't need to include a scale parameter given that we don't have a fixed model
T, theta = pcm.fit_model_individ(D, M, fixed_effect='block', fit_scale=False,␣
↪verbose=False)
T.head()
```

[45]:
```
variable  likelihood                                                    \
model           0.00        0.01        0.02        0.03        0.04
0         -887.831487 -887.705964 -887.582884 -887.462226 -887.343969
1         -897.418265 -897.100121 -896.784548 -896.471492 -896.160898
2         -938.057184 -937.851286 -937.647757 -937.446563 -937.247669
3         -938.816255 -938.592624 -938.371687 -938.153403 -937.937735
4         -875.956085 -875.773279 -875.593541 -875.416842 -875.243151

variable                                                          ...  \
model           0.05        0.06        0.07        0.08        0.09  ...
0         -887.228095 -887.114588 -887.003433 -886.894614 -886.788120  ...
1         -895.852715 -895.546892 -895.243381 -894.942133 -894.643103  ...
2         -937.051042 -936.856650 -936.664465 -936.474457 -936.286599  ...
3         -937.724648 -937.514106 -937.306077 -937.100530 -936.897437  ...
4         -875.072443 -874.904691 -874.739873 -874.577969 -874.418959  ...

variable iterations
model          0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99 1.00 flex
0               6.0  6.0  6.0  6.0  6.0  6.0  6.0  6.0  6.0  5.0
1               6.0  6.0  6.0  6.0  6.0  6.0  6.0  6.0  6.0  7.0
2               5.0  5.0  5.0  5.0  5.0  5.0  5.0  5.0  5.0  5.0
3               6.0  6.0  6.0  6.0  6.0  6.0  6.0  6.0  6.0  6.0
4               6.0  6.0  6.0  6.0  6.0  6.0  6.0  6.0  6.0  5.0

[5 rows x 303 columns]
```

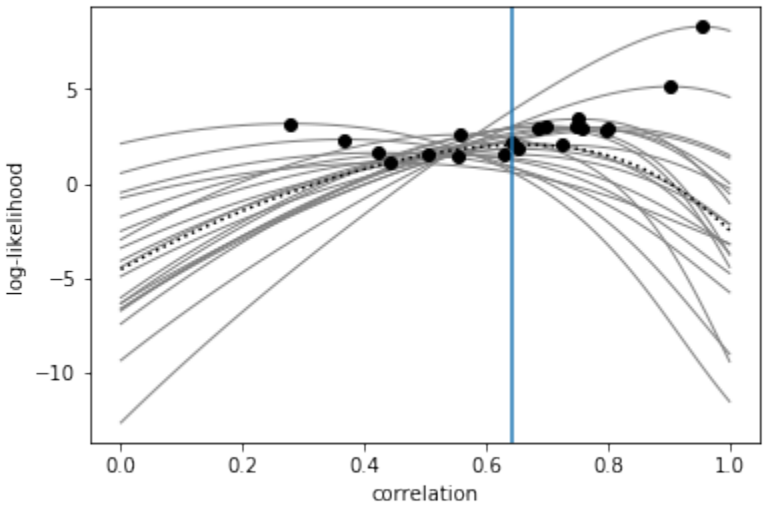
2.3 Interpreting the Fit

Again, note that the absolute values of the log-likelihoods don't mean much. Therefore, first, we remove the mean log-likelihood for each correlation model, expressing each log-likelihood as the difference against the mean.

Next, we plot the full log-likelihood curves (solid lines) and the maximum likelihood estimate (filled circles) of the correlation for each participant. We can also add the mean log-likelihood curve (dotted line) and the mean of the maximum log-likelihood estimates (vertical blue line) across participants.

[46]:
```
L = T.likelihood.to_numpy()

# express log-likelihoods relative to mean
L = L - L.mean(axis=1).reshape(-1,1)
maxL = L[:,-1] # Last model is the flexible
L = L[:,0:-1] # Remove it

# Get the correlation for each of the models
r = np.empty((nsteps,))
for i in range(nsteps):
    r[i]= M[i].corr

# Get the maximum likelihood estimate of each parameter
maxr = M[-1].get_correlation(theta[-1])

# Now we make the plot
plt.plot(r, L.T, color='gray', marker=None, linewidth=1)
plt.plot(maxr, maxL, 'k.', markersize=12)
plt.plot(r, L.mean(axis=0), 'k:')
plt.axvline(maxr.mean())
plt.xlabel('correlation')
plt.ylabel('log-likelihood')
```

[46]:
```
Text(0,0.5,'log-likelihood')
```



Again, the best way to use the log-likelihoods is to do a paired-samples t-test between the log-likelihoods for two correlation values: e.g., 0.7 vs 0.3:

[48]:
```
print(f"Testing correlation of {r[69]:0.2f} against {r[30]:0.2f}")
t, p = sps.ttest_rel(L[:,69], L[:,30])
print(f'nPaired-samples t-test: t({len(L)-1})={t:1.3f}; p({len(L)-1})={p:1.6f}')
```

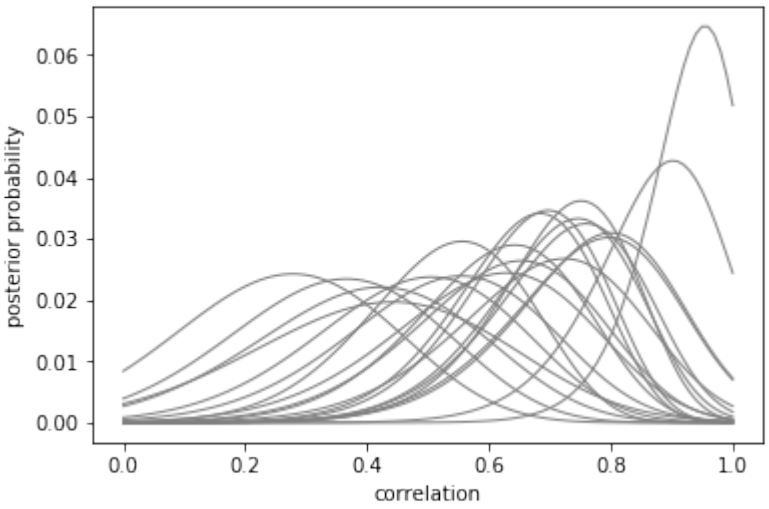
```
Testing correlation of 0.70 against 0.30
nPaired-samples t-test: t(19)=3.331; p(19)=0.003513
```

Thus, we have clear evidence that the true correlation is much more likely to be 0.7 than 0.3.

Alternatively, we can transform the log-likelihoods in approximate posterior distributions, and proceed with a Full Bayesian group analysis. For more accurate results, you probably want to space your correlation models more tightly.

[49]:
```
P = exp(L) # Take the exponential of the log-likelihoods
P = P / P.sum(axis=1).reshape(-1,1) # Normalize to 1 over domain
plt.plot(r, P.T, color='gray', marker=None, linewidth=1)
plt.xlabel('correlation')
plt.ylabel('posterior probability')
```

[49]:
```
Text(0,0.5,'posterior probability')
```



2.8.3 Component inference and model families

This notebook shows a series of examples of how to use model families and component inference with PCM.

[1]:
```
# Import necessary libraries
import PcmPy as pcm
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb
import scipy.io as io
```

Example 1: Simple 3 x 2 Design

This is a classical example of a fully crossed design with two factors (A and B) and the possibility for an interaction between those two factors. A experimental example would be that you measure the response to 6 stimuli, with 3 different objects in two different colors. You want to test whether a) objects are represented b) colors are represented c) the unique combination of color and object is represented. Thus, this is a MANOVA-like design where you want to test for the main effects of A and B, as well as their interaction.

Note: When building the features for the 3 different components, the interaction usually has to the orthogonalized for the main effects. If the interaction is not orthogonalized, the interaction effect can explain part of the variance explained by the main effects. Try out what happens when you use the non-orthogonalized version of the interaction effect - you will see that a Model Family deals correctly with this situation as well!

Generating data from a 3x2 Design

Note that for data generation, we consider the interaction fixed - meaning that when the interaction effect is present, no difference between categories of A and B occur.

[2]:
```
# Generate the three model components, each one as a fixed model
A = np.array([[1.0,0,0],[1,0,0],[0,1,0],[0,1,0],[0,0,1],[0,0,1]])
B = np.array([[1.0,0],[0,1],[1,0],[0,1],[1,0],[0,1]])
I = np.eye(6)
# Orthogonalize the interaction effect
X= np.c_[A,B]
Io = I-X @ np.linalg.pinv(X) @ I

# Now Build the second moment matrix and create the full model
# for data generation:
Gc = np.zeros((3,6,6))
Gc[0]=A@A.T
Gc[1]=B@B.T
Gc[2]=Io@Io.T

trueModel = pcm.ComponentModel('A+B+I',Gc)
```

Now generate 20 data set from the full model. The vector `theta` gives you the log-variance of the A,B, and Interaction components. Here A is absent, and the interaction stronger than the B-effect. You can play around with the values to check out other combinations.

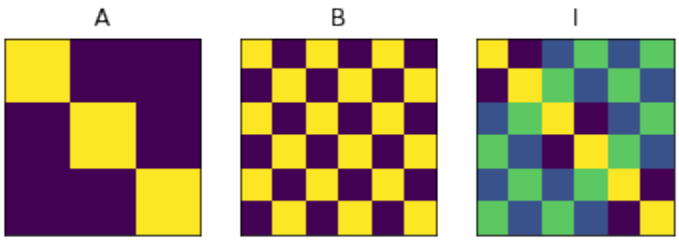
[3]:
```
[cond_vec,part_vec]=pcm.sim.make_design(6,8)
theta = np.array([-np.inf,-1,0])
D = pcm.sim.make_dataset(trueModel,theta,
        signal=0.1,
        n_sim = 20,
        n_channel=20,
        part_vec=part_vec,
        cond_vec=cond_vec)
```

Building a model family

Now we construct a model family for all possible combinations

Note:Here you can decide whether to use the orthogonalized interaction effect or not

[4]:
```
# Use orthogonlized interaction effect
Gc[2]=Io@Io.T
MF=pcm.model.ModelFamily(Gc,comp_names=['A','B','I'])

