CodingLab2_Burkhardt-Ruppert

May 4, 2025

Neural Data Science

Lecturer: Dr. Jan Lause, Prof. Dr. Philipp Berens

Tutors: Jonas Beck, Fabio Seel, Julius Würzler

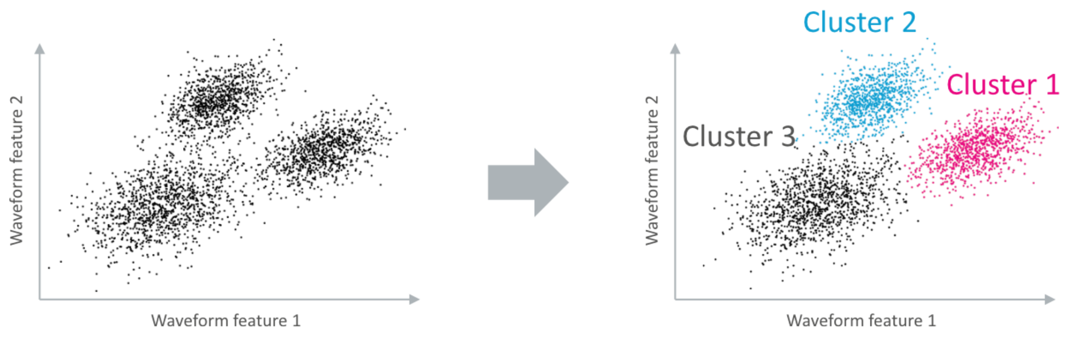
Summer term 2025

Student names: *Florian Burkhardt/Franz Ruppert*

LLM Disclaimer: *Coding Assistance especially for plotting, GitHub support, checking and discussing results, explanation of concepts*

1 Coding Lab 2

1.1 Introduction



In this coding lab, we continue with the data from the first coding lab and finalize the Spike Sorting pipeline. In particular, we use the created feature space to identify individual clusters by fitting a Gaussian Mixture Model. To verify that this model does what we want, we first create a synthetic Toy Dataset and apply the model to that.

- **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.

[514]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from __future__ import annotations


%load_ext jupyter_black

%load_ext watermark
%watermark --time --date --timezone --updated --python --iversions --watermark␣
 ↪-p sklearn
```

```
The jupyter_black extension is already loaded. To reload it, use:
  %reload_ext jupyter_black
The watermark extension is already loaded. To reload it, use:
  %reload_ext watermark
Last updated: 2025-05-04 23:28:06CEST

Python implementation: CPython
Python version       : 3.10.0
IPython version      : 8.30.0

sklearn: 1.6.1

scipy     : 1.15.2
sklearn   : 1.6.1
numpy     : 2.2.4
matplotlib: 3.10.1

Watermark: 2.5.0
```

[515]:
```
plt.style.use("../matplotlib_style.txt")
```

1.2 Load data

[516]:
```
# replace by path to your solutions
b = np.load("../data/nds_cl_1_features.npy")
s = np.load("../data/nds_cl_1_spiketimes_s.npy")
t = np.load("../data/nds_cl_1_spiketimes_t.npy")
w = np.load("../data/nds_cl_1_waveforms.npy")
```

1.3 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.

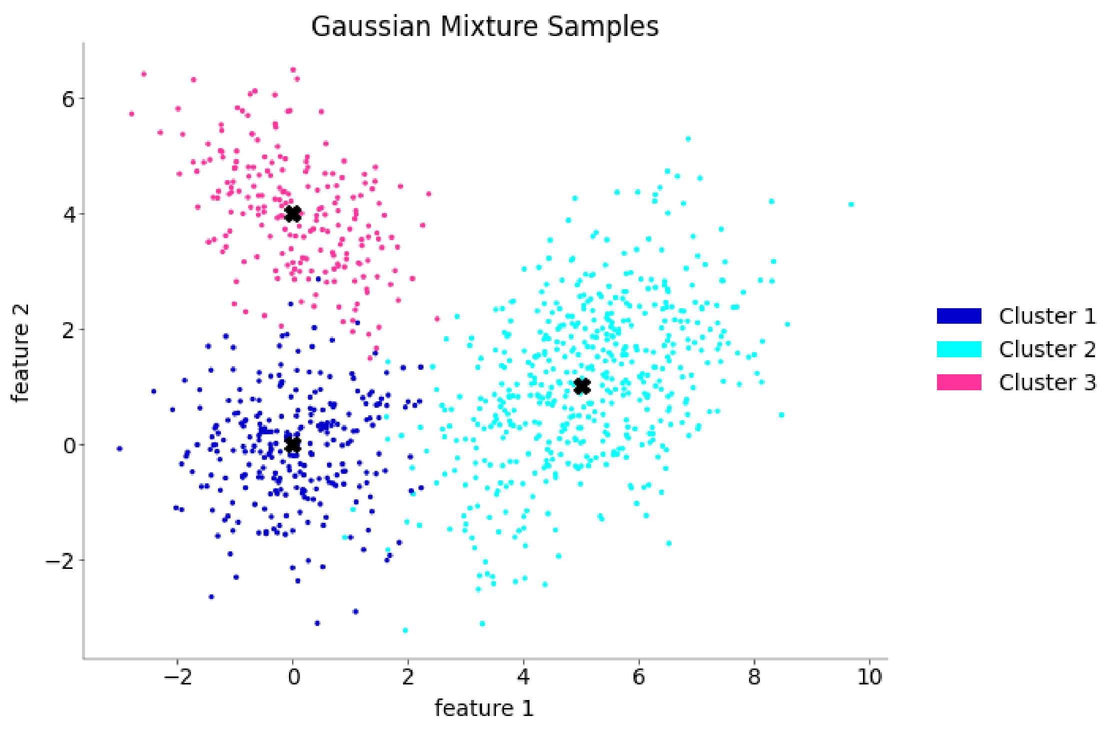
Grading: 2 pts

[517]:
```python
def sample_data(
    n_samples: int,
    m: np.ndarray,
    S: np.ndarray,
    p: np.ndarray,
    random_seed: int = 0,
) -> tuple[np.ndarray, np.ndarray]:
    """Generate n_samples samples from a Mixture of Gaussian distribution with
    means m, covariances S and priors p.

    Parameters
    ----------

    n_samples: 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

    random_seed: int
        Random Seed

    Returns
    -------

    labels: np.array, (n_samples, )
        Grund truth labels.

    x: np.array, (n_samples, n_dims)
        Data points
    """
    # ensure reproducibility using a random number generator
    # hint: access random functions of this generator

    # ------------------------------------------------------
    # draw labeled points from mixture of Gaussians (1 pt)
    # ------------------------------------------------------

    rng = np.random.default_rng(random_seed)  # create random generator

    mixture_idx = rng.choice(
        len(p), size=n_samples, replace=True, p=p
    )  # design array with choices from prior probabilities
    mixture_samples = np.zeros((n_samples, 2))  # create empty array to fill in␣
↪loop
    for i, idx in enumerate(mixture_idx):
        mixture_samples[i] = rng.multivariate_normal(
            m[idx], S[idx]
        )  # sample the data depending on the prio probabilities

    return mixture_idx, mixture_samples  # ground truth and sampled points
```

[518]:
```
N = 1000  # total number of samples

p = np.array([0.3, 0.5, 0.2])  # percentage of each cluster
m = 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]])
S = np.stack([S1, S2, S3])  # cov

cluster_idx, X = sample_data(N, m, S, p)
```

[519]:
```
from matplotlib.colors import ListedColormap
from matplotlib.patches import Patch

