CodingLab2

May 2, 2022

Neural Data Science

Lecturer: Prof. Dr. Philipp Berens

Tutors: Jonas Beck, Ziwei Huang, Rita González Márquez

Summer term 2022

Student names: <Moritz Kniebel, Vanessa Tsingunidis>

1 Coding Lab 2

If needed, download the data files `nda_ex_1_*.npy` from ILIAS and save it in the subfolder `../data/`. Use a subset of the data for testing and debugging. But be careful not to make it too small, since the algorithm may fail to detect small clusters in this case.

[74]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
from scipy import signal
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import sklearn.metrics as metrics
import scipy as sp
from scipy.io import loadmat
import copy
from scipy import linalg

sns.set_style('whitegrid')
%matplotlib inline
```

1.1 Load data

[75]:
```
# replace by path to your solutions
b = np.load('./data/nda_ex_1_features.npy')
s = np.load('./data/nda_ex_1_spiketimes_s.npy')
w = np.load('./data/nda_ex_1_waveforms.npy')
```

```
np.random.seed(0)
```

1.2 Task 1: Generate toy data

Sample 1000 data points from a two dimensional mixture of Gaussian model with three clusters and the following parameters:

$$\mu_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \Sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \pi_1 = 0.3$$

$$\mu_2 = \begin{bmatrix} 5 \\ 1 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \pi_2 = 0.5$$

$$\mu_3 = \begin{bmatrix} 0 \\ 4 \end{bmatrix}, \Sigma_3 = \begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix}, \pi_3 = 0.2$$

Plot the sampled data points and indicate in color the cluster each point came from. Plot the cluster means as well.

Grading: 1 pts

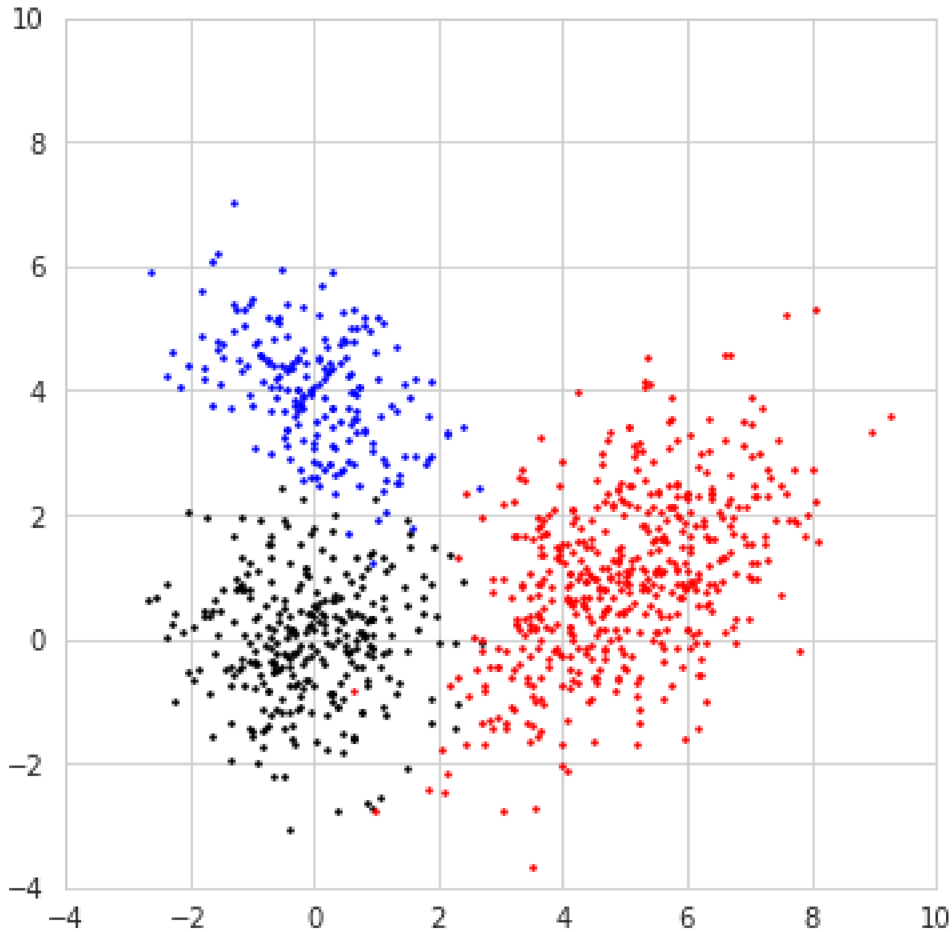
[76]:
```
def sample_data(N, m, S, p):

    '''Generate N samples from a Mixture of Gaussian distribution with
    means m, covariances S and priors p.

