

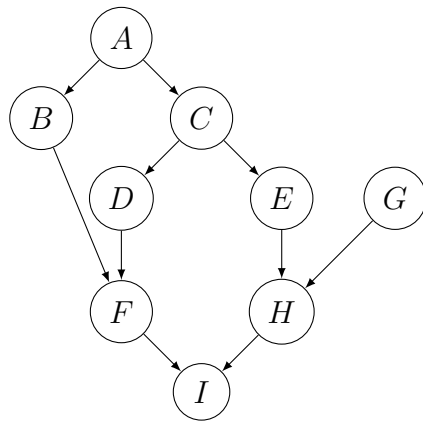
COMP0085 Summative Assignment

Jan 4, 2023

Question 1

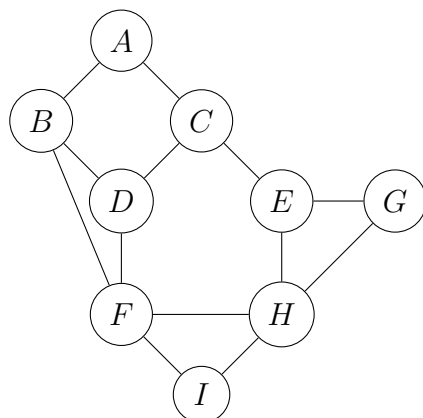
(a)

The directed acyclic graph:

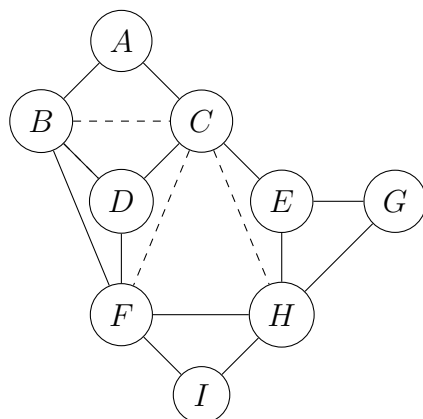


(b)

The moralised graph:

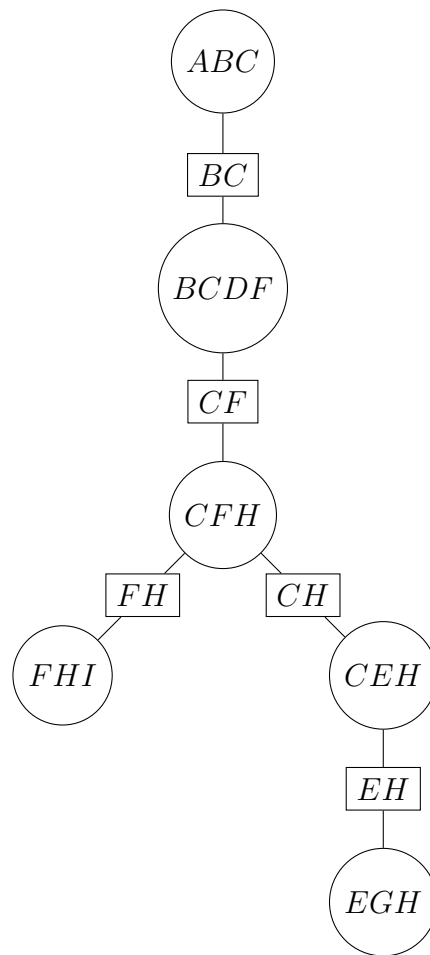


An effective triangulation:

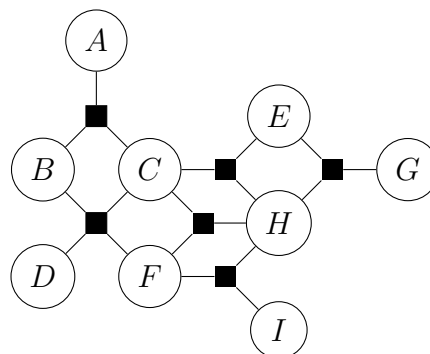


where the dashed lines are edges added to triangulate the moralised graph.

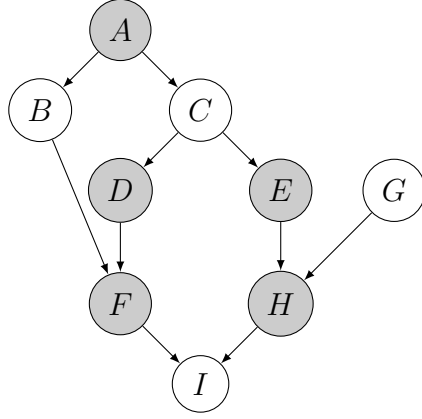
The resulting junction tree:



where the circular nodes are cliques and the square nodes are separators/factors.
 The junction tree redrawn as a factor graph:



(c)



The set $\{A, D, E, F, H\}$ is a non-unique smallest set 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 (conditioned on the measured ones).

(d)

Using our factor analysis model, we can describe the biochemical pathway as:

$$\delta[\mathbf{x}] = \Lambda \mathbf{z} + \epsilon$$

where $\delta[\mathbf{x}]$ are the concentration perturbations, $\epsilon \sim \mathcal{N}(0, \Psi)$, and the latent factors $z \sim \mathcal{N}(0, I)$. From the graph structure, we know that:

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

Having observations for $\delta[B]$, $\delta[D]$, $\delta[E]$ and $\delta[G]$:

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

We can see that these simplify to the equations:

$$\begin{aligned}
\delta[B] &= \Lambda_{BA}z_A + \epsilon_B \\
\delta[D] &= \Lambda_{DC}z_C + \epsilon_D \\
\delta[E] &= \Lambda_{EC}z_C + \epsilon_E \\
\delta[G] &= \epsilon_G
\end{aligned}$$

Thus, we see that the only latent variables present are z_A and z_C , so would expect to recover the factors of A and C , the two parent nodes of the observations.

(e)

Question 2

(a)

We want the posterior mean and covariance over a and b . Defining a weight vector \mathbf{w} :

$$\mathbf{w} = \begin{bmatrix} a \\ b \end{bmatrix}$$

Our distribution for \mathbf{w} :

$$P(\mathbf{w}) = \mathcal{N} \left(\begin{bmatrix} \mu_a \\ \mu_b \end{bmatrix}, \begin{bmatrix} \sigma_a^2 & 0 \\ 0 & \sigma_b^2 \end{bmatrix} \right) = \mathcal{N}(\mu_{\mathbf{w}}, \Sigma_{\mathbf{w}})$$

Moreover, for our data $\mathcal{D} = \{\mathbf{X}, \mathbf{Y}\}$:

$$P(\mathcal{D}|\mathbf{w}) = \mathcal{N}(\mathbf{Y} - \mathbf{w}^T \mathbf{X}, \sigma^2 \mathbf{I})$$

where $\mathbf{X} = \begin{bmatrix} t_1 & t_2 & \dots & t_N \\ 1 & 1 & \dots & 1 \end{bmatrix} \in \mathbb{R}^{2 \times N}$ and $\mathbf{Y} \in \mathbb{R}^{1 \times N}$.

Knowing:

$$P(\mathbf{w}|\mathcal{D}) \propto P(\mathcal{D}|\mathbf{w})P(\mathbf{w})$$

we can substitute the above distributions:

$$P(\mathbf{w}|\mathcal{D}) \propto \exp \left(\frac{-1}{2\sigma^2} (\mathbf{Y} - \mathbf{w}^T \mathbf{X}) (\mathbf{Y} - \mathbf{w}^T \mathbf{X})^T \right) \exp \left(\frac{-1}{2} (\mathbf{w} - \mu_{\mathbf{w}})^T \Sigma_{\mathbf{w}}^{-1} (\mathbf{w} - \mu_{\mathbf{w}}) \right)$$

expanding:

$$\log P(\mathbf{w}|\mathcal{D}) \propto \frac{-1}{2} \left(\frac{\mathbf{Y}\mathbf{Y}^T}{\sigma^2} - 2\mathbf{w}^T \frac{\mathbf{X}\mathbf{Y}^T}{\sigma^2} + \mathbf{w}^T \frac{\mathbf{X}\mathbf{X}^T}{\sigma^2} \mathbf{w} + \mathbf{w}^T \Sigma_{\mathbf{w}}^{-1} \mathbf{w} - 2\mathbf{w}^T \Sigma_{\mathbf{w}}^{-1} \mu_{\mathbf{w}} + \mu_{\mathbf{w}}^T \Sigma_{\mathbf{w}}^{-1} \mu_{\mathbf{w}} \right)$$

collecting \mathbf{w} terms:

$$\log P(\mathbf{w}|\mathcal{D}) \propto \frac{-1}{2} \left(\mathbf{w}^T \left(\frac{\mathbf{X}\mathbf{X}^T}{\sigma^2} + \Sigma_{\mathbf{w}}^{-1} \right) \mathbf{w} - 2\mathbf{w}^T \left(\frac{\mathbf{X}\mathbf{Y}^T}{\sigma^2} + \Sigma_{\mathbf{w}}^{-1} \mu_{\mathbf{w}} \right) \right)$$

Knowing that the posterior $P(\mathbf{w}|\mathcal{D})$ will be Gaussian with mean $\bar{\mu}_w$ and covariance $\bar{\Sigma}_w$, we can see that expanding the exponent component would have the form:

$$(\mathbf{w} - \bar{\mu}_w)^T \bar{\Sigma}_w^{-1} (\mathbf{w} - \bar{\mu}_w) = \mathbf{w}^T \bar{\Sigma}_w^{-1} \mathbf{w} - 2\mathbf{w}^T \bar{\Sigma}_w^{-1} \bar{\mu}_w + \bar{\mu}_w^T \bar{\Sigma}_w^{-1} \bar{\mu}_w$$