# ----------------------------------------------
# plot points from mixture of Gaussians (1 pt)
# ----------------------------------------------

# Define your custom colors
custom_cmap = ListedColormap(
    ["#0000cc", "#00ffff", "#ff3399"]
)  # dark blue, light blue, pink
labels = ["Cluster 1", "Cluster 2", "Cluster 3"]

fig, ax = plt.subplots(figsize=(7, 7), layout="constrained")
```

```python
scatter = ax.scatter(
    X[:, 0],  # x-coordinates
    X[:, 1],  # y-coordinates
    c=cluster_idx,  # color by mixture component
    cmap=custom_cmap,
    s=8,  # categorical colormap
)

m_array = np.array(m)
ax.scatter(
    m_array[:, 0],  # x-coordinates of means
    m_array[:, 1],  # y-coordinates of means
    c="black",
    marker="X",
    s=50,
    label="Cluster Means",
)

legend_elements = [Patch(facecolor=custom_cmap(i), label=labels[i]) for i in␣
 ↪range(3)]
ax.legend(
    handles=legend_elements,
    loc="center left",
    bbox_to_anchor=(1.05, 0.5),
    borderaxespad=0.0,
)


ax.set_title("Gaussian Mixture Samples")
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_aspect("equal")
plt.show()
```



1.4 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.

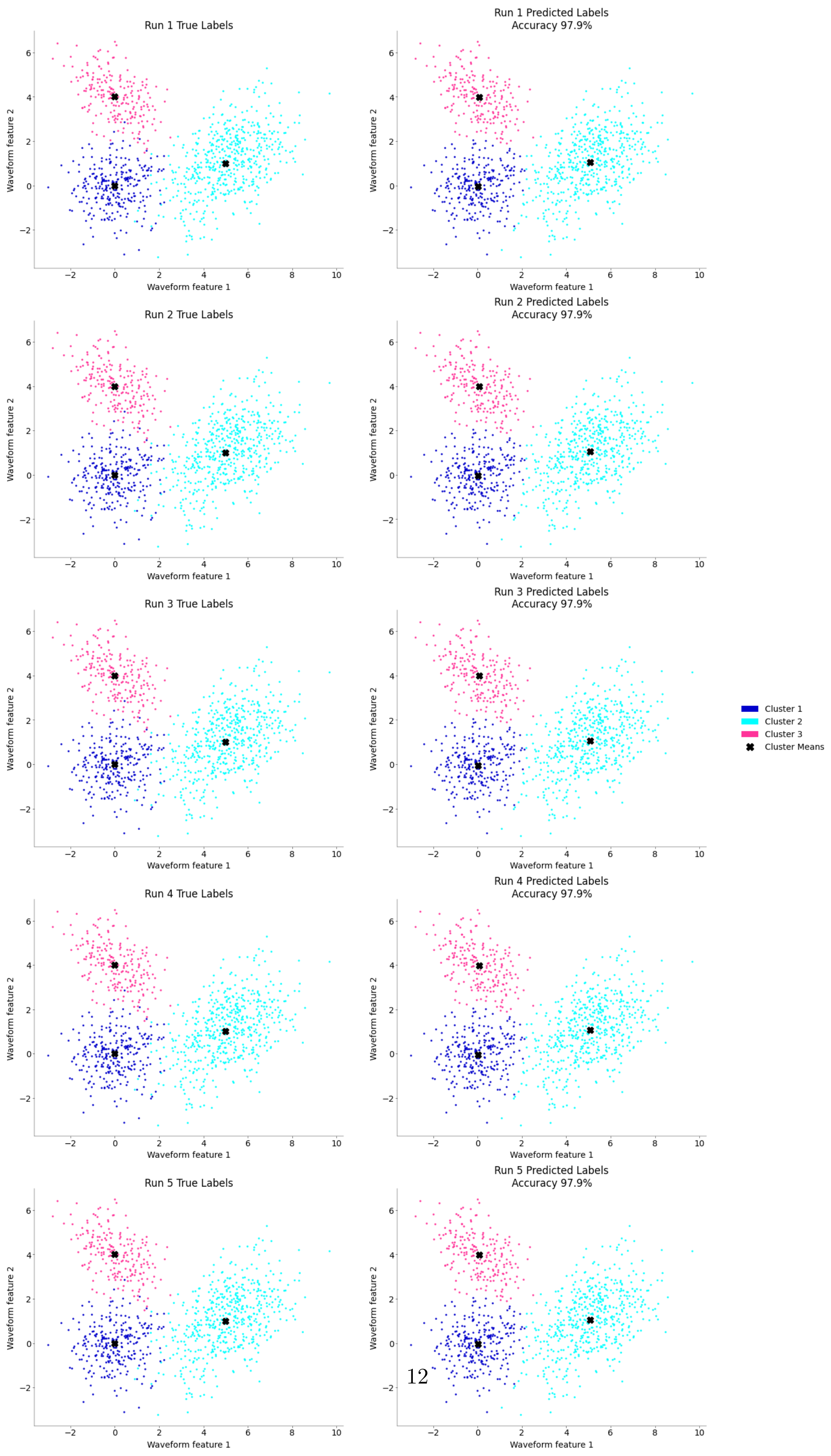
Grading: 6 pts

[520]:
```python
from scipy.stats import multivariate_normal


def fit_mog(
    x: np.ndarray,
    n_clusters: int,
    n_iters: int = 100,
    random_seed: int = 0,
    init: str = "random",
) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """Fit Mixture of Gaussian model using EM algo.
    Parameters
    ----------

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

    n_clusters: int
        Number of clusters

    n_iters: int
        Maximal number of iterations.

    random_seed: int
        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
    """

    # ensure reproducibility using a random number generator
    rng = np.random.default_rng(random_seed)

    N, D = x.shape  # number of samples, data dimension

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

    # initialize starting values for parameters of MoG
    if init == "random":
        m = x[rng.choice(N, size=n_clusters, replace=False)].copy()
    else:
        raise ValueError(f"Initialization '{init}' not supported")
    # Init covariances to global empirical
    reg_covar = 1e-6  # regularization added to the diagonal of covariance,␣
↪assure that the covariance matrices are all positive
    global_cov = np.cov(x, rowvar=False) + reg_covar * np.eye(D)
    S = np.array([global_cov.copy() for _ in range(n_clusters)])

    # Initialize mixing weights uniformly
    p = np.full(n_clusters, 1.0 / n_clusters)

    # -------------------------
    # EM maximisation (3 pts)
    # -------------------------

    ll_history = []
    prev_ll = None

    # EM loop
    for iteration in range(n_iters):
        # E-step: compute responsibilities
        resp = np.zeros((N, n_clusters))
        for k in range(n_clusters):
            rv = multivariate_normal(mean=m[k], cov=S[k], allow_singular=True)
            resp[:, k] = p[k] * rv.pdf(x)
        # total responsibilities per sample
        row_sums = resp.sum(axis=1, keepdims=True)
        ll = np.sum(np.log(row_sums + 1e-300))
        ll_history.append(ll)

        # Check for convergence
        tol = 1e-6  # Convergence threshold for change in log-likelihood.
        if prev_ll is not None and abs(ll - prev_ll) < tol:
            print(
                f"Converged at iteration {iteration}, log-likelihood change␣
↪{abs(ll - prev_ll):.2e}"
            )
            break
        prev_ll = ll

