neuroHMM

Audren Butery, Timothée Fronteau, Solenzio Konlé, Vincent Mous

Mar 23, 2022

CONTENTS:

The neuroHMM project started as a student project in collaboration with the Neuroscience Institute of La Timone. The aim of the project is to provide Machine Learning tools, initially adapted to neurophysiological signals, which allow spatial, temporal and frequency analysis of experimental data. Most of the tools developed are based on hidden Markov models (HMMs), and are inspired by the HMM-MAR library developed in Matlab by a team of Oxford researchers. Our library aims to be an extension of the scikit-learn library, a reference for Machine Learning experts, and follows as much as possible the scikit-learn Estimators development standards. Some of the tools developed are also based on the hmmlearn library, which implements the more classical HMMs with scikit-learn development standards.

On this website, you will find:

- A presentation of the theory behind each of the implemented tools by following the *Theory* tab;
- A guide to installing and using the library on the *User Guide* page;
- Documentation of each class and associated methods in the *API Reference*;
- Example notebooks of how to use the Estimators on the *Examples* page.

CHAPTER

ONE

THEORY

What is a hidden Markov model (HMM) ?

1.1 Theory behind Cluster Decoder

1.1.1 Regression Method and EM Algorithm

1.1.2 Hierarchical Method

1.1.3 Sequential Method

The sequential method is characterized by a first strong hypothesis which supposes that the k states of the signal will follow one after the other without ever reappearing, even if each ones of them can have a different duration. For example, we could have a signal where the states follow each other in this way 1-2-3-4-... but not such as 1-2-1-3-4-2-3-...

The fit method of the sequential algorithm proceeds as follows:

We initialize an error e0 by first assuming an uniform distribution in time of the k states. Thus for each state we will take into account only the relevant part of the X and y which allows us to calculate a decoding matrix for each state. Once this is done, we then calculate the cumulative error on each state by computing the distance between y and X*Wk (Wk the decoding matrix). After that we launch a loop that will do the same thing as before but choosing random distributions of states in time. If the error is smaller we save this distribution then we continue the loop. At the end we obtain a distribution as well as decoding matrices that have minimized the error on Max_iter random draws.

1.2 Theory behind TDE-HMM

(Description, Développement, Intérêt et exemples)

Le time-delay embedded hidden Markov model (TDE-HMM) est la méthode de clustering des papiers d'Oxford que nous maîtrisions le mieux. En effet, il s'agit de la méthode utilisée lors du stage de Timothée à l'INT pour effectuer la détection de bouffées de signal dans les données d'expérience présentées plus haut, caractérisées par des spectres fréquentiels différents. Elle repose sur un modèle de Markov caché dont le signal observé(les composantes indépendantes extraites de l'EEG) est représenté par le modèle d'observation décrit dans le papier de [Seedat et al., 2020] et que l'on résume dans la figure ci-contre.

Dans ce modèle d'observation, les données [y_t-N, ..., y_t+N], une fenêtre d'observation de taille 2N+1, sont supposées générées par une loi gaussienne vectorielle, dont les paramètres (le vecteur moyenne et la matrice de covariance) dépendent de l'état caché x_t.

1.3 Theory behind TUDA

CHAPTER

TWO

USER GUIDE

2.1 Installation

To install the neuroHMM library, you can copy the following command line into your terminal:

```
git clone https://github.com/sconle/HMM.git
```

or download the ZIP repository on GitHub.

To be able to use our library, you will also need to install the necessary packages by using the following command line into your terminal:

```
pip install -r requirements.txt
```

2.2 Using our Estimators

Classifiers and Regressors are subclasses of the scikit-learn Estimator class, and correspond to supervised or unsupervised classification or regression models. All Estimators of scikit-learn and of the libraries which are inspired by them, including ours, have a structure and methods in common to unify their use. In particular, here are the basic lines of code that can be used with our Estimators:

```
model = Cluster_Decoder(n_components=n_states, n_iter=n_iter, covariance_
↪type=covariance_type, tol=tol)
```

This line of instructions is used to create the `Cluster_Decoder` Estimator according to the model and constraints specific to the cluster decoding method, and to define the hyperparameters of this Estimator.

```
model.fit(X, y)     # or model.fit(X)
```

This line of code is used to adjust the parameters of the Estimator and to find those that best fit the training data (X, y) - or X for unlabelled data, depending on the Estimator used - using the inference algorithm implemented in the Estimator.

```
y_predict = model.predict(X)
```

This method returns the estimate y_predict obtained from the data X and the Estimator's own parameters.

```
proba = model.predict_proba(X)
```

This method returns the likelihood of each element of X belonging to each cluster according to the Estimator's own parameters.

CHAPTER

THREE

API REFERENCE

On this API reference you will find the documentation for the Estimators listed below:

3.1 Cluster Decoder

class myHmmPackage.cluster_decoder.**ClusterDecoder**(*n_clusters=4*, *gamma_init=None*, *decoding_mats_init=None*, *method='regression'*, *measure='error'*, *max_iter=100.0*, *reg_param=1e-05*, *transition_scheme=None*, *init_scheme=None*)

ClusterDecoder is an Estimator that performs supervised decoding with a predefined number of decoding matrices. A clustering method is used to choose which decoding matrix to use for each sample of each input data. The metaparameters are gamma_(ndarray, shape (n_time_points, n_clusters)) and decoding_mats_(ndarray, shape (n_clusters, n_time_points, n_label_features))

Parameters

- **n_clusters** (*int*) – Number of desired clusters. default=4
- **gamma_init** (*ndarray*) – The initial value of **gamma_**. shape=(n_time_points, n_clusters) or None. default=None
- **decoding_mats_init** (*ndarray*) – The initial value of **decoding_mats_**. shape=(n_clusters, n_time_points, n_label_features) or None. default=None
- **method** (*str*) – Name of the decoding method used among 'sequential' or 'regression'. default='regression'
- **measure** (*str*) – Measure used for the 'sequential' method. default='error'
- **max_iter** (*int*) – Number of iterations. default=100
- **reg_param** (*float*) – Regularization parameter. default=10e-5
- **transition_scheme** (*ndarray*) – Constraints for the cluster transitions. shape=(n_clusters, n_clusters) or None. default=None
- **init_scheme** (*ndarray*) – Initial probability for each cluster. shape=(n_clusters,) or None. default=None

Variables

- **gamma** (*ndarray*) – The tensor containing each cluster's probability time-course, of shape=(n_time_points, n_clusters).

