Neural Data Science

Lecturer: Prof. Dr. Philipp Berens

Tutors: Jonas Beck, Ziwei Huang, Rita González Márquez

Summer term 2023

Student names: *Kathrin Root, Alexander Wendt, Patrick Weygoldt*

Coding Lab 2

- **Data**: Use the saved data `nds_cl_1_*.npy` from Coding Lab 1. Or, if needed, download the data files `nds_cl_1_*.npy` from ILIAS and save it in the subfolder `../data/`.
- **Dependencies**: You don't have to use the exact versions of all the dependencies in this notebook, as long as they are new enough. But if you run "Run All" in Jupyter and the boilerplate code breaks, you probably need to upgrade them.

```python
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from IPython import embed

# from __future__ import annotations
```

```python
plt.style.use("../matplotlib_style.txt")
```

Load data

```python
# replace by path to your solutions
import pathlib

datapath = pathlib.Path("../data")
pc1s = np.load(datapath / "nds_cl_1_features.npy")
spiketimes = np.load(datapath / "nds_cl_1_spiketimes_s.npy")
waveforms = np.load(datapath / "nds_cl_1_waveforms.npy")
np.random.seed(1024)
```

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

```
def sample_data(
    N: int, m: np.ndarray, S: np.ndarray, p: np.ndarray
) -> tuple[np.ndarray, np.ndarray]:
    """Generate N samples from a Mixture of Gaussian distribution with
    means m, covariances S and priors p.

    Parameters
    ----------

    N: int
        Number of samples

    m: np.ndarray, (n_clusters, n_dims)
        Means

    S: np.ndarray, (n_clusters, n_dims, n_dims)
        Covariances

    p: np.ndarray, (n_clusters, )
        Cluster weights / probablities

    Returns
    -------

    labels: np.array, (n_samples, )
```

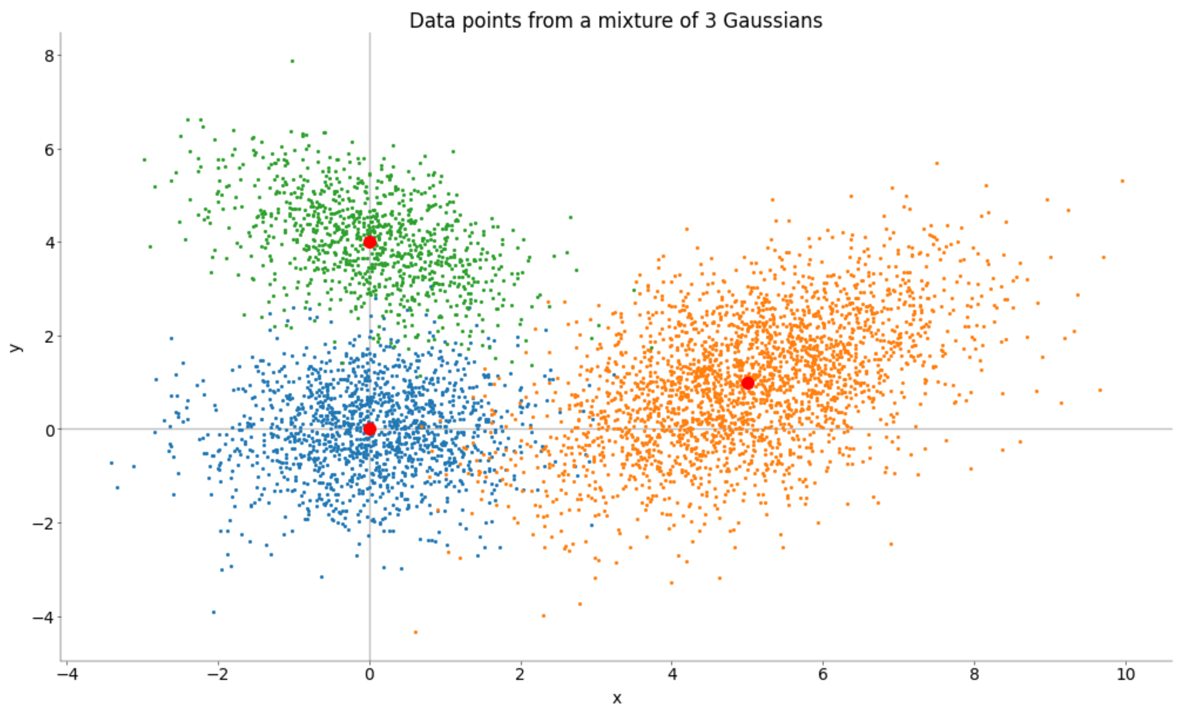
```python
        Grund truth labels.

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

