Neural Data Science

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

Tutors: Jonas Beck, Fabio Seel, Julius Würzler

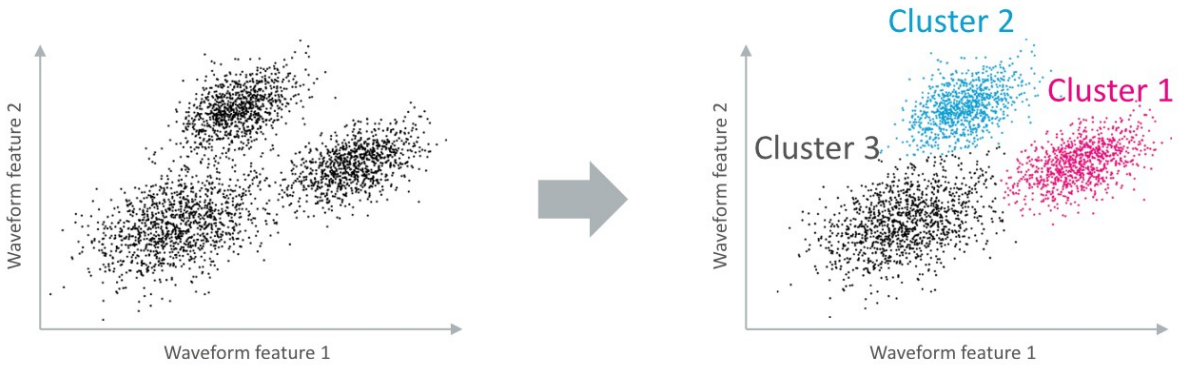
Summer term 2025

Student names: Maria Elisabeth Dirnberger, Akseli Ilmanen, Caroline Grießhaber

LLM Disclaimer: A lot for plots, and especially task 5.

Coding Lab 2

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.

```
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
```

```
Last updated: 2025-05-04 20:14:24W. Europe Summer Time

Python implementation: CPython
Python version       : 3.10.0
IPython version      : 8.35.0

sklearn: 1.6.1

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

Watermark: 2.5.0
```

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

import os

print("Working directory:", os.getcwd())
print("Files in ../data:", os.listdir("../data"))
```

```
Working directory: c:\Users\aksel\OneDrive\03 Tübingen\Semester 4\
Neural Data Science\NDS_homework\notebooks
Files in ../data: ['nds_cl_1.csv', 'nds_cl_1_features.npy',
'nds_cl_1_spiketimes_s.npy', 'nds_cl_1_spiketimes_t.npy',
'nds_cl_1_waveforms.npy']
```

Load data

```
# 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")
```

Task 1: Generate toy data

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

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

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

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

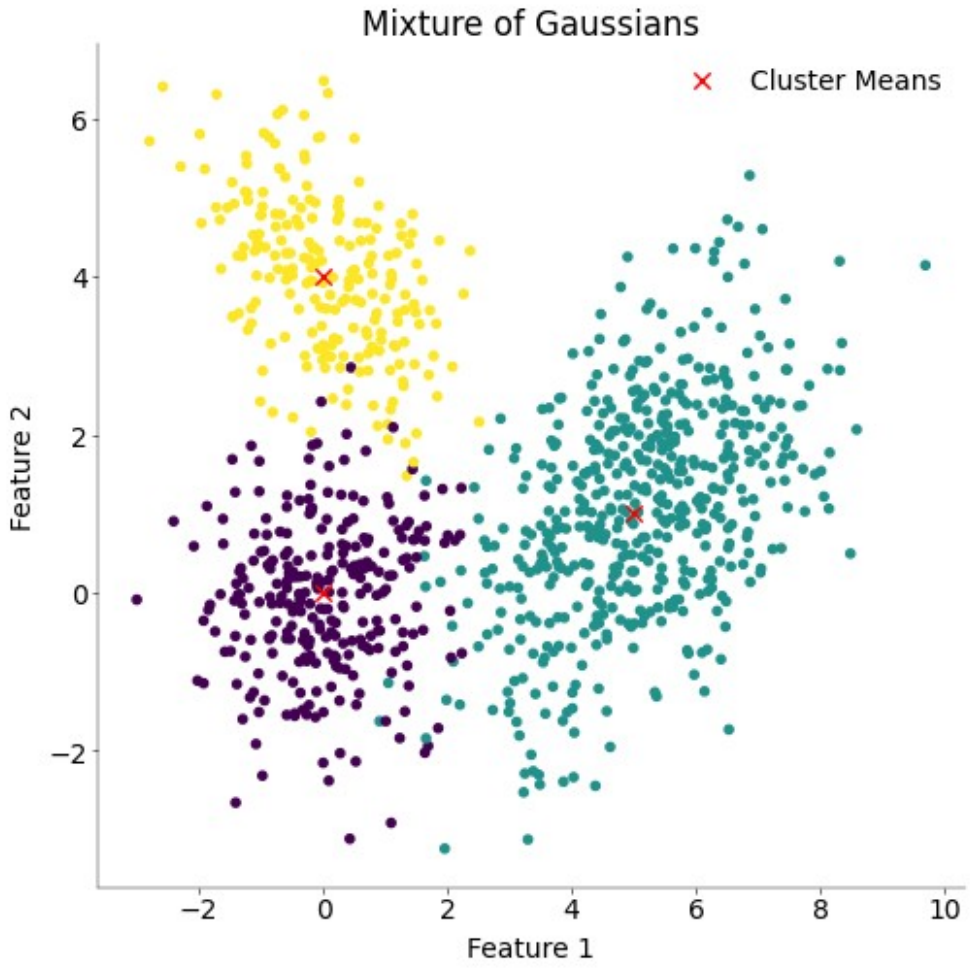
Plot the sampled data points and indicate in color the cluster each point came from. Plot the cluster means as well.

Grading: 2 pts

```
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
    rng = np.random.default_rng(random_seed)

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

```
    # assess number of clusters and dimension
    n_clusters = len(p)
    n_dims = m.shape[1]

    # randomly assign cluster label to each sample based on prior
probabilities
    labels = rng.choice(n_clusters, size=n_samples, p=p)

    # empty list to store data points that are generated in the next
line
    data_points = []

    # for each sample, generate a new data point based on the samples
cluster label
    for label in labels:
        # samples data point from corresponding Gaussian distribution
        data_points.append(rng.multivariate_normal(m[label],
S[label]))

    # convert list into array
    x = np.array(data_points)

    return labels, x

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

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

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

fig, ax = plt.subplots(figsize=(5, 5), layout="constrained")

# scatter plot of points, colored by label
scatter = ax.scatter(x[:, 0], x[:, 1], c=labels)
# plot cluster means
ax.scatter(m[:, 0], m[:, 1], c="red", marker="x", label="Cluster
Means")

# set labels and title
ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
```

```
ax.set_title("Mixture of Gaussians")

ax.legend()

plt.show()
```



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

```python
from scipy.stats import multivariate_normal

def fit_mog(
    x: np.ndarray,
    n_clusters: int,
    n_iters: int = 10,
    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)

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

    n_samples, n_dims = x.shape
```

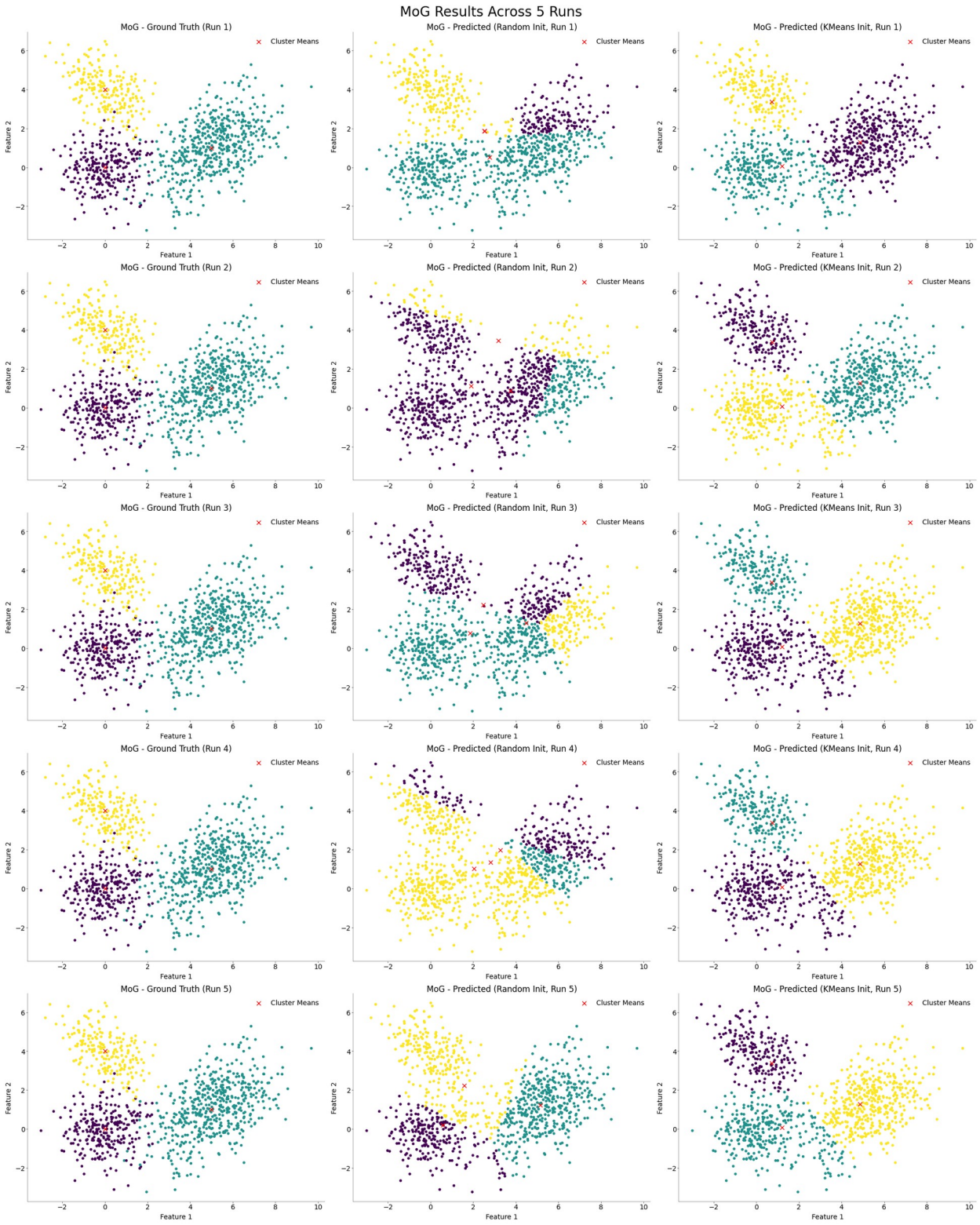
```
    # Initialization of parameters
    S = np.array([np.cov(x.T)] * n_clusters)  # global covariance
    p = np.ones(n_clusters) / n_clusters  # uniform probability
between clusters

    if init == "random":

        # randomly initialize means and covariances
        indeces = rng.choice(n_samples, n_clusters, replace=False)
        m = x[indeces]  # random means from data points

    elif init == "kmeans":

        # initialize using kmaans
        kmeans = KMeans(n_clusters=n_clusters,
random_state=random_seed).fit(x)
        ind = kmeans.labels_
        m = kmeans.cluster_centers_

    else:
        raise ValueError("Init not known")

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

    for step in range(n_iters):

        # E step - probability that each sample belongs to a cluster
        # Evaluate the posterior probablibities `r`
        # using the current values of `m` and `S`
        responsibilities = np.zeros((n_samples, n_clusters))
        for cluster in range(n_clusters):
            responsibilities[:, cluster] = p[cluster] *
multivariate_normal.pdf(
                x, mean=m[cluster], cov=S[cluster]
            )
        # normalize responsibilities to sum to 1 for each sample
        responsibilities /= responsibilities.sum(axis=1)[:,
np.newaxis]

        # M step
        # Estimate new `m`, `S` and `p`
        for cluster in range(n_clusters):
            # update means
            m[cluster] = (responsibilities[:, cluster][:, np.newaxis]
* x).sum(
                axis=0
            ) / responsibilities[:, cluster].sum()
```

```
        for cluster in range(n_clusters):
            # update covariances
            diff = x - m[cluster]
            S[cluster] = (
                (responsibilities[:, cluster][:, np.newaxis] * diff).T
                @ diff
                / responsibilities[:, cluster].sum()
            )

        # Update priors
        p = responsibilities.mean(axis=0)

        # assign labels to each sample based on highest responsibility
        labels = responsibilities.argmax(axis=1)

        return labels, m, S, p
```

Run Mixture of Gaussian on toy data

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

n_clusters = 3

fig, axes = plt.subplots(5, 3, figsize=(20, 25),
constrained_layout=True)


for i, seed in enumerate(range(5)):
    # Ground truth plot
    axes[i, 0].scatter(x[:, 0], x[:, 1], c=labels)
    axes[i, 0].scatter(m[:, 0], m[:, 1], c="red", marker="x",
label="Cluster Means")
    axes[i, 0].set_xlabel("Feature 1")
    axes[i, 0].set_ylabel("Feature 2")
    axes[i, 0].set_title(f"MoG - Ground Truth (Run {seed+1})")
    axes[i, 0].legend()

    # Random initialization
    labels_pred, m_pred, S_pred, p_pred = fit_mog(
        x, n_clusters, n_iters=5, random_seed=seed, init="random"
    )
    axes[i, 1].scatter(x[:, 0], x[:, 1], c=labels_pred)
```

```
    axes[i, 1].scatter(
        m_pred[:, 0], m_pred[:, 1], c="red", marker="x",
label="Cluster Means"
    )
    axes[i, 1].set_xlabel("Feature 1")
    axes[i, 1].set_ylabel("Feature 2")
    axes[i, 1].set_title(f"MoG - Predicted (Random Init, Run
{seed+1})")
    axes[i, 1].legend()

    # KMeans initialization
    labels_pred, m_pred, S_pred, p_pred = fit_mog(
        x, n_clusters, n_iters=5, random_seed=seed, init="kmeans"
    )
    axes[i, 2].scatter(x[:, 0], x[:, 1], c=labels_pred)
    axes[i, 2].scatter(
        m_pred[:, 0], m_pred[:, 1], c="red", marker="x",
label="Cluster Means"
    )
    axes[i, 2].set_xlabel("Feature 1")
    axes[i, 2].set_ylabel("Feature 2")
    axes[i, 2].set_title(f"MoG - Predicted (KMeans Init, Run
{seed+1})")
    axes[i, 2].legend()

plt.suptitle("MoG Results Across 5 Runs", fontsize=20)
plt.show()
```



Questions

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

For random initialization:

- Not all runs converge to good solutions - some solutions are only good with respect to one or two clusters. The solution of the last run is the best.

For kmeans initalization

- All runs converge on the same good solution, as kmeans can easily find a good centers with these well separated, uniformly distributed clusters, and then MoG finds a similar final solution.

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?

We don't always get the same colors, but that is not a problem. The indeces of our clusters are assigned randomly and do not have their own meaning.

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.*

```
# YOUR CODE HERE
```

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

Question (0.5 pts)

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

- $P = 11$
- total mean params = n_clusters * n_dims
- covariance parameters = n_clusters * (n_dims * (n_dims + 1) / 2)
- priors = (n_clusters - 1)

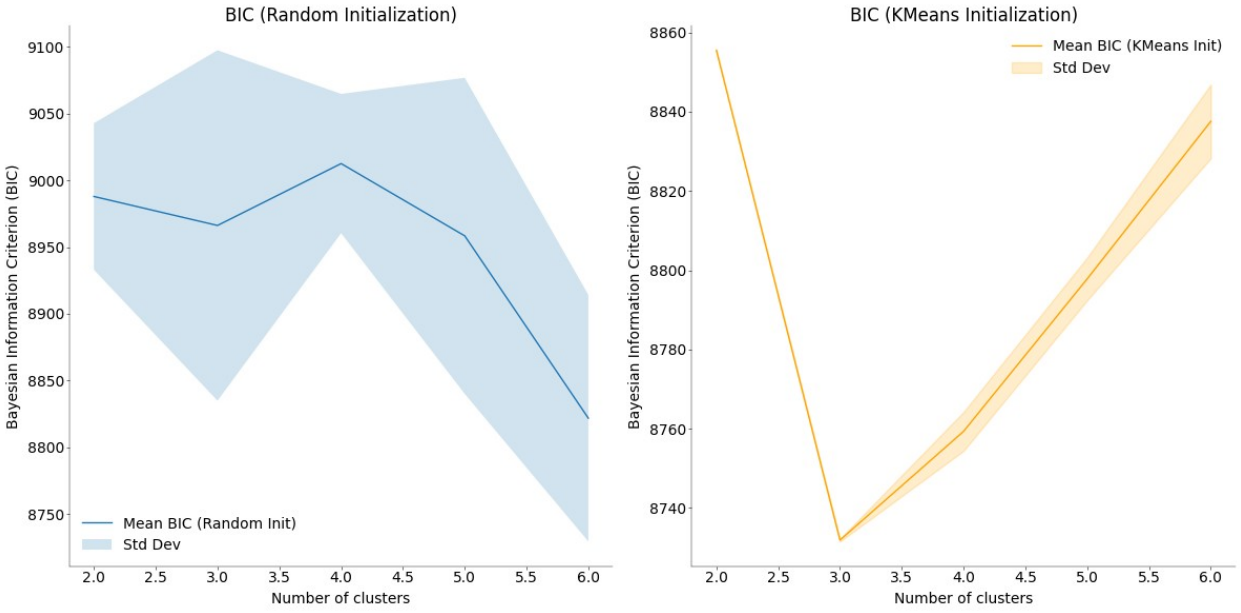
```
def mog_bic(
    x: np.ndarray, m: np.ndarray, S: np.ndarray, p: np.ndarray
) -> tuple[float, float]:
    """Compute the BIC for a fitted Mixture of Gaussian model

    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
    """

    # -------------------------
    # implement the BIC (1.5 pts)
    # -------------------------

    # access n_clusters and n_samples
    n_clusters = p.shape[0]
    n_samples, n_dims = x.shape

    # define the number of params of a GMM for the calculation of the
LL and the BIC
    # total mean params = n_clusters * n_dims
    # covariance parameters = n_clusters * (n_dims * (n_dims + 1) / 2)
    # priors = (n_clusters - 1)
    n_params = (
        (n_dims * n_clusters)
        + (n_clusters * n_dims * (n_dims + 1) / 2)
```

```
        + (n_clusters - 1)
    )

    # Compute the probability densities for all data points and
clusters at once
    pdf_values = np.array(
        [
            sp.stats.multivariate_normal(mean=m[k], cov=S[k]).pdf(x)
            for k in range(n_clusters)
        ]
    )

    # Weight the PDF values by the cluster probabilities (p)
    weighted_pdf_values = pdf_values.T * p  # Transpose to align
dimensions

    # Sum over clusters for each data point
    LL = np.sum(np.log(np.sum(weighted_pdf_values, axis=1)))

    bic = -2 * LL + n_params * np.log(n_samples)

    return bic, LL

#
----------------------------------------------------------------------
------------------------------------------------
# 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

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

# run mog and BIC multiple times here
for idx, k in enumerate(K):
    for seed in range(num_seeds):
        mog_labels, mog_m, mog_S, mog_p = fit_mog(
            x, n_clusters=k, random_seed=seed, n_iters=100,
init="random"
        )
        BIC_rand[seed, idx], LL[seed, idx] = mog_bic(x, mog_m, mog_S,
mog_p)
```

```
BIC_kmeans = np.zeros((num_seeds, len(K)))
LL = np.zeros((num_seeds, len(K)))

# run mog and BIC multiple times here
for idx, k in enumerate(K):
    for seed in range(num_seeds):
        mog_labels, mog_m, mog_S, mog_p = fit_mog(
            x, n_clusters=k, random_seed=seed, n_iters=100,
init="kmeans"
        )

        BIC_kmeans[seed, idx], LL[seed, idx] = mog_bic(x, mog_m,
mog_S, mog_p)

#
--------------------------------------------------------------------
--------------------
# 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!
#
--------------------------------------------------------------------
--------------------

# should this be one line? -> check in with Caros code
# something seems to be off -> check with Caros code

fig, ax = plt.subplots(1, 2, figsize=(12, 6))

# Plot BIC_rand
mean_BIC_rand = BIC_rand.mean(axis=0)
std_BIC_rand = BIC_rand.std(axis=0)
ax[0].plot(K, mean_BIC_rand, label="Mean BIC (Random Init)")
ax[0].fill_between(
    K,
    mean_BIC_rand - std_BIC_rand,
    mean_BIC_rand + std_BIC_rand,
    alpha=0.2,
    label="Std Dev",
)
ax[0].set_title("BIC (Random Initialization)")
ax[0].set_xlabel("Number of clusters")
ax[0].set_ylabel("Bayesian Information Criterion (BIC)")
ax[0].legend()

# Plot BIC_kmeans
mean_BIC_kmeans = BIC_kmeans.mean(axis=0)
std_BIC_kmeans = BIC_kmeans.std(axis=0)
```

```
ax[1].plot(K, mean_BIC_kmeans, label="Mean BIC (KMeans Init)",
color="orange")
ax[1].fill_between(
    K,
    mean_BIC_kmeans - std_BIC_kmeans,
    mean_BIC_kmeans + std_BIC_kmeans,
    alpha=0.2,
    color="orange",
    label="Std Dev",
)
ax[1].set_title("BIC (KMeans Initialization)")
ax[1].set_xlabel("Number of clusters")
ax[1].set_ylabel("Bayesian Information Criterion (BIC)")
ax[1].legend()

plt.tight_layout()
plt.show()
```

```
C:\Users\aksel\AppData\Local\Temp\ipykernel_32100\4183639780.py:44:
UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



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.

If the GMM lands in a local minimum, it fails to capture the true structure of the data. This results in a lower log-likelihood (LL) because the data points are less probable under the model. Since BIC contains -2 * LL, a lower LL means a larger (worse) BIC. Therefore, relying on a single initialization may produce misleading BIC values

In the case of the toy data (task 2), the model with random initialization might have fallen into a local minima, where it made some of the clusters too large/too small, and as a solution increasing the number of clusters actually further decreases BIC, since it can't get even get close to the global optimum of 3.

With kmeans initialization, we find the right solution: 3 clusters.

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 is the number of clusters with the lowest average BIC and low variance across multiple runs. This reduces the impact of local minima and ensures stability regardless of initialization. In this case, 3 clusters is the most robust choice.

Task 4: Spike sorting using Mixture of Gaussian

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

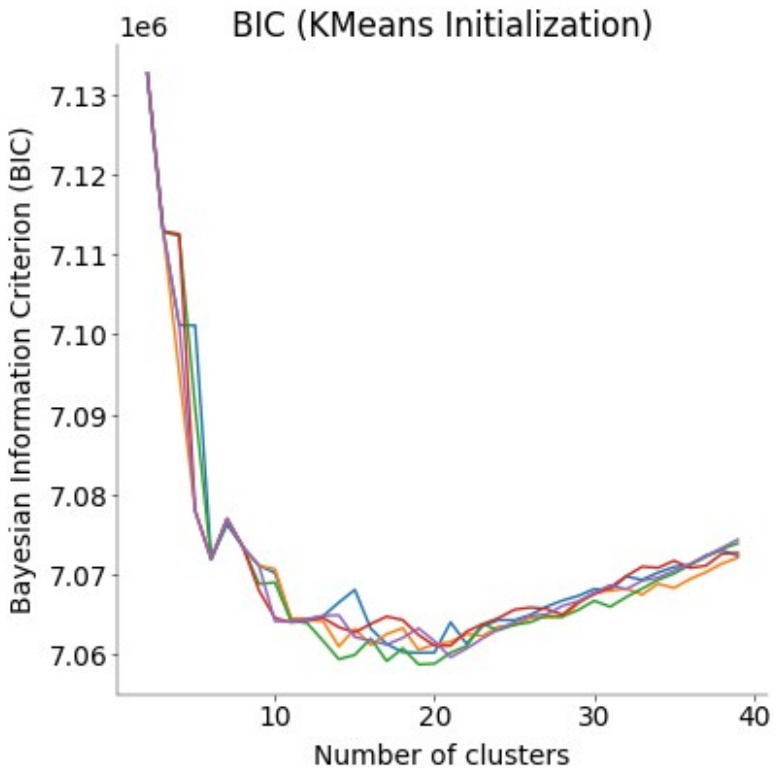
Show the plot of the BIC as a function of the number of mixture components on the real data, highlight the robust estimate and based on that the best number of clusters.

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

```
# ------------------------------------------------------------
# Run the algorithm on the set of extracted features (0.5 pts)
# ------------------------------------------------------------

K = np.arange(2, 40)
num_seeds = 5

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


for idx, k in enumerate(K):
    for seed in range(num_seeds):
        mog_labels, mog_m, mog_S, mog_p = fit_mog(
            b,
            n_clusters=k,
            random_seed=seed,
            init="kmeans",
        )
        BIC[seed, idx], LL[seed, idx] = mog_bic(b, mog_m, mog_S,
mog_p)
```

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

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

# plot BIC
for idx in range(BIC.shape[0]):
    plt.plot(K, BIC[idx])

plt.title("BIC (KMeans Initialization)")
plt.xlabel("Number of clusters")
plt.ylabel("Bayesian Information Criterion (BIC)")
plt.show()
```



Refit model with lowest BIC and plot data points

```
random_seed, kk = np.where(BIC == BIC.min())
random_seed = random_seed[0]
kk = kk[0]
```

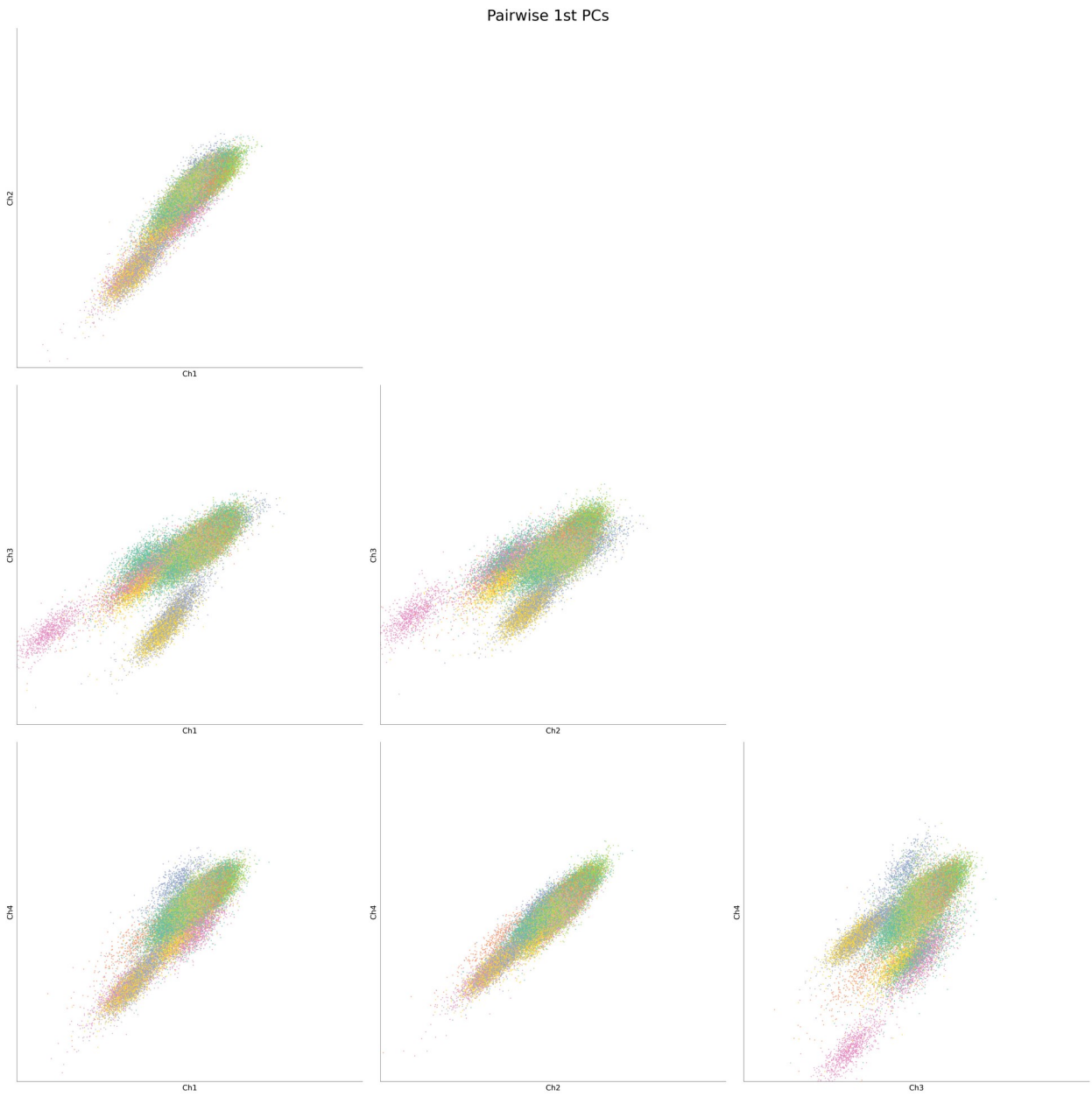
```
print(f"lowest BIC: # clusters = {K[kk]}")
labels, m, S, p = fit_mog(b, K[kk], random_seed=random_seed,
init="kmeans")
```

```
lowest BIC: # clusters = 19
```

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


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=(20, 20), layout="constrained", dpi=400
)

# index of the 1st PC in `b`
i = {"Ch1": 0, "Ch2": 3, "Ch3": 6, "Ch4": 9}


#
-------------------------------------------------------------------------
-----------------------------------
# Create a scatterplot of the projections of the spikes for all
pairwise combinations of the 1st PCs (1 pt)
cmap = "Set2"
ax["Ch2 vs Ch1"].scatter(
    b[:, i["Ch2"]], b[:, i["Ch1"]], c=labels, s=1, alpha=0.7,
cmap=cmap
)
ax["Ch3 vs Ch1"].scatter(
    b[:, i["Ch3"]], b[:, i["Ch1"]], c=labels, s=1, alpha=0.7,
cmap=cmap
)
ax["Ch3 vs Ch2"].scatter(
    b[:, i["Ch3"]], b[:, i["Ch2"]], c=labels, s=1, alpha=0.7,
cmap=cmap
)
ax["Ch4 vs Ch1"].scatter(
    b[:, i["Ch4"]], b[:, i["Ch1"]], c=labels, s=1, alpha=0.7,
cmap=cmap
```

```
)
ax["Ch4 vs Ch2"].scatter(
    b[:, i["Ch4"]], b[:, i["Ch2"]], c=labels, s=1, alpha=0.7,
cmap=cmap
)
ax["Ch4 vs Ch3"].scatter(
    b[:, i["Ch4"]], b[:, i["Ch3"]], c=labels, s=1, alpha=0.7,
cmap=cmap
)


#
------------------------------------------------------------------------
-------------------------------------

for mo in np.ravel(mosaic):
    if mo == ".":
        continue
    y, x = mo.split(" vs ")

    ax[mo].set_xlabel(x)
    ax[mo].set_ylabel(y)
    ax[mo].set_xlim((-1200, 1200))
    ax[mo].set_ylim((-1200, 1200))
    ax[mo].set_xticks([])
    ax[mo].set_yticks([])

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

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



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)

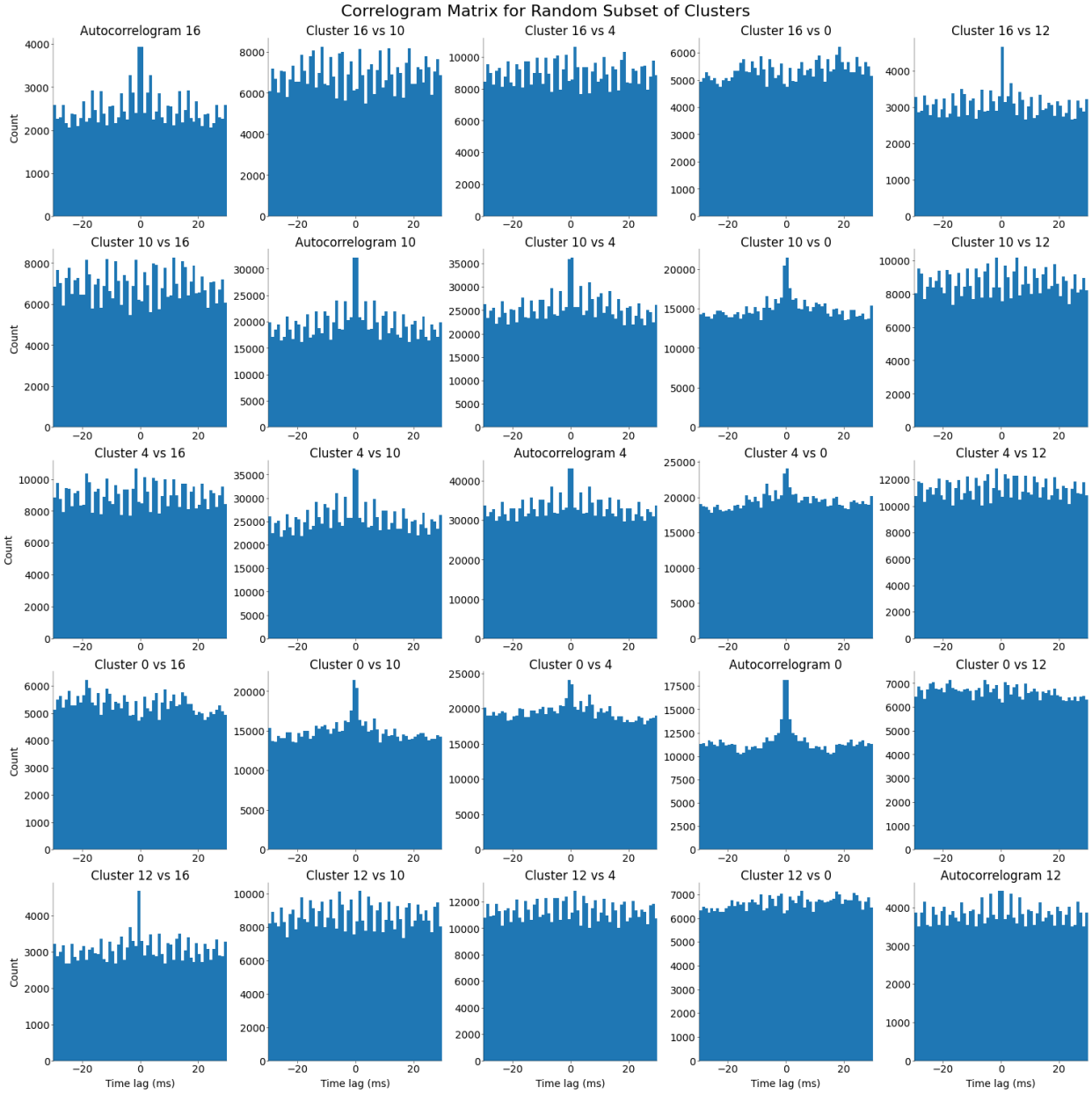
For the correlogram an efficient implementation is very important - looping over all spike times is not feasible. Instead, make use of numpy vectorization and broadcasting - you can use functions such as tile or repeat.

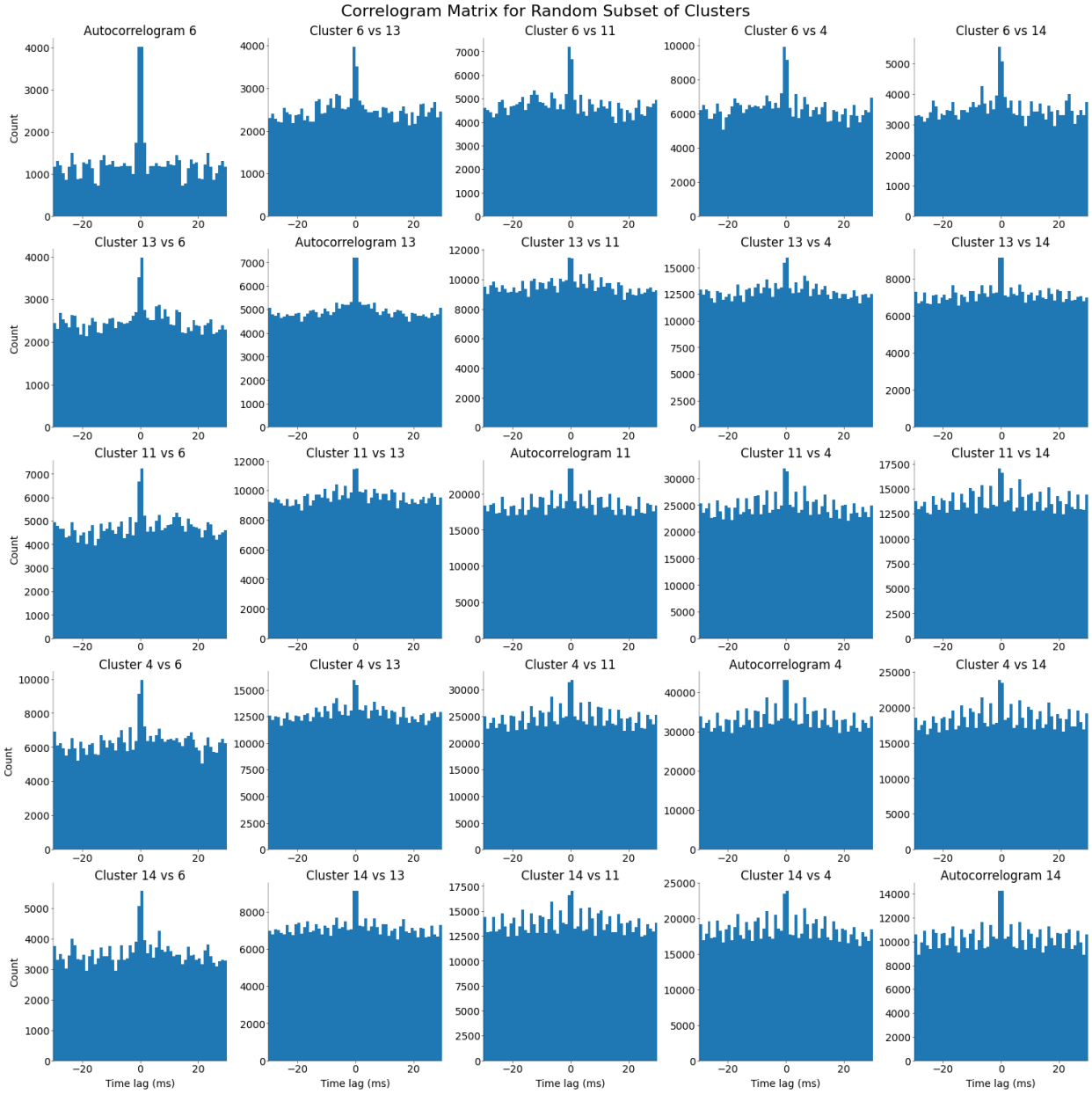
```
#
-----------------------------------------------------------------------
# Implement a function for calculating the spike time differences
(1pt)
#
-----------------------------------------------------------------------
def cross_time_diff(spiketimes1: np.ndarray, spiketimes2: np.ndarray)
-> np.ndarray:
    """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[:, np.newaxis] - spiketimes2[np.newaxis, :]

#
-----------------------------------------------------------------------
-------------
# Calculate and plot auto- and cross correlograms and answer the
questions (1+1 pts)
#
-----------------------------------------------------------------------
-------------


def compute_correlogram(
```

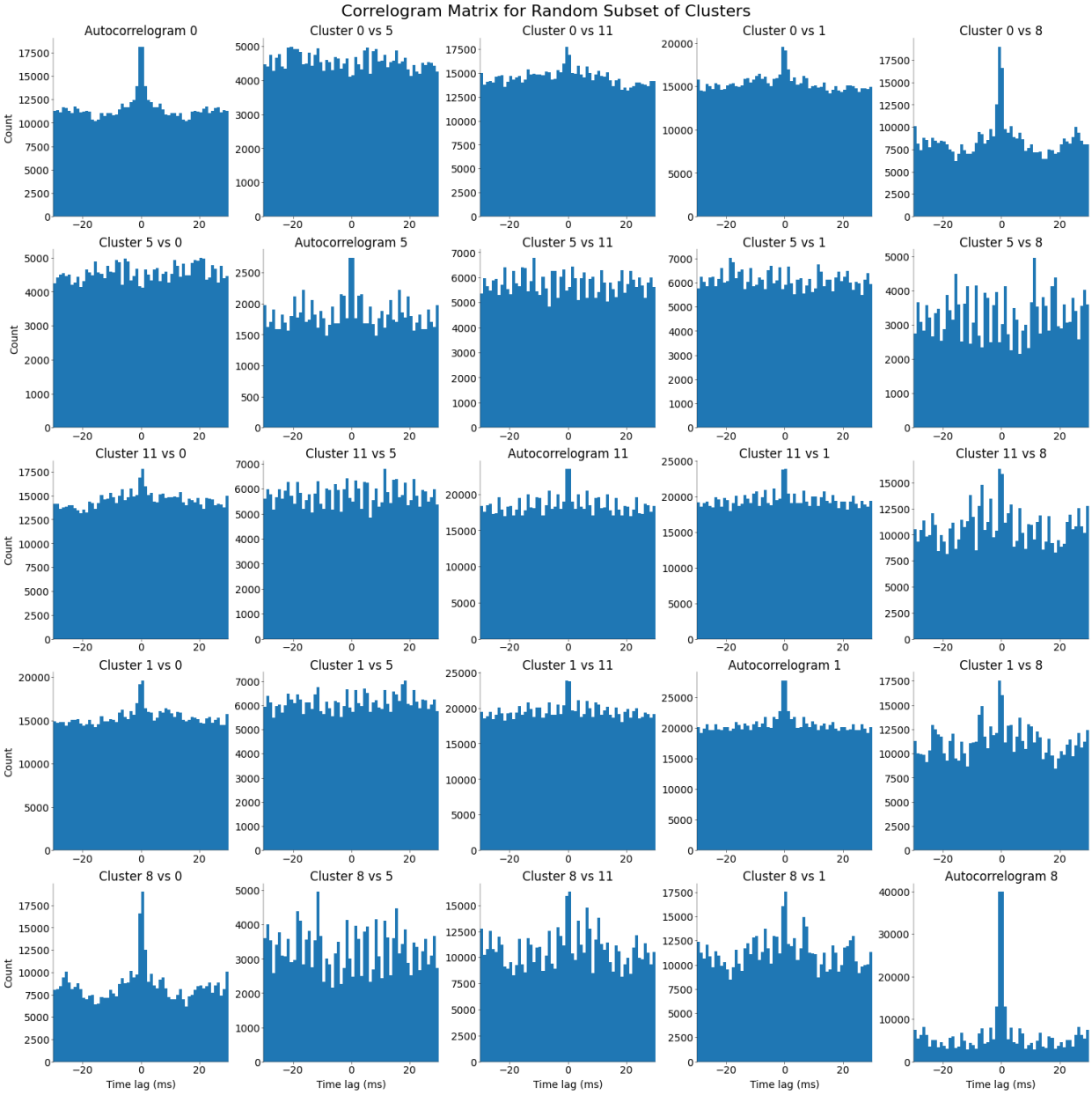
```
    spiketimes1, spiketimes2, bin_size=1.0, window_ms=30.0, auto=False
):
    """
    Computes an auto- or cross-correlogram.

    Parameters
    ----------
    spiketimes1: np.ndarray
        Spike times of cluster 1 (in ms)
    spiketimes2: np.ndarray
        Spike times of cluster 2 (in ms)
    bin_size: float
        Bin size for the histogram in ms
    window_ms: float
        Half-width of time window in ms
    auto: bool
        Whether to compute autocorrelogram (avoid zero-lag if True)

    Returns
    -------
    bins: np.ndarray
        Bin centers
    counts: np.ndarray
        Histogram counts for each bin
    """
    # Compute pairwise differences
    time_diffs = cross_time_diff(spiketimes1, spiketimes2).ravel()

    # Exclude self-pairs (0 ms) in autocorrelogram
    if auto:
        # time_diffs = time_diffs[time_diffs != 0]
        time_diffs = time_diffs[
            np.abs(time_diffs) > 1e-6
        ]  # floating point error, therefore may not be exaclty 0

    # Filter within range
    mask = (time_diffs >= -window_ms) & (time_diffs <= window_ms)
    time_diffs = time_diffs[mask]

    # Compute histogram
    bins = np.arange(-window_ms, window_ms + bin_size, bin_size)
    counts, _ = np.histogram(time_diffs, bins=bins)

    # Return bin centers for plotting
    bin_centers = bins[:-1] + bin_size / 2
    return bin_centers, counts


def plot_correlogram_matrix(
    spiketimes, labels, bin_size=1.0, window_ms=30.0, random_seed=42
```

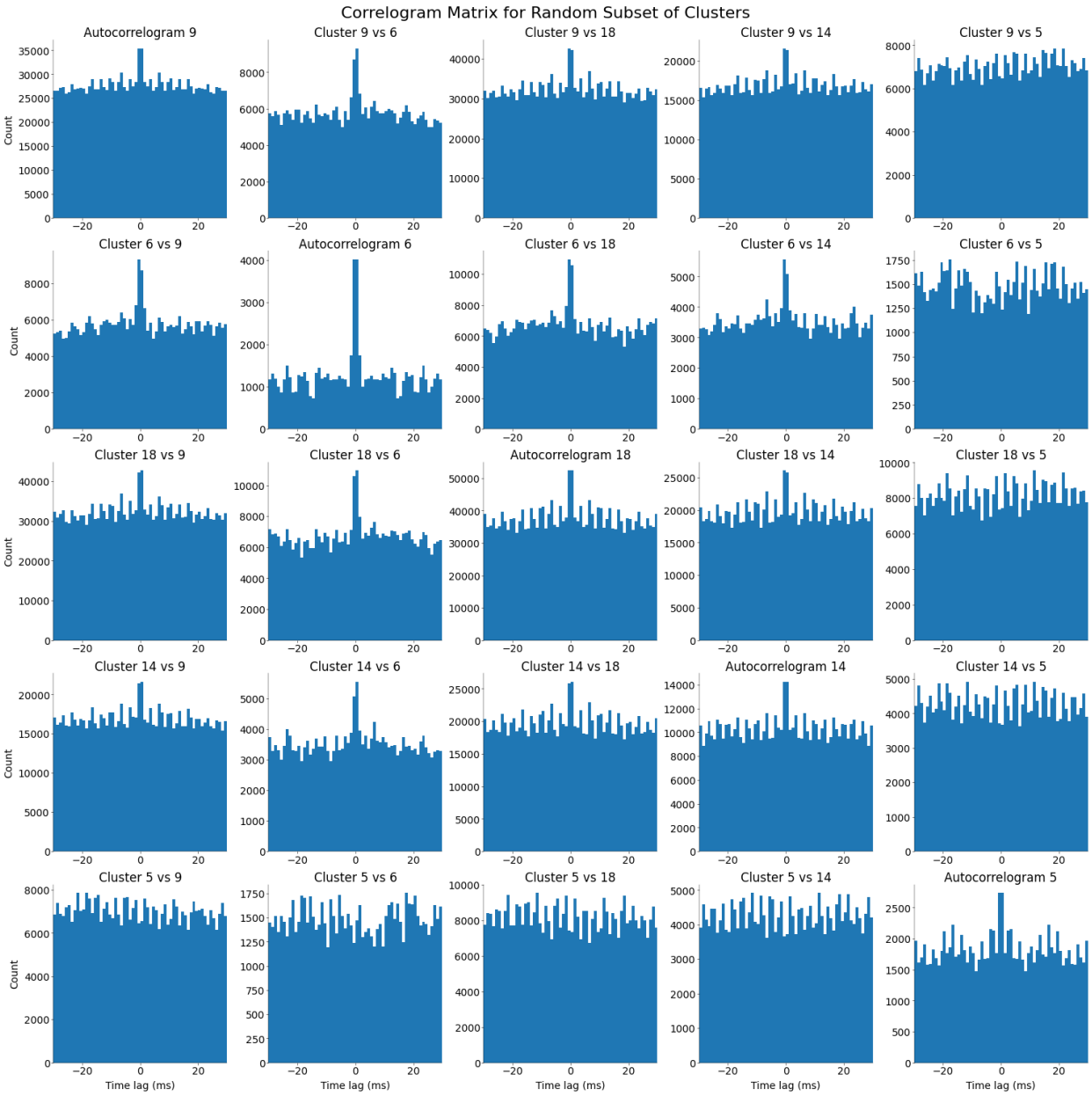
```
):
    """
    Plot an NxN correlogram matrix with auto-correlograms on the
diagonal
    and cross-correlograms on the off-diagonal for a random subset of
5 clusters.

    Parameters:
    - spiketimes: np.ndarray, spike times of all clusters
    - labels: np.ndarray, cluster labels for each spike
    - bin_size: float, bin size for the histogram in ms
    - window_ms: float, half-width of the time window in ms
    - random_seed: int, seed for reproducibility
    """
    np.random.seed(random_seed)

    # Get unique cluster labels and randomly select 5 clusters
    unique_labels = np.unique(labels)
    if len(unique_labels) > 5:
        selected_labels = np.random.choice(unique_labels, size=5,
replace=False)
    else:
        selected_labels = unique_labels

    n_clusters = len(selected_labels)

    # Create NxN subplot grid
    fig, axes = plt.subplots(
        n_clusters, n_clusters, figsize=(15, 15),
constrained_layout=True
    )

    for i, cluster_i in enumerate(selected_labels):
        for j, cluster_j in enumerate(selected_labels):
            ax = axes[i, j]
            spiketimes_i = spiketimes[labels == cluster_i]
            spiketimes_j = spiketimes[labels == cluster_j]

            if i == j:
                # Auto-correlogram
                bins, counts = compute_correlogram(
                    spiketimes_i, spiketimes_j, bin_size, window_ms,
auto=True
                )
                ax.set_title(f"Autocorrelogram {cluster_i}")
            else:
                # Cross-correlogram
                bins, counts = compute_correlogram(
                    spiketimes_i, spiketimes_j, bin_size, window_ms,
auto=False
```

```
                )
                ax.set_title(f"Cluster {cluster_i} vs {cluster_j}")

            ax.bar(bins, counts, width=bin_size, align="center")
            ax.set_xlim(-window_ms, window_ms)

            if i == n_clusters - 1:
                ax.set_xlabel("Time lag (ms)")

            if j == 0:
                ax.set_ylabel("Count")

    plt.suptitle("Correlogram Matrix for Random Subset of Clusters",
fontsize=16)
    plt.show()


# Doing a Correlogram/Auto-correlogram matrix for all 16 clusters is
too large, therefore we
# we will do it for random subsets of clusters, multiple times
plot_correlogram_matrix(t, labels, bin_size=1.0, window_ms=30.0,
random_seed=40)
plot_correlogram_matrix(t, labels, bin_size=1.0, window_ms=30.0,
random_seed=41)
plot_correlogram_matrix(t, labels, bin_size=1.0, window_ms=30.0,
random_seed=42)
plot_correlogram_matrix(t, labels, bin_size=1.0, window_ms=30.0,
random_seed=43)
```


Correlogram Matrix for Random Subset of Clusters
Autocorrelogram 16
Cluster 16 vs 10
Cluster 16 vs 4
Cluster 16 vs 0
Cluster 16 vs 12
Cluster 10 vs 16
Autocorrelogram 10
Cluster 10 vs 4
Cluster 10 vs 0
Cluster 10 vs 12
Cluster 4 vs 16
Cluster 4 vs 10
Autocorrelogram 4
Cluster 4 vs 0
Cluster 4 vs 12
Cluster 0 vs 16
Cluster 0 vs 10
Cluster 0 vs 4
Autocorrelogram 0
Cluster 0 vs 12
Cluster 12 vs 16
Cluster 12 vs 10
Cluster 12 vs 4
Cluster 12 vs 0
Autocorrelogram 12
Count
Time lag (ms)


Correlogram Matrix for Random Subset of Clusters
Autocorrelogram 6
Cluster 6 vs 13
Cluster 6 vs 11
Cluster 6 vs 4
Cluster 6 vs 14
Cluster 13 vs 6
Autocorrelogram 13
Cluster 13 vs 11
Cluster 13 vs 4
Cluster 13 vs 14
Cluster 11 vs 6
Cluster 11 vs 13
Autocorrelogram 11
Cluster 11 vs 4
Cluster 11 vs 14
Cluster 4 vs 6
Cluster 4 vs 13
Cluster 4 vs 11
Autocorrelogram 4
Cluster 4 vs 14
Cluster 14 vs 6
Cluster 14 vs 13
Cluster 14 vs 11
Cluster 14 vs 4
Autocorrelogram 14
Count
Time lag (ms)


Correlogram Matrix for Random Subset of Clusters
Autocorrelogram 0
Cluster 0 vs 5
Cluster 0 vs 11
Cluster 0 vs 1
Cluster 0 vs 8
Cluster 5 vs 0
Autocorrelogram 5
Cluster 5 vs 11
Cluster 5 vs 1
Cluster 5 vs 8
Cluster 11 vs 0
Cluster 11 vs 5
Autocorrelogram 11
Cluster 11 vs 1
Cluster 11 vs 8
Cluster 1 vs 0
Cluster 1 vs 5
Cluster 1 vs 11
Autocorrelogram 1
Cluster 1 vs 8
Cluster 8 vs 0
Cluster 8 vs 5
Cluster 8 vs 11
Cluster 8 vs 1
Autocorrelogram 8
Count
Time lag (ms)



Questions

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

In theory, one should be able to see this well from the auto-correlogram (diagonal), but it seems like a mistake happend, and we can not see clusters that do belong to a single neuron. It's possible that we have double-count spikes, although we should have excluded these in `CodingLab1`.

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

Hard to say. Don't see any evidence of this here.

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

Given the high correlation around time-lag ~ 0ms, I think many of these clusters belong to multiple neurons.

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

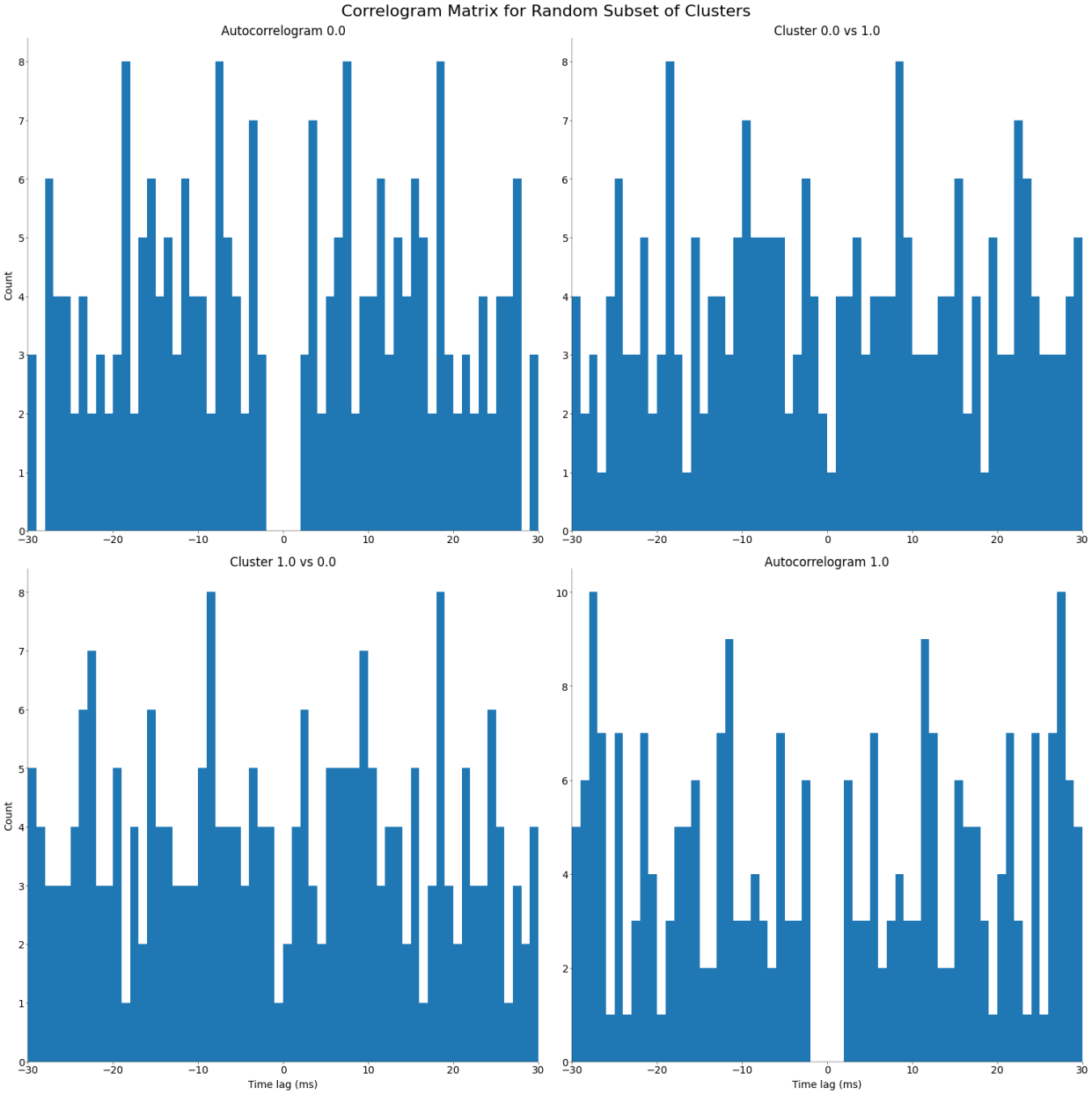
The refractory period is the brief time period (1-2ms), where a neuron is not able to send an action potential after it just has sent an action potential. In the autocorrelogram plots, we can see that the spiketimes of a cluster with itself are highly correlated around a time lag of 1ms. Normally, we would expect the opposite that there is a gap (see 'Task 5B') in the correlogram around time lag 0ms.

Task 5B: Testing autocorrelogram/crosscorrelogram with toy data

I wanted to see whether the lack of a gap around timelag = 0ms in the plots above was due to our MoG implementation or Task 5 code. Below is some toy data for 2 neurons, I can see that the Task 5 code works fine, so maybe it's due to our MoG implementation..

```
import numpy as np
import matplotlib.pyplot as plt


def generate_spike_train(duration_ms=10000, rate_hz=20,
refractory_ms=2, seed=None):
    """
    Generate a spike train with a refractory period using a simple
rejection method.

    Parameters:
    - duration_ms: total duration in ms
    - rate_hz: average firing rate in Hz
    - refractory_ms: minimum time between spikes
    - seed: random seed

    Returns:
    - spiketimes: np.ndarray of spike times (in ms)
    """
    if seed is not None:
        np.random.seed(seed)

    spiketimes = []
    tt = 0
    while tt < duration_ms:
        isi = np.random.exponential(scale=1000 / rate_hz)  # ISI in ms
        if isi >= refractory_ms:
            tt += isi
            if tt < duration_ms:
                spiketimes.append(tt)
    return np.array(spiketimes)
```

```
# Parameters
duration = 10000  # ms (10 seconds)
rate = 20  # Hz
refractory = 2  # ms

# Generate spike trains
spikes_neuron1 = generate_spike_train(duration, rate, refractory,
seed=1)
spikes_neuron2 = generate_spike_train(duration, rate, refractory,
seed=2)


all_spikes = np.concatenate([spikes_neuron1, spikes_neuron2])
labels = np.concatenate([np.zeros_like(spikes_neuron1),
np.ones_like(spikes_neuron2)])

# Test
plot_correlogram_matrix(all_spikes, labels, bin_size=1.0,
window_ms=30.0)
```


Correlogram Matrix for Random Subset of Clusters
Autocorrelogram 0.0
Cluster 0.0 vs 1.0
Cluster 1.0 vs 0.0
Autocorrelogram 1.0
Count
Time lag (ms)