# Standard Gaussian Hidden Markov Model

This notebook covers the fundamental procedures for training and examining a Gaussian Hidden Markov Model (HMM). The model is designed for analyzing a single set of time series data, such as neuroimaging or electrophysiological recordings. If you need to model interactions between two sets of time series, you can look at *Gaussian-Linear Hidden Markov Model* provided in the toolbox.

**Note:** Due to rendering issues when viewing this notebook through github, internal links, like the table of contents, may not work correctly. To ensure that the notebook renders correctly, you can view it through this link.

Authors: Christine Ahrends christine.ahrends@cfin.au.dk

## Outline



## Background

The HMM is a generative probabilistic model that assumes that an observed timeseries (e.g., neuroimaging or electrophysiological recordings) were generated by a sequence of hidden "states". In the Gaussian HMM, we model states as Gaussian distributions, so we assume that the observations $Y$ at time point $t$ were generated by a Gaussian distribution with parameters $\mu$ and $\Sigma$ when state $k$ is active, i.e.:

$$Y_t \sim N(\mu^k, \Sigma^k)$$

Additionally, the HMM estimates the probabilities $\theta$ of transitioning between each pair of states, i.e., the probability that the currently active state at time point $t$ is $k$ given that the state at the previous timepoint $t-1$ was $l$:

$$P(s_t = k | s_{t-1} = l) = \theta_{k,l}$$

And the probability $\pi$ that a segment of the timeseries starts with state $k$:

$$P(s_0 = k) = \pi_k$$

When we fit the model to the observations, we aim to estimate the parameters of the prior distributions for these parameters ($\mu$, $\Sigma$, $\theta$, and $\pi$) using variational inference.

We define the posterior estimates as

$$\gamma_{t,k} := P(s_t = k | s_{>t}, s_{<t}, Y)$$
$$\xi_{t,k,l} := P(s_t = k, s_{t-1} = l | s_{>t}, s_{<t-1}, Y)$$

where $\gamma$ are the probabilities of state $k$ being active at time point $t$ (the state time courses) and $\xi$ are the joint state probabilities. Instead of the state time courses, we can also use the Viterbi path, a discrete representation of which state is active at each time point.

A common application for the standard Gaussian HMM is the estimation of time-varying amplitude and functional connectivity in fMRI recordings (e.g., Vidaurre et al., 2017).

# Example: Modelling time-varying amplitude and functional connectivity in fMRI recordings

We will now go through an example illustrating how to fit and inspect a standard Gaussian HMM. The example uses simulated data that can be found in the `example_data` folder. The data were generated to resemble fMRI timeseries, and our goal is to estimate time-varying amplitude and functional connectivity (FC) for a group of subjects. Imagine that the data were recorded from 20 different

subjects. Each subject has been recorded for 1,000 timepoints and their timeseries were extracted in a parcellation with 50 brain regions. The data *Y* here thus has dimensions ((20 subjects * 1000 timepoints), 50 brain regions). We use the *indices* array to specify where in the timeseries each of the 20 subjects' session starts and ends.

## Preparation

If you dont have the **GLHMM-package** installed, then run the following command in your terminal:

`pip install glhmm`

**Import libraries**

Let's start by importing the required libraries and modules.

```
pip install glhmm
```

```
Collecting glhmm
  Downloading glhmm-1.1.2-py3-none-any.whl.metadata (982 bytes)
Requirement already satisfied: scipy in /usr/local/lib/python3.12/dist-packages (from glhmm) (1.16.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from glhmm) (2.0.2)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.12/dist-packages (from glhmm) (1.6.1)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-packages (from glhmm) (3.10.0)
Requirement already satisfied: numba in /usr/local/lib/python3.12/dist-packages (from glhmm) (0.60.0)
Requirement already satisfied: seaborn in /usr/local/lib/python3.12/dist-packages (from glhmm) (0.13.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.12/dist-packages (from glhmm) (2.2.2)
Collecting igraph (from glhmm)
  Downloading igraph-1.0.0-cp39-abi3-manylinux_2_28_x86_64.whl.metadata (4.4 kB)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from glhmm) (4.67.1)
Requirement already satisfied: scikit-image in /usr/local/lib/python3.12/dist-packages (from glhmm) (0.25.2)
Requirement already satisfied: statsmodels in /usr/local/lib/python3.12/dist-packages (from glhmm) (0.14.5)
Requirement already satisfied: h5py in /usr/local/lib/python3.12/dist-packages (from glhmm) (3.15.1)
Requirement already satisfied: nibabel in /usr/local/lib/python3.12/dist-packages (from glhmm) (5.3.2)
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from glhmm) (2.32.4)
Collecting nilearn (from glhmm)
  Downloading nilearn-0.12.1-py3-none-any.whl.metadata (9.9 kB)
Collecting texttable>=1.6.2 (from igraph->glhmm)
  Downloading texttable-1.7.0-py2.py3-none-any.whl.metadata (9.8 kB)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib->glhmm) (
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib->glhmm) (0.12
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib->glhmm)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib->glhmm)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib->glhmm) (2
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotlib->glhmm) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib->glhmm) (
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib->glhm
Requirement already satisfied: typing-extensions>=4.6 in /usr/local/lib/python3.12/dist-packages (from nibabel->glhmm
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.12/dist-packages (from nilearn->glhmm) (1.5.2)
Requirement already satisfied: lxml in /usr/local/lib/python3.12/dist-packages (from nilearn->glhmm) (6.0.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-packages (from pandas->glhmm) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-packages (from pandas->glhmm) (2025.2
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests->gl
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests->glhmm) (3.11)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests->glhmm) (
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests->glhmm) (
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn->gl
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/python3.12/dist-packages (from numba->glh
Requirement already satisfied: networkx>=3.0 in /usr/local/lib/python3.12/dist-packages (from scikit-image->glhmm) (3
Requirement already satisfied: imageio!=2.35.0,>=2.33 in /usr/local/lib/python3.12/dist-packages (from scikit-image->
Requirement already satisfied: tifffile>=2022.8.12 in /usr/local/lib/python3.12/dist-packages (from scikit-image->glh
Requirement already satisfied: lazy-loader>=0.4 in /usr/local/lib/python3.12/dist-packages (from scikit-image->glhmm)
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.12/dist-packages (from statsmodels->glhmm) (1.0
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.7->matplo
Downloading glhmm-1.1.2-py3-none-any.whl (173 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 173.3/173.3 kB 4.3 MB/s eta 0:00:00
Downloading igraph-1.0.0-cp39-abi3-manylinux_2_28_x86_64.whl (5.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.7/5.7 MB 56.6 MB/s eta 0:00:00
Downloading nilearn-0.12.1-py3-none-any.whl (12.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.7/12.7 MB 105.7 MB/s eta 0:00:00
Downloading texttable-1.7.0-py2.py3-none-any.whl (10 kB)
Installing collected packages: texttable, igraph, nilearn, glhmm
Successfully installed glhmm-1.1.2 igraph-1.0.0 nilearn-0.12.1 texttable-1.7.0
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
from glhmm import glhmm, preproc, utils, graphics
```

## Load and prepare data

Example data for this tutorial are available for download from the OSF. This notebook will fetch the relevant data from the OSF project page using the osfclient package. If you prefer, you can also directly download the files from the OSF project page and skip the next two cells.

```
# checks if osfclient is installed and otherwise installs it using pip install
# skip this if you have manually downloaded the data
import sys
import pip

def install(package):
    pip.main(['install', package])

try:
    import osfclient
except ImportError:
    print('osfclient is not installed, installing it now')
    install('osfclient')
```

```
osfclient is not installed, installing it now
WARNING: pip is being invoked by an old script wrapper. This will fail in a future version of pip.
Please see https://github.com/pypa/pip/issues/5599 for advice on fixing the underlying issue.
To avoid this problem you can invoke Python with '-m pip' instead of running pip directly.
Collecting osfclient
  Downloading osfclient-0.0.5-py2.py3-none-any.whl.metadata (5.5 kB)
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from osfclient) (2.32.4)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from osfclient) (4.67.1)
Requirement already satisfied: six in /usr/local/lib/python3.12/dist-packages (from osfclient) (1.17.0)
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests->os
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests->osfclient) (3.
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests->osfclien
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests->osfclien
Downloading osfclient-0.0.5-py2.py3-none-any.whl (39 kB)
```

```
! osf -p 8qcyj fetch GLHMM/data.csv ./example_data/data.csv
! osf -p 8qcyj fetch GLHMM/T.csv ./example_data/T.csv
```

```
100% 18.2M/18.2M [00:01<00:00, 15.7Mbytes/s]
100% 218/218 [00:00<00:00, 992kbytes/s]
```

The standard HMM requires only one input: a timeseries `Y`. When running the model on a concatenated timeseries, e.g., a group of subjects or several scanning sessions, you also need to provide the `indices` indicating where each subject/session in the concatenated timeseries `Y` starts and ends.

Input data for the glhmm should be in numpy format. Other data types, such as .csv, can be converted to numpy using e.g. pandas, as shown below. Alternatively, the `io` module provides useful functions to load input data in the required format, e.g., from existing .mat-files. If you need to create indices from session lengths (as used in the HMM-MAR toolbox) you can use the `auxiliary.make_indices_from_T` function.

Synthetic data for this example should now be in the `glhmm/docs/notebooks/example_data` folder. The file `data.csv` contains synthetic timeseries. The data should have the shape ((no subjects/sessions * no timepoints), no features), meaning that all subjects and/or sessions have been concatenated along the first dimension. The second dimension is the number of features, e.g., the number of parcels or channels. The file `T.csv` specifies the indices in the concatenated timeseries corresponding to the beginning and end of individual subjects/sessions in the shape (no subjects, 2). In this case, we have generated timeseries for 20 subjects and 50 features. Each subject has 1,000 timepoints. The timeseries has the shape (20000, 50) and the indices have the shape (20, 2).

```
#data = pd.read_csv('./example_data/data.csv', header=None).to_numpy()
#T_t = pd.read_csv('./example_data/T.csv', header=None).to_numpy()
data = np.load('/data.npy')
T_t = np.load('/T_t.npy')
# using sub-54095s001_ses-V2 resting fMRI 1, 2 concat as example
```

```
print('data.shape', data.shape)
print('T_t.shape', T_t.shape)
```

```
data.shape (738, 37)
T_t.shape (1, 2)
```

```
data, T_t
```

```
(array([[ 0.22403984,  0.21565321,  0.10232311, ...,  0.12425109,
          0.00361728,  0.0416446 ],
        [ 0.0929393 ,  0.24215753, -0.00200954, ...,  0.0468285 ,
         -0.01136422,  0.03525054],
        [ 0.01238843,  0.2338059 , -0.09660398, ...,  0.01625017,
         -0.04834436, -0.02062193],
        ...,
        [-0.33565763,  0.14275946, -1.0725627 , ...,  0.31710088,
         -0.7390459 ,  1.1055639 ],
        [-0.16116655,  0.06982403, -0.7534976 , ...,  0.2007035 ,
         -0.38398713,  0.76640165],
        [-0.12584803, -0.16710599, -0.24606533, ...,  0.10528208,
          0.02102197,  0.3004009 ]], dtype=float32),
 array([[  0, 738]]))
```

**NOTE: It is important to standardise your timeseries and, if necessary, apply other kinds of preprocessing before fitting the model.**
Standardising will be done separately for each session/subject as specified in the indices. The data provided here are already close to standardised (so the code below will not do much). Other preprocessing steps may be filtering or dimensionality reduction, which may be reasonable depending on the data and model variety. The preprocessing function also outputs a log to keep track of which preprocessing has been performed that can be passed on to the glhmm object. This is required if you have performed preprocessing steps that require backtransforming (PCA/ICA).

```
data,_,log = preproc.preprocess_data(data, T_t)
```