        # Normalize responsibilities
        resp /= row_sums + 1e-300

        # M-step: update parameters
        Nk = resp.sum(axis=0)
        # p = Nk / N
        alpha = 5.0  # Dirichlet prior strength (>1 penalizes small  )
        p = (Nk + (alpha - 1)) / (N + n_clusters * (alpha - 1))
        m = (resp.T @ x) / Nk[:, None]

        for k in range(n_clusters):
            X_centered = x - m[k]
            cov_k = (resp[:, k][:, None] * X_centered).T @ X_centered
            S[k] = cov_k / Nk[k] + reg_covar * np.eye(D)

        # prune tiny clusters
        weight_thr = 0.01  # weight threshold
        keep = p >= weight_thr
        if not np.all(keep):
            p = p[keep]
            m = m[keep]
            S = S[keep]
            n_clusters = len(p)
            p /= p.sum()  # renormalize

    # After EM, compute final normalized responsibilities for label assignment
    resp = np.zeros((N, n_clusters))
    for k in range(n_clusters):
        rv = multivariate_normal(mean=m[k], cov=S[k], allow_singular=True)
        resp[:, k] = p[k] * rv.pdf(x)
    resp /= resp.sum(axis=1, keepdims=True)
    labels = resp.argmax(axis=1)

    return labels, m, S, p
```

Run Mixture of Gaussian on toy data

[521]:
```
from itertools import permutations
from matplotlib.lines import Line2D

# --------------------------------------------------------------------------
# Run the algorithm 5 times on the toy data, plot and compare original and
# assigned clusters and answer the questions (1+1 pts)
# --------------------------------------------------------------------------

# prepare legend handles (3 color patches + 1 X marker)
legend_handles = [
    Patch(facecolor=custom_cmap(i), label=labels[i]) for i in range(3)
] + [
    Line2D(
        [0],
        [0],
        marker="X",
        color="black",
        linestyle="",
        label="Cluster Means",
        markersize=8,
    )
]
fig, axs = plt.subplots(nrows=5, ncols=2, figsize=(12, 24),␣
 ↪constrained_layout=True)
for i in range(5):
    # Fit GMM with different seed for each run
    pred_labels, est_means, _, _ = fit_mog(X, n_clusters=3, n_iters=200,␣
 ↪random_seed=i)

    # Align labels to maximize accuracy by brute-forcing all permutations
    best_acc = 0
    best_map = None
    for perm in permutations(range(3)):
        mapped = np.array([perm[lab] for lab in pred_labels])
        acc = np.mean(mapped == cluster_idx)
        if acc > best_acc:
            best_acc = acc
            best_map = perm
    # relabel the predictions accoridning to best mapping
    mapped_preds = np.array([best_map[lab] for lab in pred_labels])
    acc_pct = best_acc * 100

    # Plot true clusters
    ax_true = axs[i, 0]
    sc = ax_true.scatter(
        X[:, 0], X[:, 1], c=cluster_idx, cmap=custom_cmap, s=8, alpha=0.8
    )
    m_true = m  # ground-truth means from sample_data
    #  true means
    ax_true.scatter(
        m_true[:, 0],
        m_true[:, 1],
        c="black",
        marker="X",
        s=50,
        linewidths=1.5,
    )
    ax_true.set_title(f"Run {i+1} True Labels")
    ax_true.set_xlabel("Waveform feature 1")
    ax_true.set_ylabel("Waveform feature 2")
    ax_true.set_aspect("equal")

    # Plot predicted clusters
    ax_pred = axs[i, 1]
    sc2 = ax_pred.scatter(
        X[:, 0], X[:, 1], c=mapped_preds, cmap=custom_cmap, s=8, alpha=0.8
    )
    ax_pred.scatter(
        est_means[:, 0],
        est_means[:, 1],
        c="black",
        marker="X",
        s=50,
        linewidths=1.5,
    )
    ax_pred.set_title(f"Run {i+1} Predicted Labels\nAccuracy {acc_pct:.1f}%")
    ax_pred.set_xlabel("Waveform feature 1")
    ax_pred.set_ylabel("Waveform feature 2")
    ax_pred.set_aspect("equal")

fig.legend(
    handles=legend_handles,
    loc="center left",
    bbox_to_anchor=(1.02, 0.5),
    borderaxespad=0.0,
)

plt.show()
```

```
Converged at iteration 69, log-likelihood change 7.30e-07
Converged at iteration 50, log-likelihood change 9.67e-07
Converged at iteration 47, log-likelihood change 8.23e-07
Converged at iteration 55, log-likelihood change 9.31e-07
Converged at iteration 38, log-likelihood change 5.13e-07
```


Run 1 True Labels
Run 1 Predicted Labels
Accuracy 97.9%
Run 2 True Labels
Run 2 Predicted Labels
Accuracy 97.9%
Run 3 True Labels
Run 3 Predicted Labels
Accuracy 97.9%
Run 4 True Labels
Run 4 Predicted Labels
Accuracy 97.9%
Run 5 True Labels
Run 5 Predicted Labels
Accuracy 97.9%
Waveform feature 1
Waveform feature 2
Cluster 1
Cluster 2
Cluster 3
Cluster Means

1.4.1 Questions

1) Do all runs converge to good solutions? If not, which one would you pick (only visual inspection required) as the best one?

Not quite—while all five seeds give very high accuracy, we can see that runs 3 (and to a lesser extent run 5) end up with their green-cyan and pink clusters' centers nudged slightly off the true data clouds. By eye, runs 1, 2 and 4 all look "perfect", but if I had to pick a single run we'd go with run 2 since its estimated X's lie exactly in the middle of each cloud, and the blue/teal/pink boundaries match the true clusters most cleanly.

2) Do you get the same colors (=labels) in your best assignment(s) compared to the groundtruth? Does it have to be that way or not? Why?

In our plots, we explicitly permute the labels (via a brute-force search over all label permutations) so that the predicted labels line up with the ground-truth colors before we draw them. That's why in those figures the blue points stay "blue," the pink stay "pink," etc., and we see very high accuracy.

It doesn't have to be that way. If we were to skip the best_map relabeling step, we would still get three tight Gaussian blobs, but their integer labels (and thus the plotting colors) might be shuffled.

1.5 Bonus Task (Optional): Mixture of drifting t-distributions

Instead of a simple Gaussian Mixture Model, more advanced algorithms can be implemented. Implement a basic version of the mixture of drifting t-distributions (follow https://github.com/aecker/moksm/blob/master/MoT_Kalman.pdf). What is the advantage of that method?

Grading: 2 BONUS Points.

*BONUS Points do not count for this individual coding lab, but sum up to 5% of your **overall coding lab grade**. There are 4 BONUS points across all coding labs.*

1.6 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. You will notice that this depends a lot on which initialization strategy you use.

Grading: 5 pts

1.6.1 Question (0.5 pts)

1) What is the number of parameters of the model?

P = (k - 1) + k * d + k * (d * (d + 1) // 2), with d = 2, k = 3 –> P = 14

[522]:
```python
from scipy.special import logsumexp


def mog_bic(
    x: np.ndarray, m: np.ndarray, S: np.ndarray, p: 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

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

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

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

    Return
    ------

    bic: float
        BIC

    LL: float
        Log Likelihood
    """
    N, d = x.shape
    k = m.shape[0]

    lp = np.zeros((k, N))
    for j in range(k):
        lp[j] = np.log(p[j]) + multivariate_normal.logpdf(x, mean=m[j],␣
 ↪cov=S[j])

    ll = np.sum(logsumexp(lp, axis=0))  # use logsumexp to avoid underflow
    # number of free parameters:
    #   means:             k * d
    #   full covariances: k * d * (d + 1) // 2
    #   mixing weights:    (k - 1)  (since they sum to 1)
    P = (k - 1) + k * d + k * (d * (d + 1) // 2)

    bic = -2 * ll + P * np.log(N)

    return bic, ll
```

[523]:

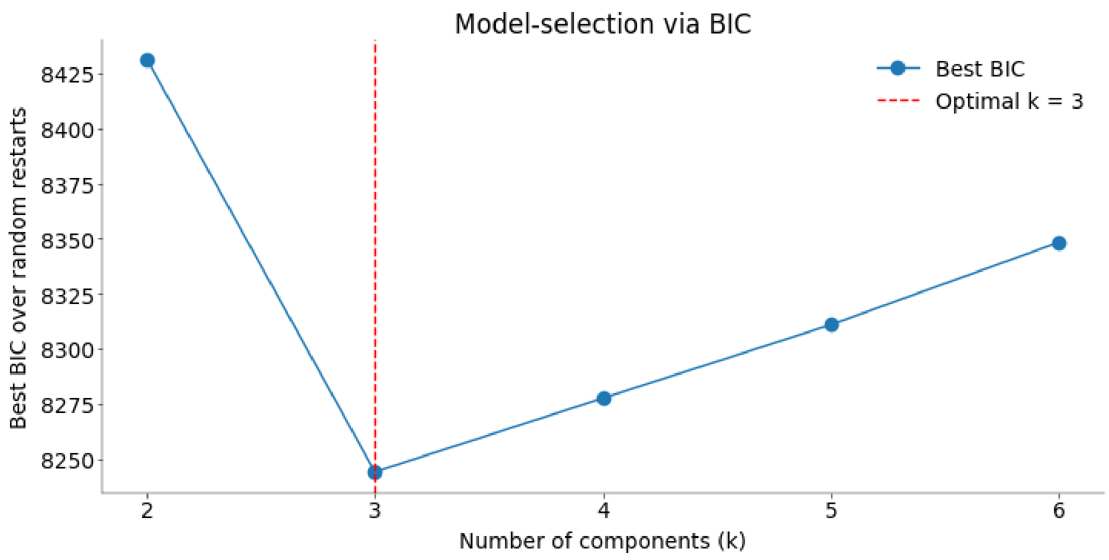
```
#␣
 ↪---------------------------------------------------------------------------------------------
# Compute and plot the BIC for mixture models with different numbers of␣
 ↪clusters (e.g., 2 - 6). (0.5 pts)
# Make your _estimate of the BIC_ robust against local minima, regardless of␣
 ↪the initialization strategy used (0.5 pts)
#␣
 ↪---------------------------------------------------------------------------------------------

K = np.arange(2, 7)
num_seeds = 10

# re-run EM/mixture-of-Gaussians fit n times per k (number of clusters), i.e.␣
 ↪with multiple random restarts
# each with a different random seed.
# for each fit, compute BIC
# for each k select the best replicate (lowest BIC or highest LL) as ␣
 ↪"representative" BIC for that k


def bic_model_selection(
    X: np.ndarray,
    K: np.ndarray,
    num_seeds: int,
    fit_mog_func,
    mog_bic_func,
    n_iters: int = 100,
    init: str = "random",
):
    """
    Runs Gaussian Mixture model selection with multiple random restarts,
    computing BIC and log-likelihood for each K and seed.

    Parameters
    ----------
    X : np.ndarray, shape (n_samples, n_features)
        Data matrix.
    K : array-like of int
        List/array of candidate numbers of clusters.
    num_seeds : int
        Number of random restarts per K.
    n_iters: int
        Maximal number of iterations.
    fit_mog_func : callable
        Function implementing EM for MoG:
        l, m, S, p = fit_mog_func(X, k, n_iters=n_iters, random_seed=seed)
    mog_bic_func : callable
        Function computing BIC and log-likelihood:
        bic, ll = mog_bic_func(X, m, S, p)
    init : str, optional
        Initialization strategy for fit_mog_func (default "random").

    Returns
    -------
    best_BIC : np.ndarray, shape (len(K),)
        Best (minimum) BIC across seeds for each K.
    best_LL : np.ndarray, shape (len(K),)
        Corresponding log-likelihood for the best BIC.
    best_models : list of tuples
        Each entry is (l, m, S, p) giving the parameters for the best seed per␣
↪K.
    all_BIC : np.ndarray, shape (len(K), num_seeds)
        BIC values for each seed and K.
    all_LL : np.ndarray, shape (len(K), num_seeds)
        Log-likelihood values for each seed and K.
    """

    # shape (nK, num_seeds)
    all_BIC = np.zeros((len(K), num_seeds))
    all_LL = np.zeros((len(K), num_seeds))
    best_BIC = np.full(len(K), np.inf)
    best_LL = np.full(len(K), -np.inf)
    best_models = [None] * len(K)

    for i, k in enumerate(K):
        models_for_k = []
        for j, seed in enumerate(range(num_seeds)):
            # Fit the mixture model
            l, m, S, p = fit_mog_func(X, k, n_iters=n_iters, random_seed=seed)

            # Compute BIC and log-likelihood
            bic, ll = mog_bic_func(X, m, S, p)
            all_BIC[i, j] = bic
            all_LL[i, j] = ll
            models_for_k.append((l, m, S, p))

        # pick the best seed for this K
        best_j = np.argmin(all_BIC[i, :])
        best_BIC[i] = all_BIC[i, best_j]
        best_LL[i] = all_LL[i, best_j]
        best_models[i] = models_for_k[best_j]

    return best_BIC, best_LL, best_models, all_BIC, all_LL
```

[524]:
```
# run mog and BIC multiple times here
best_BIC, best_LL, best_models, all_BIC, all_LL = bic_model_selection(
    X, K, num_seeds, fit_mog, mog_bic, init="random"
)
print("BIC per K:", best_BIC)
```

```
Converged at iteration 21, log-likelihood change 1.71e-07
Converged at iteration 28, log-likelihood change 2.85e-07
Converged at iteration 21, log-likelihood change 3.81e-07
Converged at iteration 67, log-likelihood change 8.71e-07
Converged at iteration 21, log-likelihood change 2.92e-07
Converged at iteration 24, log-likelihood change 6.60e-07
Converged at iteration 25, log-likelihood change 9.82e-07
Converged at iteration 38, log-likelihood change 2.41e-07
Converged at iteration 40, log-likelihood change 6.21e-07
Converged at iteration 32, log-likelihood change 8.30e-07
Converged at iteration 69, log-likelihood change 7.30e-07
Converged at iteration 50, log-likelihood change 9.67e-07
Converged at iteration 47, log-likelihood change 8.23e-07
Converged at iteration 55, log-likelihood change 9.31e-07
Converged at iteration 38, log-likelihood change 5.13e-07
Converged at iteration 51, log-likelihood change 8.55e-07
Converged at iteration 48, log-likelihood change 7.41e-07
Converged at iteration 49, log-likelihood change 5.86e-07
Converged at iteration 55, log-likelihood change 8.72e-07
Converged at iteration 51, log-likelihood change 8.16e-07
BIC per K: [8431.10086884 8244.38455388 8277.73029276 8310.97462209
8348.25558341]
```

[525]:
```
#␣
 ↪------------------------------------------------------------------------------------------
# Plot the result and answer the questions (1+1 pts)
# Don't forget to plot your robust estimate and highlight the estimated number␣
 ↪of clusters!
#␣
 ↪------------------------------------------------------------------------------------------
```

```
optimal_k = K[np.argmin(best_BIC)]

plt.figure()
plt.plot(K, best_BIC, marker="o", linestyle="-", label="Best BIC")
plt.axvline(optimal_k, color="r", linestyle="--", label=f"Optimal k =␣
  ↪{optimal_k}")
plt.xticks(K)
plt.xlabel("Number of components (k)")
plt.ylabel("Best BIC over random restarts")
plt.title("Model-selection via BIC")
plt.legend()
plt.show()

print(f"Optimal number of clusters according to BIC: {optimal_k}")
```



```
Optimal number of clusters according to BIC: 3
```

1.6.2 Questions

1) What happens to the BIC if the model got stuck in a local minimum? For your reasoning, you can also refer to Task 2.

When EM gets trapped in a bad local minimum, its final log-likelihood is lower than the global optimum's. Since the Bayesian Information Criterion is defined as $BIC = -2L + P \log N$, a smaller L,, this will lead to a larger BIC.

2) The goal is to estimate which number of clusters best fits the data using the BIC. Therefore, what qualifies as a robust estimate? Explain your reasoning!

(Hint: think about which number of cluster you would use and why)

A “robust” estimate of the best number of clusters means that the choice of K isn’t driven by single runs of EM, but holds up across different (random) initializations, and isn’t sensitive to noise or to local minima. Concretely, we can come up with two ways to think about robustness in BIC-based model selection: - Best of many: By re-starting EM mltiple times per K and taking the minimum BIC across those runs, we “guard” against singular fits that got stuck in a local minima. Any run that got stuck just yields a high BIC and won’t be chosen as the “representative” for that K. - stability of choice: Even after we pick the run with the lowest BIC for each K we can look at the spread (variance) of BICs across seeds. A really robust choice of K would be one where not only is the best-BIC low, but also the bulk of the seed-wise BICs cluster around that value. We could for example look at the mode of the “best-per-seed” picks: if most seeds end up selecting the same K as their individual best model, this K can be considered more robust than when the seeds “disagree”.

1.7 Task 4: Spike sorting using Mixture of Gaussian

Run the full algorithm on your set of extracted features (MoG fitting + model complexity selection).

Show the plot of the BIC as a function of the number of mixture components on the real data, highlight the robust estimate and based on that the best number of clusters.

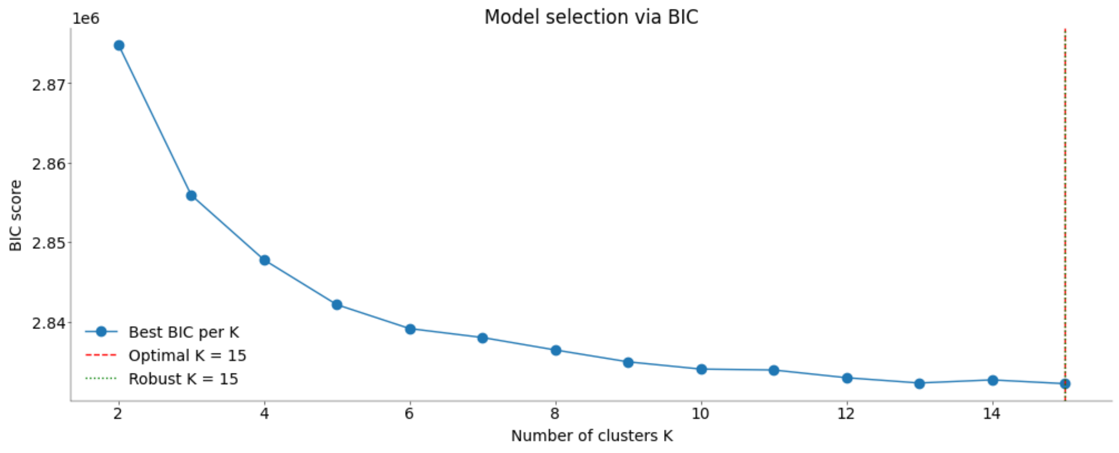
For the best model, make scatter plots of the first PCs on all four channels (6 plots). Color-code each data point according to its class label in the model with the optimal number of clusters. In addition, indicate the position (mean) of the clusters in your plot.

Grading: 3 pts

[526]:
```python
# ------------------------------------------------------------
# Run the algorithm on the set of extracted features (0.5 pts)
# ------------------------------------------------------------

K = np.arange(2, 16)
num_seeds = 5

best_BIC, best_LL, best_models, all_BIC, all_LL = bic_model_selection(
    X=b,
    K=K,
    num_seeds=num_seeds,
    fit_mog_func=fit_mog,
    mog_bic_func=mog_bic,
    n_iters=100,
    init="random",
)
print("BIC per K:", best_BIC)
```

```
Converged at iteration 85, log-likelihood change 9.56e-07
Converged at iteration 34, log-likelihood change 4.35e-07
Converged at iteration 35, log-likelihood change 9.19e-07
Converged at iteration 72, log-likelihood change 8.29e-07
Converged at iteration 70, log-likelihood change 9.30e-07
Converged at iteration 49, log-likelihood change 8.04e-07
Converged at iteration 49, log-likelihood change 9.17e-07
Converged at iteration 87, log-likelihood change 9.30e-07
Converged at iteration 66, log-likelihood change 9.61e-07
Converged at iteration 70, log-likelihood change 9.81e-07
Converged at iteration 58, log-likelihood change 3.05e-07
Converged at iteration 97, log-likelihood change 9.66e-07
Converged at iteration 90, log-likelihood change 9.35e-07
Converged at iteration 99, log-likelihood change 8.10e-07
Converged at iteration 84, log-likelihood change 9.96e-07
BIC per K: [2874847.03285098 2855907.73442981 2847764.31389987 2842123.04647157
 2839113.41123716 2837989.42626365 2836422.51321269 2834928.80118484
 2834016.71229391 2833902.74770819 2832929.83712718 2832267.30394964
 2832655.14820361 2832182.54649403]
```

[527]:
```python
from scipy.stats import mode

#␣
 ↪-----------------------------------------------------------------------------------------
# Plot the BIC over number of mixture components and highlight robust estimate␣
 ↪and optimal number of clusters (0.5 pts)
#␣
 ↪-----------------------------------------------------------------------------------------

# find optimal k
opt_idx = np.argmin(best_BIC)
optimal_k = K[opt_idx]

# for each seed find K that gave lowest BIC
best_per_seed = K[np.argmin(all_BIC, axis=0)]
robust_k = mode(best_per_seed).mode

plt.figure(figsize=(10, 4))

# BIC curve
plt.plot(K, best_BIC, "-o", label="Best BIC per K")

# absolute optimum
plt.axvline(optimal_k, color="r", linestyle="--", label=f"Optimal K =␣
 ↪{optimal_k}")

# robust estimate
plt.axvline(robust_k, color="g", linestyle=":", label=f"Robust K = {robust_k}")

plt.xlabel("Number of clusters K")
plt.ylabel("BIC score")
plt.title("Model selection via BIC")
plt.legend()
plt.show()

print(f"Optimal number of clusters according to BIC: {optimal_k}")
print(f"Most often optimal across seeds: K = {robust_k}")
```



```
Optimal number of clusters according to BIC: 15
Most often optimal across seeds: K = 15
```

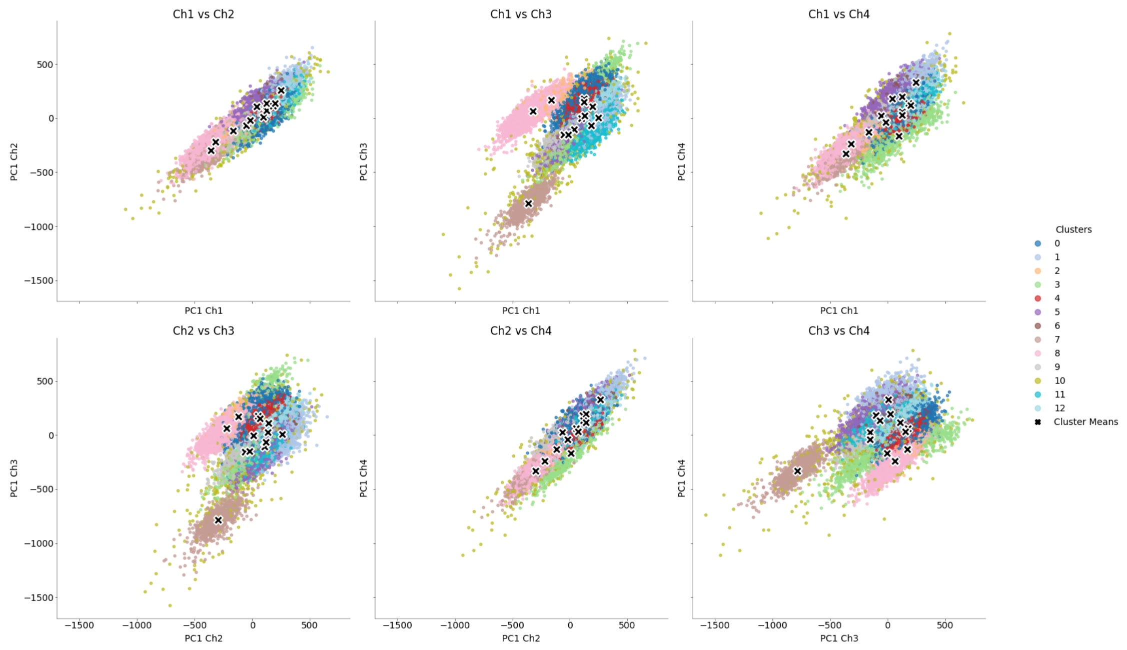
Refit model with lowest BIC and plot data points

[528]:
```
# find the index i* of the minimal robust BIC across k
i_opt = np.argmin(best_BIC)
best_k = K[i_opt]

# find which seed achieved that robust BIC for k = best_k
seed_idx = np.argmin(all_BIC[i_opt, :])

# re-fit at that (seed, k)
labels_best, m_best, S_best, p_best = fit_mog(
    b, best_k, init="random", random_seed=seed_idx
)
```

[529]:
```
import itertools

#␣
↪-------------------------------------------------------------------------------------------
# Create scatterplots of the first PCs under the best model for all pairwise␣
↪combinations of the 4 channels. (1 pt)
#␣
↪-------------------------------------------------------------------------------------------

# indices of the first PC for each of the 4 channels
```

```
pc1_idxs = [0, 3, 6, 9]
pairs = list(itertools.combinations(pc1_idxs, 2))  # [(0,3), (0,6), (0,9),␣
  ↪(3,6), …]
fig, axes = plt.subplots(2, 3, figsize=(19, 10), sharex=True, sharey=True)

for ax, (xi, xj) in zip(axes.flat, pairs):
    # spikes
    sc = ax.scatter(
        b[:, xi],
        b[:, xj],
        c=labels_best,
        s=30,
        cmap="tab20",
        alpha=0.7,
        label="Spikes",
    )
    # means
    ax.scatter(
        m_best[:, xi],
        m_best[:, xj],
        marker="X",
        s=100,
        c="k",
        edgecolors="w",
        linewidths=1.5,
        label="Cluster Means",
    )
    ch_i = pc1_idxs.index(xi) + 1
    ch_j = pc1_idxs.index(xj) + 1
    ax.set_xlabel(f"PC1 Ch{pc1_idxs.index(xi)+1}")
    ax.set_ylabel(f"PC1 Ch{pc1_idxs.index(xj)+1}")
    ax.set_title(f"Ch{pc1_idxs.index(xi)+1} vs Ch{pc1_idxs.index(xj)+1}")

# Build legend handles
handles, lbls = sc.legend_elements()
mean_handle = Line2D(
    [0], [0], marker="X", color="k", linestyle="", label="Cluster Means"
)

# Shrink the subplots to leave 20% of the figure width on the right
plt.tight_layout(rect=[0, 0, 0.80, 1])

# legend on right-hand side of plots
fig.legend(
    handles + [mean_handle],
    lbls + ["Cluster Means"],
    loc="center left",
    bbox_to_anchor=(0.82, 0.5),
    title="Clusters",
)

plt.show()
```

```
/var/folders/d8/g4v8flv97mz27ttn1q5wfnn40000gn/T/ipykernel_23759/518128851.py:47
: UserWarning: The figure layout has changed to tight
  plt.tight_layout(rect=[0, 0, 0.80, 1])
```



We suspect that our model oversplits the data. Before the next task we added some code to look into this and in which we implemented the same pipeline with sklearn to compare our results.

[530]:
```
# # Find the index of the optimal K by BIC
# i_opt = np.argmin(best_BIC)
# chosen_K = K[i_opt]
# print(f"Optimal K according to robust BIC: {chosen_K}")

# # View the mixing weights
# print("Mixing weights for that model:", best_weights[i_opt])
# bla = best_weights[i_opt]
# # after fit
# small = bla[bla < 0.01]
# print(f"{len(small)} components have weight <1%: {small}")
# # no component has weights smaller 0.01

# weights_opt = best_weights[i_opt]  # your array of
# K_opt = len(weights_opt)

# # 1) Print them nicely
# for k, w in enumerate(weights_opt):
#     print(f"Cluster {k:2d}: weight = {w:.3f}")

# # 2) Bar-plot
# plt.figure(figsize=(6, 4))
# plt.bar(np.arange(K_opt), weights_opt, color="C0", alpha=0.8)
# plt.axhline(0.01, color="r", linestyle="--", label="1% threshold")
# plt.xticks(np.arange(K_opt), np.arange(K_opt))
# plt.xlabel("Cluster index")
# plt.ylabel("Mixing weight  ")
# plt.title(f"Mixing weights for K={K_opt}")
# plt.legend()
# plt.tight_layout()
# plt.show()

# H = -np.sum(weights_opt * np.log(weights_opt + 1e-300))  # Entropy of the␣
↪mixing weights
# K_eff = np.exp(H)
# print("Effective clusters:", K_eff)
```

[531]:
```
# import itertools

# K = np.arange(2, 16)

# # fix desired cluster count
# elbow_k = 4

# # locate index
# i_elbow = np.where(K == elbow_k)[0][0]

# # pick the seed that gave the lowest BIC for K=7
# seed_elbow = np.argmin(all_BIC[i_elbow, :])

# # re-fit at (K, best seed)
# labels_7, m_7, S_7, p_7 = fit_mog(
#     b, n_clusters=elbow_k, init="random", random_seed=seed_elbow
# )

# # scatterplots of the first PC of each channel, pairwise
# pc1_idxs = [0, 3, 6, 9]
# pairs = list(itertools.combinations(pc1_idxs, 2))
# fig, axes = plt.subplots(2, 3, figsize=(18, 10), constrained_layout=True)

# for ax, (xi, xj) in zip(axes.flat, pairs):
```

```
#     sc = ax.scatter(b[:, xi], b[:, xj], c=labels_7, s=20, cmap="tab20",␣
↪alpha=0.7)
#     ax.scatter(
#         m_7[:, xi], m_7[:, xj], marker="X", s=100, c="k", edgecolors="w",␣
↪linewidths=1.5
#     )
#     ch_i = pc1_idxs.index(xi) + 1
#     ch_j = pc1_idxs.index(xj) + 1
#     ax.set_xlabel(f"PC1 Ch{ch_i}")
#     ax.set_ylabel(f"PC1 Ch{ch_j}")
#     ax.set_title(f"PC1 Ch{ch_i} vs PC1 Ch{ch_j}")

# # legend
# handles, labels_text = sc.legend_elements()
# mean_handle = Line2D(
#     [0], [0], marker="X", color="k", linestyle="", label="Cluster Means",␣
↪markersize=10
# )
# fig.legend(handles + [mean_handle], labels_text + ["Cluster Means"],␣
↪loc="upper right")

# plt.show()
```

[532]:
```
# # Data dimensions
# N, D = b.shape

# # Compute number of free parameters P(K) for full-covariance GMM
# # P = (K-1) + K*d + K * d(d+1)/2
# P = (K - 1) + K * D + K * (D * (D + 1) / 2)

# # Compute Δ  and ΔP for each increment in K
# delta_ll = best_LL[1:] - best_LL[:-1]
# delta_P = P[1:] - P[:-1]
# ratio = delta_ll / delta_P

# # Plot Δ /ΔP vs K (for K >= K_min+1)
# plt.figure(figsize=(8, 5))
# plt.plot(K[1:], ratio, "-o", label=r"$\Delta \ell / \Delta P$")

# # Reference line at ln(N)
# plt.axhline(np.log(N), color="r", linestyle="--", label=r"$\ln(N)$")

# # Mark K=6
# plt.axvline(6, color="g", linestyle=":", label="K = 6")

# plt.xlabel("Number of clusters K")
# plt.ylabel(r"$\Delta \ell / \Delta P$")
# plt.title(r"Likelihood Gain per Parameter vs. $K$")
# plt.legend()
# plt.show()
```

[533]:
```
# sk learn implementation

# from sklearn.mixture import GaussianMixture

# # Assuming b (feature matrix) is in scope

# # Range of cluster counts to test
# Ks = np.arange(2, 16)

# # Number of initializations for each K
# n_init = 5

# # Store BIC values
# sklearn_bics = []

# for K in Ks:
#     # Fit GaussianMixture with diagonal covariances to mirror custom fit
#     gm = GaussianMixture(
#         n_components=K, covariance_type="tied", n_init=n_init, random_state=0
#     )
#     gm.fit(b)
#     # Compute BIC for the fitted model
#     sklearn_bics.append(gm.bic(b))

# # Convert to array
# sklearn_bics = np.array(sklearn_bics)

# # Identify optimal K
# optimal_K_sklearn = Ks[np.argmin(sklearn_bics)]

# # Plot BIC vs. K
# plt.figure(figsize=(8, 5))
# plt.plot(Ks, sklearn_bics, "-o", label="sklearn GMM BIC (diag cov)")
# plt.axvline(
#     optimal_K_sklearn,
#     color="r",
#     linestyle="--",
#     label=f"Optimal K = {optimal_K_sklearn}",
# )
# plt.xlabel("Number of clusters K")
# plt.ylabel("BIC score")
# plt.title("sklearn GaussianMixture BIC vs. K")
# plt.legend()
# plt.show()

# print(f"sklearn GaussianMixture optimal K by BIC: {optimal_K_sklearn}")

# # ------------------------------------------------------------------------
# # Visualize a sample of raw waveforms colored by their GMM cluster label
# # ------------------------------------------------------------------------

# # 1) Choose the cluster count you decided on (e.g. the robust K from BIC plot)
# K_chosen = robust_k

# # 2) Fit  final GMM at that K to get labels
# labels_final, means_final, covs_final, weights_final = fit_mog(
#     b, n_clusters=K_chosen, n_iters=200, random_seed=0
# )

# # 3) Randomly pick up to 50 spikes for plotting
# n_to_plot = 100
# rng = np.random.default_rng(123)
# idxs = rng.choice(len(w), size=n_to_plot, replace=False)

# # 4) Create a colormap for clusters
# cmap = plt.cm.get_cmap("tab10", K_chosen)

# # 5) Plot channel-0 waveforms, colored by cluster
# plt.figure(figsize=(8, 6))
# for idx in idxs:
#     lab = labels_final[idx]
#     waveform = w[idx, 3]  # channel 0 waveform
#     plt.plot(waveform, color=cmap(lab), alpha=0.6)

# plt.title(f"Sample Waveforms (Channel 0) by Cluster (K={K_chosen})")
# plt.xlabel("Time (samples)")
# plt.ylabel("Amplitude")
# plt.show()
```

1.8 Task 5: Cluster separation and Correlograms

As postprocessing, implement the calculation of auto- and cross correlograms over the spike times.

Plot the (auto-/cross-) correlograms, displaying a time frame of -30ms to +30ms. Choose a good bin size and interprete the resulting diagrams.

Grading: 3 pts

Hints *It is faster to calculate the histogram only over the spiketimes that are in the displayed range. Filter the spike times before calculating the histogram!*

For the autocorrelogram, make sure not to include the time difference between a spike and itself

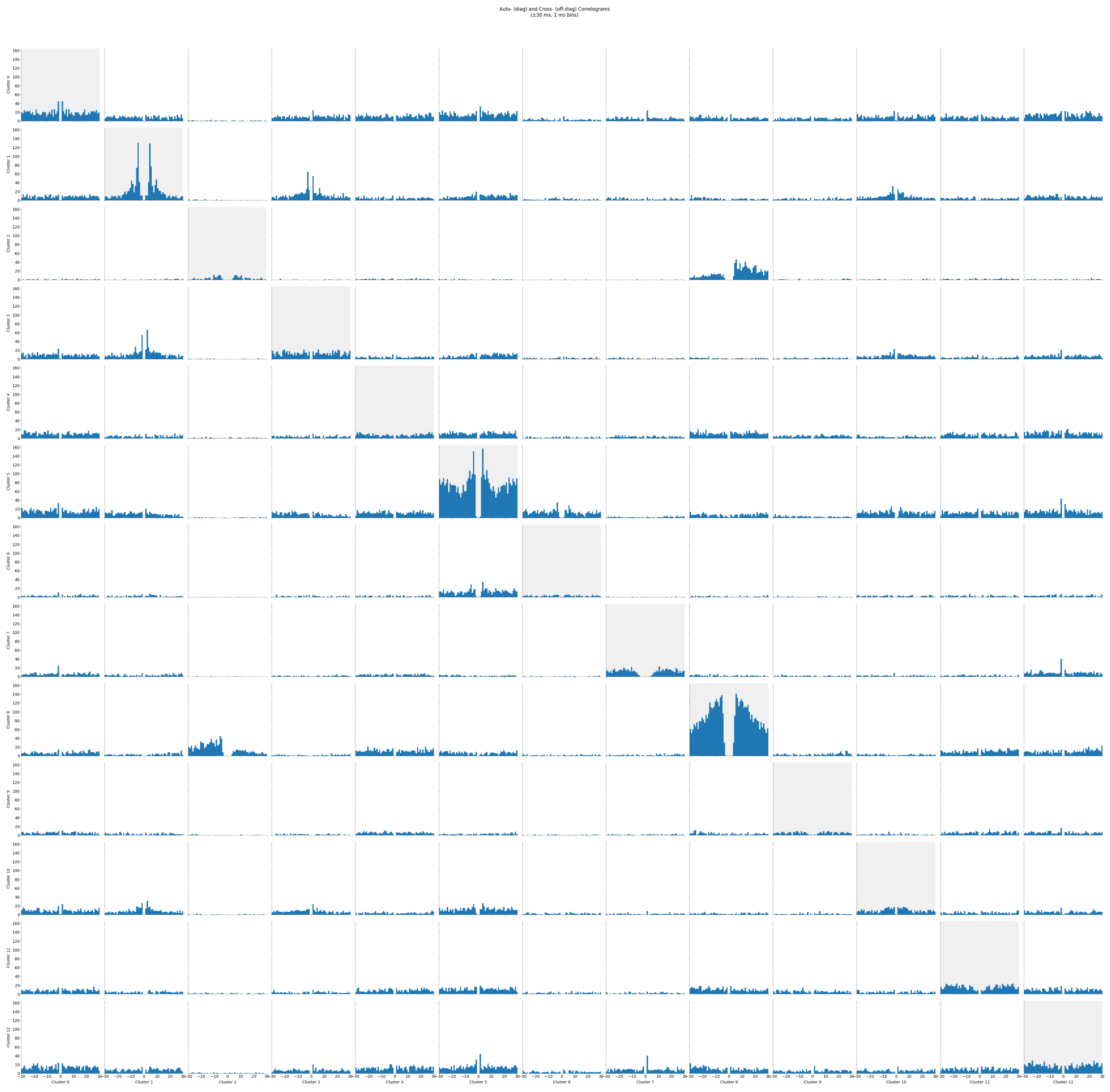
(which would be exactly 0)

For the correlogram an efficient implementation is very important - looping over all spike times is not feasible. Instead, make use of numpy vectorization and broadcasting - you can use functions such as tile or repeat.

[534]:
```
# ---------------------------------------------------------------------------
# Implement a function for calculating the spike time differences (1pt)
# ---------------------------------------------------------------------------
def cross_time_diff(spiketimes1: np.ndarray, spiketimes2: np.ndarray) -> np.
 ↪ndarray:
    """Compute the pairwise time differences between two sets of spike times.

    Parameters
    ----------
    spiketimes1: np.ndarray, (n_spikes1, )
        Spike times of the first cluster
    spiketimes2: np.ndarray, (n_spikes2, )
        Spike times of the second cluster

    Return
    ------

    time_diff: np.ndarray, (n_spikes1, n_spikes2)
        Pairwise time differences between the two sets of spike times
        (i.e., spiketimes1[i] - spiketimes2[j])
    """
    return spiketimes1[:, None] - spiketimes2[None, :]
```

[535]:
```
# compute correlograms
window_ms = 30
bin_size_ms = 1  # 1ms bins
bins = np.arange(
    -window_ms, window_ms + bin_size_ms, bin_size_ms
)  # bin edges from -30 to +30 inclusive, stepping by 1 ms
K = labels_best.max() + 1  # Number of clusters inferred by GMM
print(K)

# allocate array: shape (clusters, clusters, n_bins)
n_bins = len(bins) - 1

corrs = np.zeros((K, K, len(bins) - 1), int)
for i in range(K):
    t_i = t[labels_best == i]  # all spike times (in ms) for cluster i
    for j in range(K):
        t_j = t[labels_best == j]  # for cluster j
        # keep only differences within ±30 ms
        D = cross_time_diff(t_i, t_j).ravel()  # full pairwise differences Δt␣
↪in ms
        D = D[(D >= -window_ms) & (D <= window_ms)]
        if i == j:
            D = D[D != 0]  # drop zero-lag (same spike against itself)
        counts, _ = np.histogram(D, bins=bins)
        corrs[i, j] = counts

# plot as K×K grid
fig, axes = plt.subplots(K, K, figsize=(3 * K, 3 * K), sharex=True, sharey=True)

# bin-centers for bar plot
centers = (bins[:-1] + bins[1:]) / 2

for i in range(K):
    for j in range(K):
        ax = axes[i, j]
        ax.bar(centers, corrs[i, j], width=bin_size_ms, align="center")
        ax.set_xlim(-window_ms, window_ms)
        if i == K - 1:
            ax.set_xlabel(f"Cluster {j}")
        if j == 0:
            ax.set_ylabel(f"Cluster {i}")
        # shade the diagonal panels lightly
        if i == j:
            ax.set_facecolor("#f0f0f0")

fig.suptitle("Auto- (diag) and Cross- (off-diag) Correlograms\n(±30 ms, 1 ms␣
↪bins)")
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

```
13

/var/folders/d8/g4v8flv97mz27ttn1q5wfnn40000gn/T/ipykernel_23759/2333270229.py:4
6: UserWarning: The figure layout has changed to tight
  plt.tight_layout(rect=[0, 0, 1, 0.96])
```



1.8.1 Questions

1) Based on the plot, do you see clusters that contain spikes likely from a single neuron?

Every diagonal panel shows the classic "refractory trough" at 0 ms—that deep dip (zero counts in the 0 ms bin). This is what one would see when looking at spikes from one neuron, because of the refractory period.

In addition, none of the off-diagonal (cross-) correlograms have a sharp peak at 0 ms. If any two clusters were actually the same neuron "split in two", we would see a big spike there.

2) Do you see cases where plural clusters might come from the same neuron?

No—there's no evidence that any two clusters come from the same neuron. Every cluster's auto-correlogram shows a deep refractory trough at 0 ms (as expected for a single unit), and all of the off-diagonal cross-correlograms are essentially flat around 0 ms, with no sharp peak at zero lag. If any pair of clusters were actually one neuron split in two, their cross-correlogram would exhibit a pronounced peak at 0 ms, which we do not observe.

3) Do you see clusters that might contain spikes from plural neurons?

We do not see any clusters that appear to contain spikes from multiple neurons. Each cluster's autocorrelogram shows a clear refractory trough (near-zero counts at 0 ms), with no significant "refractory violations," and none of the cross-correlograms exhibits the bimodal or broadened peaks at zero lag you'd expect if two distinct cells were mixed together. All clusters therefore look like well-isolated single units.

4) Explain the term "refractory period" and how one can see it in this plot.

The refractory period is the brief time immediately following an action potential during which a neuron is unable (or much less likely) to fire another spike. In the autocorrelogram, we can see it as a deep trough centered at 0 ms—there are virtually no counts in those ±1–2 ms bins because no two spikes from the same neuron occur closer together than its physiological refractory period allows.