- **decoding_mats** (*ndarray*) – The tensor containing the decoding matrices associated to each cluster, of shape=(n_clusters, n_time_points, n_label_features).

fit (*X*, *y*)
Estimate model parameters.

Parameters

- **X** (*array-like*) – The training input samples (brain data) of shape=(n_samples, n_time_points, n_regions)
- **y** (*array-like*) – The target values, An array of int and of shape=(n_samples, n_time_points, n_label_features)

Returns Returns self

predict (*X*)
Find most likely state sequence corresponding to X, then computes y_predict, the predicted labels given X.

Parameters **X** – The training input samples (brain data) of shape=(n_samples, n_time_points, n_regions)

Returns Returns y_predict, the predicted labels in an tensor of shape=(n_samples, n_time_points, n_label_features)

3.2 TDE-HMM

class myHmmPackage.tde_hmm.**TDE_HMM**(*n_components=3*, *covariance_type='full'*, *min_covar=0.001*, *startprob_prior=1.0*, *transmat_prior=1.0*, *means_prior=0*, *means_weight=0*, *covars_prior=0.01*, *covars_weight=1*, *algorithm='viterbi'*, *random_state=None*, *n_iter=10*, *tol=0.01*, *verbose=False*, *params='stmc'*, *init_params='stmc'*)
TDE_HMM is an Estimator that performs supervised decoding with a predefined number of decoding matrices. A clustering method is used to choose which decoding matrix to use for each sample of each input data.

Parameters

- **n_components** (*int*) – Number of states. Default 3
- **n_iter** (*int*) – Optional. Maximum number of iterations to perform.
- **covariance_type** (*str*) – Optional. The type of covariance parameters to use: * “spherical” — each state uses a single variance value that

  applies to all features (default).

  – ”diag” — each state uses a diagonal covariance matrix.

  – ”full” — each state uses a full (i.e. unrestricted) covariance matrix.

  – ”tied” — all states use **the same** full covariance matrix.

- **min_covar** (*float*) – Optional. Floor on the diagonal of the covariance matrix to prevent overfitting. Defaults to 1e-3.
- **startprob_prior** (*array*) – Optional. Shape of (n_components,). Parameters of the Dirichlet prior distribution for `startprob_`.

- **transmat_prior** (*array*) – Optional. Shape of (n_components, n_components). Parameters of the Dirichlet prior distribution for each row of the transition probabilities `transmat_`.
- **means_prior/means_weight** (*array*) – Optional. Shape of (n_components,). Mean and precision of the Normal prior distribtion for `means_`.
- **covars_prior/covars_weight** (*array*) – Optional. Shape of (n_components,). Parameters of the prior distribution for the covariance matrix `covars_`. If `covariance_type` is "spherical" or "diag" the prior is the inverse gamma distribution, otherwise — the inverse Wishart distribution.
- **algorithm** (*str*) – Optional. {"viterbi", "map"} Decoder algorithm.
- **random_state** (*int_seed*) – Optional. A random number generator instance.
- **tol** (*float*) – Optional. Convergence threshold. EM will stop if the gain in log-likelihood is below this value.
- **verbose** (*bool*) – Optional. Whether per-iteration convergence reports are printed to `sys.stderr`. Convergence can also be diagnosed using the `monitor_` attribute.
- **params/init_params** (*str*) – Optional. The parameters that get updated during (`params`) or initialized before (`init_params`) the training. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars. Defaults to all parameters.
- **implementation** (*str*) – Optional. Determines if the forward-backward algorithm is implemented with logarithms ("log"), or using scaling ("scaling"). The default is to use logarithms for backwards compatability.

fit (*X*, *y=None*)

Estimate model parameters.

Parameters

- **X** (*array-like*) – The training input samples
- **y** (*array-like*) – The target values

Returns Returns self

predict_proba (*X*, *y=None*)

Find most likely state sequence corresponding to X, then computes y_predict, the predicted labels given X.

Parameters **X** – The training input samples

Returns Returns posteriors, the predicted labels in an tensor

class myHmmPackage.cluster_decoder.**ClusterDecoder**(*n_clusters=4*, *gamma_init=None*, *decoding_mats_init=None*, *method='regression'*, *measure='error'*, *max_iter=100.0*, *reg_param=1e-05*, *transition_scheme=None*, *init_scheme=None*)

ClusterDecoder is an Estimator that performs supervised decoding with a predefined number of decoding matrices. A clustering method is used to choose which decoding matrix to use for each sample of each input data. The metaparameters are gamma_(ndarray, shape (n_time_points, n_clusters)) and decoding_mats_(ndarray, shape (n_clusters, n_time_points, n_label_features))

