Illustration of cocoLMB algorithm

The algorithm cocoLMB is a variant of the well-known LMB algorithm. It assume a partition of rows and columns but each column cluster are conditionnal to the row cluster while traditional LMB assume that the row and column clusters are independent.

The algorithm is based on the maximization of the likelihood of the data. The algorithm is iterative and the number of clusters is fixed. The algorithm is implemented in the R package `cocoLMB`.

```python
import numpy as np
from scipy.stats import norm
from sklearn.cluster import KMeans

def log_err(x):
    return np.log(np.maximum(x, np.finfo(float).eps))

def softmax_log(X):
    exp_X = np.exp(X - np.max(X, axis=1, keepdims=True))
    return exp_X / np.sum(exp_X, axis=1, keepdims=True)

def elbo_cocolbm(params):
    tau, rho_ks, nu, pi_k, tmp = params['tau'], params['rho_ks'], params['nu'], params['pi_k
    N, K, p, Q = tau.shape[0], tmp.shape[1], nu.shape[1], nu.shape[2]

    tau_tmp = np.tile(tau[:, :, None, None], (1, 1, p, Q))
    nu_tmp = np.tile(nu[None, :, :, :], (N, 1, 1, 1))

    res = np.sum(tau_tmp * tmp * nu_tmp)
    res += np.sum(nu * log_err(np.tile(rho_ks[:, None, :], (1, p, 1))))
    res += np.sum(tau * log_err(np.tile(pi_k[None, :], (N, 1))))

    entropy_nu = -np.sum(nu * log_err(nu))
    entropy_tau = -np.sum(tau * log_err(tau))
    return res + entropy_nu + entropy_tau

def update_tau_cocolbm(params):
    nu, pi_k, tmp = params['nu'], params['pi_k'], params['tmp']
    N, K, p, Q = tmp.shape

    nu_tmp = np.tile(nu[None, :, :, :], (N, 1, 1, 1))
    tmp2 = np.sum(tmp * nu_tmp, axis=(2, 3))
    Tik = tmp2 + np.log(pi_k)[None, :] - 1
    params['tau'] = softmax_log(Tik)
    return params

def update_nu_cocolbm(params):
    tau, rho_ks, tmp = params['tau'], params['rho_ks'], params['tmp']
    N, K, p, Q = tmp.shape

    tau_tmp = np.tile(tau[:, :, None, None], (1, 1, p, Q))
    tmp2 = np.sum(tmp * tau_tmp, axis=0)
    V_kjs = tmp2 + np.log(rho_ks)[:, None, :] - 1
    res = np.array([softmax_log(V_kjs[k, :, :]) for k in range(K)])
    params['nu'] = res
    return params

def update_pi_cocolbm(params):
    tau = params['tau']
    params['pi_k'] = np.mean(tau, axis=0)
    return params

def update_rho_cocolbm(params):
    nu = params['nu']
    params['rho_ks'] = np.sum(nu, axis=1) / nu.shape[1]
    return params

def update_mu_cocolbm(params):
    X, tau, nu = params['X'], params['tau'], params['nu']
    K, Q = nu.shape[0], nu.shape[2]

    res = np.zeros((K, Q))  # Initialize result matrix

    for k in range(K):
        for s in range(Q):
            num = ((tau[:, k][:, None] * X) @ nu[k, :, s]).sum()  # Ensure scalar
            den = (tau[:, k].sum() * nu[k, :, s].sum())  # Ensure scalar

            res[k, s] = num / den if den != 0 else 0  # Avoid division by zero

    params['mu_ks'] = res
    return params

def update_sigma2_cocolbm(params):
    X, tau, nu, mu_ks = params['X'], params['tau'], params['nu'], params['mu_ks']
    K, Q = mu_ks.shape

    res = np.zeros((K, Q))  # Initialize result matrix

    for k in range(K):
        for s in range(Q):
            num = ((tau[:, k][:, None] * (X - mu_ks[k, s])**2) @ nu[k, :, s]).sum()  # Ensure
            den = tau[:, k].sum() * nu[k, :, s].sum()  # Ensure scalar

            res[k, s] = num / den if den != 0 else np.finfo(float).eps  # Avoid division by

    params['sigma2_ks'] = res
    return params

def initialization_cocolbm(X, K, Q):
    params = {'X': X, 'K': K, 'Q': Q, 'N': X.shape[0], 'p': X.shape[1]}

    kmeans = KMeans(n_clusters=K, n_init=50).fit(X)
    tau = np.eye(K)[kmeans.labels_]

    nu = np.zeros((K, X.shape[1], Q))
    for k in range(K):
        sub_X = X[kmeans.labels_ == k].T
        km_q = KMeans(n_clusters=Q, n_init=50).fit(sub_X)
        nu[k] = np.eye(Q)[km_q.labels_]

    params['tau'] = tau
    params['nu'] = nu
    return params

def calcul_log_prob_cocolbm(params):
    N, K, p, Q = params['N'], params['K'], params['p'], params['Q']
    mu_ks, sigma2_ks = params['mu_ks'], params['sigma2_ks']
    tmp = np.zeros((N, K, p, Q))
    for k in range(K):
        for s in range(Q):
            tmp[:, k, :, s] = norm.logpdf(params['X'], loc=mu_ks[k, s], scale=np.sqrt(sigma2_
    params['tmp'] = tmp
    return params

def cocolbm_fixe(X, K, Q, iter_max=10):
    params = initialization_cocolbm(X, K, Q)
    for _ in range(iter_max):
        params = update_pi_cocolbm(params)
        params = update_rho_cocolbm(params)
        params = update_mu_cocolbm(params)
        params = update_sigma2_cocolbm(params)
        params = calcul_log_prob_cocolbm(params)
        params = update_nu_cocolbm(params)
        params = update_tau_cocolbm(params)

    params['elbo'] = elbo_cocolbm(params)
    params['ICL'] = params['elbo'] - 0.5 * (K - 1) * np.log(params['N']) - 0.5 * K * (Q - 1)
    return params

import numpy as np
from scipy.stats import norm
from sklearn.cluster import KMeans

def cocolbm(X, K_set, Q_set, iter_max=10):
    best_elbo, best_icl = {'elbo': -np.inf}, {'ICL': -np.inf}
    best_ELBO_model, best_ICL_model = None, None

    for K in K_set:
        for Q in Q_set:
            model = cocolbm_fixe(X, K, Q, iter_max)
            if model['elbo'] > best_elbo['elbo']:
                best_elbo = model
                best_ELBO_model = model.copy()
            if model['ICL'] > best_icl['ICL']:
                best_icl = model
                best_ICL_model = model.copy()

    return {'best_ICL': best_icl, 'best_ELBO': best_elbo, 'best_ICL_model': best_ICL_model,
```

Simulation of a matrix with blocks

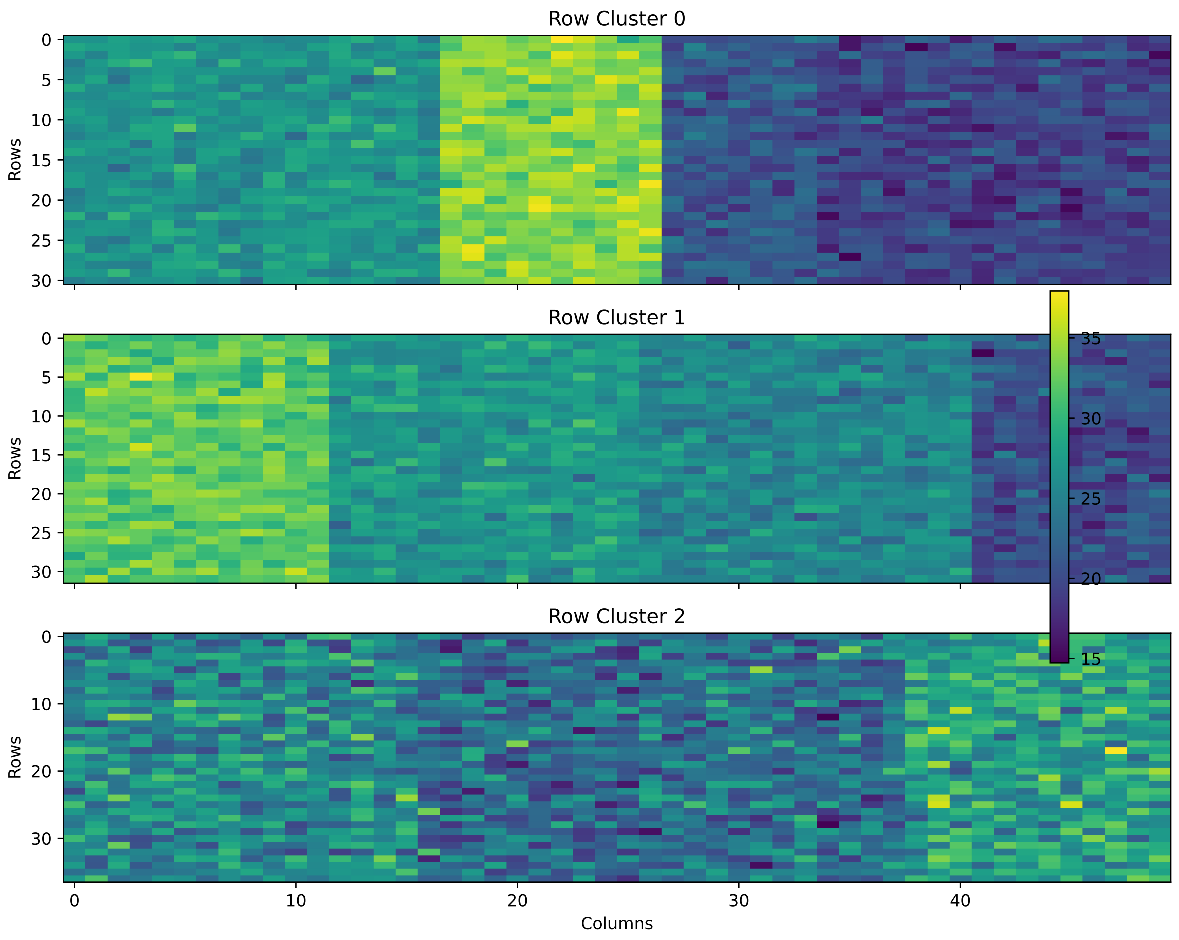
Let simulate a block matrix according the assumptions of the cocoLMB model. We assume that the matrix is of size $n \times p$ and that the matrix is partitioned into K row clusters and Q column clusters coditionnal to each row cluster.

```python
import numpy as np

def simulate_block_matrix(N, p, K, Q, mean_range=(0, 30), std=3):
    """
    Simulate a block matrix with given dimensions and block structure.

    Parameters:
    - N: Number of rows
    - p: Number of columns
    - K: Number of row clusters
    - Q: Number of column clusters per row cluster
    - mean_range: Range of means for Gaussian distributions
    - std: Standard deviation for Gaussian distributions

    Returns:
    - X: Simulated block matrix
    - row_labels: Row cluster labels
    - col_partitions: Dictionary of column partitions per row cluster
    """
    # Generate random row labels
    row_labels = np.random.choice(K, N)

    # Initialize the block matrix
    X = np.zeros((N, p))

    # Dictionary to store column partitions for each row cluster
    col_partitions = {}

    for k in range(K):
        # Assign a different column partition for each row cluster
        col_partitions[k] = np.random.choice(Q, p)  # Different partition per row cluster

    # Fill the block matrix with samples from Gaussian distributions
    for k in range(K):
        row_indices = np.where(row_labels == k)[0]

        for q in range(Q):
            col_indices = np.where(col_partitions[k] == q)[0]
            mean = np.random.uniform(*mean_range)

            # Assign Gaussian values only if both row and col clusters exist
            if len(row_indices) > 0 and len(col_indices) > 0:
                X[np.ix_(row_indices, col_indices)] = np.random.normal(mean, std, size=(len(i

    return X, row_labels, col_partitions



# Example usage
np.random.seed(321)
N, p, K, Q = 100, 50, 3, 4
X, row_labels, col_partitions = simulate_block_matrix(N, p, K, Q)
print(X)
```

```
[[14.52499093 -1.52758851  6.73154465 ...  5.46236296 18.2596033
  12.99680522]
 [31.17334493 30.96172676 23.25572059 ... 25.61615849 25.71034681
  23.35440934]
 [18.19116422  2.66312805 -7.26465458 ...  2.7002901  11.76001043
  16.82641762]
 ...
 [23.67130934 26.54010016 21.94641658 ... 28.96591846 23.21591072
  23.81415705]
 [14.41917511  5.37898892 -0.29289188 ... 14.76055563  8.37715737
  28.80131283]
 [13.38680354  9.00161686 -2.7157856  ...  8.27603894 13.51596849
  22.62810116]]
```

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

def plot_clustered_matrix(X, row_labels, col_partitions, K):
    """
    Plot K submatrices corresponding to row clusters with columns ordered according to the co

    Parameters:
    - X: The block matrix (N, p)
    - row_labels: Array of row cluster labels (size N)
    - col_partitions: Dictionary mapping each row cluster to its column partitions
    - K: Number of row clusters
    """
    # Create a figure with K subplots (one for each row cluster)
    fig, axes = plt.subplots(K, 1, figsize=(10, 8), sharex=True)

    for k in range(K):
        # Get row indices for the current row cluster
        row_indices = np.where(row_labels == k)[0]

        # Get column indices ordered according to their column cluster for this row cluster
        sorted_cols = np.argsort(col_partitions[k])

        # Extract and sort the corresponding submatrix
        X_sub = X[row_indices, :][:, sorted_cols]

        # Plot the submatrix
        ax = axes[k] if K > 1 else axes
        im = ax.imshow(X_sub, aspect='auto', cmap='viridis')
        ax.set_title(f'Row Cluster {k}')
        ax.set_ylabel('Rows')

    # Add colorbar
    fig.colorbar(im, ax=axes, orientation='vertical', fraction=0.02, pad=0.04)
    plt.xlabel('Columns')
    plt.tight_layout()
    plt.show()

# Example usage
plot_clustered_matrix(X, row_labels, col_partitions, K)
```

```
/var/folders/c5/vctv2wln4sx9_796g1nmhl0r0000gn/T/ipykernel_54147/2092487613.py:36: UserWarni
  plt.tight_layout()
```



The labels of the row clusters are stored in the vector `row_cluster` and the labels of the column clusters are stored in the vector `col_cluster`. The matrix is stored in the matrix `X`. The matrix is generated as follows:

```python
import numpy as np

def create_labels_matrix(N, p, row_labels, col_partitions):
    """
    Create a matrix to store the labels indicating row and column cluster assignments.

    Parameters:
    - N: Number of rows
    - p: Number of columns
    - row_labels: Array of row cluster labels (size N)
    - col_partitions: Dictionary mapping each row cluster to its column partitions
    Returns:
    - labels_matrix: Matrix (N, p) with labels in the format "rowCluster-colCluster"
    """
    labels_matrix = np.empty((N, p), dtype=object)  # Create an empty matrix to store labels

    for i in range(N):
        for j in range(p):
            row_cluster = row_labels[i]  # Get the row cluster
            col_cluster = col_partitions[row_cluster][j]  # Get the column cluster from the p
            labels_matrix[i, j] = f"{row_cluster}-{col_cluster}"  # Assign formatted label

    return labels_matrix

# Example usage
N, p = 100, 50
row_labels = np.random.choice(3, N)
col_partitions = {k: np.random.choice(4, p) for k in range(3)}  # Example column partitions
labels_matrix = create_labels_matrix(N, p, row_labels, col_partitions)
print(labels_matrix)
```

```
[['1-3' '1-2' '1-0' ... '1-2' '1-2' '1-0']
 ['2-2' '2-2' '2-1' ... '2-1' '2-3' '2-3']
 ['2-2' '2-2' '2-1' ... '2-1' '2-3' '2-3']
 ...
 ['0-1' '0-3' '0-3' ... '0-0' '0-2' '0-2']
 ['0-1' '0-3' '0-3' ... '0-0' '0-2' '0-2']
 ['1-3' '1-2' '1-0' ... '1-2' '1-2' '1-0']]
```

Test of the cocoLMB algorithm

Run the algorithm with model choice strategy using the simulated matrix.

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

# Test the cocoLMB algorithm on the simulated matrix
np.random.seed(321)
N, p, K, Q = 100, 50, 3, 4
X, row_labels, col_partitions = simulate_block_matrix(N, p, K, Q)

# Define the range of clusters to test
```

```python
K_set = range(2, 6)  # Possible values for row clusters
Q_set = range(2, 6)  # Possible values for column clusters

# Run the cocoLMB algorithm
result = cocolbm(X, K_set, Q_set, iter_max=10)

# Extract best ICL model
best_ICL_model = result['best_ICL_model']
row_partition_estimated = np.argmax(best_ICL_model['tau'], axis=1)  # Row partition

# Extract column partitions for each row cluster
col_partition_estimated = {}
for k in range(best_ICL_model['K']):
    col_partition_estimated[k] = np.argmax(best_ICL_model['nu'][k], axis=1)

# Print the best results
print("Best ICL Model:")
print("K:", best_ICL_model['K'])
print("Q:", best_ICL_model['Q'])
print("ICL Score:", best_ICL_model['ICL'])
print("Row Partition:", row_partition_estimated)
print("Column Partition Dictionary:", col_partition_estimated)

print("\nBest ELBO Model:")
print("K:", result['best_ELBO']['K'])
print("Q:", result['best_ELBO']['Q'])
print("ELBO Score:", result['best_ELBO']['elbo'])
```

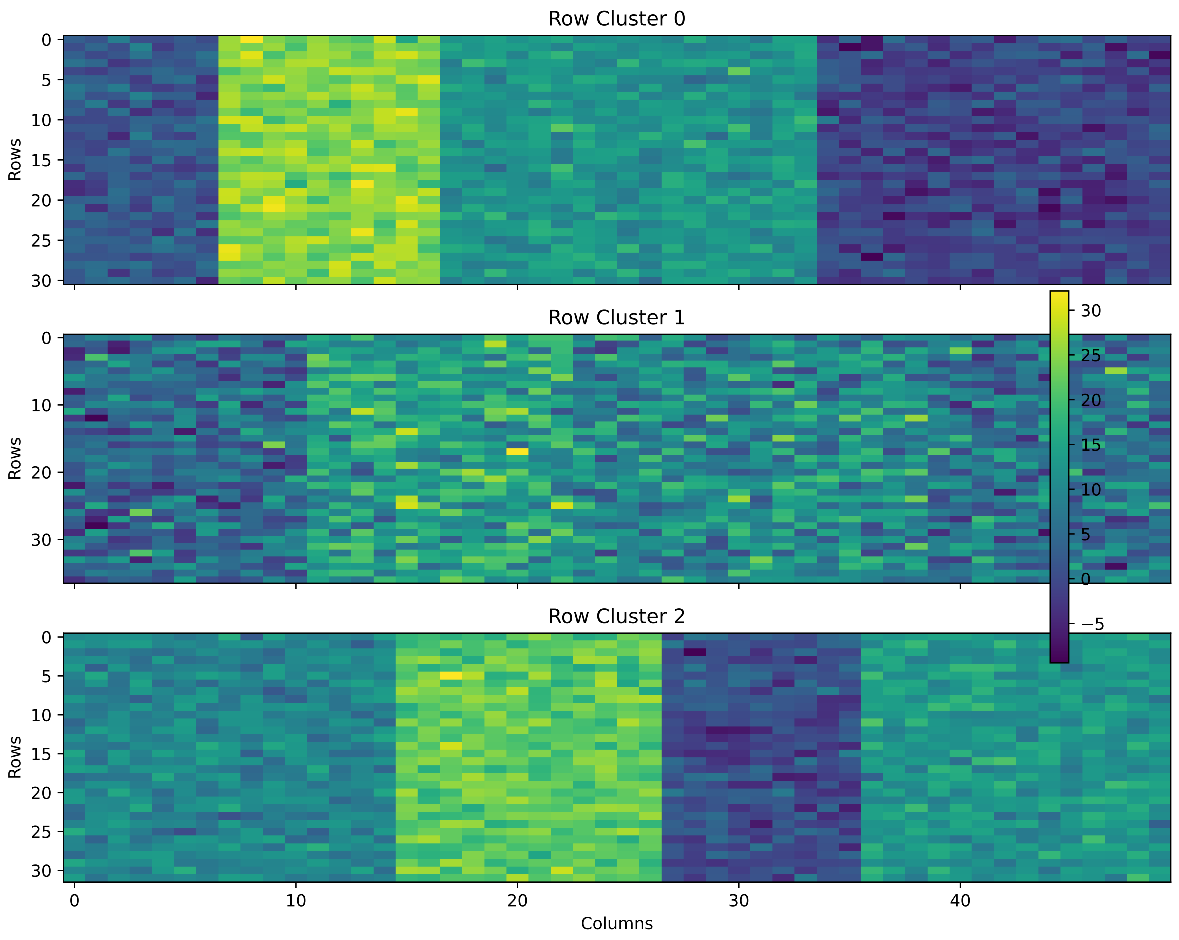
```
Best ICL Model:
K: 3
Q: 4
ICL Score: -13031.572413693373
Row Partition: [0 1 0 2 0 2 0 1 0 0 2 0 2 2 1 0 2 2 1 1 1 2 1 1 2 1 0 1 2 2 1 1 2 1 2 2 0
 1 0 0 1 0 1 0 2 2 0 1 0 2 0 1 2 1 2 2 0 1 0 0 0 2 0 1 0 2 1 0 1 2 2 1 0 2
 0 1 2 2 0 1 1 1 2 2 1 0 1 1 1 2 1 0 0 1 1 1 0 1 2 2]
Column Partition Dictionary: {0: array([2, 3, 3, 0, 2, 2, 1, 2, 1, 2, 3, 2, 3, 3, 2, 3, 1, 2
       2, 3, 3, 0, 0, 3, 3, 1, 0, 1, 1, 2, 3, 0, 3, 1, 2, 3, 1, 2, 1, 3,
       1, 3, 2, 0, 2, 2]), 1: array([1, 2, 3, 2, 1, 2, 1, 2, 2, 0, 0, 0, 1, 3, 2, 2, 2, 1, 3
       3, 3, 1, 0, 1, 2, 1, 1, 0, 2, 3, 3, 1, 0, 0, 0, 2, 3, 2, 0, 2, 2,
       1, 3, 0, 1, 2, 3]), 2: array([3, 0, 2, 0, 2, 0, 0, 2, 2, 1, 3, 2, 0, 1, 2, 3, 0, 3, 1
       3, 2, 1, 3, 3, 3, 3, 3, 1, 2, 1, 0, 1, 0, 0, 1, 1, 0, 0, 3, 0, 3,
       1, 3, 0, 0, 0, 1])}
```

```
Best ELBO Model:
K: 3
Q: 5
ELBO Score: -12901.37259989807
```

Plot the results.

```python
from sklearn.metrics import adjusted_rand_score
plot_clustered_matrix(X, row_partition_estimated, col_partition_estimated, K)


# Compute and print the Rand Index
truth = create_labels_matrix(N, p, row_labels, col_partitions)
estimation = create_labels_matrix(N, p, row_partition_estimated, col_partition_estimated)

# Flatten the matrices for Rand Index computation
truth_flat = truth.flatten()
estimation_flat = estimation.flatten()

rand_index = adjusted_rand_score(truth_flat, estimation_flat)
print("Adjusted Rand Index:", rand_index)
```

```
/var/folders/c5/vctv2wln4sx9_796g1nmhl0r0000gn/T/ipykernel_54147/2092487613.py:36: UserWarni
  plt.tight_layout()
```



```
Adjusted Rand Index: 0.9611662823017506
```

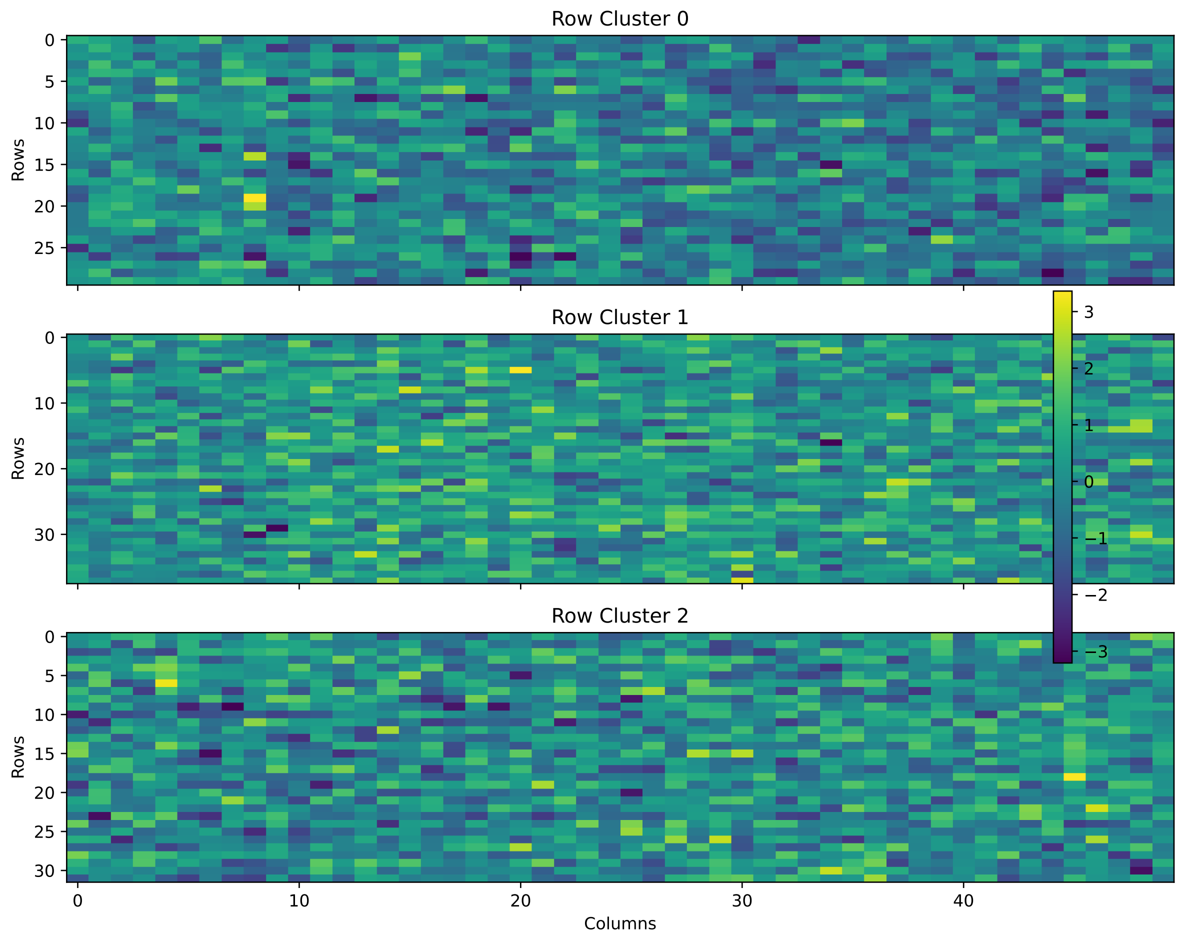
```python
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.cluster import KMeans
import numpy as np

def cocolbm_pytorch(X, K, Q, iter_max=100, lr=0.01):
    """
    PyTorch-based optimization of the cocoLMB model using gradient descent.

    Parameters:
    - X: Input data matrix (N, p)
    - K: Number of row clusters
    - Q: Number of column clusters per row cluster
    - iter_max: Number of optimization iterations
    - lr: Learning rate for the optimizer

    Returns:
    - Dictionary with optimized parameters
    """
    N, p = X.shape

    # Convert X to a PyTorch tensor
    X_tensor = torch.tensor(X, dtype=torch.float32)

    # KMeans initialization for row clusters
    kmeans = KMeans(n_clusters=K, n_init=50).fit(X)
    tau_init = np.eye(K)[kmeans.labels_]

    # KMeans initialization for column clusters
    nu_init = np.zeros((K, p, Q))
    for k in range(K):
        sub_X = X[kmeans.labels_ == k].T
        if sub_X.shape[1] > 0:  # Avoid empty clusters
            km_q = KMeans(n_clusters=Q, n_init=50).fit(sub_X)
            nu_init[k] = np.eye(Q)[km_q.labels_]
        else:
            nu_init[k] = np.random.rand(p, Q)  # Fallback for empty clusters

    # Convert initialized values to PyTorch tensors
    tau = nn.Parameter(torch.tensor(tau_init, dtype=torch.float32))
    nu = nn.Parameter(torch.tensor(nu_init, dtype=torch.float32))
    mu_ks = nn.Parameter(torch.randn((K, Q)))
    sigma2_ks = nn.Parameter(torch.ones((K, Q)))
    pi_k = nn.Parameter(torch.ones((K,)) / K)
    rho_ks = nn.Parameter(torch.ones((K, Q)) / Q)

    # Define optimizer
    optimizer = optim.Adam([tau, nu, mu_ks, sigma2_ks, pi_k, rho_ks], lr=lr)

    # Training loop
    for epoch in range(iter_max):
        optimizer.zero_grad()

        # Softmax constraints
        tau_softmax = torch.softmax(tau, dim=1)
        nu_softmax = torch.softmax(nu, dim=2)

        # Compute ELBO loss
        log_prob = torch.zeros((N, K, p, Q))
        for k in range(K):
            for s in range(Q):
                log_prob[:, k, :, s] = torch.distributions.Normal(
                    mu_ks[k, s], torch.sqrt(sigma2_ks[k, s])
                ).log_prob(X_tensor)

        log_prob_sum = (tau_softmax[:, :, None, None] * log_prob * nu_softmax[None, :, :, :]

        elbo = (
            log_prob_sum
            + (nu_softmax * torch.log(rho_ks[:, None, :] + 1e-8)).sum()
            + (tau_softmax * torch.log(pi_k[None, :] + 1e-8)).sum()
            - (nu_softmax * torch.log(nu_softmax + 1e-8)).sum()
            - (tau_softmax * torch.log(tau_softmax + 1e-8)).sum()
        )

        # Compute negative ELBO loss to minimize
        loss = -elbo
        loss.backward()
        optimizer.step()

        if epoch % 10 == 0:
            print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

    # Return the trained parameters
    return {
        "tau": tau_softmax.detach().numpy(),
        "nu": nu_softmax.detach().numpy(),
        "mu_ks": mu_ks.detach().numpy(),
        "sigma2_ks": sigma2_ks.detach().numpy(),
        "pi_k": pi_k.detach().numpy(),
        "rho_ks": rho_ks.detach().numpy(),
        "elbo": elbo.item(),
    }

# Example usage
N, p, K, Q = 100, 50, 3, 4
```

```
X = np.random.randn(N, p)  # Simulated data
model = cocolbm_pytorch(X, K, Q, iter_max=1, lr=0.1)
```

```
Epoch 0, Loss: 8856.5176
```

```
torch_row_partition_estimated = np.argmax(model['tau'], axis=1)  # Row partition
torch_col_partition_estimated = {}
for k in range(K):
    torch_col_partition_estimated[k] = np.argmax(model['nu'][k], axis=1)

plot_clustered_matrix(X, torch_row_partition_estimated, torch_col_partition_estimated,K)
```

```
/var/folders/c5/vctv2wln4sx9_796g1nmhl0r0000gn/T/ipykernel_54147/2092487613.py:36: UserWarnin
  plt.tight_layout()
```


Row Cluster 0
Row Cluster 1
Row Cluster 2
Rows
Columns