    # make k cluster labels
    labels = np.arange(0, len(m[:, 0]))

    # itialize empty array to store the data
    data_values = np.zeros((N, len(m[0])))
    data_labels = np.zeros(N)

    # loop over the number of samples
    for iter in range(N):
        # assing the current sample to one of the clusters with probability p
        cluster = np.random.choice(labels, p=p)
        data_labels[iter] = cluster

        # sample from a 2D gaussian with mean m[k] and covariance S[k]
        data_values[iter, :] = np.random.multivariate_normal(m[cluster], S[cluster])

    return data_labels, data_values
```

```python
num_samples = 5000  # total number of samples

cluster_probs = np.array([0.3, 0.5, 0.2])  # percentage of each cluster
cluster_means = np.array([[0.0, 0.0], [5.0, 1.0], [0.0, 4.0]])  # means

S1 = np.array([[1.0, 0.0], [0.0, 1.0]])
S2 = np.array([[2.0, 1.0], [1.0, 2.0]])
S3 = np.array([[1.0, -0.5], [-0.5, 1.0]])
cluster_covs = np.stack([S1, S2, S3])  # cov

data_labels, data_values = sample_data(
    num_samples, cluster_means, cluster_covs, cluster_probs
)
```

```python
colors = ["tab:blue", "tab:orange", "tab:green", "tab:red", "tab:purple", "tab:brown"]
fig, ax = plt.subplots(figsize=(10, 6))
```

```python
for i, label in enumerate(np.unique(data_labels)):
    ax.scatter(
        data_values[data_labels == label, 0],
        data_values[data_labels == label, 1],
        marker=".",
        linewidths=0,
        s=20,
        c=colors[i],
    )
ax.scatter(cluster_means[:, 0], cluster_means[:, 1], marker="o", s=50, c="red")
ax.axvline(x=0, c="gray", linewidth=1, alpha=0.5, zorder=-1)
ax.axhline(y=0, c="gray", linewidth=1, alpha=0.5, zorder=-1)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Data points from a mixture of 3 Gaussians")
```

```
Text(0.5, 1.0, 'Data points from a mixture of 3 Gaussians')
```