Parameters

- **n_clusters** (*int*) – Number of desired clusters. default=4
- **gamma_init** (*ndarray*) – The initial value of **gamma_**. shape=(n_time_points, n_clusters) or None. default=None
- **decoding_mats_init** (*ndarray*) – The initial value of **decoding_mats_**. shape=(n_clusters, n_time_points, n_label_features) or None. default=None
- **method** (*str*) – Name of the decoding method used among 'sequential' or 'regression'. default='regression'
- **measure** (*str*) – Measure used for the 'sequential' method. default='error'
- **max_iter** (*int*) – Number of iterations. default=100
- **reg_param** (*float*) – Regularization parameter. default=10e-5
- **transition_scheme** (*ndarray*) – Constraints for the cluster transitions. shape=(n_clusters, n_clusters) or None. default=None
- **init_scheme** (*ndarray*) – Initial probability for each cluster. shape=(n_clusters,) or None. default=None

Variables

- **gamma** (*ndarray*) – The tensor containing each cluster's probability time-course, of shape=(n_time_points, n_clusters).
- **decoding_mats** (*ndarray*) – The tensor containing the decoding matrices associated to each cluster, of shape=(n_clusters, n_time_points, n_label_features).

fit (*X*, *y*)

Estimate model parameters.

Parameters

- **X** (*array-like*) – The training input samples (brain data) of shape=(n_samples, n_time_points, n_regions)
- **y** (*array-like*) – The target values, An array of int and of shape=(n_samples, n_time_points, n_label_features)

Returns Returns self

predict (*X*)

Find most likely state sequence corresponding to X, then computes y_predict, the predicted labels given X.

Parameters **X** – The training input samples (brain data) of shape=(n_samples, n_time_points, n_regions)

Returns Returns y_predict, the predicted labels in an tensor of shape=(n_samples, n_time_points, n_label_features)

class myHmmPackage.tde_hmm.**TDE_HMM**(*n_components=3*, *covariance_type='full'*, *min_covar=0.001*, *startprob_prior=1.0*, *transmat_prior=1.0*, *means_prior=0*, *means_weight=0*, *covars_prior=0.01*, *covars_weight=1*, *algorithm='viterbi'*, *random_state=None*, *n_iter=10*, *tol=0.01*, *verbose=False*, *params='stmc'*, *init_params='stmc'*)

TDE_HMM is an Estimator that performs supervised decoding with a predefined number of decoding matrices. A clustering method is used to choose which decoding matrix to use for each sample of each input data.

Parameters

- **n_components** (*int*) – Number of states. Default 3
- **n_iter** (*int*) – Optional. Maximum number of iterations to perform.
- **covariance_type** (*str*) – Optional. The type of covariance parameters to use: * “spherical” — each state uses a single variance value that

  applies to all features (default).

  – ”diag” — each state uses a diagonal covariance matrix.

  – ”full” — each state uses a full (i.e. unrestricted) covariance matrix.

  – ”tied” — all states use **the same** full covariance matrix.

- **min_covar** (*float*) – Optional. Floor on the diagonal of the covariance matrix to prevent overfitting. Defaults to 1e-3.
- **startprob_prior** (*array*) – Optional. Shape of (n_components,). Parameters of the Dirichlet prior distribution for `startprob_`.
- **transmat_prior** (*array*) – Optional. Shape of (n_components, n_components). Parameters of the Dirichlet prior distribution for each row of the transition probabilities `transmat_`.
- **means_prior/means_weight** (*array*) – Optional. Shape of (n_components,). Mean and precision of the Normal prior distribtion for `means_`.
- **covars_prior/covars_weight** (*array*) – Optional. Shape of (n_components,). Parameters of the prior distribution for the covariance matrix `covars_`. If `covariance_type` is “spherical” or “diag” the prior is the inverse gamma distribution, otherwise — the inverse Wishart distribution.
- **algorithm** (*str*) – Optional. {“viterbi”, “map”} Decoder algorithm.
- **random_state** (*int_seed*) – Optional. A random number generator instance.
- **tol** (*float*) – Optional. Convergence threshold. EM will stop if the gain in log-likelihood is below this value.
- **verbose** (*bool*) – Optional. Whether per-iteration convergence reports are printed to `sys.stderr`. Convergence can also be diagnosed using the `monitor_` attribute.
- **params/init_params** (*str*) – Optional. The parameters that get updated during (`params`) or initialized before (`init_params`) the training. Can contain any combination of ‘s’ for startprob, ‘t’ for transmat, ‘m’ for means, and ‘c’ for covars. Defaults to all parameters.
- **implementation** (*str*) – Optional. Determines if the forward-backward algorithm is implemented with logarithms (“log”), or using scaling (“scaling”). The default is to use logarithms for backwards compatability.

fit (*X*, *y=None*)

Estimate model parameters.

Parameters

- **X** (*array-like*) – The training input samples
- **y** (*array-like*) – The target values

Returns Returns self

predict_proba(*X*, *y=None*)

Find most likely state sequence corresponding to X, then computes y_predict, the predicted labels given X.

Parameters **X** – The training input samples

Returns Returns posteriors, the predicted labels in an tensor

CHAPTER

FOUR

EXAMPLES

On this page you will find basic examples of how to use the different Estimators of our library. Don't hesitate and look for yourself!

4.1 Sequential cluster detection with decoding analysis (Cluster_Decoder)

This notebook shows the capabilities of our decoder. Several signals are generated according to 4 different models for an input X constant, X random and X sinusoidal with additional noise.