Thus we can identify the posterior covariance:

$$\bar{\Sigma}_w = \left(\frac{\mathbf{X}\mathbf{X}^T}{\sigma^2} + \Sigma_{\mathbf{w}}^{-1} \right)^{-1}$$

and the posterior mean:

$$\bar{\mu}_w = \bar{\Sigma}_w \left(\frac{\mathbf{X}\mathbf{Y}^T}{\sigma^2} + \Sigma_{\mathbf{w}}^{-1} \mu_{\mathbf{w}} \right)$$

Computing the posterior mean and covariance over a and b given by the CO_2 data:

value		
parameters	a	1.828457
	b	334.203782

Figure 1: The Posterior Mean

parameters			
	a	b	
parameters	a	0.000014	-0.000287
	b	-0.000287	0.007976

Figure 2: The Posterior Covariance

(b)

Plotting the residuals:

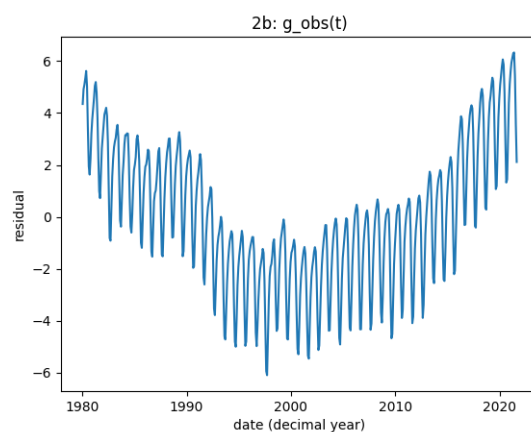


Figure 3: $g_{obs}(t)$

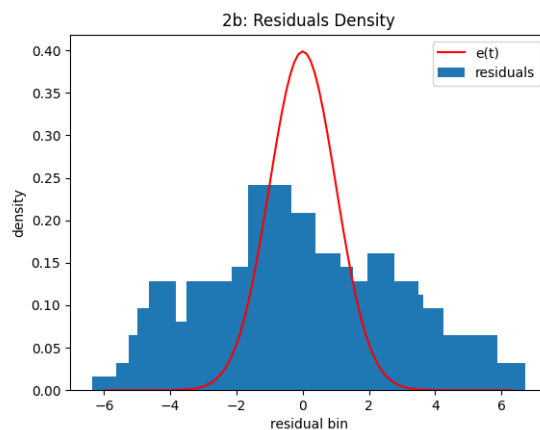


Figure 4: Density Estimation of Residuals vs $e(t) \sim \mathcal{N}(0, 1)$

We can see that the residuals do not perfectly conform to our prior over $e(t) \sim \mathcal{N}(0, 1)$. The density estimation shows that a mean of zero is a reasonable prior belief however the data does not seem to exhibit unit variance. Also we know it's not iid because timeseries.

(c & d)

We are considering the kernel:

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

We can make qualitative observations this kernel by visualising the covariance (gram) matrix:

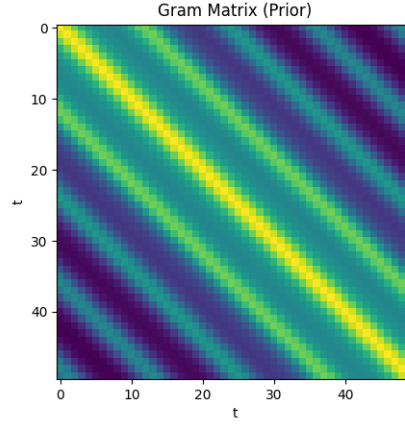


Figure 5: Covariance Matrix

We can observe a striped pattern which indicate higher covariance at regular intervals. This can be attributed to the sinusoidal term in the kernel and encourages sinusoidal functions. Additionally, we can see that covariance values also decay as they are further away from the diagonal. This can be attributed to the exponential term in the kernel, encouraging points closer in time to be more correlated and vice versa. From our CO_2 data, we would want a class of functions which exhibit both of these behaviours as the data looks sinusoidal (seasonal with respect to each year) and correlations locally.

We can also visualise some samples from a Gaussian Process with the same covariance matrix and zero mean. This verifies our observations about the covariance matrix.

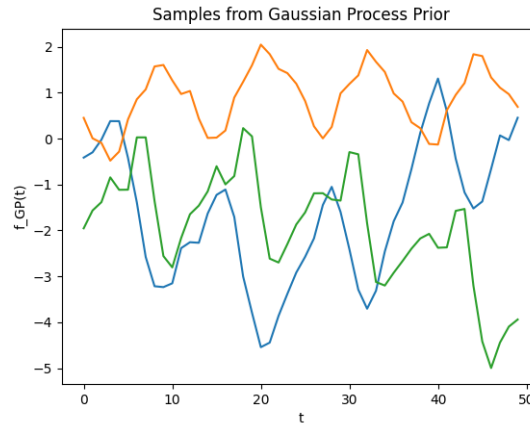


Figure 6: Samples from a zero mean GP with the provided covariance kernel

More specifically, we can see how changing each hyper-parameter will affect the characteristics of the function.

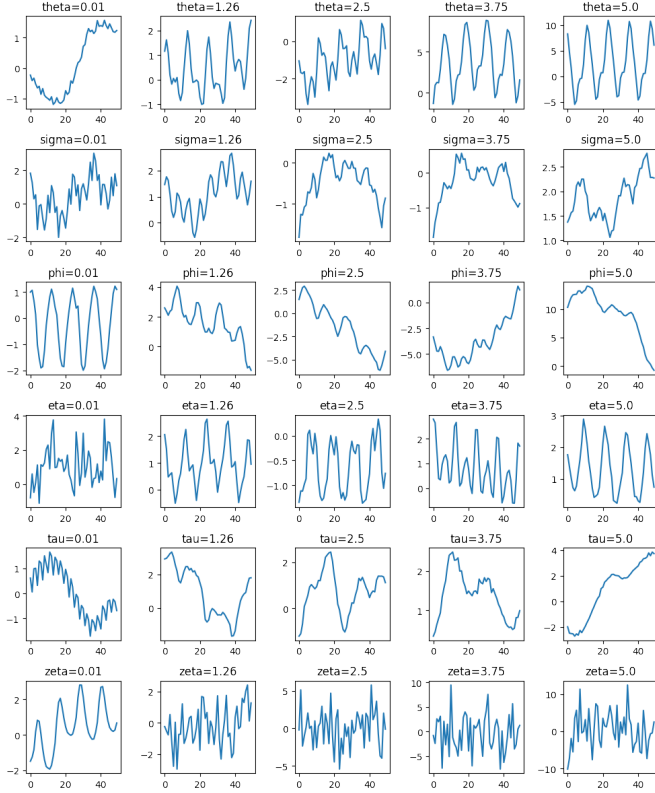


Figure 7: Samples for different parameters

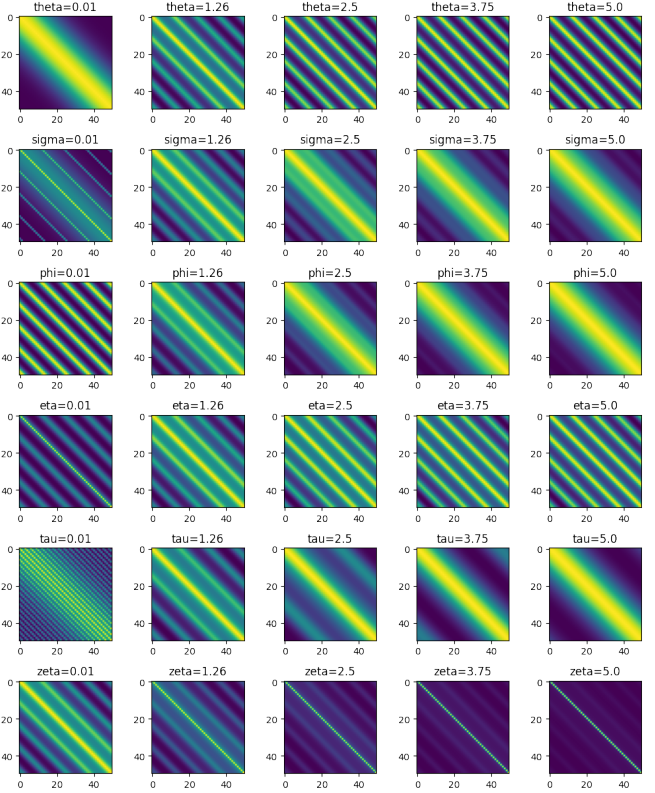


Figure 8: Covariances for different parameters

θ : As θ increases, we see more pronounced periodic behavior in the sample function. The covariance matrix shows how increasing θ visually reveals the striped periodic component. This is expected because it is the parameter that adjusts the weight of the periodic component.

σ : As σ increases, we see reduced periodic behaviour in the sample function. The covariance matrix shows how increasing σ will increase covariance values in the off-diagonals. This is expected because it adjusts the lengthscale of the periodic portion of the kernel, which ends up dominating the function.

ϕ : As ϕ increases, we see the ratio of the amplitude of the periodicity component of the sample function reduces compared to the baseline. The covariance matrix shows how increasing ϕ will start to increase the non-periodic component. This is expected because it adjusts the weight of the non-periodic portion of the kernel, thus the periodic component remains the same (i.e. same amplitude) but the large baseline shifts from increasing ϕ ends up dominating the function visually.

η : As η increases we see smoother sample functions. This is expected because the η increases the lengthscale of the non-periodic component, allowing for smoother functions. This causes the off-diagonals of the gram matrix to increase, however the periodic component is still maintained because η doesn't affect the relative weight of the two components.