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

```python
class GMM2D:
    def __init__(self, k, niters=10):
        self.k = k
        self.niters = niters

    def initialize(self, x):
        self.shape = x.shape
        self.n, self.d = self.shape

        self.prior_probs = np.ones(self.k) / self.k
        self.posterior_probs = np.ones(self.k) / self.k
        kmeans = KMeans(n_clusters=self.k, n_init="auto").fit(x)
        self.means = kmeans.cluster_centers_
        # self.means = np.random.choice(x.flatten(), (self.k, self.d))
        # self.cov = np.asarray([np.eye(self.d) for _ in range(self.k)])
        self.cov = np.asarray([np.cov(x.T) for _ in range(self.k)])

    def fit(self, x):
        self.initialize(x)
        for iter in range(self.niters):
            self.__expectation_step(x)
            self.__maximization_step(x)

    def predict(self, x):
        return np.argmax(self.__predict_probability(x), axis=1)

    def __predict_probability(self, x):
        # compute the likelihood of each data point that it comes from each of
        # n clusters
        likelihoods = np.zeros((self.n, self.k))
        for cluster in range(self.k):
            distribution = sp.stats.multivariate_normal(
                mean=self.means[cluster], cov=self.cov[cluster]
            )
            likelihoods[:, cluster] = distribution.pdf(x)

        # update the posterior probability of each cluster (probabilities of
        # the data to have been drawn from each of the gaussians)
        num = likelihoods * self.prior_probs
        denom = np.sum(num, axis=1)[:, np.newaxis]
        posterior_probs = num / denom

        return posterior_probs

    def __expectation_step(self, x):
        self.posterior_probs = self.__predict_probability(x)
        self.prior_probs = np.mean(self.posterior_probs, axis=0)

    def __maximization_step(self, x):
        epsilon = 1e-5  # a small number to avoid division by zero
        for cluster in range(self.k):
            # compute the posterior probability of each cluster (probabilities of
            # the data to have been drawn from each of the gaussians)
            # by updating our initial guess (the prior probabilities) with
            # the likelihoods
            posterior_prob = self.posterior_probs[:, cluster]
            total_posterior_prob = np.sum(posterior_prob)
            self.means[cluster] = np.sum(posterior_prob * x.T, axis=1) / (
                total_posterior_prob + epsilon
            )
            self.cov[cluster] = (
                np.cov(
                    x.T,
                    aweights=(posterior_prob / (total_posterior_prob + epsilon)),
                    bias=True,
                )
                + np.identity(self.d) * epsilon
            )
```