4.1.1 Librairies importation

[138]:
```
print(f"Importing librairies ... ", end='')
import sys
sys.path.append(r"C:\Users\valentin\Documents\HMM")
import numpy as np
from myHmmPackage.cluster_decoder import ClusterDecoder
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_percentage_
↪error
import matplotlib.pyplot as plt
print('done')

print("Creating a directory for saving the figures ...", end="")
FIG_PATH = "data_generation_figures"
import os
if not os.path.exists(FIG_PATH):
    os.makedirs(FIG_PATH)
print(f"done - All figures will be downloaded under the directory '{FIG_PATH}/'")
```

```
Importing librairies ... done
Creating a directory for saving the figures ...done - All figures will be downloaded␣
↪under the directory 'data_generation_figures/'
```

4.1.2 Hyperparameters

[139]:
```
### data ###
n_samples = 100 #n_trials
n_time_points = 100
n_regions = 2 #IC ou region ou electrod ou channels
n_label_features = 48
n_clusters = 4 # n_clusters
```

4.1.3 Data generation

[157]:
```
def get_sinusoidal_X() :
    def a(t,r):
        idx = np.count_nonzero(transition <= t) - 1
        return a_list[idx,r]

    def omega(t,r):
        idx = np.count_nonzero(transition <= t) - 1
        return omega_list[idx,r]

    def f(s,t,r):
        return a(t,r)*np.cos(omega(t,r)*t) + sigma[s]*np.random.normal(loc=0,␣
↪scale=sigma[s])


    return np.array([[[f(s,t,r) for r in range(n_regions)]
                           for t in range(n_time_points)]
                           for s in range(n_samples)],
                    dtype=float)

def get_random_X():
    return np.random.uniform(low=-10, high=10, size=(n_samples, n_time_points, n_
↪regions))

def get_constant_X():
    return np.random.uniform(low=-10, high=10)*np.ones(shape=(n_samples, n_time_
↪points, n_regions))

def get_mix_models(gamma,W):
    gamma = np.transpose(gamma,(1,0))
    W_temp = np.reshape(W, newshape=(n_regions, n_clusters, n_label_features))
    return np.sum(gamma @ W_temp, axis=1)

def get_y(X, W):
    y = X @ W
    y = np.reshape(y, newshape=(n_samples, n_time_points, n_label_features))
    return y
```

[158]:
```
transition = np.random.randint(n_time_points,size=(n_clusters-1))
transition = np.concatenate(([0],np.sort(transition),[n_time_points]))
a_list = np.random.uniform(low=0,high=2*np.pi/n_time_points, size=(n_clusters,n_
↪regions))
omega_list = np.random.uniform(low=0,high=2*np.pi/n_time_points, size=(n_clusters,n_
↪regions))
sigma = np.random.uniform(low=0.001,high=0.1,size=n_samples)
```

(continues on next page)

(continued from previous page)

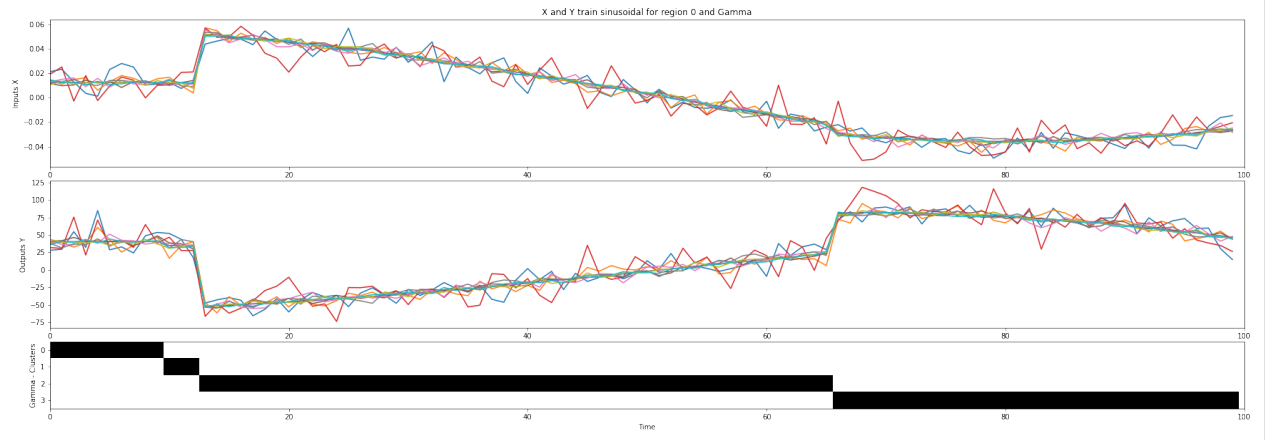
```
data_type = 'sinusoidal' # valid input : constant, random, sinusoidal

if data_type == 'constant' :
    print("Creating a directory for saving the figures ...", end="")
    if not os.path.exists(FIG_PATH+"/"+data_type):
        os.makedirs(FIG_PATH+"/"+data_type)
    print(f"done - All figures will be downloaded under the directory '{FIG_PATH}/
↪{data_type}/'")
    X_train = get_constant_X()
    X_test = get_constant_X()

elif data_type == 'random' :
    print("Creating a directory for saving the figures ...", end="")
    if not os.path.exists(FIG_PATH+"/"+data_type):
        os.makedirs(FIG_PATH+"/"+data_type)
    print(f"done - All figures will be downloaded under the directory '{FIG_PATH}/
↪{data_type}/'")
    X_train = get_random_X()
    X_test = get_random_X()

elif data_type == 'sinusoidal' :
    print("Creating a directory for saving the figures ...", end="")
    if not os.path.exists(FIG_PATH+"/"+data_type):
        os.makedirs(FIG_PATH+"/"+data_type)
    print(f"done - All figures will be downloaded under the directory '{FIG_PATH}/
↪{data_type}/'")
    X_train = get_sinusoidal_X()
    X_test = get_sinusoidal_X()

else :
    raise ValueError(f"Input received : '{data_type}. Valid input : 'constant',
↪'random', 'sinusoidal'")

W = np.random.randint(low=-10, high=10,size=(n_clusters, n_regions, n_label_features))

gamma = np.array([[0 if (transition[(j)] <= i) and (i < transition[(j+1)]) else 1 for␣
↪i in range(n_time_points)]
                  for j in range(n_clusters)],dtype=int)
W_mix = get_mix_models(gamma, W)

y_train = get_y(X_train, W_mix)
y_test = get_y(X_test, W_mix)
```

```
Creating a directory for saving the figures ...done - All figures will be downloaded␣
↪under the directory 'data_generation_figures/sinusoidal/'
```

