COMP0085: Approximate Inference

January 24, 2025

Contents

Question 1

1.(a)

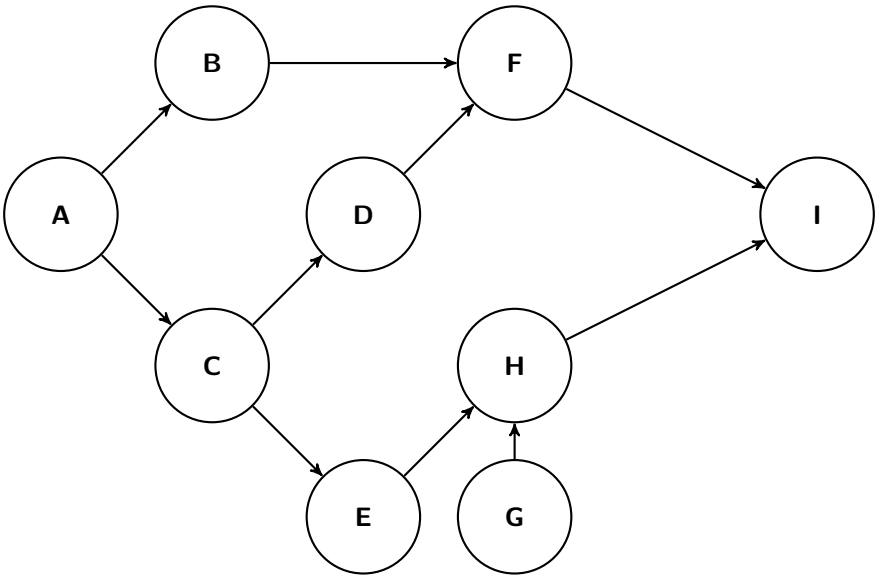


Figure 1: Directed Acyclic Graph (DAG) representing the biochemical cascade, showing the dependencies between molecule concentrations.

1.(b)

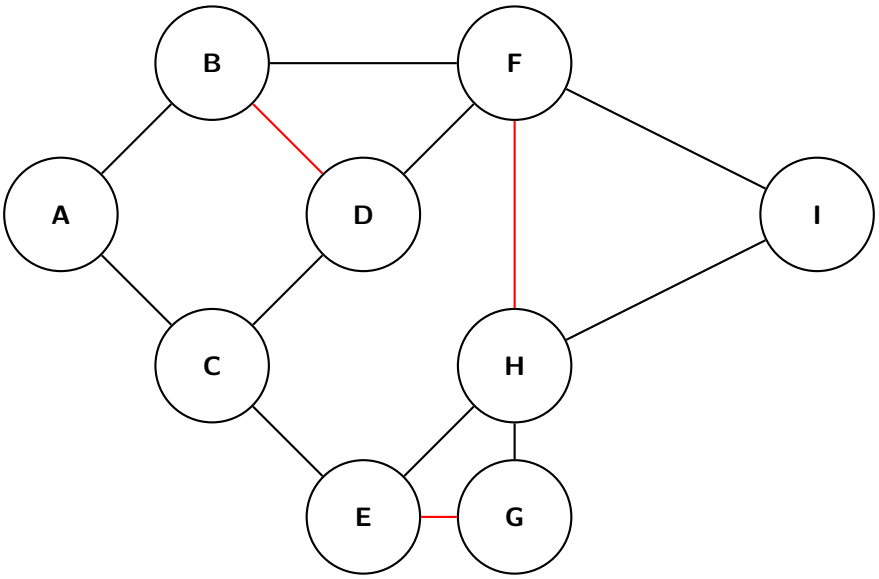


Figure 2: Moralized graph of the biochemical cascade. Red edges represent the added moralized connections between parent nodes.

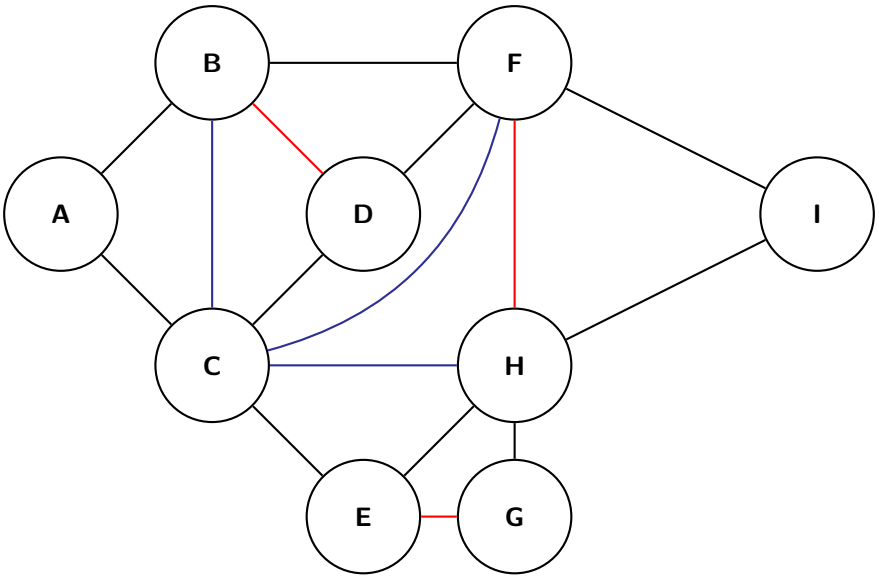


Figure 3: Triangulated graph of the biochemical cascade. Blue edges are added to eliminate cycles of length greater than three.

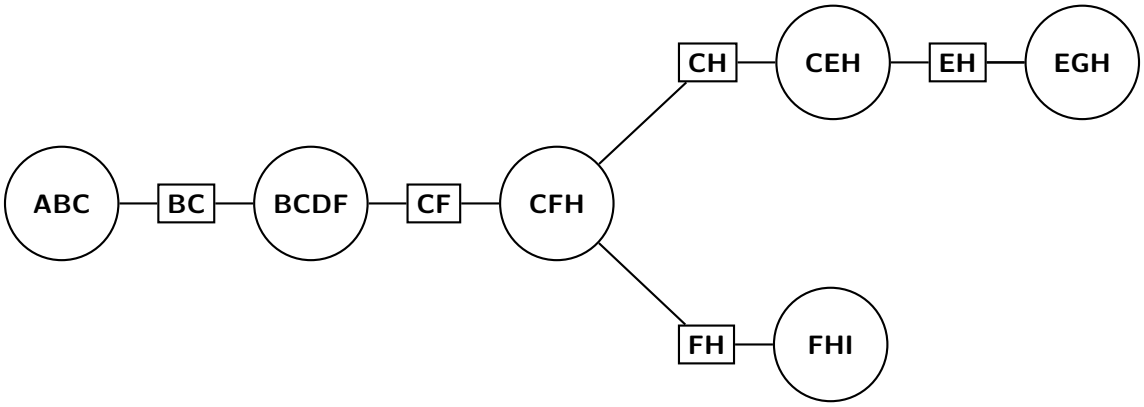


Figure 4: Junction tree derived from the triangulated graph. Nodes represent cliques of the graph, and edges maintain the running intersection property.

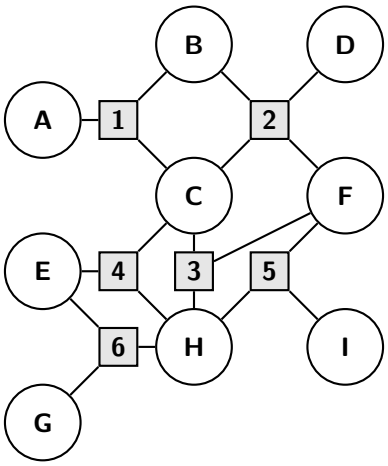


Figure 5: Factor graph representation of the junction tree. Circles represent variables, and rectangles represent factors.

1.(c)

Here is the plot for question (c).

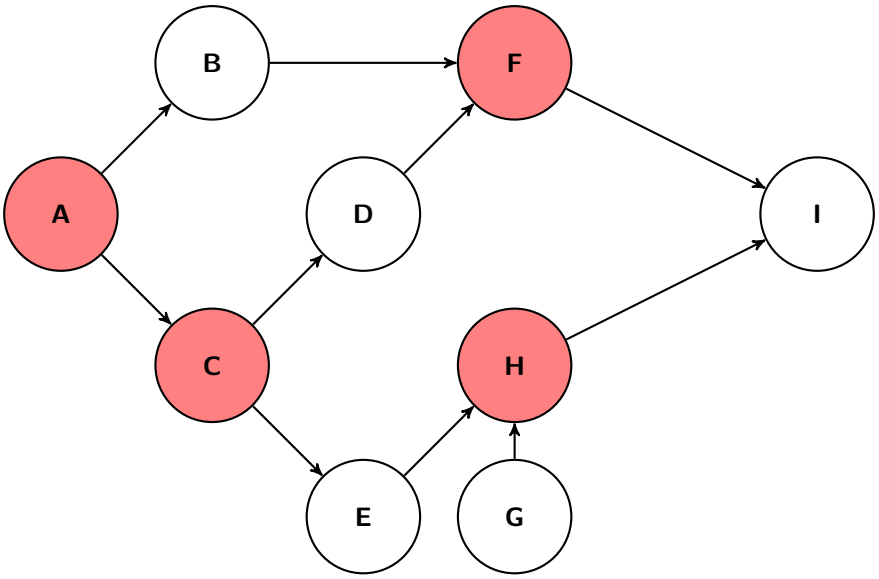


Figure 6: Directed Acyclic Graph (DAG) representing the biochemical cascade, showing the dependencies between molecule concentrations. Nodes A, C, F, H are shaded as the minimal set of molecules that, if measured, render the rest conditionally independent.

1.(d)

We decide to apply factor analysis on these measurements. Following the given model:

$$\begin{aligned} \mathbf{z} &\sim \mathcal{N}(0, I), \\ \mathbf{x} \mid \mathbf{z} &\sim \mathcal{N}(\Lambda \mathbf{z}, \Psi), \end{aligned} \tag{1}$$

we can express the factor of dependence in Λ. The concentration perturbations can be written as:

$$\delta(\cdot) = \Lambda \mathbf{z} + \psi. \tag{2}$$

From the graph in 1(a), we can write the relations as:

$$\mathbf{\Lambda} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \lambda_{BA} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \lambda_{CA} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \lambda_{DC} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \lambda_{EC} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \lambda_{FB} & 0 & \lambda_{FD} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \lambda_{HE} & 0 & \lambda_{HG} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \lambda_{IF} & 0 & \lambda_{IH} & 0 \end{bmatrix}. \tag{3}$$

Here we use λ_{IJ} to show the contribution of J to I. The Λ is a 9 by 9 matrix that indicates the parents and children relation of molecules. However, we only have the informations of δ[B], δ[D], δ[E], and δ[G]. In that case, we have the following equation:

$$\begin{bmatrix} \delta[B] \\ \delta[D] \\ \delta[E] \end{bmatrix} = \begin{bmatrix} \Lambda_{BA} & 0 \\ 0 & \Lambda_{DC} \\ 0 & \Lambda_{EC} \end{bmatrix} \begin{bmatrix} z_A \\ z_C \end{bmatrix} + \begin{bmatrix} \psi_B \\ \psi_D \\ \psi_E \end{bmatrix}, \tag{4}$$

where we leave the relevant elements only. This indicates that we are supposed to recover the latent factors A and C.

1.(e)

The results of the factor analysis cannot be used to recover the concentration perturbations of any other species in the cascade. From the factor loading matrix Λ and the structure of the graph, we observe that it is not possible to determine the perturbations of species such as A and C from the measured variables. More generally, we can only recover the perturbations of nodes that are directly related to the measured parent nodes in our graph.

This limitation also applies to the identifiability of nodes and weights in the graph. Only the measured nodes (B, D, E, G) and their upstream nodes (A, C) are identifiable. The downstream nodes cannot be determined and are only identifiable up to an unknown scale factor.

Question 2

2.(a)

We need to compute the posterior mean and covariance first. We can start with the Bayes Rule:

$$P(\omega \mid X, y) = \frac{P(y \mid X, \omega) \cdot P(\omega)}{P(y \mid X)}, \tag{5}$$

where ω presents the parameter a and b. Knowing that posterior, likelihood and prior should be Gaussian distributions like

$$\begin{aligned} P(y \mid X, \omega) &\propto \mathcal{N}(y \mid \phi^\top \omega, \sigma^2 I), \\ P(\omega) &\propto \mathcal{N}(\omega \mid \mu_0, \Sigma_0), \\ P(\omega \mid X, y) &\propto \mathcal{N}(\omega \mid \mu_{pos}, \Sigma_{pos}), \end{aligned} \tag{6}$$

we can use these assumption to derive the expression of posterior mean and covariance. One can regard the evidence term as a constant which gives a relation

$$P(w \mid X, y) \propto P(y \mid X, w) \cdot P(w). \tag{7}$$

Taking logarithm on both sides, we can have

$$\begin{aligned} \log P(\omega \mid X, y) &\propto \log P(y \mid X, \omega) + \log P(\omega), \\ \log \mathcal{N}(\omega \mid \mu_{pos}, \Sigma_{pos}) &\propto \log \mathcal{N}(y \mid \phi^\top \omega, \sigma^2 I) + \log \mathcal{N}(\omega \mid \mu_0, \Sigma_0). \end{aligned} \tag{8}$$

The right-hand side can be further written as

$$\begin{aligned} &\log \mathcal{N}(y \mid \phi^\top \omega, \sigma^2 I) + \log \mathcal{N}(\omega \mid \mu_0, \Sigma_0), \\ \Rightarrow &-\frac{1}{2}\left(\sigma^{-2}(y - \phi^\top \omega)^\top (y - \phi^\top \omega) + (\omega - \mu_0)^\top \Sigma_0^{-1} (\omega - \mu_0)\right) + const, \\ \Rightarrow &-\frac{1}{2}\left(\sigma^{-2} y^\top y - 2\sigma^{-2} y^\top \phi^\top \omega + \sigma^{-2}\omega^\top \phi\phi^\top \omega + \omega^\top \Sigma_0^{-1}\omega - 2\mu_0^\top \Sigma_0^{-1}\omega + \mu_0^\top \Sigma_0^{-1}\mu_0\right) + const, \\ \Rightarrow &-\frac{1}{2}\left(\omega^\top \left(\sigma^{-2}\phi\phi^\top + \Sigma_0^{-1}\right)\omega - 2\left(\sigma^{-2} y^\top \phi^\top + \mu_0^\top \Sigma_0^{-1}\right)\omega\right) + const. \end{aligned} \tag{9}$$

At the same time, we can expend the left-hand side in the similar layout

$$\begin{aligned} \log \mathcal{N}(\omega \mid \mu_{pos}, \Sigma_{pos}) &= -\frac{1}{2}(\omega - \mu_{pos})^\top \Sigma_{pos}^{-1}(\omega - \mu_{pos}), \\ &= -\frac{1}{2}\left(\omega^\top \Sigma_{pos}^{-1}\omega - 2\mu_{pos}^\top \Sigma_{pos}^{-1}\omega\right) + const. \end{aligned} \tag{10}$$

Combining RHS with LHS, we can find

$$\begin{aligned} \Sigma_{pos}^{-1} &= \sigma^{-2}\phi\phi^\top + \Sigma_0^{-1}, \\ \mu_{pos}^\top \Sigma_{pos}^{-1} &= \sigma^{-2} y^\top \phi^\top + \mu_0^\top \Sigma_0^{-1}, \end{aligned} \tag{11}$$

which can be rearranged as

$$\begin{aligned} \Sigma_{pos} &= \left(\sigma^{-2}\phi\phi^\top + \Sigma_0^{-1}\right)^{-1}, \\ \mu_{pos} &= \Sigma_{pos}\left(\sigma^{-2}\phi y + \Sigma_0^{-1}\mu_0\right). \end{aligned} \tag{12}$$

In our questions, we have

$$\Sigma_0 = \begin{pmatrix} 10^2 & 0 \\ 0 & 100^2 \end{pmatrix}, \quad \mu_0 = \begin{pmatrix} 0 \\ 360 \end{pmatrix}. \tag{13}$$

Computing this in code, we get

```python
# Parameters
sigma = 1
sigma_0 = np.array([[100, 0], [0, 10000]])
mu_0 = np.array([0, 360])

# Calculate the posterior distribution
sigma_pos = np.linalg.inv(np.linalg.inv(sigma_0) + (1 / sigma**2) * X @ X.T)
mu_pos = sigma_pos @ (np.linalg.inv(sigma_0) @ mu_0 + (1 / sigma**2) * X @ Y)

print(sigma_pos)
print(mu_pos)
```

Question 2(a)

The final results are

$$\Sigma_{pos} = \begin{pmatrix} 1.37479698e-05 & -2.75073908e-02 \\ -2.75073908e-02 & 5.50396935e+01 \end{pmatrix}, \quad \mu_{pos} = \begin{pmatrix} 1.81842808e+00 \\ -3.26615096e+03 \end{pmatrix}. \tag{14}$$

We can also show the results in a plot.

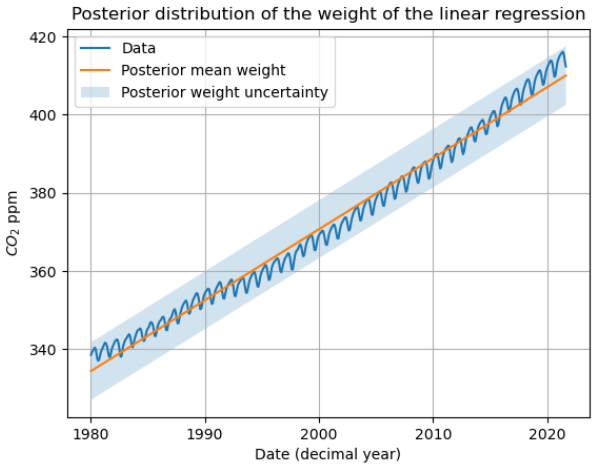
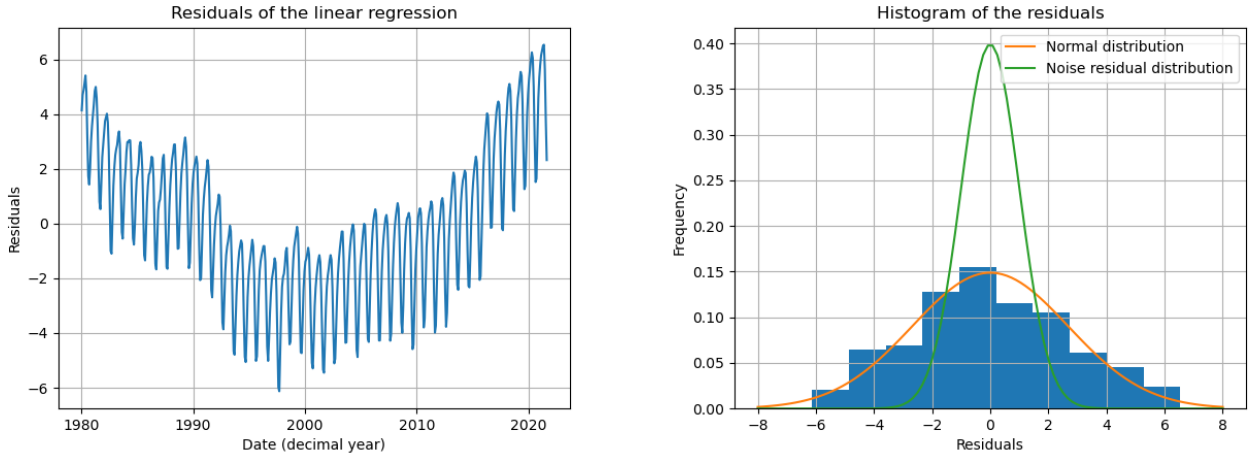


Figure 7: Question 2(a)

2.(b)

We can plot the residue as



(a) Residuals vs time

(b) Histogram of Residuals

Figure 8: Question 2(b)

This shows that the residuals depend on time. In that case, these residuals do not conform to our prior in the characteristics of Independence. We can also plot the residuals in a histogram. We can observe the inconsistency of the real distribution and approximate distribution.

2.(c)

We consider a squared-exponential kernel:

$$k\left(x_i, x_j\right) = \sigma^2 \exp\left(-\frac{\left(x_i - x_j\right)^2}{2\ell^2}\right), \tag{15}$$

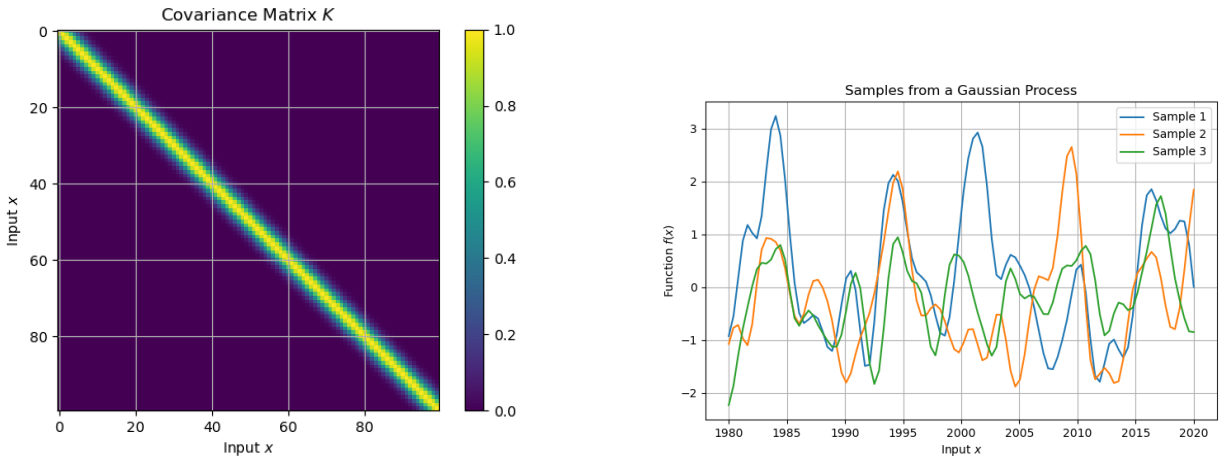
where ℓ is the lengthscale and σ^2 is the variance.

Then, we can draw the sample from following code.

```python
def gp_sample(kernel, x, n_samples=1):
    # Compute the covariance matrix K
    n = len(x)
    K = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            K[i, j] = kernel(x[i], x[j])

    plt.imshow(K, interpolation='nearest')
    plt.colorbar()
    plt.title('Covariance Matrix $K$')
    plt.xlabel('Input $x$')
    plt.ylabel('Input $x$')
    plt.grid()
    plt.show()
    # Add a small noise for numerical stability
    K += 1e-8 * np.eye(n)

    # Draw samples from the multivariate normal distribution
    mean = np.zeros(n)
    samples = np.random.multivariate_normal(mean, K, n_samples)

    return samples

# define an squared exponential kernel
def squared_exponetial_kernel(x1, x2, lengthscale=1.0, variance=1.0):
    return variance * np.exp(-0.5 * (x1 - x2)**2 / lengthscale**2)
```

Question 2(c)

The plots of covariance matrix and samplings are shown as



(a) Covariance Matrix

(b) GP samplings

Figure 9: Question 2(b)

2.(d)

Now, we use the given kernel

$$k(s,t) = \theta^2 \left(\exp\left(-\frac{2\sin^2(\pi(s-t)/\tau)}{\sigma^2} \right) + \phi^2 \exp\left(-\frac{(s-t)^2}{2\eta^2} \right) \right) + \zeta^2 \delta_{s=t}, \tag{16}$$

and test its parameters. By changing the specific parameters, we can find the trend and its characteristics.

Parameter θ

Here is the plot for variation of θ:

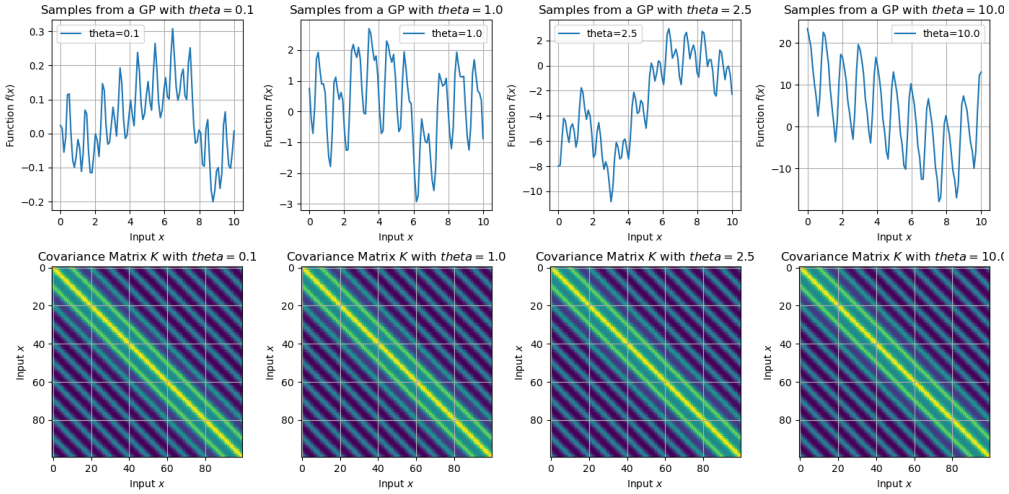


Figure 10: Plots of samples and covariance matrices for theta

As θ increases, the amplitude of the GP samples increases significantly. For smaller θ (e.g., $\theta = 0.1$), the fluctuations are small in magnitude, and the samples appear constrained around zero. For larger θ, such as $\theta = 10$, the fluctuations become much more pronounced, with a broader range of values. The covariance matrix K reflects this scaling, with larger values throughout the matrix as θ increases, while the structure remains unchanged.

Parameter σ

Here is the plot for variation of σ:

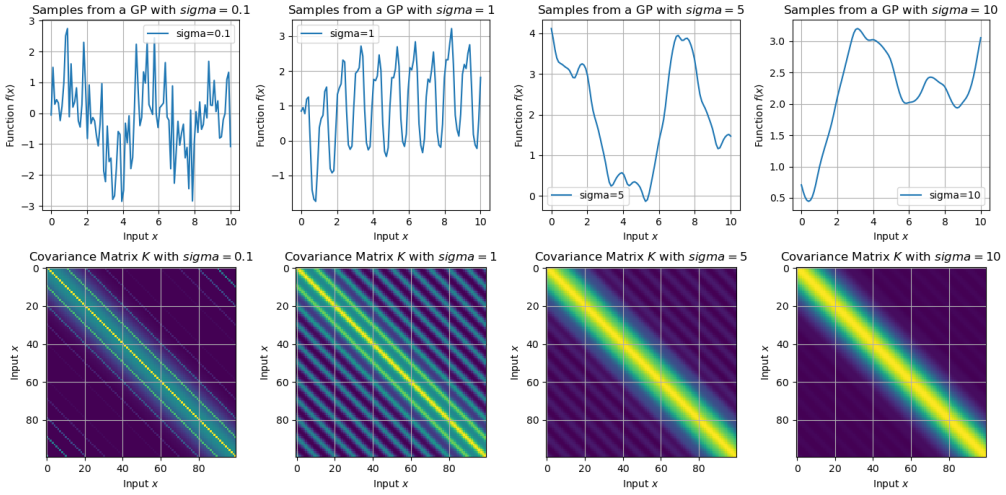


Figure 11: Plots of samples and covariance matrices for sigma

The plots show that smaller σ values result in rapidly fluctuating GP samples due to weak correlations between neighbouring points, as seen in the covariance matrices with sharp diagonal dominance. In contrast, larger σ values produce smoother samples, with covariance matrices displaying more uniform correlations across the input space. This illustrates how σ controls the smoothness and correlation scale of the GP.

Parameter τ

Here is the plot for variation of τ:

The parameter τ controls the periodicity of the GP samples. For smaller τ (e.g., $\tau = 0.1$), the samples show high-frequency oscillations with rapid periodic patterns. As τ increases, the periodicity becomes broader, leading to smoother oscillations with larger wavelengths. The covariance matrix K reflects this behaviour, with the off-diagonal bands extending further as τ increases, indicating greater correlation between points separated by larger distances.

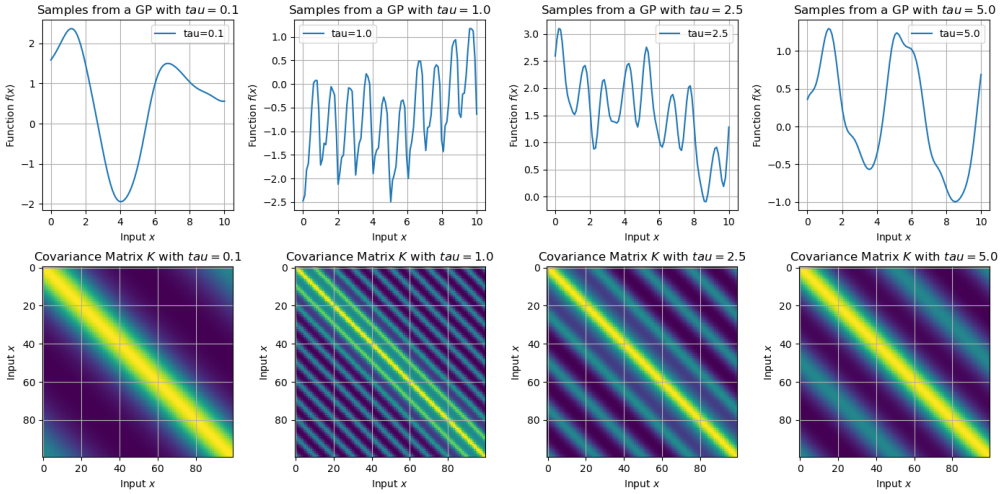


Figure 12: Plots of samples and covariance matrices for tau

Parameter ϕ

Here is the plot for variation of ϕ:

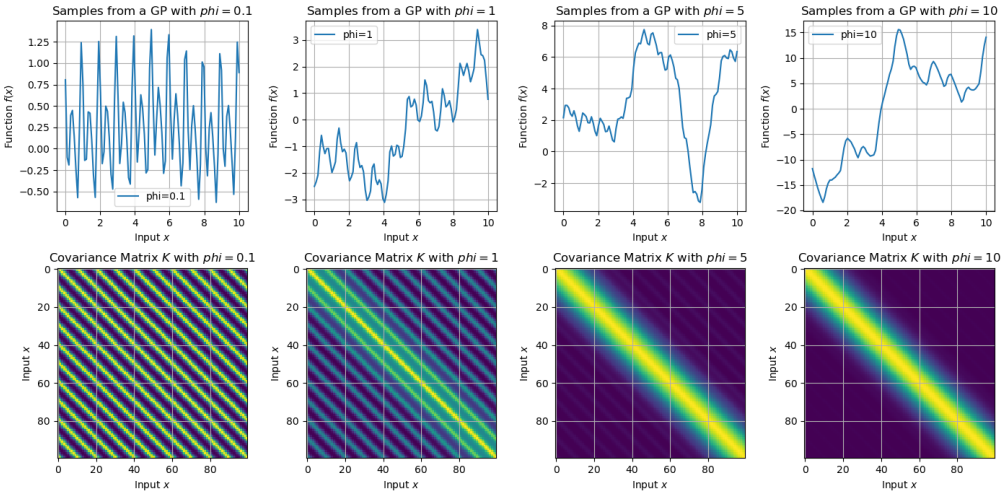


Figure 13: Plots of samples and covariance matrices for phi

As ϕ increases, the amplitude of the smoothness component in the GP samples increases. For smaller values of ϕ (e.g., $\phi = 0.1$), the samples appear primarily dominated by other components of the kernel i.e the periodic component, and the smoothness effect of the squared-exponential kernel is less pronounced. As ϕ increases, the samples become smoother and exhibit larger variations in amplitude, reflecting the increased contribution of the squared-exponential kernel.

The covariance matrix K also reflects this behaviour. For small ϕ, the values in K are lower, indicating weaker correlations between points. As ϕ increases, the diagonal and off-diagonal values of K grow, showing stronger correlations across the input space due to the amplified effect of the squared-exponential kernel.

Parameter η

Here is the plot for variation of η:

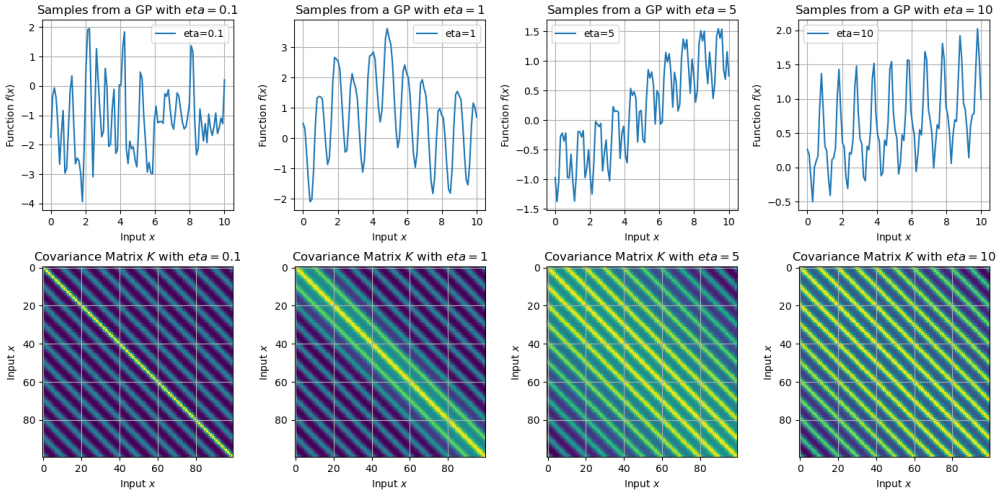


Figure 14: Plots of samples and covariance matrices for eta

As η increases, the GP samples become smoother. For smaller η (e.g., $\eta = 0.1$), the samples show rapid variations with sharp transitions between points, as neighbouring points have weaker correlations. As η increases, the variations become more gradual, and the samples become smoother. From the expression of the kernel, η represents the lengthscale of the squared-exponential component. The covariance matrix K reflects this by having wider off-diagonal correlations as η increases, indicating stronger correlations between points separated by greater distances.

Parameter ζ

Here is the plot for variation of ζ:

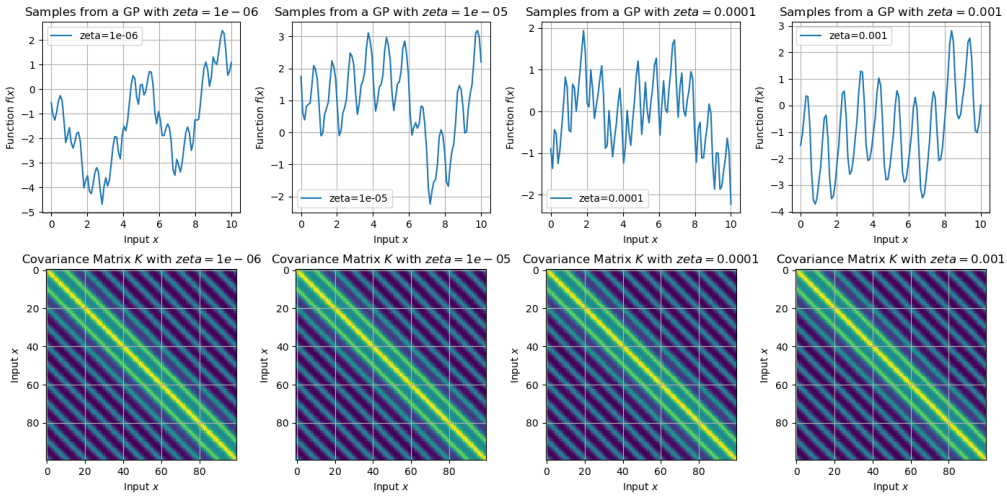


Figure 15: Plots of samples and covariance matrices for zeta

As ζ increases, the GP samples exhibit increasing noise in their structure. For very small ζ (e.g., $\zeta = 10^{-6}$), the samples are smooth and free from visible noise. As ζ grows, the samples start showing jagged variations due to the added noise. The covariance matrix reflects this by having larger diagonal elements, which dominate the off-diagonal elements and emphasize the independent noise contribution at each input point.

Code for testing parameters

Here is th code for kernel function and GP samples.

```python
def custom_kernel(s, t, theta=1.0, tau=1.0, sigma=1.0, phi=1.0, eta=1.0, zeta=1e-6):
    periodic_term = np.exp(-2 * np.sin(np.pi * (s - t) / tau)**2 / sigma**2)
    squared_exp_term = phi**2 * np.exp(-(s - t)**2 / (2 * eta**2))
    noise_term = zeta**2 if s == t else 0
    return theta**2 * (periodic_term + squared_exp_term) + noise_term

def gp_sample_custom_kernel(x, kernel, n_samples=1, **kernel_params):
    n = len(x)
    K = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            K[i, j] = kernel(x[i], x[j], **kernel_params)

    # Add a small jitter for numerical stability
    #K += 1e-8 * np.eye(n)
    np.random.seed(0)
    # Draw samples from the multivariate normal distribution
    mean = np.zeros(n)
    samples = np.random.multivariate_normal(mean, K, n_samples)

    return samples, K
```

Question 2(d) Kernel function

Here is the code used to test our samples with different parameters.

```python
# Input points
x = np.linspace(0, 10, 100)

# Define the hyperparameter values
sigma_values = [0.1, 1, 5, 10]
phi_values = [0.01, 1, 5, 10]
eta_values = [0.1, 1, 5, 10]
tau_values = [0.1, 1.0, 2.5, 5.0]
theta_values = [0.1, 1.0, 2.5, 10.0]
zeta_values = [1e-06, 1e-05, 0.0001, 0.001]

# Function to plot samples and covariance matrix for given hyperparameter values
def plot_gp_samples_and_covariance(x, param_values, param_name):
    fig, axs = plt.subplots(2, len(param_values), figsize=(14, 7))
    for i, param in enumerate(param_values):
        samples, K = gp_sample_custom_kernel(x, custom_kernel, n_samples=1, **{param_name: param})

        # Plot the samples
        axs[0, i].plot(x, samples[0], label=f'{param_name}={param}')
        axs[0, i].set_xlabel('Input $x$')
        axs[0, i].set_ylabel('Function $f(x)$')
        axs[0, i].set_title(f'Samples from a GP with ${param_name}={param}$')
        axs[0, i].legend()
        axs[0, i].grid()

        # Plot the covariance matrix
        im = axs[1, i].imshow(K, interpolation='nearest', cmap='viridis')
        axs[1, i].set_title(f'Covariance Matrix $K$ with ${param_name}={param}$')
        axs[1, i].set_xlabel('Input $x$')
        axs[1, i].set_ylabel('Input $x$')
        axs[1, i].grid()

    plt.tight_layout()
    plt.savefig(f"gp_samples_and_covariance_matrices_{param_name}.png")
    plt.show()
```

Question 2(d) test hyperparameters

2.(e)

As we discuss how the parameters influence the plots in (d), we can adjust our kernel function via comparing with the residual plot. By carefully tuning the parameters, we get following combinations

Parameter	θ	τ	σ	ϕ	η	ζ
Value	1.5	1.0	1.0	0.9	9.0	0.5

Table 1: Suitable values for the hyperparameters

Using these parameters, we can plot the samplings and compare this with the residuals.

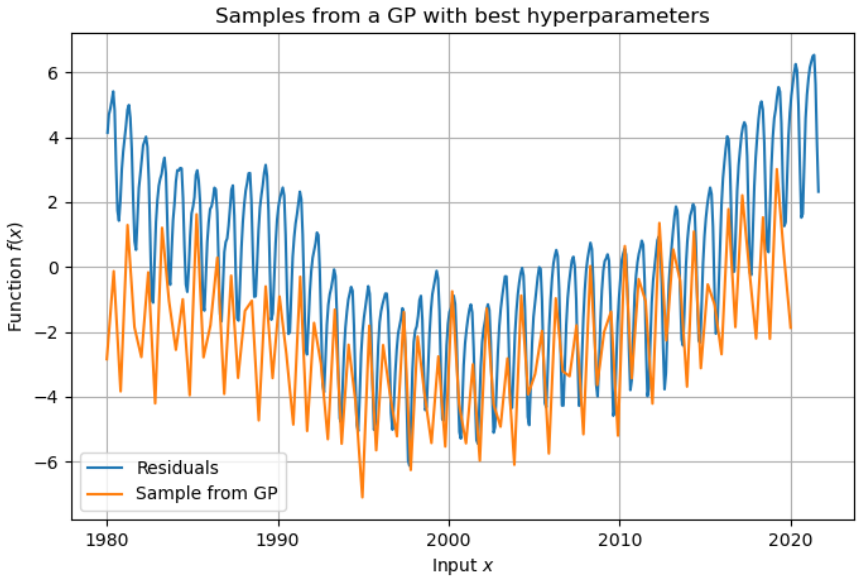
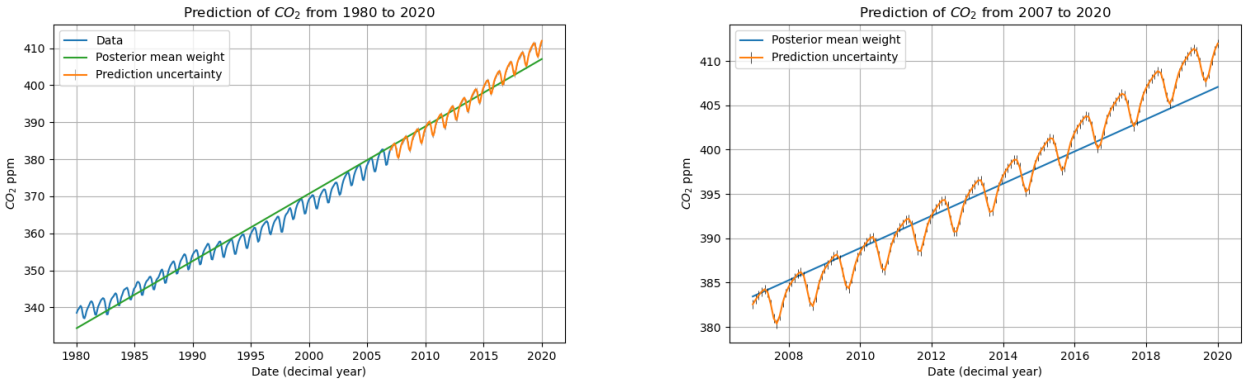


Figure 16: GP samples and Residuals

2.(f)

We extrapolate the CO_2 concentration levels to 2020 using the GP with covariance kernel given in (d). Here are the plots for extrapolation.



(a) Prediction from 1980 to 2020

(b) Prediction only

Figure 17: Question 2(f)

The extrapolation aligns with our expectations, as it continues the trend observed in the original data. It shows an increase in CO_2 concentration with periodic oscillations. However, the predictions are highly sensitive to the

settings of the kernel hyperparameters. Even small changes in these parameters can significantly alter the trends, highlighting the importance of carefully tuning the hyperparameters to ensure reliable extrapolations.

Here is the code.

```
# question 2 (f)
# here we define the kernel function again with the parameters found in the previous question
def kernel_function(s, t):
    # vectorised Kernel function
    s = s.reshape(-1, 1)
    t = t.reshape(1, -1)

    theta = 1.5
    tau = 1
    sigma = 1
    phi = 0.9
    eta = 9
    zeta = 0.5

    periodic_term = np.exp(-2 * np.sin(np.pi * (s - t) / tau)**2 / sigma**2)
    squared_exp_term = phi**2 * np.exp(-(s - t)**2 / (2 * eta**2))
    noise_term = zeta**2 * (s == t)
    return theta**2 * (periodic_term + squared_exp_term) + noise_term

# we calculate the mean and covariance of the conditioned GP
x = co2[:, 2]
K = kernel_function(x, x)

# predict c02 concentration to 2020
begin = 2007
end = 2020
domain = np.linspace(begin, end, 12*(end-begin))

# calculate the mean and covariance of the conditioned GP
K_s = kernel_function(domain, x)
K_ss = kernel_function(domain, domain)
K_inv = np.linalg.inv(K)
mu_s = K_s @ K_inv @ residuals
cov_s = K_ss - K_s @ K_inv @ K_s.T

# sample from the conditioned GP - predicted residuals
samples = np.random.multivariate_normal(mu_s, cov_s, 1)

# f(t) =  a * t + b + g(t)
features = np.concatenate((domain[:, None], np.ones((len(domain)))[:, None]), axis=1)
predictions = samples + (features @ mu_pos)[:, None]
predictions_mean = mu_s + (features @ mu_pos)

z = np.linspace(min(x), end, 1000)
# plot the prediction from 1980 to 2020
plt.figure(figsize=(8, 5))
plt.plot(time[time < 2007], labels[time < 2007], label="Data")
plt.errorbar(domain, predictions_mean, yerr=np.sqrt(np.diag(cov_s)), elinewidth=0.7, ecolor="
    darkgrey", label="Prediction uncertainty")
plt.plot(z, z * mu_pos[0] + mu_pos[1], label="Posterior mean weight")
plt.xlabel("Date (decimal year)")
plt.ylabel("$CO_2$ ppm")
plt.title("Prediction of $CO_2$ from 1980 to 2020")
plt.grid()
plt.legend()
plt.savefig("prediction_1980_2020.png")
plt.show()

# plot the prediction from 2007 to 2020
z = np.linspace(begin, end, 1000)
plt.figure(figsize=(8, 5))
plt.plot(z, z * mu_pos[0] + mu_pos[1], label="Posterior mean weight")
plt.errorbar(domain, predictions_mean, yerr=np.sqrt(np.diag(cov_s)), elinewidth=0.5, ecolor="black"
    , label="Prediction uncertainty")
plt.xlabel("Date (decimal year)")
plt.ylabel("$CO_2$ ppm")
plt.title("Prediction of $CO_2$ from 2007 to 2020")
plt.grid()
plt.legend()
plt.savefig("prediction_2007_2020.png")
```

```
plt.show()
```

Question 2(f) extrapolation

2.(g)

The above procedure is not fully Bayesian because it uses point estimates, such as the MAP estimates of the linear parameters a and b, instead of integrating over their posterior distributions. In a fully Bayesian approach, the uncertainty in a and b would be incorporated by marginalizing over their posterior distributions.

To model $f(t)$ in a Bayesian framework, we would:

- Compute the posterior distribution of the linear parameters a and b, given the data, such that:

$$p(a, b \mid \text{data}) \sim \mathcal{N}(\boldsymbol{\mu}_{\mathbf{w}}, \boldsymbol{\Sigma}_{\mathbf{w}}),$$

  where $\boldsymbol{\mu}_{\mathbf{w}}$ and $\boldsymbol{\Sigma}_{\mathbf{w}}$ are the mean and covariance of the posterior distribution.

- Sample a and b from their posterior distributions and combine these with the GP posterior for $g(t)$ to compute:

$$f(t) = at + b + g(t).$$

- Integrate over both the linear parameters and the GP predictions to obtain the full posterior predictive distribution of $f(t)$, accounting for all sources of uncertainty.

This fully Bayesian approach ensures that the predictions incorporate uncertainties from both the linear parameters and the GP, leading to a more robust and accurate model.

Question 3

3.(a)

To find the $\boldsymbol{\lambda}^{(n)}$'s that maximize $\mathcal{F}_n$ holding $\boldsymbol{\theta}$ fixed, we start with the general expression of free energy $\mathcal{F}$:

$$\begin{aligned}
&\mathcal{F}(q,\theta) = \langle \log P(\mathcal{X}, s \mid \theta)\rangle_{q(s)} + \mathbf{H}[q], \\
\Rightarrow &\mathcal{F}(q(\mathbf{s}),\theta) = \sum_{n=1}^{N} \langle \log P(x^{(n)}, s^{(n)} \mid \theta, \pi)\rangle_{q(s^{(n)})} + \mathbf{H}[q(s^{(n)})], \\
\Rightarrow &\mathcal{F}(q(\mathbf{s}),\theta) = \sum_{n=1}^{N} \langle \log P(x^{(n)} \mid s^{(n)}, \theta)\rangle_{q(s^{(n)})} + \langle \log P(s^{(n)} \mid \pi)\rangle_{q(s^{(n)})} - \langle \log q(s^{(n)})\rangle_{q(s^{(n)})}.
\end{aligned} \tag{17}$$

In variational E-step, we maximise the free energy with respect to the distribution over latents given parameters:

$$q^{(n)}(s) := \underset{q(s)\in\mathcal{Q}}{\operatorname{argmax}} \mathcal{F}\left(q(s),\theta\right). \tag{18}$$

To find this maxmizer, we take the variational derivative of the Lagrangian:

$$\begin{aligned}
&\frac{\delta}{\delta q_i}\left(\mathcal{F} + \lambda\left(\int q_i - 1\right)\right) = 0, \\
\Rightarrow &\langle \log P(\mathcal{X}, \mathbf{s} \mid \theta)\rangle_{\prod_{j\neq i} q_j(s_j)} - \log q_i(s_i) + \lambda = 0, \\
\Rightarrow &q_i(s_i) \propto \exp\left(\langle \log P(\mathcal{X}, \mathbf{s} \mid \theta)\rangle_{\prod_{j\neq i} q_j(s_j)}\right), \\
\Rightarrow &q_i^n(s_i^n) \propto \exp\left(\langle \log P(x^{(n)}, s^{(n)} \mid \theta)\rangle_{\prod_{j\neq i, m\neq n} q_j^m(s_j^m)}\right), \\
\Rightarrow &q_i^n(s_i^n) \propto \exp\left(\langle \log P(x^{(n)} \mid s^{(n)}, \theta)\rangle_{\prod_{j\neq i, m\neq n} q_j^m(s_j^m)} + \langle \log P(s^{(n)} \mid \pi)\rangle_{\prod_{j\neq i, m\neq n} q_j^m(s_j^m)}\right).
\end{aligned} \tag{19}$$

We can evaluate two terms in above equation separately. The first term can be further written as:

$$\begin{aligned}
&\left\langle \log P\left(x^{(n)} \mid s^{(n)}, \boldsymbol{\mu}, \sigma^2\right)\right\rangle \propto -\frac{1}{2\sigma^2}\left\langle \left\| x^{(n)} - \sum_{i=1}^{K} s_i^{(n)} \boldsymbol{\mu}_i \right\|^2 \right\rangle, \\
\Rightarrow &\left\langle \log P\left(x^{(n)} \mid s^{(n)}, \boldsymbol{\mu}, \sigma^2\right)\right\rangle \propto -\frac{1}{2\sigma^2}\left(\left\| x^{(n)} \right\|^2 - 2\sum_{i=1}^{K} \lambda_{in} \boldsymbol{\mu}_i^\top x^{(n)} + \sum_{i=1}^{K} \lambda_{in} \left\| \boldsymbol{\mu}_i \right\|^2 + \sum_{i\neq j}^{K}\sum_{j=1}^{K} \lambda_{in}\lambda_{jn} \boldsymbol{\mu}_i^\top \boldsymbol{\mu}_j. \right), \\
\Rightarrow &\left\langle \log P\left(x^{(n)} \mid s^{(n)}, \boldsymbol{\mu}, \sigma^2\right)\right\rangle \propto -\frac{1}{2\sigma^2}\left(-2\mu_i^\top x^{(n)} + \mu_i^\top \mu_i + 2\left(\sum_{j=1}^{k} \lambda_{jn}\mu_i^\top \mu_j - \lambda_{in}\mu_i^\top \mu_i\right)\right),
\end{aligned} \tag{20}$$

The second term can be expressed as:

$$\begin{aligned}
&\left\langle \log P\left(s^{(n)} \mid \boldsymbol{\pi}\right)\right\rangle = s_i^{(n)} \log \pi_i + \left(1 - s_i^{(n)}\right) \log\left(1 - \pi_i\right), \\
\Rightarrow &\left\langle \log P\left(s^{(n)} \mid \boldsymbol{\pi}\right)\right\rangle \propto s_i^{(n)} \log \frac{\pi_i}{1-\pi_i}.
\end{aligned} \tag{21}$$

Combining with the expression of $q(s)$:

$$q_n\left(\mathbf{s}^{(n)}\right) = \prod_{i=1}^{K} \lambda_{in}^{s_i^{(n)}} \left(1 - \lambda_{in}\right)^{\left(1 - s_i^{(n)}\right)}, \tag{22}$$

we have:

$$\begin{aligned}
&\log q_i^n(s_i^n) \propto s_i^{(n)} \log \frac{\lambda_{in}}{1-\lambda_{in}}, \\
\Rightarrow &\log \frac{\lambda_{in}}{1-\lambda_{in}} \propto \log \frac{\pi_j}{1-\pi_j} - \frac{1}{2\sigma^2}\left(-2\mu_i^\top x^{(n)} + \mu_i^\top \mu_i + 2\left(\sum_{j=1}^{k} \lambda_{jn}\mu_i^\top \mu_j - \lambda_{in}\mu_i^\top \mu_i\right)\right), \\
\Rightarrow &\lambda_{in} = \text{sigmod}\left(\log \frac{\pi_j}{1-\pi_j} - \frac{1}{2\sigma^2}\left(-2\mu_i^\top x^{(n)} + \mu_i^\top \mu_i + 2\left(\sum_{j=1}^{k} \lambda_{jn}\mu_i^\top \mu_j - \lambda_{in}\mu_i^\top \mu_i\right)\right)\right).
\end{aligned} \tag{23}$$

To finish our code, we should write a detailed expression of free energy $\mathcal{F}$ and we can continue with Eq(17):

$$\begin{aligned}\mathcal{F} = &-\frac{ND}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{n=1}^{N}\left(\left\|x^{(n)}\right\|^2 - 2\sum_{i=1}^{K}\lambda_{in}\boldsymbol{\mu}_i^\top x^{(n)} + \sum_{i=1}^{K}\lambda_{in}\left\|\boldsymbol{\mu}_i\right\|^2 + \sum_{i\neq j}\lambda_{in}\lambda_{jn}\boldsymbol{\mu}_i^\top\boldsymbol{\mu}_j\right) \\ &+ \sum_{n=1}^{N}\sum_{i=1}^{K}\left[\lambda_{in}\log\pi_i + (1-\lambda_{in})\log\left(1-\pi_i\right)\right] - \sum_{n=1}^{N}\sum_{i=1}^{K}\left[\lambda_{in}\log\lambda_{in} + (1-\lambda_{in})\log\left(1-\lambda_{in}\right)\right].\end{aligned} \tag{24}$$

Here is the code for VE step. Instead of definition a long function, we split our idea into several different functions.

```python
def Find_ESS(lambda0):
    # find the ES=E[z] and ESS=E[zz^T]
    N, K = lambda0.shape
    ES = lambda0
    ESS = lambda0.T @ lambda0
    ESS = ESS - np.diag(np.diag(ESS)) + np.diag(np.sum(ES, axis=0))

    return ES, ESS

def Find_lambda(X, mu, sigma, pie, lambda0):
    N, D = X.shape
    _, K = lambda0.shape
    lambda0 = lambda0.copy()
    for i in range(K):
        z = (
            np.log(pie[:, i] / (1 - pie[:, i]))  # Scalar
            - (1 / sigma**2) * (lambda0 @ mu.T  - X) @ mu[:, i] )
        lambda0[:, i] = 1 / (1 + np.exp(-z))  # Apply sigmoid
    return lambda0

def Find_lambda(X, mu, sigma, pie, lambda0):
    N_, K = lambda0.shape
    lambda_new = lambda0
    diag_mu = np.diag(mu.T@mu).flatten()
    for k in range(K):
        x = (np.log(pie[:, k]/(1-pie[:, k]))
                 + 1 / (sigma**2)
                 * ((X-lambda_new@mu.T)@mu[:, k]
                +lambda_new[:, k]*diag_mu[k]
                - 0.5*diag_mu[k]
                 )
                 )
        lambda_new[:, k] = 1/(1+np.exp(-x))
    return lambda_new

def Find_Free_Energy(X, mu, sigma, pie, lambda0):
    # Compute the free energy
    N, D = X.shape # Number of samples and dimensionality
    _, K = lambda0.shape # Number of clusters

    # Regularize lambda0 to avoid numerical issues
    epsilon2 = 1e-12
    lambda0[lambda0 >= 1] = 1 - epsilon2
    lambda0[lambda0 <= 0] = epsilon2

    # Compute ES and ESS
    ES, ESS = Find_ESS(lambda0)

    # Free energy calculation
    F = - 0.5 * N * D * np.log(2 * np.pi * sigma**2)  # Constant term
    F -= 0.5 / sigma**2 * (
        np.trace(X.T @ X) +
        np.trace(mu.T @ mu @ ESS) -
        2 * np.trace(ES.T @ X @ mu)
    )  # Quadratic term
    F += np.sum(lambda0 * np.log(pie/lambda0) + (1 - lambda0) * np.log((1 - pie)/(1 - lambda0)))
      # Prior and entropy term

    return F

def MeanField(X, mu, sigma, pie, lambda0, maxsteps):
    # track the free energy
    Fs = []
```

```
    lambda_old = lambda0
    # early stopping
    eplison = 1e-10
    for i in range(maxsteps):
        lambda_new = Find_lambda(X, mu, sigma, pie, lambda_old)
        F = Find_Free_Energy(X, mu, sigma, pie, lambda_new)
        Fs.append(F)
        if i > 10 and np.abs(Fs[-1] - Fs[-2]) < eplison:
            break
        lambda_old = lambda_new
    return lambda_old, F
```

Question 3(a) MeanField Function

3.(b)

We can start with an example $P(\boldsymbol{y} \mid \boldsymbol{x}, W, \Sigma_y)$ where y is a linear function of x with Gaussian noise. The ML estimate W_{ML} is:

$$\widehat{\mathrm{W}} = \sum_i \mathbf{y}_i \mathbf{x}_i^\top \left(\sum_i \mathbf{x}_i \mathbf{x}_i^\top \right)^{-1}, \tag{25}$$

where we have the similar expression for M-step. Specifically, we have the same process to find the solution. In regression case, we take the derivative of log-likelihood wrt W. In M-step, we take the derivative of free energy wrt μ. These give a similar form of solution:

$$\boldsymbol{\mu}_j = \sum_i \left[\sum_{n=1}^N \left\langle \mathbf{s}^{(n)} \mathbf{s}^{(n)\top} \right\rangle_{q(\mathbf{s}^{(n)})} \right]_{ji}^{-1} \sum_{n=1}^N \left\langle s_i^{(n)} \right\rangle_{q(\mathbf{s}^{(n)})} \mathbf{x}^{(n)}. \tag{26}$$

This connection arises because the Gaussian likelihood in this model assumes a linear relationship between the latent factors s and the observed data X, similar to the design matrix in linear regression.

3.(c)

To determine the computational complexity of the given M-step implementation, we should go through parameter-update steps.

1. For μ, we have:

$$\boldsymbol{\mu}_j = \sum_i \left[\sum_{n=1}^N \left\langle \mathbf{s}^{(n)} \mathbf{s}^{(n)\top} \right\rangle_{q(\mathbf{s}^{(n)})} \right]_{ji}^{-1} \sum_{n=1}^N \left\langle s_i^{(n)} \right\rangle_{q(\mathbf{s}^{(n)})} \mathbf{x}^{(n)}. \tag{27}$$

   The complexity of inversing a $K \times K$ matrix is $\mathcal{O}(K^3)$. At the same time, we have two matrix multiplications with complexity $\mathcal{O}(K^2 N + KND)$.

2. For σ, we have:

$$\begin{aligned} \sigma^2 = \frac{1}{ND} \Bigg[& \sum_{n=1}^N \mathbf{x}^{(n)\top} \mathbf{x}^{(n)} + \sum_{i,j} \boldsymbol{\mu}_i^\top \boldsymbol{\mu}_j \sum_{n=1}^N \left\langle s_i^{(n)} s_j^{(n)} \right\rangle_{q(\mathbf{s}^{(n)})} \\ & -2 \sum_i \boldsymbol{\mu}_i^\top \sum_{n=1}^N \left\langle s_i^{(n)} \right\rangle_{q(\mathbf{s}^{(n)})} \mathbf{x}^{(n)} \Bigg], \end{aligned} \tag{28}$$

   which gives five multiplications. The first term $\mathbf{x}^{(n)\top} \mathbf{x}^{(n)}$ gives $\mathcal{O}(D^2 N)$. The second term gives $\mathcal{O}(K^2 D + K^3)$ and last term gives $\mathcal{O}(KND + K^2 D)$

3. For π, we have:

$$\boldsymbol{\pi} = \frac{1}{N} \sum_{n=1}^N \left\langle \mathbf{s}^{(n)} \right\rangle_{q(\mathbf{s}^{(n)})}, \tag{29}$$

   which gives $\mathcal{O}(NK)$.

The overall complexity for M-step is:

$$\mathcal{O}(K^3 + K^2 N + KND + D^2 N + K^2 D). \tag{30}$$

3.(d)

Here we find 8 possible features.

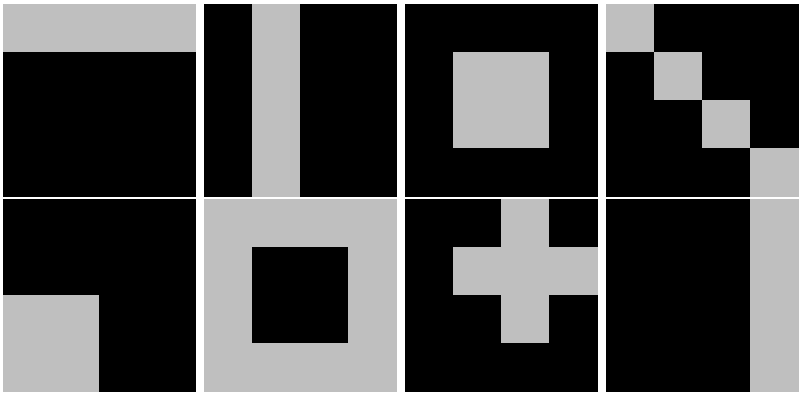


Figure 18: Possible features

How to model these data?

- Factor Analysis: It is not a good idea as our data is binary(discrete) rather than continuos data. To us factor analysis, we need the distribution to be Gaussian.
- Mixture of Gaussian: Again, it is not good idea as our data is discrete. Gaussian assumptions are not aligned with the data distribution. Also, MOG models clusters rather than independent features.
- ICA: It is better than other two methods. Based on the distribution of data, ICA explicitly extracts independent components.

3.(e)

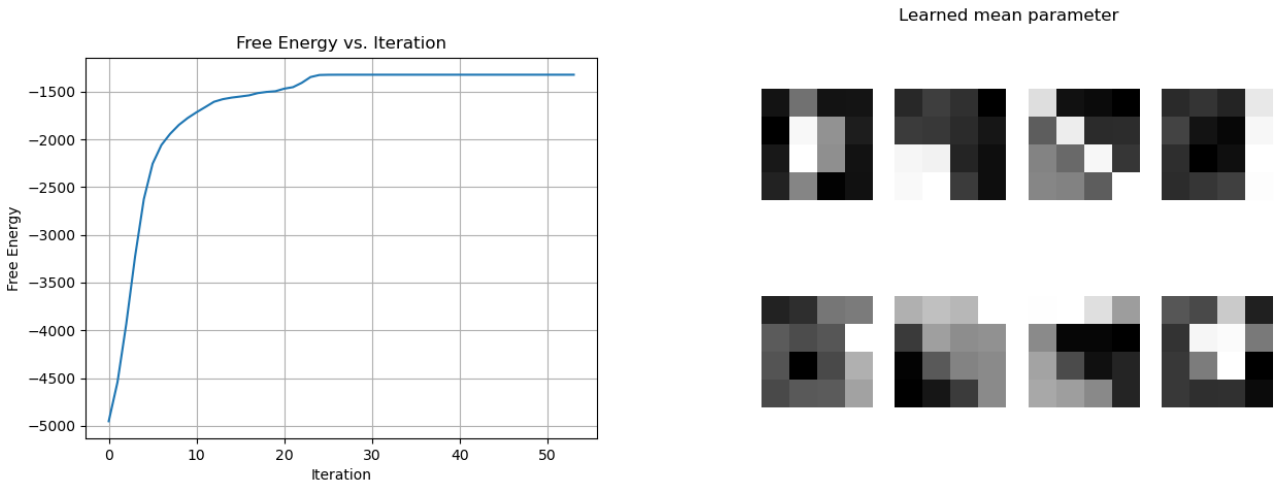
```
def LearnBinFactors(X, K, iterations):
    # LearnBinFactors learns the parameters of a binary factor analysis model
    N, D = X.shape
    random_seed = 42

    # Initialize parameters
    lambda_ = np.random.rand(N, K)  # Shape (N, K)
    ES, ESS = Find_ESS(lambda_)
    # Generate values for mu, sigma, pie
    mu, sigma, pie = m_step(X, ES, ESS)
    # Keep track of free energy
    F_list = []

    for it in range(iterations):
        print(f"Iteration {it+1}:")

        # E-step: Update lambda using MeanField
        lambda_, F = MeanField(X, mu, sigma, pie, lambda_, maxsteps=300)

        # M-step: Update parameters
        ES, ESS = Find_ESS(lambda_)
        mu, sigma, pie = m_step(X, ES, ESS)

        # Keep track of free energy
        F_list.append(F)
        print(f"Free Energy = {F}")

        # stopping criterion
        if it > 2:
            if F_list[-1]-F_list[-2] < 1e-100:
                print("Reached cut-off after {} iterations".format(it))
                break
            # check for increase in F
            assert F_list[-1] >= F_list[-2]

    # Plot free energy
    plt.plot(F_list)
    plt.xlabel("Iteration")
```

```
    plt.ylabel("Free Energy")
    plt.title("Free Energy vs. Iteration")
    plt.show()


    return mu, sigma, pie, lambda_
```

Question 3(e) LearnBinFactors function

3.(f)

We run the LearnBinFactors function. Here are the track of the free energy and plot of features μ.



(a) Free Energy vs Iterations

(b) Learned Features

Figure 19: Results from one run

The learned feature from a random generated data is not perfect. In order to get a better result, we generate a set of data and take 10 runs. In each run, we track the final free energy and μ values. Here is corresponding code:

```
F_list = []
mu_list = []
for i in range(10):
    Y = generate_feature_data(seed = i)
    mu, sigma, pie, lambda_ = LearnBinFactors(Y, K, 300)
    F = Find_Free_Energy(Y, mu, sigma, pie, lambda_)
    F_list.append(F)
    mu_list.append(mu)
```

Question 3(f) Multiple runs for improvement

Then, we can plot the best output:

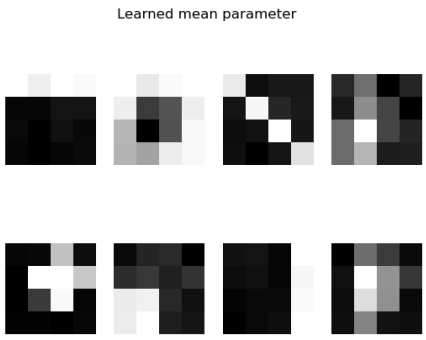


Figure 20: Best learned features in 10 run

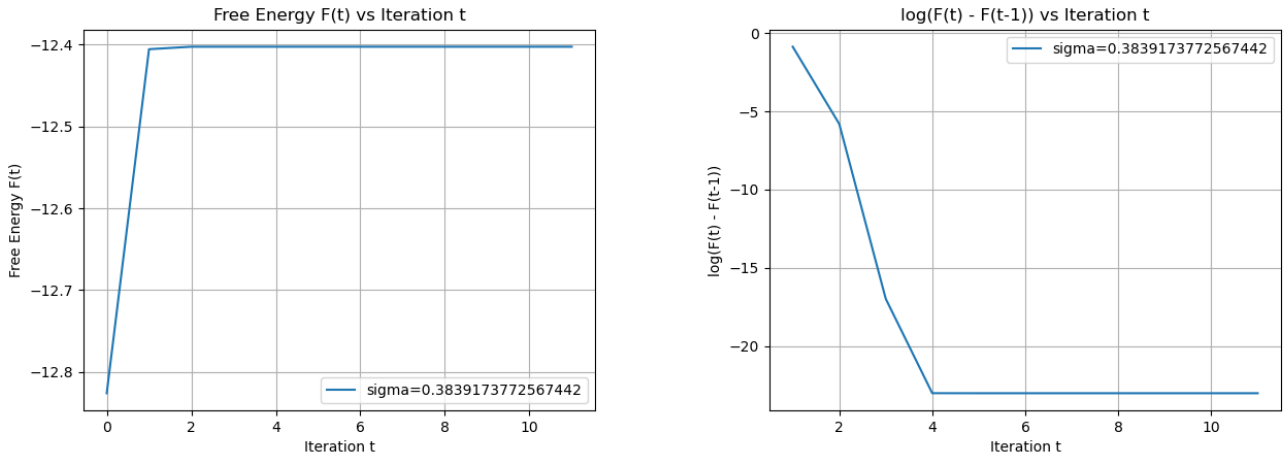
Comparing to the first set of features, this one is much clear.

Choice of Parameters

- **K**: Since the data is generated based on 8 features, we set $K = 8$ in our algorithm.
- λ: We generate a $N \times K$ matrix with random elements as our initial value.
- **Other parameters**: Based on the random λ, we then can compute other parameters ES, ESS, π, μ, and σ. These values can be regraded as initial values for the algorithm.

3.(g)

Here are the plots using the learned parameters from LearnBinFactors function.

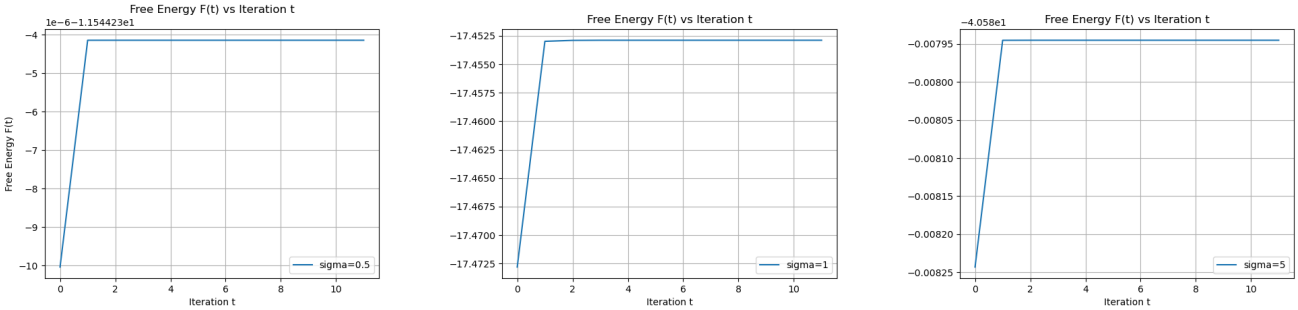


(a) Free energy with learned parameters

(b) $\log(F(t) - F(t-1))$

Figure 21

Both plots show rapid convergence. The original one is faster as we use the learned parameters. The variational case is also fast with only 4 steps. Now, we vary our choice of sigma and plot their free energy against iterations. Here are the results.



(a) $\sigma = 0.5$

(b) $\sigma = 1$

(c) $\sigma = 5$

From the above plots, we find all our choice converge rapidly. However, it is hard for us to make further observations. In that case, we plot $\log(F(t) - F(t-1))$ for different σ:

For $\sigma = 0.5$, the quadratic penalty term dominates, strongly constraining the updates to λ. This leads to faster convergence but might result in stricter approximations of the posterior. For Large $\sigma(\sigma = 5)$: The quadratic penalty term becomes negligible, and the updates are guided primarily by other terms in the free energy. While this allows flexibility, the updates are also less constrained, allowing smoother but slower stabilization. At $\sigma = 1$, there is a balance between the quadratic term and other contributions in $F(t)$. This intermediate state results in neither strong constraints (as in small σ) nor completely smooth updates (as in large σ). This balance may lead to oscillations or longer adjustment periods as the variational parameters stabilize, increasing the number of iterations required for convergence.

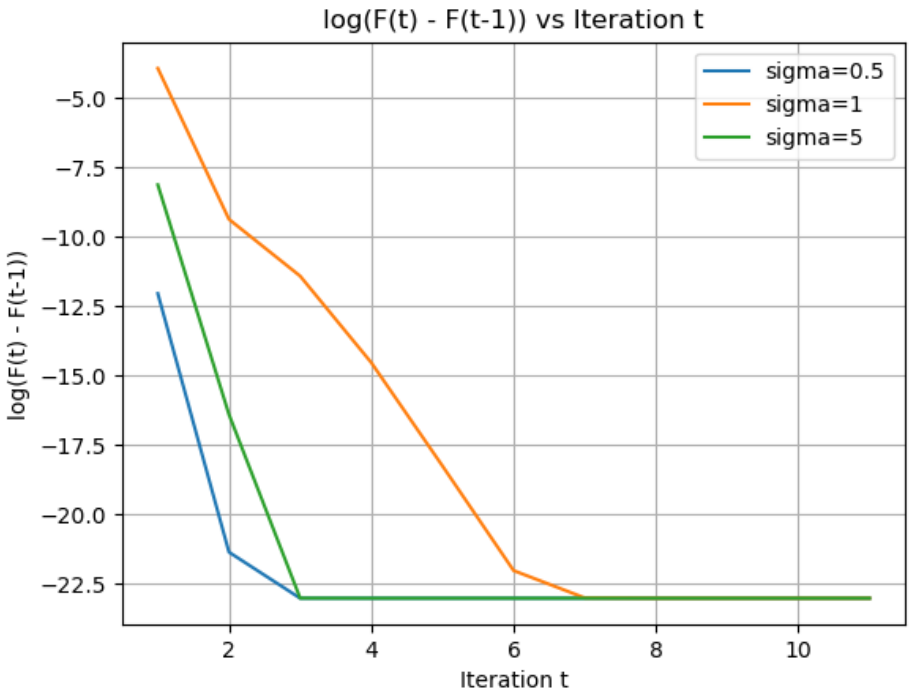


Figure 23: Logarithmic Difference

In summary, smaller σ ensures faster convergence with stricter constraints, while larger σ provides more flexibility but slower stabilization. Moderate σ balances these effects but may converge more slowly. The choice of σ depends on the desired trade-off between precision and flexibility.

Here are the codes for (g).

```
# Single data point X_1 (N=1)
Y = generate_feature_data(seed = 10)

# Initialize parameters
sigma_values = [0.1, 0.5, 1, 5]
maxsteps = 100

# Store results for each sigma
results = {}
mu, sigma, pie, lambd = LearnBinFactors(Y, K, 300)
sigma_values.append(sigma)
for sigma in sigma_values:
    # Run MeanField for a single data point
    lambda0 = np.random.rand(1, K)  # Initialize lambda for N=1
    lambd, F_list = MeanField(Y[0,:][None, :], mu, sigma, pie, lambd[0,:][None,:], 100)

    # Compute log difference of F
    log_differences = [np.log(F_list[i] - F_list[i - 1]+1e-10) if i > 0 else None for i in range(
    len(F_list))]

    # Store results
    results[sigma] = {
        "F_list": F_list,
        "log_differences": log_differences
    }
#plt.figure(figsize=(12, 6))
for sigma, data in results.items():
    #plt.figure(figsize=(10, 8))
    plt.plot(data["F_list"], label=f'sigma={sigma}')
    plt.xlabel('Iteration t')
    plt.ylabel('Free Energy F(t)')
    plt.title('Free Energy F(t) vs Iteration t')
    plt.grid()
    plt.legend()
    # save for each sigma
```

```
    plt.savefig(f"free_energy_sigma_{sigma}.png")
    plt.show()

# Plot log(F(t) - F(t-1)) for each sigma
#plt.figure(figsize=(12, 6))
for sigma in [0.5, 1, 5]:
    data = results[sigma]
    plt.plot(data["log_differences"], label=f'sigma={sigma}')
    plt.xlabel('Iteration t')
    plt.ylabel('log(F(t) - F(t-1))')
    plt.title('log(F(t) - F(t-1)) vs Iteration t')
    plt.legend()
    plt.grid()
    # save for each sigma
plt.savefig('combined_log_diff.png')
plt.show()
```

Question 3(g) Plot F and Logarithmic Difference

Question 4 (Bonus)

4.(a)

We aim to maximise free energy with respect to Q_θ rather than θ in Q3. Here we define distribution:

$$p\left(\boldsymbol{\mu}_i \mid \alpha_i\right)=\mathcal{N}\left(\boldsymbol{\mu}_i \mid \mathbf{0}, \alpha_i^{-1} \mathbf{I}\right), \tag{31}$$

which s a Gaussian prior for μ. The precision prior is:

$$p\left(\alpha_i\right)=\operatorname{Gamma}\left(\alpha_i \mid a_0, b_0\right). \tag{32}$$

Using the expression of VB free energy in lecture:

$$\mathcal{F}\left(Q_{\mathcal{Z}}(\mathcal{Z}), Q_\Lambda(\Lambda), \Psi, \boldsymbol{\alpha}\right)=\langle \log P(\mathcal{X}, \mathcal{Z} \mid \Lambda, \Psi)+\log P(\Lambda \mid \boldsymbol{\alpha})+\log P(\Psi)\rangle_{Q_{\mathcal{Z}} Q_\Lambda}+\ldots, \tag{33}$$

we can find that the original free energy remain unchanged and we just need to add new prior and entropy term. Before deriving the new form of free energy, we need to find the posterior for μ:

$$\underbrace{p\left(\boldsymbol{\mu}_k \mid \alpha_k\right)}_{\text{Gaussian prior}} \times \underbrace{\prod_{n=1}^N p\left(\mathbf{x}^{(n)} \mid \mathbf{s}^{(n)},\left\{\boldsymbol{\mu}_j\right\}, \sigma^2\right)}_{\text{Gaussian likelihood}} \propto \underbrace{q\left(\boldsymbol{\mu}_k\right)}_{\text{Gaussian posterior}}. \tag{34}$$

Thus, we have:

$$q\left(\boldsymbol{\mu}_k\right)=\mathcal{N}\left(\boldsymbol{\mu}_k \mid \boldsymbol{\mu}_k^*, \Sigma_k^*\right), \quad \Sigma_k^*=\left(\alpha_k \mathbf{I}+\frac{1}{\sigma^2} \sum_{n=1}^N \lambda_{n, k}\right)^{-1}, \quad \boldsymbol{\mu}_k^*=\Sigma_k^* \frac{1}{\sigma^2} \sum_{n=1}^N \lambda_{n, k}\left[\mathbf{x}^{(n)}-\sum_{j \neq k} \lambda_{n, j} \boldsymbol{\mu}_j^*\right]. \tag{35}$$

Now, we can express as:

$$\begin{aligned}\mathcal{F}=\mathcal{F}_{Q 3}-\frac{1}{2} D N \sum_{k=1}^K \ln\left(2 \pi \alpha_k^{-1}\right)-N \sum_{k=1}^K \frac{\alpha_k}{2}\left\|\mu_k\right\|^2 \\ +\frac{1}{2} D N \sum_{k=1}^K \ln\left(2 \pi \beta_k\right)+\frac{1}{2} N K D\end{aligned} \tag{36}$$

where $\beta_k=\Sigma_k^*$. Based on the form of new free energy, our VB-EM also need to change. The update rule for λ is unchanged. Apart from that, new parameters α and β will update in the loop. Also, we need to update μ differently from Q3.

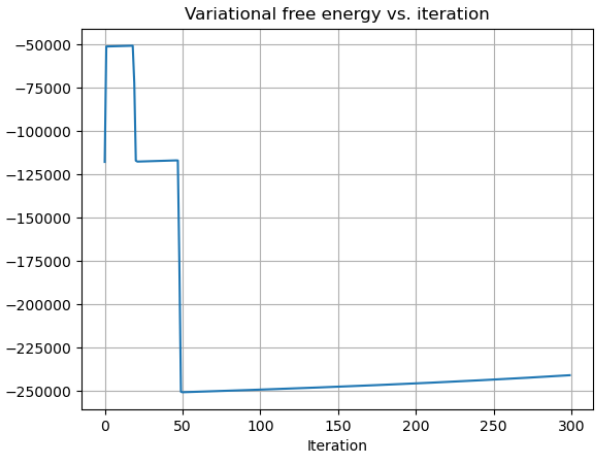
Here are the expressions for updates:

$$\begin{aligned}\alpha_i^* &= \frac{a_0+\frac{D}{2}}{b_0+\frac{1}{2}\left\langle\left\|\boldsymbol{\mu}_i\right\|^2\right\rangle_{q\left(\boldsymbol{\mu}_i\right)}} \approx \frac{D}{\left\|\boldsymbol{\mu}_k\right\|^2}, \\ \beta_i^* &= \left(\alpha_k \mathbf{I}+\frac{1}{\sigma^2} \sum_{n=1}^N \lambda_{n, k}\right)^{-1}, \\ \boldsymbol{\mu}_k^* &= \beta_k^* \frac{1}{\sigma^2} \sum_{n=1}^N \lambda_{n, k}\left[\mathbf{x}^{(n)}-\sum_{j \neq k} \lambda_{n, j} \boldsymbol{\mu}_j^*\right].\end{aligned} \tag{37}$$

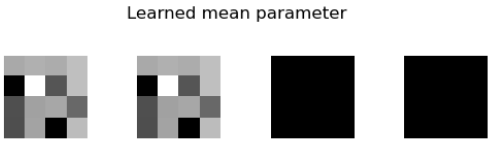
Under training, some α_i will inflate to large numbers which shows $\boldsymbol{\mu}_i \approx \mathbf{0}$. In that case, the model "discovers" the number of active factors from data, exactly as in ARD for continuous factor analysis. This is how the procedure automatically determines K.

4.(b)

Here are the results. There are something wrong with the algorithm but it do make some output. Here is the corresponding code.



(a) Free Energy vs Iterations



(b) Learned Features

Figure 24: Set K = 4

```python
def Find_Free_Energy(X, mu, sigma, pie, lambda0, alpha, beta):
    # Compute the free energy
    N, D = X.shape # Number of samples and dimensionality
    _, K = lambda0.shape # Number of clusters

    # Regularize lambda0 to avoid numerical issues
    lambda0[lambda0 >= 1] = 1 - 1e-12
    lambda0[lambda0 <= 0] = 1e-12

    # Compute ES and ESS
    ES, ESS = Find_ESS(lambda0)

    # Compute the diagonal of mu
    diagmu = np.diag(mu.T @ mu).flatten()

    # Free energy calculation
    F = - 0.5 * N * D * np.log(2 * np.pi * sigma**2)  # Constant term
    F -= 0.5 / sigma**2 * (
        np.trace(X.T @ X) +
        np.trace(mu.T @ mu @ ESS) -
        2 * np.trace(ES.T @ X @ mu)
    )  # Quadratic term
    F += np.sum(lambda0 * np.log(pie/lambda0) + (1 - lambda0) * np.log((1 - pie)/(1 - lambda0)))
      # Prior and entropy term

    F -=   0.5 * D * N * np.sum(np.log(2 * np.pi * (1/alpha))) + N * np.sum( alpha/2 * np.diag(
    diagmu) )

    F +=  0.5 * D * N * np.sum(np.log(2 * np.pi * beta)) + 0.5 * N * K * D

    return F

def Find_Mu(X, lambda0, sigma, alpha):
    N, D = X.shape
    _, K = lambda0.shape
    mu = np.zeros((D, K))
    lambda_new = lambda0
    for k in range(K):
        mu[:, k] = (np.sum(X.T*lambda_new[:, k], axis=1) - np.sum((lambda_new@mu.T).T*lambda_new[:,
    k], axis=1)
                            + (mu[:, k] * np.sum(lambda_new[:, k]**2))
                    /(np.sum(lambda_new[:, k]) + N * sigma**2 * alpha[k])
        )

    mu[mu > 1e100] = 1e100
    mu[mu < 1e-100] = 1e-100
    return mu
def Find_Beta(sigma, lambda0, alpha, beta):
    _, K = lambda0.shape
    for k in range(K):
```

```
        beta[k] = (np.sum(lambda0[:, k]/(sigma**2) + alpha[k])) ** (-1)
    return beta

def MeanField(X, mu, sigma, pie, lambda0, alpha, beta, maxsteps):
    # track the free energy
    Fs = []
    lambda_old = lambda0
    # early stopping
    eplison = 1e-10
    for i in range(maxsteps):
        lambda_new = Find_lambda(X, mu, sigma, pie, lambda_old, beta)
        mu = Find_Mu(X, lambda_new, sigma, alpha)
        beta = Find_Beta(sigma, lambda_new, alpha, beta)
        F = Find_Free_Energy(X, mu, sigma, pie, lambda_new, alpha, beta)
        Fs.append(F)
        if i > 10 and np.abs(Fs[-1] - Fs[-2]) < eplison:
            break
        lambda_old = lambda_new
    return lambda_old, Fs, mu, beta
def LearnBinFactors(X, K, iterations):
    # LearnBinFactors learns the parameters of a binary factor analysis model
    N, D = X.shape

    # Initialize parameters
    lambda_ = np.random.rand(N, K)  # Shape (N, K)
    mu = np.random.rand(D, K)  # Shape (D, K)
    beta = np.ones(K)
    alpha = np.ones(K)*2
    sigma = 1
    pie = np.random.rand(1, K)  # Shape (1, K)


    # Keep track of free energy
    F_list = []

    for it in range(iterations):
        print(f"Iteration {it+1}:")

        # E-step: Update lambda using MeanField
        lambda_, Fs, mu, beta = MeanField(X, mu, sigma, pie, lambda_, alpha, beta, maxsteps=300)

        # M-step: Update parameters
        ES, ESS = Find_ESS(lambda_)
        alpha, sigma, pie = m_step(X, ES, ESS, mu, beta)

        # Keep track of free energy
        F = Find_Free_Energy(X, mu, sigma, pie, lambda_, alpha, beta)
        F_list.append(F)
        print(f"Free Energy = {F}")

        # stopping criterion
        if it > 100:
            if F_list[-1]-F_list[-2] < 1e-100:
                print("Reached cut-off after {} iterations".format(it))
                break
            # check for increase in F
            assert F_list[-1] >= F_list[-2]

    # Plot free energy
    plt.plot(F_list)
    plt.xlabel("Iteration")
    plt.ylabel("Free Energy")
    plt.grid()
    plt.title("Free Energy vs. Iteration")
    plt.savefig("free_energy.png")
    plt.show()


    return mu, sigma, pie, lambda_, alpha, F_list
```

Question 4(b) Implement ARD

Question 5

5.(a)

The log-joint probability can be expressed as:

$$\begin{aligned}\log p(\mathbf{s},\mathbf{x}) &= \log p(\mathbf{x}\mid\mathbf{s}) + \log p(\mathbf{s}\mid\pi),\\ \log p(\mathbf{s},\mathbf{x}) &= \sum_{i=1}^{K}\left[s_i\log\pi_i + (1-s_i)\log(1-\pi_i)\right] + \log\mathcal{N}\left(\mathbf{x}\mid\sum_{i=1}^{K}s_i\boldsymbol{\mu}_i,\sigma^2 I\right).\end{aligned} \tag{38}$$

The Gaussian log-likelihood can be further expand as:

$$\begin{aligned}\log\mathcal{N}\left(\mathbf{x}\mid\boldsymbol{\mu},\sigma^2 I\right) &= -\frac{D}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\|\mathbf{x}-\boldsymbol{\mu}\|^2,\\ \Rightarrow\log\mathcal{N}\left(\mathbf{x}\mid\boldsymbol{\mu},\sigma^2 I\right) &= -\frac{1}{2\sigma^2}\left(-2\sum_{i=1}^{K}s_i\boldsymbol{\mu}_i^\top x + \sum_{i=1}^{K}s_i\|\boldsymbol{\mu}_i\|^2 + \sum_{i\neq j}^{K}\sum_{j=1}^{K}s_i s_j\boldsymbol{\mu}_i^\top\boldsymbol{\mu}_j\right) + \text{constant},\\ \Rightarrow\log\mathcal{N}\left(\mathbf{x}\mid\boldsymbol{\mu},\sigma^2 I\right) &= \sum_{i=1}^{K}\left(\frac{1}{\sigma^2}\boldsymbol{\mu}_i^\top x - \frac{1}{2\sigma^2}\|\boldsymbol{\mu}_i\|^2\right)s_i - \frac{1}{2\sigma^2}\sum_{i\neq j}^{K}\sum_{j=1}^{K}\boldsymbol{\mu}_i^\top\boldsymbol{\mu}_j s_i s_j + \text{constant},\end{aligned} \tag{39}$$

which is rearranged into linear term, pairwise term and constant. For the prior term, we have

$$\log p(\mathbf{s},\mathbf{x}) = \sum_{i=1}^{K}\left(\log\frac{\pi_i}{1-\pi_i}\right)s_i + \sum_{i=1}^{K}\log(1-\pi_i). \tag{40}$$

Combining them together, we have:

$$\log p(\mathbf{s},\mathbf{x}) = \sum_{i=1}^{K}\left(\frac{1}{\sigma^2}\boldsymbol{\mu}_i^\top x - \frac{1}{2\sigma^2}\|\boldsymbol{\mu}_i\|^2 + \log\frac{\pi_i}{1-\pi_i}\right)s_i + \sum_{i\neq j}^{K}\sum_{j=1}^{K}\left(-\frac{\boldsymbol{\mu}_i^\top\boldsymbol{\mu}_j}{2\sigma^2}\right)s_i s_j, \tag{41}$$

where we ignore the constant terms. Based on the expression shown in question, we have following relations:

$$\begin{aligned}f_i(s_i) &= \exp\left\{\left(\frac{1}{\sigma^2}\boldsymbol{\mu}_i^\top x - \frac{1}{2\sigma^2}\|\boldsymbol{\mu}_i\|^2 + \log\frac{\pi_i}{1-\pi_i}\right)s_i\right\},\\ g_{i,j}(s_i,s_j) &= \exp\left\{\left(-\frac{\boldsymbol{\mu}_i^\top\boldsymbol{\mu}_j}{2\sigma^2}\right)s_i s_j\right\}.\end{aligned} \tag{42}$$

Then, we have:

$$\log(p(\mathbf{s},\mathbf{x})) = \sum_i\log f_i(s_i) + \sum_{i\neq j}\log g_{ij}(s_i,s_j), \tag{43}$$

and this exactly matches the pairwise form of a Boltzmann Machine:

$$\log p(\mathbf{s},\mathbf{x}) = \sum_i a_i s_i + \sum_{i<j}b_{ij}s_i s_j + \text{ constant}, \tag{44}$$

where we have:

$$\begin{aligned}a_i &= \frac{1}{\sigma^2}\boldsymbol{\mu}_i^\top x - \frac{1}{2\sigma^2}\|\boldsymbol{\mu}_i\|^2 + \log\frac{\pi_i}{1-\pi_i},\\ b_{ij} &= -\frac{\boldsymbol{\mu}_i^\top\boldsymbol{\mu}_j}{2\sigma^2}.\end{aligned} \tag{45}$$

Further more, our joint probability can be written as:

$$\begin{aligned}p(\mathbf{s}\mid\mathbf{a},\mathbf{b}) &\propto \exp\left\{\sum_i a_i s_i + \sum_{i,j}b_{ij}s_i s_j\right\},\\ \Rightarrow p(\mathbf{s}) &\propto \prod_i f_i(s_i)\prod_{i,j}g_{i,j}(s_i,s_j).\end{aligned} \tag{46}$$

5.(b)

Using the distribution we derived in (a), we can write the approximate probability as:

$$q(s) = \prod_i \widetilde{f}_i(s_i) \prod_{i,j} \widetilde{g}_{ij}(s_i, s_j). \tag{47}$$

Now, we need to consider the update for $\widetilde{f}_i$ and $\widetilde{g}_{ij}$. Let us start with $\widetilde{f}_i$. The cavity distribution is:

$$q_{\neg f_i}(\mathbf{s}) = \prod_{j \neq i} \tilde{f}_j(\int_j) \prod_{i,j} \widetilde{g}_{ij}(s_i, s_j) \tag{48}$$

and we can update it as:

$$\tilde{f}_i^{\text{new}}(\int_i) \leftarrow \underset{f \in \{\tilde{f}\}}{\operatorname{argmin}} \operatorname{KL}\left[f_i(\int_i) q_{\neg i}(\int) \| \tilde{f}_i(\int_i) q_{\neg i}(\int) \right]. \tag{49}$$

However, we find this process is trivial as the new function is constrained to be Bernoulli distribution. Our update just varies the Bernoulli parameter. In that case, we let $\tilde{f}_i = f_i$.

Now, we approximate g_{ij}. The cavity distribution is:

$$q_{\neg g_{ij}}(\mathbf{s}) = \prod_i f_i(\int_i) \prod_{(m,n) \neq (i,j)} \widetilde{g}_{mn}(s_m, s_n). \tag{50}$$

Here we define the expression for g_{ij} and $\tilde{g}_{ij}$:

$$g_{ij}(s_i, s_j) = \exp(b_{ij} s_i s_j) \quad \text{and} \quad \tilde{g}_{ij}(s_i, s_j) = \exp(\mu_{ij} s_j + \mu_{ji} s_i) \tag{51}$$

where we take the pairwise term as a product of two Bernoulli distributions.

In that case, we have:

$$\widetilde{g}_{ij}^{\text{new}}(s_i, s_j) \leftarrow \underset{g \in \tilde{g}}{\operatorname{argmin}} KL\left\{ g_{ij}(s_i, s_j) q_{\neg g_{ij}}(\mathbf{s}) \| \tilde{g}_{ij}(s_i, s_j) q_{\neg g_{ij}}(\mathbf{s}) \right\}. \tag{52}$$

First, we calculate the LHS of KL:

$$\begin{aligned} g_{ij} q_{\neg g_{ij}} &= \exp\left\{ b_{ij} s_i s_j + \sum_i a_i s_i + \sum_{k \neq i,j} (\mu_{ki} s_i + \mu_{kj} s_j) \right\}, \\ \Rightarrow g_{ij} q_{\neg g_{ij}} &\propto \exp\left\{ b_{ij} s_i s_j + a_i s_i + a_j s_j + \sum_{k \neq i,j} (\mu_{ki} s_i + \mu_{kj} s_j) \right\}, \end{aligned} \tag{53}$$

and the RHS of KL:

$$\begin{aligned} \tilde{g}_{ij} q_{\neg g_{ij}} &= \exp\left\{ \mu_{ij} s_j + \mu_{ji} s_i + \sum_i a_i s_i + \sum_{k \neq i,j} (\mu_{ki} s_i + \mu_{kj} s_j) \right\}, \\ \Rightarrow g_{ij} q_{\neg g_{ij}} &\propto \exp\left\{ \mu_{ij} s_j + \mu_{ji} s_i + a_i s_i + a_j s_j + \sum_{k \neq i,j} (\mu_{ki} s_i + \mu_{kj} s_j) \right\}, \\ \Rightarrow g_{ij} q_{\neg g_{ij}} &\propto \exp\left\{ \left(\mu_{ji} + a_i + \sum_{k \neq i,j} \mu_{ki} \right) s_i + \left(\mu_{ij} + a_j + \sum_{k \neq i,j} \mu_{kj} \right) s_j \right\}. \end{aligned} \tag{54}$$

For our convenience, we denote:

$$\nu_i = a_j + \sum_{k \neq i,j} \mu_{kj} \quad \text{and} \quad \nu_j = a_j + \sum_{k \neq i,j} \mu_{kj}. \tag{55}$$

Then, we have:

$$\begin{aligned} \mathbb{E}_p(s_i) &= \frac{\sum_{s_i=1} \sum_{s_j} g_{ij}(s_i, s_j) q_{\neg ij}}{\sum_{s_i} \sum_{s_j} g_{ij}(s_i, s_j) q_{\neg ij}}, \\ \mathbb{E}_p(s_i) &= \frac{\exp(b_{ij} + \nu_i + \nu_j) + \exp(\nu_i)}{\exp(b_{ij} + \nu_i + \nu_j) + \exp(\nu_i) + \exp(\nu_j) + 1}, \end{aligned} \tag{56}$$

From the minimisation of KL(moment matching), we find that $\mathbb{E}_p(s_i) = \mathbb{E}_q(s_i)$. Together with Bernoulli distribution assumption on $q(s)$, we have the following relation:

$$\mu_{ji}^{\text{new}} + \nu_i = \log \frac{\mathbb{E}_q(s_i)}{1 - \mathbb{E}_q(s_i)} \Rightarrow \mu_{ji}^{\text{new}} = \log \frac{\mathbb{E}_q(s_i)}{1 - \mathbb{E}_q(s_i)} - \nu_i, \tag{57}$$

where can substitute the expression of $\mathbb{E}_q$:

$$\begin{aligned} \mu_{ji}^{\text{new}} &= \log \frac{\exp(b_{ij} + \nu_i + \nu_j) + \exp(\nu_i)}{1 + \exp(\nu_j)} + \log \exp(-\nu_j), \\ \Rightarrow \mu_{ji}^{\text{new}} &= \log \frac{\exp(b_{ij} + \nu_j) + 1}{1 + \exp(\nu_j)}. \end{aligned} \tag{58}$$

We have similar expression for μ_{ij}^{new} by switch the index.

In summary, we have:

$$\mu_{ji}^{\text{new}} = \log \frac{\exp(b_{ij} + \nu_j) + 1}{1 + \exp(\nu_j)} \quad \text{and} \quad \mu_{ij}^{\text{new}} = \log \frac{\exp(b_{ij} + \nu_i) + 1}{1 + \exp(\nu_i)}. \tag{59}$$

5.(c)

We can rewrite each pairwise factor $\tilde{g}_{ij}(s_i, s_j)$ as a product of "messages":

$$\tilde{g}_{ij}(s_i, s_j) = m_{i \to j}(s_j) \times m_{j \to i}(s_i) \times (\text{ some normalization constant }), \tag{60}$$

Then, the update of $\tilde{g}_{ij}$ is also the update of message. The cavity distribution in terms of message can be expressed as:

$$q_{\neg ij}(s_i, s_j) = f_i(s_i) f_j(s_j) \prod_{k \in \text{ne}(i) \setminus j} m_{k \to i}(s_i) \prod_{l \in \text{ne}(j) \setminus i} m_{l \to j}(s_j) \tag{61}$$

Thus, we have the projection:

$$\left\{ m_{i \to j}^{\text{new}}, m_{j \to i}^{\text{new}} \right\} = \operatorname{argmin} \text{KL}\left[g_{ij}(s_i, s_j) q_{\neg ij}(s_i, s_j) \| m_{j \to i}(s_i) m_{i \to j}(s_j) q_{\neg ij}(s_i, s_j) \right]. \tag{62}$$

The minimisation is achieved by marginals of $g_{ij}(\cdot) q_{\neg ij}(\cdot)$:

$$\begin{aligned} M_{j \to i}^{\text{new}}(s_i) &= \sum_{s_j} \left(g_{ij}(s_i, s_j) f_j(s_j) \prod_{l \in \text{ne}(j) \setminus i} M_{l \to j}(s_j) \right), \\ M_{i \to j}^{\text{new}}(s_j) &= \sum_{s_i} \left(g_{ij}(s_i, s_j) f_i(s_i) \prod_{k \in \text{ne}(i) \setminus j} M_{k \to i}(s_i) \right). \end{aligned} \tag{63}$$

Thus, **EP** on a pairwise model with factored site approximations leads to the familiar loopy BP equations on the factor graph.

5.(d)

We introduce a latent parameter:

$$\mu = (\mu_1, \ldots, \mu_K), \quad \alpha = (\alpha_1, \ldots, \alpha_K), \tag{64}$$

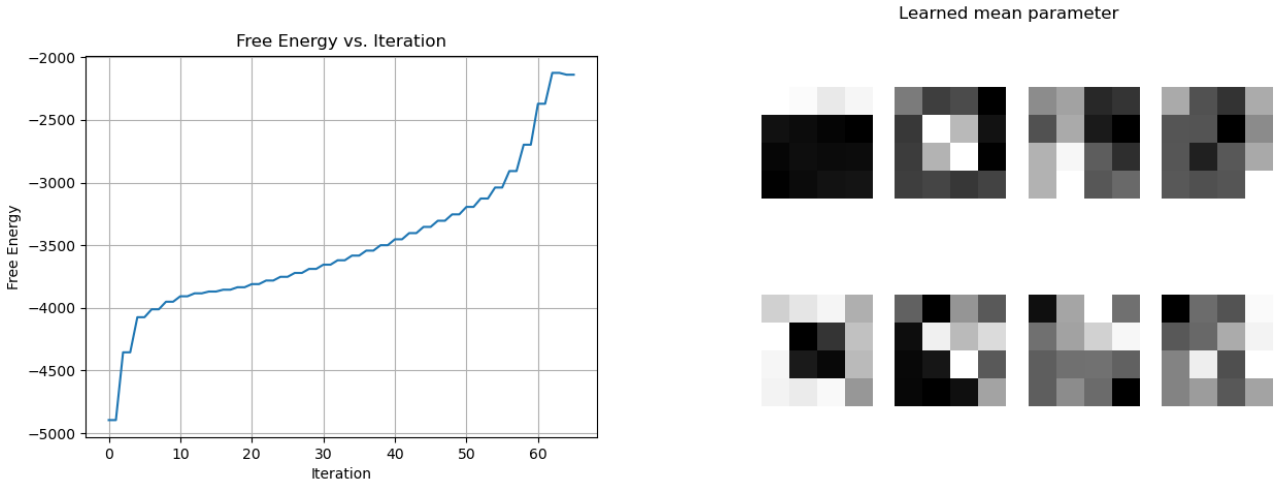
with a Gaussian prior:

$$p(\boldsymbol{\mu}_i \mid \alpha_i) = \mathcal{N}\left(\boldsymbol{\mu}_i \mid \mathbf{0}, \alpha_i^{-1} \mathbf{I}\right) \tag{65}$$

Then, we apply Loopy-BP with ARD to approximate the joint probability $p(\mathbf{s}, \mu \,|\, \alpha)$. After the convergence, we maximise the log posterior. This M-step gives us a optimised hyper parameter. Any $\alpha_d^* \to -\infty$ implies the corresponding μ_d is pushed close to zero. In that case, the number of dimensions d for which α_d^* does not go to $-\infty$ tells you how many latent parameters are relevant-in effect determining K.

Question 6 (Bonus)

Here we implement the loop-BP as the E-step and keep the M-step unchanged. The following graphs show the change of free energy in each iteration and the learned feature.

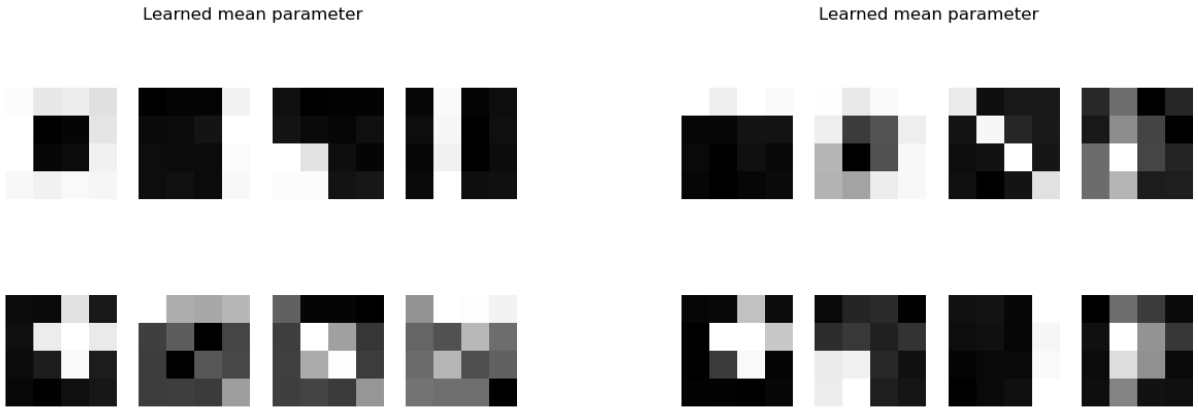


(a) Free Energy vs Iterations

(b) Learned Features

Figure 25: Loopy BP Results from one run

As we use the same conditions as mean field case in Question 3, the mean field algorithm performs better in a single run. Then, we perform 10 runs and take the best output. This is also performed in Question 3. Thus, we can compare the results in 10 runs for loopy BP and mean field.



(a) Loopy BP

(b) Mean Field

Figure 26: Best learned features in 10 run

From the experimental results, both the Loopy-BP and Mean-Field algorithms learn distinctive features and achieve comparable performance in most scenarios. However, in certain runs, the Loopy-BP method can suffer from poor convergence due to oscillations in the free energy. These oscillations may result in "black blocks" or overly uniform patterns in the inferred parameters, indicating that the Loopy-BP updates have not converged properly. Consequently, under these circumstances, Loopy-BP can yield worse outcomes than the Mean-Field approach.

In summary, while both algorithms can discover meaningful patterns, Mean-Field generally exhibits more stable convergence, whereas Loopy-BP may yield suboptimal results if it fails to converge due to free energy oscillations.

Here are the codes for Loopy-BP. Since large part of code is the same as Question 3, we just show the changed code.

```
def EP(X, mu, sigma, pie, message0, maxsteps):
    # Dimensions
```

```
    N, D = X.shape
    _, K, _ = message0.shape

    # initialisation
    message = np.zeros_like(message0)
    Fval = -np.inf
    epsilon = 1e-10

    # single variable update f_i(s_i)
    f_i = np.zeros((N, K))
    diag_mu = np.diag(mu.T@mu).flatten()
    for n in range(N):
        f_i[n, :] = np.log(pie/(1-pie)) + 1/(sigma**2) * X[n, :]@mu - 1/(2*sigma**2) * diag_mu

    # pairwise variable update g_ij(s_i, s_j)
    for _ in range(maxsteps):
        for n in range(N):
            message_n = message0[:, :, n].copy() # KxK
            for i in range(K):
                for j in range(i+1, K):
                    alpha = 0.5 # damping parameter
                    # j to i
                    b_ij = -mu[:, i] @ mu[:, j]/(sigma**2)
                    nv_j = f_i[n, j] + np.sum(message_n[:, j]) - message_n[i, j]
                    message_ji = (np.exp(nv_j + b_ij) + 1)/(1 + np.exp(nv_j))
                    message_ji_new = np.log(message_ji)
                    # apply damping
                    message_n[j, i] = alpha*message_n[j, i] + (1-alpha)*message_ji_new

                    # i to j
                    b_ji = -mu[:, j] @ mu[:, i]/(sigma**2)
                    nv_i = f_i[n, i] + np.sum(message_n[:, i]) - message_n[j, i]
                    message_ij = (np.exp(nv_i + b_ji) + 1)/(1 + np.exp(nv_i))
                    message_ij_new = np.log(message_ij)
                    # apply damping
                    message_n[i, j] = alpha*message_n[i, j] + (1-alpha)*message_ij_new
            message[:, :, n] = message_n

        lambdas = np.zeros((N, K))
        for n in range(N):
            z = f_i[n, :] + np.sum(message[:, :, n], axis=0)
            lambdas[n, :] = 1/(1 + np.exp(-z))

        ES, ESS = Find_ESS(lambdas)

        lambdas[lambdas >= 1] = 1 - 1e-15
        lambdas[lambdas <= 0] = 1e-15
        F_new = Find_Free_Energy(X, mu, sigma, pie, lambdas)

        diff_message = np.max(np.abs(message - message0))
        if diff_message < epsilon:
            Fval = F_new
            break
        Fval = F_new
        message0 = message.copy()
    return lambdas, Fval, message

def LearnBinFactors(X, K, iterations):
    # dimensions
    N, D = X.shape
    F_list = []
    Fval = -np.inf
    epsilon = 1e-100
    maxsteps = 50
    # initialisation
    lambda0 = np.random.rand(N, K)
    ES, ESS = Find_ESS(lambda0)
    mu, sigma, pie = m_step(X, ES, ESS)
    message0 = np.random.rand(K, K, N)

    for n in range(N):
        message_n = message0[:, :, n]
        # zero diagonal
        np.fill_diagonal(message_n, 0)
        message0[:, :, n] = message_n

```

```
    for i in range(iterations):
        # E-step
        lambd, F_new, message = EP(X, mu, sigma, pie, message0, maxsteps)
        F_list.append(F_new)
        # M-step
        ES, ESS = Find_ESS(lambd)
        mu, sigma, pie = m_step(X, ES, ESS)
        F_list.append(F_new)
        print("Iteration number {} with free energy{:.4f}".format(i, F_new))
        if (F_new - Fval) < epsilon:
            break
        Fval = F_new
        message0 = message

    return mu, sigma, pie, lambd, F_list
```

Question 6 Implement the EP/loopy-BP algorithm