## Initialise and train an HMM

We first initialise the glhmm object and specify hyperparameters. In the case of the standard Gaussian HMM, since we do not want to model an interaction between two sets of variables, we set `model_beta='no'`. The number of states is specified using the `K` parameter. We here estimate `K=4` states. If you want to model a different number of states, change K to a different value. In this example, we want to model states as Gaussian distributions with a mean and full covariance matrix, so that each state is described by a mean amplitude and functional connectivity pattern. To do this, specify `covtype='full'`, the state-specific mean is already set by default. If you do not want to model the mean, add `model_mean='no'`.

```
hmm = glhmm.glhmm(model_beta='no', K=4, covtype='full', preproclogY=log)
```

Optionally, you can check the hyperparameters (including the ones set by default) to make sure that they correspond to how you want the model to be set up.

```
print(hmm.hyperparameters)
```

```
{'K': 4, 'covtype': 'full', 'model_mean': 'state', 'model_beta': 'no', 'dirichlet_diag': 10, 'connectivity': None, 'P
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]]), 'Pistructure': array([ True,  True,  True,  True])}
```

We then train the HMM using the data and indices loaded above. Since we here do not model an interaction between two sets of timeseries but run a standard HMM instead, we set `X=None`. `Y` should be the timeseries in which we want to estimate states (in here called `data`) and indices should be the beginning and end indices of each subject (here called `T_t`). During training, the output will usually show the progress in model fit at each iteration. This can be quite long, so we have here suppressed it for displaying purposes. You can remove the first line (`%%capture`) to show it.

```
%%capture
np.random.seed(123)
hmm.train(X=None, Y=data, indices=T_t)
```

Optionally, you can also return `Gamma` (the state probabilities at each timepoint), `Xi` (the joint probabilities of past and future states conditioned on the data) and `FE` (the free energy of each iteration) at this step, but it is also possible to retrieve them later using the `decode` function for Gamma and Xi and the `get_fe` function for the free energy.

```
%%capture
np.random.seed(123)
Gamma,Xi,FE = hmm.train(X=None, Y=data, indices=T_t)
```

## Inspect model

We can then inspect some interesting aspects of the model. We start by checking what the states look like by interrogating the state means and the state covariances. We will then look at the dynamics, i.e., how states transition between each other (transition probabilities) and how the state sequence develops over time (the Viterbi path). Finally, we will have a look at some summary metrics that can be useful to describe the overall patterns or to relate the model to behaviour using statistical testing or machine learning/out-of-sample prediction.

> We here show some simple result plots. For more plotting options, see the Graphics module

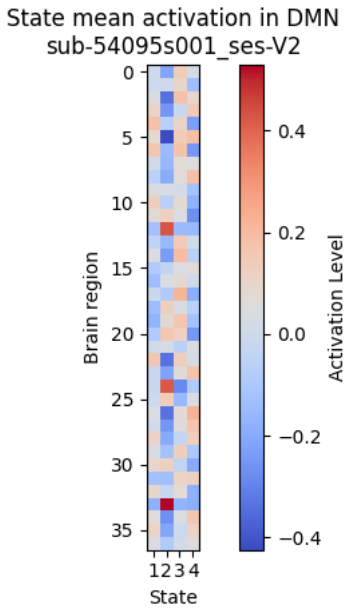
### State means: Time-varying amplitude patterns

The state means can be interpreted as time-varying patterns of amplitude (relative to the baseline). They can be retrieved from the model using the `get_means()` function, or `get_mean(k)` to retrieve only the mean for state `k`:

```
K = hmm.hyperparameters["K"] # the number of states
q = data.shape[1] # the number of parcels/channels
state_means = np.zeros(shape=(q, K))
state_means = hmm.get_means() # the state means in the shape (no. features, no. states)
```

We can then plot these amplitude patterns. This will show the states on the x-axis, each parcel/brain region/channel on the y-axis, and the mean activation of each parcel in each state as the color intensity.

```
cmap = "coolwarm"
plt.imshow(state_means, cmap=cmap,interpolation="none")
plt.colorbar(label='Activation Level') # Label for color bar
plt.title("State mean activation in DMN\nsub-54095s001_ses-V2")
plt.xticks(np.arange(K), np.arange(1,K+1))
plt.gca().set_xlabel('State')
plt.gca().set_ylabel('Brain region')
plt.tight_layout()  # Adjust layout for better spacing
plt.show()
```



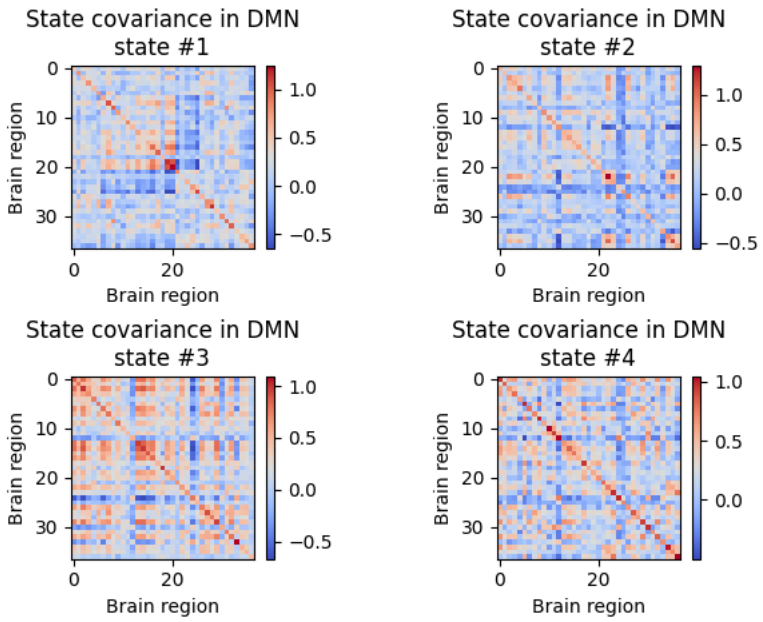

### State covariances: Time-varying functional connectivity

The state covariances represent the time-varying functional connectivity patterns that we have estimated in the fMRI recordings. They can be obtained from the model using the `get_covariance_matrix` function:

```
state_FC = np.zeros(shape=(q, q, K))
for k in range(K):
    state_FC[:,:,k] = hmm.get_covariance_matrix(k=k) # the state covariance matrices in the shape (no. features, no.
```

We can then plot the covariance (i.e., functional connectivity) of each state. These are square matrices showing the brain region by brain region functional connectivity patterns:

```
for k in range(K):
    plt.subplot(2,2,k+1)
    plt.imshow(state_FC[:,:,k],cmap=cmap)
    plt.xlabel('Brain region')
    plt.ylabel('Brain region')
    plt.colorbar()
    plt.title("State covariance in DMN\nstate #%s" % (k+1))
plt.subplots_adjust(hspace=0.7, wspace=0.8)
plt.show()
```




## Transition probabilities

The transition probabilities indicate the temporal order in the state sequence, i.e., the probability of transitioning from any one state to any other state. They are contained in `hmm.P` with a shape of [K, K]:

```
TP = hmm.P.copy() # the transition probability matrix
```

We can then plot the transition probability matrix. Note that self-transitions (i.e., staying in the same state) are considerably more likely in a timeseries that has some order, so there should be a strong diagonal pattern. For comparison, we also show the transition probabilities excluding self-transitions:

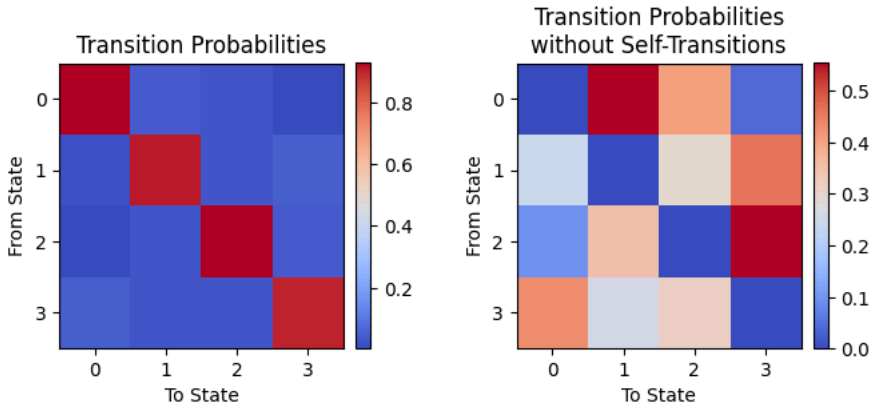
```
# Plot Transition Probabilities
plt.figure(figsize=(7, 4))

# Plot 1: Original Transition Probabilities
plt.subplot(1, 2, 1)
plt.imshow(TP, cmap=cmap, interpolation='nearest')  # Improved color mapping
plt.title('Transition Probabilities')
plt.xlabel('To State')
plt.ylabel('From State')
plt.colorbar(fraction=0.046, pad=0.04)

# Plot 2: Transition Probabilities without Self-Transitions
TP_noself = TP - np.diag(np.diag(TP))  # Remove self-transitions
TP_noself2 = TP_noself / TP_noself.sum(axis=1, keepdims=True)  # Normalize probabilities
plt.subplot(1, 2, 2)
plt.imshow(TP_noself2, cmap=cmap, interpolation='nearest')  # Improved color mapping
plt.title('Transition Probabilities\nwithout Self-Transitions')
```

```
plt.xlabel('To State')
plt.ylabel('From State')
plt.colorbar(fraction=0.046, pad=0.04)

plt.tight_layout()  # Adjust layout for better spacing
plt.show()
```



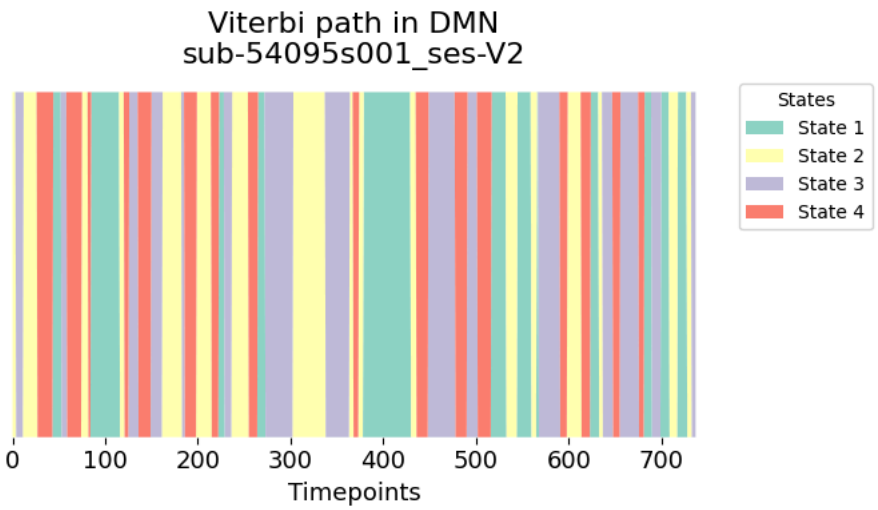

## Viterbi path

The Viterbi path is a discrete representation of the state timecourse, indicating which state is active at which timepoint. We may be able to see whether some states tend to occur more for certain subjects, or are related to a stimulus that occurs at specific timepoints. The Viterbi path can also be informative to understand whether the HMM is "mixing", i.e., states occur across subjects, or whether the model estimates the entire session of one subject as one state (see Ahrends et al. 2022). You can retrieve the Viterbi path using the `decode` function and setting `viterbi=True`:

```
vpath = hmm.decode(X=None, Y=data, indices=T_t, viterbi=True)
```

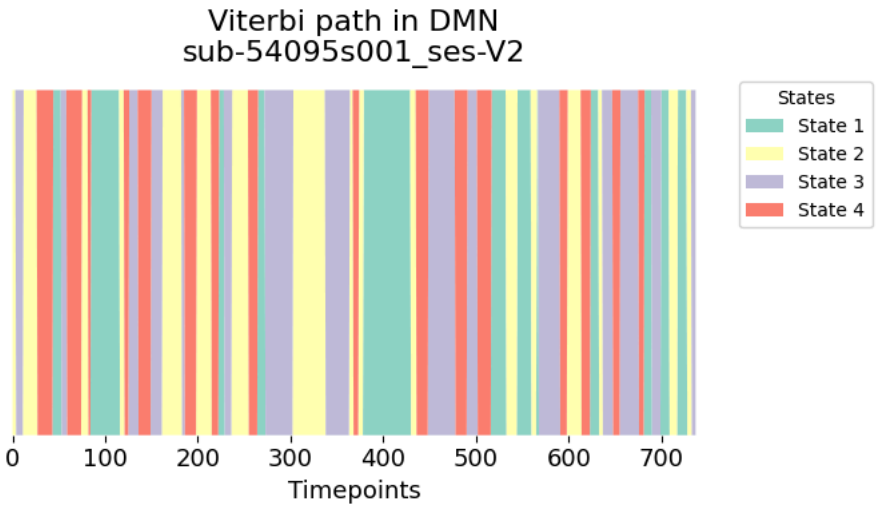
And plot the Viterbi path (see also `graphics` module):

```
graphics.plot_vpath(vpath, title="Viterbi path in DMN\nsub-54095s001_ses-V2")
```




We can also visualize the Viterbi path for the first subject from timepoints 0-1000

```
num_subject = 0
graphics.plot_vpath(vpath[T_t[num_subject,0]:T_t[num_subject,1],:], title="Viterbi path in DMN\nsub-54095s001_ses-V2
```




Plotting the Viterbi path indicates that the states show fast dynamics across all subjects (concatenated along the y-axis).

## Summary metrics

Once we have estimated the model parameters, we can compute some summary metrics to describe the patterns we see more broadly. These summary metrics can be useful to relate patterns in the timeseries to behaviour, using e.g., statistical testing (see Statistical testing tutorial) or machine learning (see Prediction tutorial). The module `utils` provides useful functions to obtain these summary metrics.

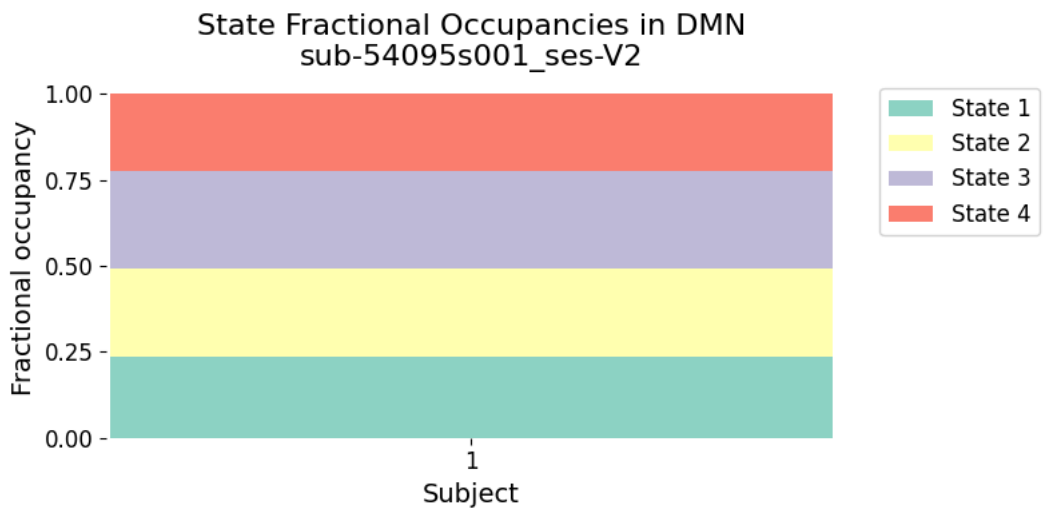
The *fractional occupancy (FO)* indicates the fraction of time in each session that is occupied by each state. For instance, if one state was active for the entire duration of the session, this state's FO would be 1 (100%) and all others would be 0. If, on the other hand, all states are present for an equal amount of timepoints in total, the FO of all states would be $1/K$ (the number of states). This can be informative to understand whether one state is more present in a certain group of subjects or experimental condition, or to interrogate mixing (explained above).

You can obtain the fractional occupancies using the `get_FO` function. The output is an array containing the FO of each subject along the first dimension and each state along the second dimension.

```
FO = utils.get_FO(Gamma, indices=T_t)
```

And plot the Fractional Occupancies (see also `graphics` module):

```
graphics.plot_FO(FO, num_x_ticks=FO.shape[0], title='State Fractional Occupancies in DMN\nsub-54095s001_ses-V2')
```

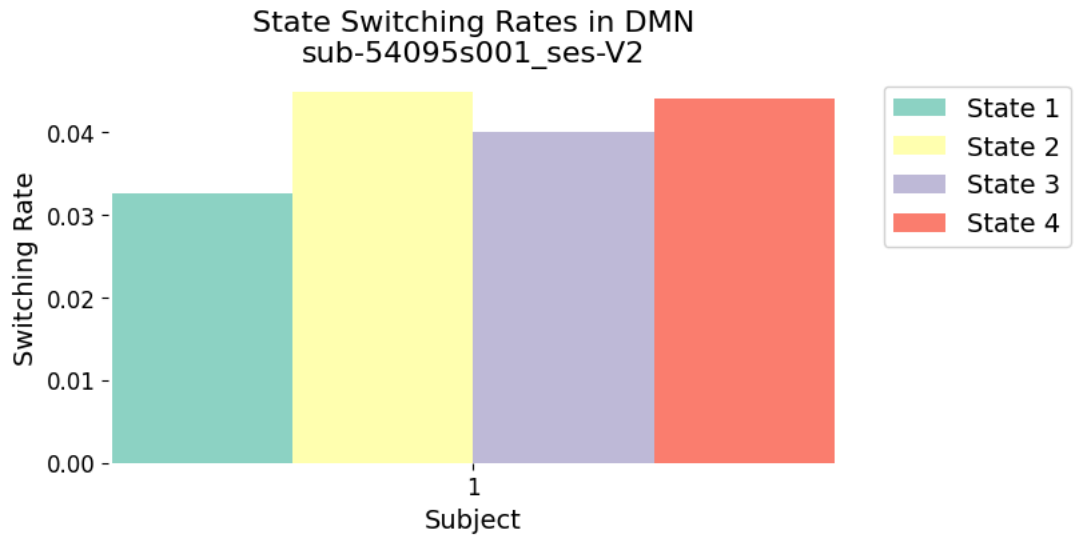



The switching rate indicates how quickly subjects switch between states (as opposed to stay in the same state).

```
SR = utils.get_switching_rate(Gamma, T_t)
```

And plot the switching rate (see also `graphics` module):

```
graphics.plot_switching_rates(SR, num_x_ticks=SR.shape[0], title = 'State Switching Rates in DMN\nsub-54095s001_ses-
```




The state lifetimes (also called *dwell times*) indicate how long a state is active at a time (either on average, median, or maximum). This can be informative to understand whether states tend to last longer or shorter times, pointing towards slower vs. faster dynamics:

```
LTmean, LTmed, LTmax = utils.get_life_times(vpath, T_t)
```

Now we will plot the mean the state lifetime (see also `graphics` module):

```
aphics.plot_state_lifetimes(LTmean, num_x_ticks=LTmean.shape[0], ylabel='Mean lifetime', title='State Lifetimes in DM
```