    Parameters
    ----------

    N: int
        Number of samples

    m: list or np.array, (n_clusters, n_dims)
        Means

    S: list or np.array, (n_clusters, n_dims, n_dims)
        Covariances

    p: list or np.array, (n_clusters, )
        Cluster weights / probablities

    Returns
    -------

    x: np.array, (n_samples, n_dims)
        Data points

    ind: np.array, (N,)
        Labels.
    '''
```

```python
    samples_p_cluster = p*N  # number of samples per cluster according to the␣
↪probability/weight
    labels = []
    x = []

    for cluster in range(len(p)):
        # sample from Gaussian
        cluster_data = np.random.multivariate_normal(m[cluster], S[:,:,cluster],␣
↪int(samples_p_cluster[cluster]))
         # add data point to list data points
        x.append(cluster_data)
        # save label for cluster and add to list labels
        cluster_label = np.ones(int(samples_p_cluster[cluster]))*cluster
        labels.append(cluster_label)
    x = np.concatenate(x)
    labels = np.concatenate(labels)

    # ------------------------------------------------------
    # draw labeled points from mixture of Gaussians (0.5 pt)
    # ------------------------------------------------------

    return (labels, x)
```

[77]:
```python
N = 1000 # total number of samples
m = np.array([[0, 0], [5, 1], [0, 4]])
S_1 = np.array([[1, 0], [0, 1]])
S_2 = np.array([[2, 1], [1, 2]])
S_3 = np.array([[1, -.5], [-.5, 1]])
S = np.concatenate((S_1[:,:,np.newaxis],
                    S_2[:,:,np.newaxis],
                    S_3[:,:,np.newaxis]), axis=2)
p = np.array([.3, .5, .2])


labels, x = sample_data(N, m, S, p)
```

[78]:
```python
plt.figure(figsize=(6, 6))

ax = plt.subplot(1,1,1, aspect='equal')
plt.plot(x[labels==0,0],x[labels==0,1],'.k', markersize=3)
plt.plot(x[labels==1,0],x[labels==1,1],'.r', markersize=3)
plt.plot(x[labels==2,0],x[labels==2,1],'.b', markersize=3)

plt.xlim((-4,10))
plt.ylim((-4,10))
```

```
plt.show()
```



1.3 Task 2: Implement a Gaussian mixture model

Implement the EM algorithm to fit a Gaussian mixture model in `fit_mog()`. Sort the data points by inferring their class labels from your mixture model (by using maximum a-posteriori classification). Fix the seed of the random number generator to ensure deterministic and reproducible behavior. Test it on the toy dataset specifying the correct number of clusters and make sure the code works correctly. Plot the data points from the toy dataset and indicate in color the cluster each point was assigned to by your model. How does the assignment compare to ground truth? If you run the algorithm multiple times, you will notice that some solutions provide suboptimal clustering solutions - depending on your initialization strategy.

Grading: 4 pts

```
[79]: def fit_mog(x,k, niters=100, random_seed=None):

          '''Fit Mixture of Gaussian model using EM algo.

          Parameters
          ----------

          x: np.array, (N, n_dims)
              Input data

          k: int
              Number of clusters

          niters: int
              Maximal number of iterations.

          random_seed: int or None
              Random Seed


          Returns
          -------

          labels: np.array, (n_samples)
              Cluster labels

          m: list or np.array, (n_clusters, n_dims)
              Means

          S: list or np.array, (n_clusters, n_dims, n_dims)
              Covariances

          p: list or np.array, (n_clusters, )
              Cluster weights / probablities
          '''



          ### Initialization with k-means

          np.random.seed(random_seed)

          D = x.shape[1] # number features
          N = x.shape[0] # number samples


          # choose inital clusters with k-means
```

```
    k_means = KMeans(n_clusters=k, init='random', random_state=random_seed).
↪fit(x)
    means = k_means.cluster_centers_
    m_k = np.zeros((N,D))