4.1.4 Plot

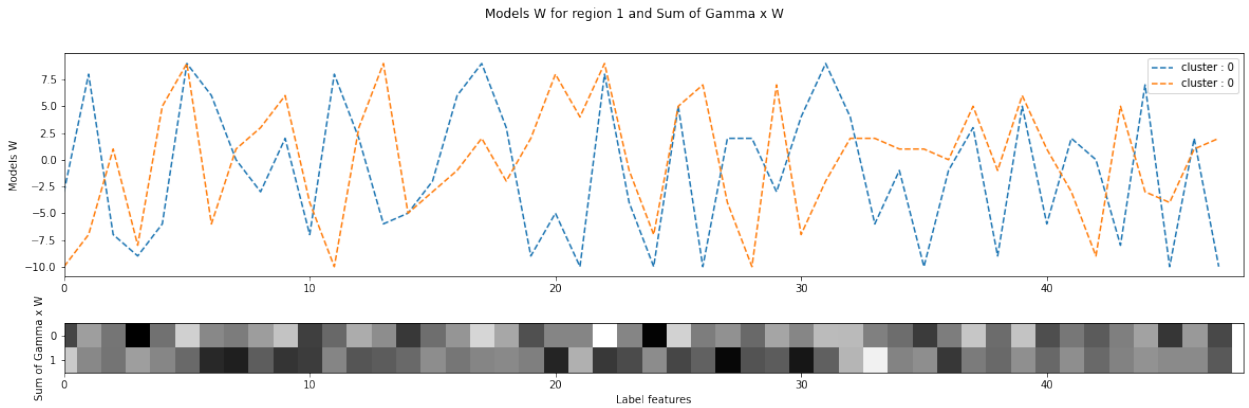
Inputs X for a given region id_r

[159]:
```
id_r = 0
fig = plt.figure(figsize=(30,10))

grid = plt.GridSpec(5, 1)
ax0 = fig.add_subplot(grid[:2, 0])
for id_s in range(10):
    ax0.plot(range(n_time_points), X_train[id_s,:,id_r])
plt.ylabel('Inputs X')
plt.xlim(0,n_time_points)
plt.title(f"X and Y train {data_type} for region {id_r} and Gamma")


ax1 = fig.add_subplot(grid[2:4, 0])
for id_s in range(10):
    ax1.plot(range(n_time_points), y_train[id_s,:,id_r])
plt.ylabel('Outputs Y')
plt.xlim(0,n_time_points)


ax2 = fig.add_subplot(grid[4, 0])
ax2.imshow(gamma, cmap='gray', aspect='auto')
plt.ylabel("Gamma - Clusters")
plt.xlabel('Time')
plt.xlim(0,n_time_points)

plt.savefig(FIG_PATH+"/"+data_type+"/x_train_gamma.png")
plt.show()
```



Models W

[160]:
```
id_cluster = 0
fig = plt.figure(figsize=(20,6))
grid = plt.GridSpec(3, 1)

ax0 = fig.add_subplot(grid[:2, 0])
for id_r in range(n_regions):
    plt.plot(W[id_cluster,id_r,:].T, '--', label=f"cluster : {id_cluster}" )
plt.ylabel("Models W")
plt.xlim(0,n_label_features)
plt.legend()

ax1 = fig.add_subplot(grid[2, 0])
plt.imshow(W_mix,cmap='gray')
plt.ylabel("Sum of Gamma x W")
plt.xlim(0,n_label_features)

plt.xlabel("Label features")
plt.suptitle(f"Models W for region {id_r} and Sum of Gamma x W")

plt.savefig(FIG_PATH+"/"+data_type+"/models_W_;_sum_gamma_W.png")
plt.show()
```



4.1.5 Training

[161]:
```
print(f"Starting cluster decoder ... ")
clf = ClusterDecoder(n_clusters=n_clusters, method='sequential', max_iter=1e3)
clf.fit(X_train,y_train) # renvoie gamma et W
print('done\n')
```

```
Starting cluster decoder ...
```

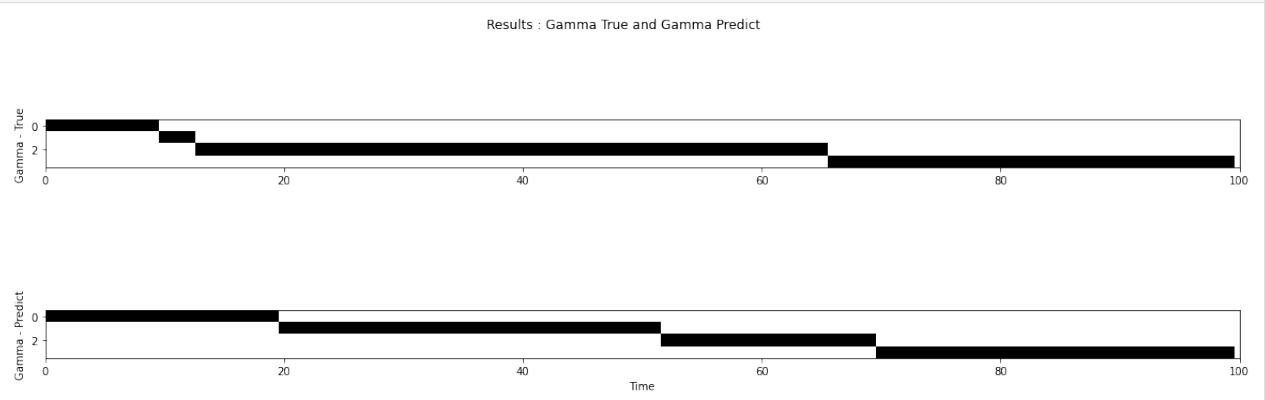
```
100%|| 999/999 [00:08<00:00, 111.74it/s]
```

```
done
```