- τ : As τ increases, the period of the periodic function increases. We can see this reflected in the stripes in the gram matrix getting further apart. This makes sense because we are adjusting the period in the sinusoid function of the periodic term with τ .
- ζ : As ζ increases, the function becomes less smooth. This is because the ζ parameter adjusts the weight of the $\delta_{s=t}$ parameter. This places stronger emphasis on the independence of each timestep, which can be seen with the reduction of relative magnitude of off-diagonals in the gram matrix. However, this is simply masking the periodic and squared-exponential terms as we can see with the increased magnitude of the functions as ζ increases.

(e)

Suitable values for hyper-parameters can be chosen through a combination of visual inspection and prior knowledge. For example, it is a reasonable assumption that the CO_2 concentration levels have a strong yearly seasonality behaviour due to the cyclic changes in temperature, humidity, etc. Thus we can choose $\tau = 1$ to ensure functions with a period of one year to reflect this knowledge. It can be difficult to quantitatively choose values for the other parameters as they can relate to the uncertainty exhibited in the data (i.e. the smoothness of the function). One approach is to maximise:

$$\log P(\mathbf{Y}|\mathbf{X}) = -\frac{1}{2}\mathbf{Y}^T(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{Y} - \frac{1}{2}\log |\mathbf{K} + \sigma^2\mathbf{I}| - \frac{n}{2}\log(2\pi)$$

the log-likelihood of the posterior distribution with respect to the given data where \mathbf{K} is the gram matrix for the kernel (equation 2.30 from <http://gaussianprocess.org/gpml/chapters/RW2.pdf>). We can define a loss function as the negative log-likelihood and employ gradient-based algorithms to find optimal parameters.

Comparing the hyperparameters corresponding to before and after training side by side:

parameter	value
eta (kernel)	5.0
phi (kernel)	10.0
sigma	1.0
sigma (kernel)	5.0
tau (kernel)	1.0
theta (kernel)	5.0
zeta (kernel)	2.0

Figure 9: Untrained hyperparameters

parameter	value
eta (kernel)	5.060295
phi (kernel)	4.991508
sigma	0.372548
sigma (kernel)	2.816059
tau (kernel)	0.998625
theta (kernel)	7.019629
zeta (kernel)	0.745096

Figure 10: Trained Hyperparameters

We can analyse some of the changes in these parameters after training to gain some insights. We can see that τ remains the same as we would expect given the yearly seasonality we have prior knowledge of. On the other hand, the value for ζ is significantly reduced signifying that $\delta_{s=t}$ is not a very good kernel for representing the data as datapoints at different timesteps do exhibit correlations.

(f)

Extrapolating the CO_2 concentration levels:

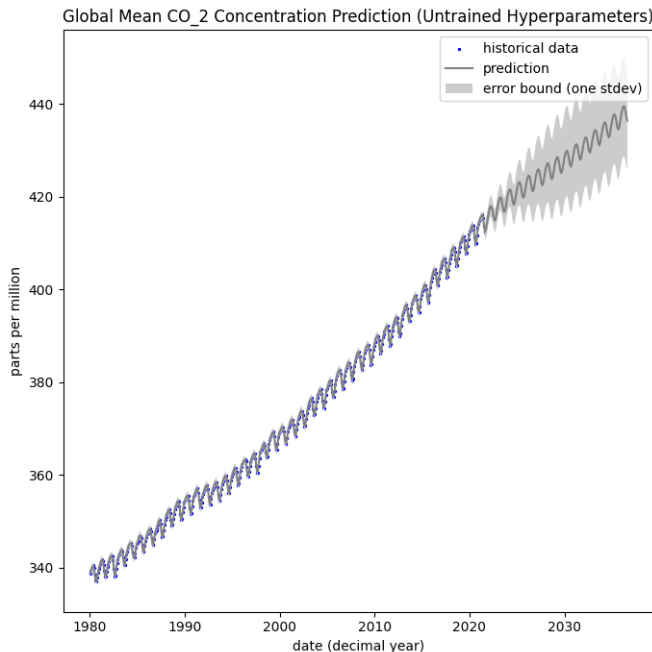


Figure 11: Untrained extrapolation

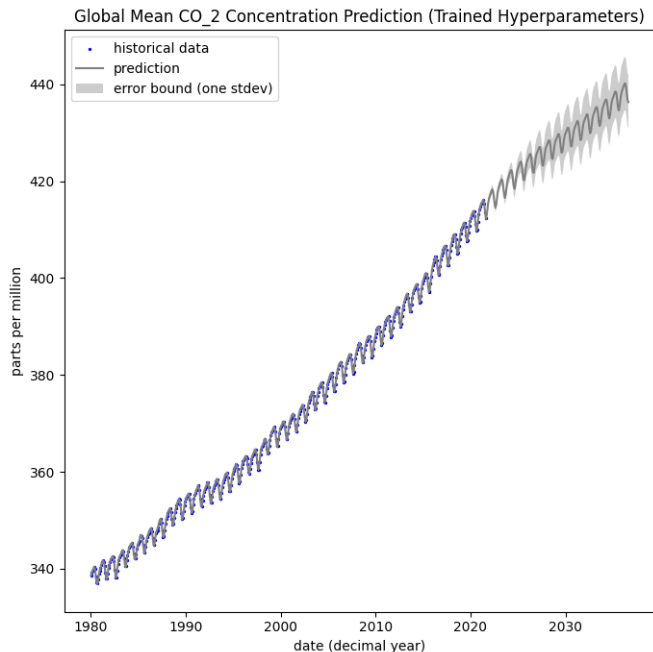


Figure 12: Trained extrapolation

We can see that the extrapolation shows a continued increase in CO_2 in the future. This follows our expectations given that the levels have been steadily increasing in the past. Moreover, the concentration continues to exhibit yearly seasonality (for the trained extrapolation) as we would expect. We can see that the conclusions can be quite sensitive to kernel hyperparameters when comparing the values from before and after training. Prior to training, the extrapolated prediction is not representative of the given data, with pretty much no seasonal behaviour and very large uncertainty. After training, we can see that the prediction is much more reasonable, and qualitatively the uncertainty bounds seem to exhibit the historical variability in the data.

(g)

This procedure is not fully Bayesian because despite using a posterior estimate of our linear regression terms, we only use a point estimate when making prediction. For a fully Bayesian approach, we should also incorporate the uncertainty of the linear regression parameters into our extrapolation/uncertainty bounds. For our procedure, we only include the uncertainty of $g(t)$ however it can be observed in the plots that the trend is not perfectly linear so this should be reflected in the uncertainty of our extrapolation. Another approach could be to add a linear kernel to our combined kernel function and model $f(t)$ directly with our kernel, removing the linear regression component in our procedure. Thus our kernel extrapolation would incorporate the uncertainty of all components of our signal.

The Python code for Bayesian Linear Regression:

```
1 from dataclasses import dataclass
2
3 import numpy as np
4
5
6 @dataclass
7 class LinearRegressionParameters:
8     mean: np.ndarray
9     covariance: np.ndarray
10
11     @property
12     def precision(self) -> np.ndarray:
13         return np.linalg.inv(self.covariance)
14
15     def predict(self, x: np.ndarray) -> np.ndarray:
16         return self.mean.T @ x
17
18
19 @dataclass
20 class Theta:
21     linear_regression_parameters: LinearRegressionParameters
22     sigma: float
23
24     @property
25     def variance(self) -> float:
26         return self.sigma**2
27
28     @property
29     def precision(self) -> float:
30         return 1 / self.variance
31
32
33 def compute_linear_regression_posterior(
34     x: np.ndarray,
35     y: np.ndarray,
36     prior_linear_regression_parameters: LinearRegressionParameters,
37     residuals_precision: float,
38 ) -> LinearRegressionParameters:
39     """
40     Compute the parameters of the posterior distribution on the linear regression weights
41
42     :param x: design matrix (number of features, number of data points)
43     :param y: response matrix (1, number of data points)
44     :param prior_linear_regression_parameters: parameters for the prior distribution on the linear regression
45           weights
46     :param residuals_precision: the precision of the residuals of the linear regression
47     :return: parameters for the posterior distribution on the linear regression weights
48     """
49     posterior_covariance = np.linalg.inv(
50         residuals_precision * x @ x.T + prior_linear_regression_parameters.precision
51     )
52     posterior_mean = posterior_covariance @ (
53         residuals_precision * x @ y.T
54         + prior_linear_regression_parameters.precision
55         @ prior_linear_regression_parameters.mean
56     )
57     return LinearRegressionParameters(
58         mean=posterior_mean, covariance=posterior_covariance
59     )
```