    # overall cov for initialization
    cov_init = np.array((1/N) * (x.T@x))
    covs = np.zeros((D, D, k))
    for i in range(k):
        covs[:,:,i] = np.array(cov_init)
    cov = np.zeros((D, D, N))
    # initialize latent variables array
    z = np.zeros((N, k))
    z_i = np.zeros((N, k))

    const = 1e-4*np.eye(D)
    epsilon = 1e-6
    last_means = np.zeros(means.shape)
    # equal class probabilities
    priors = (1/k) * np.ones(k)


    ### EM algorithm ###

    for i in range(niters):

        # E-step
        for j in range(k):
            # compute latent variables
            z[:,j] = sp.stats.multivariate_normal.pdf(x, mean=means[j].
↪squeeze(), cov=covs[:,:,j]) * priors[j]
        # normalize
        z_i = z / z.sum(axis=-1)[:,None]

        # M-step
        for j in range(k):
            # estimate priors
            priors[j] = (1/N)* np.sum(z_i[:,j])
            # estimate means
            for m in range(N):
                m_k[m,:] = z_i[m,j]*x[m]
            means[j] = np.sum(m_k, axis=0)/np.sum(z_i[:,j])
            # estimate covs
            for m in range(N):
                deviation = x[m]-means[j]
                cov[:,:,m] = z_i[m,j]*np.outer(deviation, deviation)
```

```
            covs[:,:,j] =  (1/np.sum(z_i[:,j]))*(np.sum(cov, axis=-1) + const) #␣
↪to prevent singular matrices add small value to diagonal
            # check cov for monitoring
            if (not np.all(np.linalg.eigvals(covs[:,:,j]) > 0)):
                # cov not positive definite reset
                covs[:,:,j] = cov_init
                print('Cov not positive def: reset')
            if np.trace(covs[:,:,j]) < 0.1*D:
                # cov disappears reset
                covs[:,:,j] = cov_init
                print('Cov disappeared: reset')
                # check for singular matrix
            if np.linalg.cond(covs[:,:,j]) < 1/np.finfo(x.dtype).eps: #1/sys.
↪float_info.epsilon:
                pass
            else:
                covs[:,:,j] = cov_init
                print('Cov singular: reset')

        # Print nb iterations
        if (i%10==0):
            print('Iteration '+str(i))


        ### Convergence
        mean_change = (last_means-means)**2
        # if change in mean estimation is sufficiently small end
        if np.sum(np.sum(mean_change, axis=-1))<=epsilon:
            print('EM converged')
            break
        last_means[:] = np.array(means)

    ### Assign cluster labels
    # label gets assigned to cluster with the highest probability
    labels = np.argmax(z_i,axis=-1)
    m = means
    S = covs
    p = priors



    # Use numpy broadcasting, Notes from tutorial:
    #win = np.arange[-10,20] # define window
    #wins = win + S[:, np.newaxis] # windows of all spikes
    #w = x[wins] # Extract shape of spike
```

```
    # -----------
    # init (1 pt)
    # -----------

    # -------------------------
    # EM maximisation (2.5 pts)
    # -------------------------

    return (labels, m, S, p)
```

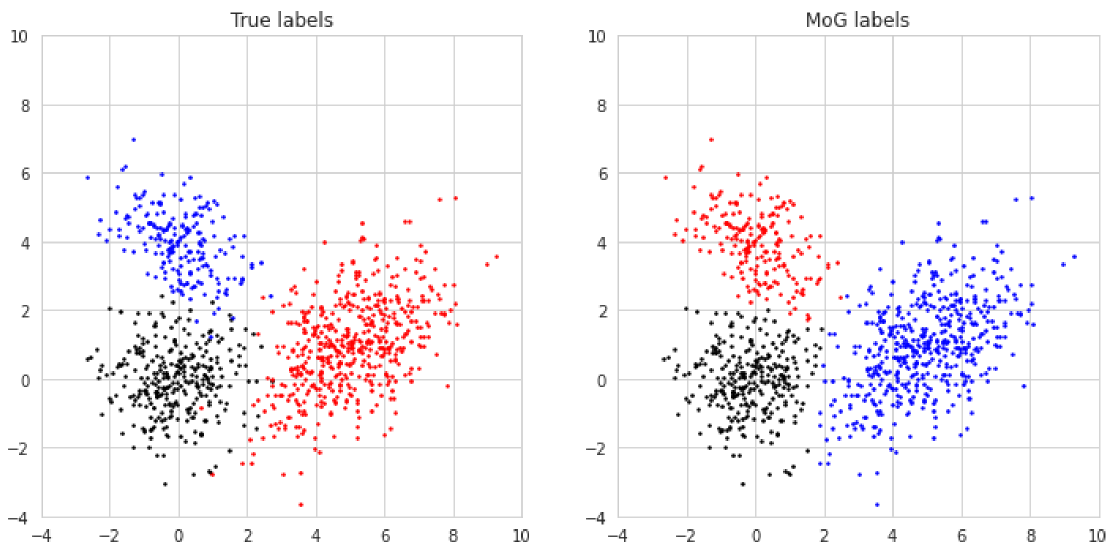
Run Mixture of Gaussian on toy data

[80]:
```
mog_labels, m, S, p = fit_mog(x,3, random_seed=42)
```