4.1.6 Results

Gamma

[162]:
```
fig = plt.figure(figsize=(20,6))
grid = plt.GridSpec(2, 1)

ax0 = fig.add_subplot(grid[:1, 0])
plt.imshow(gamma,cmap='gray')
plt.ylabel("Gamma - True")
plt.xlim(0,n_time_points)

ax1 = fig.add_subplot(grid[1:, 0])
plt.imshow(1-clf.gamma_.T,cmap='gray')
plt.ylabel("Gamma - Predict")
plt.xlim(0,n_time_points)


plt.xlabel("Time")
plt.suptitle(f"Results : Gamma True and Gamma Predict")

plt.savefig(FIG_PATH+"/"+data_type+"/gamma_true_gamma_predict.png")
plt.show()
```



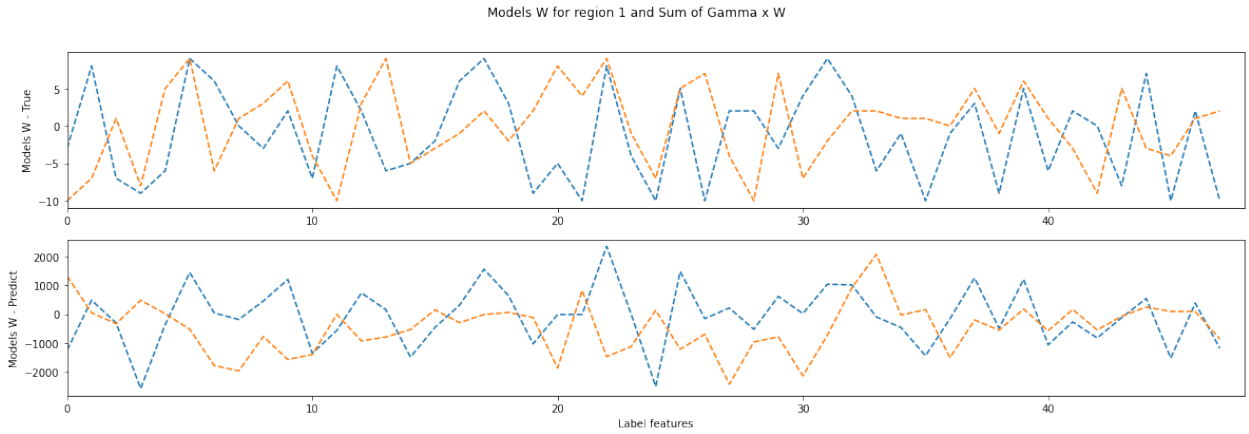
Models W

[163]:
```
id_cluster = 0
fig = plt.figure(figsize=(20,6))
grid = plt.GridSpec(2, 1)

ax0 = fig.add_subplot(grid[:1, 0])
for id_r in range(n_regions):
    plt.plot(W[id_cluster,id_r,:].T, '--', label=f"cluster : {id_cluster}" )
plt.ylabel("Models W - True")
plt.xlim(0,n_label_features)

ax1 = fig.add_subplot(grid[1:, 0])
for id_r in range(n_regions):
    plt.plot(clf.decoding_mats_[id_cluster,id_r,:].T, '--', label=f"cluster : {id_
↪cluster}" )
plt.ylabel("Models W - Predict")
```

(continues on next page)

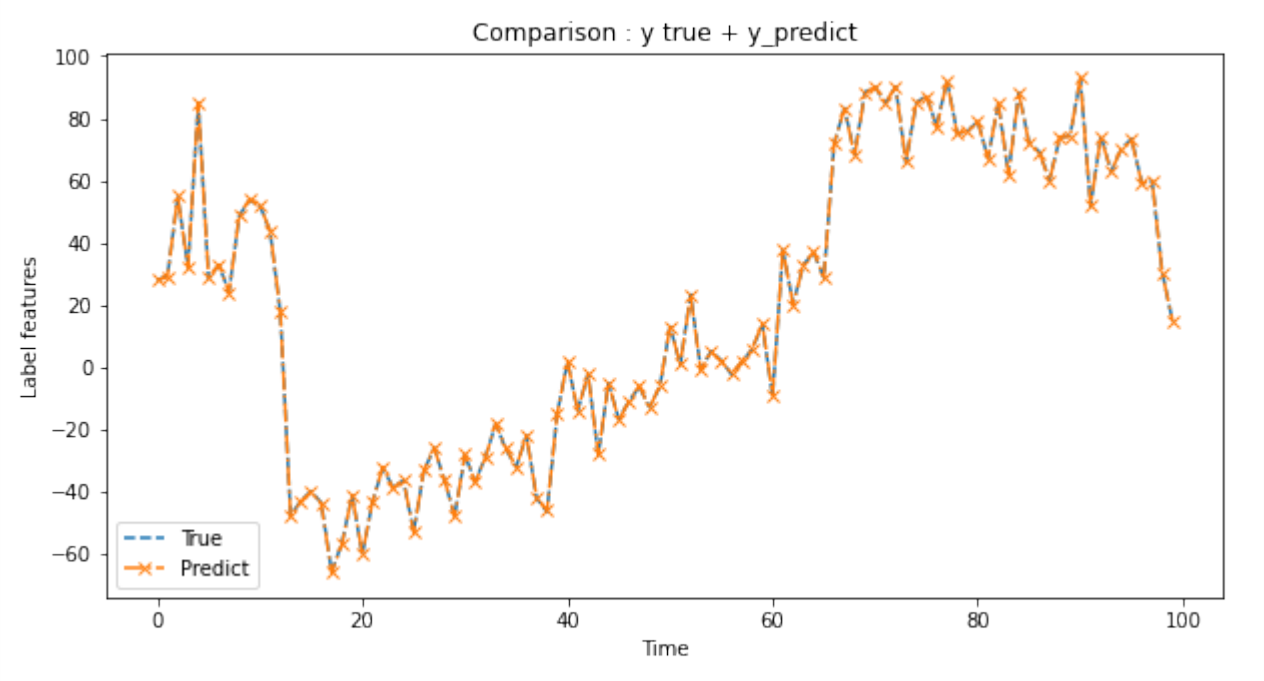
(continued from previous page)

```
plt.xlim(0,n_label_features)


plt.xlabel("Label features")
plt.suptitle(f"Models W for region {id_r} and Sum of Gamma x W")

plt.savefig(FIG_PATH+"/"+data_type+"/results_W_true;W_predict.png")
plt.show()
```