src/models/bayesian_linear_regression.py

The Python code for kernels:

```
1 from abc import ABC, abstractmethod
2 from dataclasses import dataclass
3
4 import jax.numpy as jnp
5 from jax import vmap
6
7
8 @dataclass
9 class KernelParameters(ABC):
10     """
11     An abstract dataclass containing the parameters for a kernel.
12     """
13
14
15 class Kernel(ABC):
16     """
17     An abstract kernel.
18     """
19
20     Parameters: KernelParameters = None
21
22     @abstractmethod
23     def _kernel(
24         self, parameters: KernelParameters, x: jnp.ndarray, y: jnp.ndarray
25     ) -> jnp.ndarray:
26         """Kernel evaluation between a single feature x and a single feature y.
27
28         Args:
29             parameters: parameters dataclass for the kernel
30             x: ndarray of shape (number_of_dimensions,)
31             y: ndarray of shape (number_of_dimensions,)
32
33         Returns:
34             The kernel evaluation. (1, 1)
35         """
36         raise NotImplementedError
37
38     def kernel(
39         self, parameters: KernelParameters, x: jnp.ndarray, y: jnp.ndarray = None
40     ) -> jnp.ndarray:
41         """Kernel evaluation for an arbitrary number of x features and y features. Compute k(x, x) if y is None.
42         This method requires the parameters dataclass and is better suited for parameter optimisation.
43
44         Args:
45             parameters: parameters dataclass for the kernel
46             x: ndarray of shape (number_of_x-features, number_of_dimensions)
47             y: ndarray of shape (number_of_y-features, number_of_dimensions)
48
49         Returns:
50             A gram matrix k(x, y), if y is None then k(x,x). (number_of_x-features, number_of_y-features)
51         """
52         # compute k(x, x) if y is None
53         if y is None:
54             y = x
55
56         # add dimension when x is 1D, assume the vector is a single feature
57         x = jnp.atleast_2d(x)
58         y = jnp.atleast_2d(y)
59
60         assert (
61             x.shape[1] == y.shape[1]
62         ), f"Dimension Mismatch: {x.shape[1]=} != {y.shape[1]=}"
63
64         return vmap(
65             lambda x_i: vmap(
66                 lambda y_i: self._kernel(parameters, x_i, y_i),
67             )(y),
68         )(x)
69
70     def __call__(
71         self, x: jnp.ndarray, y: jnp.ndarray = None, **parameter_args
72     ) -> jnp.ndarray:
73         """Kernel evaluation for an arbitrary number of x features and y features.
74         This method is more user-friendly without the need for a parameter data class.
75         It wraps the kernel computation with the initial step of constructing the parameter data class from the
76         provided parameter arguments.
77
78         Args:
79             x: ndarray of shape (number_of_x-features, number_of_dimensions)
80             y: ndarray of shape (number_of_y-features, number_of_dimensions)
81             **parameter_args: parameter arguments for the kernel
82
83         Returns:
84             A gram matrix k(x, y), if y is None then k(x,x). (number_of_x-features, number_of_y-features).
85         """
86         parameters = self.Parameters(**parameter_args)
87         return self.kernel(parameters, x, y)
88
89     def diagonal(
90         self,
91         x: jnp.ndarray,
92         y: jnp.ndarray = None,
93         **parameter_args,
94     ) -> jnp.ndarray:
```

```

95     """Kernel evaluation of only the diagonal terms of the gram matrix.
96
97     Args:
98         x: ndarray of shape (number_of_x_features, number_of_dimensions)
99         y: ndarray of shape (number_of_y_features, number_of_dimensions)
100         **parameter_args: parameter arguments for the kernel
101
102     Returns:
103         A diagonal of gram matrix k(x, y), if y is None then trace(k(x,x)).
104         (number_of_x_features, number_of_y_features)
105     """
106     # compute k(x, x) if y is None
107     if y is None:
108         y = x
109
110     # add dimension when x is 1D, assume the vector is a single feature
111     x = jnp.atleast_2d(x)
112     y = jnp.atleast_2d(y)
113
114     assert (
115         x.shape[1] == y.shape[1]
116     ), f"Dimension Mismatch: {x.shape[1]=} != {y.shape[1]=}"
117     assert (
118         x.shape[0] == y.shape[0]
119     ), f"Must have same number of features for diagonal: {x.shape[0]=} != {y.shape[0]=}"
120
121     return vmap(
122         lambda x_i, y_i: self._kernel(
123             parameters=self.Parameters(**parameter_args),
124             x=x_i,
125             y=y_i,
126         ),
127     )(x, y)
128
129     def trace(
130         self, x: jnp.ndarray, y: jnp.ndarray = None, **parameter_args
131     ) -> jnp.ndarray:
132         """Trace of the gram matrix, calculated by summation of the diagonal matrix.
133
134     Args:
135         x: ndarray of shape (number_of_x_features, number_of_dimensions)
136         y: ndarray of shape (number_of_y_features, number_of_dimensions)
137         **parameter_args: parameter arguments for the kernel
138
139     Returns:
140         The trace of the gram matrix k(x, y).
141     """
142     parameters = self.Parameters(**parameter_args)
143     return jnp.trace(self.kernel(parameters, x, y))
144
145
146 @dataclass
147 class CombinedKernelParameters(KernelParameters):
148     """
149     Parameters for the Combined Kernel:
150     """
151
152     log_theta: float
153     log_sigma: float
154     log_phi: float
155     log_eta: float
156     log_tau: float
157     log_zeta: float
158
159     @property
160     def theta(self) -> float:
161         return jnp.exp(self.log_theta)
162
163     @property
164     def sigma(self) -> float:
165         return jnp.exp(self.log_sigma)
166
167     @property
168     def phi(self) -> float:
169         return jnp.exp(self.log_phi)
170
171     @property
172     def eta(self) -> float:
173         return jnp.exp(self.log_eta)
174
175     @property
176     def tau(self) -> float:
177         return jnp.exp(self.log_tau)
178
179     @property
180     def zeta(self) -> float:
181         return jnp.exp(self.log_zeta)
182
183     @theta.setter
184     def theta(self, value: float) -> None:
185         self.log_theta = jnp.log(value)
186
187     @sigma.setter
188     def sigma(self, value: float) -> None:
189         self.log_sigma = jnp.log(value)
190

```

```

191 @phi.setter
192 def phi(self, value: float) -> None:
193     self.log_phi = jnp.log(value)
194
195 @eta.setter
196 def eta(self, value: float) -> None:
197     self.log_eta = jnp.log(value)
198
199 @tau.setter
200 def tau(self, value: float) -> None:
201     self.log_tau = jnp.log(value)
202
203 @zeta.setter
204 def zeta(self, value: float) -> None:
205     self.log_zeta = jnp.log(value)
206
207
208 class CombinedKernel(Kernel):
209     """
210     The kernel defined as:
211      $k(x, y) = \theta^2 * (\exp(-(2\sin^2(\pi(x-y)/\tau))/(\sigma^2)) + \phi^2 * \exp(-(x-y)^2/(2 * \eta^2)))$ 
212     +  $\zeta^2 * \delta(x=y)$ 
213     """
214
215     Parameters = CombinedKernelParameters
216
217     def _kernel(
218         self,
219         parameters: CombinedKernelParameters,
220         x: jnp.ndarray,
221         y: jnp.ndarray,
222     ) -> jnp.ndarray:
223         """Kernel evaluation between a single feature x and a single feature y.
224
225         Args:
226             parameters: parameters dataclass for the Gaussian kernel
227             x: ndarray of shape (1,)
228             y: ndarray of shape (1,)
229
230         Returns:
231             The kernel evaluation.
232         """
233         return jnp.dot(
234             jnp.ones(1),
235             (
236                 (parameters.theta**2)
237                 * (
238                     (
239                         jnp.exp(
240                             (-2 * jnp.sin(jnp.pi * (x - y) / parameters.tau) ** 2)
241                             / (parameters.sigma**2)
242                         )
243                     )
244                     + (parameters.phi**2)
245                     * (jnp.exp(-((x - y) ** 2) / (2 * parameters.eta**2)))
246                     + parameters.zeta**2 * (x == y)
247                 )
248             ),
249         )

