University College London

Department of Computer Science

COMP0085: Approximate Inference

Summative Assessment

Author:
Mr. Frederico Wieser

Student Number:
18018699

Last Updated:
January 13, 2025

Contents

1 Biochemical Pathway Analysis

1.1 Directed Acyclic Graph (DAG) Representation

Using the information our friend has given us, we draw the following directed acyclic graph (DAG):

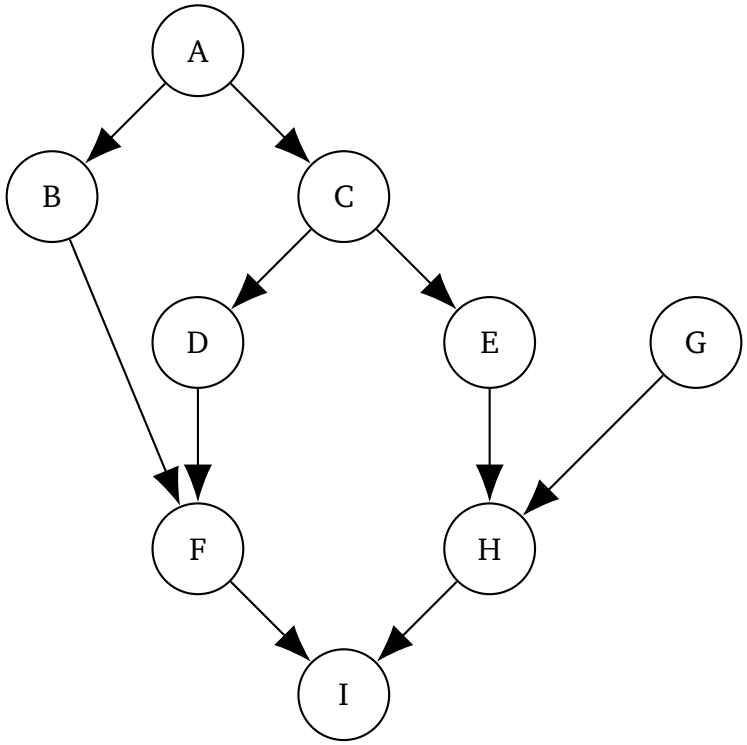


Figure 1: Bayesian Network showing how the concentrations of the 9 species A - I depend on one another.

1.2 Moralisation, Triangulation, and Junction Tree

Moralisation, is the process of converting a directed acyclic graph into an undirected graph by removing the directionality of all current nodes and adding undirected edges between co-parents of each node.

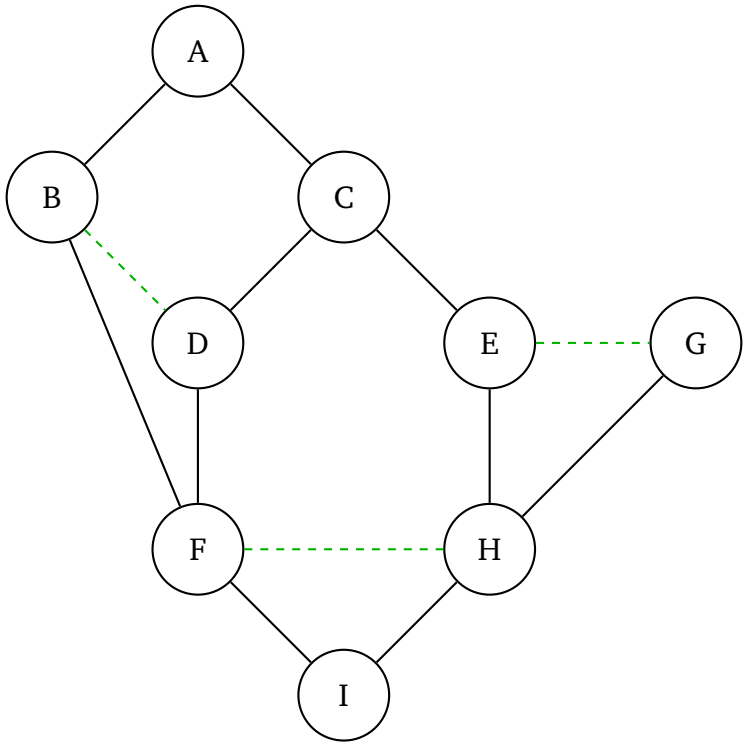


Figure 2: Moralised Markov Random Field of the original Bayesian Network, showing how the concentrations of the 9 species A - I depend on one another.

Triangulation is the process by which we add edges so that there are no chordless cycles of length greater than 3.

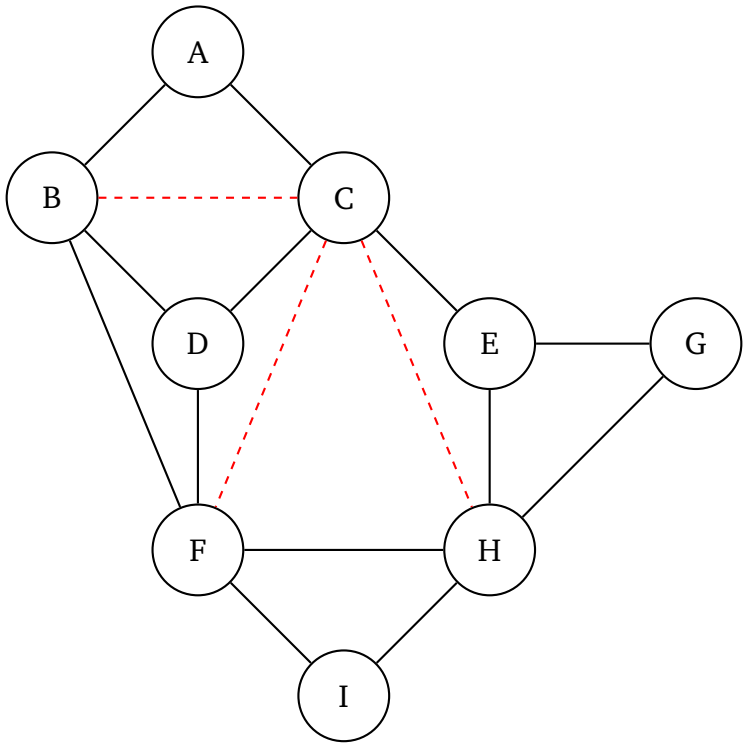


Figure 3: Triangulated Markov Random Field of the moralised version, showing how the concentrations of the 9 species A - I depend on one another.

The maximal cliques of the triangulated graph become the Junction Tree ”clique nodes”, where adjacency is determined by intersections of variables among cliques. Example of clique selection (one possible set): $ABC, BCDF, CFH, CEHG, FHI$. Then we connect them so that shared variables appear in the ”separator” nodes on each edge. In this case, we will select and showcase another Junction Tree of a different set of clique nodes.

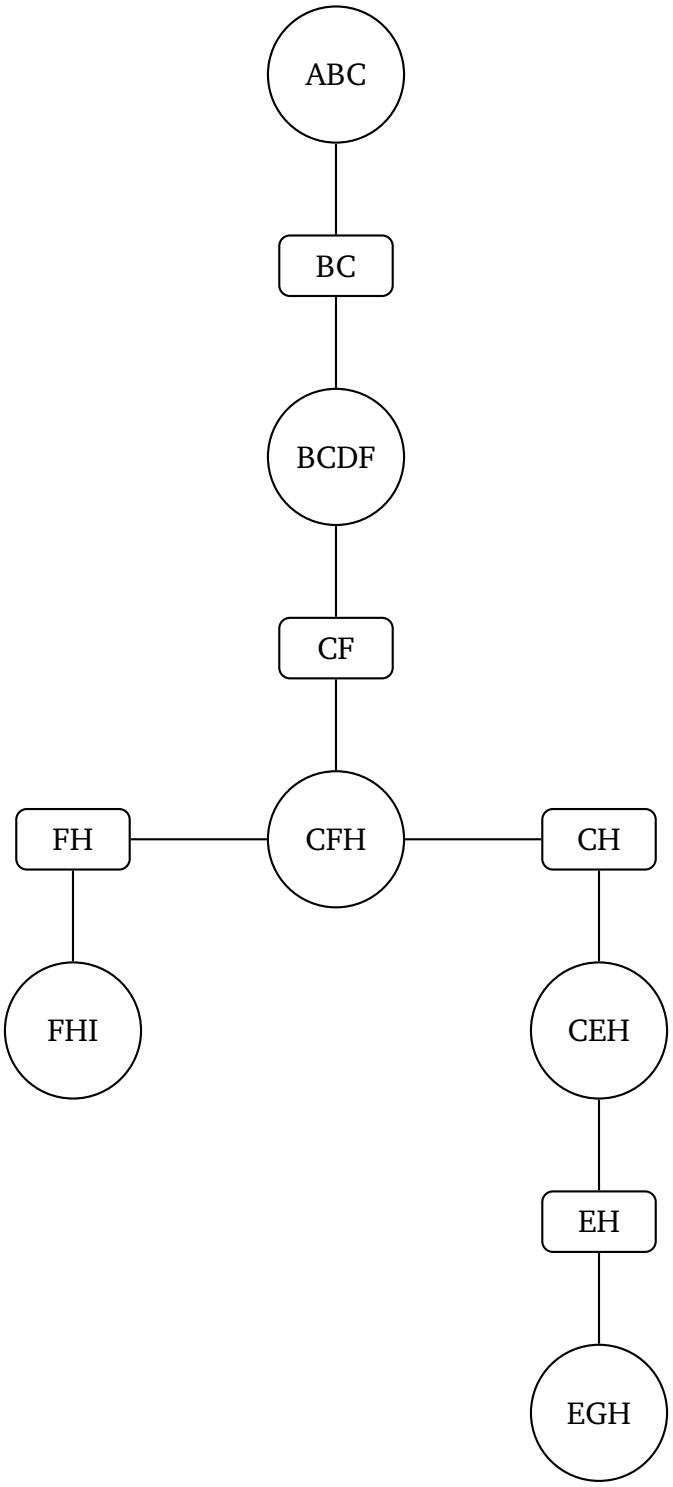
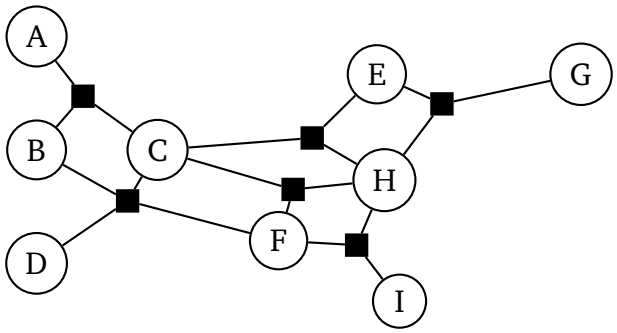


Figure 4: Junction Tree of the Triangulated Markov Random Field, showing how the concentrations of the 9 species A - I depend on one another.

In the factor graph, each clique node becomes a 'factor' (square) and each variable is a circle. A variable is connected to a factor if that variable is in the corresponding clique.



1.3 Identifying the Smallest Sufficient Set of Molecules

In order to determine the smallest (but non-unique) set of molecules such that all other concentrations become independent when conditioned on this set, we analyse the structure of the directed acyclic graph (DAG) using D-separation principles.

By considering Markov blankets and inspecting the graphical structure, we find that the minimal conditioning set is $\{A, D, E, F, H\}$. This selection ensures that conditioning on these variables renders the remaining concentrations—$\{B, C, G, I\}$—independent. Specifically, A captures the primary source of dependency, D and E account for key enzymatic influences, while F and H govern the final stages leading to I. Since these nodes fully mediate information flow, any remaining species become independent upon conditioning on this set.

1.4 Factor Analysis: Expected Number of Factors

Given the model assumes a linear-Gaussian structure, we can write that the concentration perturbations are given by:

$$\delta[x] = \Lambda \delta[z] + \epsilon,$$

where $\delta[z]$ are the parent concentration perturbations, Λ is the loading matrix encoding dependencies, and ϵ represents reaction-specific Gaussian noise. Observations are available for $\delta[B], \delta[D], \delta[E]$, and $\delta[G]$. Examining the DAG, we observe that B depends on A, while both D and E depend on C. Notably, G has no parents and thus does not contribute to a common latent factor.

By rewriting the system in matrix form, we obtain:

$$\begin{bmatrix} \delta[B] \\ \delta[D] \\ \delta[E] \end{bmatrix} = \begin{bmatrix} \Lambda_{AB} & 0 \\ 0 & \Lambda_{CD} \\ 0 & \Lambda_{CE} \end{bmatrix} \begin{bmatrix} \delta[A] \\ \delta[C] \end{bmatrix} + \begin{bmatrix} \epsilon_B \\ \epsilon_D \\ \epsilon_E \end{bmatrix}.$$

From this structure, it is evident that factor analysis should recover two latent factors, corresponding to $\delta[A]$ and $\delta[C]$. These factors explain the observed dependencies in the measured

perturbations. Since G has no parents, its variance is explained purely by noise and does not contribute to a common latent factor. Thus, in expectation, the number of recoverable factors is two, aligning with the number of direct parent nodes influencing the observed variables.

1.5 Identifiability of Concentration Perturbations and Linear Weights

The above factor analysis recovers the latent factors $\delta[A]$ and $\delta[C]$, this allows for estimating the weights connecting these latent factors to their directly observed children: $\Lambda_{AB}, \Lambda_{CD}$, and Λ_{CE}. Due to the nature of factor analysis, these weights are only able to be calculated up to an unknown scale factor.

For the species F, H, and I, their perturbations remain unidentifiable because they are not directly observed and also not recoverable from the latent factors. When performing factor analysis, we assume that latent factors are uncorrelated, meaning that dependencies between latent factors, such as between A and C, are not explicitly captured. Looking at the structure of the original DAG, it suggests a causal influence of A on C though, while factor analysis only detects shared variance rather than explicit directed dependencies. In essence, while factor analysis provides valuable insights into parent-child relationships, it does not allow for full identification of all species perturbations or precise determination of all weights. Instead, only the relationships between recovered latent factors and observed variables can be estimated, subject to an arbitrary scaling factor.

2 Bayesian Linear & Gaussian Process Regression

2.1 Bayesian Linear Regression: Posterior Mean and Covariance

Given the Bayesian linear regression model:

$$f(t) = at + b + \epsilon(t), \quad \epsilon(t) \sim \mathcal{N}(0, 1),$$

we assume a prior:

$$p(\mathbf{w}) = \mathcal{N}(\boldsymbol{\mu}_{\mathbf{w}}, \Sigma_{\mathbf{w}}),$$

where

$$\boldsymbol{\mu}_{\mathbf{w}} = \begin{bmatrix} 0 \\ 360 \end{bmatrix}, \quad \Sigma_{\mathbf{w}} = \begin{bmatrix} 10^2 & 0 \\ 0 & 100^2 \end{bmatrix}.$$

The likelihood follows:

$$p(\mathbf{y}|\mathbf{w}) = \prod_{t=1}^{T} \mathcal{N}(y_t|\mathbf{x}_t^T\mathbf{w}, 1),$$

where

$$\mathbf{x}_t = \begin{bmatrix} t \\ 1 \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} t_1 & 1 \\ t_2 & 1 \\ \vdots & \vdots \\ t_T & 1 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_T \end{bmatrix}.$$

By Bayes' rule:

$$p(\mathbf{w}|\mathcal{D}) \propto p(\mathbf{y}|\mathbf{w})p(\mathbf{w}).$$

Expanding:

$$p(\mathbf{y}|\mathbf{w}) \propto \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})\right),$$

$$p(\mathbf{w}) \propto \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_{\mathbf{w}})^T\Sigma_{\mathbf{w}}^{-1}(\mathbf{w} - \boldsymbol{\mu}_{\mathbf{w}})\right).$$

Thus:

$$p(\mathbf{w}|\mathcal{D}) \propto \exp\left(-\frac{1}{2}(\mathbf{y}-\mathbf{X}\mathbf{w})^T(\mathbf{y}-\mathbf{X}\mathbf{w})\right)\exp\left(-\frac{1}{2}(\mathbf{w}-\boldsymbol{\mu}_\mathbf{w})^T\Sigma_\mathbf{w}^{-1}(\mathbf{w}-\boldsymbol{\mu}_\mathbf{w})\right).$$

Expanding the quadratic terms:

$$(\mathbf{y}-\mathbf{X}\mathbf{w})^T(\mathbf{y}-\mathbf{X}\mathbf{w}) = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\mathbf{w} + \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w},$$

$$(\mathbf{w}-\boldsymbol{\mu}_\mathbf{w})^T\Sigma_\mathbf{w}^{-1}(\mathbf{w}-\boldsymbol{\mu}_\mathbf{w}) = \mathbf{w}^T\Sigma_\mathbf{w}^{-1}\mathbf{w} - 2\boldsymbol{\mu}_\mathbf{w}^T\Sigma_\mathbf{w}^{-1}\mathbf{w} + \boldsymbol{\mu}_\mathbf{w}^T\Sigma_\mathbf{w}^{-1}\boldsymbol{\mu}_\mathbf{w}.$$

Rewriting:

$$p(\mathbf{w}|\mathcal{D}) \propto \exp\left(-\frac{1}{2}\left[\mathbf{w}^T(\mathbf{X}^T\mathbf{X}+\Sigma_\mathbf{w}^{-1})\mathbf{w} - 2\mathbf{w}^T(\mathbf{X}^T\mathbf{y}+\Sigma_\mathbf{w}^{-1}\boldsymbol{\mu}_\mathbf{w})\right]\right).$$

Comparing with the standard multivariate Gaussian quadratic form:

$$(\mathbf{w}-\boldsymbol{\mu}_\mathbf{w}^{\text{post}})^T\Sigma_\mathbf{w}^{\text{post}^{-1}}(\mathbf{w}-\boldsymbol{\mu}_\mathbf{w}^{\text{post}}),$$

we identify:

$$\Sigma_\mathbf{w}^{\text{post}} = \left(\mathbf{X}^T\mathbf{X}+\Sigma_\mathbf{w}^{-1}\right)^{-1},$$

$$\boldsymbol{\mu}_\mathbf{w}^{\text{post}} = \Sigma_\mathbf{w}^{\text{post}}\left(\mathbf{X}^T\mathbf{y}+\Sigma_\mathbf{w}^{-1}\boldsymbol{\mu}_\mathbf{w}\right).$$

Thus, the posterior distribution follows:

$$\mathbf{w}|\mathcal{D} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{w}^{\text{post}}, \Sigma_\mathbf{w}^{\text{post}}).$$

Using the code written below, we arrive at the following final values:

$$\text{Posterior Mean:} \quad \mu_a^{\text{post}} = 1.82846, \quad \mu_b^{\text{post}} = 334.20378.$$

$$\text{Posterior Covariance:} \quad \Sigma_\mathbf{w}^{\text{post}} = \begin{bmatrix} 1.38240460\times10^{-5} & -2.87420300\times10^{-4} \\ -2.87420300\times10^{-4} & 7.97584986\times10^{-3} \end{bmatrix}.$$

CODE:

```python
# Load and preprocess CO2 data
df = pd.read_csv("co2.txt", delim_whitespace=True, comment='#', header=None,
                 names=["year", "month", "decimal", "average", "trend"])
t, y = df["decimal"].values, df["average"].values
t_min = t.min()
t -= t_min # Normalize time for numerical stability

# Define prior mean and covariance
mu_prior = np.array([0, 360])
Sigma_prior = np.diag([100, 10000])

# Compute posterior
X = np.vstack([t, np.ones_like(t)]).T
Sigma_post = np.linalg.inv(X.T @ X + np.linalg.inv(Sigma_prior))
mu_post = Sigma_post @ (X.T @ y + np.linalg.inv(Sigma_prior) @ mu_prior)

# Print results
print(f"Posterior Mean: a = {mu_post[0]:.5f}, b = {mu_post[1]:.5f}")
print("Posterior Covariance:\n", Sigma_post)
```

2.2 Residual Analysis: Model Assumptions vs. Reality

The residuals, defined as $g_{\text{obs}}(t) = y - (\alpha_{\text{MAP}} t + b_{\text{MAP}})$, exhibit periodic behaviour, indicating autocorrelation and a dependency on time. This contradicts the assumption that $\epsilon(t) \sim \mathcal{N}(0,1)$ represents independent and identically distributed Gaussian noise.

Examining the residual time series reveals a seasonal pattern, likely arising from periodic fluctuations in CO_2 levels. The histogram of residuals indicates a mean close to zero, which aligns with prior expectations, but the variance deviates from unity, contradicting the original assumption of Gaussian noise that the question laid out. These findings suggest that a simple linear model may be inadequate, and a more flexible approach, such as Gaussian Processes, could better capture the underlying structure in the data.

CODE:

```python
# Extract MAP estimates for a and b
a_MAP, b_MAP = mu_post

# Compute residuals
g_obs = y - (a_MAP * t + b_MAP)

# Plot residuals over time
plt.figure(figsize=(10, 5))
plt.plot(t + t_min, g_obs, label="Residuals", color="C0")  # Shift time back
plt.xlabel("Decimal Year")
plt.ylabel("Residuals")
```

```
plt.title("Residuals over Time")
plt.legend()
plt.savefig("q2_residuals_over_time.png")
plt.show()

# Histogram of residuals
plt.figure(figsize=(10, 5))
sns.histplot(g_obs, bins=50, color="C0", edgecolor="k", alpha=0.7)
plt.xlabel("Residual Value")
plt.ylabel("Frequency")
plt.title("Histogram of Residuals")
plt.savefig("q2_residual_histogram.png")
plt.show()

# PDF comparison: residual distribution vs. N(0,1)
plt.figure(figsize=(10, 5))
sns.histplot(g_obs, bins=50, stat="density", color="C0", edgecolor="k", alpha=0.7,
 ↪  label="Residuals")
x_vals = np.linspace(np.min(g_obs), np.max(g_obs), 100)
plt.plot(x_vals, norm.pdf(x_vals, 0, 1), 'r', label=r"$\mathcal{N}(0,1)$ PDF")  #
 ↪  Standard Normal PDF
plt.xlabel("Residual Value")
plt.ylabel("Probability Density")
plt.title("Residual Distribution vs. Standard Normal")
plt.legend()
plt.savefig("q2_residual_pdf_comparison.png")
plt.show()

# Print residual statistics
print(f"Mean of residuals: {np.mean(g_obs):.5f}")
print(f"Standard deviation of residuals: {np.std(g_obs):.5f}")
```

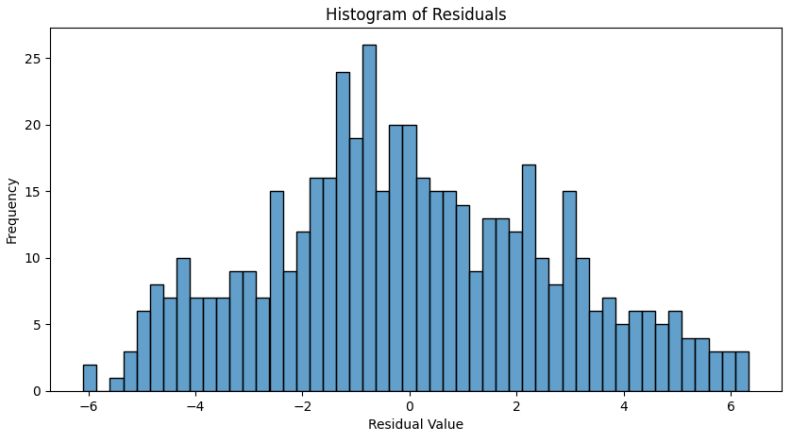


Figure 5: Histogram of residuals showing their distribution. The shape suggests deviations from normality and possible heteroscedasticity.

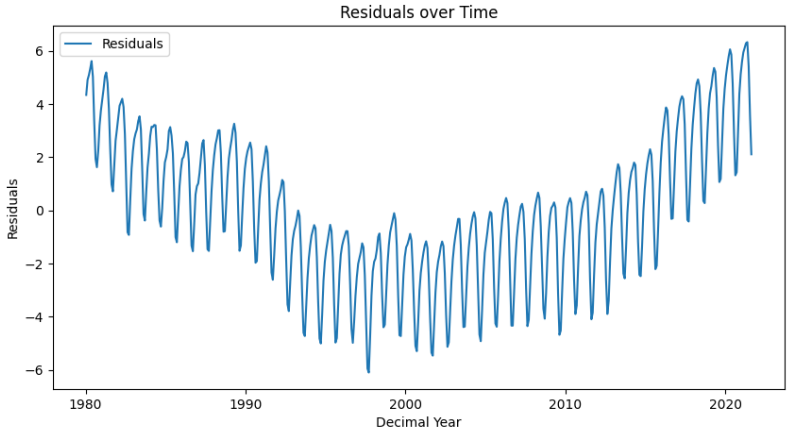


Figure 6: Residuals plotted over time, displaying a clear seasonal pattern that suggests autocorrelation and dependence on time.

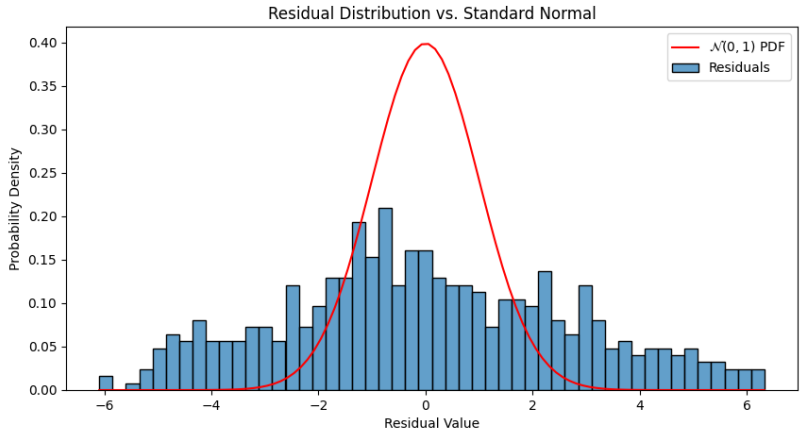


Figure 7: Comparison between the residual distribution and a standard normal distribution. The observed residuals deviate from a standard normal assumption.

Furthermore, here are the calculated residual statistics:

$$\begin{aligned} \text{Mean of residuals:} \quad & -0.00001 \\ \text{Standard deviation of residuals:} \quad & 2.67837 \end{aligned}$$

2.3 Sampling from a Gaussian Process

To sample from a Gaussian Process (GP), we first define a covariance kernel function $k(s,t)$, which models the correlation between function values at different points. Using this kernel, we construct the covariance matrix K, where each element is given by $K_{ij} = k(x_i, x_j)$. Once the covariance matrix is established, we sample from the multivariate Gaussian distribution $\mathcal{N}(0, K)$, generating functions that follow the GP distribution.

This method is efficient, utilizing NumPy broadcasting for fast matrix computation, while also maintaining the probabilistic interpretation of GPs.

CODE:

```python
def samples_from_GP(kernel_function, x_points, num_samples=1, **kernel_params):
    """Generate samples from a Gaussian Process with a given covariance kernel."""
    num_points = len(x_points)
    X1, X2 = np.meshgrid(x_points, x_points, indexing="ij")
    covariance_matrix = kernel_function(X1, X2, **kernel_params)

    return np.random.multivariate_normal(mean=np.zeros(num_points),
    ↪  cov=covariance_matrix, size=num_samples)
```

2.4 Visualizing GP Samples for Different Kernels

The kernel encodes properties like smoothness, periodicity, and flexibility of the sampled functions.

The kernel used here is:

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

```python
def kernel_function(s, t, theta=1.0, sigma=1.0, phi=1.0, eta=1.0, tau=1.0,
↪  zeta=0.1):
    """Compute the covariance kernel function k(s, t)."""
    periodic_term = np.exp(-2 * (np.sin(np.pi * (s - t) / tau) ** 2) / sigma**2)
    squared_exp_term = phi**2 * np.exp(-0.5 * ((s - t) / eta) ** 2)
    noise_term = (zeta**2) * (s == t)

    return (theta**2) * (periodic_term + squared_exp_term) + noise_term
```

where:

- θ scales function magnitude.
- σ controls periodicity smoothness.
- ϕ weights the squared-exponential (non-periodic) component.
- η adjusts the lengthscale of local correlations.
- τ sets the periodic cycle length.
- ζ governs the noise term.

By varying these parameters:

- Increasing θ amplifies function values.
- Larger σ smooths periodic variations.
- Increasing ϕ strengthens non-periodic structure.
- Smaller η makes functions fluctuate more locally.
- Higher ζ introduces more noise, reducing smoothness.

Through visualization, we confirm that the kernel structure aligns with the assumptions about residuals, reinforcing the need for GP Regression to capture CO_2 trends effectively.

Code for testing different hyperparamaater settings.

```
ef plot_gp_for_hyperparams(
    kernel_func,
    x_points,
    base_params,
    param_names,
    param_ranges,
    param_colors,
    n_samples_per_setting=5
):
    """
    Creates a grid of subplots with len(param_names) rows (one row per
    ↪ hyperparameter)
    and len(param_ranges[i]) columns (the number of values for each
    ↪ hyperparameter).
    We draw multiple GP samples in each subplot to visualize variability.
    """
    num_hparams = len(param_names)              # Rows
    num_values  = len(param_ranges[0])          # Columns (assuming all ranges have
    ↪ same length)

    fig, axes = plt.subplots(
        nrows=num_hparams,
        ncols=num_values,
        figsize=(4 * num_values, 3 * num_hparams),
        sharex=False,
        sharey=False
    )

    for row_idx, (pname, pcolor, prange) in enumerate(zip(param_names,
    ↪ param_colors, param_ranges)):
        for col_idx, pval in enumerate(prange):
            ax = axes[row_idx, col_idx] if num_hparams > 1 else axes[col_idx]

            # Update the one hyperparameter
            current_params = base_params.copy()
            current_params[pname] = pval
```

```
33
34              # Sample from GP
35              gp_samples = samples_from_GP(
36                  kernel_func,
37                  x_points,
38                  n_samples_per_setting,
39                  **current_params
40              )
41
42              # Plot each sample
43              for sample in gp_samples:
44                  ax.plot(x_points, sample, color=pcolor, alpha=0.5)
45
46              ax.set_title(f"{pname}={pval:.2f}", fontsize=10)
47              ax.set_xlabel("t")
48              ax.set_ylabel("f(t)")
49              ax.grid(True)
50
51      fig.suptitle("GP Samples for Different Hyperparameters", fontsize=16)
52      plt.tight_layout()
53      plt.savefig("q2_gp_hyperparam_samples.png")
54      plt.show()
55
56  # Number of values per parameter
57  samples = 4
58
59  # Define each hyperparameter's range (3 values for demonstration)
60  theta_range = np.linspace(0.01, 5, samples)
61  phi_range   = np.linspace(0.01, 5, samples)
62  tau_range   = np.linspace(0.01, 5, samples)
63  eta_range   = np.linspace(0.01, 5, samples)
64  sigma_range = np.linspace(0.01, 5, samples)
65  zeta_range  = np.linspace(0.01, 5, samples)
66
67  # Put them into lists for iteration
68  param_names  = ["theta", "phi", "tau", "eta", "sigma", "zeta"]
69  param_ranges = [
70      theta_range,
71      phi_range,
72      tau_range,
73      eta_range,
74      sigma_range,
75      zeta_range
76  ]
77
78  # Assign a color for each hyperparameter (6 unique colors)
79  param_colors = ["blue", "red", "green", "purple", "orange", "brown"]
80
81  # Base parameters (will be overridden for the parameter we're varying)
82  base_params = {
```

```
    "theta": 1.0,
    "phi":   1.0,
    "tau":   1.0,
    "eta":   1.0,
    "sigma": 1.0,
    "zeta":  0.1
}

# x-values for evaluating the GP
x_test = np.linspace(0, 4, 2000)

# Create the figure: 6 rows x 3 columns
plot_gp_for_hyperparams(
    kernel_func=kernel_function,
    x_points=x_test,
    base_params=base_params,
    param_names=param_names,
    param_ranges=param_ranges,
    param_colors=param_colors,
    n_samples_per_setting=3
)
```

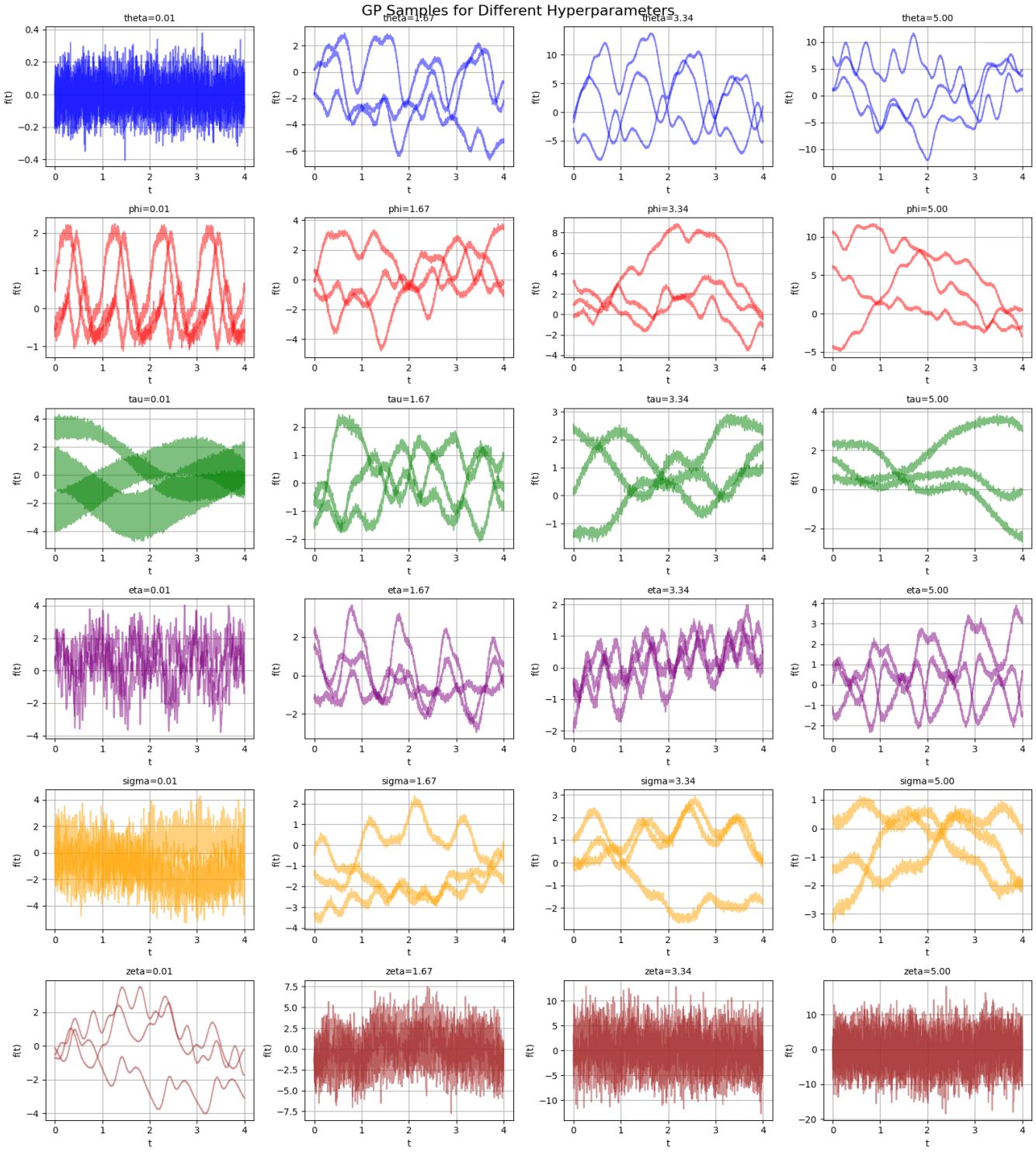


Figure 8: Gaussian Process samples generated using different hyperparameter values for the covariance kernel. Each row corresponds to variations in a specific parameter $(\theta, \phi, \tau, \eta, \sigma, \zeta)$, showing their influence on the sampled functions. The values for each parameter are sampled from the range $\{0.01, 1.67, 3.34, 5.00\}$, demonstrating their effects on function smoothness, periodicity, and variability.

2.5 Estimation of Hyperparameters

The residual time series (Figure 6) reveals a strong seasonal pattern, indicating periodicity in the data. This justifies incorporating a periodic kernel term, with $\tau = 1.0$ reflecting the observed cycle length. The smooth but non-sinusoidal structure suggests $\sigma = 2.0$, ensuring the periodic component captures the variations without excessive rigidity. Additionally, the histogram (Figure 5) shows a broader spread than a standard normal distribution, indicating increased variance, which supports setting $\theta = 3.0$ to scale the function appropriately. The squared-exponential term is necessary to model local correlations, and the presence of short-range fluctuations suggests $\eta = 0.5$ as a reasonable length scale. The residuals' deviation from Gaussian noise (Figure 7) also motivates $\zeta = 0.1$, capturing the stochastic nature of the residual process. Finally, the reduced amplitude of local variability supports setting $\phi = 0.1$.

2.6 (Bonus) Extrapolating CO_2 Concentrations with GP

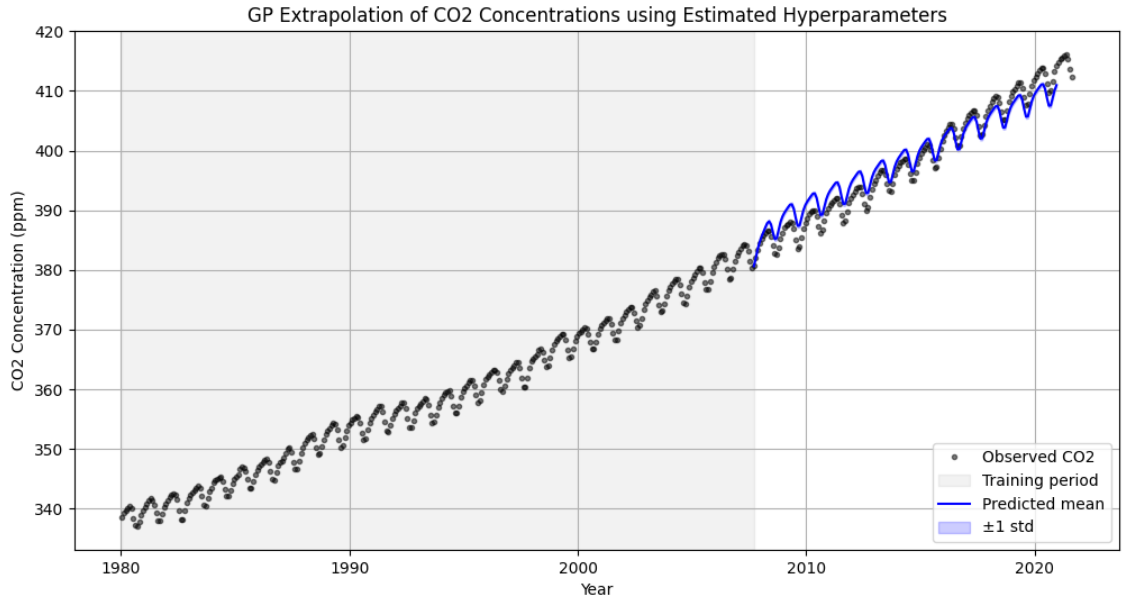


Figure 9: Extrapolated CO_2 concentration levels from September 2007 to December 2020 using Gaussian Process regression. The blue line represents the predicted mean, while the shaded region indicates ± 1 standard deviation, capturing uncertainty. Observed CO_2 levels are plotted as black dots, illustrating alignment between the model and historical data, with clear seasonal fluctuations and an overall upward trend.

The extrapolation of CO_2 concentration levels using the GP regression model suggests a continued increase in CO_2 levels through 2020, aligning with historical trends. The inclusion of a periodic kernel component effectively captures seasonal fluctuations in CO_2 levels, ensuring that the model retains yearly seasonality in its predictions.

The sensitivity of the extrapolation to hyperparameter choices highlights the importance of proper kernel selection and parameter tuning, which could be improved by using a more data driven

method instead of visually inspecting the graphs and using trial and error. Specifically, the periodic component τ maintains the expected yearly seasonality, while other parameters such as η and ζ influence the smoothness and variability of the predictions. We can further observer how our hyperparameters influence the GP by looking at Figure 9.

Although our estimates don't fully align with the given data, they still follow the general trend and are a good estimate for a rough approximation.

Code:

```
# Load and preprocess CO2 data
df = pd.read_csv("co2.txt", delim_whitespace=True, comment='#', header=None,
                 names=["year", "month", "decimal", "average", "trend"])
t, y = df["decimal"].values, df["average"].values
t_min = t.min()
t_normalized = t - t_min
g_obs = y - (a_MAP * t + b_MAP)

def kernel_matrix(s, t, theta=1.0, sigma=1.0, phi=1.0, eta=1.0, tau=1.0, zeta=0.1):
    """Compute covariance matrix using the periodic + squared exponential
    ↪ kernel."""
    periodic_term = np.exp(-2 * (np.sin(np.pi * (s[:, None] - t[None, :]) / tau) **
    ↪ 2) / sigma**2)
    squared_exp_term = phi**2 * np.exp(-0.5 * ((s[:, None] - t[None, :]) / eta) **
    ↪ 2)
    noise_term = zeta**2 * np.eye(len(s)) if np.array_equal(s, t) else
    ↪ np.zeros((len(s), len(t)))

    return (theta**2) * (periodic_term + squared_exp_term) + noise_term

# Define Training Data
cutoff_year = 2007.708
test_start_year = 2007.708
test_end_year = 2020.958

cutoff = cutoff_year - t_min
train_mask = t_normalized <= cutoff

t_train, g_train = t_normalized[train_mask], g_obs[train_mask]

# Define Test Data (Extrapolation)
test_t = np.linspace(test_start_year, test_end_year, int((test_end_year -
↪ test_start_year) * 12) + 1)
test_t_normalized = test_t - t_min  # Normalize test time as well

# Hyperparameters
theta = 3
tau = 1
sigma = 2
phi = 0.1
```

```
eta = 0.5
zeta = 0.1

# Compute Covariance Matrices
K_train = kernel_matrix(t_train, t_train, theta, sigma, phi, eta, tau, zeta)
K_star = kernel_matrix(test_t_normalized, t_train, theta, sigma, phi, eta, tau,
 ↪  zeta)
K_star_star = kernel_matrix(test_t_normalized, test_t_normalized, theta, sigma,
 ↪  phi, eta, tau, zeta)

# GP Inference
L = cholesky(K_train + 1e-6 * np.eye(len(t_train)), lower=True)  # Stabilize
K_inv = cho_solve((L, True), np.eye(len(t_train)))
g_mean = K_star @ (K_inv @ g_train)
g_cov = K_star_star - K_star @ (K_inv @ K_star.T)
g_std = np.sqrt(np.diag(g_cov))

# Compute Extrapolated CO2 Predictions
f_pred = a_MAP * (test_t) + b_MAP + g_mean

# Plot Results
plt.figure(figsize=(12, 6))
plt.plot(t, y, 'k.', alpha=0.5, label='Observed CO2')
plt.axvspan(t_min, cutoff_year, color='gray', alpha=0.1, label='Training period')
plt.plot(test_t, f_pred, 'b-', label='Predicted mean')
plt.fill_between(test_t, f_pred - g_std, f_pred + g_std, color='blue', alpha=0.2,
 ↪  label='±1 std')
plt.xlabel("Year")
plt.ylabel("CO2 Concentration (ppm)")
plt.title("GP Extrapolation of CO2 Concentrations using Estimated Hyperparameters")
plt.legend()
plt.grid(True)
plt.savefig("q2f_gp_extrap.png")
plt.show()
```

2.7 (Bonus) Bayesian Approach to Modelling $f(t)$

The procedure is not fully Bayesian because it uses a Maximum A Posteriori (MAP) estimate for the linear regression parameters (a and b) instead of integrating over their posterior distribution, thereby underestimating the overall uncertainty in predictions. A fully Bayesian approach would propagate the uncertainty of these parameters into the Gaussian Process (GP) model. This could be achieved by directly modelling $f(t)$ with a combined kernel that incorporates both the linear trend and the GP components, eliminating the need for separate steps. Alternatively, treating the linear regression parameters as random variables and integrating their posterior distribution using methods like Markov Chain Monte Carlo (MCMC) or Variational Inference would provide uncertainty bounds that account for variability in both the trend and residual components, offering a more rigorous Bayesian framework.

3 Mean-Field Learning for a Binary Latent Factor Model

Let $X = \{x^{(n)}\}_{n=1}^{N}$ be a dataset where each $x^{(n)} \in \mathbb{R}^D$ is assumed to be generated by a set of latent binary variables $s^{(n)} = (s_1^{(n)}, ..., s_K^{(n)})$, with $s_i^{(n)} \in \{0, 1\}$. The generative process follows:

Each binary latent variable follows an independent Bernoulli distribution:

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

where π_i represents the prior probability of $s_i = 1$.

Conditioned on the latent variables, the observed data follows a Gaussian distribution:

$$p(x|s, \mu, \sigma^2) = \mathcal{N}\left(\sum_i s_i \mu_i, \sigma^2 I\right).$$

This model captures the intuition that different latent components contribute additively to the mean of x, with Gaussian noise.

The central challenge in inference is computing the true posterior $p(s|x)$, which follows from Bayes' rule:

$$p(s|x) = \frac{p(x|s)p(s)}{p(x)}.$$

While evaluating the numerator is straightforward, normalizing this posterior requires computing the marginal likelihood $p(x)$, obtained by summing over all possible latent states:

$$p(x) = \sum_s p(x|s)p(s).$$

Since each s_i is binary, there are 2^K possible configurations of s. This results in an exponentially large sum over 2^K terms, making direct computation intractable for even moderate values of K.

Expanding the likelihood:

$$p(x|s) = \frac{1}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{1}{2\sigma^2}\|x - \sum_i s_i \mu_i\|^2\right).$$

Since the mean $\sum_i s_i \mu_i$ involves summing binary variables, the exponent contains nonlinear interactions between s_i terms, preventing factorization. Consequently, the sum:

$$p(x) = \sum_s \frac{1}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{1}{2\sigma^2}\|x - \sum_i s_i \mu_i\|^2\right) p(s)$$

remains an intractable combinatorial sum over 2^K terms.

Because of this exponential growth, direct computation is infeasible, necessitating approximate inference methods.

3.1 Implementing Mean-Field Variational Inference

Since exact inference is intractable, we approximate the posterior distribution $p(\mathbf{s}|\mathbf{x}, \boldsymbol{\theta})$ using a fully factorized variational distribution:

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

where $\lambda_{in}^{(n)}$ is a variational parameter representing the approximate posterior probability $P(s_{in}^{(n)} = 1|\mathbf{x}^{(n)})$. The optimal $\lambda_{in}^{(n)}$ maximizes the ELBO:

$$\mathcal{F}_n(q, \boldsymbol{\theta}) = \mathbb{E}_q[\log p(\mathbf{s}, \mathbf{x}|\boldsymbol{\theta})] - \mathbb{E}_q[\log q(\mathbf{s})].$$

The joint probability factorizes as:

$$\log p(\mathbf{s}, \mathbf{x}|\boldsymbol{\theta}) = \log p(\mathbf{x}|\mathbf{s}, \boldsymbol{\theta}) + \log p(\mathbf{s}|\boldsymbol{\pi}),$$

where the prior distribution is given by:

$$p(\mathbf{s}|\boldsymbol{\pi}) = \prod_{n=1}^{N} p(\mathbf{s}^{(n)}|\boldsymbol{\pi}),$$

with each latent vector $\mathbf{s}^{(n)} = (s_1^{(n)}, \ldots, s_K^{(n)})$ following a factorized Bernoulli prior:

$$p(\mathbf{s}^{(n)}|\boldsymbol{\pi}) = \prod_{i=1}^{K} (\pi_i)^{s_i^{(n)}} (1 - \pi_i)^{(1-s_i^{(n)})}.$$

For a single observation $\mathbf{x}^{(n)}$, the likelihood is given by:

$$p(\mathbf{x}^{(n)}|\mathbf{s}^{(n)}, \boldsymbol{\theta}) = \mathcal{N}\left(\mathbf{x}^{(n)} \,\middle|\, \sum_{i=1}^{K} s_i^{(n)} \boldsymbol{\mu}_i, \sigma^2 \mathbf{I}\right).$$

Since the data points are i.i.d., the full likelihood factorizes as:

$$p(\mathbf{X}|\mathbf{S},\boldsymbol{\theta}) = \prod_{n=1}^{N} p(\mathbf{x}^{(n)}|\mathbf{s}^{(n)},\boldsymbol{\theta}).$$

Taking the log-likelihood:

$$\log p(\mathbf{X}|\mathbf{S},\boldsymbol{\theta}) = \sum_{n=1}^{N} \log p(\mathbf{x}^{(n)}|\mathbf{s}^{(n)},\boldsymbol{\theta}).$$

Substituting the Gaussian log-density:

$$\log p(\mathbf{x}^{(n)}|\mathbf{s}^{(n)},\boldsymbol{\theta}) = -\frac{D}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\|\mathbf{x}^{(n)} - \sum_{i=1}^{K} s_i^{(n)}\boldsymbol{\mu}_i\|^2.$$

Taking expectation under $q(\mathbf{S})$, the first term is constant, while the expectation of the quadratic term expands as:

$$\mathbb{E}_q\left[\|\mathbf{x}^{(n)} - \sum_{i=1}^{K} s_i^{(n)}\boldsymbol{\mu}_i\|^2\right] = \|\mathbf{x}^{(n)}\|^2 - 2\mathbf{x}^{(n)T}\sum_{i=1}^{K}\lambda_i^{(n)}\boldsymbol{\mu}_i + \sum_{i=1}^{K}\lambda_i^{(n)}\|\boldsymbol{\mu}_i\|^2 + \sum_{i\neq j}\lambda_i^{(n)}\lambda_j^{(n)}\boldsymbol{\mu}_i^T\boldsymbol{\mu}_j.$$

Thus, summing over all N, we obtain:

$$\mathbb{E}_q[\log p(\mathbf{X}|\mathbf{S},\boldsymbol{\theta})] = -\frac{ND}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{n=1}^{N}\left(\|\mathbf{x}^{(n)}\|^2 - 2\mathbf{x}^{(n)T}\sum_{i=1}^{K}\lambda_i^{(n)}\boldsymbol{\mu}_i + \sum_{i=1}^{K}\lambda_i^{(n)}\|\boldsymbol{\mu}_i\|^2 + \sum_{i\neq j}\lambda_i^{(n)}\lambda_j^{(n)}\boldsymbol{\mu}_i^T\boldsymbol{\mu}_j\right).$$

To optimize $\mathcal{F}_n(q,\boldsymbol{\theta})$, we differentiate with respect to $\lambda_i^{(n)}$:

$$\lambda_i^{(n)} = \sigma\left(\log\frac{\pi_i}{1-\pi_i} + \frac{1}{\sigma^2}\boldsymbol{\mu}_i^\top\left(\mathbf{x}^{(n)} - \sum_{j\neq i}\lambda_j^{(n)}\boldsymbol{\mu}_j\right) - \frac{1}{2\sigma^2}\|\boldsymbol{\mu}_i\|^2\right),$$

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

Code:

```python
def compute_entropy(lambda_matrix: np.ndarray) -> float:
    lam_clipped = np.clip(lambda_matrix, 1e-12, 1 - 1e-12)
    return -np.sum(
```

```
        lam_clipped * np.log(lam_clipped) + (1 - lam_clipped) * np.log(1 -
        ↪ lam_clipped)
    )

def compute_log_p_x_given_s(data: np.ndarray, model: dict, lambda_matrix:
↪ np.ndarray) -> float:
    N, D = data.shape
    mu = model['mu']   # shape (D, K)
    sigma = model['sigma']
    precision = 1.0 / (sigma**2)

    # Expand (X - sum_k lambda_k mu_k)^2
    mu_product = lambda_matrix @ mu.T
    x_sq_sum = np.sum(data * data)
    cross_term = -2.0 * np.sum(data * mu_product)
    mu_sq_sum = np.sum(mu_product * mu_product)

    log_p_x_s = -0.5 * precision * (x_sq_sum + mu_sq_sum + cross_term)
    log_constant = -0.5 * N * D * np.log(2.0 * np.pi * (sigma**2))
    return log_constant + log_p_x_s

def compute_log_p_s(model: dict, lambda_matrix: np.ndarray) -> float:
    pi = np.clip(model['pi'], 1e-12, 1 - 1e-12)  # shape (1, K)
    lam_clipped = np.clip(lambda_matrix, 1e-12, 1 - 1e-12)

    log_pi = np.log(pi)
    log_one_minus_pi = np.log(1 - pi)
    log_ps = lam_clipped * log_pi + (1 - lam_clipped) * log_one_minus_pi
    return np.sum(log_ps)

def compute_free_energy(data: np.ndarray, model: dict, lambda_matrix: np.ndarray)
↪ -> float:
    term_x_given_s = compute_log_p_x_given_s(data, model, lambda_matrix)
    term_s = compute_log_p_s(model, lambda_matrix)
    entropy = compute_entropy(lambda_matrix)
    N = data.shape[0]
    return (term_x_given_s + term_s + entropy) / N

def partial_variational_update(data, model, lambda_matrix, factor_index):
    mu = model['mu']  # shape (D, K)
    sigma = model['sigma']
    pi_f = model['pi'][0, factor_index]
    precision = 1.0 / (sigma**2)

    # Exclude factor f
    lambda_excluding = np.delete(lambda_matrix, factor_index, axis=1)
    mu_excluding = np.delete(mu, factor_index, axis=1)
    mu_f = mu[:, factor_index]  # shape (D,)

    # Reconstruction from other factors
```

```
    recon_excl = lambda_excluding @ mu_excluding.T  # shape (N, D)
    recon = data - recon_excl                       # shape (N, D)

    # Log-odds update
    log_p_x = precision * np.sum(recon * mu_f, axis=1) \
              - 0.5 * precision * np.sum(mu_f**2)

    # Prior log-odds
    log_p_s = np.log(pi_f / (1.0 - pi_f + 1e-12))
    log_odds = log_p_x + log_p_s

    # Sigmoid update
    updated_lambda = 1.0 / (1.0 + np.exp(np.clip(-log_odds, -100, 100)))
    # ALSO CLIP final lambda to avoid 0/1
    updated_lambda = np.clip(updated_lambda, 1e-10, 1 - 1e-10)
    return updated_lambda


def mean_field(
    data: np.ndarray, model: dict, lambda_matrix: np.ndarray,
    max_steps: int, convergence_threshold: float
) -> np.ndarray:
    free_energy_history = []

    old_fe = compute_free_energy(data, model, lambda_matrix)
    free_energy_history.append(old_fe)

    K = model['mu'].shape[1]

    for _ in range(max_steps):
        for factor in range(K):
            lambda_matrix[:, factor] = partial_variational_update(
                data, model, lambda_matrix, factor
            )

        new_fe = compute_free_energy(data, model, lambda_matrix)
        free_energy_history.append(new_fe)

        if (new_fe - old_fe) <= convergence_threshold:
            break
        old_fe = new_fe

    return free_energy_history
```

3.2 Comparing Mean-Field Learning to Linear Regression

The M-step solution relates to linear regression because it estimates the parameters μ and σ^2 by minimizing the residuals, similar to ordinary least squares (OLS). The expected values $\mathbb{E}[s]$

act as the design matrix, and the observed data X serves as the response variable, with $\mathbb{E}[ss^T]$ contributing regularization. This approach effectively reduces the problem to solving a weighted least squares regression with closed-form updates for the parameters.

3.3 Computational Complexity Analysis of the M-Step

The M-step function `m_step(X, ES, ESS)` performs parameter updates in the mean-field variational inference setting. To analyse its computational complexity, we consider the following inputs and outputs:

- **Inputs:**
  - X: $N \times D$ matrix (observed data)
  - ES: $N \times K$ matrix (expected values of hidden binary latent variables)
  - ESS: $K \times K$ matrix (sum over data points of second-order expectations $\mathbb{E}[ss^T]$)
- **Outputs:**
  - μ: $D \times K$ matrix of means
  - σ: scalar standard deviation
  - π: $1 \times K$ vector of prior probabilities of latent factors

To analyze the computational complexity, we decompose the key operations.

The parameter update for μ is given by:

$$\mu = (ESS^{-1} \cdot ES^T \cdot X)^T.$$

Breaking it down:

- Computing ESS^{-1} (inverse of a $K \times K$ matrix): $\mathcal{O}(K^3)$
- Matrix multiplication $ESS^{-1} \cdot ES^T$ ($K \times N$): $\mathcal{O}(K^2 N)$
- Matrix multiplication with X ($N \times D$): $\mathcal{O}(KND)$

Thus, the total computational complexity for computing μ is:

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

The standard deviation update is given by:

$$\sigma^2 = \frac{1}{ND}\Big[\text{trace}(X^T X) + \text{trace}(\mu^T \mu ESS) - 2\text{trace}(ES^T X \mu)\Big].$$

Breaking it down:

- Computing trace($X^T X$): $\mathcal{O}(D^2 N)$
- Computing trace($\mu^T \mu ESS$):
  - $\mu^T \mu$ ($K \times K$): $\mathcal{O}(DK^2)$
  - Multiplication with ESS ($K \times K$): $\mathcal{O}(K^3)$
- Computing -2trace($ES^T X \mu$):
  - $ES^T X$ ($K \times D$): $\mathcal{O}(KND)$
  - Multiplication with μ ($K \times D$): $\mathcal{O}(K^2 D)$

Thus, the total computational complexity for computing σ is:

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

The prior probabilities update is given by:

$$\pi = \frac{1}{N} \sum_{n=1}^{N} ES_n.$$

Since this involves computing the mean of ES across N, the computational complexity is:

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

Summing all terms, we obtain:

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

This simplifies to:

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

- **For small** K: The dominant term is $\mathcal{O}(D^2 N)$, meaning the M-step is primarily data-driven.
- **For small** D: The dominant term is $\mathcal{O}(2K^3)$, suggesting that computational cost is determined by latent variable interactions.
- **For large** N: The dominant term is $\mathcal{O}(D^2 N + 2KND)$, meaning the M-step scales quadratically with D and linearly with N.

3.4 Exploratory Data Analysis on Generated Images

From the given images, we can observe that there are 8 distinct features, and they appear to be independent.

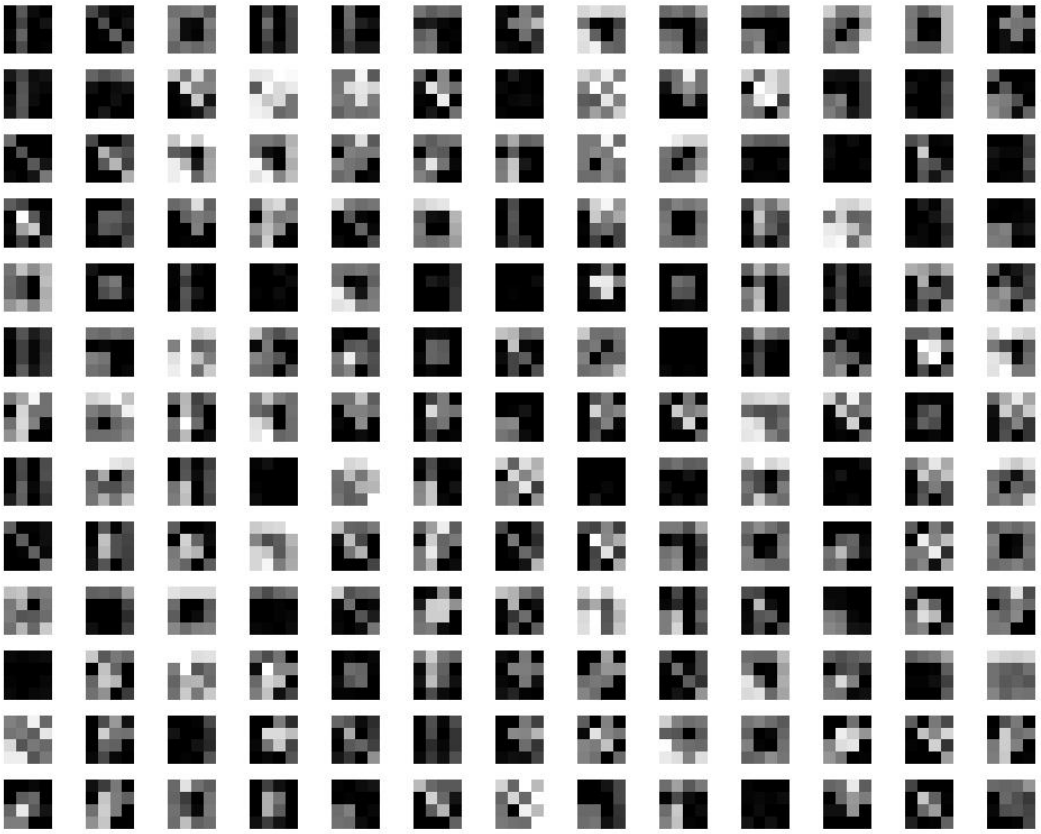


Figure 10: Given images for binary latent factor question

Examining the dataset of 100 grayscale 4×4 images, we observe that they are generated by the superposition of a small number of independent binary features, combined with Gaussian noise. By analyzing the structure of the images, we infer that there are $K = 8$ distinct features, each corresponding to a fundamental pattern such as edges, crosses, or localized pixel clusters. Mathematically, we assume that each observed image $x \in \mathbb{R}^D$ (where $D = 16$ for a 4×4 image) is generated as a linear combination of these features via a latent variable model:

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

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

where $s \in \{0,1\}^K$ is a binary latent vector indicating the presence of each feature, $\mu_i \in \mathbb{R}^D$ represents the feature basis, and σ^2 is the noise variance.

Given this generative process, Factor Analysis (FA) is not a suitable model because it assumes that the latent variables $z \sim \mathcal{N}(0, I)$ are continuous and Gaussian-distributed, whereas our model

relies on binary latent factors. Since FA models the data as $x = \Lambda z + \epsilon$ with Gaussian z and noise $\epsilon \sim \mathcal{N}(0, \Psi)$, it will struggle to correctly capture the discrete nature and independence of the binary features, instead approximating them with continuous factors. Moreover, FA assumes that the latent variables are jointly Gaussian, while our underlying sources are independent and sparse, making FA an inadequate choice for recovering the true features.

Mixture of Gaussians (MoG) is also unsuitable because it assumes that each image is drawn from a single Gaussian cluster. In our case, however, each image is a mixture of multiple independent binary features, not a sample from one specific Gaussian distribution. MoG fits a probability density function of the form:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

where each component k represents a Gaussian cluster with mean μ_k and covariance Σ_k. This approach does not align with our data, where multiple features contribute to a single image, and the superposition of independent components cannot be well represented by a hard clustering approach.

Independent Component Analysis (ICA) is the most appropriate method because it assumes that the observed data is a linear mixture of independent sources:

$$x = As$$

where A is a mixing matrix and s are the statistically independent components (features). Unlike FA, ICA does not impose a Gaussian prior on the latent variables; instead, it seeks maximally statistically independent components, making it well-suited for recovering non-Gaussian binary sources like those in our dataset. Additionally, ICA does not assume that each data point is drawn from a single cluster, making it more flexible than MoG for modeling overlapping feature compositions. Given that our latent variables are binary and independent, ICA is the best choice for discovering the underlying feature structure.

3.5 Implementing the Full EM Algorithm

```python
def m_step(data: np.ndarray, lambda_matrix: np.ndarray):
    N, D = data.shape
    K = lambda_matrix.shape[1]

    ES = lambda_matrix
    # E[s s^T], with diagonal corrected:
    ESS = ES.T @ ES
    np.fill_diagonal(ESS, np.sum(ES, axis=0))  # crucial for Bernoulli factors

    inv_ESS = np.linalg.inv(ESS)
    ES_T_X = ES.T @ data  # shape (K, D)
```

```
    mu = (inv_ESS @ ES_T_X).T  # shape (D, K)

    # sigma
    term1 = np.trace(data.T @ data)
    term2 = np.trace((mu.T @ mu) @ ESS)
    term3 = 2.0 * np.trace((ES.T @ data) @ mu)
    sigma_sq = (term1 + term2 - term3) / (N * D)
    sigma = np.sqrt(np.maximum(sigma_sq, 1e-12))  # safeguard nonnegative

    pi = np.mean(ES, axis=0, keepdims=True)
    pi = np.clip(pi, 1e-6, 1.0 - 1e-6)  # avoid pi=0 or 1

    return mu, sigma, pi


def learn_bin_factors(
    data: np.ndarray,
    model: dict,
    lambda_matrix: np.ndarray,
    max_em_iterations: int = 100,
    max_e_iterations: int = 100,
    e_convergence_threshold: float = 1e-9
):
    free_energy_history = [compute_free_energy(data, model, lambda_matrix)]

    for iteration in range(max_em_iterations):
        old_fe = free_energy_history[-1]
        old_lambda = lambda_matrix.copy()

        # E-step (coordinate-ascent, measure FE only after entire sweep)
        fe_hist = mean_field(
            data, model, lambda_matrix,
            max_steps=max_e_iterations,
            convergence_threshold=e_convergence_threshold
        )
        # Extend all but the first entry (to avoid repeating FE)
        free_energy_history.extend(fe_hist[1:])

        # M-step
        mu, sigma, pi = m_step(data, lambda_matrix)
        model['mu'] = mu
        model['sigma'] = sigma
        model['pi'] = pi

        # Compute new FE
        new_fe = compute_free_energy(data, model, lambda_matrix)
        free_energy_history.append(new_fe)

        # Check convergence
        if abs(new_fe - old_fe) < 1e-9:
```

```
            print(f"EM Converged at iteration {iteration + 1}")
            break

    return lambda_matrix, model, free_energy_history
```

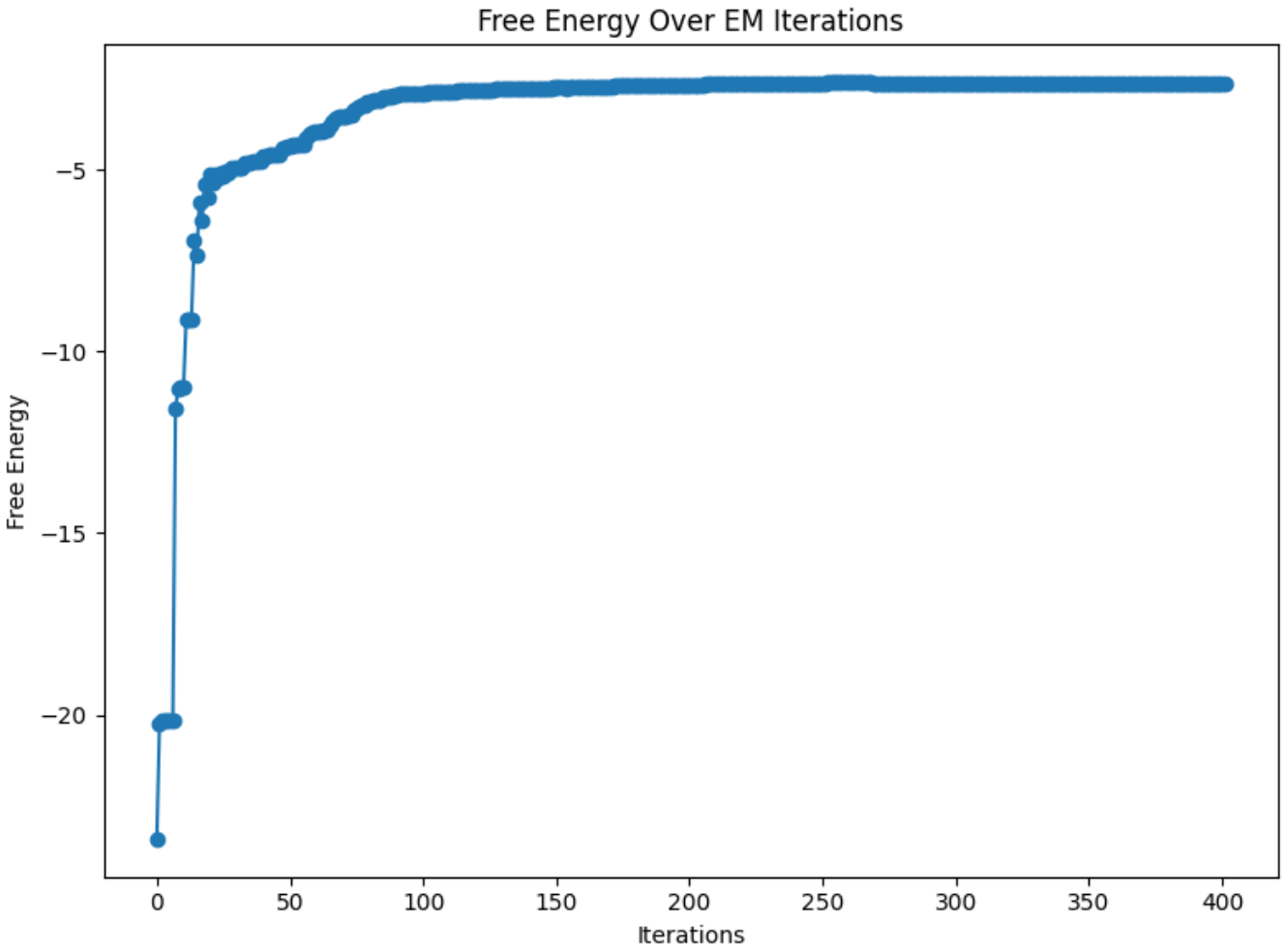


Figure 11: The Free Energy (FE) plot over iterations for the Mean-Field Variational EM algorithm. The plot shows a monotonic increase in FE, confirming proper optimization and convergence of the algorithm.

3.6 Evaluating Learned Features

The Mean-Field Variational EM Algorithm was applied to learn a binary latent factor model on data generated by `genimages.m`. The model was configured with eight latent factors ($K = 8$), and the EM algorithm was run for 100 iterations with a convergence criterion of 10^{-6} for the E-step. The results indicate that the algorithm successfully learned a set of latent features, which were visualized as 4×4 greyscale images.

To evaluate the performance of the model, the Free Energy (FE) plot was analysed. The plot demonstrates a monotonic increase in free energy, confirming that the optimization is proceeding

correctly. The learned features, represented by the parameter μ, appear to reflect underlying structures in the dataset, but some noise and artefacts remain.

Several improvements can be introduced to enhance the algorithm's effectiveness. Better initialization could significantly reduce the effect of poor starting conditions; using Principal Component Analysis (PCA) to initialize μ rather than random values would likely yield more stable and meaningful features. Additionally, the features can be post-processed to improve clarity by applying a simple thresholding function, to make the learned μ values binary, removing minor noise and enhancing interpretability.

Another potential enhancement involves running the algorithm multiple times and averaging the learned features across different runs. This approach mitigates the impact of local optima, providing a more consistent set of factors. Furthermore, an adaptive learning rate in the E-step could help improve convergence speed by dynamically adjusting updates to the variational parameters λ.

Overall, while the model successfully extracts meaningful patterns, these refinements could further enhance the quality of the learned features, making them more robust and interpretable.

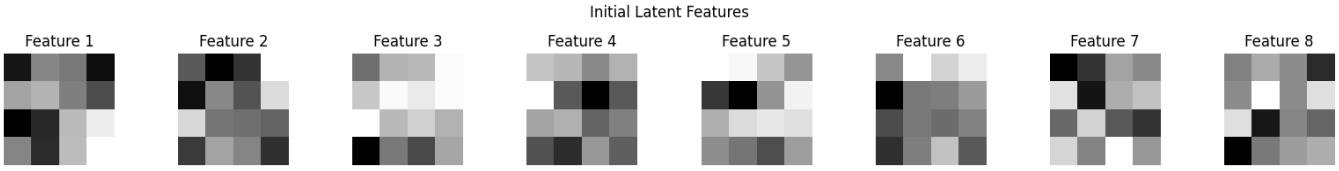


Figure 12: Initial latent features (8 factors, $K = 8$) visualized as 4×4 grayscale images before running the algorithm. These represent randomly initialized patterns.

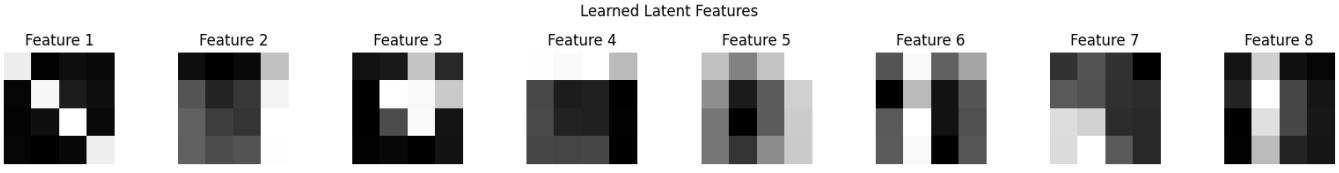


Figure 13: Learned latent features (μ) visualized as 4×4 grayscale images after applying the EM algorithm. These features reflect underlying patterns in the dataset but may still include noise and artifacts.

3.7 Convergence Analysis for Variational Approximation

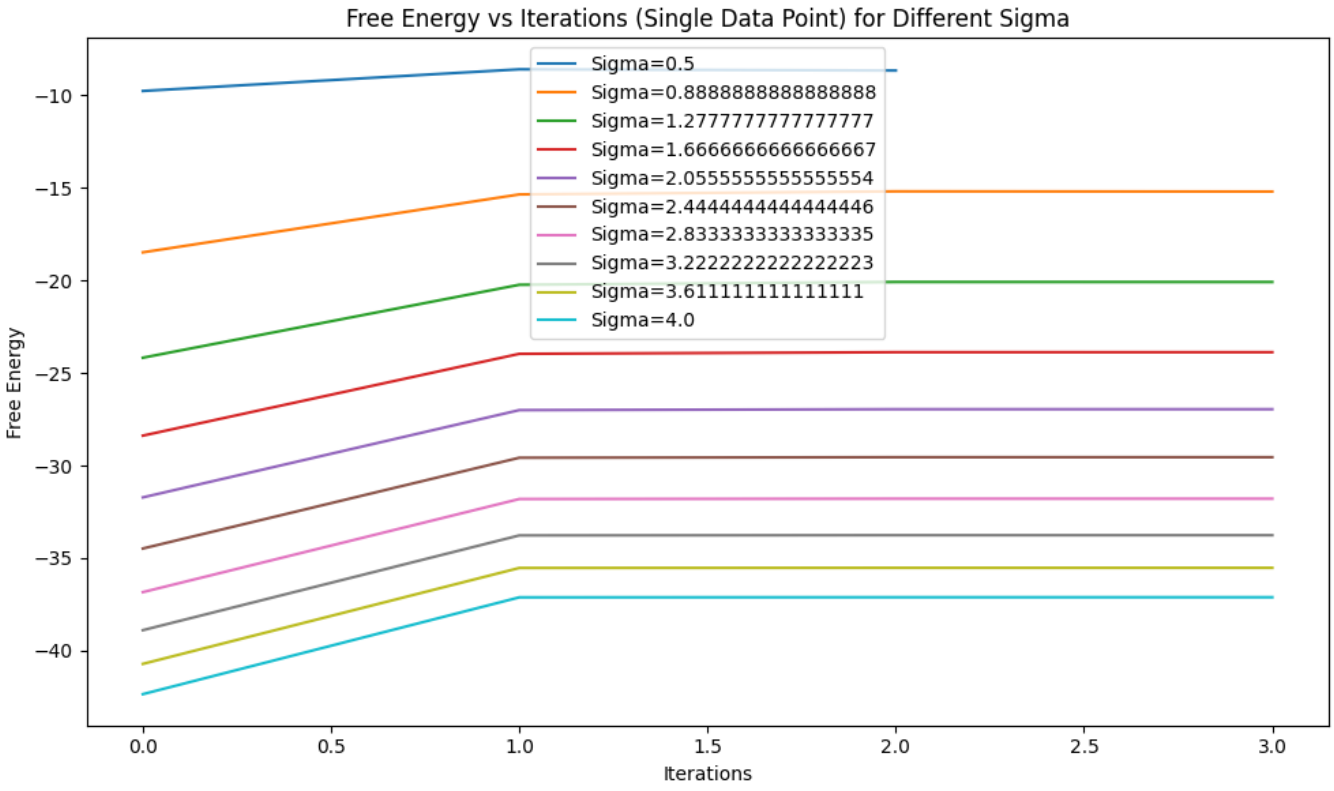


Figure 14: Free Energy convergence for a single data point across EM iterations for different values of σ. Higher σ values result in lower Free Energy, showing the impact of noise variance on optimization.

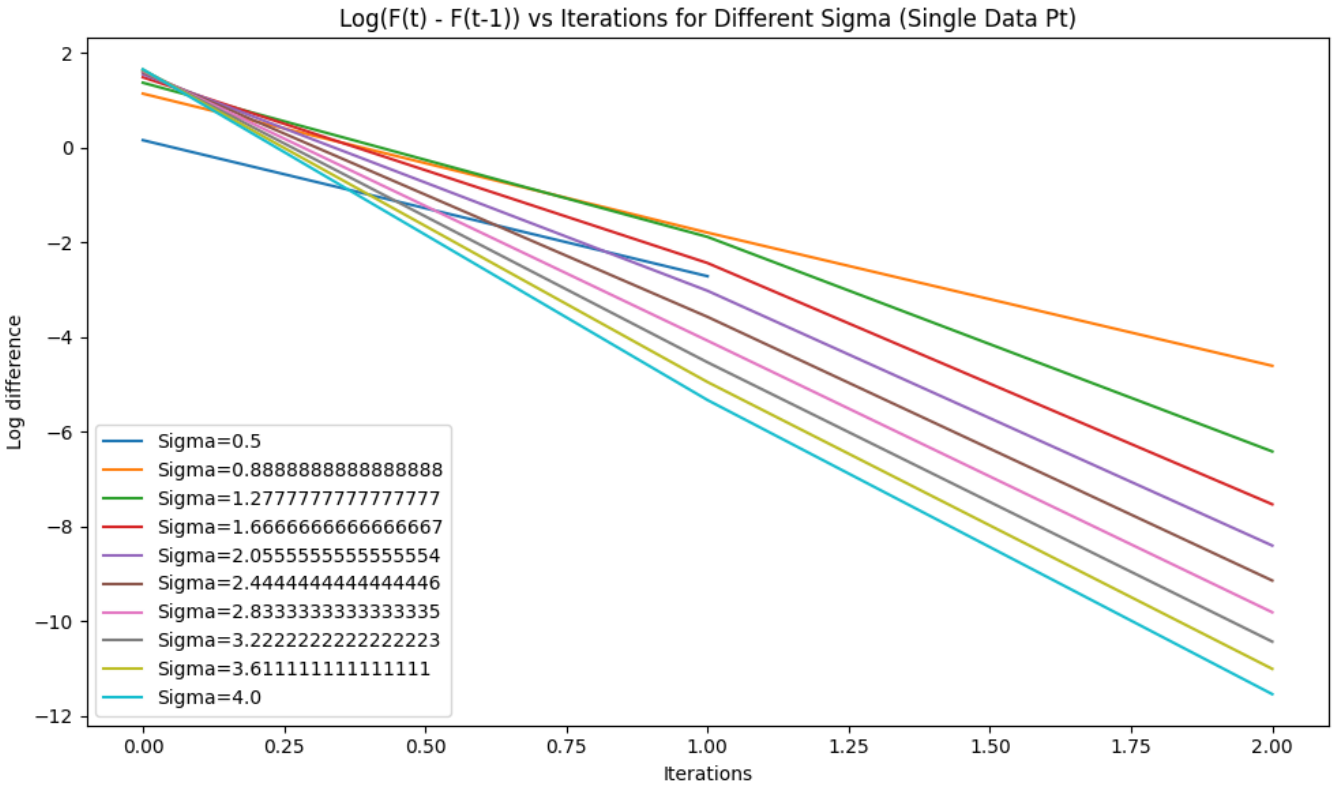


Figure 15: Log difference in Free Energy ($\log(F_t - F_{t-1})$) as a function of EM iterations for varying σ. This plot illustrates the convergence rate under different noise variance conditions.

The Free Energy (FE) plots demonstrate that optimization through EM converges consistently across iterations, regardless of the noise variance σ. Larger values of σ reduce the maximum achievable FE, reflecting the trade-off between model complexity and noise. The log difference in FE highlights a rapid convergence for small σ values, suggesting that lower noise leads to quicker updates of the variational parameters, while higher σ slows the process due to broader uncertainty. These findings confirm the robustness of the variational EM algorithm under varying noise conditions.

Code:

```
def main():
    # Create output directory
    out_dir = "question_3_output"
    os.makedirs(out_dir, exist_ok=True)

    # Initialize model parameters
    initial_sigma = 1.0
    initial_pi = np.full((1, num_factors), 0.5)
    initial_mu = np.random.randn(num_features, num_factors)

    model = {
        'mu': initial_mu,         # shape (D,K)
        'sigma': initial_sigma,   # float
```

```
        'pi': initial_pi          # shape (1,K)
    }

    # Initialize lambda
    initial_lambda = np.random.rand(num_samples, num_factors)

    # Visualize initial features
    visualize_latent_features(model['mu'], "Initial Latent Features",
     ↪  save_path=os.path.join(out_dir, "3f1.png"))

    # Run EM
    lambda_matrix, model, free_energy_hist = learn_bin_factors(
        data=data,
        model=model,
        lambda_matrix=initial_lambda,
        max_em_iterations=100,
        max_e_iterations=100,
        e_convergence_threshold=1e-6
    )

    # Visualize learned features
    visualize_latent_features(model['mu'], "Learned Latent Features",
     ↪  save_path=os.path.join(out_dir, "3f2.png"))
    # Plot free energy
    plot_free_energy(free_energy_hist, "Free Energy Over EM Iterations",
                     save_path=os.path.join(out_dir, "3e1.png"))

    # Analyze effect of different sigma values using a single data point
    single_data = data[:1, :]  # shape (1,D)
    test_sigmas = np.linspace(0.5, 4.0, 10)
    fe_sigma_list = []

    # We'll keep a local copy of lambda for N=1
    single_lambda = lambda_matrix[:1, :].copy()

    for s_val in test_sigmas:
        # Temporarily override model's sigma
        model_copy = {
            'mu': model['mu'].copy(),
            'sigma': s_val,
            'pi': model['pi'].copy()
        }
        # We'll re-run the E-step for the single data point
        lam_copy = single_lambda.copy()
        fe_hist = mean_field(
            single_data, model_copy, lam_copy,
            max_steps=100, convergence_threshold=1e-9
        )
        fe_sigma_list.append(fe_hist)

```

```
    # Plot free energy for different sigma
    plt.figure(figsize=(10, 6))
    for idx, fe_hist in enumerate(fe_sigma_list):
        plt.plot(fe_hist, label=f"Sigma={test_sigmas[idx]}")
    plt.title("Free Energy vs Iterations (Single Data Point) for Different Sigma")
    plt.xlabel("Iterations")
    plt.ylabel("Free Energy")
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(out_dir, "3g1.png"), bbox_inches="tight")
    plt.show()

    # Plot log-differences
    plt.figure(figsize=(10, 6))
    for idx, fe_hist in enumerate(fe_sigma_list):
        if len(fe_hist) > 1:
            diffs = np.diff(fe_hist)
            log_diffs = np.log(np.abs(diffs) + 1e-12)
            plt.plot(log_diffs, label=f"Sigma={test_sigmas[idx]}")
    plt.title("Log(F(t) - F(t-1)) vs Iterations for Different Sigma (Single Data
    ↪ Pt)")
    plt.xlabel("Iterations")
    plt.ylabel("Log difference")
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(out_dir, "3g2.png"), bbox_inches="tight")
    plt.show()

    print(f"Results saved to directory: {out_dir}")

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

4 (Bonus) Variational Bayes for Binary Factors

4.1 Bayesian Hyperparameter Optimization for K

4.2 Implementing and Evaluating the Model Selection Algorithm

5 Expectation Propagation (EP) for Binary Factor Model

5.1 Log-Joint Probability and Relation to Boltzmann Machines

We want to write the log-joint as

$$\log p(s,x) \;=\; \log p(s) \;+\; \log p(x \mid s).$$

First, we expand the prior:

$$\log p(s) \;=\; \sum_{i=1}^{K}\Big[s_i \log \pi_i \;+\; (1-s_i)\log\big(1-\pi_i\big)\Big].$$

For the likelihood term $\log p(x \mid s)$, ignoring additive constants in x, we have

$$\log p(x \mid s) \;\propto\; -\frac{1}{2\sigma^2}\Big\|x - \sum_{i=1}^{K} s_i\,\mu_i\Big\|^2.$$

Expanding the quadratic:

$$\Big\|x - \sum_i s_i\mu_i\Big\|^2 = x^\top x \;-\; 2\,x^\top \sum_i s_i\mu_i \;+\; \sum_{i,j} s_i\,s_j\,\mu_i^\top\mu_j.$$

Hence,

$$-\frac{1}{2\sigma^2}\Big\|x - \sum_i s_i\mu_i\Big\|^2 \;=\; -\frac{1}{2\sigma^2}\,x^\top x \;+\; \frac{1}{\sigma^2}\sum_i s_i\,\mu_i^\top x \;-\; \frac{1}{2\sigma^2}\sum_{i,j} s_i\,s_j\,\mu_i^\top\mu_j.$$

The term $-\frac{1}{2\sigma^2}\,x^\top x$ is a constant wrt s.

Putting these pieces together, up to constants in x,

$$\log p(s,x) \;=\; \sum_{i=1}^{K}\Big[s_i \log \pi_i \;+\; (1-s_i)\log\big(1-\pi_i\big)\Big] \;+\; \frac{1}{\sigma^2}\sum_{i=1}^{K} s_i\,\mu_i^\top x \;-\; \frac{1}{2\sigma^2}\sum_{i,j} s_i\,s_j\,\mu_i^\top\mu_j \;+\; \text{const.}$$

Since $\sum_{i,j}$ includes diagonal ($i = j$) and off-diagonal ($i \neq j$), we note $s_i^2 = s_i$ for binary s_i. This splits the double sum into

$$\sum_{i,j} s_i\,s_j\,\mu_i^\top\mu_j \;=\; \sum_i s_i\,\mu_i^\top\mu_i \;+\; \sum_{i\neq j} s_i\,s_j\,\mu_i^\top\mu_j.$$

Hence,

$$\log p(s,x) \;=\; \sum_{i=1}^{K}\Big[s_i\,\log\pi_i \;+\; (1-s_i)\log\big(1-\pi_i\big) \;+\; \frac{1}{\sigma^2}\,s_i\,\mu_i^\top x \;-\; \frac{1}{2\sigma^2}\,s_i\,\mu_i^\top\mu_i\Big] \;-\; \frac{1}{2\sigma^2}\sum_{i\neq j} s_i\,s_j\,\mu_i^\top\mu_j \;+\; C.$$

We can group the terms that depend only on s_i into a "site factor" $\log f_i(s_i)$, and the terms that depend on the product $s_i s_j$ into a "pairwise factor" $\log g_{ij}(s_i, s_j)$. Namely,

$$\log f_i(s_i) \;=\; s_i \log \pi_i \;+\; (1 - s_i) \log\Big(1 - \pi_i\Big) \;+\; \frac{1}{\sigma^2}\, s_i\, \mu_i^\top x \;-\; \frac{1}{2\sigma^2}\, s_i\, \mu_i^\top \mu_i,$$

and for $i \neq j$,

$$\log g_{ij}(s_i, s_j) \;=\; -\frac{1}{2\sigma^2}\, \mu_i^\top \mu_j\; s_i\, s_j.$$

Thus the log-joint factorizes as

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

A binary Boltzmann Machine has

$$p(s) \;=\; \frac{1}{Z} \exp\Big[\sum_{i<j} W_{ij}\, s_i\, s_j \;+\; \sum_i b_i\, s_i\Big].$$

Comparing, we identify

$$W_{ij} \;=\; -\,\frac{\mu_i^\top \mu_j}{\sigma^2}, \quad b_i \;=\; \log \frac{\pi_i}{1 - \pi_i} \;+\; \frac{\mu_i^\top x}{\sigma^2} \;-\; \frac{1}{2\sigma^2}\, \mu_i^\top \mu_i,$$

plus additive constants collected in $-\log Z$. Thus, $\log p(s \mid x)$ has precisely the form of a (pairwise) Boltzmann Machine in $\{s_i\}$.

$$\log p(s, x) \;=\; \sum_{i=1}^{K} \log f_i\Big(s_i\Big) \;+\; \sum_{i<j} \log g_{ij}\Big(s_i, s_j\Big) \;+\; \text{const}, \quad \text{where } f_i(\cdot),\, g_{ij}(\cdot,\cdot) \text{ as above.}$$

This completes the derivation.

5.2 Message Passing for EP: Derivation and Approximation

We consider a binary latent factor model where the hidden variables $s = (s_1, \ldots, s_K)$ take values in $\{0, 1\}$. The joint distribution is given by:

$$p(s) = \frac{1}{Z} \exp\left(\sum_i b_i s_i + \sum_{i<j} W_{ij} s_i s_j \right),$$

where:

- b_i is the singleton potential for s_i, which acts as a bias term,
- W_{ij} is the pairwise interaction weight between s_i and s_j,

- Z is the partition function ensuring normalization.

Expectation Propagation (EP) approximates the posterior distribution by factorizing it into singleton and pairwise site approximations:

$$q(s) = \prod_i \tilde{f}_i(s_i) \prod_{ij} \tilde{g}_{ij}(s_i, s_j),$$

where:

- $\tilde{f}_i(s_i)$ is the singleton site approximation, capturing the marginal effect on s_i,
- $\tilde{g}_{ij}(s_i, s_j)$ is the pairwise site approximation, modeling dependencies between s_i and s_j.

Each of these approximations is in the exponential family:

$$\tilde{f}_i(s_i) = \exp(\theta_{ii} s_i),$$

$$\tilde{g}_{ij}(s_i, s_j) = \exp(\theta_{ji} s_i + \theta_{ij} s_j).$$

Here, θ_{ii}, θ_{ji}, and θ_{ij} are natural parameters that will be updated iteratively using EP.

The true singleton factor is

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

Since $s_i \in \{0, 1\}$, this represents a Bernoulli distribution with log-odds parameter b_i. We approximate it using an EP site of the same form:

$$\tilde{f}_i(s_i) = \exp(\theta_{ii} s_i).$$

To determine θ_{ii}, we minimize the KL divergence:

$$\tilde{f}_i^{\text{new}}(s_i) = \arg\min_{\tilde{f}_i} \text{KL}\Big(f_i(s_i) q_{-\tilde{f}_i}(s) \Big\| \tilde{f}_i(s_i) q_{-\tilde{f}_i}(s) \Big).$$

Since both $f_i(s_i)$ and $\tilde{f}_i(s_i)$ are Bernoulli distributions, they belong to the same exponential family, allowing direct moment matching. The expectation under $f_i(s_i)$ is:

$$\mathbb{E}_{f_i}[s_i] = \frac{e^{b_i}}{1 + e^{b_i}}.$$

Similarly, under the cavity distribution $q_{-\tilde{f}_i}(s)$, the mean is determined by contributions from neighboring nodes:

$$q_{-\tilde{f}_i}(s_i) \propto \exp\left(\sum_{j \neq i} \theta_{ji} s_i \right).$$

Thus, the expected value is

$$\mathbb{E}_{q_{-\tilde{f}_i}}[s_i] = \frac{e^{\sum_{j \neq i} \theta_{ji}}}{1 + e^{\sum_{j \neq i} \theta_{ji}}}.$$

Matching these expectations gives:

$$\frac{e^{b_i}}{1+e^{b_i}} = \frac{e^{\sum_{j\neq i}\theta_{ji}}}{1+e^{\sum_{j\neq i}\theta_{ji}}}.$$

Solving for θ_{ii}, we obtain:

$$\theta_{ii} = b_i.$$

Thus, the singleton sites are directly set as:

$$\tilde{f}_i(s_i) = \exp(b_i s_i).$$

Since $\tilde{f}_i(s_i)$ exactly matches the true factor $f_i(s_i)$, no further approximations are needed.

To update $\tilde{g}_{ij}(s_i, s_j)$, we define the cavity distribution:

$$q_{-\tilde{g}_{ij}}(s_i, s_j) = \frac{q(s)}{\tilde{g}_{ij}(s_i, s_j)}.$$

Since $q(s)$ is factorized over all pairs except (i, j), the cavity distribution factors as:

$$q_{-\tilde{g}_{ij}}(s_i, s_j) = A(s_i)B(s_j),$$

where:

$$A(s_i) = \exp(\eta_{i\neg j} s_i), \quad B(s_j) = \exp(\eta_{j\neg i} s_j),$$

with:

$$\eta_{i\neg j} = b_i + \sum_{m\neq i,j} \theta_{mi}, \quad \eta_{j\neg i} = b_j + \sum_{m\neq i,j} \theta_{mj}.$$

The true pairwise factor is:

$$g_{ij}(s_i, s_j) = \exp(W_{ij} s_i s_j).$$

Thus, the tilted distribution becomes:

$$f_{ij}(s_i, s_j) = g_{ij}(s_i, s_j) q_{-\tilde{g}_{ij}}(s_i, s_j),$$

which expands as:

$$f_{ij}(s_i, s_j) = \exp(W_{ij} s_i s_j) \exp(\eta_{i\neg j} s_i + \eta_{j\neg i} s_j).$$

EP minimizes the KL divergence:

$$\tilde{g}_{ij}^{\text{new}} = \arg\min_{\tilde{g}_{ij}} \mathrm{KL}\Big(f_{ij}(s_i, s_j) \,\Big\|\, \tilde{g}_{ij}(s_i, s_j) q_{-\tilde{g}_{ij}}(s_i, s_j) \Big).$$

Since $g_{ij}(s_i, s_j)$ is a pairwise Bernoulli factor, we approximate it by a factorized Bernoulli distribution:

$$\tilde{g}_{ij}(s_i, s_j) = \exp\Big(\theta_{ji} s_i + \theta_{ij} s_j\Big).$$

Expanding the log representation:

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

Simplifying:

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

Thus, the first moments are:

$$\mathbb{E}_{s_i}\left[\sum_{s_j\in\{0,1\}} \tilde{g}_{ij}(s_i,s_j)q_{-\tilde{g}_{ij}}(s_i,s_j)\right] = \frac{1}{1+\exp(-(\theta_{ji}+\eta_{i\neg j}))},$$

$$\mathbb{E}_{s_j}\left[\sum_{s_i\in\{0,1\}} \tilde{g}_{ij}(s_i,s_j)q_{-\tilde{g}_{ij}}(s_i,s_j)\right] = \frac{1}{1+\exp(-(\theta_{ij}+\eta_{j\neg i}))}.$$

We now expand the true factor:

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

To derive the first moment for $g_{ij}(s_i,s_j)q_{-\tilde{g}_{ij}}(s_i,s_j)$ with respect to s_i, we marginalize out s_j:

$$\sum_{s_j\in\{0,1\}} g_{ij}(s_i,s_j)q_{-\tilde{g}_{ij}}(s_i,s_j) \propto \exp(W_{ij}s_i+\eta_{j\neg i}) + \exp(\eta_{i\neg j}s_i).$$

Thus, the first moment:

$$\mathbb{E}_{s_i}\left[\sum_{s_j\in\{0,1\}} g_{ij}(s_i,s_j)q_{-\tilde{g}_{ij}}(s_i,s_j)\right] = \frac{\exp(W_{ij}+\eta_{j\neg i})+\exp(\eta_{i\neg j})}{\exp(W_{ij}+\eta_{j\neg i})+\exp(\eta_{i\neg j})+1}.$$

Similarly:

$$\mathbb{E}_{s_j}\left[\sum_{s_i\in\{0,1\}} g_{ij}(s_i,s_j)q_{-\tilde{g}_{ij}}(s_i,s_j)\right] = \frac{\exp(W_{ij}+\eta_{i\neg j})+\exp(\eta_{j\neg i})}{\exp(W_{ij}+\eta_{i\neg j})+\exp(\eta_{j\neg i})+1}.$$

By moment matching, we equate:

$$\mathbb{E}_{s_i}\left[\sum_{s_j\in\{0,1\}} \tilde{g}_{ij}(s_i,s_j)q_{-\tilde{g}_{ij}}(s_i,s_j)\right] = \mathbb{E}_{s_i}\left[\sum_{s_j\in\{0,1\}} g_{ij}(s_i,s_j)q_{-\tilde{g}_{ij}}(s_i,s_j)\right].$$

Solving for θ_{ji}:

$$\frac{1}{1+\exp(-(\theta_{ji}+\eta_{i\neg j}))} = \frac{\exp(W_{ij}+\eta_{j\neg i})+\exp(\eta_{i\neg j})}{\exp(W_{ij}+\eta_{j\neg i})+\exp(\eta_{i\neg j})+1}.$$

Rearranging:

$$\frac{1}{1+\exp(-(\theta_{ji}+\eta_{i\neg j}))} = \frac{\exp(\eta_{i\neg j})+\exp(W_{ij}+\eta_{j\neg i})}{1+\exp(\eta_{i\neg j})+\exp(\eta_{j\neg i})+\exp(W_{ij}+\eta_{i\neg j}+\eta_{j\neg i})}.$$

This implies:

$$\exp(-(\theta_{ji}+\eta_{i\neg j})) = \frac{1+\exp(\eta_{j\neg i})+\exp(\eta_{i\neg j}+\eta_{j\neg i}+W_{ij})}{\exp(\eta_{i\neg j})+\exp(W_{ij}+\eta_{j\neg i})}.$$

Taking the logarithm:

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

Similarly, solving for θ_{ij}:

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

Thus, the final parameter updates are:

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

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

So the final message passing scheme is:

- Compute cavity parameters:

$$\eta_{i\neg j} = b_i + \sum_{m\neq i,j} \theta_{mi}, \quad \eta_{j\neg i} = b_j + \sum_{m\neq i,j} \theta_{mj}.$$

- Update site parameters:

$$\theta_{ji} \leftarrow \log\left(\frac{1+\exp(W_{ij}+\eta_{i\neg j})}{1+\exp(\eta_{i\neg j})}\right),$$

$$\theta_{ij} \leftarrow \log\left(\frac{1+\exp(W_{ij}+\eta_{j\neg i})}{1+\exp(\eta_{j\neg i})}\right).$$

These updates iterate until convergence.

5.3 Loopy Belief Propagation and Factorized Messages

We now reformulate the Expectation Propagation (EP) message passing scheme using factored approximate messages and demonstrate how this leads to a Loopy Belief Propagation (BP) algorithm.

In the original EP framework, the pairwise factor $g_{ij}(s_i, s_j)$ was approximated as:

$$\tilde{g}_{ij}(s_i, s_j) \approx \tilde{g}_{ij}(s_i)\tilde{g}_{ij}(s_j),$$

where we assume that the dependencies between s_i and s_j are captured through independent Bernoulli distributions.

Thus, the approximate posterior can be rewritten as:

$$q(s) = \frac{1}{Z}\prod_i f_i(s_i)\prod_{ij}\tilde{g}_{ij}(s_i, s_j).$$

By substituting the factorized approximation, this becomes:

$$q(s) = \frac{1}{Z} \prod_i f_i(s_i) \prod_{ij} \mathcal{M}_{j\to i}(s_i)\mathcal{M}_{i\to j}(s_j),$$

where we define messages as:

$$\tilde{g}_{ij}(s_i) = \mathcal{M}_{j\to i}(s_i), \quad \tilde{g}_{ji}(s_j) = \mathcal{M}_{i\to j}(s_j).$$

Using the cavity distribution, we express the updated messages as:

$$\mathcal{M}^{\text{new}}_{j\to i}(s_i) = \sum_{s_j} g_{ij}(s_i, s_j) f_j(s_j) \prod_{m\in \text{ne}(j)\setminus i} \mathcal{M}_{m\to j}(s_j),$$

$$\mathcal{M}^{\text{new}}_{i\to j}(s_j) = \sum_{s_i} g_{ij}(s_i, s_j) f_i(s_i) \prod_{m\in \text{ne}(i)\setminus j} \mathcal{M}_{m\to i}(s_i).$$

Since we approximate pairwise factors as a product of singleton messages, the messages iteratively refine the estimates of marginals rather than exact pairwise beliefs.

In a tree-structured graph, messages can be passed in a directed fashion from leaves to root, leading to exact marginals. However, in a fully connected Boltzmann Machine, every latent variable s_i is connected to all others, forming a complete subgraph.

Since messages are not disjoint but depend on each other, the iterative update process revisits nodes multiple times, forming loops in message passing. This results in Loopy BP, where updates propagate iteratively until convergence:

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

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

These recursive dependencies prevent exact inference in a single pass, leading to cycles in the update scheme.

By rewriting the EP message updates using factored approximations, we recover an iterative message-passing scheme. However, due to the fully connected nature of the underlying graphical model, this results in Loopy BP, where updates propagate cyclically rather than in a tree-like fashion.

This formulation allows efficient inference but at the cost of approximate marginals, which may not be exact due to the looped dependencies. The EP framework helps mitigate errors by iteratively refining site approximations.

5.4 Bayesian Model Selection Using EP

To determine the number of hidden binary variables K in a Bayesian manner, we employ Automatic Relevance Determination (ARD) within the Expectation Propagation (EP) framework. ARD provides a principled approach to pruning irrelevant latent variables by assigning hierarchical Gaussian priors to their weights.

In this approach, we introduce a latent variable π that represents the prior probability of each hidden binary variable being active. The joint probability of the model is given by:

$$P(s, x, \pi) = P(s|\pi)P(x|s, \theta)P(\pi)$$

where the prior over s, denoted as $P(s|\pi)$, follows a Bernoulli distribution for each hidden variable. Additionally, $P(\pi)$ is a hierarchical Gaussian prior, with each π_i following a normal distribution parameterized by α_i^{-1}. The likelihood term, $P(x|s, \theta)$, follows a standard factor model assumption.

The primary goal of ARD is to learn the hyperparameters α_i, which control the relevance of each latent variable. Expectation Propagation (EP) is used to approximate the posterior distribution over the latent variables while ensuring computational tractability. The approximation takes the form:

$$q(s) \approx \prod_{i=1}^{K} q_i(s_i).$$

The EP framework iteratively refines this approximation through three key steps. First, site approximation updates refine each factor in the posterior using moment matching. Second, cavity distributions are computed by removing the influence of a specific site, allowing an updated estimate of its contribution. Finally, a Kullback-Leibler (KL) minimization step updates each site factor by minimizing the divergence between the true and approximate posterior distributions.

The ARD framework determines the relevance of each latent variable by learning the values of α_i. If α_i grows to infinity, then the corresponding prior π_i collapses to zero, effectively removing that hidden variable from the model. The number of relevant latent variables is then computed as:

$$K = \sum_{i=1}^{K_{\max}} \mathbb{I}(\alpha_i < \tau),$$

where τ is a predefined threshold ensuring numerical stability.

Despite its advantages, this approach presents several computational challenges. The presence of cycles in the Boltzmann machine structure introduces difficulties due to loopy belief propagation, leading to slow convergence. One way to mitigate this issue is through Power EP, which stabilizes updates using a damping factor. Additionally, high-dimensional parameter spaces pose a challenge, as large values of $K_{\max}$ result in an exponential number of latent states. This issue can be addressed by grouping latent variables into supernodes or selectively pruning interactions.

Finally, hyperparameter optimization instability may arise due to fluctuations in the iterative updates of EP. To counteract this, early stopping criteria based on free energy minimization can be employed to ensure a stable selection of K.

By integrating ARD within the EP framework, we develop a Bayesian approach for automatically selecting the optimal number of latent variables. The EP-based site approximations allow for efficient inference while ARD ensures that irrelevant variables are pruned. Although challenges such as loopy dependencies and hyperparameter instability exist, these can be mitigated through structured approximations and optimization techniques. As a result, this method provides an efficient and principled approach to determining the dimensionality of the latent space in probabilistic models.

6 (Bonus) Implementing and Comparing EP/Loopy-BP

```python
def initialize_messages(n, k):
    """ Initialize small random messages for stability """
    return np.random.rand(n, k, k) * 0.01

def compute_local_potentials(x, mu, sigma, pi):
    """ Compute local potentials (log-odds) for each node """
    precision = 1 / (sigma ** 2)
    self_m = np.diag(mu.T @ mu)
    return logit(pi) + (x @ mu) * precision - self_m / (2 * sigma ** 2)

def update_messages(x, mu, sigma, pi, messages, max_iterations=50, damping=0.35, tol=1e-6):
    """ Loopy Belief Propagation for updating messages """
    n, k, _ = messages.shape
    f = compute_local_potentials(x, mu, sigma, pi)

    for _ in range(max_iterations):
        new_messages = np.copy(messages)

        for i in range(k):
            for j in range(i + 1, k):
                np_old = f[:, j] + np.sum(messages[:, :, j], axis=1) - messages[:, i, j]
                np_true = - (mu[:, i] @ mu[:, j]) / sigma ** 2
                np_new = (1 + np.exp(np_old + np_true)) / (1 + np.exp(np_old))

                # Apply damping for stability
                new_messages[:, i, j] = damping * messages[:, i, j] + (1 - damping) * np.log(np_new)
                new_messages[:, j, i] = damping * messages[:, j, i] + (1 - damping) * np.log(np_new)

        if np.max(np.abs(new_messages - messages)) < tol:
            break  # Stop if messages have converged

        messages = new_messages

    lambda_matrix = expit(f + np.sum(messages, axis=2))  # Compute posteriors
    return messages, lambda_matrix

def m_step(data, lambda_matrix):
    """ Maximization step: Update mu, sigma, and pi """
    N, D = data.shape
    K = lambda_matrix.shape[1]

    ES = lambda_matrix
    ESS = ES.T @ ES
```

```
    np.fill_diagonal(ESS, np.sum(ES, axis=0))  # Diagonal correction

    inv_ESS = np.linalg.inv(ESS)
    ES_T_X = ES.T @ data
    mu = (inv_ESS @ ES_T_X).T  # Update mu

    # Compute sigma
    term1 = np.trace(data.T @ data)
    term2 = np.trace((mu.T @ mu) @ ESS)
    term3 = 2.0 * np.trace((ES.T @ data) @ mu)
    sigma_sq = (term1 + term2 - term3) / (N * D)
    sigma = np.sqrt(np.maximum(sigma_sq, 1e-12))  # Prevent numerical issues

    # Compute prior probabilities
    pi = np.mean(ES, axis=0, keepdims=True)
    pi = np.clip(pi, 1e-6, 1.0 - 1e-6)  # Avoid extreme values

    return mu, sigma, pi

def compute_entropy(lambda_matrix: np.ndarray) -> float:
    """
    Compute the entropy term -E_q[log q(S)] for free energy calculation.
    """
    lam_clipped = np.clip(lambda_matrix, 1e-12, 1 - 1e-12)
    return -np.sum(
        lam_clipped * np.log(lam_clipped) + (1 - lam_clipped) * np.log(1 -
        ↪ lam_clipped)
    )

def compute_log_p_x_given_s(data: np.ndarray, mu: np.ndarray, sigma: float,
↪ lambda_matrix: np.ndarray) -> float:
    """
    Compute E_q[log p(X | S)] term in free energy.
    """
    N, D = data.shape
    precision = 1.0 / (sigma**2)

    # Compute (X - sum_k lambda_k mu_k)^2
    mu_product = lambda_matrix @ mu.T
    x_sq_sum = np.sum(data * data)
    cross_term = -2.0 * np.sum(data * mu_product)
    mu_sq_sum = np.sum(mu_product * mu_product)

    log_p_x_s = -0.5 * precision * (x_sq_sum + mu_sq_sum + cross_term)
    log_constant = -0.5 * N * D * np.log(2.0 * np.pi * (sigma**2))

    return log_constant + log_p_x_s

def compute_log_p_s(pi: np.ndarray, lambda_matrix: np.ndarray) -> float:
    """
```

```
    Compute E_q[log p(S)] term in free energy.
    """
    pi = np.clip(pi, 1e-12, 1 - 1e-12)  # Ensure valid log inputs
    lam_clipped = np.clip(lambda_matrix, 1e-12, 1 - 1e-12)

    log_pi = np.log(pi)
    log_one_minus_pi = np.log(1 - pi)
    log_ps = lam_clipped * log_pi + (1 - lam_clipped) * log_one_minus_pi

    return np.sum(log_ps)

def compute_free_energy(data: np.ndarray, mu: np.ndarray, sigma: float, pi:
↪  np.ndarray, lambda_matrix: np.ndarray) -> float:
    """
    Compute the Free Energy (ELBO) for the Loopy BP EM algorithm.
    Returns a properly scaled negative free energy value.
    """
    term_x_given_s = compute_log_p_x_given_s(data, mu, sigma, lambda_matrix)
    term_s = compute_log_p_s(pi, lambda_matrix)
    entropy = compute_entropy(lambda_matrix)

    # Normalize free energy per data point
    N = data.shape[0]
    return (term_x_given_s + term_s + entropy) / N


def em_algorithm(data, mu, sigma, pi, max_em_iterations=200, tol=1e-6):
    """ Full Expectation-Maximization (EM) Algorithm """
    free_energy_history = []
    lambda_matrix = np.random.rand(data.shape[0], mu.shape[1])

    for iteration in range(max_em_iterations):
        # E-Step: Loopy BP
        messages, lambda_matrix = update_messages(
            data, mu, sigma, pi, initialize_messages(data.shape[0], mu.shape[1])
        )

        # M-Step
        mu, sigma, pi = m_step(data, lambda_matrix)

        # Compute Free Energy
        free_energy = compute_free_energy(data, mu, sigma, pi, lambda_matrix)
        free_energy_history.append(free_energy)

        print(f"Iteration {iteration+1}: Free Energy = {free_energy:.5f}")

        # Convergence check: Only stop if 2+ iterations & free energy stabilizes
        if iteration > 2 and abs(free_energy_history[-1] - free_energy_history[-2])
        ↪  < tol:
            break
```

```
140
141     return mu, sigma, pi, lambda_matrix, free_energy_history
142
143 def plot_results(mu, free_energy_history, save_path="output"):
144     """ Plot final results: Learned features and Free Energy trend """
145     K = mu.shape[1]
146
147     # Plot Free Energy Trend
148     plt.figure()
149     plt.plot(free_energy_history, marker=None)
150     plt.title("Free Energy (Loopy BP)")
151     plt.xlabel("EM Iteration")
152     plt.ylabel("Free Energy")
153     plt.savefig(f"{save_path}-free-energy.png")
154     plt.close()
155
156     # Plot Initial and Learned Features
157     fig, axes = plt.subplots(1, K, figsize=(K * 2, 2))
158     for i in range(K):
159         axes[i].imshow(mu[:, i].reshape(4, 4), cmap='gray', interpolation='none')
160         axes[i].set_title(f"Feature {i+1}")
161         axes[i].axis('off')
162     fig.suptitle("Learned Features (Loopy BP)")
163     plt.tight_layout()
164     plt.savefig(f"{save_path}-latent-factors.png")
165     plt.close()
166
167 if __name__ == "__main__":
168     # Load Data
169     np.random.seed(0)
170     D = data.shape[1]
171     K = 8
172
173     # Initialize model parameters
174     mu_init = np.random.randn(D, K) * 0.1
175     sigma_init = 1.0
176     pi_init = np.full((1, K), 0.5)
177
178     # Run EM Algorithm
179     mu_final, sigma_final, pi_final, lambda_final, free_energy_history =
        ↪ em_algorithm(
180         data, mu_init, sigma_init, pi_init
181     )
182
183     # Plot Results
184     plot_results(mu_final, free_energy_history)
```

Results:

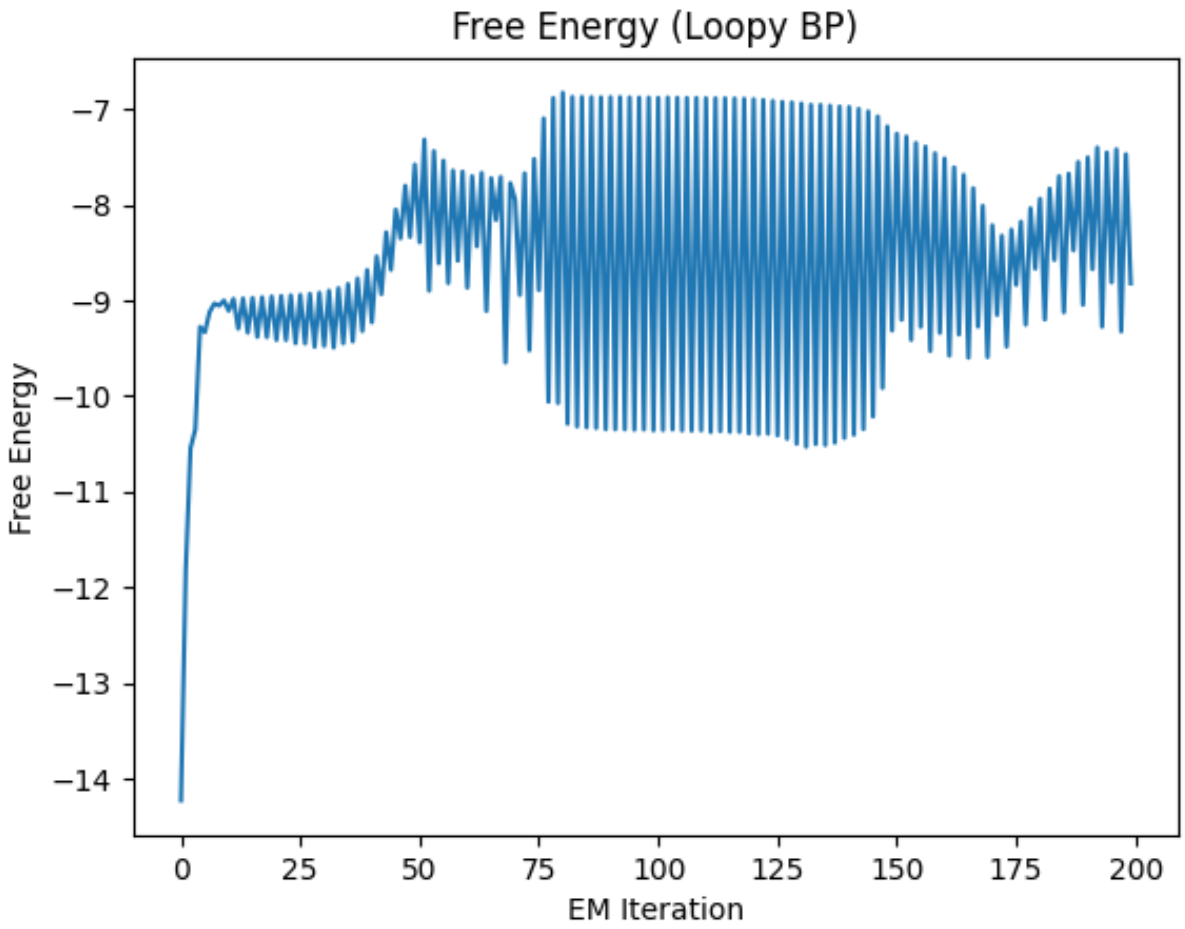


Figure 16: Free Energy convergence using Loopy Belief Propagation (LBP) across EM iterations. The oscillations in free energy indicate challenges in achieving stable convergence compared to the smoother progression observed in the mean-field approximation.

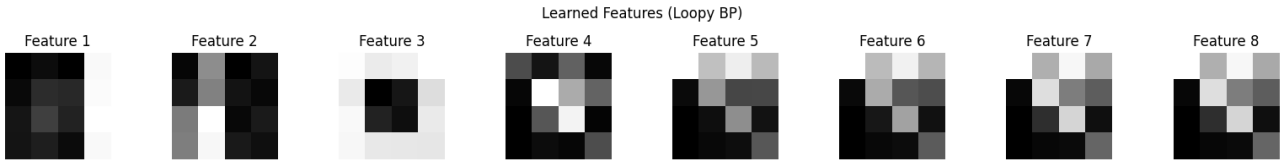


Figure 17: Learned latent features using Loopy Belief Propagation (LBP) visualized as 4×4 greyscale images.

Loopy Belief Propagation (LBP) and Mean-Field Variational Inference (MFVI) differ significantly in their approaches to approximate inference. MFVI assumes a fully factorized posterior, ensuring stable and monotonic convergence of free energy but at the cost of neglecting correlations between latent variables. In contrast, LBP uses iterative message passing to capture complex dependencies, yielding richer latent representations. However, this expressivity introduces instability, as observed in the oscillatory free energy plot, a behavior absent in MFVI.

To address this instability, damping was implemented in LBP, blending new and previous updates,

which reduced oscillations and improved convergence stability. Despite this improvement, LBP remains computationally more demanding due to its iterative updates across dependencies, unlike MFVI's independent variable updates. While MFVI offers stable and scalable solutions, LBP provides a more expressive alternative capable of capturing richer latent structures, though at the cost of greater complexity and sensitivity to initialization.

As we can see, the results from LBP are good and seemed to have captured the resulting latent variables well, just as MFVI did.