4.1.7 Test

[164]:
```
id_sample = 0
id_feat = 0
plt.figure(figsize=(10,5))
plt.plot(y_train[id_sample,:,id_feat].T, '--', label="True")
plt.plot(clf.predict(X_train)[id_sample,:,id_feat].T, '-.x', label="Predict")
plt.legend()
plt.xlabel("Time")
plt.ylabel("Label features")
plt.title("Comparison : y true + y_predict")
plt.savefig(FIG_PATH+"/"+data_type+"/results_y_train_;_y_predict")
plt.show()
```



[165]:
```
y_true = y_train.flatten()
y_predict = clf.predict(X_train).flatten()
print(f"r2_score : {r2_score(y_true, y_predict)}")
print(f"mean_squared_error : {mean_squared_error(y_true, y_predict)}")
print(f"mean_absolute_percentage_error : {mean_absolute_percentage_error(y_true, y_
↪predict)}")
```

```
r2_score : 0.9999372615674609
mean_squared_error : 0.08354959988271655
mean_absolute_percentage_error : 0.048958484919800684
```

[ ]:
```
```

4.2 TDE-HMM for wide-range burst detection

In this example, we show how our scikit-learn-like TDE_HMM Estimator can be used for wide-range burst detection in real EEG data. - First we will load the EEG data and visualize it, - Then we will use the TDE_HMM Class to infer the parameters of our time-delay embedded hidden Markov model (TDE-HMM), - Finally we will use the TDE_HMM Class to predict the probability of presence of each state considering our EEG data.

[1]:
```
# Storage management
import xarray as xr    # Manages .nc (netCDF) files in Python.
                       # The states' informations are stored in a .nc file for each␣
↪subject.

# Scientific computing
import numpy as np
import scipy.signal as signal

# Visualization
import matplotlib.pyplot as plt
```

4.2.1 1. Loading EEG data

[2]:
```
# Get our data
import os
import sys
sys.path.append(r"D:\centrale\3A\info\HMM\myHmmPackage")

subj=2
IC=1

dirname = "../"
filename = dirname + f"data/su{subj}IC{IC}_rawdata.nc"
filename2 = dirname + f"data/su{subj}IC{IC + 2}_rawdata.nc"
ds = xr.open_dataset(filename)
ds2 = xr.open_dataset(filename2)
ds
```

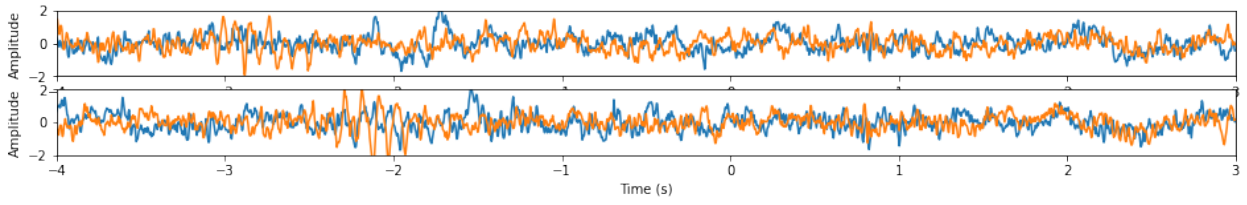
[2]:
```
<xarray.Dataset>
Dimensions:     (time: 1793, trials: 675, info: 1)
Coordinates:
  * time        (time) float64 -4.0 -3.996 -3.992 -3.988 ... 2.992 2.996 3.0
Dimensions without coordinates: trials, info
Data variables:
    timecourse  (trials, time) float64 ...
    trialinfo   (trials, info) float64 ...
```

[3]:
```
# X is the signal timecourse
X1 = ds['timecourse'].values[:, :, np.newaxis]
X2 = ds2['timecourse'].values[:, :, np.newaxis]
X = np.concatenate((X1, X2), axis=2)

# time is the time axis
time = ds['time'].values
```

[4]:
```
print(X.shape, time.shape)
```

```
(675, 1793, 2) (1793,)
```

[5]:
```
n = 2

plt.figure(figsize=(16, n))
for i in range(n):
    plt.subplot(n, 1, i+1)
    plt.plot(time, X[i])
    plt.xlim([-4, 3])
    plt.xlabel('Time (s)')
    plt.ylim([-2, 2])
    plt.ylabel('Amplitude')
```

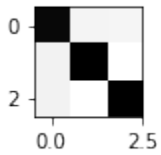


[6]:
```
from myHmmPackage.tde_hmm import TDE_HMM

n_states = 3

hmm = TDE_HMM(n_components=n_states)
hmm.fit(X)
```