```
Iteration 0
Iteration 10
Iteration 20
EM converged
```

Plot toy data with cluster assignments and compare to original labels

[81]:
```
plt.figure(figsize=(12, 6))

ax = plt.subplot(1,2,1, aspect='equal')
plt.plot(x[labels==0,0],x[labels==0,1],'.k', markersize=3)
plt.plot(x[labels==1,0],x[labels==1,1],'.r', markersize=3)
plt.plot(x[labels==2,0],x[labels==2,1],'.b', markersize=3)

plt.xlim((-4,10))
plt.ylim((-4,10))
plt.title('True labels')

ax = plt.subplot(1,2,2, aspect='equal')
plt.plot(x[mog_labels==0,0],x[mog_labels==0,1],'.k', markersize=3)
plt.plot(x[mog_labels==1,0],x[mog_labels==1,1],'.r', markersize=3)
plt.plot(x[mog_labels==2,0],x[mog_labels==2,1],'.b', markersize=3)

plt.xlim((-4,10))
plt.ylim((-4,10))
plt.title('MoG labels')

plt.show()
```



With the recommended initialization method K-means, MoG and true labels compare well. Only for data points that lie in regions where the clusters overlap, MoG fails to assign the right labels.

1.4 Task 3: Model complexity

A priori we do not know how many neurons we recorded. Extend your algorithm with an automatic procedure to select the appropriate number of mixture components (clusters). Base your decision on the Bayesian Information Criterion:

$BIC = -2L + P \log N,$

where L is the log-likelihood of the data under the best model, P is the number of parameters of the model and N is the number of data points. You want to minimize the quantity. Plot the BIC as a function of mixture components. What is the optimal number of clusters on the toy dataset?

You can also use the BIC to make your algorithm robust against suboptimal solutions due to local minima. Start the algorithm multiple times and pick the best solutions for extra points. You will notice that this depends a lot on which initialization strategy you use.

Grading: 2 pts + 1 extra pt

[82]:
```python
# Note from tutorial: Take care of robustness
```

[83]:
```python
def mog_bic(x, m, S, p):

    '''Compute the BIC for a fitted Mixture of Gaussian modelRandom Seed

    Parameters
    ----------
```

```
    x: np.array, (n_samples, n_dims)
        Input data

    m: list or np.array, (n_clusters, n_dims)
        Means

    S: list or np.array, (n_clusters, n_dims, n_dims)
        Covariances

    p: list or np.array, (n_clusters, )
        Cluster weights / probablities

    Return
    ------

    bic: float
        BIC

    LL: float
        Log Likelihood
    '''

    N, D = x.shape
    k = m.shape[0]


    # Compute log likelihood
    LL = 0
    for j in range(k):
        LL += sp.stats.multivariate_normal.pdf(x, mean=m[j,:], cov=S[:,:,j]) *␣
↪p[j]
    LL = np.sum(np.log(LL))

    # compute number of parameters (cov + means + priors)
    num_P = k*D*(D-1)/2 + k*D + (k-1)

    # Compute BIC
    bic = -2*LL + num_P*np.log(N)
    # -------------------------
    # implement the BIC (1.5 pts)
    # -------------------------

    return (bic, LL)
```

[84]:
```python
#
↪---------------------------------------------------------------------------------
# Compute and plot the BIC for mixture models with different numbers of clusters
↪(e.g., 2 - 6). (0.5 pts)
# Make your algorithm robust against local minima. (1 extra pts)
#
↪---------------------------------------------------------------------------------

### Notes from tutorial
# robustness influenced by the way of initialization
# find way to initialize Gaussian mixture such that it is robust
# initialize it with k-means
# dont use random initialization for Gaussian mixture
```

[85]:
```python
K = [2,3,4,5,6]
bic = np.zeros((3,len(K)))
LL = np.zeros((3,len(K)))


for j,k in zip(range(len(K)), K):
    for i in range(3):
      _, m, S, p = fit_mog(x,k,random_seed=i)
      bic[i,j], LL[i,j] = mog_bic(x, m, S, p)
    print('MoG and BIC iterations ' + str(j+1))
```

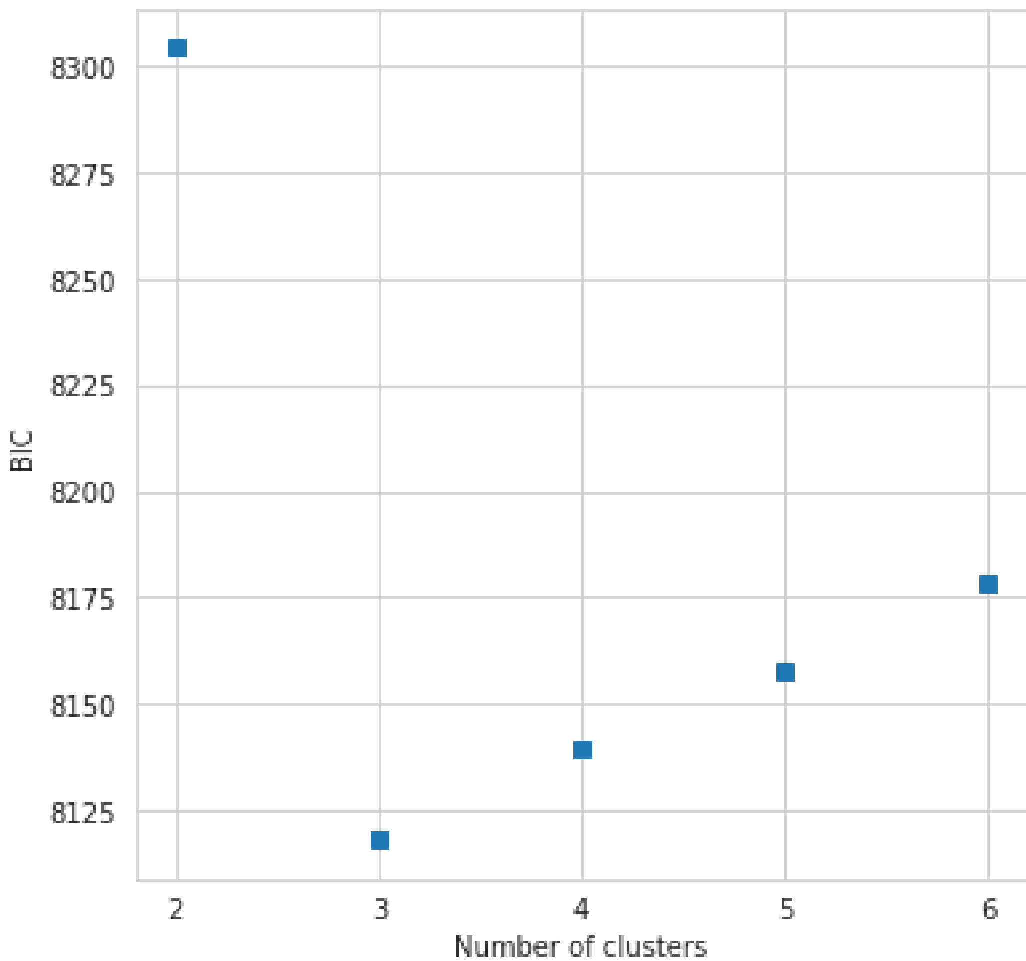
```
Iteration 0
Iteration 10
EM converged
Iteration 0
Iteration 10
EM converged
Iteration 0
Iteration 10
EM converged
MoG and BIC iterations 1
Iteration 0
Iteration 10
Iteration 20
EM converged
Iteration 0
Iteration 10
Iteration 20
EM converged
Iteration 0
Iteration 10
Iteration 20
EM converged
MoG and BIC iterations 2
```

```
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 3
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
```

```
Iteration 70
Iteration 80
Iteration 90
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 4
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 5
```

[86]:

```
plt.figure(figsize=(6,6))


plt.plot(K,np.min(bic,axis=0),'s')
plt.xlabel('Number of clusters')
plt.ylabel('BIC')
plt.xticks(K)
plt.xlim((1.8,6.2))

plt.show()
```



1.5 Task 4: Spike sorting using Mixture of Gaussian

Run the full algorithm on your set of extracted features (including model complexity selection). Plot the BIC as a function of the number of mixture components on the real data. For the best model, make scatter plots of the first PCs on all four channels (6 plots). Color-code each data point according to its class label in the model with the optimal number of clusters. In addition, indicate the position (mean) of the clusters in your plot.

Grading: 3 pts

[87]:
```
# ---------------------------------------------------------------------------
# Select the model that best represents the data according to the BIC (1 pt)
# ---------------------------------------------------------------------------

K = np.arange(2,14)
BIC = np.zeros(len(K))
LL = np.zeros(len(K))

for j in np.arange(0,len(K)):
    _, m, S, p = fit_mog(b,K[j])
    BIC[j], LL[j] = mog_bic(b, m, S, p)
    print('MoG and BIC iterations ' + str(j+1))
```

```
Iteration 0
Iteration 10
Iteration 20
Iteration 30
EM converged
MoG and BIC iterations 1
Iteration 0
Iteration 10
Iteration 20
Iteration 30
EM converged
MoG and BIC iterations 2
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
EM converged
MoG and BIC iterations 3
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
```

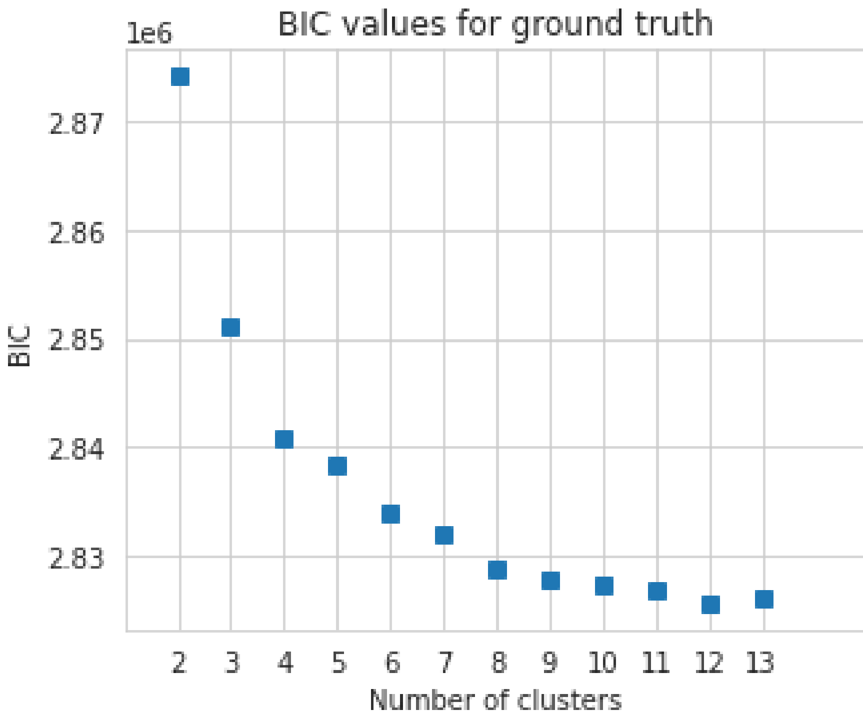
```
Iteration 80
Iteration 90
MoG and BIC iterations 4
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 5
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 6
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 7
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 8
Iteration 0
```

```
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 9
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 10
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 11
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
MoG and BIC iterations 12
```

[88]:

```
plt.figure(figsize=(5, 4))

plt.plot(K,BIC,'s')
```

```
plt.xlabel('Number of clusters')
plt.ylabel('BIC')
plt.xticks(K)
plt.xlim((1,15))
plt.title('BIC values for ground truth')

plt.show()
```



Refit model with lowest BIC and plot data points

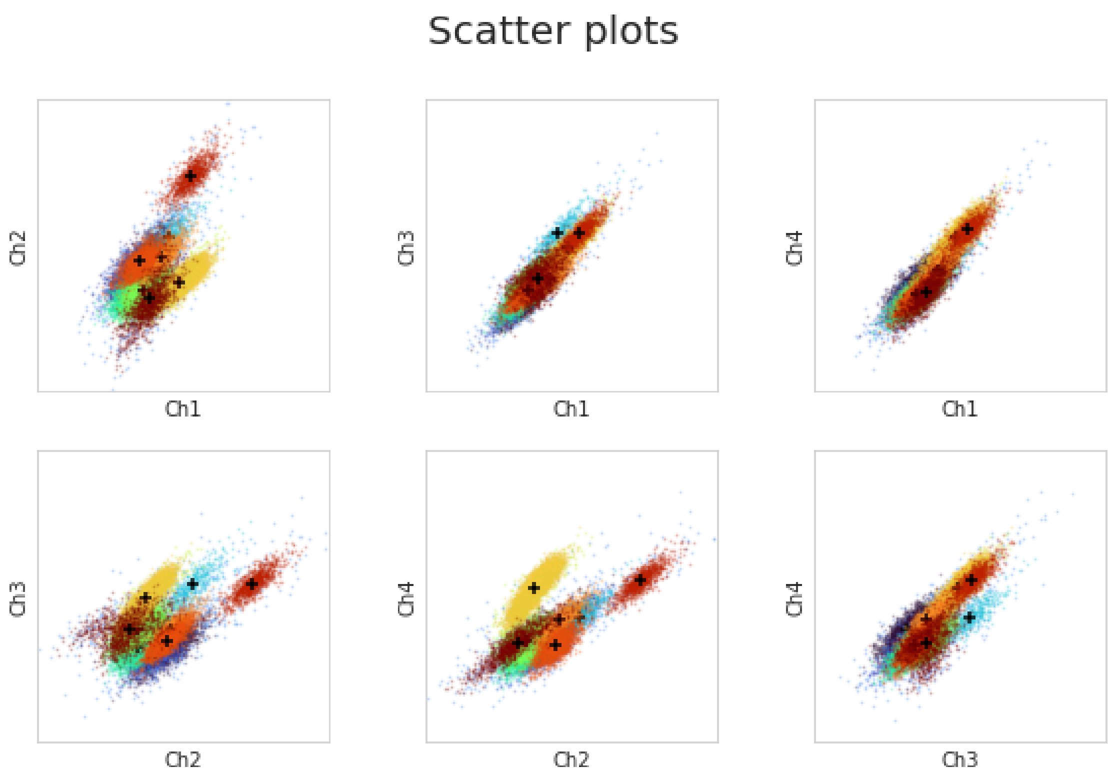
[89]:
```
# refit model
a = int(K[BIC==np.min(BIC)])
print('The optimal number of clusters: ' + str(a))
label, m, S, p = fit_mog(b,a)
```

```
The optimal number of clusters: 12
Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
```

```
Iteration 60
Iteration 70
Iteration 80
Iteration 90
```

[96]:

```
#␣
↪------------------------------------------------------------------------------------
# Create scatterplots of the first PCs under the best model for all 4 channels.␣
↪(2 pts)
#␣
↪------------------------------------------------------------------------------------

colors = plt.cm.turbo(np.linspace(0,1,a))

plt.figure(figsize=(10, 6))
plt.suptitle('Scatter plots',fontsize=20)

idx = [0, 3, 6, 9]
p = 1
channels = ['Ch1','Ch2','Ch3','Ch4']
for i in np.arange(0,4):
    for j in np.arange(i+1,4):
        ax = plt.subplot(2,3,p, aspect='equal')
        for k in range(a):
          plt.scatter(b[label==k,idx[i]],b[label==k,idx[j]],color=colors[k],s=.
↪7,alpha=.2)
          plt.scatter(m[k,idx[i]], m[k,idx[j]],color='k',marker='+',s=40)
        plt.xlabel(channels[i])
        plt.ylabel(channels[j])
        plt.xlim((-1000,1500))
        plt.ylim((-1000,1500))
        ax.set_xticks([])
        ax.set_yticks([])
        p = p+1
plt.show()
```



[91]:
```
# np.save('../data/nda_ex_2_means',m)
# np.save('../data/nda_ex_2_covs',S)
# np.save('../data/nda_ex_2_pis',p)
# np.save('../data/nda_ex_2_labels',a)
```

[92]:
```
# Source of code for plots adapted from: https://github.com/berenslab/
↪neural_data_science
```