```

src/models/kernels.py

The Python code for Gaussian Process Regression:

```
1 from dataclasses import dataclass
2 from typing import Any, Dict, Tuple
3
4 import jax
5 import jax.numpy as jnp
6 import optax
7 from jax import grad
8 from optax import GradientTransformation
9
10 from src.models.kernels import Kernel
11
12
13 @dataclass
14 class GaussianProcessParameters:
15     """
16     Parameters for a Gaussian Process:
17     log-sigma: logarithm of the noise parameter
18     kernel: parameters for the chosen kernel
19     """
20
21     log_sigma: float
22     kernel: Dict[str, Any]
23
24     @property
25     def variance(self) -> float:
26         return self.sigma**2
27
28     @property
29     def sigma(self) -> float:
30         return jnp.exp(self.log_sigma)
31
32     @sigma.setter
33     def sigma(self, value: float) -> None:
34         self.log_sigma = jnp.log(value)
35
36
37 class GaussianProcess:
38     """
39     A Gaussian measure defined with a kernel, better known as a Gaussian Process.
40     """
41
42     Parameters = GaussianProcessParameters
43
44     def __init__(self, kernel: Kernel, x: jnp.ndarray, y: jnp.ndarray) -> None:
45         """Initialising requires a kernel and data to condition the distribution.
46
47         Args:
48             kernel: kernel for the Gaussian Process
49             x: design matrix (number_of_features, number_of_dimensions)
50             y: response vector (number_of_features, )
51         """
52         self.number_of_train_points = x.shape[0]
53         self.x = x
54         self.y = y
55         self.kernel = kernel
56
57     def _compute_kxx_shifted_cholesky_decomposition(
58         self, parameters
59     ) -> Tuple[jnp.ndarray, bool]:
60         """
61         Cholesky decomposition of  $(k_{xx} + (1/\sigma^2)I)$ 
62
63         Args:
64             parameters: parameters dataclass for the Gaussian Process
65
66         Returns:
67             cholesky_decomposition_kxx_shifted: the cholesky decomposition (number_of_features,
68             number_of_features)
69             lower_flag: flag indicating whether the factor is in the lower or upper triangle
70         """
71         kxx = self.kernel(self.x, **parameters.kernel)
72         kxx_shifted = kxx + parameters.variance * jnp.eye(self.number_of_train_points)
73         a=kxx_shifted, lower=True
74     )
75     return kxx_shifted_cholesky_decomposition, lower_flag
76
77     def posterior_distribution(
78         self, x: jnp.ndarray, **parameter_args
79     ) -> Tuple[jnp.ndarray, jnp.ndarray]:
80         """Compute the posterior distribution for test points x.
81         Reference: http://gaussianprocess.org/gpml/chapters/RW2.pdf
82
83         Args:
84             x: test points (number_of_features, number_of_dimensions)
85             **parameter_args: parameter arguments for the Gaussian Process
86
87         Returns:
88             mean: the distribution mean (number_of_features, )
89             covariance: the distribution covariance (number_of_features, number_of_features)
90         """
91         parameters = self.Parameters(**parameter_args)
92         kxy = self.kernel(self.x, x, **parameters.kernel)
93         kyy = self.kernel(x, **parameters.kernel)
```

```

94     (
95         kxx_shifted_cholesky_decomposition,
96         lower_flag,
97     ) = self._compute_kxx_shifted_cholesky_decomposition(parameters)
98
99     mean = (
100         kxy.T
101         @ jax.scipy.linalg.cho_solve(
102             c_and_lower=(kxx_shifted_cholesky_decomposition, lower_flag), b=self.y
103         )
104     ).reshape(
105         -1,
106     )
107     covariance = kyy - kxy.T @ jax.scipy.linalg.cho_solve(
108         (kxx_shifted_cholesky_decomposition, lower_flag), kxy
109     )
110     return mean, covariance
111
112 def posterior_negative_log_likelihood(self, **parameter_args) -> jnp.float64:
113     """The negative log likelihood of the posterior distribution for the training data (x, y).
114     Reference: http://gaussianprocess.org/gpml/chapters/RW2.pdf
115
116     Args:
117         **parameter_args: parameter arguments for the Gaussian Process
118
119     Returns:
120         The negative log likelihood.
121     """
122     parameters = self.Parameters(**parameter_args)
123     (
124         kxx_shifted_cholesky_decomposition,
125         lower_flag,
126     ) = self._compute_kxx_shifted_cholesky_decomposition(parameters)
127
128     negative_log_likelihood = -(
129         -0.5
130         * (
131             self.y.T
132             @ jax.scipy.linalg.cho_solve(
133                 c_and_lower=(kxx_shifted_cholesky_decomposition, lower_flag),
134                 b=self.y,
135             )
136         )
137         - jnp.trace(jnp.log(kxx_shifted_cholesky_decomposition))
138         - (self.number_of_train_points / 2) * jnp.log(2 * jnp.pi)
139     )
140     return negative_log_likelihood
141
142 def _compute_gradient(self, **parameter_args) -> Dict[str, Any]:
143     """Calculate the gradient of the posterior negative log likelihood with respect to the parameters.
144
145     Args:
146         **parameter_args: parameter arguments for the Gaussian Process
147
148     Returns:
149         A dictionary of the gradients for each parameter argument.
150     """
151     gradients = grad(
152         lambda params: self.posterior_negative_log_likelihood(**params)
153     )(parameter_args)
154     return gradients
155
156 def train(
157     self,
158     optimizer: GradientTransformation,
159     number_of_training_iterations: int,
160     **parameter_args,
161 ) -> GaussianProcessParameters:
162     """Train the parameters for a Gaussian Process by optimising the negative log likelihood.
163
164     Args:
165         optimizer: jax optimizer object
166         number_of_training_iterations: number of iterations to perform the optimizer
167         **parameter_args: parameter arguments for the Gaussian Process
168
169     Returns:
170         A parameters dataclass containing the optimised parameters.
171     """
172     opt_state = optimizer.init(parameter_args)
173     for _ in range(number_of_training_iterations):
174         gradients = self._compute_gradient(**parameter_args)
175         updates, opt_state = optimizer.update(gradients, opt_state)
176         parameter_args = optax.apply_updates(parameter_args, updates)
177     return self.Parameters(**parameter_args)

```

src/models/gaussian_process_regression.py

The rest of the Python code for question 2:

```

1 from dataclasses import asdict, fields
2
3 import dataframe_image as dfi
4 import jax
5 import jax.numpy as jnp
6 import matplotlib.pyplot as plt
7 import numpy as np
8 import optax
9 import pandas as pd
10 import scipy
11
12 from src.models.bayesian_linear_regression import (
13     LinearRegressionParameters,
14     Theta,
15     compute_linear_regression_posterior,
16 )
17 from src.models.gaussian_process_regression import (
18     GaussianProcess,
19     GaussianProcessParameters,
20 )
21 from src.models.kernels import CombinedKernel, CombinedKernelParameters
22
23 jax.config.update("jax_enable_x64", True)
24
25
26 def construct_design_matrix(t: np.ndarray):
27     return np.stack((t, np.ones(t.shape)), axis=1).T
28
29
30 def a(
31     t: np.ndarray,
32     y: np.ndarray,
33     sigma: float,
34     prior_linear_regression_parameters: LinearRegressionParameters,
35     save_path: str,
36 ) -> LinearRegressionParameters:
37     x = construct_design_matrix(t)
38     prior_theta = Theta(
39         linear_regression_parameters=prior_linear_regression_parameters,
40         sigma=sigma,
41     )
42     posterior_linear_regression_parameters = compute_linear_regression_posterior(
43         x,
44         y,
45         prior_linear_regression_parameters,
46         residuals_precision=prior_theta.precision,
47     )
48     df_mean = pd.DataFrame(
49         posterior_linear_regression_parameters.mean, columns=["value"]
50     )
51     df_mean.index = ["a", "b"]
52     df_mean = pd.concat([df_mean], keys=["parameters"])
53     dfi.export(df_mean, save_path + "-mean.png")
54
55     df_covariance = pd.DataFrame(
56         posterior_linear_regression_parameters.covariance, columns=["a", "b"]
57     )
58     df_covariance.index = ["a", "b"]
59     df_covariance = pd.concat([df_covariance], keys=["parameters"])
60     df_covariance = pd.concat([df_covariance.T], keys=["parameters"])
61     dfi.export(df_covariance, save_path + "-covariance.png")
62     return posterior_linear_regression_parameters
63
64
65 def b(
66     t_year: np.ndarray,
67     t: np.ndarray,
68     y: np.ndarray,
69     linear_regression_parameters: LinearRegressionParameters,
70     error_mean: float,
71     error_variance: float,
72     save_path,
73 ) -> None:
74     x = construct_design_matrix(t)
75     residuals = y - linear_regression_parameters.predict(x)
76     plt.plot(t_year.reshape(-1), residuals.reshape(-1))
77     plt.xlabel("date (decimal year)")
78     plt.ylabel("residual")
79     plt.title("2b: g_obs(t)")
80     plt.savefig(save_path + "-residuals-timeseries")
81     plt.close()
82
83     count, bins = np.histogram(residuals, bins=100, density=True)
84     plt.bar(bins[1:], count, label="residuals")
85     plt.plot(
86         bins[1:],
87         scipy.stats.norm.pdf(bins[1:], loc=error_mean, scale=error_variance),
88         color="red",
89         label="e(t)",
90     )
91     plt.xlabel("residual bin")
92     plt.ylabel("density")
93     plt.title("2b: Residuals Density")
94     plt.legend()

```

```

95 plt.savefig(save_path + "-residuals-density-estimation")
96 plt.close()
97
98
99 def c(
100     kernel: CombinedKernel,
101     kernel_parameters: CombinedKernelParameters,
102     log_theta_range: np.ndarray,
103     t: np.ndarray,
104     number_of_samples: int,
105     save_path: str,
106 ) -> None:
107     gram = kernel(t, **asdict(kernel_parameters))
108     plt.imshow(gram)
109     plt.xlabel("t")
110     plt.ylabel("t")
111     plt.title("Gram Matrix (Prior)")
112     plt.savefig(save_path + "-gram-matrix")
113     plt.close()
114
115     for _ in range(number_of_samples):
116         plt.plot(
117             np.random.multivariate_normal(
118                 jnp.zeros(gram.shape[0]), gram, size=1
119             ).reshape(-1)
120         )
121     plt.xlabel("t")
122     plt.ylabel("f.GP(t)")
123     plt.title("Samples from Gaussian Process Prior")
124     plt.savefig(save_path + "-samples")
125     plt.close()
126
127     fig_samples, ax_samples = plt.subplots(
128         len(fields(kernel_parameters._.class_)),
129         len(log_theta_range),
130         figsize=(
131             len(log_theta_range) * 2,
132             len(fields(kernel_parameters._.class_)) * 2,
133         ),
134         frameon=False,
135     )
136     for i, field in enumerate(fields(kernel_parameters._.class_)):
137         default_value = getattr(kernel_parameters, field.name)
138         for j, log_value in enumerate(log_theta_range):
139             setattr(kernel_parameters, field.name, log_value)
140             gram = kernel(t, **asdict(kernel_parameters))
141             ax_samples[i][j].plot(
142                 np.random.multivariate_normal(
143                     jnp.zeros(gram.shape[0]), gram, size=1
144                 ).reshape(-1),
145             )
146             ax_samples[i][j].set_title(
147                 f"{field.name.strip('log_')}={np.round(np.exp(log_value), 2)}"
148             )
149             setattr(kernel_parameters, field.name, default_value)
150     plt.tight_layout()
151     plt.savefig(save_path + f"-parameter-samples", bbox_inches="tight")
152     plt.close(fig_samples)
153
154     fig_gram, ax_gram = plt.subplots(
155         len(fields(kernel_parameters._.class_)),
156         len(log_theta_range),
157         figsize=(
158             len(log_theta_range) * 2,
159             len(fields(kernel_parameters._.class_)) * 2,
160         ),
161         frameon=False,
162     )
163     for i, field in enumerate(fields(kernel_parameters._.class_)):
164         default_value = getattr(kernel_parameters, field.name)
165         for j, log_value in enumerate(log_theta_range):
166             setattr(kernel_parameters, field.name, log_value)
167             gram = kernel(t, **asdict(kernel_parameters))
168             ax_gram[i][j].imshow(gram)
169             ax_gram[i][j].set_title(
170                 f"{field.name.strip('log_')}={np.round(np.exp(log_value), 2)}"
171             )
172             setattr(kernel_parameters, field.name, default_value)
173     plt.tight_layout()
174     plt.savefig(save_path + f"-parameter-grams", bbox_inches="tight")
175     plt.close(fig_gram)
176
177
178 def f(
179     t_train: np.ndarray,
180     y_train: np.ndarray,
181     t_test: np.ndarray,
182     min_year: float,
183     prior.linear_regression_parameters: LinearRegressionParameters,
184     linear_regression_sigma: float,
185     kernel: CombinedKernel,
186     gaussian_process_parameters: GaussianProcessParameters,
187     learning_rate: float,
188     number_of_iterations: int,
189     save_path: str,
190 ) -> None:

```

```

191 # Train Bayesian Linear Regression
192 x_train = construct_design_matrix(t_train)
193 prior_theta = Theta(
194     linear_regression_parameters=prior_linear_regression_parameters,
195     sigma=linear_regression_sigma,
196 )
197 posterior_linear_regression_parameters = compute_linear_regression_posterior(
198     x_train,
199     y_train,
200     prior_linear_regression_parameters,
201     residuals_precision=prior_theta.precision,
202 )
203
204 residuals = y_train - posterior_linear_regression_parameters.predict(x_train)
205 gaussian_process = GaussianProcess(
206     kernel, t_train.reshape(-1, 1), residuals.reshape(-1)
207 )
208
209 # Prediction
210 x_test = construct_design_matrix(t_test)
211 linear_prediction = posterior_linear_regression_parameters.predict(x_test).reshape(
212     -1
213 )
214 mean_prediction, covariance_prediction = gaussian_process.posterior_distribution(
215     t_test.reshape(-1, 1), **asdict(gaussian_process_parameters)
216 )
217
218 # Plot
219 plt.figure(figsize=(7, 7))
220 plt.scatter(
221     t_train + min_year,
222     y_train.reshape(-1),
223     s=2,
224     color="blue",
225     label="historical data",
226 )
227 plt.plot(
228     t_test + min_year,
229     linear_prediction + mean_prediction,
230     color="gray",
231     label="prediction",
232 )
233 plt.fill_between(
234     t_test + min_year,
235     linear_prediction
236     + mean_prediction
237     - 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
238     linear_prediction
239     + mean_prediction
240     + 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
241     facecolor=(0.8, 0.8, 0.8),
242     label="error bound (one stdev)",
243 )
244 plt.xlabel("date (decimal year)")
245 plt.ylabel("parts per million")
246 plt.title("Global Mean CO2 Concentration Prediction (Untrained Hyperparameters)")
247 plt.legend()
248 plt.tight_layout()
249 plt.savefig(save_path + "-extrapolation-untrained", bbox_inches="tight")
250 plt.close()
251
252 df_parameters = pd.DataFrame(
253     [
254         [
255             x.strip("log-") + " (kernel)",
256             np.exp(gaussian_process_parameters.kernel[x]),
257         ]
258         for x in gaussian_process_parameters.kernel.keys()
259     ]
260     + [{"sigma", float(gaussian_process_parameters.sigma)}],
261     columns=["parameter", "value"],
262 )
263 df_parameters = df_parameters.set_index("parameter").sort_values(by=["parameter"])
264 dfi.export(df_parameters, save_path + "-untrained-parameters.png")
265
266 # Train Gaussian Process Regression (Hyperparameter Tune)
267 optimizer = optax.adam(learning_rate)
268 gaussian_process_parameters = gaussian_process.train(
269     optimizer, number_of_iterations, **asdict(gaussian_process_parameters)
270 )
271 df_parameters = pd.DataFrame(
272     [
273         [
274             x.strip("log-") + " (kernel)",
275             np.exp(gaussian_process_parameters.kernel[x]),
276         ]
277         for x in gaussian_process_parameters.kernel.keys()
278     ]
279     + [{"sigma", float(gaussian_process_parameters.sigma)}],
280     columns=["parameter", "value"],
281 )
282 df_parameters = df_parameters.set_index("parameter").sort_values(by=["parameter"])
283 dfi.export(df_parameters, save_path + "-trained-parameters.png")
284
285 # Prediction
286 x_test = construct_design_matrix(t_test)

```

```

287 linear_prediction = posterior_linear_regression_parameters.predict(x_test).reshape(
288     -1
289 )
290 mean_prediction, covariance_prediction = gaussian_process.posterior_distribution(
291     t_test.reshape(-1, 1), **asdict(gaussian_process_parameters)
292 )
293
294 # Plot
295 plt.figure(figsize=(7, 7))
296 plt.scatter(
297     t_train + min_year,
298     y_train.reshape(-1),
299     s=2,
300     color="blue",
301     label="historical data",
302 )
303 plt.plot(
304     t_test + min_year,
305     linear_prediction + mean_prediction,
306     color="gray",
307     label="prediction",
308 )
309 plt.fill_between(
310     t_test + min_year,
311     linear_prediction
312     + mean_prediction
313     - 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
314     linear_prediction
315     + mean_prediction
316     + 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
317     facecolor=(0.8, 0.8, 0.8),
318     label="error bound (one stdev)",
319 )
320 plt.xlabel("date (decimal year)")
321 plt.ylabel("parts per million")
322 plt.title("Global Mean CO2 Concentration Prediction (Trained Hyperparameters)")
323 plt.legend()
324 plt.tight_layout()
325 plt.savefig(save_path + "-extrapolation-trained", bbox_inches="tight")
326 plt.close()

```

src/solutions/q2.py

Question 3

(a)

The free energy is can be calculated as:

$$\mathcal{F}(q, \theta) = \langle \log P(\mathbf{x}, \mathbf{s}|\theta) \rangle_{q(\mathbf{s})} + H[Q(\mathbf{s})]$$

Knowing,

$$\log P(\mathbf{x}, \mathbf{s}|\theta) = \log P(\mathbf{x}|\mathbf{s}, \theta) + \log P(\mathbf{s}|\theta)$$

we can write:

$$\mathcal{F}(Q, \theta) = \langle \log P(\mathbf{x}|\mathbf{s}, \theta) \rangle_{q(\mathbf{s})} + \langle \log P(\mathbf{s}|\theta) \rangle_{q(\mathbf{s})} + H[q(\mathbf{s})]$$

Moreover, our mean field approximation:

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

where $q_i(s_i) = \lambda_i^{s_i} (1 - \lambda_i)^{(1-s_i)}$.

To compute the first term:

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

substituting the appropriate terms:

$$P(\mathbf{x}|\mathbf{s}, \theta) = 2\pi^{-\frac{d}{2}} |\sigma^2 \mathbf{I}|^{-\frac{1}{2}} \exp \left(-\frac{1}{2} \left(\mathbf{x} - \sum_{i=1}^K s_i \mu_i \right)^T \frac{1}{\sigma^2} \mathbf{I} \left(\mathbf{x} - \sum_{i=1}^K s_i \mu_i \right) \right)$$

with d being the number of dimensions.

Taking the logarithm:

$$\log P(\mathbf{x}|\mathbf{s}, \theta) = -\frac{d}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{i=1}^K s_i \mu_i + \sum_{i=1}^K \sum_{j=1}^K s_i s_j \mu_i^T \mu_j \right)$$

The expectation distributed to the relevant terms:

$$\langle \log P(\mathbf{x}|\mathbf{s}, \theta) \rangle_{q(\mathbf{s})} = -\frac{d}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{i=1}^K \langle s_i \rangle_{q_i(s_i)} \mu_i + \sum_{i=1}^K \sum_{j=1}^K \langle s_i s_j \rangle_{q_i(s_i) q_j(s_j)} \mu_i^T \mu_j \right)$$

Evaluating the expectations:

$$\langle \log P(\mathbf{x}|\mathbf{s}, \theta) \rangle_{q(\mathbf{s})} = -\frac{d}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{i=1}^K \lambda_i \mu_i + \sum_{i=1}^K \sum_{j=1, j \neq i}^K \lambda_i \lambda_j \mu_i^T \mu_j + \sum_{i=1}^K \lambda_i \mu_i^T \mu_i \right)$$

where $\langle s_i s_j \rangle_{q_i(s_i)} = \langle s_i \rangle_{q_i(s_i)}$ because $s_i \in \{0, 1\}$.

To compute the second term:

$$P(\mathbf{s}|\theta) = \prod_{i=1}^K \pi_i^{s_i} (1 - \pi_i)^{(1-s_i)}$$

Taking the logarithm:

$$\log P(\mathbf{s}|\theta) = \sum_{i=1}^K s_i \log \pi_i + (1 - s_i) \log(1 - \pi_i)$$

The expectation distributed to the relevant terms:

$$\langle \log P(\mathbf{s}|\theta) \rangle_{q(\mathbf{s})} = \sum_{i=1}^K \langle s_i \rangle_{q_i(s_i)} \log \pi_i + (1 - \langle s_i \rangle_{q_i(s_i)}) \log(1 - \pi_i)$$

Evaluating the expectations:

$$\langle \log P(\mathbf{s}|\theta) \rangle_{q(\mathbf{s})} = \sum_{i=1}^K \lambda_i \log \pi_i + (1 - \lambda_i) \log(1 - \pi_i)$$

To compute the third term, we use the mean field factorisation:

$$H[q(\mathbf{s})] = \sum_{i=1}^K H[q_i(s_i)]$$

Thus,

$$H[q(\mathbf{s})] = - \sum_{i=1}^K \sum_{s_i \in \{0,1\}} q_i(s_i) \log q_i(s_i)$$

Substituting the appropriate values:

$$H[q(\mathbf{s})] = - \sum_{i=1}^K \lambda_i \log \lambda_i + (1 - \lambda_i) \log(1 - \lambda_i)$$

Combining, we have our free energy expression:

$$\begin{aligned} \mathcal{F}(q, \theta) = & \frac{-d}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{i=1}^K \lambda_i \mu_i + \sum_{i=1}^K \sum_{j=1, j \neq i}^K \lambda_i \lambda_j \mu_i^T \mu_j + \sum_{i=1}^K \lambda_i \mu_i^T \mu_i \right) \\ & + \sum_{i=1}^K \lambda_i \log \pi_i + (1 - \lambda_i) \log(1 - \pi_i) \\ & - \sum_{i=1}^K \lambda_i \log \lambda_i + (1 - \lambda_i) \log(1 - \lambda_i) \end{aligned}$$

To derive the partial update for $q_i(s_i)$ we take the variational derivative of the Lagrangian, enforcing the normalisation of q_i :

$$\frac{\partial}{\partial q_i} \left(\mathcal{F}(q, \theta) + \lambda^{LG} \int q_i - 1 \right) = \langle \log P(\mathbf{x}, \mathbf{s}|\theta) \rangle_{\prod_{j \neq i} q_j(s_j)} - \log q_i(s_i) - 1 + \lambda^{LG}$$

where λ^{LG} is the Lagrange multiplier.

Setting this to zero we can solve for the λ_i that maximises the free energy:

$$\log q_i(s_i) = \langle \log P(\mathbf{x}, \mathbf{s} | \theta) \rangle_{\prod_{j \neq i} q_j(s_j)} - 1 + \lambda^{LG}$$

Similar to our free energy derivation:

$$\langle \log P(\mathbf{x} | \mathbf{s}, \theta) \rangle_{\prod_{j \neq i} q_j(s_j)} \propto -\frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{k=1}^K \langle s_k \rangle_{\prod_{j \neq i} q_j(s_j)} \mu_k + \sum_{k=1}^K \sum_{j=1}^K \langle s_k s_j \rangle_{\prod_{j \neq i} q_j(s_j)} \right)$$

and

$$\langle \log P(\mathbf{s} | \theta) \rangle_{\prod_{j \neq i} q_j(s_j)} = \sum_{k=1}^K \langle s_k \rangle_{\prod_{j \neq i} q_j(s_j)} \log \pi_k + (1 - \langle s_k \rangle_{\prod_{j \neq i} q_j(s_j)}) \log(1 - \pi_k)$$

We can write:

$$\log q_i(s_i) \propto \log P(\mathbf{x} | \mathbf{s}, \theta)_{\prod_{j \neq i} q_j(s_j)} + \langle \log P(\mathbf{s} | \theta) \rangle_{\prod_{j \neq i} q_j(s_j)}$$

Substituting the relevant terms:

$$\log q_i(s_i) \propto -\frac{1}{2\sigma^2} \left(-2s_i \mathbf{x}^T \mu_i + s_i s_i \mu_i^T \mu_i + 2 \sum_{j=1, j \neq i}^K s_i \lambda_j \mu_i^T \mu_j \right) + s_i \log \pi_i + (1 - s_i) \log(1 - \pi_i)$$

Knowing $\log q_i(s_i) = s_i \log \lambda_i + (1 - s_i) \log(1 - \lambda_i)$:

$$\log q_i(s_i) \propto s_i \log \frac{\lambda_i}{1 - \lambda_i}$$

Thus,

$$s_i \log \frac{\lambda_i}{1 - \lambda_i} \propto -\frac{1}{2\sigma^2} \left(-2s_i \mathbf{x}^T \mu_i + s_i s_i \mu_i^T \mu_i + 2 \sum_{j=1, j \neq i}^K s_i \lambda_j \mu_i^T \mu_j \right) + s_i \log \frac{\pi_i}{1 - \pi_i}$$

Also, because $s_i \in \{0, 1\}$ we know that $s_i^2 = s_i$:

$$s_i \log \frac{\lambda_i}{1 - \lambda_i} \propto -\frac{1}{2\sigma^2} \left(-2s_i \mathbf{x}^T \mu_i + s_i \mu_i^T \mu_i + 2 \sum_{j=1, j \neq i}^K s_i \lambda_j \mu_i^T \mu_j \right) + s_i \log \frac{\pi_i}{1 - \pi_i}$$

Because we have only kept terms with s_i , this is an equality:

$$s_i \log \frac{\lambda_i}{1 - \lambda_i} = \frac{s_i \mu_i^T}{2\sigma^2} \left(2\mathbf{x} - \mu_i - 2 \sum_{j=1, j \neq i}^K \lambda_j \mu_j \right) + s_i \log \frac{\pi_i}{1 - \pi_i}$$

Solving for λ_i :

$$\lambda_i = \frac{1}{1 + \exp \left[- \left(\frac{\mu_i^T}{\sigma^2} \left(\mathbf{x} - \frac{\mu_i}{2} - \sum_{j=1, j \neq i}^K \lambda_j \mu_j \right) + \log \frac{\pi_i}{1 - \pi_i} \right) \right]}$$

we have our partial update.

(b)

The provided derivations for the M step of the mean parameter μ :

$$\mu = \left(\langle \mathbf{s}\mathbf{s}^T \rangle_{q(\mathbf{s})} \right)^{-1} \langle \mathbf{s} \rangle_{q(\mathbf{s})} \mathbf{x}$$

where $\mu \in \mathbb{R}^{K \times D}$, $\mathbf{s} \in \mathbb{R}^{K \times N}$, and $\mathbf{x} \in \mathbb{R}^{N \times D}$.

This mimics the least squares solution:

$$\hat{\beta} = (\mathbf{X}\mathbf{X}^T)\mathbf{X}\mathbf{Y}$$

for the linear regression problem $\mathbf{Y} = \mathbf{X}^T\beta$ where β corresponds to the mean parameters μ , the design matrix \mathbf{X} corresponds to the input \mathbf{s} and the response Y corresponds to the image pixels denoted \mathbf{x} . This makes sense because our resulting images \mathbf{x} are modeled as linear combinations of features μ , weighted by \mathbf{s} .

(c)

The computational complexity of the implemented M step function can be broken down for each parameter:

- μ :
 - The inversion ESS^{-1} where $\text{ESS} \in \mathbb{R}^{K \times K}$ is $\mathcal{O}(K^3)$
 - The dot product $\text{ESS}^{-1}\text{ES}^T$ where $\text{ESS}^{-1} \in \mathbb{R}^{K \times K}$ and $\text{ES} \in \mathbb{R}^{N \times K}$ is $\mathcal{O}(K^2N)$
 - The dot product $(\text{ESS}^{-1}\text{ES}^T)\mathbf{x}$ where $(\text{ESS}^{-1}\text{ES}^T) \in \mathbb{R}^{K \times N}$ and $\mathbf{x} \in \mathbb{R}^{N \times D}$ is $\mathcal{O}(KND)$
- σ :
 - The dot product $(\mathbf{x}^T\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^{N \times D}$ is $\mathcal{O}(D^2N)$
 - The dot product $\mu^T\mu$ where $\mu \in \mathbb{R}^{D \times K}$ is $\mathcal{O}(K^2D)$
 - The dot product $(\mu^T\mu)\text{ESS}$ where $\mu^T\mu \in \mathbb{R}^{K \times K}$ and $\text{ESS} \in \mathbb{R}^{K \times K}$ is $\mathcal{O}(K^3)$
- π :
 - The mean operation for $\text{ES} \in \mathbb{R}^{N \times K}$ along the first dimension is $\mathcal{O}(NK)$

Thus, the computational complexity of the M step is $\mathcal{O}(K^3 + K^2N + KND + D^2N + K^2D)$ where we do not assume that any of N , K , or D is large compared to the others.

(d)

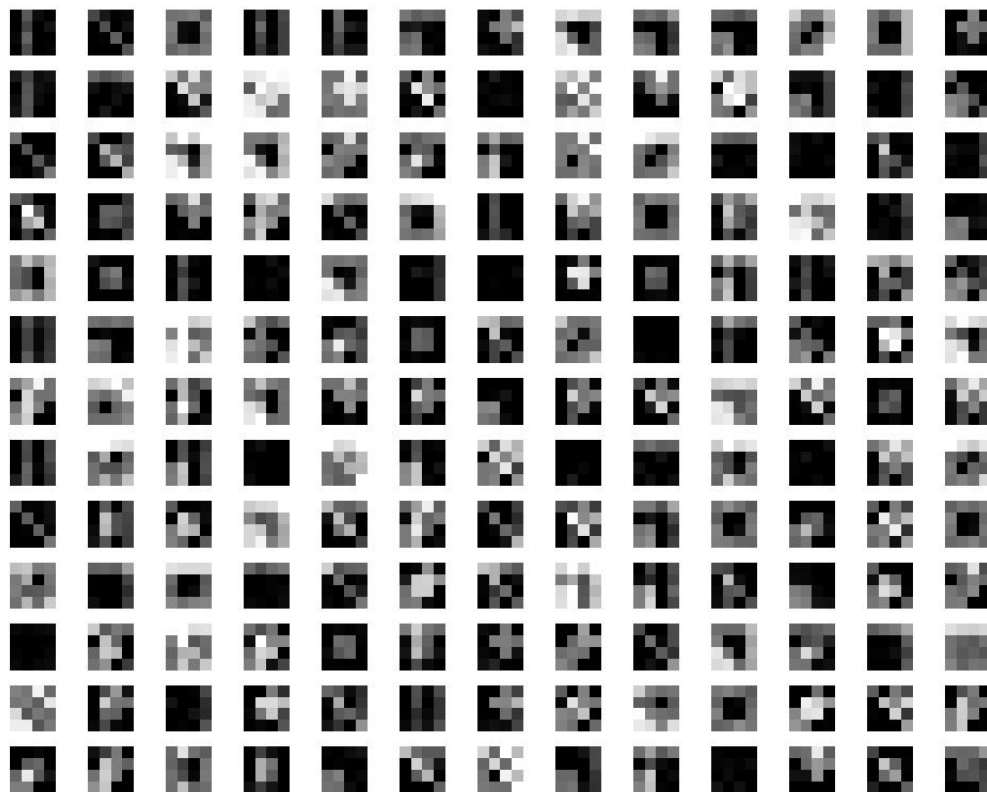


Figure 13: Images generated by randomly combined features with noise

Examining the generated images, we can see eight features:

- (1) a cross
- (2) a border
- (3) a two by two square in the middle
- (4) a two by two square in the bottom left corner
- (5) a diagonal from top left to bottom right
- (6) a vertical line in the second column
- (7) a vertical line in the fourth column
- (8) a horizontal line in the first row

Factor analysis assumes a model:

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

where $\epsilon \sim \mathcal{N}(\mu_\epsilon, \Sigma_\epsilon)$ and $\mathbf{s} \sim \mathcal{N}(\mu_{\mathbf{s}}, \Sigma_{\mathbf{s}})$. Factor analysis would be inappropriate for this data because the our latent variables are binary (i.e. whether or not a feature is present) and not Gaussians. Moreover, the presence of each feature is independent of the presence of another which is not enforced in this model with a covariance matrix that might not be diagonal.

