Behavpy - Hidden Markov Model

Behavpy_HMM is an extension of behavpy that revolves around the use of Hidden Markov Models (HMM) to segment and predict sleep and awake stages in Drosophila. This method is based on Wiggin et al PNAS 2020

- Training an HMM
- Visualising with the HMM

Training an HMM

Behavpy_HMM is an extension of behavpy that revolves around the use of Hidden Markov Models (HMM) to segment and predict sleep and awake stages in *Drosophila*. A good introduction to HMMs is available here: https://web.stanford.edu/~jurafsky/slp3/A.pdf.

The provide a basic overview, HMMS are a generative probabilistic model, in which there is an assumption that a sequence of observable variables is generated by a sequence of internal hidden states. The transition probabilities between hidden states are assumed to follow the principles of a first order Markov chain.

Within this tutorial we will apply this framework to the analysis of sleep in *Drosophila*, using the observable variable `movement` to predict the hidden states: `deep sleep`, `light sleep`, `quiet awake`, `active awake`.

Hmmlearn

BehavpyHMM utilises the brilliant Python package Hmmlearn under the hood. Currently behavpy_HMM only uses the categorical version of the models available, as such it can only be trained and decode sequences that are categorical and in an integer form. If you wish to use their other models (multinomial, gaussian and guassian mixture models) or want to read more on their capabilities, head to their docs page for more information: https://hmmlearn.readthedocs.io/en/latest/index.html

Initialising behavpy_HMM

The HMM variant of behavpy is initialised in the same manner as the behavpy class. In fact, it is just a sub-classed version of behavpy so all the previous methods are still applicable, alongside the usual Pandas capabilities.

```
import ethoscopy as etho


# You can load you data using the previously mentioned functions
meta = etho.link_meta_index(meta_loc, remote, local)
data = etho.load_ethoscope(meta, min_time = 24, max_time = 48, reference_hour = 9.0)


# Remember you can save the data and metadata as pickle files, see the downloading section for
a reminder about it


# to initialise a behavpy object, just call the class with the data and meta as arguments.
Have check = True to ensure the ids match.
df = etho.behavpy_HMM(data, meta, check = True)
```

Data curation

The data passed to the model needs to be good quality with long sequences containing minimal gaps and all a decent length in total.

> Be careful when loading in your data. If you use the `sleep_annotation()` loading analysis function then it will automatically interpolate missing data as sleep giving you `NaN` (filler values) throughout your dataset. It's preferential to use the `max_velocity_detector()` function if you plan to use the HMM.

Use the `.curate()` and `.bin_time()` methods mentioned previously to curate the data. It is recommended to bin the data points to 60 second bins when analysing sleep and movement. But feel free to experiment with what works with your data.

```
import ethoscopy as etho
# load your data and metadata in and intialse your behavpy_HMM object
df = etho.behavpy_HMM(data, meta, check = True)

# it's often a good idea to filter out the start and beginning of experiments
df = df.t_filter(start_time = 24, end_time = 96)

# remove specimens with less than a days worth of data (if binned to 60 seconds)
df = df.curate(points = 1440)

# Any NaN values will throw an error when trying to train the HMM. It's good practice to check the data for any
# If below returns True you have NaN values in your data
print(np.any(np.isnan(df['YOUR COLUMN'].to_numpy())))
# If it is True then look at those values and replace or remove (the whole specimen) from the dataset
# Use below to view which rows have NaNs, see pandas documnetation for more information on how to replace NaN values
df[df['YOUR COLUMN'].isnull()]

# binning the variable can be done within the .hmm_train() method, see next!
```

Training a HMM

Behavpy_HMM provides a wrapper function for hmmlearn multinomial model. The method extends the hmmlearn model by automatically manipulating the data into a readable format for training. Additionally, the method provides the option to randomise the initialised transmission / observable parameters to train the model several times. Each iterative model is compared to the best previous performing model with the hopes of avoiding reach a local minima during hmmlearns own internal iterative process.
As mentioned previously the HMM used here is categorical, therefore only select data that is categorically discrete. For example here we will be using activity, with not moving being one state and moving the other. These states as data must be in integer form. So this example, not moving will = 0 and moving will = 1. Follow this

example if using more states, so if you have three states use the integers 0, 1, 2.
In setting up the HMM you have to specify how many hidden states your system will have and the what states can transition into one another. See below for an example.

Setting up

```
# the setup for .hmm_train()

# name your hidden states and observables. This is just purely for visualising the output and setting the number of hidden states!
hidden_states = ['Deep sleep', 'Light sleep', 'Light awake', 'Active awake']
observable_variables = ['inactive', 'active']

# if you believe your hidden states can only transition into specific states or there's a flow you can setup the model to train with this in mind. Have the transitions you want to happen set as 'rand' and those that can't as 0
t_prob = np.array([['rand', 'rand', 'rand', 0.00],
                   ['rand', 'rand', 'rand', 0.00],
                   [0.00, 'rand', 'rand', 'rand'],
                   [0.00, 'rand', 'rand', 'rand']])

# Make sure your emission matrix aligns with your categories
em_prob =  np.array([[1, 0],
                     [1,0],
                     ['rand', 'rand'],
                     ['rand', 'rand']])

# the shape of each array should be len(hidden_states) X len(names). E.g. 4x4 and 4x2 respectively
```

> You can leave `trans_probs` and `em_prob` as `None` to have all transitions randomised before each iteration

Training

```
# add the above variables to the hmm_train method
# iterations is the number of loops with a new randomised parameters
# hmm_interactions is the number of loops within hmmlearn before it stops. This is superceeded by tol, with is the difference in the logliklihood score per interation within hmmlearn
# save the best performing model as a .pkl file
df.hmm_train(
        states = hidden_states,
```

```
        observables = observable_variables,
        var_column = 'moving',
        trans_probs = t_prob,
        emiss_probs = em_prob,
        start_probs = None,
        iterations = 10,
        hmm_iterations = 100,
        tol = 2,
        t_column = 't',
        bin_time = 60,
        file_name = 'experiment_hmm.pkl', # replace with your own file name
        verbose = False)
# If verbose is true the logliklihood per hmm iteraition is printed to screen
```

> Diving deeper! To find the best model the dataset will be split into a test (10%) and train (90%) datasets. Each successive model will be scored against the best on the test dataset. The best scoring model will be the final save to your file name.

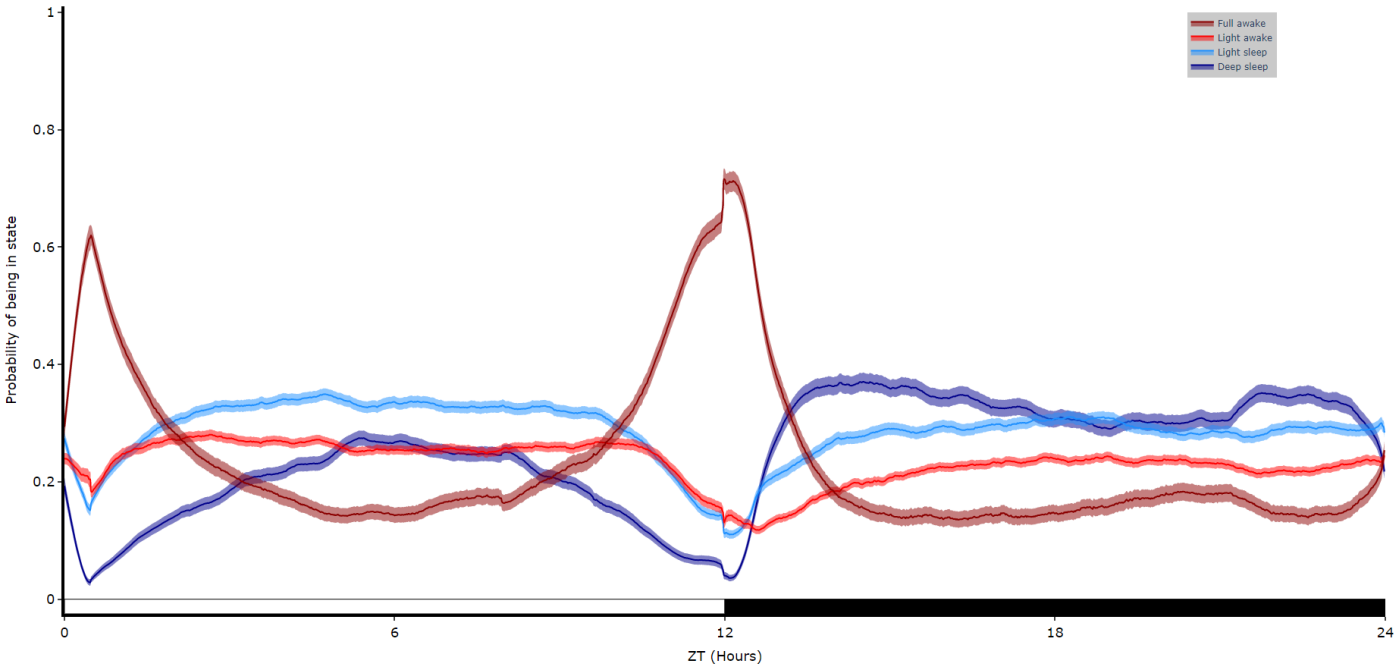
```
Starting probabilty table:
|    |   Deep_sleep |   Light_sleep |   Light_awake |   Full_awake |
|----|--------------|---------------|---------------|--------------|
|  0 |  0.000323396 |     0.0350274 |       0.10907 |     0.855579 |

Transition probabilty table:
|             |   Deep_sleep |   Light_sleep |   Light_awake |   Full_awake |
|-------------|--------------|---------------|---------------|--------------|
| Deep_sleep  |     0.834628 |     0.0101301 |      0.155242 |    0         |
| Light_sleep |     0.112523 |     0.672489  |      0.214988 |    0         |
| Light_awake |     0        |     0.333665  |      0.653137 |    0.0131978 |
| Full_awake  |     0        |     0.0183509 |      0        |    0.981649  |

Emission probabilty table:
|             |   inactive |   active |
|-------------|------------|----------|
| Deep_sleep  |  1         | 0        |
| Light_sleep |  1         | 0        |
| Light_awake |  0.0432473 | 0.956753 |
| Full_awake  |  0.016272  | 0.983728 |
```