# Show the three model components
for i in range(3):
    ax = plt.subplot(1,3,i+1)
    plt.imshow(Gc[i])
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.title(MF.comp_names[i])
```

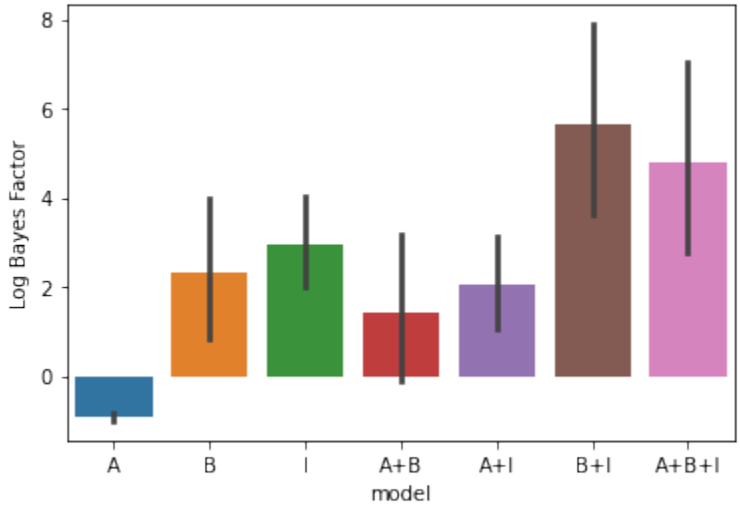


Fitting the data

Now we can fit the data with the entire model family. An intercept is added as a fixed effect for each partition (block) seperately, as common for fMRI data. The result is a likelihood for each of the model combination.

[5]:
```
# Fit the data and display the relative likelihood.
T,theta=pcm.fit_model_individ(D,MF,verbose=False,fixed_effect='block', fit_scale=False)
# Here we correcting for the number of parameters (using AIC)
pcm.vis.model_plot(T.likelihood-MF.num_comp_per_m)
```

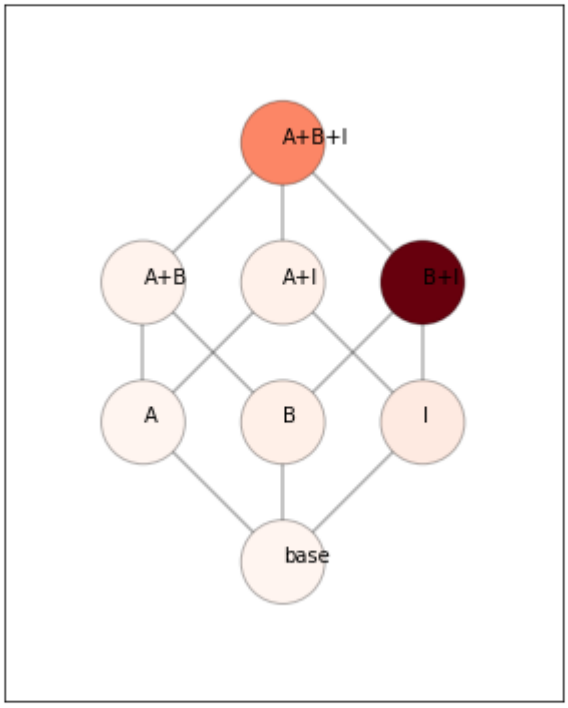
[5]:
```
<AxesSubplot:xlabel='model', ylabel='Log Bayes Factor'>
```



Inference: Model posterior

For the inference, we can use either crossvalidated pseudo-likelihoods (within- subject or between subjects - see inference), or we can use the fitted likelihood, correcting for the number of paramaters using an AIC approach. We use latter approach here.

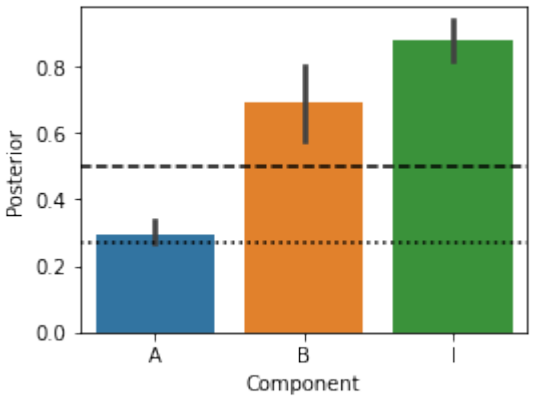
[6]:
```
# We can computer the posterior probability of each of the mixture models.
# This uses the flat prior over all possible model combinations
# The whole model family can visualized as a model tree.
plt.figure(figsize=(5,7))
# Get the mean likelihood

mposterior = MF.model_posterior(T.likelihood.mean(axis=0),method='AIC',format='DataFrame
↪')
pcm.vis.plot_tree(MF,mposterior,show_labels=True,show_edges=True)
# mposterior.to_numpy()
```



Inference: Component posterior

We can also get the posterior probability for each component. This is simply the sum of the posterior probabilties of all the model combinations that contain that component. The 0.5 line (this is the prior probability) is drawn. The lower line is the most evidence we can get for the absence of a model component using AIC. This is because, in the worst case, a new component does not increase the likelihood at all. This would result in the new component having a realtive likelihood that is 1.0 lower than the simpler model (parameter penality). Thus, overall the worst we can get p=1/(1+exp(1)).

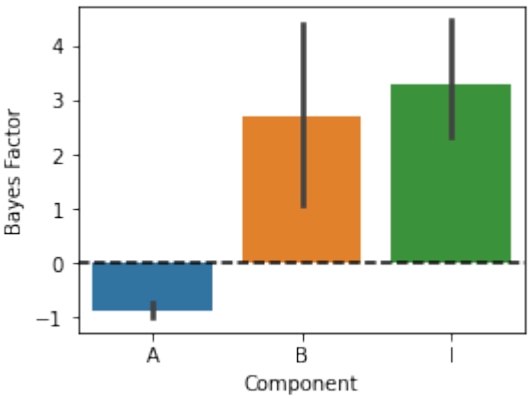
[7]:
```
# Component posterior
plt.figure(figsize=(4,3))
cposterior = MF.component_posterior(T.likelihood,method='AIC',format='DataFrame')
pcm.vis.plot_component(cposterior,type='posterior')
```



Component Bayes factor

For frequentist statistical testing and display, it is also often useful use the log-odds of the posterior $log(p/(1-p))$. For a flat prior across the model family, this is the bayes factor for the specific component.

[8]:
```
# Component Bayes Factor
plt.figure(figsize=(4,3))
c_bf = MF.component_bayesfactor(T.likelihood,method='AIC',format='DataFrame')
pcm.vis.plot_component(c_bf,type='bf')
```

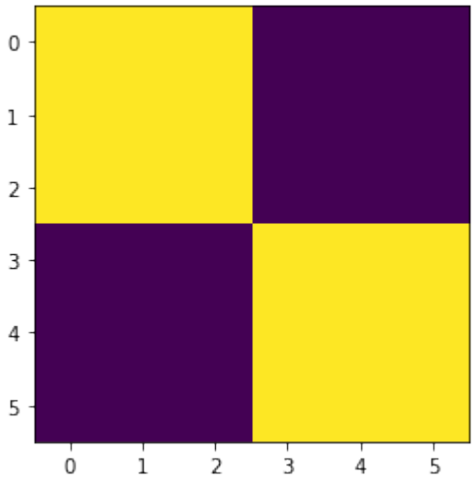


Base components

Often, there are components in a Model family that we always want to have in our model - these can be specified as "base components". By default (i.e. no base components are specified), the base model will be the zero model (there are no differences between any of the patterns). In this example, we add a strong pattern that predicts that pattern 1-3 and pattern 4-6 are correlated.

[9]:
```
basecomp = np.zeros((1,6,6))
basecomp[0] = np.block([[np.ones((3,3)),np.zeros((3,3))],[np.zeros((3,3)),np.ones((3,
↪3))]])
plt.imshow(basecomp[0])
```

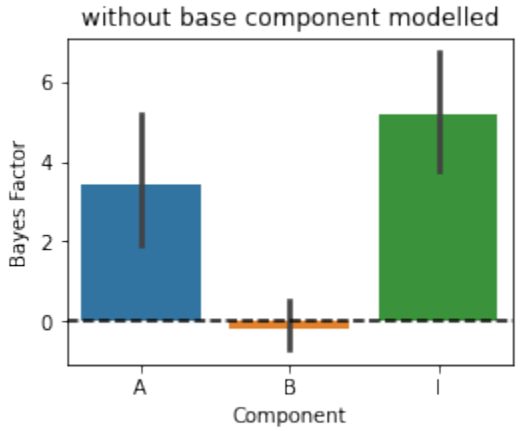
[9]:
```
<matplotlib.image.AxesImage at 0x1296d46d0>
```



[10]:
```
trueModel = pcm.ComponentModel('A+B+I+basecomponent',np.r_[Gc,basecomp])

# Now generate 20 data set from the full model
[cond_vec,part_vec]=pcm.sim.make_design(6,8)
# Interaction effect and base component are present
theta = np.array([-np.inf,-np.inf,0,0])
D = pcm.sim.make_dataset(trueModel,theta,
        signal=0.1,
        n_sim = 20,
        n_channel=20,part_vec=part_vec,
        cond_vec=cond_vec)
```

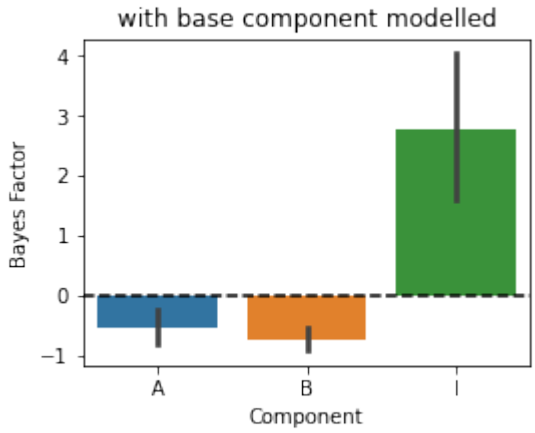
First we fit this data with a model without the base component. You will see that the base-effect mimics as the main effect A and gives us a false positive - component A is not really present.

[11]:
```
MF=pcm.model.ModelFamily(Gc,comp_names=['A','B','I'])
# Fit the data and display the relative likelihood.
T,theta=pcm.fit_model_individ(D,MF,verbose=False,fixed_effect='block',fit_scale=False)

# Component Bayes Factor
plt.figure(figsize=(4,3))
plt.title('without base component modelled')
c_bf = MF.component_bayesfactor(T.likelihood,method='AIC',format='DataFrame')
pcm.vis.plot_component(c_bf,type='bf')
```



Now we are adding the base component to all the models - so all models fitted will contain this extra component.

[12]:
```
MF=pcm.model.ModelFamily(Gc,comp_names=['A','B','I'],basecomponents=basecomp)
# Fit the data and display the relative likelihood.
T,theta=pcm.fit_model_individ(D,MF,verbose=False,fixed_effect='block',fit_scale=False)

# Component Bayes Factor
plt.figure(figsize=(4,3))
plt.title('with base component modelled')
c_bf = MF.component_bayesfactor(T.likelihood,method='AIC',format='DataFrame')
pcm.vis.plot_component(c_bf,type='bf')
```



Now the inference is correct again. So if you have a strong correlation structure in your data that needs to be modeled (but that you do not want to draw inferences on), add it as a base component!

Example 2: Random example with 5 partly co-linear representations

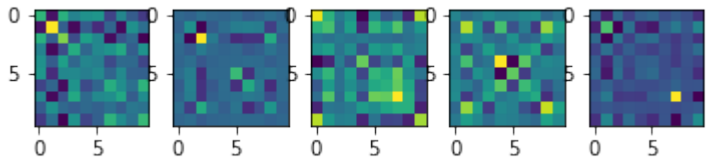
In this example, we provide an example with random components that are partly overlapping with each other. We also provide a function that performs different forms of component inference.

[13]:
```
# Another function that makes a random feature model
# using Q feature groups (components), each consisting of num_feat features.
def random_design(N=10,Q=5,num_feat=2,seed=1):
    Gc = np.empty((Q,N,N))
    rng = np.random.default_rng(seed)
    for q in range(Q):
        X= rng.normal(0,1,(N,num_feat))
        X = X/np.sqrt(np.sum(X**2,axis=0))
        Gc[q,:,:]= X @ X.T
    M = pcm.ComponentModel('full',Gc)
    MF=pcm.model.ModelFamily(Gc)
    return M,MF
```

[14]:
```
def component_inference(D,MF):
    T,theta=pcm.fit_model_individ(D,MF,verbose=False)

    # pcm.vis.model_plot(T.likelihood-MF.num_comp_per_m)
    mposterior = MF.model_posterior(T.likelihood.mean(axis=0),method='AIC',format=
↪'DataFrame')
    cposterior = MF.component_posterior(T.likelihood,method='AIC',format='DataFrame')
    c_bf = MF.component_bayesfactor(T.likelihood,method='AIC',format='DataFrame')

    fig=plt.figure(figsize=(18,3.5))
    plt.subplot(1,3,1)
    pcm.vis.plot_tree(MF,mposterior,show_labels=False,show_edges=True)
    ax=plt.subplot(1,3,2)
    pcm.vis.plot_component(cposterior,type='posterior')
    ax=plt.subplot(1,3,3)
    pcm.vis.plot_component(c_bf,type='bf')
```

Now we make an example with 5 component and 10 conditions…

[19]:
```
# make and plot a random design
N = 10
Q = 5
M,MF = random_design(N=N,Q=Q,seed=1)
for q in range(Q):
        plt.subplot(1,Q,q+1)
        plt.imshow(M.Gc[q,:,:])
```

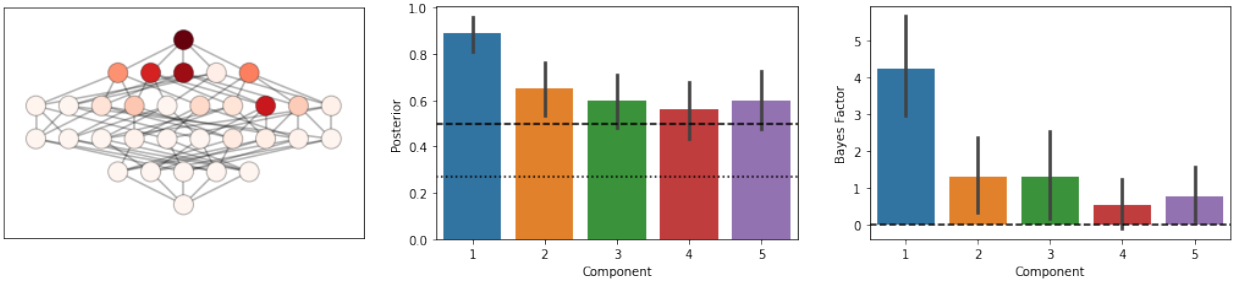


[20]:
```
# Let's check the co-linearity of these particular components by looking at their␣
↪correlation
GG = np.reshape(M.Gc,(Q,-1)) # Flatten out
np.corrcoef(GG)
```

[20]:
```
array([[1.        , 0.24581198, 0.05609763, 0.3498265 , 0.42614867],
       [0.24581198, 1.        , 0.10612771, 0.20962178, 0.24458923],
       [0.05609763, 0.10612771, 1.        , 0.05158634, 0.3199829 ],
       [0.3498265 , 0.20962178, 0.05158634, 1.        , 0.15116567],
       [0.42614867, 0.24458923, 0.3199829 , 0.15116567, 1.        ]])
```

In this example, component 2 and 3 are relatively uncorrelated

[22]:
```
# Now make artifical data and plot component inference
# In this example, component 1 and 4 are not present
theta = np.array([0,-1,-1,-1,-1])

[cond_vec,part_vec]=pcm.sim.make_design(N,8)
D = pcm.sim.make_dataset(M,theta,
                         signal=0.2,
                         n_sim = 20,
                         n_channel=20,part_vec=part_vec,
                         cond_vec=cond_vec)
component_inference(D,MF)
```



[ ]:

2.9 Mathematical details

2.9.1 Likelihood

In this section, we derive the likelihood in the case that there are no fixed effects. In this case the distribution of the data would be

$$\mathbf{y} \sim N(0, \mathbf{V})$$
$$\mathbf{V} = \mathbf{ZGZ^T} + \mathbf{S}\sigma_\epsilon^\mathbf{2}$$

To calculate the likelihood, let us consider at the level of the single voxel, namely, $\mathbf{Y} = [\mathbf{y_1}, \mathbf{y_2}, ..., \mathbf{y_p}]$. Then the likelihood over all voxels, assuming that the voxels are independent (e.g. effectively pre-whitened) is

$$p(\mathbf{Y}|\mathbf{V}) = \prod_{i=1}^{P} (2\pi)^{-\frac{N}{2}} |\mathbf{V}|^{-\frac{1}{2}} exp\left(-\frac{1}{2}\mathbf{y}_i^T \mathbf{V}^{-1} \mathbf{y}_i\right)$$

When we take the logarithm of this expression, the product over the individual Gaussian probabilities becomes a sum and the exponential disappears:

$$L = \ln\left(p\left(\mathbf{Y}|\mathbf{V}\right)\right) = \sum_{i=1}^{P} \ln p\left(\mathbf{y_i}\right)$$

$$= -\frac{NP}{2}\ln\left(2\pi\right) - \frac{P}{2}\ln\left(|\mathbf{V}|\right) - \frac{1}{2}\sum_{i=1}^{P}\mathbf{y}_i^T\mathbf{V}^{-1}\mathbf{y}_i$$

$$= -\frac{NP}{2}\ln\left(2\pi\right) - \frac{P}{2}\ln\left(|\mathbf{V}|\right) - \frac{1}{2}trace\left(\mathbf{Y}^T\mathbf{V}^{-1}\mathbf{Y}\right)$$

Using the trace trick, which allows $\mathrm{trace}\left(\mathbf{ABC}\right) = \mathrm{trace}\left(\mathbf{BCA}\right)$, we can obtain a form of the likelihood that does only depend on the second moment of the data, $\mathbf{YY^T}$,as a sufficient statistics:

$$L = -\frac{NP}{2}\ln\left(2\pi\right) - \frac{P}{2}\ln\left(|\mathbf{V}|\right) - \frac{1}{2}trace\left(\mathbf{Y}\mathbf{Y}^T\mathbf{V}^{-1}\right)$$

2.9.2 Restricted likelihood

In the presence of fixed effects (usually effects of no interest), we have the problem that the estimation of these fixed effects depends iterativly on the current estimate of $\mathbf{V}$ and hence on the estimates of the second moment matrix and the noise covariance.

$$\hat{\mathbf{B}} = \left(\mathbf{X}^T\mathbf{V}^{-1}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{V}^{-1}\mathbf{Y}$$

Under the assumption of fixed effects, the distribution of the data is

$$\mathbf{y_i} \sim N\left(\mathbf{X}\mathbf{b_i}, \mathbf{V}\right)$$

To compute the likelihood we need to remove these fixed effects from the data, using the residual forming matrix

$$\mathbf{R} = \mathbf{I} - \mathbf{X}\left(\mathbf{X^T}\mathbf{V^{-1}}\mathbf{X}\right)^{-1}\mathbf{X^T}\mathbf{V^{-1}}$$

$$\mathbf{r_i} = \mathbf{R}\mathbf{y_i}$$

For the optimization of the random effects we therefore also need to take into account the uncertainty in the fixed effects estimates. Together this leads to a modified likelihood - the restricted likelihood.

$$L_{ReML} = -\frac{NP}{2}\ln\left(2\pi\right) - \frac{P}{2}\ln\left(|\mathbf{V}|\right) - \frac{1}{2}trace\left(\mathbf{Y}\mathbf{Y}^T\mathbf{R}^T\mathbf{V}^{-1}\mathbf{R}\right) - \frac{P}{2}\ln|\mathbf{X^T}\mathbf{V^{-1}}\mathbf{X}|$$

Note that the third term can be simplified by noting that

$$\mathbf{R^T}\mathbf{V^{-1}}\mathbf{R} = \mathbf{V^{-1}} - \mathbf{V^{-1}}\mathbf{X}(\mathbf{X}\mathbf{V^{-1}}\mathbf{X})^{-1}\mathbf{X^T}\mathbf{V^{-1}} = \mathbf{V^{-1}}\mathbf{R} = \mathbf{V_R^{-1}}$$

2.9.3 First derivatives of the log-likelihood

Next, we find the derivatives of *L* with respect to each hyper parameter θ_i, which influence G. Also we need to estimate the hyper-parameters that describe the noise, at least the noise parameter σ_ϵ^2. To take these derivatives we need to use two general rules of taking derivatives of matrices (or determinants) of matrices:

$$\frac{\partial \ln|\mathbf{V}|}{\partial\theta_i} = trace\left(\mathbf{V}^{-1}\frac{\partial\mathbf{V}}{\partial\theta_i}\right)$$

$$\frac{\partial\mathbf{V}^{-1}}{\partial\theta_i} = \mathbf{V}^{-1}\left(\frac{\partial\mathbf{V}}{\partial\theta_i}\right)\mathbf{V}^{-1}$$

Therefore the derivative of the log-likelihood in [@eq:logLikelihood]. in respect to each parameter is given by:

$$\frac{\partial L_{ML}}{\partial\theta_i} = -\frac{P}{2}trace\left(\mathbf{V}^{-1}\frac{\partial\mathbf{V}}{\partial\theta_i}\right) + \frac{1}{2}trace\left(\mathbf{V}^{-1}\frac{\partial\mathbf{V}}{\partial\theta_i}\mathbf{V}^{-1}\mathbf{Y}\mathbf{Y}^T\right)$$

2.9.4 First derivatives of the restricted log-likelihood

First, let's tackle the last term of the restricted likelihood function:

$$l = -\frac{P}{2}\ln|\mathbf{X}^T\mathbf{V}^{-1}\mathbf{X}|$$

$$\begin{aligned}\frac{\partial l}{\partial \theta_i} &= -\frac{P}{2}trace\left(\left(\mathbf{X}^T\mathbf{V}^{-1}\mathbf{X}\right)^{-1}\mathbf{X}^T\frac{\partial \mathbf{V}^{-1}}{\partial \theta_i}\mathbf{X}\right)\\ &= \frac{P}{2}trace\left(\left(\mathbf{X}^T\mathbf{V}^{-1}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{V}^{-1}\frac{\partial \mathbf{V}}{\partial \theta_i}\mathbf{V}^{-1}\mathbf{X}\right)\\ &= \frac{P}{2}trace\left(\mathbf{V}^{-1}\mathbf{X}\left(\mathbf{X}^T\mathbf{V}^{-1}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{V}^{-1}\frac{\partial \mathbf{V}}{\partial \theta_i}\right)\end{aligned}$$

Secondly, the derivative of the third term is

$$l = -\frac{1}{2}trace\left(\mathbf{V}_R^{-1}\mathbf{Y}\mathbf{Y}^T\right)$$

$$\frac{\partial l}{\partial \theta_i} = \frac{1}{2}trace\left(\mathbf{V}_R^{-1}\frac{\partial \mathbf{V}}{\partial \theta_i}\mathbf{V}_R^{-1}\mathbf{Y}\mathbf{Y}^T\right)$$

The last step is not easily proven, except for diligently applying the product rule and seeing a lot of terms cancel. Putting these two results together with the derivative of the normal likelihood gives us:

$$\begin{aligned}\frac{\partial(L_{ReML})}{\partial \theta_i} &= -\frac{P}{2}trace\left(\mathbf{V}^{-1}\frac{\partial \mathbf{V}}{\partial \theta_i}\right)\\ &\quad+\frac{1}{2}trace\left(\mathbf{V}_R^{-1}\frac{\partial \mathbf{V}}{\partial \theta_i}\mathbf{V}_R^{-1}\mathbf{Y}\mathbf{Y}^T\right)\\ &\quad+\frac{P}{2}trace\left(\mathbf{V}^{-1}\mathbf{X}\left(\mathbf{X}^T\mathbf{V}^{-1}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{V}^{-1}\frac{\partial \mathbf{V}}{\partial \theta_i}\right)\\ &= -\frac{P}{2}trace\left(\mathbf{V}_R^{-1}\frac{\partial \mathbf{V}}{\partial \theta_i}\right)+\frac{1}{2}trace\left(\mathbf{V}_R^{-1}\frac{\partial \mathbf{V}}{\partial \theta_i}\mathbf{V}_R^{-1}\mathbf{Y}\mathbf{Y}^T\right)\end{aligned}$$

2.9.5 Derivates for specific parameters

From the general term for the derivative of the log-likelihood, we can derive the specific expressions for each parameter. In general, we model the co-variance matrix of the data $\mathbf{V}$ as:

$$\begin{aligned}\mathbf{V} &= s\mathbf{Z}\mathbf{G}(\boldsymbol{\theta}_h)\mathbf{Z}^T + S\sigma_\epsilon^2\\ s &= exp(\theta_s)\\ \sigma_\epsilon^2 &= exp(\theta_\epsilon)\end{aligned}$$

Where θ_s is the signal scaling parameter, the θ_ϵ the noise parameter. We are using the exponential of the parameter, to ensure that the noise variance and the scaling will always be strictly positive. When taking the derivatives, we use the simple rule of $\partial exp(x)/\partial x = exp(x)$. Each model provides the partial derivaratives for $\mathbf{G}$ in respect to the model parameters (see above). From this we can easily obtain the derviative of $\mathbf{V}$

$$\frac{\partial \mathbf{V}}{\partial \theta_h} = \mathbf{Z}\frac{\partial \mathbf{G}(\boldsymbol{\theta_h})}{\partial \theta_h}\mathbf{Z}^T exp(\theta_s).$$

The derivate in respect to the noise parameter

$$\frac{\partial \mathbf{V}}{\partial \theta_\epsilon} = \mathbf{S}exp(\theta_\epsilon).$$

And in respect to the signal scaling parameter

$$\frac{\partial \mathbf{V}}{\partial \theta_s} = \mathbf{Z}\mathbf{G}(\boldsymbol{\theta}_h)\mathbf{Z}^T exp(\theta_s).$$

2.9.6 Conjugate Gradient descent

One way of optiminzing the likelihood is simply using the first derviative and performing a conjugate-gradient descent algorithm. For this, the routines *pcm_likelihoodIndivid* and *pcm_likelihoodGroup* return the negative log-likelihood, as well as a vector of the first derivatives of the negative log-likelihood in respect to the parameter. The implementation of conjugate-gradient descent we are using here based on Carl Rassmussen's excellent function *minimize*.

2.9.7 Newton-Raphson algorithm

A alternative to conjugate gradients, which can be considerably faster, are optimisation routines that exploit the matrix of second derivatives of the log-liklihood. The local curvature information is then used to "jump" to suspected bottom of the bowl of the likelihood surface. The negative expected second derivative of the restricted log-likelihood, also called Fisher-information can be calculated efficiently from terms that we needed to compute for the first derivative anyway:

$$\mathbf{F}_{i,j}(\theta) = -E\left[\frac{\partial^2}{\partial\theta_i\partial\theta_j}L_{ReML}\right] = \frac{P}{2}trace\left(\mathbf{V}_R^{-1}\frac{\partial\mathbf{V}}{\partial\theta_i}\mathbf{V}_R^{-1}\frac{\partial\mathbf{V}}{\partial\theta_j}\right).$$

The update then uses a slightly regularized version of the second derviate to compute the next update on the parameters.

$$\boldsymbol{\theta}^{u+1} = \boldsymbol{\theta}^u - \left(\mathbf{F}(\boldsymbol{\theta}^u) + \mathbf{I}\lambda\right)^{-1}\frac{\partial L_{ReML}}{\partial\boldsymbol{\theta}^u}.$$

Because the update can become unstable, we are regularising the Fisher information matrix by adding a small value to the diagonal, similar to a Levenberg regularisation for least-square problems. If the likelihood increased, λ is decreases, if the liklihood accidentially decreased, then we take a step backwards and increase λ. The algorithm is implemented in *pcm_NR* .

2.9.8 Choosing an optimisation algorithm

While the Newton-Raphson algorithm can be considerably faster for many problems, it is not always the case. Newton-Raphson usually arrives at the goal with many fewer steps than conjugate gradient descent, but on each step it has to calculate the matrix second derviatives, which grows in the square of the number of parameters . So for highly-parametrized models, the simple conjugate gradient algorithm is better. You can set for each model the desired algorithm by setting the field *M.fitAlgorithm = 'NR';* for Newton-Raphson and *M.fitAlgorithm = 'minimize';* for conjugate gradient descent. If no such field is given, then fitting function will call *M=pcm_optimalAlgorithm(M)* to obtain a guess of what will be the best algorithm for the problem. While this function provides a good heuristic strategy, it is recommended to try both and compare both the returned likelihood and time. Small differences in the likelihood (< 0.1) are due to different stopping criteria and should be of no concern. Larger differences can indicate failed convergence.

2.9.9 Acceleration of matrix inversion

When calculating the likelihood or the derviatives of the likelihood, the inverse of the variance-covariance has to be computed. Because this can become quickly very costly (especially if original time series data is to be fitted), we can exploit the special structure of $\mathbf{V}$ to speed up the computation:

$$\begin{aligned}\mathbf{V}^{-1} &= \left(s\mathbf{Z}\mathbf{G}\mathbf{Z}^T + \mathbf{S}\sigma_\varepsilon^2\right)^{-1} \\ &= \mathbf{S}^{-1}\sigma_\varepsilon^{-2} - \mathbf{S}^{-1}\mathbf{Z}\sigma_\varepsilon^{-2}\left(s^{-1}\mathbf{G}^{-1} + \mathbf{Z}^T\mathbf{S}^{-1}\mathbf{Z}\sigma_\varepsilon^{-2}\right)^{-1}\mathbf{Z}^T\mathbf{S}^{-1}\sigma_\varepsilon^{-2} \\ &= \left(\mathbf{S}^{-1} - \mathbf{S}^{-1}\mathbf{Z}\left(s^{-1}\mathbf{G}^{-1}\sigma_\varepsilon^2 + \mathbf{Z}^T\mathbf{S}^{-1}\mathbf{Z}\right)^{-1}\mathbf{Z}^T\mathbf{S}^{-1}\right)/\sigma_\varepsilon^2\end{aligned}$$

With pre-inversion of $\mathbf{S}$ (which can occur once outside of the iterations), we make a $N\times N$ matrix inversion into a *K{times}K* matrix inversion.

2.10 API reference

2.10.1 Dataset module

Definition of PCM Dataset class and subclasses @author: baihan, jdiedrichsen

class dataset.**Dataset**(*measurements*, *descriptors=None*, *obs_descriptors=None*, *channel_descriptors=None*)

Class for holding a single data set with observation descriptors.

Defines members that every class needs to have, but does not implement any interesting behavior. Defines a light version of the RSA Dataset class (which could be used instead)

Parameters

- **measurements** (*np.ndarray*) – n_obs x n_channel 2d-array,
- **descriptors** (*dict*) – descriptors (metadata)
- **obs_descriptors** (*dict*) – observation descriptors (all are array-like with shape = (n_obs,...))
- **channel_descriptors** (*dict*) – channel descriptors (all are array-like with shape = (n_channel,...))

2.10.2 Model module

Module that defines PCM Model classes and NoiseModel classes

Model Classes

class model.**Model**(*name*)

Abstract PCM Model Class

Parameters
name (*[str]*) – Name of the the model

get_prior()

Returns prior mean and precision

predict(*theta*)

Prediction function: Needs to be implemented

class model.**FixedModel**(*name*, *G*)

Fixed PCM with a rigid predicted G matrix and no parameters

Parameters

- **name** (*string*) – name of the particular model for indentification
- **G** (*numpy.ndarray*) – 2-dimensional array giving the predicted second moment

predict(*theta=None*)

Calculation of G

Returns

- **G** (*np.ndarray*) – 2-dimensional (K,K) array of predicted second moment
- **dG_dTheta** (*None*)

class model.**ComponentModel**(*name*, *Gc*)

Component model class G = sum (exp(theta_i) * Gc_i)

Parameters

- **name** (*string*) – name of the particular model for indentification
- **Gc** (*numpy.ndarray*) – 3-dimensional array with compoments of G

predict(*theta*)

Calculation of G

Parameters
theta (*numpy.ndarray*) – Vector of model parameters

Returns

- **G** (*np.ndarray*) – 2-dimensional (K,K) array of predicted second moment
- **dG_dTheta** (*np.ndarray*) – 3-d (n_param,K,K) array of partial matrix derivatives of G in respect to theta

set_theta0(*G_hat*)

Sets theta0 based on the crossvalidated second-moment

Parameters
G_hat (*numpy.ndarray*) – Crossvalidated estimate of G

class model.**FeatureModel**(*name*, *Ac*)

Feature model: A = sum (theta_i * Ac_i) G = A*A'

Parameters

- **name** (*string*) – name of the particular model for indentification
- **Ac** (*numpy.ndarray*) – 3-dimensional array with components of A

predict(*theta*)

Calculation of G

Parameters
theta (*np.ndarray*) – Vector of model parameters

Returns

- **G** (*np.ndarray*) – 2-dimensional (K,K) array of predicted second moment
- **dG_dTheta** (*np.ndarray*) – 3-d (n_param,K,K) array of partial matrix derivatives of G in respect to theta

class model.**CorrelationModel**(*name*, *within_cov=None*, *num_items=1*, *corr=None*, *cond_effect=False*)

Correlation model class for a fixed or flexible correlation model it models the correlation between different items across 2 experimental conditions. Using this paramaterization: var(x) = exp(theta_x)

var(y) = exp(theta_y)

cov(x,y) = sqrt(var(x)*var(y))* r

r = (exp(2*theta_z)-1)/(exp(2*theta_z)+1); % Fisher inverse

Parameters

- **name** (*string*) – name of the particular model for indentification
- **within_cov** (*numpy.ndarray or None*) – how to model within condition cov-ariance between items

- **num_items** (*int*) – Number of items within each condition

get_correlation(*theta*)

Returns the correlations from a set of fitted parameters

Parameters
theta (*numpy.ndarray*) – n_param vector or n_param x n_subj matrix of model parameters

Returns
correlations (*numpy.ndarray*) – Correlation value

predict(*theta*)

Calculation of G for a correlation model

Parameters
theta (*numpy.ndarray*) – Vector of model parameters

Returns

- **G** (*np.ndarray*) – 2-dimensional (K,K) array of predicted second moment
- **dG_dTheta** (*np.ndarray*) – 3-d (n_param,K,K) array of partial matrix derivatives of G in respect to theta

set_theta0(*G_hat*)

Sets theta0 based on the crossvalidated second-moment

Parameters
G_hat (*numpy.ndarray*) – Crossvalidated estimate of G

class model.**FreeModel**(*name*, *n_cond*)

Free model class: Second moment matrix is G = A*A', where A is a upper triangular matrix that is flexible

Parameters

- **name** (*string*) – name of the particular model for indentification
- **n_cond** (*int*) – number of conditions for free model

predict(*theta*)

Calculation of G

Parameters
theta (*numpy.ndarray*) – Vector of model parameters

Returns

- **G** (*np.ndarray*) – 2-dimensional (K,K) array of predicted second moment
- **dG_dTheta** (*np.ndarray*) – 3-d (n_param,K,K) array of partial matrix derivatives of G in respect to theta

set_theta0(*G_hat*)

Sets theta0 based on the crossvalidated second-moment

Parameters
G_hat (*numpy.ndarray*) – Crossvalidated estimate of G

Noise Model Classes

class model.**NoiseModel**

Abstract PCM Noise model class

class model.**IndependentNoise**

Simple Indepdennt noise model (i.i.d) the only parameter is the noise variance

derivative(*theta*, *n=0*)

Returns the derivative of S in respect to it's own parameters

Parameters

- **theta** (*[np.array]*) – Array like of noiseparamters
- **n** (*int, optional*) – Number of parameter to get derivate for. Defaults to 0.

Returns

d (*np-array*) – derivative of S in respective to theta

inverse(*theta*)

Returns S^{-1}

Parameters

theta (*[np.array]*) – Array like of noiseparamters

Returns

s (*double*) – Inverse of noise variance (scalar)

predict(*theta*)

Prediction function returns S - predicted noise covariance matrix

Parameters

theta (*[np.array]*) – Array like of noiseparamters

Returns

s (*double*) – Noise variance (for simplicity as a scalar)

set_theta0(*Y*, *Z*, *X=None*)

Makes an initial guess on noise paramters

Parameters

- **Y** (*[np.array]*) – Data
- **Z** (*[np.array]*) – Random Effects matrix
- **X** (*[np.array], optional*) – Fixed effects matrix.

class model.**BlockPlusIndepNoise**(*part_vec*)

This noise model uses correlated noise per partition (block) plus independent noise per observation For beta-values from an fMRI analysis, this is an adequate model

Parameters

part_vec (*[np.array]*) – vector indicating the block membership for each observation

derivative(*theta*, *n=0*)

Returns the derivative of S in respect to it's own parameters

Parameters

- **theta** (*np.array*) – Array like of noiseparamters
- **n** (*int, optional*) – Number of parameter to get derivate for. Defaults to 0.

Returns
d (*np.array*) – derivative of S in respective to theta

`inverse`(*theta*)

Returns S^{-1}

Parameters
theta (*np.array*) – Array like of noiseparamters

Returns
iS (*np.array*) – Inverse of noise covariance

`predict`(*theta*)

Prediction function returns S - predicted noise covariance matrix

Parameters
theta (*[np.array]*) – Array like of noiseparamters

Returns
s (*np.array*) – Noise covariance matrix

`set_theta0`(*Y*, *Z*, *X=None*)

Makes an initial guess on noise parameters :param Y: Data :type Y: [np.array] :param Z: Random Effects matrix :type Z: [np.array] :param X: Fixed effects matrix. :type X: [np.array], optional

2.10.3 Inference module

Inference module for PCM toolbox with main functionality for model fitting and evaluation. @author: jdiedrichsen

`inference.fit_model_group`(*Data*, *M*, *fixed_effect='block'*, *fit_scale=False*, *scale_prior=1000.0*, *noise_cov=None*, *algorithm=None*, *optim_param={}*, *theta0=None*, *verbose=True*, *return_second_deriv=False*)

Fits PCM models(s) to a group of subjects

The model parameters are (by default) shared across subjects. Scale and noise parameters are individual for each subject. Some model parameters can also be made individual by setting M.common_param

Parameters

- **Data** (*list of pcm.Datasets*) – List data set has partition and condition descriptors
- **M** (*pcm.Model or list of pcm.Models*) – Models to be fitted on the data sets. Optional field M.common_param indicates which model parameters are common to the group (True) and which ones are fit individually (False)
- **effect** (*fixed*) – None, 'block', or nd-array / list of nd-arrays. Default ('block') add an intercept for each partition
- **fit_scale** (*bool*) – Fit a additional scale parameter for each subject? Default is set to False.
- **scale_prior** (*float*) – Prior variance for log-normal prior on scale parameter
- **algorithm** (*string*) – Either 'newton' or 'minimize' - provides over-write for model specific algorithms
- **noise_cov** – None (i.i.d), 'block', or optional specific covariance structure of the noise
- **optim_param** (*dict*) – Additional paramters to be passed to the optimizer
- **theta0** (*list of np.arrays*) – List of starting values (same format as return argument theta)

- **verbose** (*bool*) – Provide printout of progress? Default: True

Returns

- **T** (*pandas.dataframe*) – Dataframe with the fields: SN: Subject number likelihood: log-likelihood scale: Scale parameter (if fitscale = 1)-exp(theta_s) noise: Noise parameter-exp(theta_eps) iterations: Number of interations for model fit time: Elapsed time in sec
- **theta** (*list of np.arrays*) – List of estimated model parameters each one is a vector with #num_commonparams + #num_singleparams x #numSubj elements
- **G_pred** (*list of np.arrays*) – List of estimated G-matrices under the model

inference.**fit_model_group_crossval**(*Data*, *M*, *fixed_effect='block'*, *fit_scale=False*, *scale_prior=1000.0*, *noise_cov=None*, *algorithm=None*, *optim_param={}*, *theta0=None*, *verbose=True*)

Fits PCM model(sto N-1 subjects and evaluates the likelihood on the Nth subject.

Only the common model parameters are shared across subjects.The scale and noise parameters are still fitted to each subject. Some model parameters can also be made individual by setting M.common_param to False

Parameters

- **Data** (*list of pcm.Datasets*) – List data set has partition and condition descriptors
- **M** (*pcm.Model or list of pcm.Models*) – Models to be fitted on the data sets. Optional field M.common_param indicates which model parameters are common to the group (True) and which ones are fit individually (False)
- **effect** (*fixed*) – None, 'block', or nd-array. Default ('block') add an intercept for each partition
- **fit_scale** (*bool*) – Fit a additional scale parameter for each subject? Default is set to False.
- **scale_prior** (*float*) – Prior variance for log-normal prior on scale parameter
- **algorithm** (*string*) – Either 'newton' or 'minimize' - provides over-write for model specific algorithms
- **noise_cov** – None (i.i.d), 'block', or optional specific covariance structure of the noise
- **optim_param** (*dict*) – Additional paramters to be passed to the optimizer
- **theta0** (*list of np.arrays*) – List of starting values (same format as return argument theta)
- **verbose** (*bool*) – Provide printout of progress? Default: True

Returns

- **T** (*pandas.dataframe*) – Dataframe with the fields: SN: Subject number likelihood: log-likelihood scale: Scale parameter (if fitscale = 1)-exp(theta_s) noise: Noise parameter-exp(theta_eps) iterations: Number of interations for model fit time: Elapsed time in sec
- **theta** (*list of np.arrays*) – List of estimated model parameters - common group parameters come from the training data, individual parameters from the testing data
- **G_pred** (*list of np.arrays*) – List of estimated G-matrices under the model

inference.**fit_model_individ**(*Data*, *M*, *fixed_effect='block'*, *fit_scale=False*, *scale_prior=1000.0*, *noise_cov=None*, *algorithm=None*, *optim_param={}*, *theta0=None*, *verbose=True*, *return_second_deriv=False*)

Fits Models to a data set individually.

The model parameters are all individually fit.

Parameters

- **Data** (*pcm.Dataset or list of pcm.Datasets*) – List data set has partition and condition descriptors
- **M** (*pcm.Model or list of pcm.Models*) – Models to be fitted on the data sets
- **effect** (*fixed*) – None, 'block', or nd-array. Default ('block') adds an intercept for each partition
- **fit_scale** (*bool*) – Fit a additional scale parameter for each subject? Default is set to False.
- **scale_prior** (*float*) – Prior variance for log-normal prior on scale parameter
- **algorithm** (*string*) – Either 'newton' or 'minimize' - provides over-write for model specific algorithms
- **noise_cov** – None (i.i.d), 'block', or optional specific covariance structure of the noise
- **optim_param** (*dict*) – Additional paramters to be passed to the optimizer
- **theta0** (*list of np.arrays*) – List of starting values (same format as return argument theta)
- **verbose** (*bool*) – Provide printout of progress? Default: True

Returns

- **T** (*pandas.dataframe*) – Dataframe with the fields: SN: Subject number likelihood: log-likelihood scale: Scale parameter (if fitscale = 1)-exp(theta_s) noise: Noise parameter-exp(theta_eps) run: Run parameter (if run = 'random') iterations: Number of interations for model fit time: Elapsed time in sec
- **theta** (*list of np.arrays*) – List of estimated model parameters, each a #params x #numSubj np.array
- **G_pred** (*list of np.arrays*) – List of estimated G-matrices under the model

inference.**get_scale0**(*G*, *G_hat*)

" Get approximate (log-)scaling parameter between predicted G and estimated G_hat

Parameters

- **G** (*numpy.ndarray0*) – Predicted G matrix by the model
- **G_hat** (*numpy.ndarry0*) – Directly estimated G from the data

Returns

scale0 – log-scaling parameter

inference.**group_to_individ_param**(*theta*, *M*, *n_subj*)

Takes a vector of group parameters and rearranges them To make it conform to theta you would get back from a individual fit

Parameters

- **theta** (*nd.array*) – Vector of group parameters
- **M** (*pcm.Model*) – PCM model

- **n_subj** (*int*) – Number of subjects

Returns

theta_indiv (*ndarray*) – n_params x n_subj Matrix of group parameters

inference.**likelihood_group**(*theta*, *M*, *YY*, *Z*, *X=None*, *Noise=<PcmPy.model.IndependentNoise object>*, *n_channel=1*, *fit_scale=True*, *scale_prior=1000.0*, *return_deriv=0*, *return_individ=False*)

Negative Log-Likelihood of group data and derivative in respect to the parameters

Parameters

- **theta** (*np.array*) – Vector of (log-)model parameters consisting of common model parameters (M.n_param or sum of M.common_param) + participant-specific parameters (interated by subject): individ model param (not in common_param), scale parameter,noise parameters
- **M** (*pcm.Model*) – Model object
- **YY** (*List of np.arrays*) – List of NxN Matrix of outer product of the activity data (Y*Y')
- **Z** (*List of 2d-np.array*) – NxQ Design matrix - relating the trials (N) to the random effects (Q)
- **X** (*List of np.array*) – Fixed effects design matrix - will be accounted for by ReML
- **Noise** (*List of pcm.Noisemodel*) – Pcm-noise model (default: IndependentNoise)
- **n_channel** (*List of int*) – Number of channels
- **fit_scale** (*bool*) – Fit a scaling parameter for the model (default is False)
- **scale_prior** (*float*) – Prior variance for log-normal prior on scale parameter
- **return_deriv** (*int*) – 0: Only return negative likelihood 1: Return first derivative 2: Return first and second derivative (default)
- **return_individ** (*bool*) – return individual likelihoods instead of group likelihood
- **return_deriv** – 0:None, 1:First, 2: second

Returns

- **negloglike** – Negative log-likelihood of the data under a model
- **dLdtheta** (*1d-np.array*) – First derivative of negloglike in respect to the parameters
- **ddLdtheta2** (*2d-np.array*) – Second derivative of negloglike in respect to the parameters

inference.**likelihood_individ**(*theta*, *M*, *YY*, *Z*, *X=None*, *Noise=<PcmPy.model.IndependentNoise object>*, *n_channel=1*, *fit_scale=False*, *scale_prior=1000.0*, *return_deriv=0*)

Negative Log-Likelihood of the data and derivative in respect to the parameters

Parameters

- **theta** (*np.array*) – Vector of (log-)model parameters - these include model, signal, and noise parameters
- **M** (*PcmPy.model.Model*) – Model object with predict function
- **YY** (*2d-np.array*) – NxN Matrix of outer product of the activity data (Y*Y')
- **Z** (*2d-np.array*) – NxQ Design matrix - relating the trials (N) to the random effects (Q)
- **X** (*np.array*) – Fixed effects design matrix - will be accounted for by ReML
- **Noise** (*pcm.Noisemodel*) – Pcm-noise mode to model block-effects (default: Indepenent-Noise)

- **n_channel** (*int*) – Number of channels
- **fit_scale** (*bool*) – Fit a scaling parameter for the model (default is False)
- **scale_prior** (*float*) – Prior variance for log-normal prior on scale parameter
- **return_deriv** (*int*) – 0: Only return negative loglikelihood 1: Return first derivative 2: Return first and second derivative (default)

Returns

- **negloglike** (*double*) – Negative log-likelihood of the data under a model
- **dLdtheta** (*1d-np.array*) – First derivative of negloglike in respect to the fitted parameters
- **ddLdtheta2** (*2d-np.array*) – Second derivative of negloglike in respect to the fitted parameters

inference.**sample_model_group**(*Data*, *M*, *fixed_effect='block'*, *fit_scale=False*, *scale_prior=1000.0*, *noise_cov=None*, *sample_param={'burn_in': 100, 'n_samples': 10000}*, *theta0=None*, *verbose=True*, *proposal_sd=None*)

Approximates the posterior of the parameters of a group model using MCMC sampling

The model parameters are (by default) shared across subjects. Scale and noise parameters are individual for each subject. Some model parameters can also be made individual by setting M.common_param

Parameters

- **Data** (*list of pcm.Datasets*) – List data set has partition and condition descriptors
- **M** (*pcm.Model*) – Models to sampled
- **effect** (*fixed*) – None, 'block', or nd-array / list of nd-arrays. Default ('block') add an intercept for each partition
- **fit_scale** (*bool*) – Fit a additional scale parameter for each subject? Default is set to False.
- **scale_prior** (*float*) – Prior variance for log-normal prior on scale parameter
- **noise_cov** – None (i.i.d), 'block', or optional specific covariance structure of the noise
- **sample_param** (*dict*) – Additional paramters to be passed to MCMC sampler
- **theta0** (*np.array*) – starting values

Returns

- **theta** (*np.array*) – Sampled parameters
- **l** (*np.array*) – Log-likelihood corresponding to the sampled parameters
- **G_pred** (*list of np.arrays*) – List of estimated G-matrices under the model

inference.**set_up_fit**(*Data*, *fixed_effect='block'*, *noise_cov=None*)

Utility routine pre-calculates and sets design matrices, etc for the PCM fit

Parameters

- **Data** (*pcm.dataset*) – Contains activity data (measurement), and obs_descriptors partition and condition
- **fixed_effect** – Can be None, 'block', or a design matrix. 'block' includes an intercept for each partition.
- **noise_cov** – Can be None: (i.i.d noise), 'block': a common noise paramter or a List of noise covariances for the different partitions

Returns

- **Z** (*np.array*) – Design matrix for random effects
- **X** (*np.array*) – Design matrix for fixed effects
- **YY** (*np.array*) – Quadratic form of the data (Y Y')
- **Noise** (*pcm.model.NoiseModel*) – Noise model
- **G_hat** (*np.array*) – Crossvalidated estimate of second moment of U

inference.**set_up_fit_group**(*Data*, *fixed_effect='block'*, *noise_cov=None*)

Pre-calculates and sets design matrices, etc for the PCM fit for a full group

Parameters

- **Data** (*list of pcm.dataset*) – Contains activity data (measurement), and obs_descriptors partition and condition
- **fixed_effect** – Can be None, 'block', or a design matrix. 'block' includes an intercept for each partition.
- **noise_cov** – Can be None: (i.i.d noise), 'block': a common noise paramter or a List of noise covariances for the different partitions

Returns

- **Z** (*np.array*) – Design matrix for random effects
- **X** (*np.array*) – Design matrix for fixed effects
- **YY** (*np.array*) – Quadratic form of the data (Y Y')
- **Noise** (*NoiseModel*) – Noise model
- **G_hat** (*np.array*) – Crossvalidated estimate of second moment of U

2.10.4 Optimization module

Optimization module for PCM toolbox with main functionality for model fitting. @author: jdiedrichsen

optimize.**best_algorithm**(*M*, *algorithm=None*)

Parameters

- **M** (*List of pcm.Model*) –
- **algorithm** (*string*) – Overwrite for algorithm

optimize.**mcmc**(*th0*, *likelihood_fcn*, *proposal_sd=0.1*, *burn_in=100*, *n_samples=1000*, *verbose=1*)

Implement Markov Chain Monte Carlo sampling for PCM models Metropolis-Hastings algorithm with adaptive proposal distribution

optimize.**newton**(*theta0*, *lossfcn*, *max_iter=80*, *thres=0.0001*, *hess_reg=0.0001*, *regularization='sEig'*, *verbose=0*, *fit_param=None*)

Minimize a loss function using Newton-Raphson with automatic regularization

Parameters

- **theta** (*np.array*) – Vector of parameter starting values
- **lossfcn** (*fcn*) – Handle to loss function that needs to return a) Loss (Negative log-likelihood) b) First derivative of the Loss c) Expected second derivative of the loss

- **max_iter** (*int*) – Maximal number of iterations (default: 80)
- **thres** (*float*) – Threshold for change in Loss function (default: 1e-4)
- **hess_reg** (*float*) – starting regulariser on the Hessian matrix (default 1e-4)
- **regularization** (*string*) – 'L': Levenberg 'LM': Levenberg-Marquardt 'sEig':smallest Eigenvalue (default)
- **verbose** (*int*) – 0: No feedback, 1:Important warnings 2:full feedback regularisation
- **fit_param** (*Logical*) – If provided, it will only fit the parameters indicated

Returns

- **theta** (*np.array*) – theta at minimum
- **loss** (*float*) – minimal loss
- **info** (*dict*) – Dictionary with more information abuot the fit

2.10.5 Regression module

Regression module contains bare-bones version of the PCM toolbox that can be used to tune ridge/Tikhonov coefficients in the context of tranditional regression models. No assumption are made about independent data partitions.

class regression.**RidgeDiag**(*components*, *theta0=None*, *fit_intercept=True*, *noise_model=<PcmPy.model.IndependentNoise object>*)

Class for Linear Regression with Tikhonov (L2) regularization. The regularization matrix for this class is diagnonal, with groups of elements along the diagonal sharing the same Regularisation factor.

Constructor

Parameters:

components (1d-array like)
Indicator to which column of design matrix belongs to which group

theta0 (1d np.array)
Vector of of starting values for optimization

fit_intercept (Boolean)
Should intercept be added to fixed effects (Dafault: true)

noise_model (pcm.model.NoiseModel)
Model specifying the full-rank noise effects

fit(*Z*, *Y*, *X=None*)

Estimates the regression parameters, given a specific regularization :param Z: Design matrix for random effects NxQ :type Z: 2d-np.array :param Y: NxP Matrix of data :type Y: 2d-np.array :param X: Fixed effects design matrix - will be accounted for by ReML :type X: np.array

Returns
self – Model with fitted parameters

optimize_regularization(*Z*, *Y*, *X=None*, *optim_param={}*, *like_fcn='auto'*)

Optimizes the hyper parameters (regularisation) of the regression mode :param Z: Design matrix for random effects NxQ :type Z: 2d-np.array :param Y: NxP Matrix of data :type Y: 2d-np.array :param X: Fixed effects design matrix - will be accounted for by ReML :type X: np.array :param optim_parameters: parameters for the optimization routine :type optim_parameters: dictionary of parameters

Returns
self – Model with fitted parameters

predict(*Z*, *X=None*)

Predicts new data based on a fitted model :param Z: Design matrix for random effects NxQ :type Z: 2d-np.array :param Y: NxP Matrix of data :type Y: 2d-np.array :param X: Fixed effects design matrix - will be accounted for by ReML :type X: np.array

Returns
self – Model with fitted parameters

regression.**compute_iVr**(*Z*, *G*, *iS*, *X=None*)

Fast inverse of V matrix using the matrix inversion lemma

Parameters

- **Z** (*2d-np.array*) – Design matrix for random effects NxQ
- **G** (*1d or 2d-np.array*) – Q x Q Matrix: variance of random effect
- **iS** (*scalar or NxN matrix*) – Inverse variance of noise matrix
- **X** (*2d-np.array*) – Design matrix for random effects

Returns

- **iV** (*2d-np.array*) – inv(Z*G*Z' + S);
- **iVr** (*2d-np.array*) – iV - iV * X inv(X' * iV *X) * X' *iV
- **ldet** (*scalar*) – log(det(iV))

regression.**likelihood_diagYTY_ZZT**(*theta*, *Z*, *Y*, *comp*, *X=None*, *Noise=<PcmPy.model.IndependentNoise object>*, *return_deriv=0*)

Negative Log-Likelihood of the data and derivative in respect to the parameters. This function is faster when N>>P.

Parameters

- **theta** (*np.array*) – Vector of (log-)model parameters: These include model, signal and noise parameters
- **Z** (*2d-np.array*) – Design matrix for random effects NxQ
- **Y** (*2d-np.array*) – NxP Matrix of data
- **comp** (*1d-np.array or list*) – Q-length: Indicates for each column of Z, which theta will be used for the weighting
- **X** (*np.array*) – Fixed effects design matrix - will be accounted for by ReML
- **Noise** (*pcm.Noisemodel*) – Pcm-noise mode to model block-effects (default: Indentity)
- **return_deriv** (*int*) – 0: Only return negative loglikelihood 1: Return first derivative 2: Return first and second derivative (default)

Returns

- **negloglike** – Negative log-likelihood of the data under a model
- **dLdtheta** (*1d-np.array*) – First derivative of negloglike in respect to the parameters
- **ddLdtheta2** (*2d-np.array*) – Second derivative of negloglike in respect to the parameters

regression.**likelihood_diagYYT_ZTZ**(*theta*, *Z*, *YY*, *num_var*, *comp*, *X=None*, *Noise=<PcmPy.model.IndependentNoise object>*, *return_deriv=0*)

Negative Log-Likelihood of the data and derivative in respect to the parameters. This function is faster when P>N

Parameters

- **theta** (*np.array*) – Vector of (log-)model parameters: These include model, signal and noise parameters
- **Z** (*2d-np.array*) – Design matrix for random effects NxQ
- **YY** (*2d-np.array*) – NxN Matrix: Outer product of the data
- **num_var** (*int*) – Number of variables in data set (columns of Y)
- **comp** (*1d-np.array or list*) – Q-length: Indicates for each column of Z, which theta will be used for the weighting
- **X** (*np.array*) – Fixed effects design matrix - will be accounted for by ReML
- **Noise** (*pcm.Noisemodel*) – Pcm-noise mode to model block-effects (default: Indentity)
- **return_deriv** (*int*) – 0: Only return negative loglikelihood 1: Return first derivative 2: Return first and second derivative (default)

Returns

- **negloglike** – Negative log-likelihood of the data under a model
- **dLdtheta** (*1d-np.array*) – First derivative of negloglike in respect to the parameters
- **ddLdtheta2** (*2d-np.array*) – Second derivative of negloglike in respect to the parameters

regression.**likelihood_diagYYT_ZZT**(*theta*, *Z*, *YY*, *num_var*, *comp*, *X=None*, *Noise=<PcmPy.model.IndependentNoise object>*, *return_deriv=0*)

Negative Log-Likelihood of the data and derivative in respect to the parameters. This function is faster when P>N

Parameters

- **theta** (*np.array*) – Vector of (log-)model parameters: These include model, signal and noise parameters
- **Z** (*2d-np.array*) – Design matrix for random effects NxQ
- **YY** (*2d-np.array*) – NxN Matrix: Outer product of the data
- **num_var** (*int*) – Number of variables in data set (columns of Y)
- **comp** (*1d-np.array or list*) – Q-length: Indicates for each column of Z, which theta will be used for the weighting
- **X** (*np.array*) – Fixed effects design matrix - will be accounted for by ReML
- **Noise** (*pcm.Noisemodel*) – Pcm-noise mode to model block-effects (default: Indentity)
- **return_deriv** (*int*) – 0: Only return negative loglikelihood 1: Return first derivative 2: Return first and second derivative (default)

Returns

- **negloglike** – Negative log-likelihood of the data under a model
- **dLdtheta** (*1d-np.array*) – First derivative of negloglike in respect to the parameters
- **ddLdtheta2** (*2d-np.array*) – Second derivative of negloglike in respect to the parameters

2.10.6 Matrix module

Collection of different utility Matrices

@author: jdiedrichsen

matrix.**centering**(*size*)

generates a centering matrix

Parameters
size (*int*) – size of the center matrix

Returns
centering_matrix (*numpy.ndarray*) – size * size

matrix.**indicator**(*index_vector*, *positive=False*)

Indicator matrix with one column per unique element in vector

Parameters

- **index_vector** (*numpy.ndarray*) – n_row vector to code - discrete values (one dimensional)
- **positive** (*bool*) – should the function ignore zero negative entries in the index_vector? Default: false

Returns
indicator_matrix (*numpy.ndarray*) – nrow x nconditions indicator matrix

matrix.**pairwise_contrast**(*index_vector*)

Contrast matrix with one row per unqiue pairwise contrast

Parameters
index_vector (*numpy.ndarray*) – n_row vector to code discrete values (one dimensional)

Returns
contrast matrix (*numpy.ndarray*) – n_values * (n_values-1)/2 x n_row

2.10.7 Util module

Collection of different utility functions

@author: jdiedrichsen

util.**G_to_dist**(*G*)

Transforms a second moment matrix into a squared Euclidean matrix (mostly for visualization)

Parameters
G (*ndarray*) – 2d or 3d array of second moment matrices

util.**check_grad**(*fcn*, *theta0*, *delta*)

Checks the gradient of a function around a value for theta

Parameters

- **fcn** (*function*) – needs to return criterion and derivative
- **theta0** (*ndarray*) – Vector of parameters

util.**classical_mds**(*G*, *contrast=None*, *align=None*, *thres=0*)

Calculates a low-dimensional projection of a G-matrix That preserves the relationship of different conditions Equivalent to classical MDS. If contrast is given, the method becomes equivalent to dPCA, as it finds the representation that maximizes the variance acording to this contrast. Developement: If *align* is given, it performs Procrustes alignment of the result to a given V within the found dimension

Parameters

- **G** (*ndarray*) – KxK second moment matrix
- **contrast** (*ndarray*) – Contrast matrix to optimize for. Defaults to None.
- **align** (*ndarry*) – A different loading matrix to which to align
- **thres** (*float*) – Cut off eigenvalues under a certain value

Returns

- **W** (*ndarray*) – Loading of the K different conditions on main axis
- **Glam** (*ndarray*) – Variance explained by each axis

util.**est_G_crossval**(*Y*, *Z*, *part_vec*, *X=None*, *S=None*)

Obtains a crossvalidated estimate of G Y = Z @ U + X @ B + E, where var(U) = G

Parameters

- **Y** (*numpy.ndarray*) – Activity data
- **Z** (*numpy.ndarray*) – 2-d: Design matrix for conditions / features U 1-d: condition vector
- **part_vec** (*numpy.ndarray*) – Vector indicating the partition number
- **X** (*numpy.ndarray*) – Fixed effects to be removed
- **S** (*numpy.ndarray*) –

Returns

- **G_hat** (*numpy.ndarray*) – n_cond x n_cond matrix
- **Sig** (*numpy.ndarray*) – n_cond x n_cond noise estimate per block

util.**make_pd**(*G*, *thresh=1e-10*)

Enforces that G is semi-positive definite by setting small eigenvalues to minimal value

Parameters

- **G** (*square 2d-np.array*) – estimated KxK second momement matrix
- **thresh** (*float*) – threshold for increasing small eigenvalues

Returns

Gpd (*square 2d-np.array*) – semi-positive definite version of G

2.10.8 Simulation module

Functions for data simulation from PCM-models @author: jdiedrichsen

sim.**make_dataset**(*model*, *theta*, *cond_vec*, *n_channel=30*, *n_sim=1*, *signal=1*, *noise=1*, *signal_cov_channel=None*, *noise_cov_channel=None*, *noise_cov_trial=None*, *use_exact_signal=False*, *use_same_signal=False*, *part_vec=None*, *rng=None*)

Simulates a fMRI-style data set

Parameters

- **model** (*PcmPy.Model*) – the model from which to generate data
- **theta** (*numpy.ndarray*) – vector of parameters (one dimensional)
- **cond_vec** (*numpy.ndarray*) – RSA-style model: vector of experimental conditions Encoding-style: design matrix (n_obs x n_cond)
- **n_channel** (*int*) – Number of channels (default = 30)
- **n_sim** (*int*) – Number of simulation with the same signal (default = 1)
- **signal** (*float*) – Signal variance (multiplied by predicted G)
- **signal_cov_channel** (*numpy.ndarray*) – Covariance matrix of signal across channels
- **noise** (*float*) – Noise variance
- **noise_cov_channel** (*numpy.ndarray*) – Covariance matrix of noise (default = identity)
- **noise_cov_trial** (*numpy.ndarray*) – Covariance matrix of noise across trials
- **use_exact_signal** (*bool*) – Makes the signal so that G is exactly as specified (default: False)
- **use_same_signal** (*bool*) – Uses the same signal for all simulation (default: False)
- **part_vec** (*np.array*) – Optional partition that is added to the data set obs_descriptors
- **rng** (*np.random.default_rng*) – Optional random number generator object to pass specific state

Returns

data (*list*) – List of pyrsa.Dataset with obs_descriptors

sim.**make_design**(*n_cond*, *n_part*)

Makes simple fMRI design with n_cond, each measures n_part times

Parameters

- **n_cond** (*int*) – Number of conditions
- **n_part** (*int*) – Number of partitions

Returns

- **Tuple** (*cond_vec, part_vec*)
- **cond_vec** (*np.ndarray*) – n_obs vector with condition
- **part_vec** (*np.ndarray*) – n_obs vector with partition

sim.**make_signal**(*G*, *n_channel*, *make_exact=False*, *chol_channel=None*, *rng=None*)

Generates signal exactly with a specified second-moment matrix (G)

Parameters

- **G** (*np.array*) – desired second moment matrix (ncond x ncond)
- **n_channel** (*int*) – Number of channels
- **make_exact** (*bool*) – Make the signal so the second moment matrix is exact (default: False)
- **chol_channel** – Cholensky decomposition of the signal covariance matrix (default: None - makes signal i.i.d.)
- **rng** (*np.random.default_rng*) – Optional random number generator object to pass specific state

Returns

np.array (*n_cond x n_channel*) – random signal

2.11 References

- Cai, M.B., Schuck, N.W., Pillow, J., and Niv, Y. (2016). A Bayesian method for reducing bias in neural representational similarity analysis. In Advances in Neural Information Processing Systems, pp. 4952–4960.
- Diedrichsen, J., Ridgway, G.R., Friston, K.J., and Wiestler, T. (2011). Comparing the similarity and spatial structure of neural representations: a pattern-component model. Neuroimage 55, 1665–1678.
- Diedrichsen, J. (2019). Representational models and the feature fallacy. In The Cognitive Neurosciences, M.S. Gazzaniga, G.R. Mangun, and D. Poepple, eds. (Cambridge, MA: MIT Press), p.
- Diedrichsen, J., and Kriegeskorte, N. (2017). Representational models: A common framework for understanding encoding, pattern-component, and representational-similarity analysis. PLOS Comput. Biol. 13, e1005508.
- Diedrichsen, J., Yokoi, A., and Arbuckle, S.A. (2018). Pattern component modeling: A flexible approach for understanding the representational structure of brain activity patterns. Neuroimage 180, 119–133.
- Ejaz, N., Hamada, M., and Diedrichsen, J. (2015). Hand use predicts the structure of representations in sensorimotor cortex. Nat Neurosci 18, 1034–1040.
- Khaligh-Razavi, S.M., and Kriegeskorte, N. (2014). Deep supervised, but not unsupervised, models may explain IT cortical representation. PLoS Comput Biol 10, e1003915.
- Kriegeskorte, N., and Diedrichsen, J. (2016). Inferring brain-computational mechanisms with models of activity measurements. Philos. Trans. R. Soc. B Biol. Sci. 371.
- Kriegeskorte, N., and Diedrichsen, J. (2019). Peeling the Onion of Brain Representations. Annu. Rev. Neurosci. 42, 407–432.
- Kriegeskorte, N., Mur, M., and Bandettini, P. (2008). Representational similarity analysis - connecting the branches of systems neuroscience. Front Syst Neurosci 2, 4.
- Kriegeskorte, N., Mur, M., Ruff, D.A., Kiani, R., Bodurka, J., Esteky, H., Tanaka, K., and Bandettini, P.A. (2008). Matching categorical object representations in inferior temporal cortex of man and monkey. Neuron 60, 1126–1141.
- Naselaris, T., Kay, K.N., Nishimoto, S., and Gallant, J.L. (2011). Encoding and decoding in fMRI. Neuroimage 56, 400–410.
- Nili, H., Wingfield, C., Walther, A., Su, L., Marslen-Wilson, W., and Kriegeskorte, N. (2014). A toolbox for representational similarity analysis. PLoS Comput Biol 10, e1003553.
- Walther, A., Nili, H., Ejaz, N., Alink, A., Kriegeskorte, N., and Diedrichsen, J. (2016). Reliability of dissimilarity measures for multi-voxel pattern analysis. Neuroimage 137, 188–200.
- Yokoi, A., and Diedrichsen, J. (2019). Neural Organization of Hierarchical Motor Sequence Representations in the Human Neocortex. Neuron.

- Yokoi, A., Arbuckle, S.A., and Diedrichsen, J. (2018). The Role of Human Primary Motor Cortex in the Production of Skilled Finger Sequences. J. Neurosci. 38, 1430–1442.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

P

R

S

U