Run Mixture of Gaussian on toy data

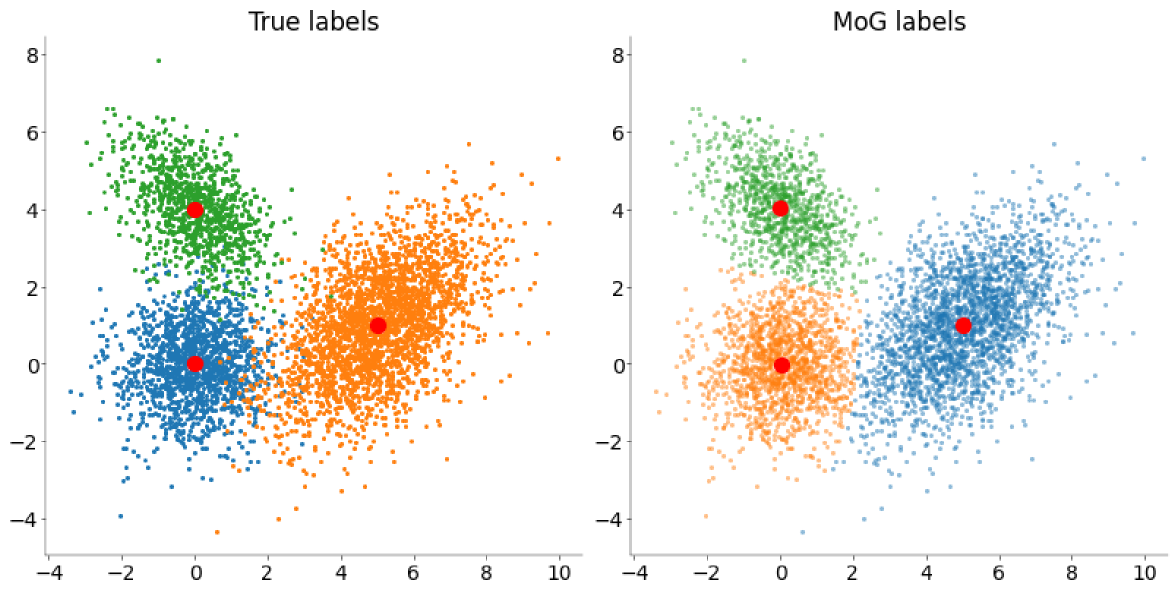
```
np.random.seed(42)
gmm = GMM2D(3, niters=30)
gmm.fit(data_values)
pred_labels = gmm.predict(data_values)

print(f"Predicted means: {gmm.means}")
```

```
Predicted means: [[ 5.00678622  1.01123488]
 [ 0.02582009 -0.00589657]
 [ 0.00854705  4.04854545]]
```

Plot toy data with cluster assignments and compare to original labels

```
mosaic = [["True", "MoG"]]
fig, ax = plt.subplot_mosaic(mosaic=mosaic, figsize=(8, 4), layout="constrained")

for i, label in enumerate(np.unique(data_labels)):
    ax["True"].set_title("True labels")
    ax["True"].scatter(
        data_values[data_labels == label, 0],
        data_values[data_labels == label, 1],
        marker=".",
        linewidths=0,
        s=20,
        c=colors[i],
    )
ax["True"].scatter(cluster_means[:, 0], cluster_means[:, 1], marker="o", s=50, c="red")
for i, predicted in enumerate(np.unique(pred_labels)):
    ax["MoG"].set_title("MoG labels")
    ax["MoG"].scatter(
        data_values[pred_labels == predicted, 0],
        data_values[pred_labels == predicted, 1],
        marker=".",
        linewidths=0,
        s=20,
        c=colors[i],
        alpha=0.5,
    )
ax["MoG"].scatter(gmm.means[:, 0], gmm.means[:, 1], marker="o", s=50, c="red")
```

```
<matplotlib.collections.PathCollection at 0x161285d90>
```



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: 3 pts

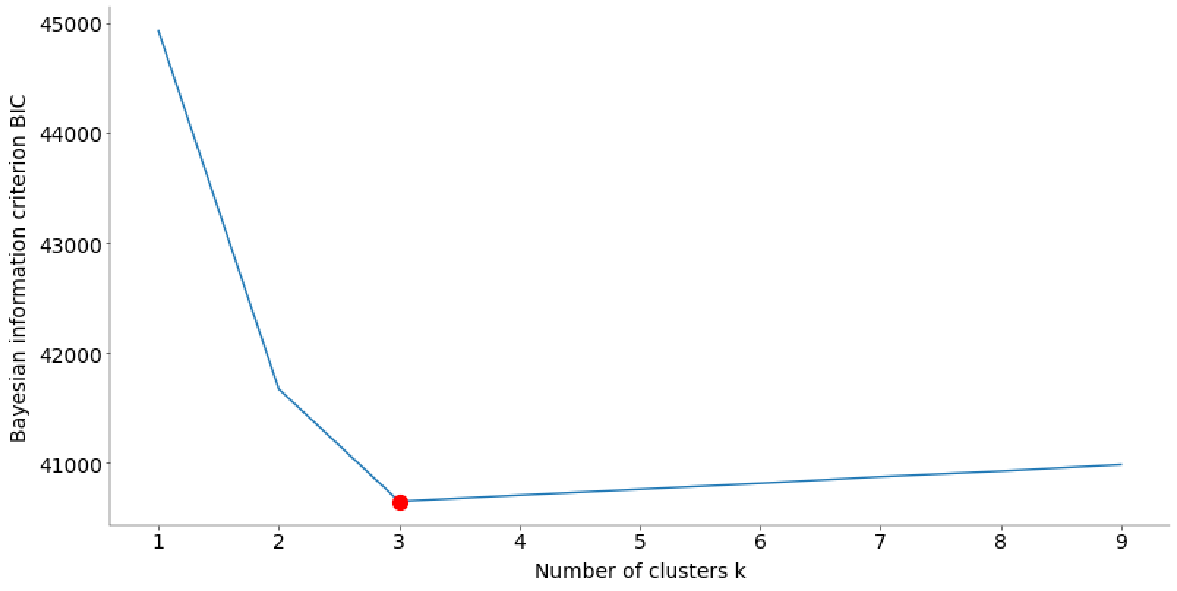
```python
def mog_bic(
    x: np.ndarray, means: np.ndarray, covs: np.ndarray, probs: np.ndarray
) -> tuple[float, float]:
    """Compute the BIC for a fitted Mixture of Gaussian model
    Parameters
    ----------

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

    means: np.array, (n_clusters, n_dims)
        Means

    covs: np.array, (n_clusters, n_dims, n_dims)
        Covariances

    probs: np.array, (n_clusters, )
        Cluster weights / probablities

    Return
    ------

    bic: float
        BIC

    LL: float
        Log Likelihood
    """

    k = means.shape[0]
    n = x.shape[0]
    n_features = means.shape[1]
    p_per_cluster = n_features + n_features**2 + 1  # mean + cov + weight
    n_params = k * p_per_cluster

    ll = np.sum(
        np.log(
            np.sum(
                [
                    p * sp.stats.multivariate_normal.pdf(x, m, c)
                    for p, m, c in zip(probs, means, covs)
                ],
                axis=0,
            )
        )
    )
    # bic = np.log(x.shape[0]) * (means.shape[0] * (means.shape[1] + 0.5 * means.shape[1]
    bic = -2 * ll + n_params * np.log(n)

    return bic, ll
```

```
K = range(1, 10)
num_seeds = 5
seeds = np.random.randint(0, 1000, size=num_seeds)

BIC = np.zeros((num_seeds, len(K)))
LL = np.zeros((num_seeds, len(K)))

for i, k in enumerate(K):
    for j, seed in enumerate(seeds):
        np.random.seed(seed)
        gmm = GMM2D(k, niters=30)
        gmm.fit(data_values)
        BIC[j, i], LL[j, i] = mog_bic(data_values, gmm.means, gmm.cov, gmm.prior_probs)

    print(
        f"Computed BIC for {k} clusters: {BIC[:, i].mean():.2f} +/- {BIC[:, i].std():.2f}"
    )

best_k = np.argmin(BIC.mean(axis=0)) + K[0]
best_bic = BIC[:, best_k - K[0]].mean()
print(f"--> Best number of clusters: {best_k} with a BIC of {best_bic:.2f}")
```

```
Computed BIC for 1 clusters: 44931.66 +/- 0.00
Computed BIC for 2 clusters: 41674.19 +/- 0.00
Computed BIC for 3 clusters: 40652.37 +/- 0.00
Computed BIC for 4 clusters: 40708.64 +/- 0.23
Computed BIC for 5 clusters: 40762.05 +/- 3.93
Computed BIC for 6 clusters: 40816.34 +/- 3.12
Computed BIC for 7 clusters: 40876.54 +/- 2.01
Computed BIC for 8 clusters: 40927.08 +/- 2.99
Computed BIC for 9 clusters: 40986.02 +/- 2.33
--> Best number of clusters: 3 with a BIC of 40652.37
```

```
fig, ax = plt.subplots(figsize=(8, 4))

ax.plot(K, BIC.mean(axis=0), label="BIC")
ax.scatter(best_k, best_bic, marker="o", s=50, c="red", zorder=10)
ax.set_ylabel("Bayesian information criterion BIC")
ax.set_xlabel("Number of clusters k")
```

```
Text(0.5, 0, 'Number of clusters k')
```