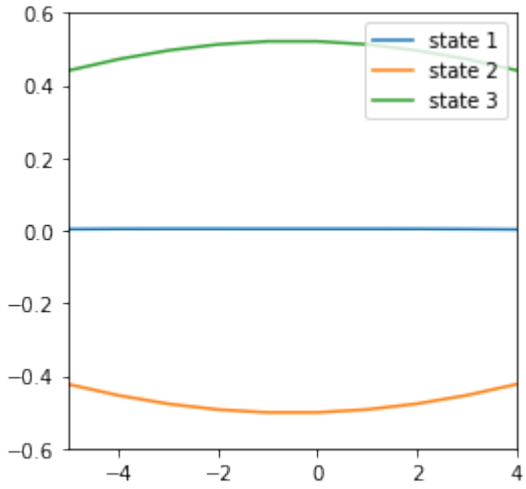
[6]:
```
TDE_HMM()
```

[13]:
```
print("Starting probability of the HMM states: \n\n", hmm.startprob_, "\n\n")

n_states = 3

print("Transition matrix of the HMM states: \n\n", hmm.transmat_, "\n\n")
plt.figure(figsize=(1,1))
plt.imshow(hmm.transmat_, cmap='gray_r')
plt.show()

print("Mean-array for each state:")
plt.figure(figsize=(4, 4))
for state in range(n_states):
    plt.plot(np.arange(-5, 5), hmm.means_[state])
    plt.xlim([-5,4])
    plt.ylim([-0.6,0.6])
plt.legend([f'state {i+1}' for i in range(n_states)], loc='upper right')
plt.show()

print("Covariance matrix for each state:")
plt.figure(figsize=(4*n_states, 4))
for state in range(n_states):
    plt.subplot(1, n_states, state+1)
    plt.imshow(hmm.covars_[state])
    plt.colorbar()
plt.show()
```

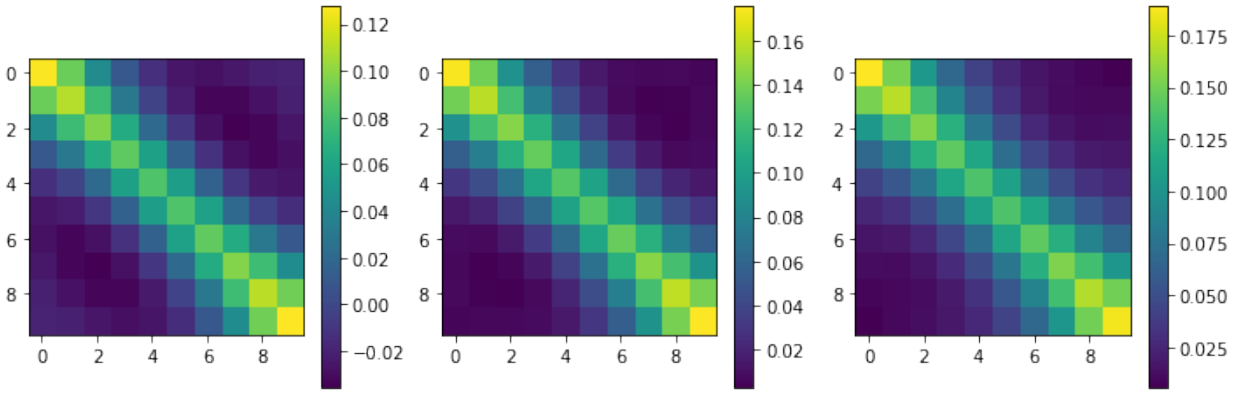
```
Starting probability of the HMM states:

 [1.00000000e+00 7.64920307e-16 3.42928429e-16]


Transition matrix of the HMM states:

 [[9.19389757e-01 4.10661085e-02 3.95441346e-02]
 [4.88116905e-02 9.51188264e-01 4.56849476e-08]
 [4.94738473e-02 1.72991423e-08 9.50526135e-01]]
```




```
Mean-array for each state:
```




```
Covariance matrix for each state:
```

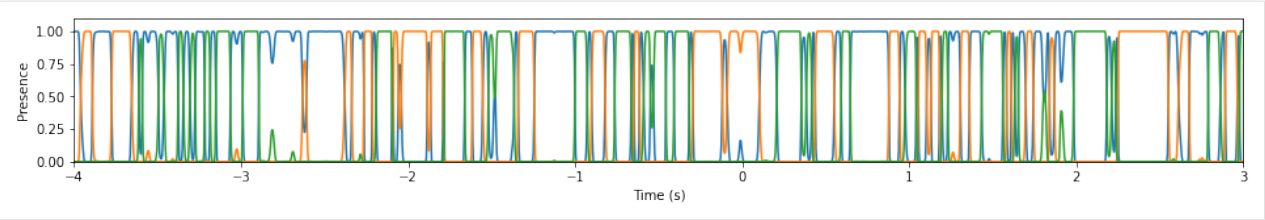


[ ]:
```
plt.figure(figsize=(16, 2))
plt.xlim([-4, 3])
plt.xlabel('Time (s)')
plt.ylim([0, 1.1])
plt.ylabel('Probability of Presence')
```

[7]:
```
Gamma = hmm.predict_proba(X)
print(Gamma.shape, time.shape)
```

[11]:
```
plt.figure(figsize=(16, 2))
plt.plot(time, Gamma[0])
plt.xlim([-4, 3])
plt.xlabel('Time (s)')
plt.ylim([0, 1.1])
plt.ylabel('Probability of Presence')
```

[11]:
```
Text(0, 0.5, 'Presence')
```


1.00
0.75
0.50
0.25
0.00
Presence
−4
−3
−2
−1
0
1
2
3
Time (s)

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX