COMP0085

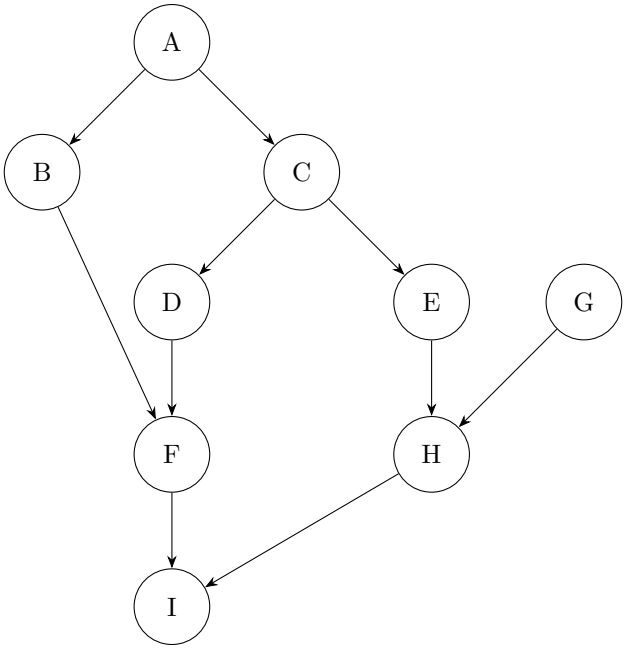
NMWX9

December 26, 2024

Contents

1 A biochemical pathway

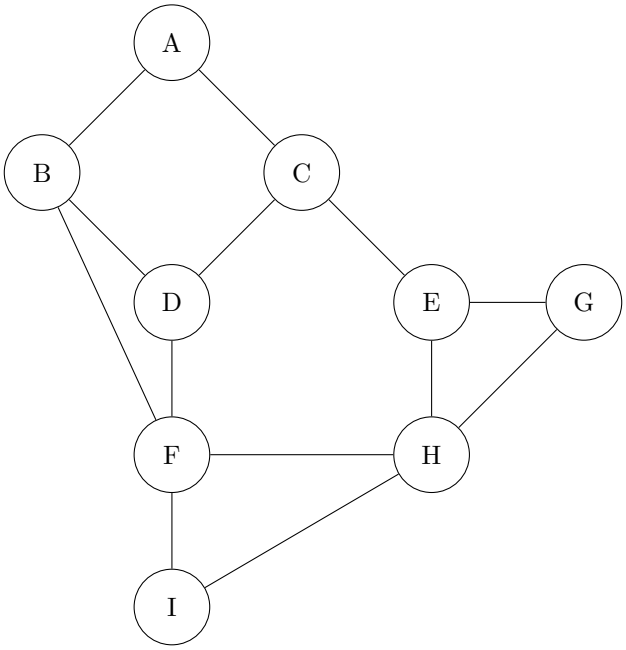
1.1 a



1.2 b

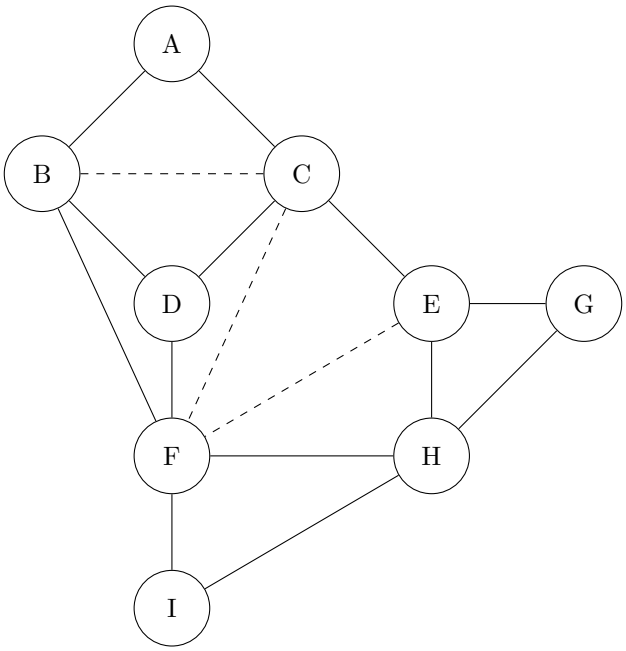
1.2.1 the moralised graph

We identify nodes with multiple parent, marrying the parents and drop the direction of edges



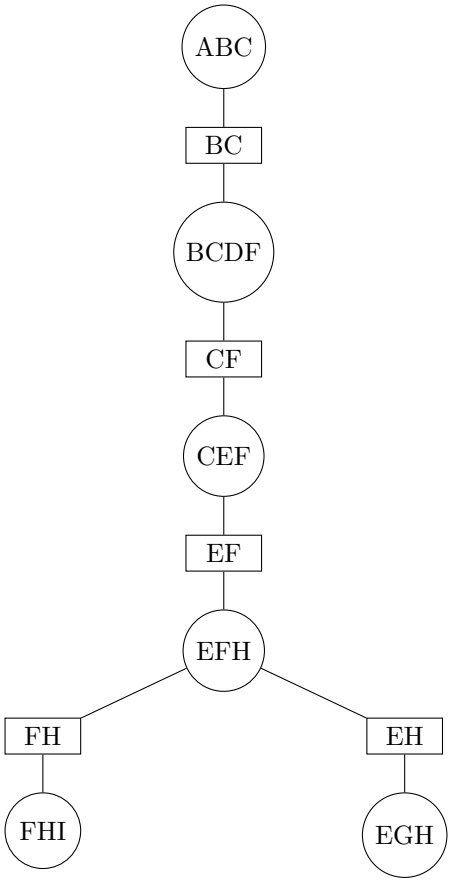
1.2.2 an efficient triangulation

we added dashed lines as edges to triangulate the moralised graph.

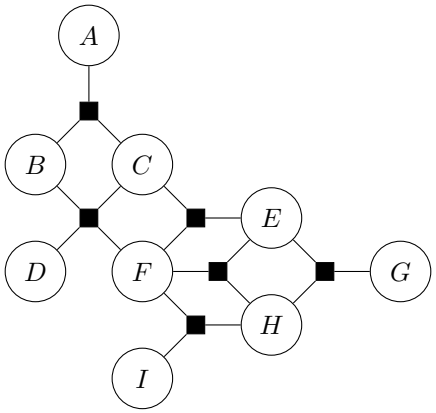


1.2.3 the resulting junction tree

circular nodes represents cliques and the square nodes represents separators or factors.

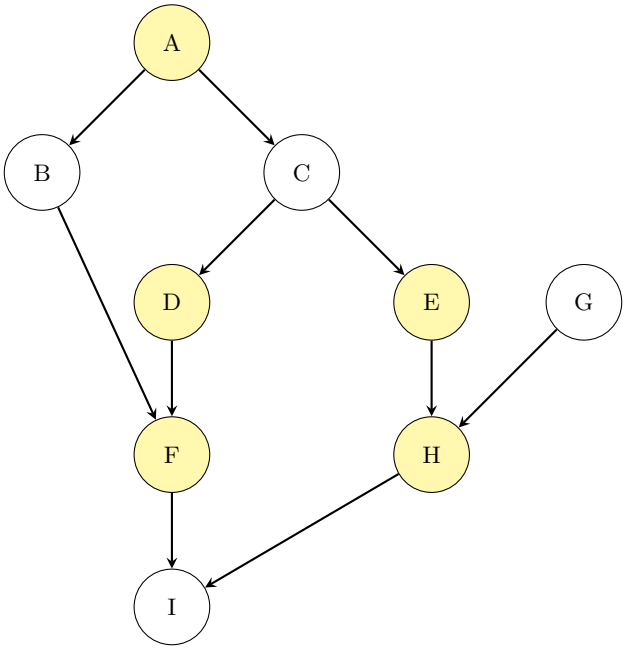


1.2.4 the junction tree redrawn as a factor graph.



1.3 c

set $\{A, D, E, F, H\}$ is a (non-unique) smallest set of of molecules, , such that if the concentrations of the species within the set are known, the concentrations of the others ($\{B, C, G, I\}$)would all be independent, other two minimal conditioning sets are $\{B, C, E, F, H\}$ and $\{B, C, G, F, H\}$



1.4 d

read out from the DAG we can represent the system using the following matrices:

$$\Lambda = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \Lambda_{BA} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \Lambda_{CA} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Lambda_{DC} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Lambda_{EC} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \Lambda_{FB} & 0 & \Lambda_{FD} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \Lambda_{HE} & 0 & \Lambda_{HG} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \Lambda_{IF} & 0 & \Lambda_{IH} & 0 \end{bmatrix} \text{ and } \mathbf{z} = \begin{bmatrix} z_A \\ z_B \\ z_C \\ z_D \\ z_E \\ z_F \\ z_G \\ z_H \\ z_I \end{bmatrix}$$

we were only given $\delta[B]$, $\delta[D]$, $\delta[E]$ and $\delta[G]$, we can simplify the system by focusing only on the relevant parent nodes. This reduction involves selecting the rows of Λ corresponding to the observed variables and the columns corresponding to their parent nodes. The reduced system focusing on the relevant observations is::

$$\begin{bmatrix} \delta[B] \\ \delta[D] \\ \delta[E] \\ \delta[G] \end{bmatrix} = \begin{bmatrix} \Lambda_{BA} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Lambda_{DC} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Lambda_{EC} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} z_A \\ z_B \\ z_C \\ z_D \\ z_E \\ z_F \\ z_G \\ z_H \\ z_I \end{bmatrix} + \begin{bmatrix} \epsilon_B \\ \epsilon_D \\ \epsilon_E \\ \epsilon_G \end{bmatrix}$$

read out from the DAG we noticed that the latent vector $\mathbf{z}$ only includes $\delta[A]$ and $\delta[C]$. we can reduce the system to focus on these parent factors. Defining the observation equation with the relevant parent nodes $\delta[A]$ and $\delta[C]$, we obtain:

$$\begin{bmatrix} \delta[B] \\ \delta[D] \\ \delta[E] \end{bmatrix} = \begin{bmatrix} \Lambda_{BA} & 0 \\ 0 & \Lambda_{DC} \\ 0 & \Lambda_{EC} \end{bmatrix} \begin{bmatrix} \delta[A] \\ \delta[C] \end{bmatrix} + \begin{bmatrix} \epsilon_B \\ \epsilon_D \\ \epsilon_E \end{bmatrix}$$

From the reduced observation equation, it is evident that the system depends only on the factors $\delta[A]$ and $\delta[C]$, which are the two parent nodes influencing the observed variables $\delta[B]$, $\delta[D]$, and $\delta[E]$. Consequently, we can recover the values of $\delta[A]$ and $\delta[C]$ based on the observed data.

1.5 e

From the previous analysis, we have the following.

- Observed variables: $\delta[B]$, $\delta[D]$, $\delta[E]$, and $\delta[G]$.
- Latent factors recovered: $\delta[A]$ and $\delta[C]$.

The DAG structure tells us:

- $\delta[B]$ depends on $\delta[A]$.
- $\delta[D]$ and $\delta[E]$ depend on $\delta[C]$.
- Other nodes like $\delta[F]$, $\delta[H]$, and $\delta[I]$ are further downstream.

This informs us the identifiability of each node:

- Latent Nodes ($\delta[A]$, $\delta[C]$):
  - Identifiable up to Scale: We can estimate $\delta[A]$ and $\delta[C]$ up to an unknown scale factor.
  - Non-identifiable Nodes: All the others: Nodes like $\delta[F]$, $\delta[H]$, and $\delta[I]$ cannot be identified because there is insufficient data directly relating to them.

- Observed Nodes ($\delta[B]$, $\delta[D]$, $\delta[E]$, $\delta[G]$):
  - Fully Identifiable: Their values are known from the observations.

Identifiability of Each Weight (Linear Coefficients):

- Weights Connecting to Observed Variables:
  - Estimable up to Scale: The weights Λ_{BA}, Λ_{DC}, and Λ_{EC} can be estimated but only up to a proportionality constant due to the scale indeterminacy of the latent factors. Kowning $\delta[D]$ from factor analysis we can also recover Λ_{CA} up to a scale
- Weights Connecting Latent Variables:
  - Non-identifiable: Weights between unobserved nodes (e.g., Λ_{FD}, Λ_{FB}) cannot be estimated without additional data or constraints.

2 Bayesian linear and Gaussian process regression

2.1 a

MAP Python Code can be found in Appendix Problem 2, Part (a). We have a dataset of N observations $\{(t_n, y_n)\}_{n=1}^N$, where y_n is the CO_2 concentration at time t_n. We model the data using a linear function plus noise:

$$f(t) = at + b + \varepsilon(t),$$

where the noise terms are independent and identically distributed Gaussian random variables:

$$\varepsilon(t) \sim \mathcal{N}(0, 1).$$

We place Gaussian priors on the parameters a and b:

$$a \sim \mathcal{N}(0,\, 10^2), \quad b \sim \mathcal{N}(360,\, 100^2).$$

Let us define the parameter vector and the prior more compactly:

$$\mathbf{w} = \begin{pmatrix} a \\ b \end{pmatrix}, \quad \boldsymbol{\mu}_0 = \begin{pmatrix} 0 \\ 360 \end{pmatrix}, \quad \boldsymbol{\Sigma}_0 = \begin{pmatrix} 10^2 & 0 \\ 0 & 100^2 \end{pmatrix}.$$

Thus, the prior distribution is:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} \mid \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0).$$

The likelihood of the data given the parameters $\mathbf{w}$ is:

$$p(\mathbf{y} \mid \mathbf{w}) = \prod_{n=1}^{N} \mathcal{N}(y_n \mid at_n + b, 1),$$

where $\mathbf{y} = (y_1, \ldots, y_N)^\top$.

We can write the model in matrix form. Define the design matrix $\mathbf{X}$ as:

$$\mathbf{X} = \begin{pmatrix} t_1 & 1 \\ t_2 & 1 \\ \vdots & \vdots \\ t_N & 1 \end{pmatrix}.$$

Then:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\varepsilon}, \quad \text{with} \quad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}).$$

The likelihood in vector form is (Note that the noise model implicitly sets $\sigma^2 = 1$):

$$p(\mathbf{y} \mid \mathbf{w}) = (2\pi)^{-N/2} \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})\right).$$

Bayes' theorem gives us the posterior distribution:

$$p(\mathbf{w} \mid \mathbf{y}) \propto p(\mathbf{y} \mid \mathbf{w}) p(\mathbf{w}).$$

Substitute the Gaussian forms:

$$p(\mathbf{w} \mid \mathbf{y}) \propto \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})\right) \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Sigma}_0^{-1} (\mathbf{w} - \boldsymbol{\mu}_0)\right).$$

Combining terms in the exponent, we get a quadratic form in $\mathbf{w}$:

$$\log p(\mathbf{w} \mid \mathbf{y}) \propto -\frac{1}{2}(\mathbf{w}^\top (\mathbf{X}^\top \mathbf{X} + \boldsymbol{\Sigma}_0^{-1})\mathbf{w} - 2(\mathbf{X}^\top \mathbf{y} + \boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0)^\top \mathbf{w} + \text{const.}).$$

This is recognized as a Gaussian distribution in $\mathbf{w}$ with:

$$\boldsymbol{\Sigma}_{\text{post}}^{-1} = \boldsymbol{\Sigma}_0^{-1} + \mathbf{X}^\top \mathbf{X}, \quad \mu_{\text{post}} = \boldsymbol{\Sigma}_{\text{post}}(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0 + \mathbf{X}^\top \mathbf{y}).$$

Inverting for the posterior covariance:

$$\boldsymbol{\Sigma}_{\text{post}} = (\boldsymbol{\Sigma}_0^{-1} + \mathbf{X}^\top \mathbf{X})^{-1}.$$

Thus, the posterior distribution is:

$$p(\mathbf{w} \mid \mathbf{y}) = \mathcal{N}(\mathbf{w} \mid \mu_{\text{post}}, \boldsymbol{\Sigma}_{\text{post}}).$$

Here:

$$\mu_{\text{post}} = \boldsymbol{\Sigma}_{\text{post}}(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0 + \mathbf{X}^\top \mathbf{y}), \quad \boldsymbol{\Sigma}_{\text{post}} = (\boldsymbol{\Sigma}_0^{-1} + \mathbf{X}^\top \mathbf{X})^{-1}.$$

Substituting the given priors:

$$\boldsymbol{\Sigma}_0 = \begin{pmatrix} 10^2 & 0 \\ 0 & 100^2 \end{pmatrix}, \quad \boldsymbol{\Sigma}_0^{-1} = \begin{pmatrix} 0.01 & 0 \\ 0 & 0.0001 \end{pmatrix}.$$

We can use these results to directly compute the posterior mean and covariance of (a, b) given the observed CO_2 data. The result is shown in the table below (Note we set our "base year" to 1980.0 as this is where the data begins and you can not have negative co2 concentration at year 0, but if we do set our intercept at year 0 then the posterior mean of b would be -3266.15, and for the covariance matrix $\boldsymbol{\Sigma}_{\text{post}} = \begin{pmatrix} 1.375 & -0.028 \\ -0.028 & 55.039 \end{pmatrix}$):

Table 1: Posterior Means of Parameters a and b

Parameter	**Posterior Mean**
a	1.8284
b	334.1269

Table 2: Posterior Covariance Matrix of Parameters a and b

	a	b
a	1.3824×10^{-5}	-2.8800×10^{-4}
b	-2.8800×10^{-4}	8.0000×10^{-3}

2.2 b

Residual and Visualisation Python Code can be found in Appendix Problem 2, Part (b).

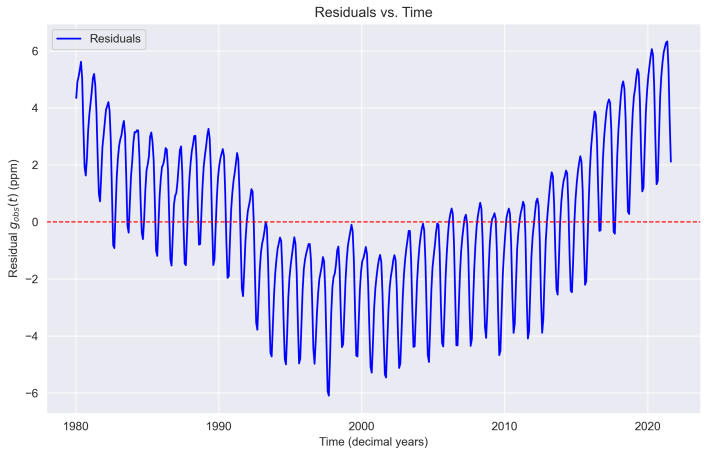


Figure 1: $g_{obs}(t)$

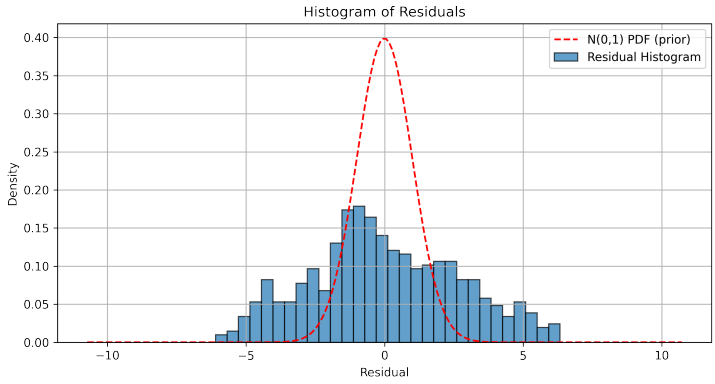


Figure 2: Density Estimation of the Residuals against $\epsilon(t)$

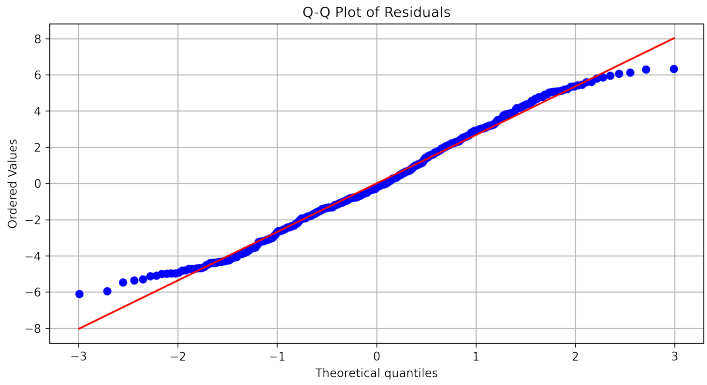


Figure 3: Q-Q plot to check normality

The residuals, defined as

$$g_{\text{obs}}(t) = f_{\text{obs}}(t) - (a_{\text{MAP}}t + b_{\text{MAP}}),$$

represent the difference between the observed CO_2 concentrations and the predicted mean function values. We analyzed whether these residuals conform to the prior assumption that $\epsilon(t) \sim \mathcal{N}(0, 1)$. Below are the key findings:

- **Gaussianity:** The residuals are approximately Gaussian, as evidenced by the Q-Q plot (Figure 3) and the Shapiro-Wilk test statistic of 0.9891 with a p-value of 0.0009. However, the variance of the residuals is 7.1881 (Figure 2), which deviates significantly from the prior assumption of unit variance ($\sigma^2 = 1$).
- **Mean:** The prior assumption of a mean of zero is reasonable, as the residuals are centered around zero (as shown in Figure 2).
- **Independence and Identical Distribution:** Observing $g_{\text{obs}}(t)$ reveals a periodic structure in the residuals, indicating that the data are not independently and identically distributed. If $\epsilon(t) \sim \mathcal{N}(0, 1)$ held true, $g_{\text{obs}}(t)$ would resemble random noise.

Based on these findings, the residuals partially conform to the prior assumptions. While Gaussianity and the mean of zero align with the prior, the variance and lack of independence indicate deviations from the prior belief.

2.3 c and d

we were given a periodic rbf kernel:

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

our function to generate samples drawn from a GP is:

$$\mathbf{f} \sim \mathcal{N}(0, \mathbf{K})$$

where $\mathbf{K}$ is the gram matrix with entries $K_{ij} = k(t_i, t_j)$. To draw a sample from this GP, our function first compute the gram matrix $\mathbf{K}$ and draw a sample $\mathbf{f}$ from $\mathcal{N}(0, \mathbf{K})$. The code implementation of our function is given below, the full code implementation can be found at Appendix Problem 2, Part (c and d).

```
@jax.jit
def compute_covariance_matrix(x, params):
    theta, tau, sigma, phi, eta, zeta = params
    K = kernel(x, x, theta, tau, sigma, phi, eta, zeta)
    return K

@jax.jit
def gp_sample(key, x, K):
    """
    Draw a function sample from a GP with mean zero and covariance K.
    This function returns one sample vector f ~ N(0, K).
    """
    N = len(x)
    mean = jnp.zeros(N)
    # Add a small jitter for numerical stability in Cholesky
    L = jnp.linalg.cholesky(K + 1e-9 * jnp.eye(N))
    eps = random.normal(key, shape=(N,))
    return mean + L @ eps
# example usage:
# Define input points
t = jnp.linspace(0, 20, 1000)

# Compute covariance matrix
K = compute_covariance_matrix(t, params)

# Generate three samples from the GP
key = random.PRNGKey(42)
key1, key2, key3 = random.split(key, 3)
f_sample1 = gp_sample(key1, t, K)
f_sample2 = gp_sample(key2, t, K)
f_sample3 = gp_sample(key3, t, K)
```

Listing 1: GP sample function snippet

We plot the samples from this function from a GP with the identical covariance matrix and zero mean under a arbitrary default Hyper-parameter set$\theta = 2.5, \tau = 5, \sigma = 1.26, \phi = 0.5, \eta = 1.26, \zeta = 0.01$:

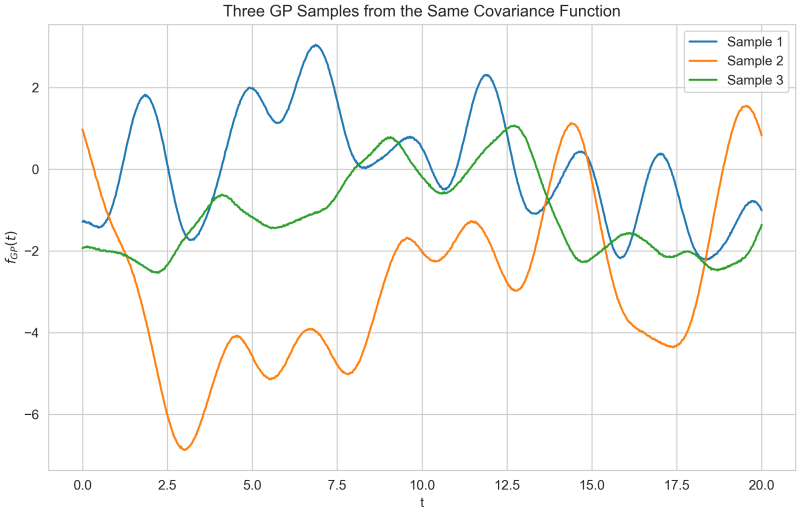


Figure 4: Samples from a zero mean GP with the provided kernel

we also plot the gram matrix to take a qualitative look of the given kernel with default Hyper-parameters:

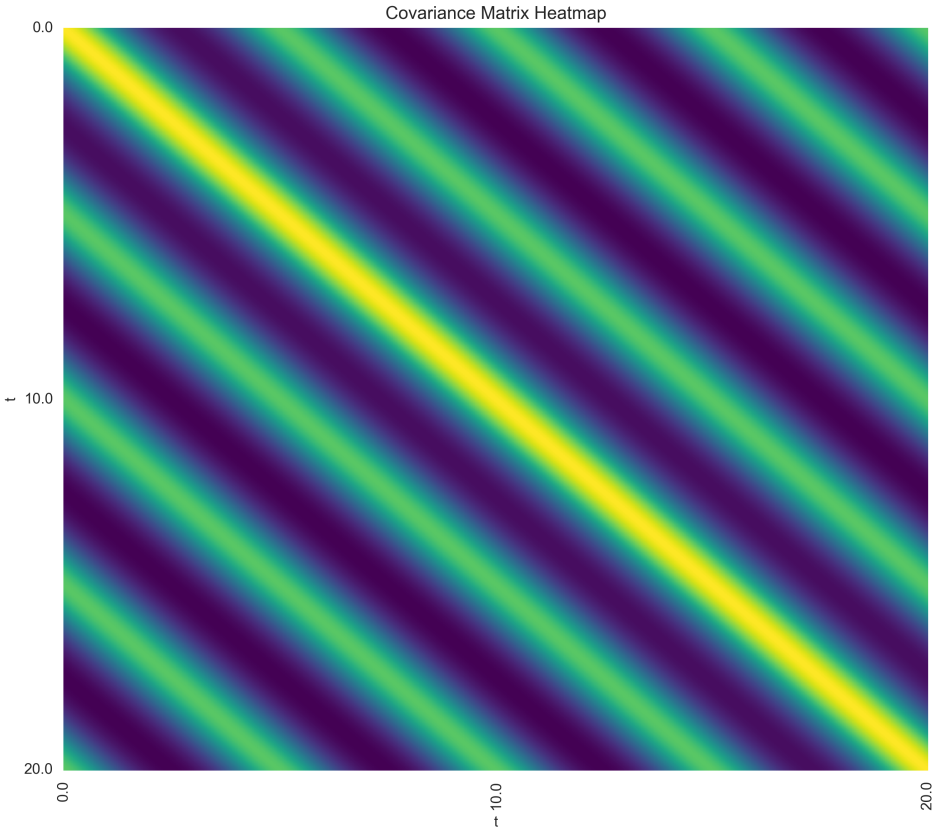


Figure 5: Gram Matrix

Based on the samples and the Gram matrix, the drawn functions exhibit a striped pattern, which indicates higher covariance at regular intervals. This regular covariance structure is a direct result of the sinusoidal component of the kernel, which superimposes a repeating seasonal pattern onto the functions. Additionally, the covariance values decay as points move further away from the diagonal of the Gram matrix. This decay is attributed to the squared-exponential term in the kernel, which enforces local correlations by ensuring that points closer in time are more strongly related, while those further apart become less correlated. Together, these components make the kernel well-suited for modeling $g(t)$, as the visualizations of $g(t)$ suggest it exhibits both periodic (seasonal) behavior and local smoothness. The periodic term captures the annual seasonal fluctuations, and the squared-exponential term ensures that the function remains smooth and locally correlated over time.

We do a hyper-parameter sweep on the default Hyper-parameters to have a qualitative view of how changing each of them will affect the characteristics of the function.

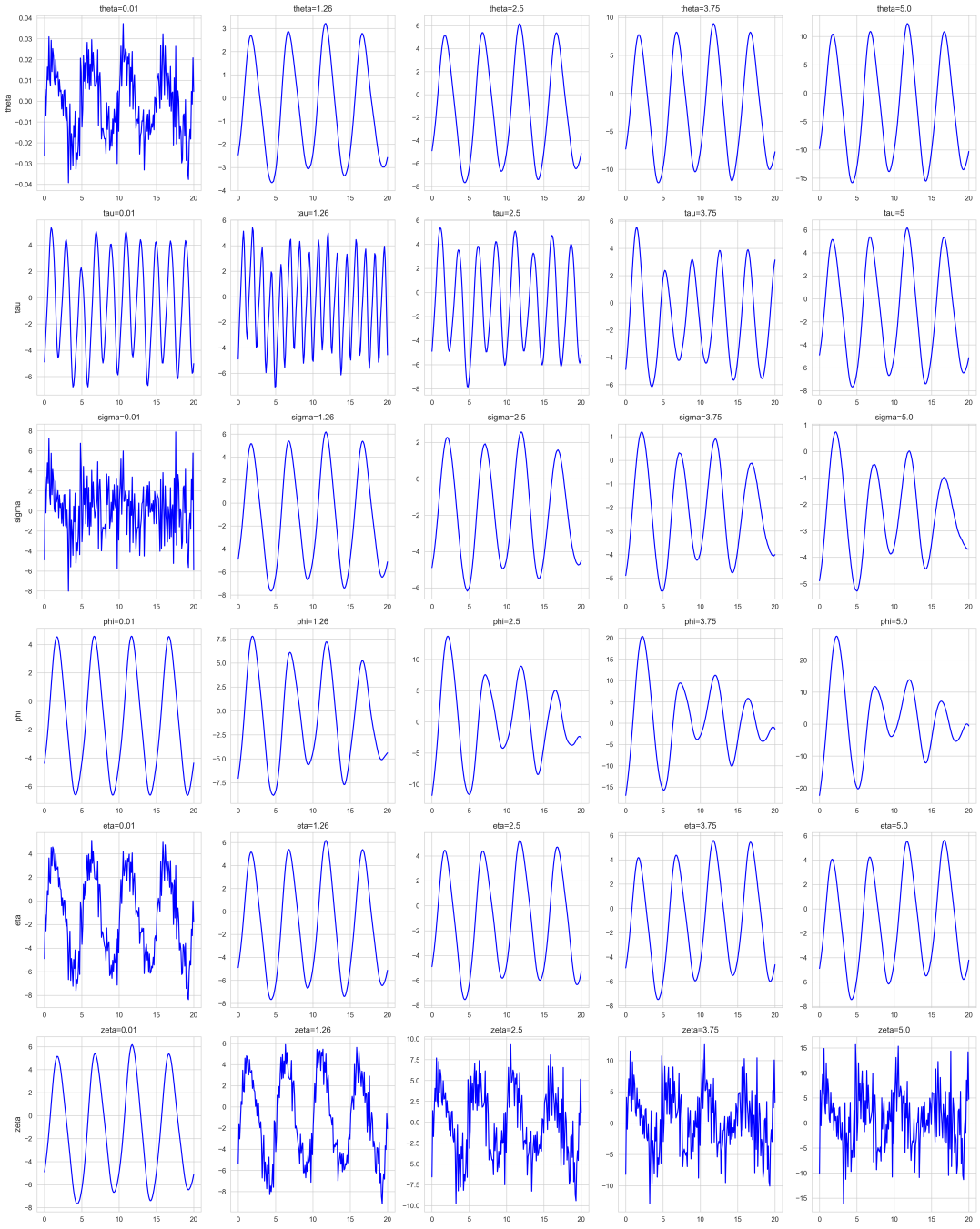


Figure 6: Samples under different changed parameters

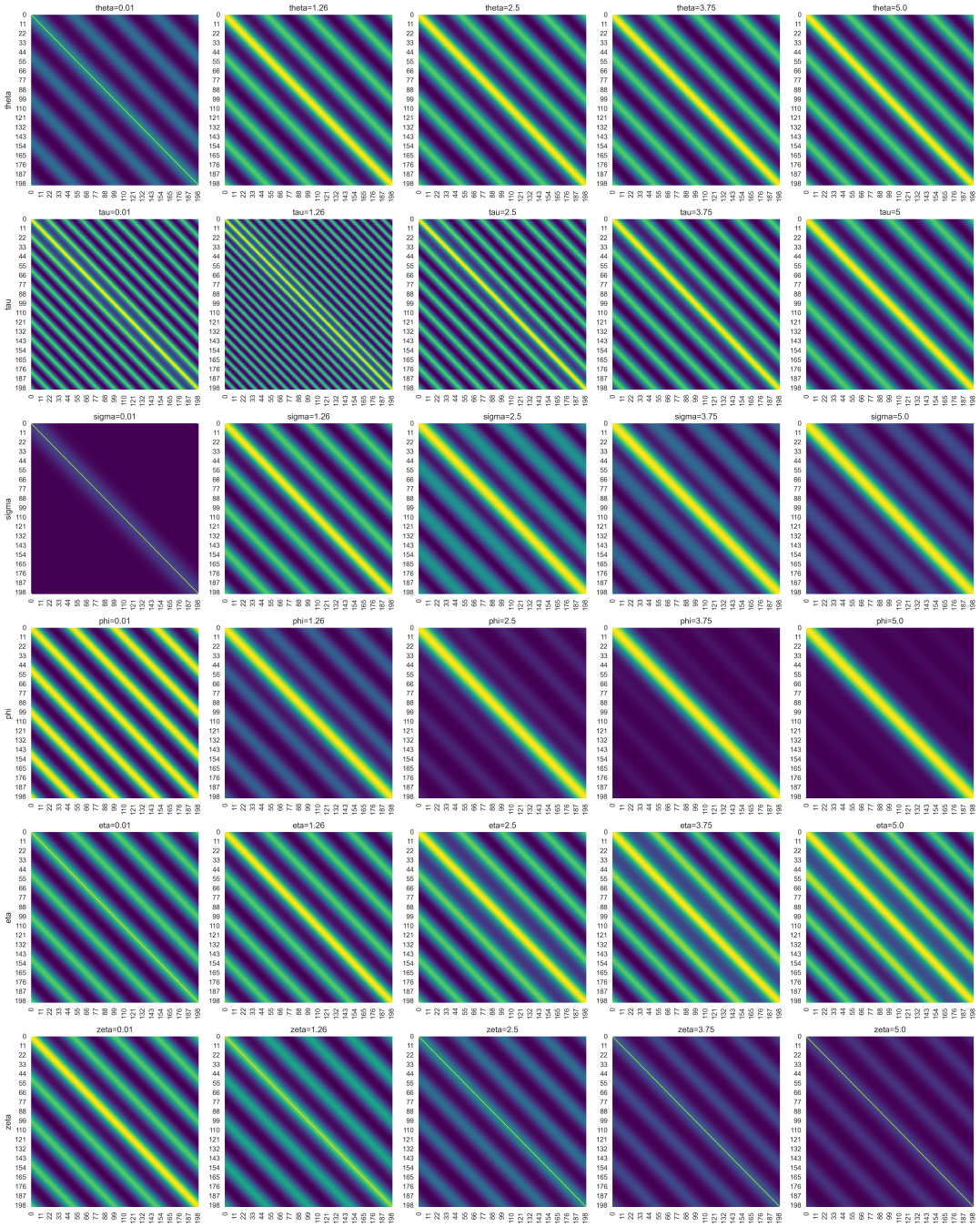


Figure 7: Gram Matrix under different changed parameters

- Overall scale (θ): The parameter θ controls the overall amplitude of the variability. Increasing θ leads to functions with greater vertical variation, making peaks and troughs more pronounced. Decreasing θ results in flatter functions that vary less about the mean.

- Period (τ): The period τ determines the fundamental timescale of the repeating structure. With a smaller τ, the drawn functions oscillate more frequently, resulting in higher-frequency seasonal patterns. Conversely, a larger τ stretches out the periodic cycle, resulting in longer, more gently repeating patterns over time.

- Periodic length-scale (σ): The parameter σ governs how sharply the periodic kernel decays within one period. A small σ yields sharper, more rapidly changing oscillations and more intricate patterns within each period. Larger σ values smooth out the periodic structure, yielding gentler, more sinusoidal-like functions.

- squared-exponential(SE) component contribution (ϕ) and SE length-scale (η): The parameter ϕ determines the relative strength of the non-periodic, "locally smooth" component of the kernel. Larger values of ϕ increase the prominence of the SE portion, resulting in overall smoother, more slowly varying trends around the periodic fluctuations. The length-scale η associated with the SE kernel controls how quickly correlation decays with distance. Smaller η values produce functions with more rapid local changes, while larger η values yield globally smoother functions that change more slowly over time.

- Diagonal noise variance (ζ^2): The noise term ζ^2 adds independent variance to each observed time point, ensuring that individual samples will not lie perfectly on a smooth curve. Increasing ζ introduces more jitter and small-scale irregularities in the sampled functions, reducing the visually smooth character of the functions. Decreasing ζ yields more deterministic-looking samples

2.4 e

To model the residual function $g(t)$ using a zero-mean Gaussian Process (GP) with the specified covariance kernel, we integrate prior knowledge with visual inspection of the data to select suitable hyperparameters. Given that CO_2 concentration exhibits yearly oscillations due to periodic changes in temperature, humidity, and other factors, we set the periodicity parameter $\tau = 1$ year to reflect this seasonality. The full code implementation can be found at the GP bit of Appendix Problem 2, Part (e and f). .

Based on the plot of the residual function $g(t)$ and our previous explorations, we propose the following initial hyperparameter values:

$$\theta = 3.0, \quad \tau = 1.0, \quad \sigma = 1.0, \quad \phi = 0.5, \quad \eta = 2.0, \quad \zeta = 0.2.$$

Justification for the Initial Hyperparameters:

- $\tau = 1.0$: Represents the yearly period of oscillation, aligning with the seasonal changes.
- $\theta = 3.0$: The residual GP function exhibits a moderately large scale, and a value of 3 effectively captures this behavior (we can read this from the hyperparamter sweep plots).
- $\sigma = 1.0$: The residual is relatively smooth, so setting $\sigma = 1$ ensures appropriate smoothness in the model.
- $\phi = 0.5$ and $\eta = 2.0$: These parameters control the squared exponential (SE) component of the kernel. Observing the residual density trend between 2-3, these values appropriately capture the trend.
- $\zeta = 0.2$: This parameter governs the variance of the random noise, and a low value of 0.2 reflects the relatively low noise in the data.

However, obtaining more accurate estimates through visual inspection alone is challenging. Therefore, we perform Maximum Likelihood Estimation (MLE) by minimizing the Negative Log-Likelihood (NLL):

$$\text{NLL} = \frac{1}{2}\mathbf{y}^\top \mathbf{K}^{-1}\mathbf{y} + \frac{1}{2}\log|\mathbf{K}| + \frac{n}{2}\log(2\pi).$$

After training, the optimized hyperparameters are presented in Table 3. The results indicate that our initial guesses were reasonably close to the learned values, thereby justifying our initial choices (the code for the MLE is also included the full code implementation can be found at the MLE bit of Appendix Problem 2, Part (e and f).

Table 3: Initial and Optimized Hyperparameters for the GP Kernel

Parameter	**Initial Value**	**Optimized Value**
θ	3.0	2.68
σ	1.0	1.75
ϕ	0.5	0.62
η	2.0	1.15
ζ	0.2	0.18
τ	1.0	1.00

2.5 f [Bonus]

The full code implementation can be found at Appendix Problem 2, Part (e and f). We showed the result for both trained and untrained hyperparamter sets here although they look quite similar

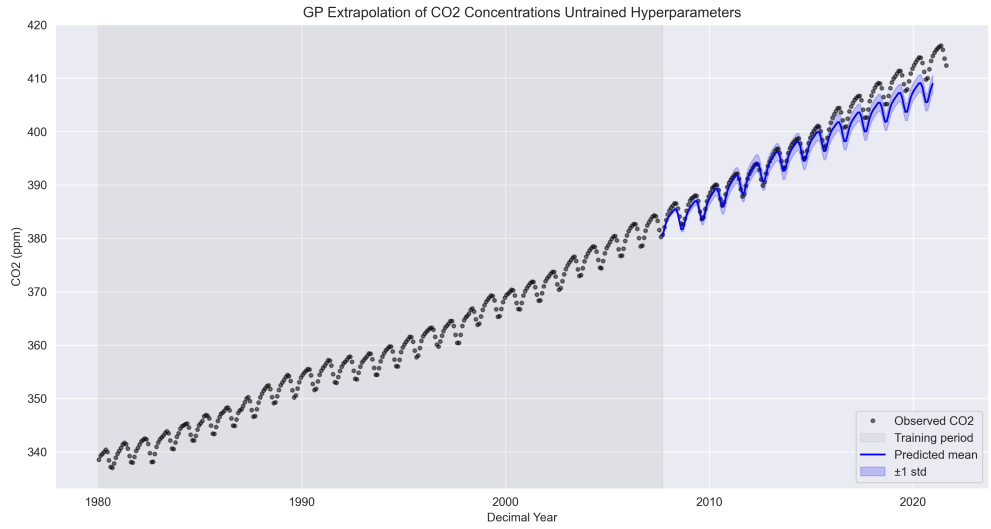


Figure 8: Samples under different changed parameters

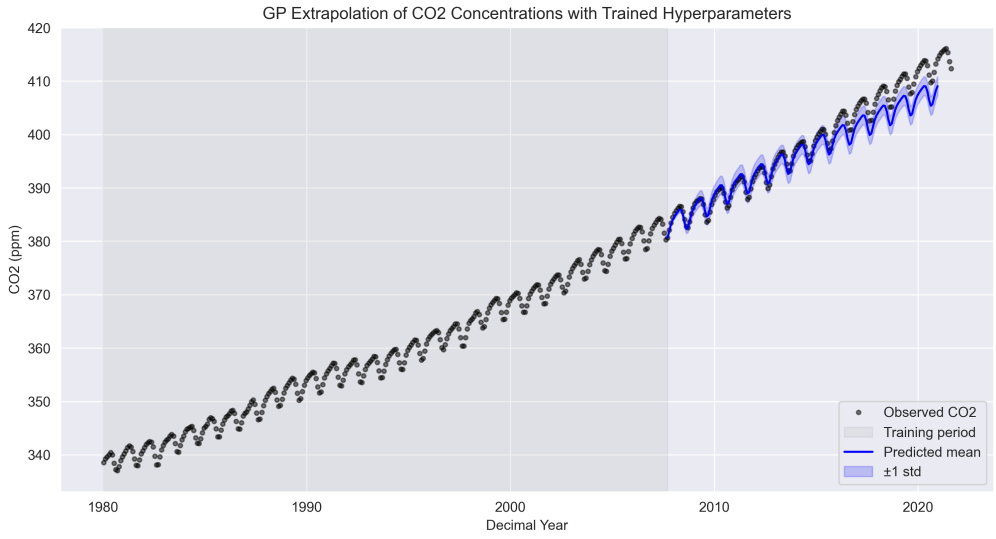


Figure 9: Samples under different changed parameters

From the figures we can see that the GP extrapolation of CO_2 concentration levels up to December 2020 demonstrates a continued upward trend, which aligns with historical observations of steadily increasing atmospheric CO_2 This trend is consistent with our Bayesian linear regression model assumptions, where the mean function captures the long-term rise in CO_2 concentrations.

Predictive Mean and Variance: The extrapolated CO levels exhibit not only a persistent increase but also maintain the yearly seasonal variability observed in the historical data. This seasonal pattern is effectively modeled through the covariance kernel K chosen for the residuals $g(t)$, which incorporates periodic components to capture such fluctuations.

Model Behavior and Expectations: Overall, the behavior of the extrapolation conforms to expectations. The predictions are smooth and continuous, reflecting the ongoing increase in CO_2 levels.

Additionally, the predicted periodic behaviors mirror the historical data, indicating that the model successfully captures seasonal variations. Notably, the standard deviation (error bars) of the predictions increases slightly as the forecast extends further into the future. This increase in uncertainty is expected, as predictions become less certain the further they are from the last observed data point.

Sensitivity to Kernel Hyperparameters: After testing multiple sets of hyperparameters, we observe that the conclusions drawn from extrapolation are actually quite sensitive to the choice of kernel hyperparameters. If we initialize with a carefully selected set of hyperparameters, they reflect historical data well and make reasonable predictions. However, using randomly chosen hyperparameters often leads to extrapolated predictions with large uncertainties that fail to accurately represent the observed data. In some cases, MLE training does not converge. This underscores the importance of selecting appropriate hyperparameters.

2.6 g [Bonus]

The procedure is not fully Bayesian because it relies on a Maximum A Posteriori (MAP) point estimate for the linear regression parameters during prediction, which neglects the inherent uncertainty in these parameters. In a fully Bayesian framework, instead of using a single point estimate, we should integrate over the entire posterior distribution of the regression coefficients to account for their uncertainty in predictions. Additionally, the current approach only considers the uncertainty of $g(t)$ and fails to capture potential non-linear trends in the data, which should also be reflected in the extrapolation uncertainty. To model $f(t)$ in a Bayesian manner, one could incorporate a linear kernel into the combined kernel function, thereby directly modeling $f(t)$ with the kernel and integrating the uncertainties of all signal components. This would ensure that both the original data and new predictions are treated as random variables generated from their respective distributions, allowing for a comprehensive integration over all parameters and fully capturing the uncertainty in the model.

3 Mean-field learning

3.1 a

The full code implementation can be found at Appendix Problem 3, Part (a).

3.1.1 Recall the Model Definition

Consider a binary latent factor model characterized by:

- **Latent Variables**: $\mathbf{s} = (s_1, s_2, \ldots, s_K)$, where each $s_i \in \{0, 1\}$.
- **Observed Data**: $\mathbf{x} \in \mathbb{R}^D$.
- **Parameters**: $\theta = \{\mu_i, \pi_i\}_{i=1}^K, \sigma^2$, where $\mu_i \in \mathbb{R}^D$ and $\pi_i \in (0, 1)$.

The joint distribution of the latent variables and the observed data is defined as:

$$p(\mathbf{s} \mid \boldsymbol{\pi}) = \prod_{i=1}^{K} \pi_i^{s_i} (1 - \pi_i)^{1 - s_i}, \tag{1}$$

$$p(\mathbf{x} \mid \mathbf{s}, \mu, \sigma^2) = \mathcal{N}\left(\mathbf{x} \mid \sum_{i=1}^{K} s_i \mu_i, \sigma^2 \mathbf{I}\right), \tag{2}$$

where $\mathbf{I}$ is the $D \times D$ identity matrix.

Given a dataset $\mathcal{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}\}$ of N independent and identically distributed observations, the goal is to approximate the posterior distribution $p(\mathbf{s}^{(n)} \mid \mathbf{x}^{(n)}, \theta)$ for each data point $\mathbf{x}^{(n)}$.

3.1.2 Variational Approximation

To facilitate tractable inference, we employ a **mean-field variational approximation**. Specifically, for each data point $\mathbf{x}^{(n)}$, the posterior over the latent variables is approximated by a fully factorized distribution:

$$q_n(\mathbf{s}^{(n)}) = \prod_{i=1}^{K} \lambda_{in}^{s_i^{(n)}} (1 - \lambda_{in})^{1 - s_i^{(n)}}, \tag{3}$$

where $\lambda_{in} \in (0, 1)$ are the variational parameters to be optimized.

3.1.3 Free Energy

The Free Energy for the n-th data point is defined as:

$$\mathcal{F}_n = \mathbb{E}_{q_n(\mathbf{s}^{(n)})}\left[\log p(\mathbf{x}^{(n)}, \mathbf{s}^{(n)} \mid \theta)\right] - \mathbb{E}_{q_n(\mathbf{s}^{(n)})}\left[\log q_n(\mathbf{s}^{(n)})\right]. \tag{4}$$

Expanding the joint distribution:

$$\log p(\mathbf{x}^{(n)}, \mathbf{s}^{(n)} \mid \theta) = \log p(\mathbf{x}^{(n)} \mid \mathbf{s}^{(n)}, \mu, \sigma^2) + \log p(\mathbf{s}^{(n)} \mid \boldsymbol{\pi}). \tag{5}$$

Thus, the Free Energy becomes:

$$\mathcal{F}_n = \mathbb{E}_{q_n}\left[\log p(\mathbf{x}^{(n)} \mid \mathbf{s}^{(n)}, \mu, \sigma^2)\right] + \mathbb{E}_{q_n}\left[\log p(\mathbf{s}^{(n)} \mid \boldsymbol{\pi})\right] - \mathbb{E}_{q_n}\left[\log q_n(\mathbf{s}^{(n)})\right]. \tag{6}$$

3.1.4 Computing the Free Energy Components

1. Expectation of the Log Prior From Equation (1):

$$\mathbb{E}_{q_n}\left[\log p(\mathbf{s}^{(n)} \mid \boldsymbol{\pi})\right] = \mathbb{E}_{q_n}\left[\sum_{i=1}^{K} s_i^{(n)} \log \pi_i + (1 - s_i^{(n)}) \log(1 - \pi_i)\right] \tag{7}$$

$$= \sum_{i=1}^{K} \left[\lambda_{in} \log \pi_i + (1 - \lambda_{in}) \log(1 - \pi_i)\right]. \tag{8}$$

2. Expectation of the Log Likelihood From Equation (2):

$$\mathbb{E}_{q_n}\left[\log p(\mathbf{x}^{(n)} \mid \mathbf{s}^{(n)}, \mu, \sigma^2)\right] = -\frac{D}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\mathbb{E}_{q_n}\left[\|\mathbf{x}^{(n)} - \sum_{i=1}^{K} s_i^{(n)}\mu_i\|^2\right]. \tag{9}$$

Expanding the squared norm:

$$\|\mathbf{x}^{(n)} - \sum_{i=1}^{K} s_i^{(n)}\mu_i\|^2 = \|\mathbf{x}^{(n)}\|^2 - 2\mathbf{x}^{(n)\top}\sum_{i=1}^{K} s_i^{(n)}\mu_i + \left(\sum_{i=1}^{K} s_i^{(n)}\mu_i\right)^\top \left(\sum_{j=1}^{K} s_j^{(n)}\mu_j\right). \tag{10}$$

Taking expectations term-by-term:

$$\mathbb{E}_{q_n}\left[\|\mathbf{x}^{(n)} - \sum_{i=1}^{K} s_i^{(n)}\mu_i\|^2\right] = \|\mathbf{x}^{(n)}\|^2 - 2\mathbf{x}^{(n)\top}\sum_{i=1}^{K}\lambda_{in}\mu_i + \sum_{i=1}^{K}\lambda_{in}\mu_i^\top\mu_i + \sum_{i\neq j}\lambda_{in}\lambda_{jn}\mu_i^\top\mu_j \tag{11}$$

$$= \|\mathbf{x}^{(n)}\|^2 - 2\mathbf{x}^{(n)\top}\mathbf{M}^{(n)} + \left(\mathbf{M}^{(n)}\right)^\top\mathbf{M}^{(n)} + \sum_{i=1}^{K}\lambda_{in}(1-\lambda_{in})\|\mu_i\|^2, \tag{12}$$

where $\mathbf{M}^{(n)} = \sum_{i=1}^{K}\lambda_{in}\mu_i$.

Thus, the expectation of the log likelihood simplifies to:

$$\mathbb{E}_{q_n}\left[\log p(\mathbf{x}^{(n)} \mid \mathbf{s}^{(n)}, \mu, \sigma^2)\right] = -\frac{D}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\left(\|\mathbf{x}^{(n)}\|^2 - 2\mathbf{x}^{(n)\top}\mathbf{M}^{(n)} + \left(\mathbf{M}^{(n)}\right)^\top\mathbf{M}^{(n)} + \sum_{i=1}^{K}\lambda_{in}(1-\lambda_{in})\|\mu_i\|^2\right) \tag{13}$$

3. Entropy of the Variational Distribution The entropy term is:

$$\mathbb{E}_{q_n}\left[\log q_n(\mathbf{s}^{(n)})\right] = \mathbb{E}_{q_n}\left[\sum_{i=1}^{K} s_i^{(n)}\log\lambda_{in} + (1-s_i^{(n)})\log(1-\lambda_{in})\right] \tag{14}$$

$$= \sum_{i=1}^{K}\left[\lambda_{in}\log\lambda_{in} + (1-\lambda_{in})\log(1-\lambda_{in})\right]. \tag{15}$$

3.1.5 Combining the Free Energy Components

Substituting the computed expectations into the Free Energy:

$$\begin{aligned}\mathcal{F}_n = &\sum_{i=1}^{K}\left[\lambda_{in}\log\pi_i + (1-\lambda_{in})\log(1-\pi_i)\right] \\ &- \frac{D}{2}\log(2\pi\sigma^2) \\ &- \frac{1}{2\sigma^2}\left(\|\mathbf{x}^{(n)}\|^2 - 2\mathbf{x}^{(n)\top}\mathbf{M}^{(n)} + \left(\mathbf{M}^{(n)}\right)^\top\mathbf{M}^{(n)} + \sum_{i=1}^{K}\lambda_{in}(1-\lambda_{in})\|\mu_i\|^2\right) \\ &- \sum_{i=1}^{K}\left[\lambda_{in}\log\lambda_{in} + (1-\lambda_{in})\log(1-\lambda_{in})\right].\end{aligned} \tag{16}$$

3.1.6 Deriving the Mean-Field Update for λ_{in}

To optimize the Free Energy with respect to λ_{in}, we take the derivative of $\mathcal{F}_n$ with respect to λ_{in} and set it to zero:

$$\frac{\partial\mathcal{F}_n}{\partial\lambda_{in}} = 0. \tag{17}$$

Derivative of the Log Prior Expectation:

$$\frac{\partial}{\partial \lambda_{in}} \left[\lambda_{in} \log \pi_i + (1-\lambda_{in}) \log(1-\pi_i)\right] = \log \pi_i - \log(1-\pi_i). \tag{18}$$

Derivative of the Log Likelihood Expectation:

$$\frac{\partial}{\partial \lambda_{in}} \left[\sum_{j=1}^{K} \lambda_{jn}(1-\lambda_{jn})\|\mu_j\|^2\right] = \frac{\partial}{\partial \lambda_{in}} \left[\lambda_{in}(1-\lambda_{in})\|\mu_i\|^2\right] \tag{19}$$

$$= (1-\lambda_{in})\|\mu_i\|^2 - \lambda_{in}\|\mu_i\|^2 \tag{20}$$

$$= (1-2\lambda_{in})\|\mu_i\|^2. \tag{21}$$

Thus,

$$-\frac{1}{2\sigma^2}\frac{\partial}{\partial \lambda_{in}} \left[\sum_{j=1}^{K} \lambda_{jn}(1-\lambda_{jn})\|\mu_j\|^2\right] = -\frac{1}{2\sigma^2}(1-2\lambda_{in})\|\mu_i\|^2 \tag{22}$$

$$= -\frac{1}{2\sigma^2}\|\mu_i\|^2 + \frac{\lambda_{in}}{\sigma^2}\|\mu_i\|^2. \tag{23}$$

Derivative of the Entropy Term:

$$\frac{\partial}{\partial \lambda_{in}} \left[\lambda_{in} \log \lambda_{in} + (1-\lambda_{in}) \log(1-\lambda_{in})\right] = \log \lambda_{in} + 1 - \log(1-\lambda_{in}) - 1 \tag{24}$$

$$= \log \lambda_{in} - \log(1-\lambda_{in}). \tag{25}$$

Combine All Derivatives
Summing all the partial derivatives:

$$\begin{aligned} \frac{\partial \mathcal{F}_n}{\partial \lambda_{in}} &= \log \pi_i - \log(1-\pi_i) \\ &\quad - \frac{1}{2\sigma^2}\|\mu_i\|^2 + \frac{\lambda_{in}}{\sigma^2}\|\mu_i\|^2 \\ &\quad - \log \lambda_{in} + \log(1-\lambda_{in}). \end{aligned} \tag{26}$$

Setting the derivative to zero for optimization:

$$\log \pi_i - \log(1-\pi_i) - \frac{1}{2\sigma^2}\|\mu_i\|^2 + \frac{\lambda_{in}}{\sigma^2}\|\mu_i\|^2 - \log \lambda_{in} + \log(1-\lambda_{in}) = 0. \tag{27}$$

Collecting like terms:

$$\log \frac{\lambda_{in}}{1-\lambda_{in}} = \log \frac{\pi_i}{1-\pi_i} + \frac{1}{2\sigma^2}\|\mu_i\|^2 - \frac{\lambda_{in}}{\sigma^2}\|\mu_i\|^2. \tag{28}$$

Notice that the term $\|\mu_i\|^2$ can be related to the inner product with the data vector. However, to align with the standard update form, we proceed by expressing the equation in terms of the data and other latent variables.

Recall that:

$$\mathbf{M}^{(n)} = \sum_{j=1}^{K} \lambda_{jn}\mu_j.$$

Subtracting the contribution of i-th factor:

$$\mathbf{x}^{(n)} - \mathbf{M}^{(n)} + \lambda_{in}\mu_i = \mathbf{x}^{(n)} - \sum_{j \neq i} \lambda_{jn}\mu_j.$$

Therefore, the inner product can be rewritten as:

$$\mu_i^\top \left(\mathbf{x}^{(n)} - \mathbf{M}^{(n)} + \lambda_{in}\mu_i \right) = \mu_i^\top \left(\mathbf{x}^{(n)} - \sum_{j \neq i} \lambda_{jn}\mu_j \right). \tag{29}$$

Substituting back into the equation:

$$\log \frac{\lambda_{in}}{1-\lambda_{in}} = \log \frac{\pi_i}{1-\pi_i} + \frac{1}{\sigma^2}\mu_i^\top \left(\mathbf{x}^{(n)} - \frac{\mu_i}{2} - \sum_{j \neq i} \lambda_{jn}\mu_j \right). \tag{30}$$

Exponentiating both sides to solve for λ_{in}:

$$\frac{\lambda_{in}}{1-\lambda_{in}} = \exp\left(\log \frac{\pi_i}{1-\pi_i} + \frac{1}{\sigma^2}\mu_i^\top \left(\mathbf{x}^{(n)} - \frac{\mu_i}{2} - \sum_{j \neq i} \lambda_{jn}\mu_j \right) \right). \tag{31}$$

Finally, solving for λ_{in}:

$$\lambda_{in} = \frac{1}{1 + \exp\left(- \left[\log \frac{\pi_i}{1-\pi_i} + \frac{1}{\sigma^2}\mu_i^\top \left(\mathbf{x}^{(n)} - \frac{\mu_i}{2} - \sum_{j \neq i} \lambda_{jn}\mu_j \right) \right] \right)}. \tag{32}$$

Recognizing the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$, we express the update equation as:

$$\lambda_{in} = \sigma\left(\log \frac{\pi_i}{1-\pi_i} + \frac{1}{\sigma^2}\mu_i^\top \left(\mathbf{x}^{(n)} - \frac{\mu_i}{2} - \sum_{j \neq i} \lambda_{jn}\mu_j \right) \right). \tag{33}$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function.

3.1.7 Algorithm Summary for Clearity

The mean-field variational inference algorithm proceeds as follows for each data point $\mathbf{x}^{(n)}$:

1. **Initialization**: Set initial values $\lambda_{in}^{(0)}$ for all $i \in \{1, \ldots, K\}$.
2. **Iterative Updates**: For each iteration $t = 1, 2, \ldots, T$:
   (a) Compute the mean vector:
   $$\mathbf{M}^{(n)} = \sum_{i=1}^{K} \lambda_{in}^{(t-1)} \mu_i.$$
   (b) Update each λ_{in} using:
   $$\lambda_{in} = \sigma\left(\log \frac{\pi_i}{1-\pi_i} + \frac{1}{\sigma^2}\mu_i^\top \left(\mathbf{x}^{(n)} - \frac{\mu_i}{2} - \sum_{j \neq i} \lambda_{jn}\mu_j \right) \right).$$
3. **Convergence Check**: Terminate the iterations if the change in Free Energy $\mathcal{F}_n$ is below a predefined threshold ϵ, $|\mathcal{F}_n^{(t)} - \mathcal{F}_n^{(t-1)}| < \epsilon$ or if the maximum number of iterations T is reached.

This iterative procedure updates the variational parameters λ_{in} to maximize the Free Energy, thereby approximating the posterior distribution $p(\mathbf{s}^{(n)} \mid \mathbf{x}^{(n)}, \theta)$.

3.2 b

The M-step solution for updating the mean parameters μ in our mean-field variational inference framework is similar to the ordinary least squares (OLS) estimator in linear regression. Specifically, the update rule derived for μ:

$$\mu = \left(\left\langle \mathbf{s}\mathbf{s}^\top \right\rangle_{q(\mathbf{s})} \right)^{-1} \langle \mathbf{s} \rangle_{q(\mathbf{s})} \mathbf{X}, \tag{34}$$

is analogous to the OLS solution:

$$\hat{\beta} = (\mathbf{X}\mathbf{X}^\top)^{-1}\mathbf{X}\mathbf{Y}, \tag{35}$$

where β represents the regression coefficients, $\mathbf{X}$ is the design matrix, and $\mathbf{Y}$ is the response vector.

Correspondence Between Parameters In our latent factor model:

- The design matrix $\mathbf{X}$ in linear regression corresponds to the latent variable matrix $\mathbf{s}$, where each entry s_{kn} represents the activation of latent factor k for data point n.
- The response vector $\mathbf{Y}$ aligns with the observed data $\mathbf{X}$, where each data point $\mathbf{x}^{(n)}$ is modeled as a linear combination of the latent features.
- The regression coefficients β are analogous to the mean parameters μ, which serve as the weights for the latent factors in generating the observed data.

Probabilistic Interpretation While linear regression provides point estimates for the coefficients by minimizing the squared error between predicted and actual responses, the M-step in our model operates within a probabilistic framework. Here, the update for μ maximizes the expected log-likelihood (Free Energy) with respect to μ, taking into account the uncertainty in the latent variables through their expected values $\langle \mathbf{s} \rangle_{q(\mathbf{s})}$ and their covariances $\left\langle \mathbf{s}\mathbf{s}^\top \right\rangle_{q(\mathbf{s})}$.

This probabilistic approach allows the model to capture latent structures and dependencies that are not explicitly modeled in standard linear regression. Consequently, the M-step not only aligns with the mathematical structure of linear regression but also extends its applicability by incorporating latent variable expectations, thereby enhancing the model's ability to represent complex data relationships.

3.3 c

The computational complexity of the `m_step` function is primarily influenced by the dimensions of the input matrices: the number of data points N, the dimensionality D of each data point, and the number of latent factors K. Assuming that the input ESS is provided as a $K \times K$ matrix, the complexity analysis proceeds as follows:

1. Matrix Inversion

- Inversion of ESS: The function computes the inverse of the $K \times K$ matrix ESS, which has a computational complexity of $O(K^3)$.

2. Matrix Multiplications for μ

- First Multiplication ($\text{ESS}^{-1} \times \text{ES}^\top$): Multiplying a $K \times K$ matrix with a $K \times N$ matrix results in a $K \times N$ matrix. This operation has a complexity of $O(K^2 N)$.
- Second Multiplication ($(\text{ESS}^{-1} \times \text{ES}^\top) \times X$): Multiplying the resulting $K \times N$ matrix with an $N \times D$ matrix X yields a $K \times D$ matrix. The complexity of this operation is $O(K^2 D)$.
- Transpose Operation: Transposing the $K \times D$ matrix to obtain μ incurs negligible computational cost compared to matrix multiplications.

3. Computation of μ

- Combining the above steps, the total complexity for computing μ is:

$$O(K^3) + O(K^2 N) + O(K^2 D) = O(K^3 + K^2 N + K^2 D)$$

4. Computation of σ

- Trace of $X^\top X$: Computing $X^\top X$ involves multiplying an $D \times N$ matrix with an $N \times D$ matrix, resulting in a $D \times D$ matrix. The complexity is $O(ND^2)$.
- Trace of $\mu^\top \mu \times \text{ESS}$: First, $\mu^\top \mu$ (a $K \times D$ matrix multiplied by a $D \times K$ matrix) has a complexity of $O(DK^2)$. Multiplying the resulting $K \times K$ matrix with ESS adds another $O(K^3)$.
- Trace of $\text{ES}^\top X\mu$: Multiplying $\text{ES}^\top$ $(K \times N)$ with $X\mu$ $(N \times K)$ results in a $K \times K$ matrix, with a complexity of $O(NKD)$ followed by $O(DK^2)$ for the subsequent multiplication with μ.
- Final Trace Operations: Computing the trace of $K \times K$ matrices incurs a complexity of $O(K^2)$, which is negligible compared to the other terms.
- Overall Complexity for σ:

$$O(ND^2) + O(DK^2) + O(K^3) + O(NKD) + O(DK^2) + O(K^2) = O(ND^2 + NKD + DK^2 + K^3)$$

5. Computation of π

- Averaging over N: Calculating the mean of ES across the N data points involves summing $N \times K$ elements, resulting in a complexity of $O(NK)$.

6. Total Computational Complexity Combining all the components, the overall computational complexity of the `m_step` function is given by the sum of the individual complexities:

$$O(K^3 + K^2 N + K^2 D) + O(ND^2 + NKD + DK^2 + K^3) + O(NK) = O(ND^2 + NKD + DK^2 + K^3)$$

Justification

- Matrix Inversion ($O(K^3)$): Essential for solving the linear equations involved in updating μ.
- Matrix Multiplications ($O(K^2N)$ and $O(K^2D)$): Arise from the sequential multiplications required to compute μ.
- Trace Computations ($O(ND^2)$, $O(NKD)$, etc.): Result from operations needed to update σ, which involve multiple matrix multiplications and trace calculations.
- Averaging ($O(NK)$): Relatively minor compared to other operations but necessary for updating π.

Since none of N, K, or D is assumed to be significantly larger than the others, all terms contribute to the overall complexity. Therefore, the dominant factors in the computational complexity are ND^2, NKD, DK^2, and K^3, leading to a final complexity of:

$$O(ND^2 + NKD + DK^2 + K^3)$$

.

3.4 d

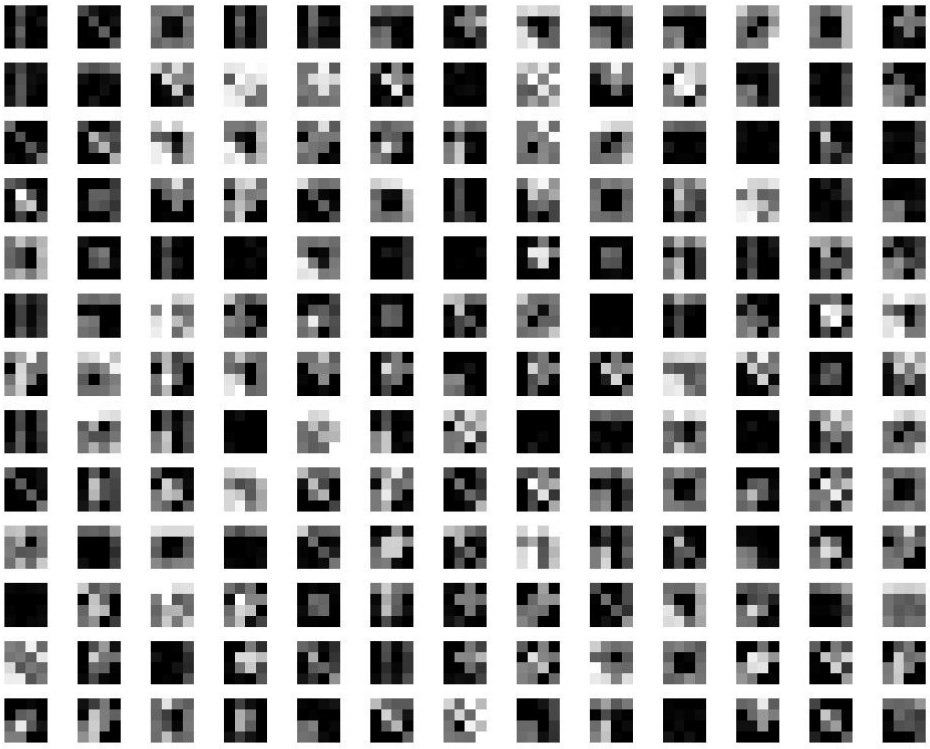


Figure 10: 100 images generated by randomly several features and adding a little noise

Considering the limited resolution of the images, the number of unique features is relatively small. We estimated about eight distinct features. Specifically, features might include:

- Horizontal lines in the first row
- Vertical lines in the second column
- Vertical lines in the fourth column
- Diagonal lines (top left to bottom right)
- 2×2 Squares in the centre
- 2×2 Squares in the bottom left corner
- Border
- 3×3 Crosses at the top right corner

Suitability of Factor Analysis (FA) Factor Analysis is designed to model observed variables as linear combinations of latent factors, assuming that the latent factors are continuous and normally distributed. The FA model is given as:

$$\mathbf{x} = \mathbf{W}\mathbf{s} + \epsilon$$

where $\epsilon \sim \mathcal{N}(0, \Psi)$ and $\mathbf{s} \sim \mathcal{N}(0, \mathbf{I})$. In this context, FA assumes Gaussianity in both the latent factors and the noise, which may not fully encapsulate the discrete activation of distinct features. Additionally, FA does not enforce independence between factors, potentially leading to correlated feature representations.

Conclusion: FA is not suitable for modeling this data. It can fall short in representing the independence and binary activation of features.

Suitability of Independent Component Analysis (ICA) Independent Component Analysis seeks to decompose data into statistically independent non-Gaussian components. It assumes model:

$$\mathbf{x} = \mathbf{W}\mathbf{s} + \epsilon$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2\mathbf{I})$ and $P(\mathbf{s}) = \prod_{k=1}^{K} P(s_k)$. In this scenario:

- **Pros**: ICA is adept at identifying independent underlying features, which aligns with the assumption that each feature (e.g., specific lines or shapes) operates independently in generating the images. This independence can lead to more interpretable feature representations.
- **Cons**: ICA typically requires non-Gaussianity in the latent factors, which may or may not hold true depending on the nature of the added noise and feature distributions.

Conclusion: ICA is highly suitable for this dataset as it aligns with the independent activation of underlying features, facilitating the discovery of distinct and interpretable components that compose each image.

Suitability of Mixture of Gaussians (MoG) The Mixture of Gaussians (MoG) model represents data as a combination of several Gaussian distributions, each corresponding to a distinct cluster or mode in the data space:

$$\mathbf{x} = \sum_{k=1}^{K} s_k \mu_k + \epsilon \quad \text{where} \quad \sum_{k=1}^{K} s_k = 1, \quad \epsilon \sim \mathcal{N}(0, \Sigma_\epsilon)$$

However, this approach assumes that each data point is generated from a single Gaussian component, which does not align with the generative process of our images. In our scenario, each image is a linear combination of multiple features with added noise, meaning that it inherently blends several Gaussian components rather than originating from just one. Additionally, MoG typically imposes a Dirichlet distribution on the mixing coefficients $\mathbf{s}$, whereas our data is constructed by independently sampling each s_k from a Bernoulli distribution to indicate the presence of feature μ_k.

These discrepancies indicate that the MoG model is inadequate for capturing the composite and feature-specific nature of our data.

Summary In summary, the dataset's generation process suggests a small number of distinct, independently activated features combined linearly with noise. Factor Analysis can model the linear relationships but may not fully capture feature independence. Independent Component Analysis, on the other hand, is more aligned with the data's underlying structure, offering better interpretability and representation of independent features. Mixture of Gaussians is less appropriate due to its inability to model the combinatorial and overlapping nature of feature contributions in image generation.

Model	**Suitability**
Factor Analysis (FA)	Not Suitable
Independent Component Analysis (ICA)	Highly Suitable
Mixture of Gaussians (MoG)	Not Suitable

Table 4: Evaluation of Modeling Techniques

3.5 e

The full code implementation can be found at Appendix Problem 3, Part (e, f, and g). Plot the free energy at each EM step to check if it increases after each iteration:

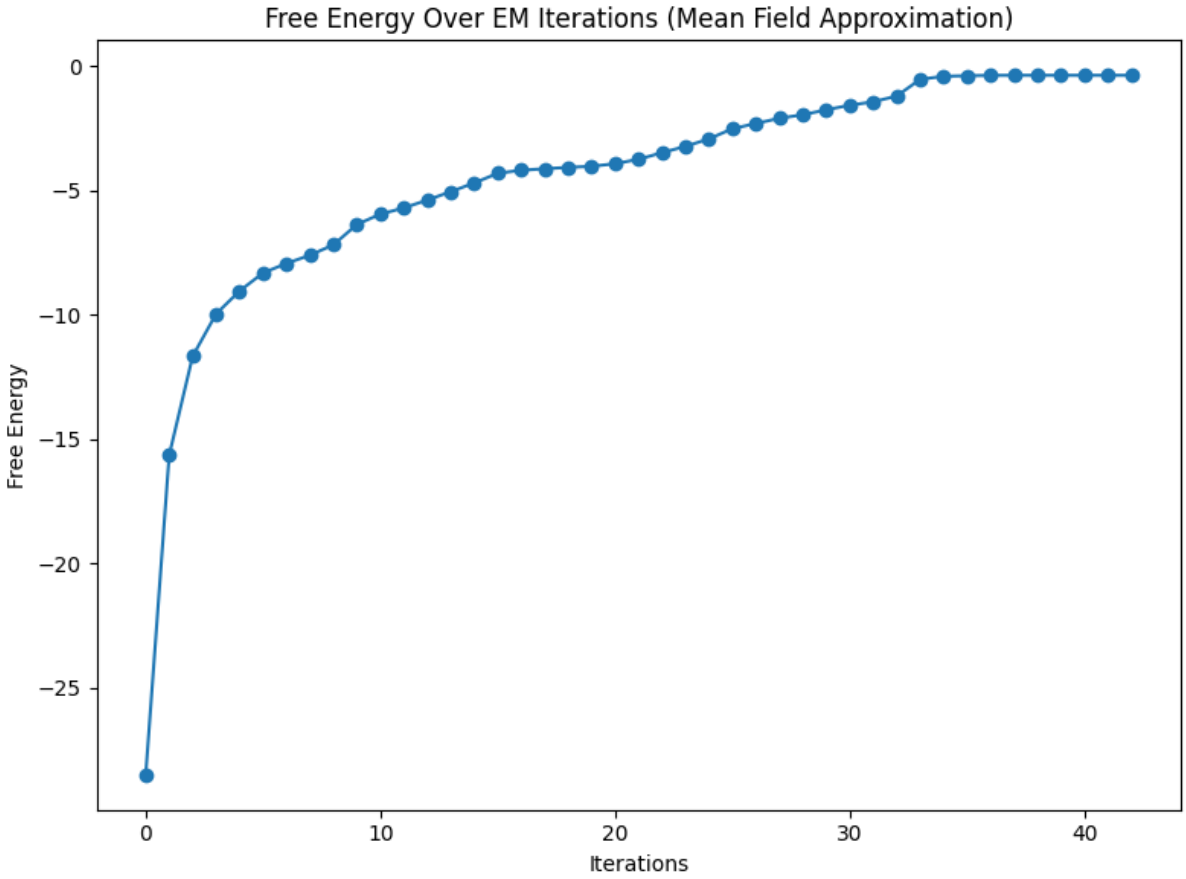


Figure 11: Free Energy in 100 iterations

3.6 f

The full code implementation can be found at Appendix Problem 3, Part (e, f, and g). Our 8 Latent Feature at Initialization: The Learned Latent Features After Training:

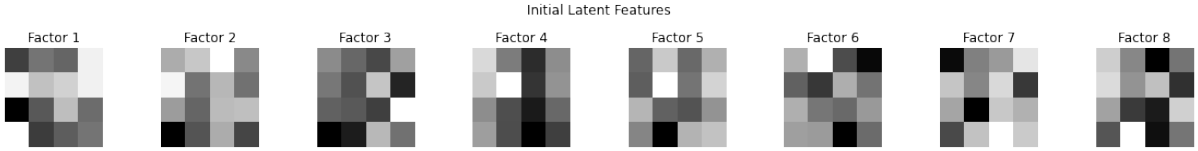


Figure 12: μ_i of the Latent Factors at Initialization

Figure 13: μ_i of the Latent Factors after Training

The algorithm successfully identified several distinct features, including a vertical line in the second column, a 2×2 square in the bottom left corner, a border, a horizontal line in the first row, and another 2×2 square in the center. Additionally, some features, such as μ_4, appear to be linear combinations of multiple basic features, like the cross and the central square.

To improve the algorithm's performance, several strategies can be employed:

- **Multiple Initializations**: Reinitializing the algorithm multiple times and selecting the model with the highest free energy can lead to better convergence outcomes.
- **Increased Data Samples**: Expanding the number of data samples may facilitate the learning of more robust and representative features.
- **Incorporating Priors**: Applying priors on π, σ^2, and μ can enhance parameter estimation through a Bayesian framework.
- **Mimic Kaiming Initialization**: A simple yet effective improvement involves initializing the model parameters using a strategy similar to Kaiming initialization. Unlike our initial approach where the mean-field parameters were sampled independently from a uniform distribution on $[0, 1]$, adopting a Kaiming-like initialization for π, σ, and μ can help maintain the variance across parameters, leading to more stable and efficient learning.

In the submitted implementation, the model parameters were initialized as follows:

- **Mean-Field Parameters**: Initialized using Kaiming-like initialization instead of sampling independently from a uniform distribution on $[0, 1]$.
- π: Initialized to a constant value of 0.5 for all factors
- σ: Initialized to 1.0,
- μ: Initialized using Kaiming initialization, specifically:

$$\mu_0 \sim \mathcal{N}(0, \sigma^2), \quad \sigma^2 = \frac{2}{D}$$

  This initialization scales the weights to preserve the variance of activations across parameters, enhancing computational stability and learning efficiency.

Furthermore, to ensure numerical stability during computations, particularly when calculating $\log(\lambda_i)$ and $\log(1 - \lambda_i)$, each λ_i was manually shifted by a small constant 1×10^{-10}. This adjustment prevents λ_i from being exactly 0 or 1, thereby avoiding undefined logarithmic values.

The number of factors K was set to eight, corresponding to the eight visually identified features. This setting was chosen based on prior analysis to align the model complexity with the underlying data structure.

3.7 g

The full code implementation can be found at Appendix Problem 3, Part (e, f, and g). It can be found after *"Analyze free energy with different sigma values"* in the **main** function.
The free energy at each partial E-step of the variational approximation for different σ

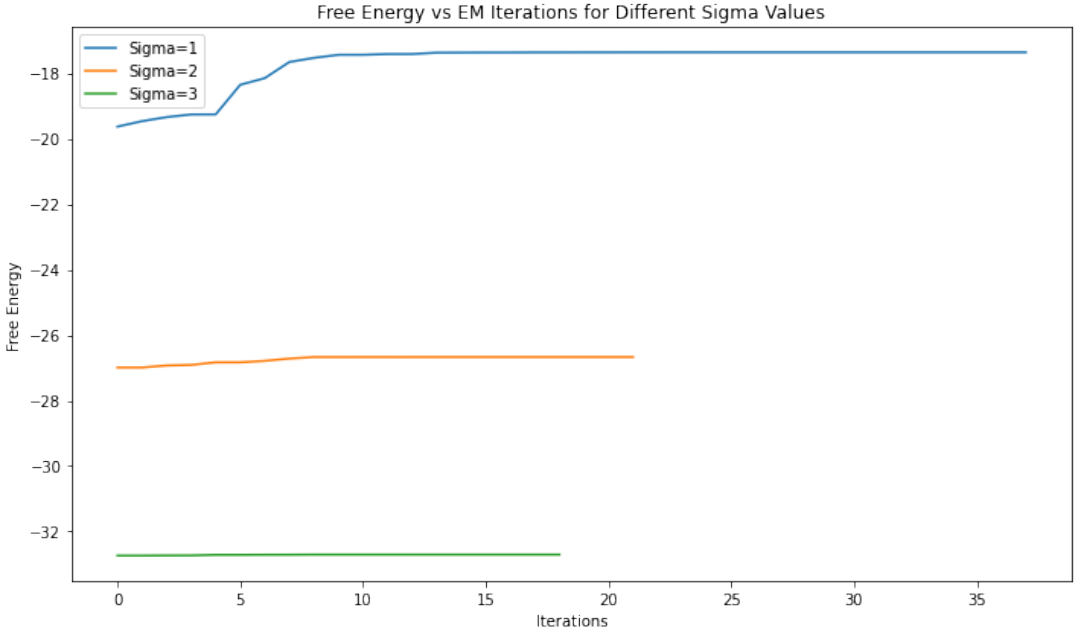


Figure 14: Free Energy with different entries of σ

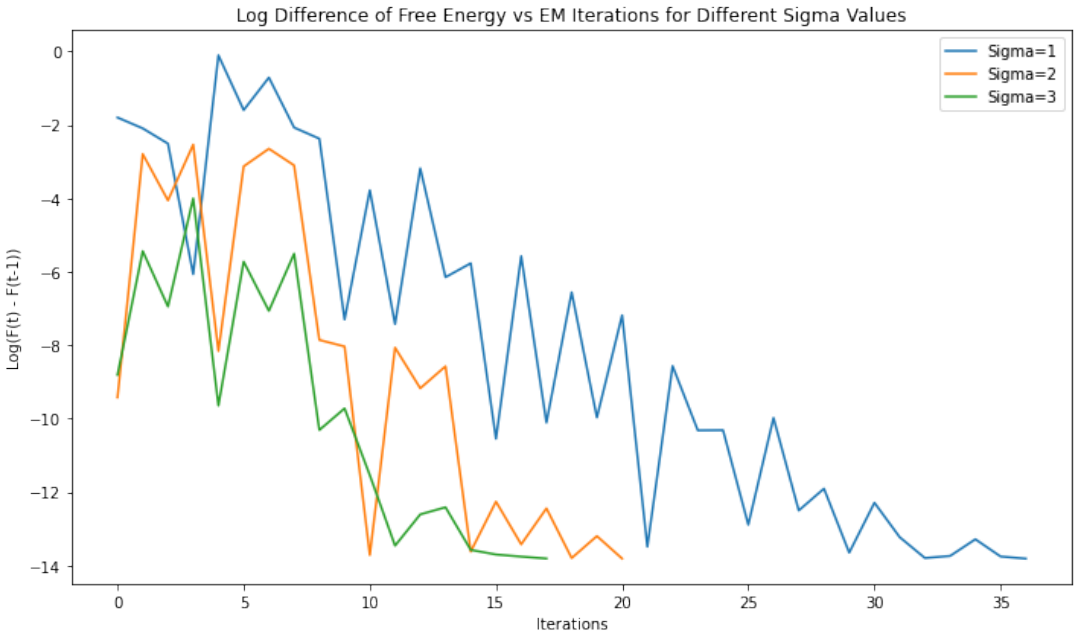


Figure 15: Convergence of Free Energy with different entries of σ

Our Free Energy is bounded by:

$$\log P(\mathcal{X}|\theta) \geq \mathcal{F}(q, \theta)$$

During the E-step, we aim to refine our approximation q (parameterized by λ) to maximize the free energy, $F(q, \theta)$. The generative parameters π, σ, and μ are fixed at this stage, and we optimize the variational parameters λ to bring the approximation q closer to the true posterior.

The parameter σ quantifies the noise in the observed data x. A higher σ indicates greater uncertainty, resulting in a wider spread of the likelihood $P(X \mid \theta)$. This increased uncertainty reduces the upper bound on $F(q, \theta)$ since the log-likelihood $\log P(X \mid \theta)$ is lower. As such, for larger values of σ, the free energy $F(q, \theta)$ converges to a lower value, reflecting the reduced upper bound imposed by the noisier data model.

From the plots of free energy F and its convergence rate, $\log(F(t) - F(t-1))$, we observe that increasing σ leads to faster convergence. This is because the lower upper bound reduces the search space for the variational parameters λ, enabling the algorithm to reach the maximum free energy more quickly. Conversely, smaller values of σ result in a higher upper bound and a slower convergence, as the algorithm has a larger search space and must fine-tune λ more precisely to maximize $F(q, \theta)$.

To investigate this, we ran the variational approximation for the first data point ($N = 1$), optimizing $q_1(s^{(1)})$. The results demonstrate the following:

- **Free Energy (F) as a Function of Iterations:** For larger σ, F converges to a lower value but does so in fewer iterations. For smaller σ, F achieves a higher value but requires more iterations.
- **Convergence Rate ($\log(F(t) - F(t-1))$):** The convergence rate is faster for larger σ, as the reduced upper bound simplifies the optimization process for λ.

4 Variational Bayes for binary factorsn [Bonus]

4.1 a

To derive a Variational Bayesian hyperparameter optimisation algorithm that automatically determines K, the number of hidden binary variables, we begin by considering our binary latent factor model:

$$s_i \sim \text{Bernoulli}(\pi_i), \quad i = 1, \ldots, K,$$

and for each observation $\mathbf{x} \in \mathbb{R}^D$:

$$\mathbf{x} \mid \mathbf{s}, \mu, \sigma^2 \sim \mathcal{N}\left(\sum_{k=1}^{K} s_k \mu_k,\ \sigma^2 I\right),$$

where $\mu = [\mu_1, \mu_2, \ldots, \mu_K] \in \mathbb{R}^{D \times K}$.

Writing μ in row form, let $\mathbf{w}_d \in \mathbb{R}^K$ be the d-th row of μ, so that:

$$x_d \mid \mathbf{s}, \mathbf{w}_d, \sigma^2 \sim \mathcal{N}(\mathbf{s}^T \mathbf{w}_d, \sigma^2).$$

To enable automatic relevance determination (ARD), we place a Gaussian prior on each $\mathbf{w}_d$:

$$p(\mathbf{w}_d \mid \alpha) = \mathcal{N}(\mathbf{0}, A^{-1}),$$

where $A = \text{diag}(\alpha)$ and $\alpha = [\alpha_1, \ldots, \alpha_K]^T$. Each α_k controls the precision of the k-th latent dimension. Additionally, we assign priors on π_i and a Gamma prior on each α_k:

$$\alpha_k \sim \text{Gamma}(a, b).$$

The joint distribution of the observed data, latent variables, and parameters is:

$$p(\mathbf{x}, \mathbf{s}, \mu \mid \pi, \sigma^2, \alpha) = p(\mathbf{s} \mid \pi) \prod_{d=1}^{D} p(x_d \mid \mathbf{s}, \mathbf{w}_d, \sigma^2) \prod_{d=1}^{D} p(\mathbf{w}_d \mid \alpha).$$

Taking the logarithm, we have:

$$\begin{aligned}
\log p(\mathbf{x}, \mathbf{s}, \mu \mid \pi, \sigma^2, \alpha) = &\sum_{k=1}^{K} [s_k \log \pi_k + (1 - s_k) \log(1 - \pi_k)] \\
&+ \sum_{d=1}^{D} \left(-\tfrac{1}{2} \log(2\pi\sigma^2) - \frac{(x_d - \mathbf{s}^T \mathbf{w}_d)^2}{2\sigma^2} \right) \\
&+ \sum_{d=1}^{D} \left(\tfrac{1}{2} \sum_{k=1}^{K} \log \alpha_k - \tfrac{1}{2} \mathbf{w}_d^T A \mathbf{w}_d \right).
\end{aligned}$$

We employ a mean-field variational approximation:

$$q(\mathbf{s}, \mu, \alpha) = q(\alpha) \prod_{i=1}^{K} q(\pi_i) \prod_{d=1}^{D} q(\mathbf{w}_d) \prod_{k=1}^{K} q_k(s_k).$$

For the latent variables $\mathbf{s}$, we let:

$$q(\mathbf{s}) = \prod_{k=1}^{K} q_k(s_k) = \prod_{k=1}^{K} \lambda_k^{s_k} (1 - \lambda_k)^{(1 - s_k)},$$

where $\lambda_k = q_k(s_k = 1)$.

Variational E-step for $q(\mathbf{s})$: To update $q(\mathbf{s})$, we write:

$$q(\mathbf{s}) \propto \exp\left(\mathbb{E}_{q(\mu, \alpha)} [\log p(\mathbf{x}, \mathbf{s}, \mu \mid \pi, \sigma^2, \alpha)] \right).$$

By comparing terms for $s_k = 1$ and $s_k = 0$, we obtain a logistic form for λ_k:

$$\lambda_k = \frac{1}{1 + \exp(-\eta_k)},$$

where

$$\eta_k = \frac{1}{\sigma^2} \langle \mu_k \rangle_{q(\mu_k)}^T \left(\mathbf{x} - \sum_{k' \neq k} \lambda_{k'} \langle \mu_{k'} \rangle_{q(\mu_{k'})} \right) - \frac{\langle \mu_k \rangle_{q(\mu_k)}^T \langle \mu_k \rangle_{q(\mu_k)}}{2\sigma^2} + \log \frac{\pi_k}{1 - \pi_k}.$$

Variational E-step for $q(\mathbf{w}_d)$: For each dimension d, the posterior $q(\mathbf{w}_d)$ is Gaussian due to the conjugacy of the Gaussian prior and Gaussian likelihood:

$$q(\mathbf{w}_d) = \mathcal{N}(\mu_{\mathbf{w}_d}, \Sigma_{\mathbf{w}_d}),$$

with

$$\Sigma_{\mathbf{w}_d}^{-1} = A + \frac{\langle \mathbf{s}\mathbf{s}^T \rangle_{q(\mathbf{s})}}{\sigma^2}, \quad \mu_{\mathbf{w}_d} = \Sigma_{\mathbf{w}_d} \frac{x_d \langle \mathbf{s} \rangle_{q(\mathbf{s})}}{\sigma^2}.$$

Since s_k are independent Bernoulli variables with parameters λ_k, we have:

$$\langle \mathbf{s} \rangle_{q(\mathbf{s})} = [\lambda_1, \ldots, \lambda_K]^T, \quad \langle \mathbf{s}\mathbf{s}^T \rangle_{q(\mathbf{s})} = \mathrm{diag}(\lambda_1, \ldots, \lambda_K).$$

Hyper-M-step for α: To automatically determine K, we leverage ARD by updating α:

$$\alpha_k \sim \mathrm{Gamma}(a, b).$$

The expected complete-data log posterior involving α is maximized w.r.t. α_k. After taking derivatives and setting to zero, we get:

$$\alpha_k = \frac{D + 2(a-1)}{2b + \sum_{d=1}^{D} [\mu_{\mathbf{w}_d}[k]^2 + \Sigma_{\mathbf{w}_d}(k,k)]}.$$

This update shrinks α_k when the k-th factor does not contribute significantly to explaining the data, effectively pruning that factor. Starting with a large K, many α_k may become large, pushing $\mathbf{w}_d$'s corresponding elements towards zero and removing unnecessary factors. Thus, ARD emerges naturally, allowing the model to determine the effective number of hidden variables K.

In summary:

1. Update $q(\mathbf{s})$ via logistic updates for each λ_k.
2. Update $q(\mathbf{w}_d)$ to a Gaussian by solving linear systems involving A and λ_k.
3. Update α_k using the ARD formula to prune redundant factors.

Through these iterative updates, factors that fail to explain the data are turned off, automatically determining the effective dimension K.

4.2 b

The full code implementation can be found at Appendix Problem 4, Part (b). Please note that while we could have refactored the code for conciseness, we need to prioritize clarity and maintain a self-contained format to enhance readability in the PDF so the full code could be a bit lengthy. First, The Free Energy of different Ks at each iterations: When we start with $K = 4$ latent features and increase K up to 8,

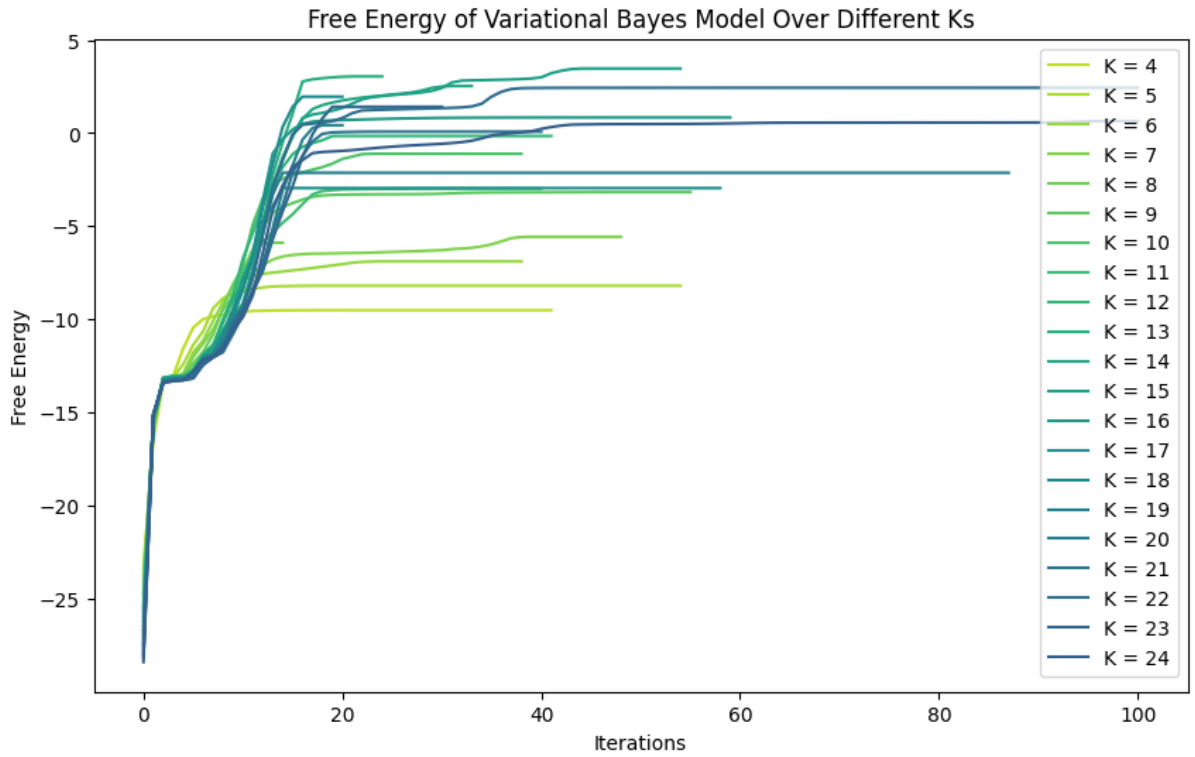


Figure 16: Free Energy for different values of Ks

we see a significant rise in the free energy at convergence, However as K continues to grow beyond 8, the free energy at convergence no longer follows a clear upward trend. This pattern is also seen in the inverse alpha α_k^{-1} plot below.

The two figures below show the learned features (μ_k) and their respective inverse α_k (for easier visualization) at convergence, which is the indicator for how relevant each factors is. (the first eight inverse α_k is showed in blue) As expected, when running the VB-ARD for a large number of features (high value of K), many of the values of α_k become very large ($\alpha_k \to \infty$), making $\alpha_k^{-1} \to 0$. We can also see a pattern of the most clear and relevant feature tends to have the highest α_k^{-1} while the irrelevant features appear to be noisy or duplicated.

Looking closer at the inverse alpha plots, when K reaches 12 or more, the number of relevant features stays roughly the same. Since we know there are only eight true latent features in the data, models with $K > 8$ are likely picking up duplicate or irrelevant features. Therefore, increasing K beyond eight doesn't lead to a significant increase in free energy because the model can't effectively use the extra features. Typically, models with much larger K values identify around ten or eleven relevant features, which may be due to slight overfitting, noise in the data, or duplicated features.

Overall, the relationship between free energy and the number of effective latent features behaves as expected with ARD in variational Bayes. This means that ARD successfully identifies the true number of relevant features, preventing the model from unnecessarily increasing complexity when more features are added.

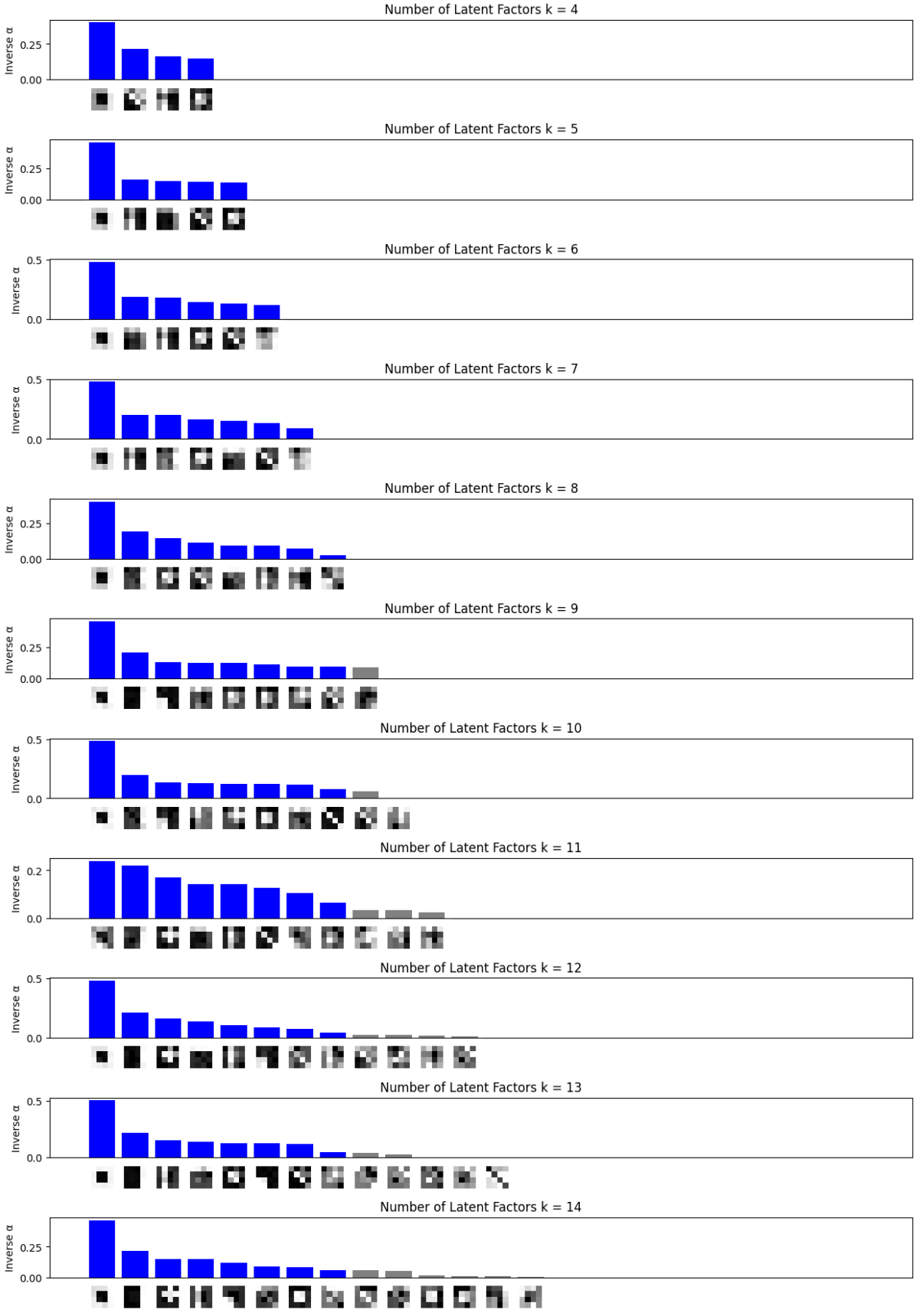


Figure 17: Learned Latent Factor at Convergence against their inverse alpha

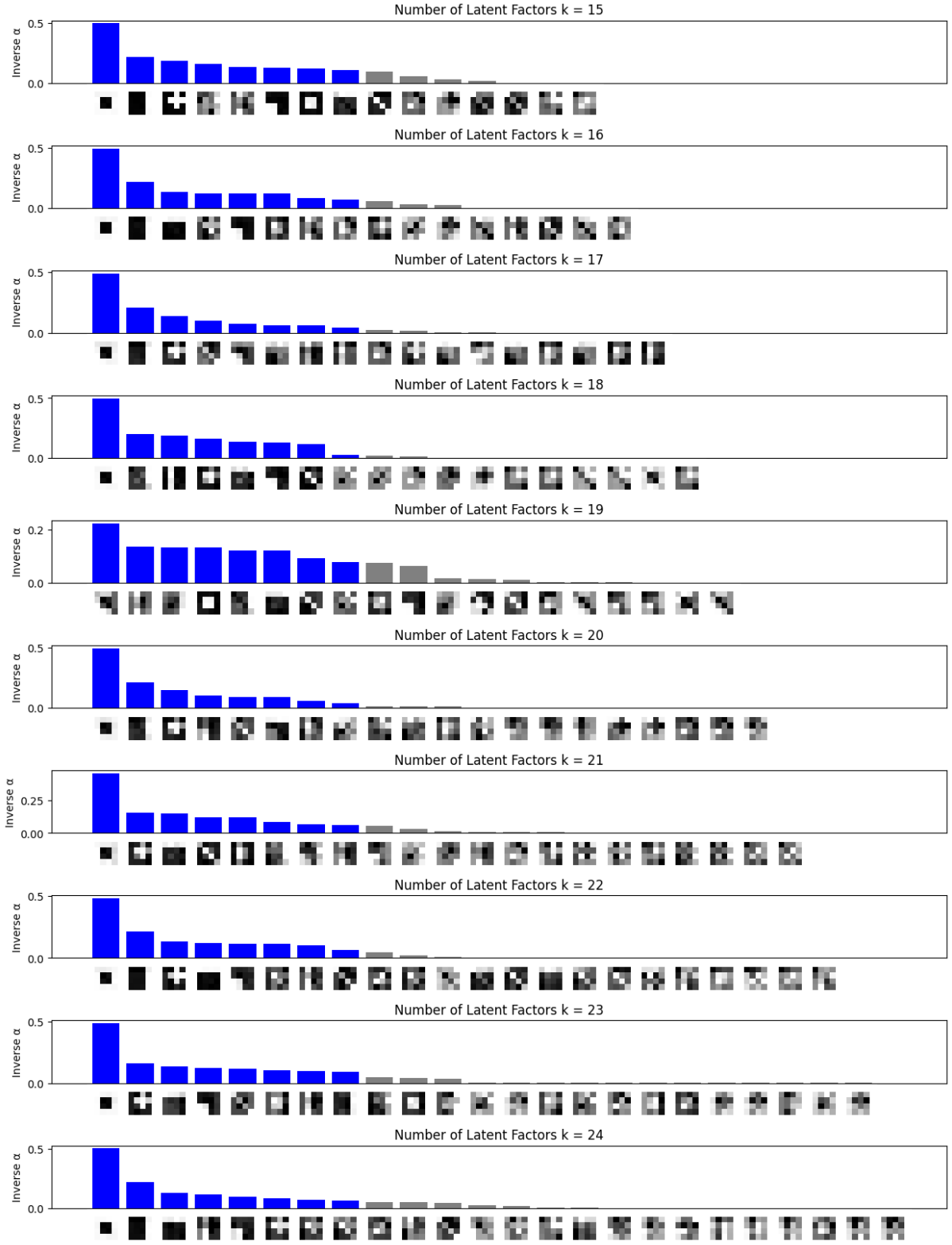


Figure 18: Learned Latent Factor at Convergence against their inverse alpha

5 EP for the binary factor model

5.1 a

Consider the binary latent factor model with binary latent variables $s = (s_1, \ldots, s_K)$, an observed real-valued vector x, and parameters $\theta = \{\mu_i, \pi_i\}_{i=1}^K, \sigma^2$. The joint probability for a single observation-source pair is given by:

$$\log p(s, x) = \log p(s \mid \pi) + \log p(x \mid s, \mu, \sigma^2).$$

Expanding the Prior Term $\log p(s \mid \pi)$ The prior distribution over the latent variables is:

$$p(s \mid \pi) = \prod_{i=1}^{K} \pi_i^{s_i} (1 - \pi_i)^{1 - s_i}.$$

Taking the logarithm, we obtain:

$$\log p(s \mid \pi) = \sum_{i=1}^{K} \left[s_i \log \pi_i + (1 - s_i) \log(1 - \pi_i) \right].$$

This can be rearranged as:

$$\log p(s \mid \pi) = \sum_{i=1}^{K} \log(1 - \pi_i) + \sum_{i=1}^{K} s_i \log \left(\frac{\pi_i}{1 - \pi_i} \right).$$

Ignoring the constant term $\sum_{i=1}^{K} \log(1 - \pi_i)$, we focus on the terms involving s_i:

$$\log p(s \mid \pi) = \sum_{i=1}^{K} s_i \log \left(\frac{\pi_i}{1 - \pi_i} \right).$$

Expanding the Likelihood Term $\log p(x \mid s, \mu, \sigma^2)$ The likelihood is defined as:

$$p(x \mid s, \mu, \sigma^2) = \mathcal{N} \left(x \mid \sum_{i=1}^{K} s_i \mu_i, \sigma^2 I \right).$$

Taking the logarithm, we have:

$$\log p(x \mid s, \mu, \sigma^2) = -\frac{D}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left\| x - \sum_{i=1}^{K} s_i \mu_i \right\|^2.$$

Expanding the quadratic form:

$$\left\| x - \sum_{i=1}^{K} s_i \mu_i \right\|^2 = x^T x - 2 \sum_{i=1}^{K} s_i \mu_i^T x + \sum_{i,j=1}^{K} s_i s_j \mu_i^T \mu_j.$$

Substituting back, the log-likelihood becomes:

$$\log p(x \mid s, \mu, \sigma^2) = -\frac{D}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} x^T x + \frac{1}{\sigma^2} \sum_{i=1}^{K} s_i \mu_i^T x - \frac{1}{2\sigma^2} \sum_{i,j=1}^{K} s_i s_j \mu_i^T \mu_j.$$

Combining the Prior and Likelihood Terms Combining $\log p(s \mid \pi)$ and $\log p(x \mid s, \mu, \sigma^2)$, we obtain:

$$\log p(s, x) = \sum_{i=1}^{K} s_i \log \left(\frac{\pi_i}{1 - \pi_i} \right) - \frac{D}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} x^T x + \frac{1}{\sigma^2} \sum_{i=1}^{K} s_i \mu_i^T x - \frac{1}{2\sigma^2} \sum_{i,j=1}^{K} s_i s_j \mu_i^T \mu_j.$$

Rearranging into Single and Pairwise Terms We aim to express $\log p(s, x)$ in the form:

$$\log p(s, x) = \sum_{i=1}^{K} \log f_i(s_i) + \sum_{i<j} \log g_{ij}(s_i, s_j) + \text{const},$$

where $f_i(s_i)$ are factors involving single variables and $g_{ij}(s_i, s_j)$ are factors involving pairs of variables.

Constant Terms:

$$\text{const} = -\frac{D}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}x^T x + \sum_{i=1}^{K}\log(1-\pi_i).$$

Single Variable Terms: For each i,

$$\log f_i(s_i) = s_i\left[\log\left(\frac{\pi_i}{1-\pi_i}\right) + \frac{\mu_i^T x}{\sigma^2} - \frac{\mu_i^T\mu_i}{2\sigma^2}\right].$$

Pairwise Terms: For each $i < j$,

$$\log g_{ij}(s_i, s_j) = -\frac{\mu_i^T\mu_j}{2\sigma^2}s_i s_j.$$

Final Factorization Putting it all together, the log-joint probability can be expressed as:

$$\log p(s, x) = \underbrace{\left(-\frac{D}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}x^T x + \sum_{i=1}^{K}\log(1-\pi_i)\right)}_{\text{constant}} + \sum_{i=1}^{K}\log f_i(s_i) + \sum_{i<j}\log g_{ij}(s_i, s_j).$$

Relation to the Boltzmann Machine The structure of $\log p(s, x)$ resembles that of a Boltzmann machine, where:

$$\log p(s) \propto \sum_i h_i s_i + \sum_{i<j} J_{ij} s_i s_j,$$

with:

$$h_i = \log\left(\frac{\pi_i}{1-\pi_i}\right) + \frac{\mu_i^T x}{\sigma^2} - \frac{\mu_i^T\mu_i}{2\sigma^2},$$

$$W_{ij} = -\frac{\mu_i^T\mu_j}{2\sigma^2}.$$

Thus, the binary latent factor model is structurally equivalent to a Boltzmann machine with data-dependent fields h_i and pairwise couplings W_{ij}.

Final Expression For Clearity:

$$\log p(s, x) = \sum_{i=1}^{K}\log f_i(s_i) + \sum_{i<j}\log g_{ij}(s_i, s_j) + \text{constant},$$

where:

$$f_i(s_i) = \exp\left(s_i\left[\log\left(\frac{\pi_i}{1-\pi_i}\right) + \frac{\mu_i^T x}{\sigma^2} - \frac{\mu_i^T\mu_i}{2\sigma^2}\right]\right),$$

$$g_{ij}(s_i, s_j) = \exp\left(-\frac{\mu_i^T\mu_j}{2\sigma^2}s_i s_j\right).$$

This formulation is similar to a Boltzmann machine, where each latent variable s_i interacts pairwise with others through the couplings J_{ij} and is influenced by individual fields h_i.

5.2 b

We consider a binary latent factor model with factors $f_i(s_i)$ and $g_{ij}(s_i, s_j)$ defined over binary variables $s_i, s_j \in \{0, 1\}$. We use Expectation Propagation (EP) to approximate these factors by simpler, tractable forms $\tilde{f}_i(s_i)$ and $\tilde{g}_{ij}(s_i, s_j)$. Our goal is to find update rules for the parameters of these site approximations based on moment matching.

Single-Site Factor Approximation

The original single-site factor:

$$f_i(s_i) \propto \exp(b_i s_i),$$

where b_i is a known parameter derived from the model (for example, from the log of a Bernoulli probability ratio).

We approximate $f_i(s_i)$ by:

$$\tilde{f}_i(s_i) = \theta_{ii}^{s_i}(1 - \theta_{ii})^{1-s_i}.$$

Taking logs and expressing in terms of natural parameters:

$$\log \tilde{f}_i(s_i) = \log\left(\frac{\theta_{ii}}{1 - \theta_{ii}}\right) s_i \equiv \eta_{ii} s_i.$$

Since f_i and $\tilde{f}_i$ are both Bernoulli-like factors in exponential form, we can match them directly:

$$\log \tilde{f}_i(s_i) = \log f_i(s_i) \implies \eta_{ii} = b_i.$$

From $\eta_{ii} = \log(\theta_{ii}/(1 - \theta_{ii}))$, we have:

$$\theta_{ii} = \frac{1}{1 + \exp(-b_i)}.$$

Thus, the update for the single-site factor is direct and involves no approximation step.

Pairwise Factor Approximation

The original pairwise factor is:

$$g_{ij}(s_i, s_j).$$

After conditioning on other variables, we consider a tilted distribution for EP:

$$\hat{p}_{ij}(s_i, s_j) \propto g_{ij}(s_i, s_j) q_{\neg \tilde{g}_{ij}}(s_i, s_j),$$

where $q_{\neg \tilde{g}_{ij}}$ is the cavity distribution excluding the site factor $\tilde{g}_{ij}$.

We approximate $g_{ij}(s_i, s_j)$ by:

$$\tilde{g}_{ij}(s_i, s_j) = (\theta_{ji}^{s_i}(1 - \theta_{ji})^{1-s_i})(\theta_{ij}^{s_j}(1 - \theta_{ij})^{1-s_j}).$$

In exponential form:

$$\log \tilde{g}_{ij}(s_i, s_j) = \eta_{ji} s_i + \eta_{ij} s_j,$$

with $\eta_{ji} = \log(\theta_{ji}/(1 - \theta_{ji}))$ and $\eta_{ij} = \log(\theta_{ij}/(1 - \theta_{ij}))$.

Cavity and Tilted Distributions

Define the cavity distribution excluding $\tilde{g}_{ij}$:

$$q_{\neg \tilde{g}_{ij}}(s_i, s_j) \propto \exp(\eta_{i,\neg s_j} s_i + \eta_{j,\neg s_i} s_j),$$

where $\eta_{i,\neg s_j}$ and $\eta_{j,\neg s_i}$ are the effective natural parameters coming from all other factors connected to nodes i and j except $\tilde{g}_{ij}$.

The tilted distribution is:

$$\hat{p}_{ij}(s_i, s_j) \propto g_{ij}(s_i, s_j) q_{\neg \tilde{g}_{ij}}(s_i, s_j).$$

Suppose $\log g_{ij}(s_i, s_j) \propto W_{ij} s_i s_j$, where W_{ij} encodes the pairwise coupling from the true factor. Then:

$$\log \hat{p}_{ij}(s_i, s_j) \propto W_{ij} s_i s_j + \eta_{i,\neg s_j} s_i + \eta_{j,\neg s_i} s_j.$$

Moment Matching Conditions

EP requires that the approximate factor $\tilde{g}_{ij}$ match certain marginal moments of $\hat{p}_{ij}$. Specifically, we match the first moments with respect to s_i and s_j.

Approximate Factor Moments: Under $\tilde{g}_{ij}(s_i, s_j)q_{\neg\tilde{g}_{ij}}(s_i, s_j)$, we have:

$$\log(\tilde{g}_{ij}(s_i, s_j)q_{\neg\tilde{g}_{ij}}(s_i, s_j)) \propto (\eta_{ji} + \eta_{i,\neg s_j})s_i + (\eta_{ij} + \eta_{j,\neg s_i})s_j.$$

This is a factorized form where the distribution over s_i and s_j is effectively like two independent Bernoulli distributions (once the other variable is summed out). Thus:

$$\mathbb{E}_q[\mathbb{I}(s_i = 1)] = \frac{1}{1 + \exp(-(\eta_{ji} + \eta_{i,\neg s_j}))},$$

$$\mathbb{E}_q[\mathbb{I}(s_j = 1)] = \frac{1}{1 + \exp(-(\eta_{ij} + \eta_{j,\neg s_i}))}.$$

True Factor Moments: Under $g_{ij}(s_i, s_j)q_{\neg\tilde{g}_{ij}}(s_i, s_j)$:

$$\log(g_{ij}(s_i, s_j)q_{\neg\tilde{g}_{ij}}(s_i, s_j)) \propto W_{ij}s_is_j + \eta_{i,\neg s_j}s_i + \eta_{j,\neg s_i}s_j.$$

To find $\mathbb{E}[\mathbb{I}(s_i = 1)]$ under this distribution, we marginalize out s_j:

$$\sum_{s_j} g_{ij}(s_i, s_j)q_{\neg\tilde{g}_{ij}}(s_i, s_j) \propto \exp(\eta_{i,\neg s_j}) + \exp(W_{ij} + \eta_{i,\neg s_j} + \eta_{j,\neg s_i}).$$

Normalizing over both $s_i = 0, 1$ and $s_j = 0, 1$ yields:

$$\mathbb{E}_{\hat{p}_{ij}}[\mathbb{I}(s_i = 1)] = \frac{\exp(\eta_{i,\neg s_j})(1 + \exp(W_{ij} + \eta_{j,\neg s_i}))}{[\exp(\eta_{i,\neg s_j})(1 + \exp(W_{ij} + \eta_{j,\neg s_i}))] + [1 + \exp(\eta_{j,\neg s_i})]}.$$

Similarly, for s_j:

$$\mathbb{E}_{\hat{p}_{ij}}[\mathbb{I}(s_j = 1)] = \frac{\exp(\eta_{j,\neg s_i})(1 + \exp(W_{ij} + \eta_{i,\neg s_j}))}{[\exp(\eta_{j,\neg s_i})(1 + \exp(W_{ij} + \eta_{i,\neg s_j}))] + [1 + \exp(\eta_{i,\neg s_j})]}.$$

Equating the Moments

EP sets:

$$\mathbb{E}_{\tilde{g}_{ij}q_{\neg}}[\mathbb{I}(s_i = 1)] = \mathbb{E}_{g_{ij}q_{\neg}}[\mathbb{I}(s_i = 1)],$$

and similarly for s_j.

Starting from:

$$\frac{1}{1 + \exp(-(\eta_{ji} + \eta_{i,\neg s_j}))} = \frac{\exp(\eta_{i,\neg s_j})(1 + \exp(W_{ij} + \eta_{j,\neg s_i}))}{[\exp(\eta_{i,\neg s_j})(1 + \exp(W_{ij} + \eta_{j,\neg s_i}))] + [1 + \exp(\eta_{j,\neg s_i})]}.$$

After simplification (cancelling terms and rearranging), this yields:

$$\eta_{ji} = \log\left(\frac{1 + \exp(W_{ij} + \eta_{j,\neg s_i})}{1 + \exp(\eta_{j,\neg s_i})}\right).$$

By symmetry, for s_j:

$$\eta_{ij} = \log\left(\frac{1 + \exp(W_{ij} + \eta_{i,\neg s_j})}{1 + \exp(\eta_{i,\neg s_j})}\right).$$

Final Update Equations

The final update rules for the site approximation parameters are:

$$\eta_{ji}^{\text{new}} = \log\left(\frac{1 + \exp(W_{ij} + \eta_{j,\neg s_i})}{1 + \exp(\eta_{j,\neg s_i})}\right), \quad \eta_{ij}^{\text{new}} = \log\left(\frac{1 + \exp(W_{ij} + \eta_{i,\neg s_j})}{1 + \exp(\eta_{i,\neg s_j})}\right).$$

These updates ensure that the approximating factor $\tilde{g}_{ij}(s_i, s_j)$ matches the first-order moments of the true tilted distribution $\hat{p}_{ij}(s_i, s_j)$.

5.3 c

From the moment-matching equations derived in part (b), we have updates to the natural parameters η_{ji} of the form:

$$\eta_{ji} = \log\left(\frac{1+\exp(W_{ij}+\eta_{j,\neg s_i})}{1+\exp(\eta_{j,\neg s_i})}\right).$$

Focusing on the term $\eta_{i,\neg s_j}$, we note that:

$$\eta_{i,\neg s_j} = \eta_{ii} + \sum_{k\in ne(i), k\neq j} \eta_{ki},$$

where η_{ii} is the natural parameter of the singleton factor for node i, and η_{ki} are the natural parameters from all neighbors k of i (except j).

Similarly, for node j:

$$\eta_{j,\neg s_i} = \eta_{jj} + \sum_{k\in ne(j), k\neq i} \eta_{kj}.$$

These summations show that the natural parameter of each node is influenced by all of its neighbors. In a fully connected graph, every node is a neighbor of every other node. Thus, each node's approximate distribution depends on parameters coming from all other nodes. This is precisely the condition under which loopy BP arises: the updates become cyclic and no node can be considered as a leaf from which to start a straightforward tree-like message passing.

Under these factored approximations, the approximate distribution for a single variable s_i can be written as:

$$q(s_i) \propto \exp\left(\sum_{j=1}^{K} \eta_{ij} s_i\right).$$

Defining $\lambda_i = \mathbb{E}[s_i]$ under $q(s_i)$, we get:

$$\lambda_i = \frac{1}{1+\exp(-\sum_{j=1}^{K}\eta_{ij})}.$$

Since every node depends on parameters from every other node, the message updates form loops in the factor graph. Therefore, using factored approximate messages leads to a Loopy BP algorithm, rather than a simple, acyclic BP that is guaranteed to converge on a tree-structured graph.

5.4 d

We can apply ARD to infer which model components are relevant for explaining observed data. ARD works by placing hierarchical priors on parameters, introducing hyperparameters that control the degree to which each component contributes to the model. As inference proceeds, some hyperparameters tend towards values that effectively prune out unnecessary components, leading to sparse and parsimonious solutions.

When applied to a latent variable model, ARD can determine which latent factors are truly needed. which suitable for selecting the number of hidden binary variables K, since ARD can effectively "turn off" latent dimensions by pushing their associated parameters' priors to configurations that imply negligible contribution.

Incorporating ARD into a Loopy BP Setup

loopy BP approximates marginal distributions in graphical models that contain loops. Here, we consider a model with K latent binary variables $s = (s_1, \ldots, s_K)$ and observed data X. Without ARD, loopy BP would just pass messages around the factor graph until approximate marginals stabilize.

To apply ARD:

1. **Hierarchical Priors:** Introduce hyperparameters α_i for each latent variable s_i. For instance, place a Gamma prior on each α_i, and condition the parameterization of the factors involving s_i on α_i. A common ARD scheme is:
$$p(\mu_i \mid \alpha_i) = \mathcal{N}(0, \alpha_i^{-1} I),$$
where μ_i is a parameter vector controlling how latent variable s_i influences the observed data. If α_i becomes very large, the prior on μ_i becomes very tight, driving $\mu_i \to 0$ and thus rendering the i-th latent variable irrelevant.

2. **Extended Factor Graph:** Extend the original factor graph to include nodes for α_i and factors that link α_i to μ_i and the observed data likelihood. This turns the model into a hierarchical Bayesian model.

3. **Loopy BP with Hyperparameters:** Now run loopy BP on the extended graph. Messages are passed not only between latent variable nodes and data nodes, but also between latent nodes and hyperparameter nodes. As the messages iterate, the posterior over each α_i adjusts. If the model finds that a latent variable s_i is not helpful in explaining the data, the posterior over α_i will concentrate at values that shrink μ_i to zero, pruning out the latent dimension s_i.

4. **Marginalization and Parameter Updates:** Each iteration of loopy BP provides approximate marginals for s_i and α_i. Using these marginals, you can update the hyperparameters and re-run loopy BP in an iterative scheme:

   (a) Update messages via loopy BP $\quad\to\quad$ (b) Update ARD hyperparameters $\quad\to\quad$ (c) Repeat.

   This iterative refinement continues until convergence.

Advantages and Difficulties

Advantages: - ARD provides a data-driven mechanism for pruning unnecessary latent variables, thus selecting K without having to run separate inference for multiple candidate values of K. - It encourages sparsity and leads to simpler models that generalize better.

Difficulties: - **Increased Complexity:** The extended factor graph is more complex. Loopy BP must now handle additional hyperparameter nodes, increasing the computational overhead. - **Non-convexity and Convergence Issues:** Loopy BP is already an approximate and sometimes unstable algorithm. Adding hierarchical ARD priors adds another layer of complexity. Convergence might require careful damping or initialization strategies. - **Tuning ARD Priors:** The choice of hyperpriors on α_i is not trivial. Poorly chosen priors can slow convergence or lead to premature pruning of latent factors.

Mitigating Computational Issues

- **Parallelization:** Parallelize message updates and hyperparameter updates to speed up computation. - **Damping and Scheduling:** Apply damping to message updates and update hyperparameters less frequently at the start, then increase update frequency as marginals stabilize. - **Warm Starts:** Begin with a smaller subset of latent variables and gradually introduce more, allowing ARD to decide whether to keep them. - **Variational/EP Hybrid:** If loopy BP struggles, we can consider using a hybrid approach where EP or mean-field approximations help stabilize the posterior estimates of α_i.

Summary

Applying ARD to loopy BP for selecting K integrates Bayesian model selection directly into the inference process. By placing hierarchical priors on parameters associated with each latent dimension, the algorithm can automatically prune irrelevant latent variables during the message passing phase. While this method adds complexity and potential convergence challenges, careful implementation and tuning can yield a principled, data-driven solution to the model selection problem.

6 Implement the EP/loopy-BP algorithm [Bonus]

The full code implementation can be found at Appendix Problem 6. Again, please note that while we could have refactored the code for conciseness, we need to prioritize clarity and maintain a self-contained format to enhance readability in the PDF so the full code could be a bit lengthy.
We compare the learned feature in EP/LoopyBP with the variational mean-field algorithm in Problem 3.

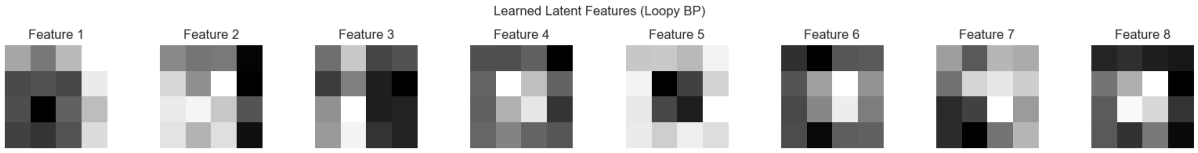


Figure 19: μ_i of the Latent Factors after Training with EP/Loopy-BP

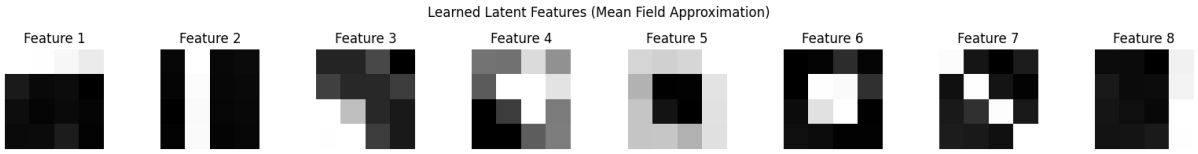


Figure 20: μ_i of the Latent Factors after Training with VMF

The variational mean-field (VMF) algorithm shows stronger performance than the EP/Loopy Belief Propagation (LoopyBP) algorithm because VMF tends to learn more coherent, less redundant features. By contrast, LoopyBP occasionally produces duplicate or highly correlated features (e.g., Feature 4 and Feature 6) and exhibits noisier outcomes overall.

A key way to see why this happens is by examining the evolution of free energy in each method. The free energy in the mean-field approach converges reliably, indicating that VMF finds a stable solution. Meanwhile, LoopyBP struggles to converge—an expected drawback given that loopy belief propagation does not come with theoretical guarantees for convergence in loopy graphs. When convergence fails, the algorithm cannot successfully identify many of the latent factors, leading to the less consistent features we observ

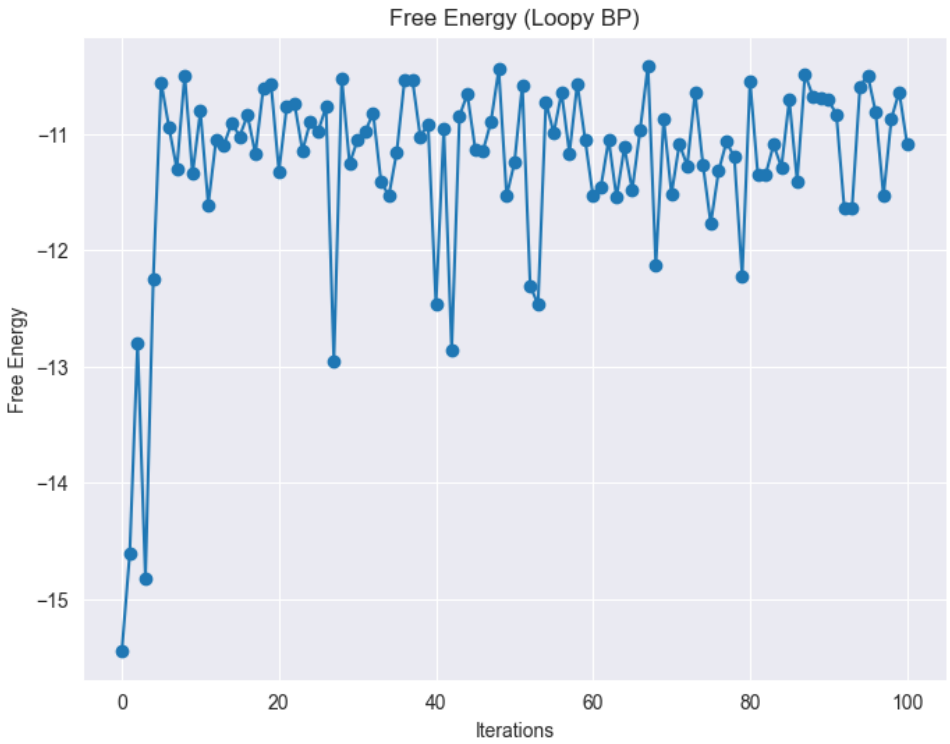


Figure 21: Free Energy with EP/Loopy-BP

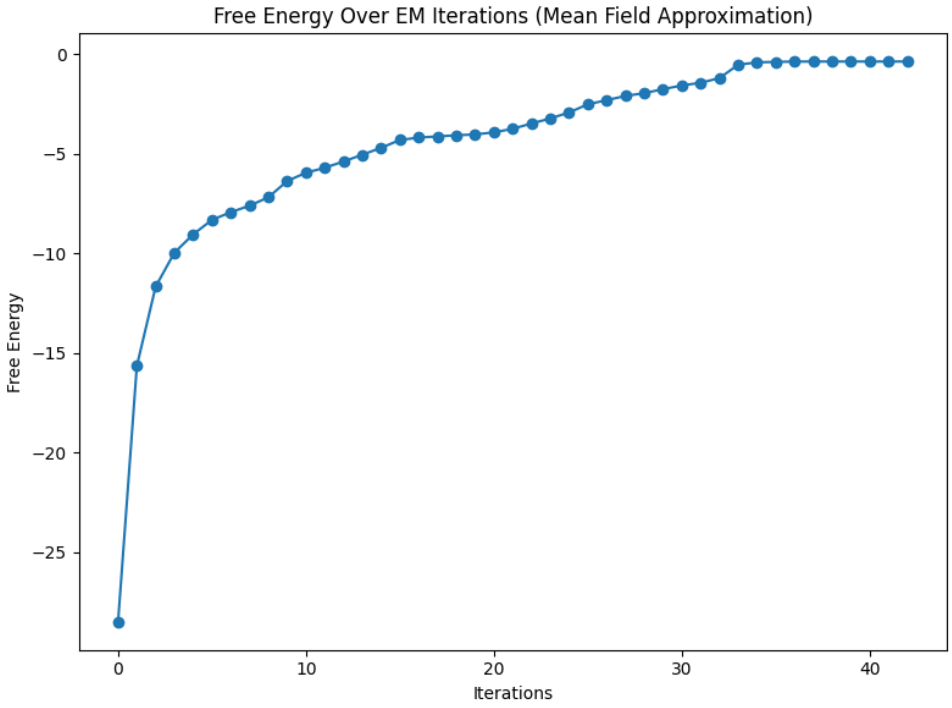


Figure 22: Free Energy with VMF

A Python Code Implementations

A.1 Problem2

Part (a)

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

# Constants
DATA_FILE = 'co2.txt'
BASE_YEAR = 1980.0  # Base year for adjusting the time variable

# Prior Parameters
PRIOR_MEAN = np.array([0.0, 360.0])  # [a, b]
PRIOR_COVARIANCE = np.diag([10.0**2, 100.0**2])  # Covariance matrix
PRIOR_COV_INV = np.linalg.inv(PRIOR_COVARIANCE)

# Initialize lists to store time and CO2 measurements
time = []
co2_average = []

# Load data from the file
try:
    with open(DATA_FILE, 'r') as file:
        for line in file:
            line = line.strip()

            # Skip comment lines and empty lines
            if line.startswith('#') or not line:
                continue

            # Split the line into columns
            columns = line.split()

            # Ensure there are at least 5 columns
            if len(columns) >= 5:
                try:
                    decimal_year = float(columns[2])
                    average_co2 = float(columns[3])

                    # Adjust time relative to the base year
                    adjusted_time = decimal_year - BASE_YEAR
                    time.append(adjusted_time)
                    co2_average.append(average_co2)
                except ValueError:
                    # Handle lines with non-numeric data
                    continue
except FileNotFoundError:
    raise FileNotFoundError(f"The file {DATA_FILE} was not found.")

# Convert lists to NumPy arrays for efficient computations
t = np.array(time)  # Time variable (years since BASE_YEAR)
y = np.array(co2_average)  # Average CO2 measurements

# Design Matrix Construction
# Each row corresponds to [t_i, 1] representing the linear model coefficients
X = np.column_stack((t, np.ones_like(t)))

# Posterior Covariance Calculation
#  _post  = ( _prior  ^{-1} + X^T X)^{-1}
posterior_cov_inv = PRIOR_COV_INV + X.T @ X
posterior_covariance = np.linalg.inv(posterior_cov_inv)

# Posterior Mean Calculation
#  _post  =  _post  ( _prior  ^{-1}  _prior  + X^T y)
posterior_mean = posterior_covariance @ (PRIOR_COV_INV @ PRIOR_MEAN + X.T @ y)

# Extract posterior estimates for parameters a and b
a_posterior, b_posterior = posterior_mean

# Display the results
```

```
print("Posterior␣Mean␣of␣a␣(slope):", a_posterior)
print("Posterior␣Mean␣of␣b␣(intercept):", b_posterior)
print("Posterior␣Covariance␣Matrix:\n", posterior_covariance)

# Optional: Visualization of the Data and Posterior Mean Line
plt.figure(figsize=(10, 6))
plt.scatter(t + BASE_YEAR, y, label='CO2␣Measurements', alpha=0.5)
plt.plot(t + BASE_YEAR, a_posterior * t + b_posterior, color='red', label='Posterior␣
    Mean')
plt.xlabel('Year')
plt.ylabel('Average␣CO2')
plt.title('CO2␣Levels␣Over␣Time')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 2: MAP Python Code Implementations

Part (b)

```
import scipy.stats as stats

# Assumptions for previous code:
# t: numpy array of shape (N,) containing times (decimal years)
# y: numpy array of shape (N,) containing observed CO2 concentrations (fobs(t))
# a_posterior, b_posterior: scalars for MAP estimates of a and b, obtained from previous
    steps

# Compute residuals: gobs(t) = fobs(t) - (a_MAP * t + b_MAP)
gobs = y - (a_posterior * t + b_posterior)

# Plot residuals against time as a line plot
plt.figure(figsize=(10, 6))
plt.plot(t, gobs, label='Residuals', color='blue')
plt.axhline(y=0, color='red', linestyle='--', linewidth=1)
plt.xlabel('Time (decimal years)')
plt.ylabel('Residual $g_{obs}(t)$ (ppm)')
plt.title('Residuals vs. Time')
plt.legend()
plt.grid(True)
plt.savefig(fname="2b1.png", dpi=300)
plt.show()

# Print mean and variance of residuals
mean_res = np.mean(gobs)
var_res = np.var(gobs, ddof=1)  # sample variance
print(f"Mean of residuals: {mean_res:.4f}")
print(f"Variance of residuals: {var_res:.4f}")

# Test normality (e.g., Shapiro-Wilk test)
shapiro_stat, shapiro_p = stats.shapiro(gobs)
print(f"Shapiro-Wilk test statistic: {shapiro_stat:.4f}, p-value: {shapiro_p:.4f}")

# Plot histogram of residuals and compare to a standard normal PDF
plt.figure(figsize=(10, 5))
plt.hist(gobs, bins=30, density=True, alpha=0.7, edgecolor='black', label='Residual
    Histogram')
plt.title('Histogram of Residuals')
plt.xlabel('Residual')
plt.ylabel('Density')

# Plot a standard normal PDF to reflect the prior belief   (t) ~ N(0,1)
xvals = np.linspace(mean_res - 4*np.sqrt(var_res), mean_res + 4*np.sqrt(var_res), 200)
normal_pdf = stats.norm.pdf(xvals, loc=0, scale=1)  # N(0,1)
plt.plot(xvals, normal_pdf, 'r--', label='N(0,1) PDF (prior)')
plt.legend()
plt.grid(True)
plt.savefig(fname="2b2.png", dpi=300)
plt.show()

# Q-Q plot to check normality
plt.figure(figsize=(10,5))
stats.probplot(gobs, dist="norm", plot=plt)
plt.title('Q-Q Plot of Residuals')
plt.grid(True)
plt.savefig(fname="2b3.png", dpi=300)
plt.show()
```

Listing 3: residual

Part (c and d)

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import seaborn as sns
from jax import random
import numpy as np

jax.config.update("jax_enable_x64", True)

def kernel(x1, x2, theta, tau, sigma, phi, eta, zeta):
    """
    Combined periodic and squared exponential kernel with noise:
    k(s,t) = theta^2[exp(-2 sin^2(pi(s-t)/tau)/sigma^2) + phi^2 exp(-((s-t)^2)/(2 eta^2)
        )] + zeta^2 delta_{s=t}.
    """
    pi = jnp.pi
    dist = pi * (x1[:, None] - x2[None, :]) / tau

    # Periodic component
    periodic_part = theta**2 * jnp.exp(-2.0 * (jnp.sin(dist))**2 / sigma**2)
    # RBF component (with theta^2 and phi^2)
    se_part = (theta**2 * phi**2) * jnp.exp(-((x1[:, None] - x2[None, :])**2) / (2 * eta
        **2))

    # Noise on the diagonal if x1 == x2
    same_input = (x1.shape == x2.shape) and jnp.all(x1 == x2)
    noise_part = jnp.where(
        same_input,
        zeta**2 * jnp.eye(len(x1)),
        jnp.zeros((len(x1), len(x2)))
    )

    return periodic_part + se_part + noise_part

@jax.jit
def compute_covariance_matrix(x, params):
    theta, tau, sigma, phi, eta, zeta = params
    K = kernel(x, x, theta, tau, sigma, phi, eta, zeta)
    return K

@jax.jit
def gp_sample(key, x, K):
    """
    Draw a function sample from a GP with mean zero and covariance K.
    This function returns one sample vector f ~ N(0, K).
    """
    N = len(x)
    mean = jnp.zeros(N)
    # Add a small jitter for numerical stability in Cholesky
    L = jnp.linalg.cholesky(K + 1e-9 * jnp.eye(N))
    eps = random.normal(key, shape=(N,))
    return mean + L @ eps

# Let's choose a particular set of parameters
theta_default = 2.5
tau_default   = 5
sigma_default = 1.26
phi_default   = 0.5
eta_default   = 1.26
zeta_default  = 0.01

params = (theta_default, tau_default, sigma_default, phi_default, eta_default,
    zeta_default)

# Define input points
t = jnp.linspace(0, 20, 1000)

# Compute covariance matrix
K = compute_covariance_matrix(t, params)

# Generate three samples from the GP
```

```
key = random.PRNGKey(42)
key1, key2, key3 = random.split(key, 3)
f_sample1 = gp_sample(key1, t, K)
f_sample2 = gp_sample(key2, t, K)
f_sample3 = gp_sample(key3, t, K)

# Plot the three samples on the same figure
plt.figure(figsize=(10,6))
plt.plot(t, f_sample1, label='Sample 1')
plt.plot(t, f_sample2, label='Sample 2')
plt.plot(t, f_sample3, label='Sample 3')
plt.title("Three GP Samples from the Same Covariance Function")
plt.xlabel("t")
plt.ylabel("$f_{GP}(t)$")
plt.legend()
plt.grid(True)
plt.savefig("2cd1.png", dpi=300)
plt.show()

# Plot the Gram matrix as a heatmap with t values on axes
K_np = np.array(K)
plt.figure(figsize=(9,8))

# Create a seaborn heatmap
ax = sns.heatmap(K_np, cmap='viridis', cbar=False)

# Define tick positions
tick_positions = [0, 500, 999]
tick_labels = [f"{t[0]:.1f}", f"{t[500]:.1f}", f"{t[-1]:.1f}"]

# Set x and y ticks
ax.set_xticks(tick_positions)
ax.set_xticklabels(tick_labels)
ax.set_yticks(tick_positions)
ax.set_yticklabels(tick_labels)

plt.title("Covariance Matrix Heatmap")
plt.xlabel("t")
plt.ylabel("t")
plt.tight_layout()
plt.savefig("2cd2.png", dpi=300)
plt.show()
```

Listing 4: draw sample and plot gram matrix

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import seaborn as sns
from jax import random
import numpy as np

# Enable 64-bit precision
jax.config.update("jax_enable_x64", True)

def kernel(x1, x2, theta, tau, sigma, phi, eta, zeta):
    """
    Combined periodic and squared exponential kernel with noise.
    """
    pi = jnp.pi
    dist = pi * (x1[:, None] - x2[None, :]) / tau
    periodic_part = theta**2 * jnp.exp(-2.0 * (jnp.sin(dist))**2 / sigma**2)
    se_part = (theta**2 * phi**2) * jnp.exp(-((x1[:, None] - x2[None, :])**2) / (2 * eta
        **2))

    # Add noise only if x1 and x2 are the same
    same_input = (x1.shape == x2.shape) and jnp.all(x1 == x2)
    noise_part = jnp.where(
        same_input,
        zeta**2 * jnp.eye(len(x1)),
        jnp.zeros((len(x1), len(x2)))
    )

    return periodic_part + se_part + noise_part

@jax.jit
def compute_covariance_matrix(x, params):
    theta, tau, sigma, phi, eta, zeta = params
    K = kernel(x, x, theta, tau, sigma, phi, eta, zeta)
    return K

@jax.jit
def gp_sample(key, x, K):
    """
    Draw a function sample from a GP with mean zero and covariance K.
    This function returns one sample vector f ~ N(0, K).
    """
    N = len(x)
    mean = jnp.zeros(N)
    L = jnp.linalg.cholesky(K + 1e-6 * jnp.eye(N))  # Add jitter for numerical stability
    eps = random.normal(key, shape=(N,))
    return mean + L @ eps

# Define input points
x = jnp.linspace(0, 20, 200)

# Parameter values
theta_values = [0.01, 1.26, 2.5, 3.75, 5.0]
tau_values   = [0.01, 1.26, 2.5, 3.75, 5]
sigma_values = [0.01, 1.26, 2.5, 3.75, 5.0]
phi_values   = [0.01, 1.26, 2.5, 3.75, 5.0]
eta_values   = [0.01, 1.26, 2.5, 3.75, 5.0]
zeta_values  = [0.01, 1.26, 2.5, 3.75, 5.0]

# Default values for parameters
theta_default = 2.5
tau_default   = 5
sigma_default = 1.26
phi_default   = 0.5
eta_default   = 1.26
zeta_default  = 0.01

# Define parameter sweeps: Each row sweeps one parameter
param_sweeps = [
    ("theta", theta_values, theta_default, tau_default, sigma_default, phi_default,
        eta_default, zeta_default),
    ("tau", tau_values, theta_default, tau_default, sigma_default, phi_default,
```

```
        eta_default, zeta_default),
    ("sigma", sigma_values, theta_default, tau_default, sigma_default, phi_default,
        eta_default, zeta_default),
    ("phi", phi_values, theta_default, tau_default, sigma_default, phi_default,
        eta_default, zeta_default),
    ("eta", eta_values, theta_default, tau_default, sigma_default, phi_default,
        eta_default, zeta_default),
    ("zeta", zeta_values, theta_default, tau_default, sigma_default, phi_default,
        eta_default, zeta_default),
]


key = random.PRNGKey(42)

### Figure for GP samples ###
fig_samples, axes_samples = plt.subplots(len(param_sweeps), 5, figsize=(20, 25), sharex=
    False, sharey=False)

for row_idx, (param_name, values, th, ta, si, ph, et, ze) in enumerate(param_sweeps):
    for col_idx, val in enumerate(values):
        # Depending on which parameter we vary, set that param to val
        if param_name == "theta":
            params = (val, ta, si, ph, et, ze)
        elif param_name == "tau":
            params = (th, val, si, ph, et, ze)
        elif param_name == "sigma":
            params = (th, ta, val, ph, et, ze)
        elif param_name == "phi":
            params = (th, ta, si, val, et, ze)
        elif param_name == "eta":
            params = (th, ta, si, ph, val, ze)
        elif param_name == "zeta":
            params = (th, ta, si, ph, et, val)
        else:
            raise ValueError(f"Unknown parameter: {param_name}")

        K = compute_covariance_matrix(x, params)
        f_sample = gp_sample(key, x, K)
        ax = axes_samples[row_idx, col_idx]
        ax.plot(x, f_sample, color='blue')
        ax.set_title(f"{param_name}={val}")
        ax.grid(True)

    # Label the first column with the parameter name
    axes_samples[row_idx, 0].set_ylabel(param_name, fontsize=12)

fig_samples.tight_layout()
fig_samples.savefig("2cd3.png", dpi=300)
plt.show()


### Figure for covariance matrices ###
fig_cov, axes_cov = plt.subplots(len(param_sweeps), 5, figsize=(20, 25))

for row_idx, (param_name, values, th, ta, si, ph, et, ze) in enumerate(param_sweeps):
    for col_idx, val in enumerate(values):
        # Depending on which parameter we vary, set that param to val
        if param_name == "theta":
            params = (val, ta, si, ph, et, ze)
        elif param_name == "tau":
            params = (th, val, si, ph, et, ze)
        elif param_name == "sigma":
            params = (th, ta, val, ph, et, ze)
        elif param_name == "phi":
            params = (th, ta, si, val, et, ze)
        elif param_name == "eta":
            params = (th, ta, si, ph, val, ze)
        elif param_name == "zeta":
            params = (th, ta, si, ph, et, val)
        else:
            raise ValueError(f"Unknown parameter: {param_name}")

        K = compute_covariance_matrix(x, params)
```

```
        K_np = np.array(K)
        ax = axes_cov[row_idx, col_idx]
        sns.heatmap(K_np, cmap='viridis', ax=ax, cbar=False)

        # Set the title for each subplot
        ax.set_title(f"{param_name}={val}", fontsize=10)

        # Set common axis labels 't' and adjust tick marks
        ax.set_xlabel('t', fontsize=8)
        ax.set_ylabel('t', fontsize=8)


# Add common x and y labels for the entire figure
fig_cov.supxlabel('t', fontsize=16)
fig_cov.supylabel('t', fontsize=16)

# Adjust layout to prevent overlap
fig_cov.tight_layout(rect=[0, 0.03, 1, 0.95])

# Save and display the figure
fig_cov.savefig("2cd4.png", dpi=300)
plt.show()
```

Listing 5: Hyper-parameter sweep

A.1.1 Part (e and f)

```
from jax.scipy.linalg import cho_factor, cho_solve


# --- Step 1: Load CO2 data ---

t = []
y = []

# Load data from 'co2.txt'
with open('co2.txt', 'r') as f:
    for line in f:
        line = line.strip()
        if line.startswith('#') or len(line) == 0:
            continue
        parts = line.split()
        if len(parts) >= 5:
            decimal_year = float(parts[2])
            average_co2 = float(parts[3])
            # Shift base year to 1980
            decimal_year -= 1980.0
            t.append(decimal_year)
            y.append(average_co2)

# to JAX arrays
t = jnp.array(t)
y = jnp.array(y)

# --- Step 2: Compute residuals g_obs(t) ---
g_obs = y - (a_posterior * t + b_posterior)

# --- Step 3: Define training and testing sets ---
cutoff = 2007.708 - 1980.0  # 27.708

train_idx = t <= cutoff
test_start = cutoff  # 27.708
test_end = 2020.958 - 1980.0  # 40.958

# Generate monthly intervals from test_start to test_end
num_months = int(round((test_end - test_start) * 12)) + 1  # +1 to include the end
test_t = jnp.linspace(test_start, test_end, num_months)

# Split training data
t_train = t[train_idx]
g_train = g_obs[train_idx]

# --- Step 4: Define the kernel function ---
def kernel(x1, x2, theta, tau, sigma, phi, eta, zeta):
    dist = jnp.pi * (x1[:, None] - x2[None, :]) / tau
    periodic_part = theta**2 * jnp.exp(-2.0 * (jnp.sin(dist)**2) / sigma**2)
    se_part = (theta**2 * phi**2) * jnp.exp(-( (x1[:, None] - x2[None, :])**2 )/(2 * eta
        **2))

    # Check if x1 and x2 represent the same array
    # Ensure that shapes match and all elements are equal
    same_input = (x1.shape == x2.shape) and bool(jnp.all(x1 == x2))
    noise_part = zeta**2 * jnp.eye(len(x1)) if same_input else 0.0

    return periodic_part + se_part + noise_part

# --- Step 5: Define Hyperparameters ---
# These are fixed as per your requirement
# Choose hyperparameters (untrained):
theta = 3.0   # amplitude of periodic
tau = 1.0   # annual period
sigma = 1.0   # smoothness of periodic part
phi = 0.5   # amplitude of SE part
eta = 2.0    # length-scale of SE part
zeta = 0.2    # noise level

# --- Step 6: Construct Covariance Matrices ---
# Compute training covariance matrix K with noise
```

```
K = kernel(t_train, t_train, theta, tau, sigma, phi, eta, zeta)

# Compute cross-covariance matrix K_star between test and train
K_star = kernel(test_t, t_train, theta, tau, sigma, phi, eta, zeta)

# Compute test covariance matrix K_star_star without noise
K_star_star = kernel(test_t, test_t, theta, tau, sigma, phi, eta, zeta)

# --- Step 7: Compute Inverse of K ---
# Perform Cholesky decomposition for numerical stability
L = jnp.linalg.cholesky(K)
# Compute K_inv using Cholesky factors
K_inv = cho_solve((L, True), jnp.eye(len(t_train)))

# --- Step 8: Predictive Mean and Covariance ---
# Predictive mean
g_mean = jnp.dot(K_star, jnp.dot(K_inv, g_train))

# Predictive covariance
# Compute v = L^{-1} K_star^T
v = cho_solve((L, True), K_star.T)
# Compute covariance matrix
g_cov = K_star_star - jnp.dot(K_star, v)
# Standard deviation
g_std = jnp.sqrt(jnp.diag(g_cov))

# --- Step 9: Compute Full CO2 Predictions ---
f_pred = a_posterior * test_t + b_posterior + g_mean

# --- Step 10: Plotting ---
plt.figure(figsize=(14, 7))
plt.plot(t + 1980.0, y, 'k.', alpha=0.5, label='Observed␣CO2')
plt.axvspan(1980.0, 1980.0 + cutoff, color='gray', alpha=0.1, label='Training␣period')
plt.plot(test_t + 1980.0, f_pred, 'b-', label='Predicted␣mean')
plt.fill_between(test_t + 1980.0, f_pred - g_std, f_pred + g_std, color='blue', alpha
    =0.2, label=' 1 ␣std')

plt.title('GP␣Extrapolation␣of␣CO2␣Concentrations␣Untrained␣Hyperparameters')
plt.xlabel('Decimal␣Year')
plt.ylabel('CO2␣(ppm)')
plt.grid(True)
plt.legend()
plt.savefig("2f1.png", dpi=300)
plt.show()
```

Listing 6: GP

```
import optax
from tqdm.autonotebook import tqdm


def neg_log_marginal_likelihood(params, X, y):
    # params is a dictionary of raw parameters in log-space
    # Exponentiate to ensure positivity
    theta = jnp.exp(params['log_theta'])
    sigma = jnp.exp(params['log_sigma'])
    phi = jnp.exp(params['log_phi'])
    eta = jnp.exp(params['log_eta'])
    zeta = jnp.exp(params['log_zeta'])
    tau = jnp.exp(params['log_tau'])
    # Construct K
    K = kernel(X, X, theta, tau, sigma, phi, eta, zeta)
    # Add a small jitter for numerical stability
    K += 1e-10 * jnp.eye(len(X))

    # Compute NLL = 0.5*y^T K^-1 y + 0.5 log|K| + n/2 log(2  )
    L = jnp.linalg.cholesky(K)
    # Solve for alpha = K^-1 y using Cholesky
    alpha = jax.scipy.linalg.cho_solve((L, True), y)
    n = len(X)
    nll = 0.5 * y.dot(alpha) + jnp.sum(jnp.log(jnp.diag(L))) + 0.5 * n * jnp.log(2.0 *
        jnp.pi)
    return nll

# --- Step 4: Setup Optimization ---
# Initial guesses for parameters in log-space
init_params = {
    'log_theta': jnp.log(3.0),
    'log_tau': jnp.log(1.0),
    'log_sigma': jnp.log(1.0),
    'log_phi': jnp.log(0.5),
    'log_eta': jnp.log(2.0),
    'log_zeta': jnp.log(0.2)
}

# Use Adam optimizer from optax
learning_rate = 0.01
optimizer = optax.adam(learning_rate)
opt_state = optimizer.init(init_params)

# We can jit the gradient function for efficiency
grad_fn = jax.grad(neg_log_marginal_likelihood)

# --- Step 5: Run Gradient-Based Optimization ---
num_iters = 200
params = init_params
for i in tqdm(range(num_iters), desc="Optimizing␣hyperparameters"):
    grads = grad_fn(params, t_train, g_train)
    updates, opt_state = optimizer.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)

    if i % 20 == 0:
        current_loss = neg_log_marginal_likelihood(params, t_train, g_train)
        print(f"Iteration␣{i},␣NLL={current_loss}")

# After optimization
opt_theta = jnp.exp(params['log_theta'])
opt_sigma = jnp.exp(params['log_sigma'])
opt_phi = jnp.exp(params['log_phi'])
opt_eta = jnp.exp(params['log_eta'])
opt_zeta = jnp.exp(params['log_zeta'])
opt_tau = jnp.exp(params['log_tau'])

print("Optimized␣Hyperparameters:")
print("theta␣=", opt_theta, "tau␣=", opt_tau, "sigma␣=", opt_sigma, "phi␣=", opt_phi, "
    eta␣=", opt_eta, "zeta␣=", opt_zeta)

# --- Step 6: Make Predictions ---
# Suppose we want to predict beyond cutoff
```

```
test_start = 2007.708 - 1980.0
test_end = 2020.958 - 1980.0
num_months = int(round((test_end - test_start) * 12)) + 1
test_t = jnp.linspace(test_start, test_end, num_months)

K = kernel(t_train, t_train, opt_theta, opt_tau, opt_sigma, opt_phi, opt_eta, opt_zeta)
K_star = kernel(test_t, t_train, opt_theta, opt_tau, opt_sigma, opt_phi, opt_eta,
    opt_zeta)
K_star_star = kernel(test_t, test_t, opt_theta, opt_tau, opt_sigma, opt_phi, opt_eta,
    opt_zeta)

K_inv = jnp.linalg.inv(K)
g_mean = K_star @ K_inv @ g_train
g_cov = K_star_star - K_star @ K_inv @ K_star.T
g_std = jnp.sqrt(jnp.diag(g_cov))

f_pred = a_posterior * test_t + b_posterior + g_mean

# --- Step 7: Plot Results ---
import matplotlib.pyplot as plt

plt.figure(figsize=(12,6))
plt.plot(t + 1980.0, y, 'k.', alpha=0.5, label='Observed␣CO2')
plt.axvspan(1980.0, 1980.0 + cutoff, color='gray', alpha=0.1, label='Training␣period')
plt.plot(test_t + 1980.0, f_pred, 'b-', label='Predicted␣mean')
plt.fill_between(test_t + 1980.0, f_pred - g_std, f_pred + g_std, color='blue', alpha
    =0.2, label=' 1 ␣std')

plt.title('GP␣Extrapolation␣of␣CO2␣Concentrations␣with␣Trained␣Hyperparameters')
plt.xlabel('Decimal␣Year')
plt.ylabel('CO2␣(ppm)')
plt.grid(True)
plt.legend()
plt.savefig("2f2.png", dpi=300)
plt.show()
```

Listing 7: MLE

A.2 Problem3

Part (a)

```
class MeanField:
    def __init__(self, lambda_matrix: np.ndarray, max_steps: int, convergence_threshold:
        float):
        """
        Initialize the Mean Field Approximation.

        Args:
            lambda_matrix (np.ndarray): Variational parameters (num_samples, num_factors
                ).
            max_steps (int): Maximum number of E-step iterations.
            convergence_threshold (float): Convergence criterion for the E-step.
        """
        self.lambda_matrix = lambda_matrix  # (num_samples, num_factors)
        self.max_steps = max_steps
        self.convergence_threshold = convergence_threshold

    @property
    def expectation_s(self) -> np.ndarray:
        """Expectation of latent factors."""
        return self.lambda_matrix

    @property
    def expectation_ss(self) -> np.ndarray:
        """Expectation of latent factor products."""
        ess = self.lambda_matrix.T @ self.lambda_matrix
        np.fill_diagonal(ess, self.lambda_matrix.sum(axis=0))
        return ess

    @property
    def log_lambda(self) -> np.ndarray:
        """Log of lambda matrix."""
        return np.log(self.lambda_matrix)

    @property
    def log_one_minus_lambda(self) -> np.ndarray:
        """Log of one minus lambda matrix."""
        return np.log(1 - self.lambda_matrix)

    def compute_free_energy(self, data: np.ndarray, model: BinaryLatentFactorModel) ->
        float:
        """
        Compute the free energy associated with current parameters and data.

        Args:
            data (np.ndarray): Data matrix (num_samples, num_features).
            model (BinaryLatentFactorModel): Binary latent factor model instance.

        Returns:
            float: Average free energy per data sample.
        """
        expectation_log_p_x_s = self._compute_expectation_log_p_x_s(data, model)
        entropy = self._compute_entropy()
        return (expectation_log_p_x_s + entropy) / self.lambda_matrix.shape[0]

    def _compute_expectation_log_p_x_s(self, data: np.ndarray, model:
        BinaryLatentFactorModel) -> float:
        """
        Compute the expectation of log P(X, S | theta).

        Args:
            data (np.ndarray): Data matrix (num_samples, num_features).
            model (BinaryLatentFactorModel): Binary latent factor model instance.

        Returns:
            float: Expectation of log P(X, S | theta).
        """
        mu_product = self.lambda_matrix @ model.mu.T
        expectation_ss_mu = np.multiply(
            self.lambda_matrix.T @ self.lambda_matrix,
```

```
            model.mu.T @ model.mu,
        )

        log_p_x_given_s = (
            -self.lambda_matrix.shape[0] * model.num_features / 2 * np.log(2 * np.pi *
                model.variance)
            - 0.5 * model.precision
            * (
                np.sum(np.multiply(data, data))
                - 2 * np.sum(data * mu_product)
                + np.sum(expectation_ss_mu)
                - np.trace(expectation_ss_mu)
                + np.sum(
                    self.lambda_matrix @ (model.mu ** 2).T
                )
            )
        )

        log_p_s = np.sum(
            self.lambda_matrix * model.log_pi
            + (1 - self.lambda_matrix) * model.log_one_minus_pi
        )

        return log_p_x_given_s + log_p_s

    def _compute_entropy(self) -> float:
        """
        Compute the entropy of the variational approximation.

        Returns:
            float: Entropy value.
        """
        return -np.sum(
            self.lambda_matrix * self.log_lambda
            + (1 - self.lambda_matrix) * self.log_one_minus_lambda
        )

    def exclude_factor_lambda(self, factor_index: int) -> np.ndarray:
        """
        Exclude a specific latent factor from the lambda matrix.

        Args:
            factor_index (int): Index of the latent factor to exclude.

        Returns:
            np.ndarray: Updated lambda matrix excluding the specified factor.
        """
        return np.concatenate(
            (
                self.lambda_matrix[:, :factor_index],
                self.lambda_matrix[:, factor_index + 1:],
            ),
            axis=1,
        )

    def _partial_e_step(self, data: np.ndarray, model: BinaryLatentFactorModel,
        factor_index: int) -> np.ndarray:
        """
        Perform a partial variational E-step for a specific latent factor.

        Args:
            data (np.ndarray): Data matrix (num_samples, num_features).
            model (BinaryLatentFactorModel): Binary latent factor model instance.
            factor_index (int): Index of the latent factor to update.

        Returns:
            np.ndarray: Updated lambda vector for the specified latent factor.
        """
        lambda_excluded = self.exclude_factor_lambda(factor_index)
        mu_excluded = model.exclude_factor_mu(factor_index)
        mu_factor = model.mu[:, factor_index]

        log_p_x = (
```

```
            model.precision
            * (data - 0.5 * mu_factor.T - lambda_excluded @ mu_excluded.T)
            @ mu_factor
        )

        log_p_s = np.log(model.pi[0, factor_index] / (1 - model.pi[0, factor_index]))

        log_p_total = log_p_x + log_p_s

        lambda_updated = 1 / (1 + np.exp(-log_p_total))
        lambda_updated = np.clip(lambda_updated, 1e-10, 1 - 1e-10)

        return lambda_updated.reshape(-1, 1)

    def variational_e_step(self, data: np.ndarray, model: BinaryLatentFactorModel) ->
        List[float]:
        """
        Perform the variational E-step.

        Args:
            data (np.ndarray): Data matrix (num_samples, num_features).
            model (BinaryLatentFactorModel): Binary latent factor model instance.

        Returns:
            List[float]: List of free energy values during the E-step iterations.
        """
        free_energy_history = [self.compute_free_energy(data, model)]
        for step in range(self.max_steps):
            for factor in range(model.num_factors):
                self.lambda_matrix[:, factor] = self._partial_e_step(data, model, factor
                    ).flatten()
                current_free_energy = self.compute_free_energy(data, model)
                free_energy_history.append(current_free_energy)
                energy_diff = free_energy_history[-1] - free_energy_history[-2]
                if energy_diff <= self.convergence_threshold:
                    break
            if energy_diff <= self.convergence_threshold:
                break
        return free_energy_history
```

Listing 8: MeanField

Part (e, f, and g)

```python
class BinaryLatentFactorModel:
    def __init__(self, mu: np.ndarray, sigma: float, pi: np.ndarray):
        """
        Initialize the Binary Latent Factor Model.

        Args:
            mu (np.ndarray): Means of the latent factors (num_features, num_factors).
            sigma (float): Standard deviation of the Gaussian noise.
            pi (np.ndarray): Mixing coefficients (1, num_factors).
        """
        self.mu = mu  # (num_features, num_factors)
        self.sigma = sigma
        self.pi = pi  # (1, num_factors)


    @property
    def mu(self):
        return self._mu

    @mu.setter
    def mu(self, value):
        self._mu = value

    @property
    def sigma(self):
        return self._sigma

    @sigma.setter
    def sigma(self, value):
        self._sigma = value

    @property
    def variance(self) -> float:
        """Gaussian noise variance."""
        return self.sigma ** 2

    @variance.setter
    def variance(self, value: float) -> None:
        self.sigma = np.sqrt(value)

    @property
    def precision(self) -> float:
        """Precision of the Gaussian noise."""
        return 1 / self.variance

    @property
    def log_pi(self) -> np.ndarray:
        """Log of mixing coefficients."""
        return np.log(self.pi)

    @property
    def log_one_minus_pi(self) -> np.ndarray:
        """Log of one minus mixing coefficients."""
        return np.log(1 - self.pi)

    @property
    def num_features(self) -> int:
        """Number of features."""
        return self.mu.shape[0]

    @property
    def num_factors(self) -> int:
        """Number of latent factors."""
        return self.mu.shape[1]

    def exclude_factor_mu(self, factor_index: int) -> np.ndarray:
        """
        Exclude a specific latent factor from the means.

        Args:
            factor_index (int): Index of the latent factor to exclude.
```

```
        Returns:
            np.ndarray: Updated means excluding the specified factor.
        """
        return np.concatenate(
            (self.mu[:, :factor_index], self.mu[:, factor_index + 1:]),
            axis=1,
        )

    @staticmethod
    def m_step(X, ES, ESS):
        """
        mu, sigma, pie = MStep(X,ES,ESS)

        Inputs:
        -----------------
               X: shape (N, D) data matrix
              ES: shape (N, K) E_q[s]
             ESS: shape (K, K) sum over data points of E_q[ss'] (N, K, K)
                               if E_q[ss'] is provided, the sum over N is done for you.

        Outputs:
        --------
              mu: shape (D, K) matrix of means in p(y|{s_i},mu,sigma)
           sigma: shape (,)    standard deviation in same
             pie: shape (1, K) vector of parameters specifying generative distribution
                  for s
        """
        N, D = X.shape
        if ES.shape[0] != N:
            raise TypeError('ES must have the same number of rows as X')
        K = ES.shape[1]
        if ESS.shape == (N, K, K):
            ESS = np.sum(ESS, axis=0)
        if ESS.shape != (K, K):
            raise TypeError('ESS must be square and have the same number of columns as
                ES')

        mu = np.dot(np.dot(np.linalg.inv(ESS), ES.T), X).T
        sigma = np.sqrt((np.trace(np.dot(X.T, X)) + np.trace(np.dot(np.dot(mu.T, mu),
            ESS))
                         - 2 * np.trace(np.dot(np.dot(ES.T, X), mu))) / (N * D))
        pi = np.mean(ES, axis=0, keepdims=True)

        return mu, sigma, pi

    def maximization_step(self, data: np.ndarray, approximation: 'MeanField') -> None:
        """
        Perform the M-step to update model parameters.

        Args:
            data (np.ndarray): Data matrix (num_samples, num_features).
            approximation (MeanField): Mean field approximation instance.
        """
        mu, sigma, pi = self.m_step(
            data, approximation.expectation_s, approximation.expectation_ss
        )
        self.mu = mu
        self.sigma = sigma
        self.pi = pi
```

Listing 9: M Step Our Implementation

```python
def em_algorithm(
    data: np.ndarray,
    model: BinaryLatentFactorModel,
    approximation: MeanField,
    max_iterations: int = 100
) -> Tuple[MeanField, BinaryLatentFactorModel, List[float]]:
    """
    Perform the EM algorithm to learn binary latent factors.

    Args:
        data (np.ndarray): Data matrix (num_samples, num_features).
        model (BinaryLatentFactorModel): Binary latent factor model instance.
        approximation (MeanField): Mean field approximation instance.
        max_iterations (int): Maximum number of EM iterations.

    Returns:
        Tuple[MeanField, BinaryLatentFactorModel, List[float]]:
            Updated approximation, updated model, and free energy history.
    """
    free_energy_history = [
        approximation.compute_free_energy(data, model)
    ]
    for iteration in range(max_iterations):
        previous_lambda = approximation.lambda_matrix.copy()

        # E-step
        free_energy_history.extend(approximation.variational_e_step(data, model))

        # M-step
        model.maximization_step(data, approximation)

        # Compute free energy
        current_free_energy = approximation.compute_free_energy(data, model)
        free_energy_history.append(current_free_energy)

        # Check for convergence
        energy_diff = abs(free_energy_history[-1] - free_energy_history[-2])
        lambda_diff = np.linalg.norm(approximation.lambda_matrix - previous_lambda)
        if energy_diff == 0 and lambda_diff == 0:
            print(f"Converged at iteration {iteration + 1}")
            break
    return approximation, model, free_energy_history
```

Listing 10: EM

```python
def visualize_latent_features(mu: np.ndarray, num_factors: int, title: str, save_path:
    str) -> None:
    """
    Visualize latent features as 4x4 images.

    Args:
        mu (np.ndarray): Means of the latent factors (num_features, num_factors).
        num_factors (int): Number of latent factors.
        title (str): Title of the plot.
        save_path (str): Path to save the figure.
    """
    fig, axes = plt.subplots(1, num_factors, figsize=(num_factors * 2, 2))
    for i in range(num_factors):
        axes[i].imshow(mu[:, i].reshape(4, 4), cmap='gray', interpolation='none')
        axes[i].set_title(f"Feature␣{i+1}")
        axes[i].axis('off')
    fig.suptitle(title)
    plt.tight_layout()
    plt.savefig(save_path, bbox_inches="tight")
    plt.show()

def plot_free_energy(free_energy: List[float], title: str, save_path: str) -> None:
    """
    Plot the free energy over EM iterations.

    Args:
        free_energy (List[float]): Free energy history.
        title (str): Title of the plot.
        save_path (str): Path to save the figure.
    """
    plt.figure(figsize=(8, 6))
    plt.plot(free_energy, marker='o')
    plt.title(title)
    plt.xlabel("Iterations")
    plt.ylabel("Free␣Energy")
    plt.tight_layout()
    plt.savefig(save_path, bbox_inches="tight")
    plt.show()
```

Listing 11: visualization

```
def main():
    # Prepare output directories
    outputs_dir = "."
    os.makedirs(outputs_dir, exist_ok=True)

    # Initialize model parameters using Kaiming initialization
    initial_sigma = 1.0
    initial_pi = np.full((1, num_factors), 0.5)

    # Kaiming initialization for mu
    initial_mu = np.random.randn(num_features, num_factors) * np.sqrt(2 / num_features)

    # Create model and approximation instances
    model = BinaryLatentFactorModel(mu=initial_mu, sigma=initial_sigma, pi=initial_pi)
    initial_lambda = np.random.rand(num_samples, num_factors)
    approximation = MeanField(
        lambda_matrix=initial_lambda,
        max_steps=100,
        convergence_threshold=1e-9
    )

    # Visualize initial latent features
    visualize_latent_features(
        mu=model.mu,
        num_factors=num_factors,
        title="Initial Latent Features",
        save_path=os.path.join(outputs_dir, "3f1.png")
    )

    # Perform EM algorithm
    approximation, model, free_energy = em_algorithm(
        data=data,
        model=model,
        approximation=approximation,
        max_iterations=1000
    )

    # Visualize learned latent features
    visualize_latent_features(
        mu=model.mu,
        num_factors=num_factors,
        title="Learned Latent Features",
        save_path=os.path.join(outputs_dir, "3f2.png")
    )

    # Plot free energy progression
    plot_free_energy(
        free_energy=free_energy,
        title="Free Energy Over EM Iterations",
        save_path=os.path.join(outputs_dir, "3e1.png")
    )

    # Analyze free energy with different sigma values
    sigmas = [1, 2, 3]
    free_energies_sigma = []
    for sigma in sigmas:

        temp_data = data[:1, :]
        model.sigma = sigma

        temp_approximation = MeanField(
            lambda_matrix=approximation.lambda_matrix[:1, :],
            max_steps=approximation.max_steps,
            convergence_threshold=approximation.convergence_threshold,
        )
        fe = temp_approximation.variational_e_step(
            data=temp_data,
            model=model
        )
        free_energies_sigma.append(fe)

    # Plot free energy for different sigma values
```

```
    plt.figure(figsize=(10, 6))
    for idx, fe in enumerate(free_energies_sigma):
        plt.plot(fe, label=f"Sigma={sigmas[idx]}")
    plt.title("Free Energy vs EM Iterations for Different Sigma Values")
    plt.xlabel("Iterations")
    plt.ylabel("Free Energy")
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(outputs_dir, "3g1.png"), bbox_inches="tight")
    plt.show()

    # Plot log differences of free energy for different sigma values
    plt.figure(figsize=(10, 6))
    for idx, fe in enumerate(free_energies_sigma):
        diffs = np.diff(fe)
        log_diffs = np.log(diffs + 1e-6)
        plt.plot(log_diffs, label=f"Sigma={sigmas[idx]}")
    plt.title("Log Difference of Free Energy vs EM Iterations for Different Sigma Values
        ")
    plt.xlabel("Iterations")
    plt.ylabel("Log(F(t) - F(t-1))")
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(outputs_dir, "3g2.png"), bbox_inches="tight")
    plt.show()

    print(f"Results saved in '{outputs_dir}' directory.")

if __name__ == "__main__":
    main()
```

Listing 12: Main Execution Block

A.3 Problem4

Part (b)

```
class GaussianPrior:
    def __init__(self, a: float, b: float, d: int, k: int):
        """
        Initialize the Gaussian prior for the mu matrix.

        Parameters
        ----------
        a : float
            Alpha parameter of the Gamma prior.
        b : float
            Beta parameter of the Gamma prior.
        d : int
            Number of dimensions in the observed data.
        k : int
            Number of latent variables.
        """
        self.a = a
        self.b = b
        self.mu = np.zeros((d, k))
        self.alpha = np.ones(k)
        self.w_covariance = np.zeros((k, k))

    def mu_k(self, latent_factor: int) -> np.ndarray:
        """
        Retrieve the column vector of the mu matrix corresponding to a specific latent
            factor.

        Parameters
        ----------
        latent_factor : int
            Index of the latent factor.

        Returns
        -------
        np.ndarray
            Column vector of mu, shape (d, 1).
        """
        return self.mu[:, latent_factor:latent_factor + 1]

    def w_d(self, dimension: int) -> np.ndarray:
        """
        Retrieve the row vector of the mu matrix corresponding to a specific data
            dimension.

        Parameters
        ----------
        dimension : int
            Index of the data dimension.

        Returns
        -------
        np.ndarray
            Row vector of mu, shape (1, k).
        """
        return self.mu[dimension:dimension + 1, :]

    @property
    def a_matrix(self) -> np.ndarray:
        """
        Compute the precision matrix for the weight vector w_d.

        Returns
        -------
        np.ndarray
            Precision matrix, shape (k, k).
        """
        return np.diag(self.alpha)
```

Listing 13: Gaussian Priors

```python
class VariationalBayes:
    def __init__(self, mu: 'GaussianPrior', variance: float, pi: np.ndarray):
        """
        Variational Bayes implementation with a Gaussian prior on mu.

        Parameters
        ----------
        mu : GaussianPrior
            Gaussian prior on latent features.
        variance : float
            Gaussian noise parameter.
        pi : np.ndarray
            Vector of prior probabilities, shape (1, number_of_latent_variables).
        """
        super().__init__()
        self.gaussian_prior = mu
        self._variance = variance
        self._pi = pi

    @property
    def log_pi(self) -> np.ndarray:
        """Compute the natural logarithm of the prior probabilities."""
        return np.log(self.pi)

    @property
    def log_one_minus_pi(self) -> np.ndarray:
        """Compute the natural logarithm of the complement of the prior probabilities.
            """
        return np.log(1 - self.pi)

    @property
    def variance(self) -> float:
        """Get the variance of the Gaussian noise."""
        return self._variance

    @property
    def pi(self) -> np.ndarray:
        """Get the vector of prior probabilities."""
        return self._pi

    @property
    def mu(self) -> np.ndarray:
        """Get the mean matrix of the Gaussian prior."""
        return self.gaussian_prior.mu

    @property
    def d(self) -> int:
        """Get the dimensionality of the observed data."""
        return self.mu.shape[0]

    @property
    def k(self) -> int:
        """Get the number of latent variables."""
        return self.mu.shape[1]

    @property
    def precision(self) -> float:
        """Get the precision (inverse of variance) of the Gaussian noise."""
        return 1.0 / self.variance

    def mu_exclude(self, exclude_latent_index: int) -> np.ndarray:
        """
        Exclude a specific latent variable index from the mean matrix.

        Parameters
        ----------
        exclude_latent_index : int
            Index of the latent variable to exclude.

        Returns
        -------
        np.ndarray
```

```
            Mean matrix excluding the specified latent variable,
            shape (number_of_dimensions, number_of_latent_variables - 1).
        """
        return np.concatenate(
            (
                self.mu[:, :exclude_latent_index],
                self.mu[:, exclude_latent_index + 1:]
            ),
            axis=1,
        )

    @staticmethod
    def calculate_maximisation_parameters(
        x: np.ndarray,
        binary_latent_factor_approximation: 'MeanFieldApproximation'
    ) -> Tuple[np.ndarray, float, np.ndarray]:
        """
        Calculate M-step parameters for the variational approximation.

        Parameters
        ----------
        x : np.ndarray
            Data matrix, shape (number_of_points, number_of_dimensions).
        binary_latent_factor_approximation : MeanFieldApproximation
            Mean field approximation object.

        Returns
        -------
        Tuple[np.ndarray, float, np.ndarray]
            Updated parameters: (means, variance, prior probabilities).
        """
        return m_step(
            X=x,
            ES=binary_latent_factor_approximation.expectation_s,
            ESS=binary_latent_factor_approximation.expectation_ss,
        )

    def _update_w_covariance(
        self,
        binary_latent_factor_approximation: 'MeanFieldApproximation',
    ) -> None:
        """
        Update the covariance matrix for the latent feature weights.

        Parameters
        ----------
        binary_latent_factor_approximation : MeanFieldApproximation
            Mean field approximation object.
        """
        a_matrix = self.gaussian_prior.a_matrix
        expectation_ss = binary_latent_factor_approximation.expectation_ss
        covariance_matrix = a_matrix + self.precision * expectation_ss
        self.gaussian_prior.w_covariance = np.linalg.inv(covariance_matrix)

    def _update_w_mean(
        self,
        x: np.ndarray,
        binary_latent_factor_approximation: 'MeanFieldApproximation',
        dimension_index: int,
    ) -> None:
        """
        Update the mean vector for the latent feature weights for a specific dimension.

        Parameters
        ----------
        x : np.ndarray
            Data matrix, shape (number_of_points, number_of_dimensions).
        binary_latent_factor_approximation : MeanFieldApproximation
            Mean field approximation object.
        dimension_index : int
            Index of the data dimension to update.
        """
        expectation_s = binary_latent_factor_approximation.expectation_s
```

```
        data_column = x[:, dimension_index : dimension_index + 1]
        updated_mean = (
            self.gaussian_prior.w_covariance
            @ (self.precision * expectation_s.T @ data_column)
        ).T
        self.gaussian_prior.mu[dimension_index : dimension_index + 1, :] = updated_mean

    def _hyper_maximisation_step(self) -> None:
        """
        Hyperparameter maximisation step to update alpha, which parameterizes
        the covariance matrix of the Gaussian prior on mu.
        """
        for latent_idx in range(self.k):
            mu_k = self.gaussian_prior.mu_k(latent_idx)
            w_cov_kk = self.gaussian_prior.w_covariance[latent_idx, latent_idx]
            numerator = 2 * self.gaussian_prior.a + self.d - 2
            denominator = (
                2 * self.gaussian_prior.b
                + np.sum(mu_k ** 2)
                + self.d * w_cov_kk
            )
            self.gaussian_prior.alpha[latent_idx] = numerator / denominator

    def maximisation_step(
        self,
        x: np.ndarray,
        binary_latent_factor_approximation: 'MeanFieldApproximation',
    ) -> None:
        """
        Perform the maximisation step, which includes the standard M-step,
        updates to the posterior distribution of mu, and a hyperparameter step.

        Parameters
        ----------
        x : np.ndarray
            Data matrix, shape (number_of_points, number_of_dimensions).
        binary_latent_factor_approximation : MeanFieldApproximation
            Mean field approximation object.
        """
        _, sigma, pi = self.calculate_maximisation_parameters(
            x, binary_latent_factor_approximation
        )
        self._variance = sigma ** 2
        self._pi = pi

        self._update_w_covariance(binary_latent_factor_approximation)

        for dim_idx in range(self.d):
            self._update_w_mean(x, binary_latent_factor_approximation, dim_idx)

        self._hyper_maximisation_step()
```

Listing 14: Variational Bayes

```python
class MeanFieldApproximation:
    def __init__(
        self,
        lambda_matrix: np.ndarray,
        max_steps: int,
        convergence_criterion: float,
    ):
        """
        Initialize the Mean Field Approximation.

        Parameters
        ----------
        lambda_matrix : np.ndarray
            Parameters of the variational approximation, shape (n, k),
            where n is the number of data points and k is the number of latent variables
                .
        max_steps : int
            Maximum number of iterations for the variational expectation step.
        convergence_criterion : float
            Threshold for convergence based on the change in free energy.
        """
        self._lambda_matrix = lambda_matrix
        self.max_steps = max_steps
        self.convergence_criterion = convergence_criterion

    @property
    def lambda_matrix(self) -> np.ndarray:
        """
        Get the lambda matrix of the variational approximation.

        Returns
        -------
        np.ndarray
            The lambda matrix, shape (n, k).
        """
        return self._lambda_matrix

    @lambda_matrix.setter
    def lambda_matrix(self, value: np.ndarray) -> None:
        """
        Set the lambda matrix of the variational approximation.

        Parameters
        ----------
        value : np.ndarray
            New lambda matrix, shape (n, k).
        """
        self._lambda_matrix = value

    def lambda_matrix_exclude(self, exclude_latent_index: int) -> np.ndarray:
        """
        Exclude a specific latent variable index from the lambda matrix.

        Parameters
        ----------
        exclude_latent_index : int
            Index of the latent variable to exclude.

        Returns
        -------
        np.ndarray
            Lambda matrix excluding the specified latent variable,
            shape (n, k - 1).
        """
        return np.concatenate(
            (
                self.lambda_matrix[:, :exclude_latent_index],
                self.lambda_matrix[:, exclude_latent_index + 1 :],
            ),
            axis=1,
        )
```

```
    @property
    def expectation_s(self) -> np.ndarray:
        """
        Get the expectation of the latent variables (E[s]).

        Returns
        -------
        np.ndarray
            Expectation matrix, shape (n, k).
        """
        return self.lambda_matrix

    @property
    def expectation_ss(self) -> np.ndarray:
        """
        Get the expectation of the product of latent variables (E[s s^T]).

        Returns
        -------
        np.ndarray
            Expectation matrix, shape (k, k).
        """
        ess = self.lambda_matrix.T @ self.lambda_matrix
        np.fill_diagonal(ess, self.lambda_matrix.sum(axis=0))
        return ess

    @property
    def log_lambda_matrix(self) -> np.ndarray:
        """
        Compute the natural logarithm of the lambda matrix.

        Returns
        -------
        np.ndarray
            Log-transformed lambda matrix, shape (n, k).
        """
        return np.log(self.lambda_matrix)

    @property
    def log_one_minus_lambda_matrix(self) -> np.ndarray:
        """
        Compute the natural logarithm of (1 - lambda matrix).

        Returns
        -------
        np.ndarray
            Log-transformed complement of lambda matrix, shape (n, k).
        """
        return np.log(1 - self.lambda_matrix)

    @property
    def n(self) -> int:
        """
        Get the number of data points.

        Returns
        -------
        int
            Number of data points (n).
        """
        return self.lambda_matrix.shape[0]

    @property
    def k(self) -> int:
        """
        Get the number of latent variables.

        Returns
        -------
        int
            Number of latent variables (k).
        """
        return self.lambda_matrix.shape[1]
```

```
    def compute_free_energy(
        self,
        x: np.ndarray,
        binary_latent_factor_model: "VariationalBayes",
    ) -> float:
        """
        Compute the free energy associated with the current EM parameters and data x.

        Parameters
        ----------
        x : np.ndarray
            Data matrix, shape (n, d), where n is the number of data points and d is the
                number of dimensions.
        binary_latent_factor_model : VariationalBayes
            A binary latent factor model.

        Returns
        -------
        float
            Average free energy per data point.
        """
        expectation_log_p_x_s_given_theta = self.
            _compute_expectation_log_p_x_s_given_theta(
            x, binary_latent_factor_model
        )
        approximation_model_entropy = self._compute_approximation_model_entropy()
        return (
            expectation_log_p_x_s_given_theta + approximation_model_entropy
        ) / self.n

    def _partial_expectation_step(
        self,
        x: np.ndarray,
        binary_latent_factor_model: "VariationalBayes",
        latent_factor: int,
    ) -> np.ndarray:
        """
        Perform a partial variational E-step for a specific latent factor across all
            data points.

        Parameters
        ----------
        x : np.ndarray
            Data matrix, shape (n, d).
        binary_latent_factor_model : VariationalBayes
            A binary latent factor model.
        latent_factor : int
            Index of the latent factor to update.

        Returns
        -------
        np.ndarray
            Updated lambda vector for the specified latent factor, shape (n, 1).
        """
        lambda_matrix_excluded = self.lambda_matrix_exclude(latent_factor)
        mu_excluded = binary_latent_factor_model.mu_exclude(latent_factor)

        mu_latent = binary_latent_factor_model.mu[:, latent_factor]

        # Compute the proportion of the expectation log p(x, s | theta)
        partial_log_prob = (
            binary_latent_factor_model.precision
            * (
                x
                - 0.5 * mu_latent.T
                - lambda_matrix_excluded @ mu_excluded.T
            )
            @ mu_latent
        )

        # Compute the log-odds for the latent factor
        log_odds = np.log(
```

```
            binary_latent_factor_model.pi[0, latent_factor]
            / (1 - binary_latent_factor_model.pi[0, latent_factor])
        )

        # Total partial expectation
        total_partial_log_prob = partial_log_prob + log_odds

        # Update lambda using the sigmoid function
        lambda_vector = 1 / (
            1 + np.exp(-total_partial_log_prob)
        )
        lambda_vector = np.clip(lambda_vector, 1e-10, 1 - 1e-10)
        return lambda_vector

    def _compute_expectation_log_p_x_s_given_theta(
        self,
        x: np.ndarray,
        binary_latent_factor_model: "VariationalBayes",
    ) -> float:
        """
        Compute the expectation of log P(X, S | theta).

        Parameters
        ----------
        x : np.ndarray
            Data matrix, shape (n, d).
        binary_latent_factor_model : VariationalBayes
            A binary latent factor model.

        Returns
        -------
        float
            Expectation of log P(X, S | theta).
        """
        mu_lambda = self.lambda_matrix @ binary_latent_factor_model.mu.T

        # Compute E[s_i s_j] * mu_i * mu_j
        expectation_s_i_s_j_mu_i_mu_j = np.multiply(
            self.lambda_matrix.T @ self.lambda_matrix,
            binary_latent_factor_model.mu.T @ binary_latent_factor_model.mu,
        )

        # Calculate the expectation log P(X | S, theta)
        expectation_log_p_x_given_s_theta = (
            - (self.n * binary_latent_factor_model.d / 2)
            * np.log(2 * np.pi * binary_latent_factor_model.variance)
            - 0.5 * binary_latent_factor_model.precision
            * (
                np.sum(x ** 2)
                - 2 * np.sum(x * mu_lambda)
                + np.sum(expectation_s_i_s_j_mu_i_mu_j)
                - np.trace(expectation_s_i_s_j_mu_i_mu_j)
                + np.sum(
                    self.lambda_matrix
                    @ np.multiply(
                        binary_latent_factor_model.mu,
                        binary_latent_factor_model.mu,
                    ).T
                )
            )
        )

        # Calculate the expectation log P(S | theta)
        expectation_log_p_s_given_theta = np.sum(
            self.lambda_matrix * binary_latent_factor_model.log_pi
            + (1 - self.lambda_matrix) * binary_latent_factor_model.log_one_minus_pi
        )

        return expectation_log_p_x_given_s_theta + expectation_log_p_s_given_theta

    def _compute_approximation_model_entropy(self) -> float:
        """
        Compute the entropy of the variational approximation model.
```

```
        Returns
        -------
        float
            Model entropy.
        """
        entropy = -np.sum(
            self.lambda_matrix * self.log_lambda_matrix
            + (1 - self.lambda_matrix) * self.log_one_minus_lambda_matrix
        )
        return entropy

    def variational_expectation_step(
        self,
        x: np.ndarray,
        binary_latent_factor_model: "VariationalBayes",
    ) -> List[float]:
        """
        Perform the variational expectation step.

        Parameters
        ----------
        x : np.ndarray
            Data matrix, shape (n, d).
        binary_latent_factor_model : VariationalBayes
            A binary latent factor model.

        Returns
        -------
        List[float]
            List of free energy values at each step.
        """
        free_energy = [self.compute_free_energy(x, binary_latent_factor_model)]
        for step in range(self.max_steps):
            for latent_factor in range(binary_latent_factor_model.k):
                updated_lambda = self._partial_expectation_step(
                    x, binary_latent_factor_model, latent_factor
                )
                self.lambda_matrix[:, latent_factor] = updated_lambda
                free_energy.append(
                    self.compute_free_energy(x, binary_latent_factor_model)
                )
                energy_change = free_energy[-1] - free_energy[-2]
                if energy_change <= self.convergence_criterion:
                    break
            if free_energy[-1] - free_energy[-2] <= self.convergence_criterion:
                break
        return free_energy


def init_mean_field_approximation(
    k: int, n: int, max_steps: int, convergence_criterion: float
) -> MeanFieldApproximation:
    """
    Initialize a MeanFieldApproximation instance with random lambda matrix.

    Parameters
    ----------
    k : int
        Number of latent variables.
    n : int
        Number of data points.
    max_steps : int
        Maximum number of iterations for the variational expectation step.
    convergence_criterion : float
        Threshold for convergence based on the change in free energy.

    Returns
    -------
    MeanFieldApproximation
        Initialized MeanFieldApproximation instance.
    """
    lambda_matrix = np.random.rand(n, k)
```

```
    return MeanFieldApproximation(
        lambda_matrix=lambda_matrix,
        max_steps=max_steps,
        convergence_criterion=convergence_criterion,
    )
```

Listing 15: Mean Field Approximation

```
def automatic_relevance_determination(
    x: np.ndarray,
    binary_latent_factor_approximation: 'MeanFieldApproximation',
    a_parameter: float,
    b_parameter: float,
    k: int,
    em_iterations: int,
) -> Tuple['VariationalBayes', List[float]]:
    """
    Perform Automatic Relevance Determination (ARD) using Variational Bayes.

    This function initializes the Variational Bayes model with a Gaussian prior,
    performs the Expectation-Maximization (EM) algorithm to optimize the model
        parameters,
    and returns the optimized model along with the history of free energy values.

    Parameters
    ----------
    x : np.ndarray
        Data matrix of shape (n_samples, n_dimensions), where n_samples is the number of
             data points
        and n_dimensions is the number of observed dimensions.
    binary_latent_factor_approximation : MeanFieldApproximation
        An instance of MeanFieldApproximation representing the current variational
            approximation
        of the binary latent factors.
    a_parameter : float
        Alpha parameter for the Gamma prior on the precision (inverse variance) of the
            Gaussian prior.
    b_parameter : float
        Beta parameter for the Gamma prior on the precision (inverse variance) of the
            Gaussian prior.
    k : int
        Number of latent variables in the model.
    em_iterations : int
        Maximum number of iterations to run the EM algorithm.

    Returns
    -------
    Tuple[VariationalBayes, List[float]]
        A tuple containing:
        - The optimized VariationalBayes model.
        - A list of free energy values recorded at each EM step, representing the
            convergence progress.
    """
    # Calculate initial maximization parameters (means, variance, prior probabilities)
    _, sigma, pi = VariationalBayes.calculate_maximisation_parameters(
        x, binary_latent_factor_approximation
    )

    # Initialize Gaussian prior with provided hyperparameters
    gaussian_prior = GaussianPrior(
        a=a_parameter,
        b=b_parameter,
        d=x.shape[1],
        k=k,
    )

    # Initialize the Variational Bayes model with the Gaussian prior, variance, and
        prior probabilities
    variational_bayes_model = VariationalBayes(
        mu=gaussian_prior,
        variance=sigma**2,
        pi=pi,
    )

    # Learn binary factors using the Variational Bayes model and update the free energy
    _, variational_bayes_model, free_energy = learn_binary_factors(
        x=x,
        k=k,
        em_iterations=em_iterations,
        binary_latent_factor_model=variational_bayes_model,
```

```
        binary_latent_factor_approximation=binary_latent_factor_approximation,
    )

    return variational_bayes_model, free_energy
```

Listing 16: ARD

```python
def is_converge(
    free_energies: List[float],
    current_lambda_matrix: np.ndarray,
    previous_lambda_matrix: np.ndarray,
    free_energy_threshold: float = 1e-6,
    lambda_threshold: float = 1e-6,
) -> bool:
    """
    Determine whether the algorithm has converged based on changes in free energy
    and the lambda matrix.

    Convergence is achieved if the change in free energy between the last two iterations
    is below a specified threshold and the change in the lambda matrix (measured by
    the Frobenius norm) is also below a specified threshold.

    Parameters
    ----------
    free_energies : List[float]
        List of free energy values recorded at each iteration.
    current_lambda_matrix : np.ndarray
        The current lambda matrix after the latest iteration.
    previous_lambda_matrix : np.ndarray
        The lambda matrix from the previous iteration.
    free_energy_threshold : float, optional
        Threshold for the change in free energy to determine convergence, by default 1e
            -6.
    lambda_threshold : float, optional
        Threshold for the change in the lambda matrix (Frobenius norm) to determine
            convergence,
        by default 1e-6.

    Returns
    -------
    bool
        True if both the change in free energy and the change in lambda matrix are below
        their respective thresholds, indicating convergence. Otherwise, False.
    """
    if len(free_energies) < 2:
        # Not enough data to determine convergence
        return False

    # Calculate the absolute change in free energy
    free_energy_change = abs(free_energies[-1] - free_energies[-2])

    # Calculate the Frobenius norm of the change in lambda matrix
    lambda_change = np.linalg.norm(current_lambda_matrix - previous_lambda_matrix)

    # Check if both changes are below their respective thresholds
    return (free_energy_change <= free_energy_threshold) and (lambda_change <=
        lambda_threshold)


def learn_binary_factors(
    x: np.ndarray,
    k: int,
    em_iterations: int,
    binary_latent_factor_model: 'VariationalBayes',
    binary_latent_factor_approximation: 'MeanFieldApproximation',
) -> Tuple['MeanFieldApproximation', 'VariationalBayes', List[float]]:
    """
    Perform the Expectation-Maximization (EM) algorithm to learn binary latent factors.

    This function iteratively performs the E-step and M-step to optimize the
    variational approximation of binary latent factors and update the
    variational Bayes model. It records the free energy at each iteration to
    monitor convergence.

    Parameters
    ----------
    x : np.ndarray
        Data matrix of shape (n_samples, n_dimensions), where n_samples is the
        number of data points and n_dimensions is the number of observed dimensions.
```

```
    em_iterations : int
        Maximum number of EM iterations to perform.
    binary_latent_factor_model : VariationalBayes
        An instance of VariationalBayes representing the current model.
    binary_latent_factor_approximation : MeanFieldApproximation
        An instance of MeanFieldApproximation representing the current variational
        approximation of the binary latent factors.

    Returns
    -------
    Tuple[MeanFieldApproximation, VariationalBayes, List[float]]
        A tuple containing:
        - The updated MeanFieldApproximation instance.
        - The updated VariationalBayes model.
        - A list of free energy values recorded at each EM iteration.
    """
    # Initialize the list of free energies with the initial free energy
    free_energies: List[float] = [
        binary_latent_factor_approximation.compute_free_energy(
            x, binary_latent_factor_model
        )
    ]

    for iteration in range(1, em_iterations + 1):
        # Store the previous lambda matrix for convergence checking
        previous_lambda_matrix = np.copy(binary_latent_factor_approximation.
            lambda_matrix)

        # E-step: Update the variational approximation (lambda matrix)
        free_energy_history = binary_latent_factor_approximation.
            variational_expectation_step(
            x=x,
            binary_latent_factor_model=binary_latent_factor_model,
        )

        # M-step: Update the variational Bayes model parameters
        binary_latent_factor_model.maximisation_step(
            x=x,
            binary_latent_factor_approximation=binary_latent_factor_approximation,
        )

        # Compute and record the new free energy
        current_free_energy = binary_latent_factor_approximation.compute_free_energy(
            x, binary_latent_factor_model
        )
        free_energies.append(current_free_energy)

        # Check for convergence
        if is_converge(
            free_energies=free_energies,
            current_lambda_matrix=binary_latent_factor_approximation.lambda_matrix,
            previous_lambda_matrix=previous_lambda_matrix,
        ):
            print(f"current K = {k},"
                  f" Convergence achieved at iteration {iteration},"
                  f" Free Energy at Convergence: {current_free_energy}.")
            break


    return binary_latent_factor_approximation, binary_latent_factor_model, free_energies
```

Listing 17: Helper Functions Convergence Check EM updates

```python
import os
import matplotlib.pyplot as plt
from matplotlib.axes import Axes
from matplotlib.offsetbox import AnnotationBbox, OffsetImage


def offset_image(coord: int, image_path: str, ax: Axes) -> None:
    """
    Add an image to a specific coordinate on a Matplotlib axis.

    Parameters
    ----------
    coord : int
        The x-coordinate on the axis where the image will be placed.
    image_path : str
        The file path to the image to be added.
    ax : Axes
        The Matplotlib axis to which the image will be added.
    """
    img = plt.imread(image_path)
    offset_img = OffsetImage(img, zoom=0.72)
    offset_img.image.axes = ax

    annotation = AnnotationBbox(
        offset_img,
        (coord, 0),
        xybox=(0.0, -19.0),
        frameon=False,
        xycoords="data",
        boxcoords="offset␣points",
        pad=0,
    )
    ax.add_artist(annotation)


def plot_factors(
    variational_bayes_models: List['VariationalBayes'],
    ks: List[int],
    max_k: int,
    save_path: str,
) -> None:
    """
    Plot the latent factors and their corresponding inverse alpha values.

    This function saves images of each latent factor and creates a bar plot
    showing the inverse alpha values for each model. The first eight bars in
    each bar plot are colored blue, and the remaining bars are grey.

    Parameters
    ----------
    variational_bayes_models : List[VariationalBayes]
        A list of VariationalBayes models corresponding to different numbers of latent
            factors.
    ks : List[int]
        A list of integers representing the number of latent factors for each model.
    max_k : int
        The maximum number of latent factors across all models.
    save_path : str
        The directory path where the plots and images will be saved.
    """
    # Ensure the save directory exists
    os.makedirs(save_path, exist_ok=True)

    # Store each feature as an image for later use
    for model_idx, k in enumerate(ks):
        sorted_indices = np.argsort(
            variational_bayes_models[model_idx].gaussian_prior.alpha
        )
        for factor_idx, alpha_idx in enumerate(sorted_indices):
            fig = plt.figure(figsize=(0.3, 0.3))
            ax = plt.Axes(fig, [0.0, 0.0, 1.0, 1.0])
            ax.set_axis_off()
```

```
            fig.add_axes(ax)
            latent_factor_image = variational_bayes_models[model_idx].mu[:, alpha_idx].
                reshape(4, 4)
            ax.imshow(latent_factor_image, cmap='gray')
            image_filename = f"-latent-factor-{model_idx}-{factor_idx}.png"
            fig.savefig(os.path.join(save_path, image_filename), bbox_inches="tight")
            plt.close()

    # Create a bar plot of inverse alphas
    fig, axes = plt.subplots(len(ks), 1, figsize=(15, 2 * len(ks)))
    plt.subplots_adjust(hspace=1)

    for model_idx, k in enumerate(ks):
        sorted_indices = np.argsort(
            variational_bayes_models[model_idx].gaussian_prior.alpha
        )
        inverse_alphas = (
            1.0 / variational_bayes_models[model_idx].gaussian_prior.alpha[
                sorted_indices]
        )
        # Pad with zeros if necessary to match max_k
        inverse_alphas_padded = list(inverse_alphas) + [0] * (max_k - k)
        axes[model_idx].set_title(f"Number␣of␣Latent␣Factors␣k␣=␣{k}")
        # Define colors: first eight bars blue, others grey
        colors = ['blue' if idx < 8 else 'grey' for idx in range(max_k)]
        axes[model_idx].bar(range(max_k), inverse_alphas_padded, color=colors)
        axes[model_idx].set_xticks([])
        axes[model_idx].set_ylabel("Inverse␣  ")

    # Add feature image ticks
    for model_idx, k in enumerate(ks):
        sorted_indices = np.argsort(
            variational_bayes_models[model_idx].gaussian_prior.alpha
        )
        for factor_idx in range(len(sorted_indices)):
            image_path = os.path.join(save_path, f"-latent-factor-{model_idx}-{
                factor_idx}.png")
            offset_image(factor_idx, image_path, axes[model_idx])
            os.remove(image_path)

    fig.savefig(save_path + f"-latent-factors-comparison", bbox_inches="tight")
    plt.close()

def plot_free_energy(
    ks: List[int],
    free_energies: List[List[float]],
    model_name: str,
    save_path: str,
) -> None:
    """
    Plot the free energy over EM iterations for different models.

    Parameters
    ----------
    ks : List[int]
        List of integers representing different numbers of latent factors.
    free_energies : List[List[float]]
        List containing lists of free energy values for each model.
    model_name : str
        Name of the model for the plot title.
    save_path : str
        Directory path where the plot will be saved.
    """
    fig, ax = plt.subplots(figsize=(10, 6))
    cmap = plt.get_cmap("viridis")
    shades = np.flip(np.linspace(0.3, 0.9, len(ks)))

    for idx, k in enumerate(ks):
        color = cmap(shades[idx])
        ax.plot(free_energies[idx], label=f"K␣=␣{k}", color=color)

    ax.set_title(f"Free␣Energy␣of␣{model_name}␣Model␣Over␣Different␣Ks")
    ax.set_xlabel("Iterations")
```

```
    ax.set_ylabel("Free Energy")
    ax.legend()
    plt.savefig(save_path + "-free-energy", bbox_inches="tight")
    plt.close()
```

Listing 18: visualations

```python
def runARD(
    x: np.ndarray,
    a_parameter: int,
    b_parameter: int,
    ks: List[int],
    max_k: int,
    em_iterations: int,
    e_maximum_steps: int,
    e_convergence_criterion: float,
    save_path: str,
) -> List[List[float]]:
    """
    Execute Automatic Relevance Determination (ARD) across multiple models.

    This function runs ARD for different numbers of latent factors, initializes the
    corresponding mean field approximations, performs the Expectation-Maximization (EM)
    algorithm to optimize the Variational Bayes models, collects free energy values
    for each model, and plots the resulting latent factors along with their inverse
    alpha values.

    Parameters
    ----------
    x : np.ndarray
        Data matrix of shape (n_samples, n_dimensions), where n_samples is the
        number of data points and n_dimensions is the number of observed dimensions.
    a_parameter : float
        Alpha parameter for the Gamma prior on the precision (inverse variance) of the
            Gaussian prior.
    b_parameter : float
        Beta parameter for the Gamma prior on the precision (inverse variance) of the
            Gaussian prior.
    ks : List[int]
        List of integers representing different numbers of latent factors to evaluate.
    max_k : int
        The maximum number of latent factors across all models.
    em_iterations : int
        Number of iterations to run the EM algorithm.
    e_maximum_steps : int
        Maximum number of steps for the variational expectation step.
    e_convergence_criterion : float
        Convergence threshold for the variational expectation step.
    save_path : str
        Directory path where plots and images will be saved.

    Returns
    -------
    List[List[float]]
        A list containing lists of free energy values for each model.
    """
    variational_bayes_models: List['VariationalBayes'] = []
    free_energies: List[List[float]] = []

    for model_idx, k in enumerate(ks):
        n = x.shape[0]

        # Initialize the MeanFieldApproximation instance
        mean_field_approximation = init_mean_field_approximation(
            k,
            n,
            max_steps=e_maximum_steps,
            convergence_criterion=e_convergence_criterion,
        )

        # Run Automatic Relevance Determination (ARD)
        variational_bayes_model, free_energy = automatic_relevance_determination(
            x=x,
            binary_latent_factor_approximation=mean_field_approximation,
            a_parameter=a_parameter,
            b_parameter=b_parameter,
            k=k,
            em_iterations=em_iterations,
        )
```

```
        variational_bayes_models.append(variational_bayes_model)
        free_energies.append(free_energy)

    # Plot the latent factors and inverse alpha values
    plot_factors(
        variational_bayes_models,
        ks,
        max_k,
        save_path,
    )

    return free_energies
```

Listing 19: run ARD

```python
import os
from typing import List

import numpy as np
import matplotlib.pyplot as plt


# Constants for output directories and random seed
OUTPUTS_FOLDER = "ARD"
DEFAULT_SEED = 42


def main():
    """
    Main function to perform Automatic Relevance Determination (ARD) using Variational
        Bayes.

    This function initializes the necessary directories, sets the random seed for
        reproducibility,
    runs the ARD algorithm across different numbers of latent factors, and plots the
        free energy
    progression for each model.
    """
    # Set the random seed for reproducibility
    np.random.seed(DEFAULT_SEED)

    # Ensure the main output directory exists
    os.makedirs(OUTPUTS_FOLDER, exist_ok=True)

    # Define output directory for Question 4


    # Define parameters for ARD
    a_parameter = 1.0
    b_parameter = 0.0
    latent_factors_list = list(range(4, 25))  # ks: number of latent factors from 4 to
        24
    max_latent_factors = 24
    em_iterations = 100
    expectation_maximization_steps = 100
    expectation_convergence_threshold = 1e-6  # Changed from 0 to a small positive value
         for practical convergence
    # Run Automatic Relevance Determination (ARD)
    fe1 = runARD(
        x=data,
        a_parameter=a_parameter,
        b_parameter=b_parameter,
        ks=list(range(4, 15)),
        max_k=max_latent_factors,
        em_iterations=em_iterations,
        e_maximum_steps=expectation_maximization_steps,
        e_convergence_criterion=expectation_convergence_threshold,
        save_path=os.path.join(OUTPUTS_FOLDER, "b-1-1"),
    )
    fe2 = runARD(
        x=data,
        a_parameter=a_parameter,
        b_parameter=b_parameter,
        ks=list(range(15, 25)),
        max_k=max_latent_factors,
        em_iterations=em_iterations,
        e_maximum_steps=expectation_maximization_steps,
        e_convergence_criterion=expectation_convergence_threshold,
        save_path=os.path.join(OUTPUTS_FOLDER, "b-1-2"),
    )

    # Run Automatic Relevance Determination (ARD)
    free_energies = runARD(
        x=data,
        a_parameter=a_parameter,
        b_parameter=b_parameter,
        ks=latent_factors_list,
```

```
        max_k=max_latent_factors,
        em_iterations=em_iterations,
        e_maximum_steps=expectation_maximization_steps,
        e_convergence_criterion=expectation_convergence_threshold,
        save_path=os.path.join(OUTPUTS_FOLDER, "b-1"),
    )

    # Plot the free energy progression for each model
    model_name = "Variational Bayes"
    plot_free_energy(
        ks=latent_factors_list,
        free_energies=free_energies,
        model_name="Variational Bayes",
        save_path=os.path.join(OUTPUTS_FOLDER, "b-2"),
    )


if __name__ == "__main__":
    main()
```

Listing 20: Main Execution Block

A.4 Problem6

```
class MessagePassing:
    """
    Implements a message passing approximation for binary latent factor models using
        variational inference.

    Attributes:
        bernoulli_params (np.ndarray):
            Bernoulli parameter tensor of shape (num_data_points, num_latents,
                num_latents).
            - Off-diagonal entries represent \tilde{g}_{ij, \neg s_i}(s_j) for data
                point n.
            - Diagonal entries represent \tilde{f}_i(s_i).
    """

    def __init__(self, bernoulli_parameter_matrix: np.ndarray):
        """
        Initializes the MessagePassing with a given Bernoulli parameter tensor.

        Args:
            bernoulli_parameter_matrix (np.ndarray):
                Initial Bernoulli parameter tensor with shape
                (num_data_points, num_latents, num_latents).
        """
        self.bernoulli_parameter_matrix = bernoulli_parameter_matrix

    @property
    def expectation_s(self) -> np.ndarray:
        """
        Computes the expectation of the latent variables S.

        Returns:
            np.ndarray: Lambda matrix representing E[S].
        """
        return self.lambda_matrix

    @property
    def expectation_ss(self) -> np.ndarray:
        """
        Computes the expectation of the outer product of latent variables S.

        Returns:
            np.ndarray: E[SS^T].
        """
        ess = self.lambda_matrix.T @ self.lambda_matrix
        # Replace diagonal with correct E[s_i^2] = E[s_i] = lambda_i
        np.fill_diagonal(ess, self.lambda_matrix.sum(axis=0))
        return ess

    @property
    def log_lambda_matrix(self) -> np.ndarray:
        """
        Computes the logarithm of the lambda matrix.

        Returns:
            np.ndarray: log(E[S]).
        """
        return np.log(self.lambda_matrix)

    @property
    def log_one_minus_lambda_matrix(self) -> np.ndarray:
        """
        Computes the logarithm of (1 - lambda matrix).

        Returns:
            np.ndarray: log(1 - E[S]).
        """
        return np.log(1 - self.lambda_matrix)

    @property
    def n(self) -> int:
        """
```

```
        Returns the number of data points.

        Returns:
            int: Number of data points (n).
        """
        return self.lambda_matrix.shape[0]

    @property
    def k(self) -> int:
        """
        Returns the number of latent variables.

        Returns:
            int: Number of latent variables (k).
        """
        return self.lambda_matrix.shape[1]

    def compute_free_energy(
        self,
        x: np.ndarray,
        binary_latent_factor_model: 'LoopyBP',
    ) -> float:
        """
        Computes the free energy associated with the current EM parameters and data.

        Args:
            x (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).
            binary_latent_factor_model (LoopyBP): Binary latent factor model instance.

        Returns:
            float: Average free energy per data point.
        """
        expectation_log_p_x_s_given_theta = (
            self._compute_expectation_log_p_x_s_given_theta(
                x, binary_latent_factor_model
            )
        )
        entropy = self._compute_approximation_model_entropy()
        free_energy = (expectation_log_p_x_s_given_theta + entropy) / self.n
        return free_energy

    def _compute_expectation_log_p_x_s_given_theta(
        self,
        x: np.ndarray,
        binary_latent_factor_model: "LoopyBP",
    ) -> float:
        """
        Computes the expectation of log P(X, S | theta).

        Args:
            x (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).
            binary_latent_factor_model (LoopyBP): Binary latent factor model instance.

        Returns:
            float: Expectation of log P(X, S | theta).
        """
        # Compute expected mean: E[S] @ mu^T
        expected_mean = self.lambda_matrix @ binary_latent_factor_model.mu.T  # Shape: (
            n, d)

        # Compute E[S S^T] element-wise multiplied by mu mu^T
        expected_ss_mu_mu = np.multiply(
            self.expectation_ss,
            binary_latent_factor_model.mu.T @ binary_latent_factor_model.mu
        )  # Shape: (k, k)

        # Compute the expectation of log P(X | S, theta)
        term1 = - (self.n * binary_latent_factor_model.k / 2) * np.log(2 * np.pi *
            binary_latent_factor_model.variance)
        term2 = -0.5 * binary_latent_factor_model.precision * (
            np.sum(x ** 2)
            - 2 * np.sum(x * expected_mean)
            + np.sum(expected_ss_mu_mu)
```

```
            - np.trace(expected_ss_mu_mu)  # Remove E[s_i^2] and add correct E[s_i]
            + np.sum(
                self.lambda_matrix @ (binary_latent_factor_model.mu ** 2).T
            )
        )
        expectation_log_p_x_given_s_theta = term1 + term2

        # Compute the expectation of log P(S | theta)
        expectation_log_p_s_given_theta = np.sum(
            self.lambda_matrix * binary_latent_factor_model.log_pi
            + (1 - self.lambda_matrix) * binary_latent_factor_model.log_one_minus_pi
        )

        return expectation_log_p_x_given_s_theta + expectation_log_p_s_given_theta


    def _compute_approximation_model_entropy(self) -> float:
        """
        Computes the entropy of the approximation model.

        Returns:
            float: Entropy.
        """

        entropy = -np.sum(
            self.lambda_matrix * self.log_lambda_matrix
            + (1 - self.lambda_matrix) * self.log_one_minus_lambda_matrix
        )
        return entropy

    @property
    def lambda_matrix(self) -> np.ndarray:
        """
        Computes the lambda matrix by aggregating natural parameters and applying the
            sigmoid function.

        Returns:
            np.ndarray: Lambda matrix representing E[S].
        """
        aggregated_natural_params = self.natural_parameter_matrix.sum(axis=1)  # Sum
            over incoming messages
        lambda_matrix = 1 / (1 + np.exp(-aggregated_natural_params))
        # Numerical stability
        lambda_matrix = np.clip(lambda_matrix, 1e-10, 1 - 1e-10)
        return lambda_matrix

    @property
    def natural_parameter_matrix(self) -> np.ndarray:
        """
        Computes the natural parameters (eta) from the Bernoulli parameters.

        Returns:
            np.ndarray: Natural parameter matrix.
        """
        odds = self.bernoulli_parameter_matrix / (1 - self.bernoulli_parameter_matrix)
        natural_params = np.log(odds)
        return natural_params

    def aggregate_incoming_binary_factor_messages(
        self, node_index: int, excluded_node_index: int
    ) -> np.ndarray:
        """
        Aggregates incoming natural messages to a target node, excluding messages from a
             specific node.

        Args:
            target_node (int): Index of the target latent variable.
            exclude_node (int): Index of the node to exclude from aggregation.

        Returns:
            np.ndarray: Aggregated natural parameters for each data point.
        """
        # Sum natural parameters from all nodes except the excluded node
```

```
        incoming_before = self.natural_parameter_matrix[:, :excluded_node_index,
            node_index]
        incoming_after = self.natural_parameter_matrix[:, excluded_node_index + 1 :,
            node_index]
        aggregated = np.sum(incoming_before, axis=1) + np.sum(incoming_after, axis=1)
        return aggregated.reshape(-1)

    @staticmethod
    def calculate_bernoulli_parameter(
            natural_parameter_matrix: np.ndarray
    ) -> np.ndarray:
        """
        Computes Bernoulli parameters from natural parameters using the sigmoid function
            .

        Args:
            natural_params (np.ndarray): Natural parameter matrix.

        Returns:
            np.ndarray: Updated Bernoulli parameters.
        """
        bernoulli_parameter = 1 / (1 + np.exp(-natural_parameter_matrix))
        # Numerical stability
        bernoulli_parameter = np.clip(bernoulli_parameter, 1e-10, 1 - 1e-10)
        return bernoulli_parameter

    def variational_expectation_step(
        self, x: np.ndarray, binary_latent_factor_model: 'LoopyBP'
    ) -> List[float]:
        """
        Performs the variational expectation step, updating singleton and binary factors
             iteratively.

        Args:
            x (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).
            binary_latent_factor_model (LoopyBP): Binary latent factor model instance.

        Returns:
            List[float]: List of free energies after each update.
        """
        free_energies = [self.compute_free_energy(x, binary_latent_factor_model)]

        for i in range(self.k):
            # Update singleton factor for latent variable i
            singleton_natural = self.calculate_singleton_message_update(
                LoopyBP=binary_latent_factor_model,
                x=x,
                i=i,
            )
            self.bernoulli_parameter_matrix[:, i, i] = self.
                calculate_bernoulli_parameter(singleton_natural)
            free_energies.append(self.compute_free_energy(x, binary_latent_factor_model)
                )

            # Update binary factors between latent variables i and j
            for j in range(i):
                # Update message from i to j
                binary_natural_ij = self.calculate_binary_message_update(
                    LoopyBP=binary_latent_factor_model,
                    x=x,
                    i=i,
                    j=j,
                )
                self.bernoulli_parameter_matrix[:, i, j] = self.
                    calculate_bernoulli_parameter(binary_natural_ij)

                # Update message from j to i
                binary_natural_ji = self.calculate_binary_message_update(
                    LoopyBP=binary_latent_factor_model,
                    x=x,
                    i=i,
                    j=j,
                )
```

```
                self.bernoulli_parameter_matrix[:, j, i] = self.
                    calculate_bernoulli_parameter(binary_natural_ji)

                free_energies.append(self.compute_free_energy(x,
                    binary_latent_factor_model))

        return free_energies


    def calculate_binary_message_update(
        self,
        x: np.ndarray,
        LoopyBP: "LoopyBP",
        i: int,
        j: int,
    ) -> float:
        """
        Updates the natural parameters for a binary factor between two latent variables.

        Args:
            x (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).
            latent_model (LoopyBP): Binary latent factor model instance.
            i, source (int): Index of the source latent variable.
            j, target (int): Index of the target latent variable.

        Returns:
            np.ndarray: Updated natural parameters for the binary factor.
        """
        # Compute natural parameters excluding the target node
        natural_parameter_i_not_j = LoopyBP.b_index(x=x, node_index=i)
        natural_parameter_i_not_j += self.aggregate_incoming_binary_factor_messages(
            node_index=i, excluded_node_index=j)

        # Retrieve interaction weight between source and target
        w_i_j = LoopyBP.w_matrix_index(i, j)

        # Update natural parameters for the binary factor
        updated_natural = np.log1p(np.exp(w_i_j + natural_parameter_i_not_j)) - np.log1p
            (np.exp(natural_parameter_i_not_j))
        return updated_natural

    @staticmethod
    def calculate_singleton_message_update(
        x: np.ndarray,
        LoopyBP: "LoopyBP",
        i: int,
    ) -> float:
        """
        Updates the natural parameters for a singleton factor of a latent variable.

        Args:
            x (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).
            latent_model (LoopyBP): Binary latent factor model instance.
            i, latent_idx (int): Index of the latent variable to update.

        Returns:
            np.ndarray: Updated natural parameters for the singleton factor.
        """
        # Singleton update does not require message aggregation
        return LoopyBP.b_index(x=x, node_index=i)


def init_message_passing(k: int, n: int) -> MessagePassing:
    """
    Message passing initialisation

    :param k: number of latent variables
    :param n: number of data points
    :return: message passing
    """
    bernoulli_parameter_matrix = np.random.random(size=(n, k, k))
    return MessagePassing(bernoulli_parameter_matrix)
```

Listing 21: Message Passing

```python
class LoopyBP:
    def __init__(
        self,
        mu: np.ndarray,
        sigma: float,
        pi: np.ndarray,
    ):
        """
        Initializes the LoopyBP model with mean, variance, and prior probabilities.

        Args:
            mu (np.ndarray): Mean matrix of shape (num_dimensions, num_latents).
            sigma (float): Standard deviation parameter.
            pi (np.ndarray): Prior probabilities of latent variables, shape (1,
                num_latents).
        """
        super().__init__()
        self._mu = mu  # Shape: (num_dimensions, num_latents)
        self._sigma = sigma
        self._pi = pi   # Shape: (1, num_latents)
    # ==========================
    # Property Definitions
    # ==========================

    @property
    def d(self) -> int:
        return self.mu.shape[0]

    @property
    def k(self) -> int:
        return self.mu.shape[1]

    @property
    def mu(self) -> np.ndarray:
        """Mean matrix."""
        return self._mu

    @mu.setter
    def mu(self, value: np.ndarray) -> None:
        self._mu = value

    @property
    def sigma(self) -> float:
        """Standard deviation."""
        return self._sigma

    @sigma.setter
    def sigma(self, value: float) -> None:
        self._sigma = value

    @property
    def pi(self) -> np.ndarray:
        """Prior probabilities of latent variables."""
        return self._pi

    @pi.setter
    def pi(self, value: np.ndarray) -> None:
        self._pi = value

    @property
    def variance(self) -> float:
        """Variance, square of the standard deviation."""
        return self.sigma ** 2

    @property
    def precision(self) -> float:
        """Precision, inverse of variance."""
        return 1.0 / self.variance

    @property
    def log_pi(self) -> np.ndarray:
        """Logarithm of prior probabilities."""
```

```
        return np.log(self.pi)

    @property
    def log_one_minus_pi(self) -> np.ndarray:
        """Logarithm of (1 - prior probabilities)."""
        return np.log(1 - self.pi)

    @property
    def log_pi_ratio(self) -> np.ndarray:
        """Log ratio of pi to (1 - pi)."""
        return self.log_pi - self.log_one_minus_pi

    @property
    def num_dimensions(self) -> int:
        """Number of dimensions in the data."""
        return self.mu.shape[0]

    @property
    def num_latents(self) -> int:
        """Number of latent variables."""
        return self.mu.shape[1]

    # ============================
    # Static Methods
    # ============================

    @staticmethod
    def calculate_maximisation_parameters(
        x: np.ndarray,
        approximation: MessagePassing,
    ) -> Tuple[np.ndarray, float, np.ndarray]:
        """
        Performs the Maximization (M) step to update model parameters based on
            expectations.

        Args:
            x (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).
            approximation (MessagePassing): Variational approximation instance.

        Returns:
            Tuple[np.ndarray, float, np.ndarray]:
                - Updated mu matrix.
                - Updated sigma (float).
                - Updated pi vector.
        """
        return m_step(
            X=x,
            ES=approximation.expectation_s,
            ESS=approximation.expectation_ss,
        )

    # ============================
    # Instance Methods
    # ============================

    def maximisation_step(
        self,
        x: np.ndarray,
        binary_latent_factor_approximation: MessagePassing,
    ) -> None:
        """
        Updates the model parameters by performing the Maximization step.

        Args:
            x (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).
            binary_latent_factor_approximation (MessagePassing): Variational
                approximation instance.
        """
        mu_updated, sigma_updated, pi_updated = self.calculate_maximisation_parameters(
            x, binary_latent_factor_approximation
        )
        self.mu = mu_updated
        self.sigma = sigma_updated
```

```
        self.pi = pi_updated

    def w_matrix(self) -> np.ndarray:
        """
        Computes the weight matrix for Loopy Belief Propagation.

        Returns:
            np.ndarray: Weight matrix of shape (num_latents, num_latents).
        """
        return -self.precision * (self.mu.T @ self.mu)

    def w_matrix_index(self, i, j) -> float:
        """
        Retrieves the weight between two specific latent variables.

        Args:
            source (int): Index of the source latent variable.
            target (int): Index of the target latent variable.

        Returns:
            float: Weight value between the specified latent variables.
        """
        return -self.precision * (self.mu[:, i] @ self.mu[:, j])

    def b(self, x: np.ndarray) -> np.ndarray:
        """
        Computes the 'b' term in LoopyBP for all data points.

        Args:
            data (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).

        Returns:
            np.ndarray: Computed 'b' matrix of shape (num_data_points, num_latents).
        """
        return -(
            self.precision * x @ self.mu
            + self.log_pi_ratio
            - 0.5 * self.precision * np.sum(self.mu ** 2, axis=0)
        )

    def b_index(self, x: np.ndarray, node_index: int) -> np.ndarray:
        """
        Computes the 'b' term for a specific node in LoopyBP across all data points.

        Args:
            x (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).
            node_index (int): Index of the target latent variable.

        Returns:
            np.ndarray: Computed 'b' vector for the specified node, shape (
                num_data_points,).
        """
        return -(
            self.precision * x @ self.mu[:, node_index]
            + (self.log_pi[0, node_index] - self.log_one_minus_pi[0, node_index])
            - 0.5 * self.precision * (self.mu[:, node_index] @ self.mu[:, node_index])
        ).reshape(-1)




def init_LoopyBP(
    x: np.ndarray,
    binary_latent_factor_approximation: "MessagePassing",
) -> LoopyBP:
    """
    Initialise the Loopy BP model by running a single m step with the parameters of a
        given binary latent factor approximation
    """
    mu, sigma, pi = LoopyBP.calculate_maximisation_parameters(
        x, binary_latent_factor_approximation
```

```
    )
    return LoopyBP(mu=mu, sigma=sigma, pi=pi)
```

Listing 22: LoopyBP

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

def runLoopyBP(
        x: np.ndarray,
        k: int,
        em_iterations: int,
        save_path: str
) -> None:
    """
    Executes the Loopy Belief Propagation (LoopyBP) algorithm for binary latent factor
        models,
    including initialization, expectation-maximization (EM) iterations, and
        visualization
    of latent features and free energy over iterations.

    Args:
        x (np.ndarray): Data matrix of shape (num_data_points, num_dimensions).
        k (int): Number of latent variables (factors) in the model.
        em_iterations (int): Number of EM iterations to perform.
        save_path (str): Base path for saving generated plots.

    """
    n = x.shape[0]

    # Initialize message passing and LoopyBP models
    message_passing = init_message_passing(k, n)
    LoopyBP = init_LoopyBP(x, message_passing)

    # Plot and save initial latent features
    plot_latent_features(
        mu=LoopyBP.mu,
        num_latents=k,
        feature_shape=(4, 4),
        title="Initial Latent Features (Loopy BP)",
        save_path=f"{save_path}-init-latent-factors.png",
    )


    # Perform EM Updates
    message_passing, LoopyBP, free_energy = learn_binary_factors(
        x=x,
        k=k,
        em_iterations=em_iterations,
        binary_latent_factor_model=LoopyBP,
        binary_latent_factor_approximation=message_passing,
    )

    # Plot and save learned latent features
    plot_latent_features(
        mu=LoopyBP.mu,
        num_latents=k,
        feature_shape=(4, 4),
        title="Learned Latent Features (Loopy BP)",
        save_path=f"{save_path}-learned-latent-factors.png",
    )


    # Plot and save free energy over EM iterations
    plot_free_energy(
        free_energy=free_energy,
        title="Free Energy (Loopy BP)",
        xlabel="Iterations",
        ylabel="Free Energy",
        save_path=f"{save_path}-free-energy.png",
    )
```

Listing 23: run LoopyBP

```
import os
from dataclasses import asdict


import numpy as np
import pandas as pd

# Constants for output directories and random seed
OUTPUTS_FOLDER = "LoopyBP"
DEFAULT_SEED = 43

if __name__ == "__main__":
    np.random.seed(DEFAULT_SEED)

    if not os.path.exists(OUTPUTS_FOLDER):
        os.makedirs(OUTPUTS_FOLDER)


    x = data
    k = 8
    em_iterations = 100
    e_maximum_steps = 50
    e_convergence_criterion = 0

    runLoopyBP(x, k, em_iterations, save_path=os.path.join(OUTPUTS_FOLDER, "all"))
```

Listing 24: Main Execution Block