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

```
print(np.shape(pc1s))
```

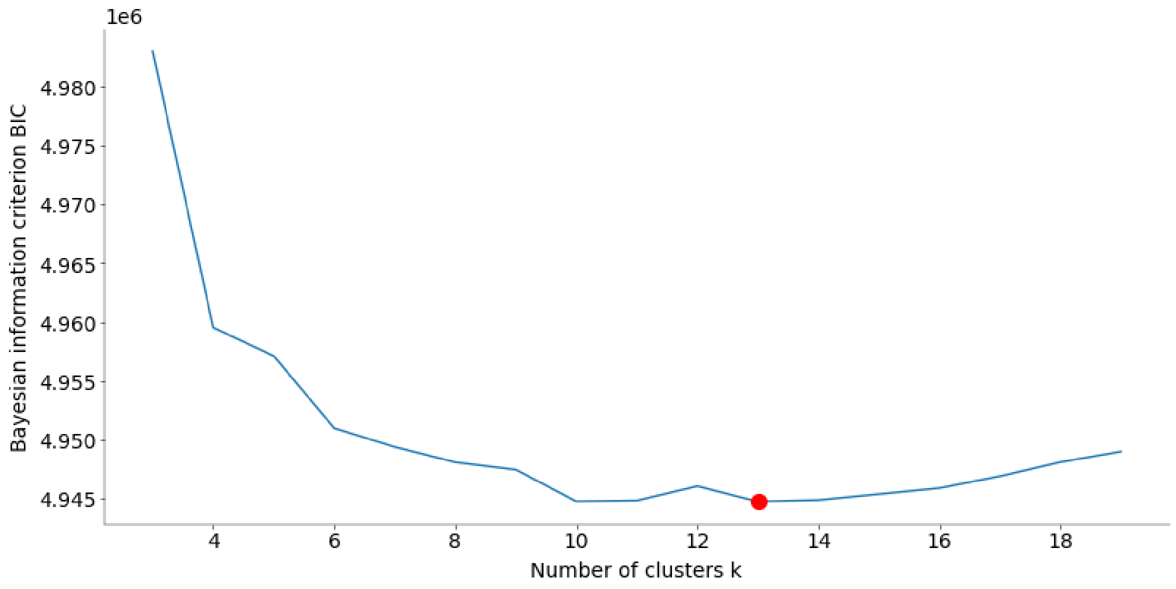
```
(35694, 12)
```

```
np.random.seed(42)
K = np.arange(3, 20)
num_seeds = 5
seeds = np.random.randint(0, 1000, size=num_seeds)

BIC = np.zeros((num_seeds, len(K)))
LL = np.zeros((num_seeds, len(K)))

for i, k in enumerate(K):
    for j, seed in enumerate(seeds):
        np.random.seed(seed)
        gmm = GMM2D(k, niters=30)
        gmm.fit(pc1s)
        BIC[j, i], LL[j, i] = mog_bic(pc1s, gmm.means, gmm.cov, gmm.prior_probs)
    print(
        f"Computed BIC for {k} clusters: {BIC[:, i].mean():.2f} +/- {BIC[:, i].std():.2f}"
    )

best_k = np.argmin(BIC.mean(axis=0)) + K[0]
best_bic = BIC[:, best_k - K[0]].mean()
best_seed = seeds[np.argmin(BIC[:, best_k - K[0]])]
print(f"--> Best number of clusters: {best_k} with a BIC of {best_bic:.2f}")
```

```
Computed BIC for 3 clusters: 4983006.11 +/- 64.00
Computed BIC for 4 clusters: 4959520.85 +/- 1570.46
Computed BIC for 5 clusters: 4957111.61 +/- 1823.11
Computed BIC for 6 clusters: 4950961.81 +/- 482.00
Computed BIC for 7 clusters: 4949374.52 +/- 335.34
Computed BIC for 8 clusters: 4948074.29 +/- 847.55
Computed BIC for 9 clusters: 4947444.13 +/- 51.94
Computed BIC for 10 clusters: 4944747.41 +/- 254.02
Computed BIC for 11 clusters: 4944819.44 +/- 844.20
Computed BIC for 12 clusters: 4946044.97 +/- 842.45
Computed BIC for 13 clusters: 4944729.70 +/- 1002.75
Computed BIC for 14 clusters: 4944862.03 +/- 871.94
Computed BIC for 15 clusters: 4945368.71 +/- 802.04
Computed BIC for 16 clusters: 4945880.02 +/- 707.68
Computed BIC for 17 clusters: 4946882.48 +/- 417.45
Computed BIC for 18 clusters: 4948087.37 +/- 385.40
Computed BIC for 19 clusters: 4948972.98 +/- 741.29
--> Best number of clusters: 13 with a BIC of 4944729.70
```

```
fig, ax = plt.subplots(figsize=(8, 4))

ax.plot(K, BIC.mean(axis=0), label="BIC")
ax.scatter(best_k, best_bic, marker="o", s=50, c="red", zorder=10)
ax.set_ylabel("Bayesian information criterion BIC")
ax.set_xlabel("Number of clusters k")

# plot BIC
```

```
Text(0.5, 0, 'Number of clusters k')
```

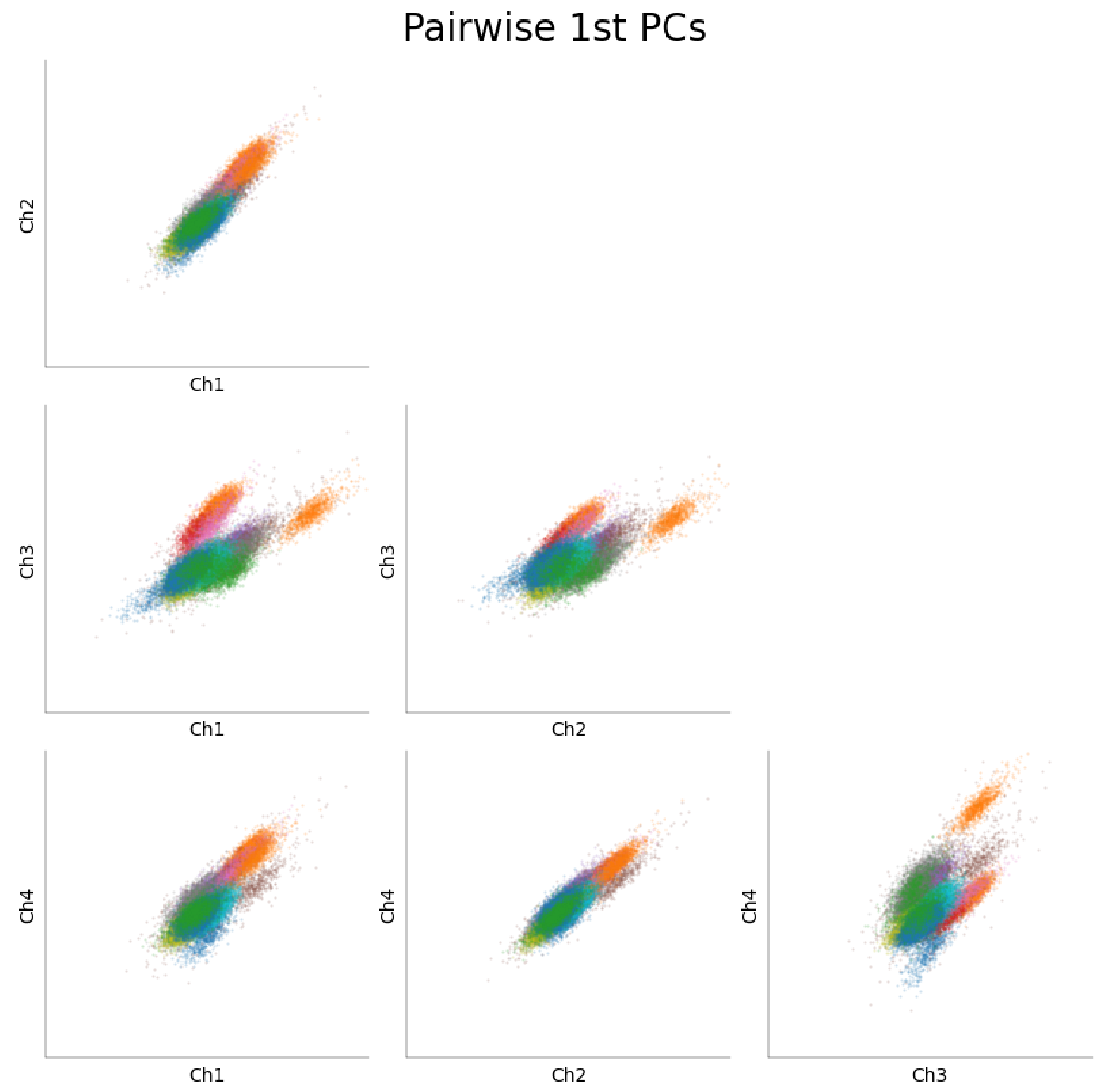


Refit model with lowest BIC and plot data points

```
np.random.seed(best_seed)
gmm = GMM2D(best_k, niters=30)
gmm.fit(pc1s)
labels = gmm.predict(pc1s)
means, covs, probs = gmm.means, gmm.cov, gmm.prior_probs
```

```
mosaic = [
    ["Ch2 vs Ch1", ".", "."],
    ["Ch3 vs Ch1", "Ch3 vs Ch2", "."],
    ["Ch4 vs Ch1", "Ch4 vs Ch2", "Ch4 vs Ch3"],
]
fig, ax = plt.subplot_mosaic(
    mosaic=mosaic, figsize=(8, 8), layout="constrained", dpi=100
)

i = {"Ch1": 0, "Ch2": 3, "Ch3": 6, "Ch4": 9}

for m in np.ravel(mosaic):
    if m == ".":
        continue
    # get the indices of the channel for the first channel vs second channel
    firstch = i[m[:3]]
    secondch = i[m[-3:]]
    for k in range(best_k):
        # plot the ellipses
        # plot_cov_ellipse(S[k, firstch, secondch], m[:3], m[-3:], ax=ax[m], nstd=2, alpha
        # plot the data points
        ax[m].scatter(
            pc1s[labels == k, firstch], pc1s[labels == k, secondch], s=1, alpha=0.2
        )

    y, x = m.split(" vs ")

    ax[m].set_xlabel(x)
    ax[m].set_ylabel(y)
    ax[m].set_xlim((-1500, 1500))
    ax[m].set_ylim((-1500, 1500))
    ax[m].set_xticks([])
    ax[m].set_yticks([])

fig.suptitle("Pairwise 1st PCs", fontsize=20)
```

```
Text(0.5, 0.98, 'Pairwise 1st PCs')
```



Task 5: Cluster separation

Implement linear discriminant analysis to visualize how well each cluster is separated from its neighbors in the high-dimensional space in the function `separation()`. Project the spikes of each pair of clusters onto the axis that optimally separates those two clusters.

Plot a matrix with pairwise separation plots, showing the histogram of the points in both clusters projected on the axis best separating the clusters (as shown in the lecture). *Hint:*

Since Python 3.5+, matrix multiplications can be compactely written as `x@y`.

Grading: 4 pts

```
def separation(
    features: np.ndarray,
    means: np.ndarray,
    covs: np.ndarray,
    probs: np.ndarray,
    labels: np.ndarray,
    nbins: int = 50,
):
    """Calculate cluster separation by LDA.

    proj, bins = separation(b, m, S, p, assignment)
    projects the data on the LDA axis for all pairs of clusters. The result
    is normalized such that the left (i.e. first) cluster has
    zero mean and unit variances. The LDA axis is estimated from the model.
    ---

    Parameters
    ----------
    features: np.array, (n_spikes, n_features)
        Features.

    means: np.array, (n_clusters, n_features)
        Means.

    covs: np.array, (n_clusters, n_features, n_features)
        Covariance.

    probs: np.array, (n_clusters, )
        Cluster weight.

    labels: np.array, (n_spikes, )
        Cluster assignments / labels for each spike

    nbins: int
        Number of bins in a lda histogram.


    Returns
```

```
-------

proj: np.array, (n_bins, n_clusters, n_clusters)
    computed lda histogram Comparing the cells in particular

proj_labels: np.array, (n_clusters, n_clusters)

bins: np.array, (n_bins)
    bin times relative to center
"""

# Initialize array to store the histograms
histograms = np.zeros((nbins, len(means), len(means), 2))
overall_mean = np.average(means, weights=probs, axis=0)
# Iterate over all possible pairs of clusters
for i in range(len(means)):
    for j in range(len(means)):
        # Number of pca features = n_pcs * n_channels
        n_features = features.shape[1]

        # get the features of the current cluster
        bin_features = features[(labels == i) | (labels == j)]
        bin_feature_labels = labels[(labels == i) | (labels == j)]

        # make label array for the current pair of clusters
        # (i.e. 0 for cluster i and 1 for cluster j)
        bin_labels = np.asarray([i, j])

        # go through each of the two clusters and
        # compute the within-cluster scatter matrix
        # and the between-cluster scatter matrix

        SW = np.zeros((features.shape[1], features.shape[1]))
        SB = np.zeros((features.shape[1], features.shape[1]))

        # get values to scale sclusters based on
        # first cluster
        for bl in bin_labels:
            # get the features for the current cluster
            m = means[bl]  # - scale_mean
            cov = covs[bl]  # / scale_cov
            # add cov to the within-cluster scatter matrix
            SW += cov

            # compute overall scatter matrix
            mean_diff = m - overall_mean.reshape(n_features, 1)
            SB += np.dot(mean_diff, mean_diff.T) * len(bin_labels)

        # solve the generalized eigenvaue problem
        A = np.linalg.inv(SW) @ SB

        # Compute eigenvalues and eigenvectors of A
        eigvals, eigvecs = np.linalg.eig(A)

        # Transpose the eigenvectors
        eigvecs = eigvecs.T

        # Sort eigenvalues and eigenvectors fron low to high
        idx = np.argsort(abs(eigvals))[::-1]
        eigvals = eigvals[idx]
        eigvecs = eigvecs[idx]

        # store the first eigenvector as our LDA axis
        lda_axis = eigvecs[0]

        # project the data onto the LDA axis
        proj = np.dot(bin_features - overall_mean, lda_axis)

        # normalize projected data by mean and std of the first cluster
        proj = (proj - np.mean(proj[bin_feature_labels == i])) / np.std(
            proj[bin_feature_labels == i]
        )

        # split the projection into the two clusters
        proj_i = proj[bin_feature_labels == i]
        proj_j = proj[bin_feature_labels == j]

        # make a array for the histogram bins because they
        # have to be the same for both clusters
        bins = np.linspace(-4, 4, nbins + 1)

        # compute the histogram of the projected data
```

```
            hist_i, _ = np.histogram(proj_i, bins=bins)
            hist_j, _ = np.histogram(proj_j, bins=bins)

            # put the histograms into the array
            histograms[:, i, j, 0] = hist_i
            histograms[:, i, j, 1] = hist_j

    return histograms, bins
```

```
hist, bins = separation(pc1s, means, covs, probs, labels, nbins=50)
```

```
# plot the pairwise histograms for each cluster
plot_shape = (len(means), len(means))
fig, axs = plt.subplots(*plot_shape, figsize=(10, 10), sharex=True, sharey=True)
# make array with colors for each cluster
colors = [
    "blue",
    "red",
    "green",
    "orange",
    "purple",
    "brown",
    "pink",
    "gray",
    "olive",
    "cyan",
    "magenta",
    "yellow",
    "lightblue",
]
# plot the histograms for all possible pairs of clusters
for i in range(len(means)):
    for j in range(len(means)):
        if j == i:
            continue
        axs[j, i].bar(bins[:-1], hist[:, i, j, 0], facecolor=colors[i], alpha=0.5)
        axs[j, i].bar(bins[:-1], hist[:, i, j, 1], facecolor=colors[j], alpha=0.5)
        axs[i, j].set_xticks([])
        axs[i, j].set_yticks([])
```