Visualising with the HMM

The best way to get to grips with your newly trained HMM is to decode some data and has a look at it visually.

Single plots

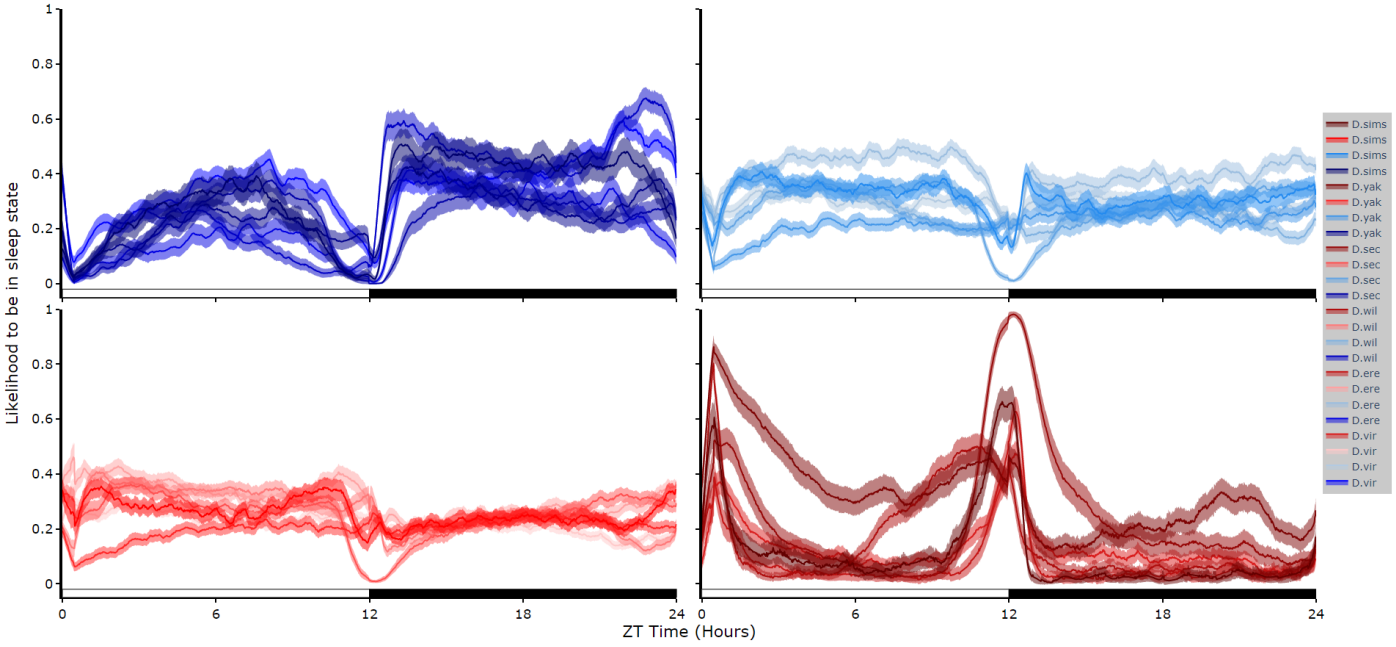
```
# Like plot_overtime() this method will take a single variable and trained hmm, and plot them over time.
# If you're using behavpy_HMM to look at sleep using the 4 state structure you don't have to specify labels and colours, as they are pre written. However, if you aren't please specify.

# use below to open your saved trained hmmlearn object
with open('/Users/trained_hmms/4_states_hmm.pkl', 'rb') as file:
    h = pickle.load(file)

fig = df.plot_hmm_overtime(
hmm = h,
variable = 'moving',
labels = ['Deep sleep', 'Light sleep', 'Light awake', 'Full awake'],
# colours = ['blue', 'green', 'yellow', 'red'], # example colours
wrapped = True,
bin = 60
)
fig.show()

# You cannot facet using this method, see the next plot for faceting
```



Faceted plots

To reduce clutter when faceting the plot will become a group of subplots, with each subplot the collective plots of each specimen for each hidden state.

```
# plot_hmm_split works just like above, however you need to specify the column to facet by.
You can also specify the arguments if you don't want all the groups from the facet column

fig = df.plot_hmm_split(
    hmm = h,
    variable = 'moving',
    facet_col = 'species',
    wrapped = True,
    bin = 60
)
fig.show()
```

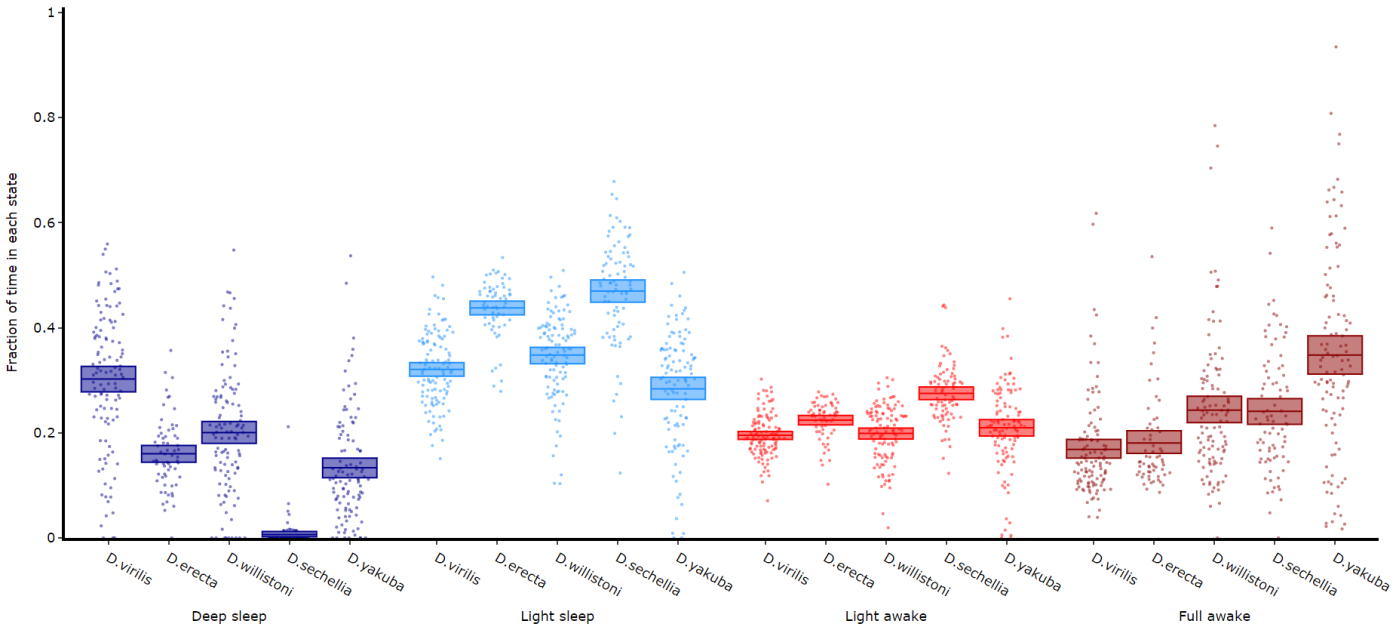



```
# You don't always want to decode every group with the same hmm. In fact with different
species, mutants,
# subgroups you should be training a fresh hmm
# if you pass a list to the hmm parameter the same length as the
# facet_arg argument the method will apply the right hmm to each group

fig = df.plot_hmm_split(
    hmm = [hv, he, hw, hs, hy],
    variable = 'moving',
    facet_labels = ['D.virilis', 'D.erecta', 'D.willistoni', 'D.sechellia', 'D.yakuba']
    facet_col = 'species',
    facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak'],
    wrapped = True,
    bin = [60, 60, 60, 60, 60]
)
fig.show()
```

Quantify time in each state

```
# Like plot_quantify() this method will quantify how much of the time each specimen is within
each state.


fig, stats = df.plot_hmm_quantify(
    hmm = [hv, he, hw, hs, hy],
    variable = 'moving',
```

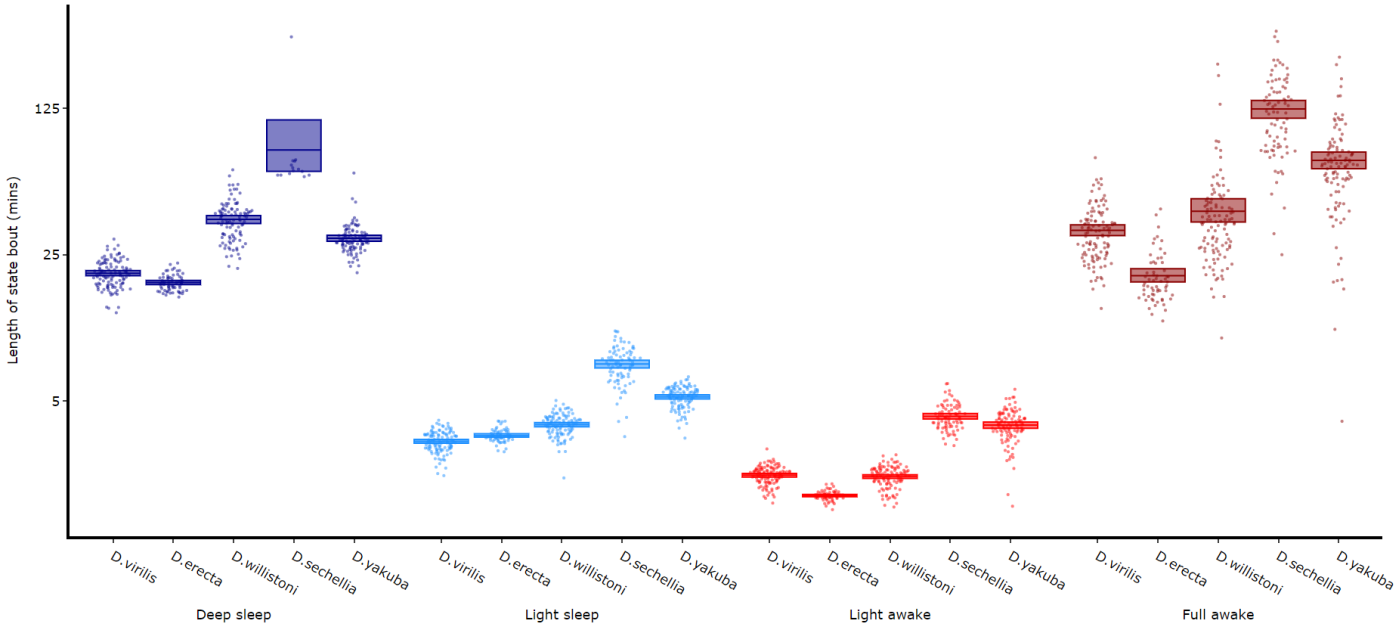
```
    facet_labels = ['D.virilis', 'D.erecta', 'D.willistoni', 'D.sechellia', 'D.yakuba']
    facet_col = 'species',
    facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak'],
    bin = [60, 60, 60, 60, 60]
)
fig.show()
```



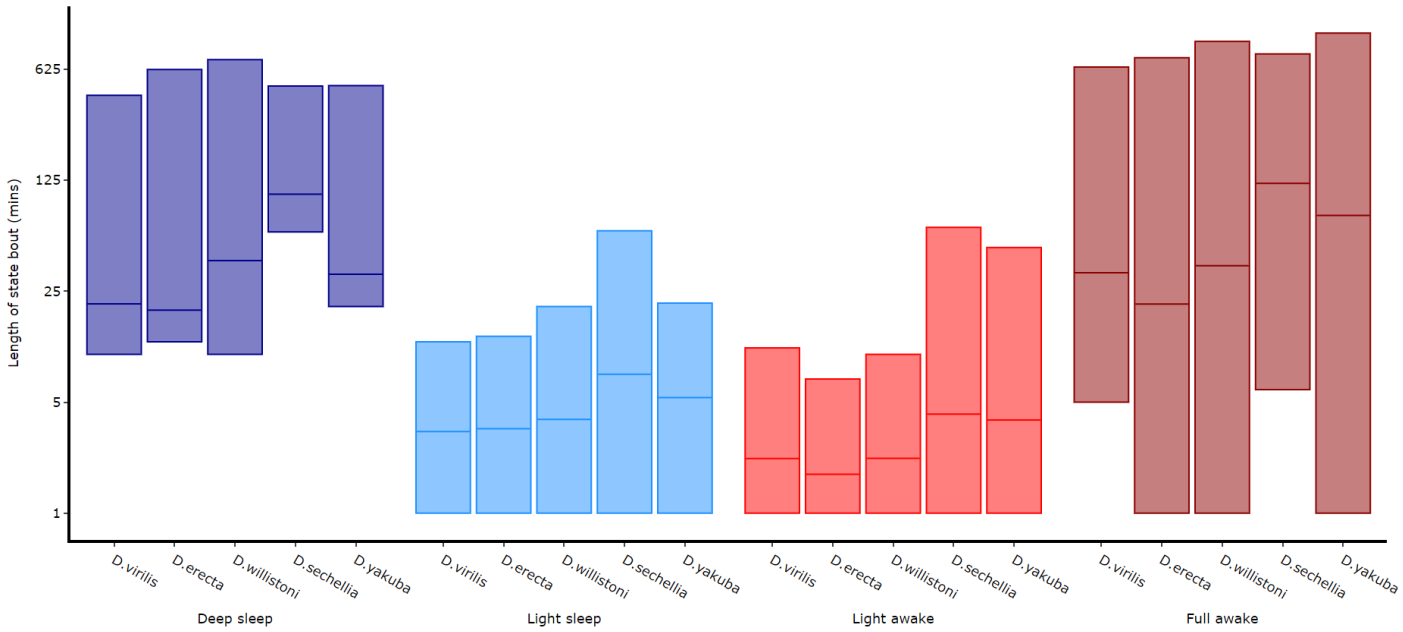
Quantifying length of each state

Its a good idea to look at the length of each state to gain an understanding of how the model is separating your data. There are two length methods 1) one to plot the mean lengths per state per specimen and 2) a plot to show the maximum and minimum state length per group.

```
# all hmm quantifying plots have the same parameters, so just change the method name and
you're good to go

fig, stats = df.plot_hmm_quantify_length(
    hmm = [hv, he, hw, hs, hy],
    variable = 'moving',
    facet_labels = ['D.virilis', 'D.erecta', 'D.willistoni', 'D.sechellia', 'D.yakuba']
    facet_col = 'species',
    facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak'],
    bin = [60, 60, 60, 60, 60]
)
fig.show()
```



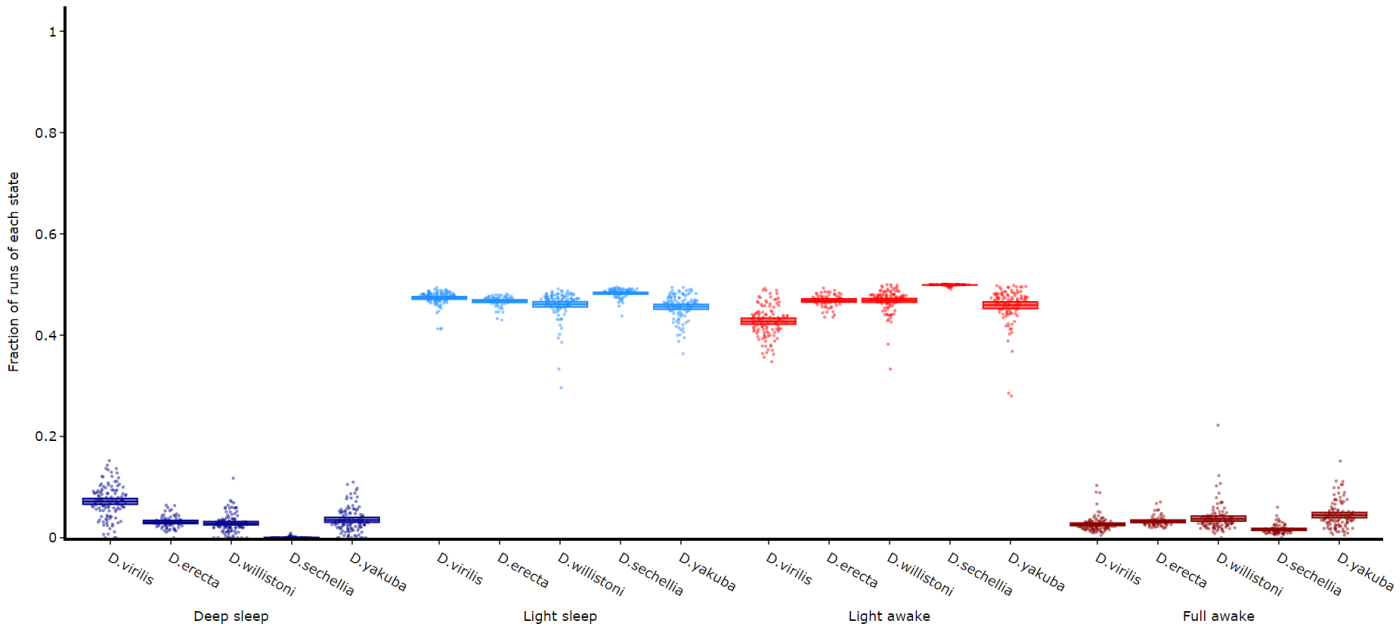

```
# The length min/max method only shows the the min and maximum points as a box
# this plot doesnt return any stats, as it is just the min/max and mean values
fig = df.plot_hmm_quantify_length_min_max(
    hmm = [hv, he, hw, hs, hy],
    variable = 'moving',
    facet_labels = ['D.virilis', 'D.erecta', 'D.willistoni', 'D.sechellia', 'D.yakuba']
    facet_col = 'species',
    facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak'],
    bin = [60, 60, 60, 60, 60]
)
fig.show()

# Below you can see when the model seperates light sleep from deep sleep
```



Quantifying transitions

The time in each state and the average length are good overview stats, but can be misleading about how often a state occurs if the state is short. This next method quantifies the amount of times a state is transitioned into, effectively counting the instances of the state regardless of time.

```
fig, stats = df.plot_hmm_quantify_transition(
    hmm = [hv, he, hw, hs, hy],
    variable = 'moving',
    facet_labels = ['D.virilis', 'D.erecta', 'D.willistoni', 'D.sechellia', 'D.yakuba']
    facet_col = 'species',
    facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak'],
    bin = [60, 60, 60, 60, 60]
)
fig.show()
```



Raw decoded plot

Sometimes you will want to view the raw decoded plots of individual specimens to get a sense of the daily ebb and flow. .plot_hmm_raw() will plot 1 or more individual specimens decoded sequences.

```
# The ID of each specimen will be printed to the screen in case you want o investigate one or
som further
fig = df.plot_hmm_raw(
    hmm = h,
    variable = 'moving',
    num_plots = 5, # the number of different specimens you want in the plot
    bin = 60,
    title = ''
)
fig.show()

# Addtionally, if you have response data from mAGO experiments you can add that behavpy
dataframe to
# highlight points of interaction and their response
# Purple = response, lime = no response
## df.plot_hmm_raw(mago_df = mdf, hmm = h, variable = 'moving', num_plots = 5, bin = 60, title
= '', save = False)
```