A mixture of Gaussians assumes as model:

$$\mathbf{x} = \sum_{k=1}^K \pi_k \mu_k + \epsilon$$

where $\epsilon \sim \mathcal{N}(0, \Sigma_\epsilon)$. This also wouldn't be appropriate because each mixture component (feature) is assumed to have some covariance, whereas our mixtures are defined as binary vectors (a cross, a border, etc) and added together before adding some noise.

The independent component analysis assumes a model:

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

where $\epsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$ and $p(\mathbf{s}) = \prod_{k=1}^K p(s_k)$. This is appropriate for our data because we are linearly combining different features and then adding noise.

Thus, it would be expected that ICA does a good job modelling this data while factor analysis and mixture of Gaussians would not.

(e)

We can plot the free energy to make sure it increases each iteration:

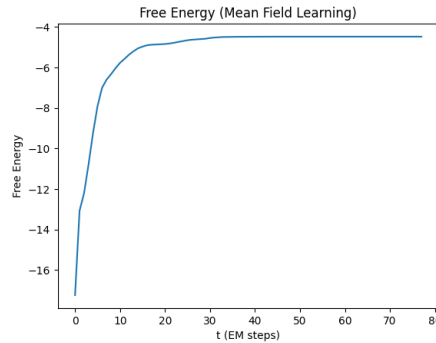


Figure 14: Free Energy

(f)

The initialised features:

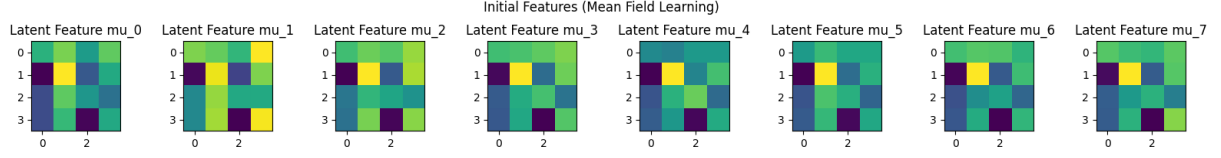


Figure 15: Initial Latent Factors

The features learned by the algorithm:

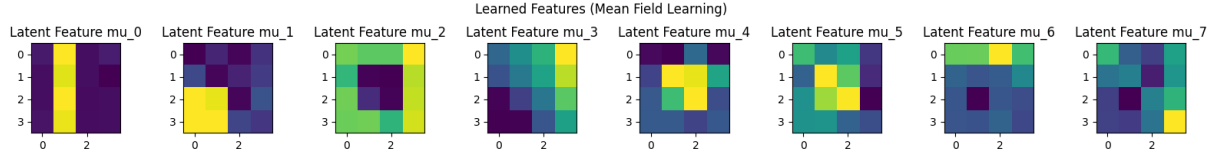


Figure 16: Learned Latent Factors

We can see that it has learned some of previously identified features, such as the vertical line in the second column, the two by two square in the bottom left corner, the border, and the a two by two square in the middle. The other features seem to be some linear combination of two or more features, such as μ_4 which looks like a combination of the cross and two by two square in the middle.

A possible way to improve our algorithm is reinitialising our algorithm a few times to find better potential convergence results (i.e. choose model with best free energy). Another way to improve the algorithm could be to increase the K , although it may learn some duplicate features, there is also a higher chance of capturing all the features. We can visualise this:

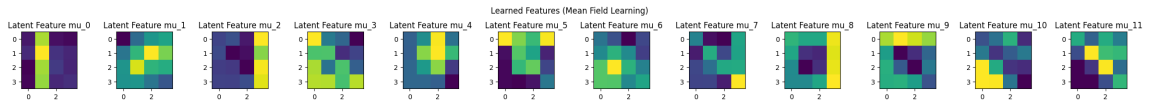


Figure 17: Increasing Number of Latent Factors

Here we can identify a few more features such as the vertical line in the fourth column the cross, and some of the diagonal feature in μ_7 .

When implementing the algorithm, the mean field parameters were initialised randomly, each independently from a uniform distribution. However π , σ , and μ by running the maximisation step using the randomly initialised mean field parameters. K was set to eight, after visually identifying eight features in part d.

(g)

Plotting the free energy at each partial expectation step of the variational approximation for different σ 's:

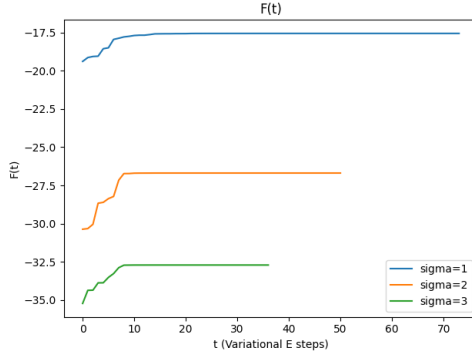


Figure 18: Free energy vs σ

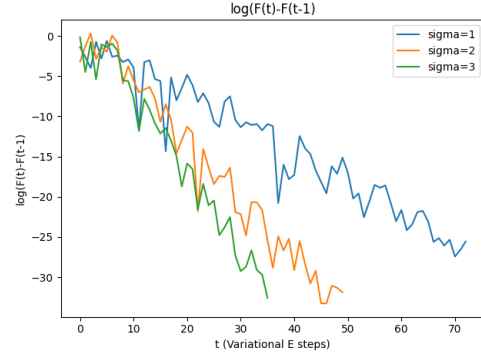


Figure 19: Free energy convergence vs σ

We know that our free energy is a upper bounded on the log likelihood:

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

In the variational expectation step, $\log P(\mathcal{X}|\theta)$ is fixed and we adjust our approximation q to reach this upper bound. We know that σ quantifies the noise of \mathbf{x} , thus a higher σ means a wider spread in our distribution $\log P(\mathcal{X}|\theta)$, meaning we are reducing our upper bound for $\mathcal{F}(q, \theta)$. As such, we can see in the plot for free energy above that when σ is increased, our free energy converges to a lower value, due to being bounded above by a lower log-likelihood. Moreover, by reducing the upper bound, we see in the plot of $\log(F(t) - F(t - 1))$ that our free energy is able to converge faster. Because we have reduced the upper bound by increasing σ , our free energy can reach this upper bound faster.

The Python code for the binary latent factor model:

```

1  from __future__ import annotations
2
3  from typing import TYPE_CHECKING, Tuple
4
5  import numpy as np
6
7  from demo_code.MStep import m_step
8  from src.models.binary_latent_factor_models.abstract_binary_latent_factor_model import (
9      AbstractBinaryLatentFactorModel,
10 )
11
12 if TYPE_CHECKING:
13     from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
14         AbstractBinaryLatentFactorApproximation,
15     )
16
17
18 class BinaryLatentFactorModel(AbstractBinaryLatentFactorModel):
19     """
20     mu: matrix of means (number_of_dimensions, number_of_latent_variables)
21     sigma: gaussian noise parameter
22     pi: vector of priors (1, number_of_latent_variables)
23     """
24
25     def __init__(
26         self,
27         mu: np.ndarray,
28         sigma: float,
29         pi: np.ndarray,
30     ):
31         self._mu = mu
32         self._sigma = sigma
33         self._pi = pi
34
35     @property
36     def mu(self):
37         return self._mu
38
39     @mu.setter
40     def mu(self, value):
41         self._mu = value
42
43     @property
44     def sigma(self):
45         return self._sigma
46
47     @sigma.setter
48     def sigma(self, value):
49         self._sigma = value
50
51     @property
52     def pi(self):
53         return self._pi
54
55     @pi.setter
56     def pi(self, value):
57         self._pi = value
58
59     @property
60     def variance(self) -> float:
61         return self.sigma**2
62
63     @staticmethod
64     def calculate_maximisation_parameters(
65         x: np.ndarray,
66         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
67     ) -> Tuple[np.ndarray, float, np.ndarray]:
68         return m_step(
69             x=x,
70             es=binary_latent_factor_approximation.expectation_s,
71             ess=binary_latent_factor_approximation.expectation_ss,
72         )
73
74     def maximisation_step(
75         self,
76         x: np.ndarray,
77         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
78     ) -> None:
79         mu, sigma, pi = self.calculate_maximisation_parameters(
80             x, binary_latent_factor_approximation
81         )
82         self.mu = mu
83         self.sigma = sigma
84         self.pi = pi
85
86
87 def init_binary_latent_factor_model(
88     x: np.ndarray,
89     binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
90 ) -> BinaryLatentFactorModel:
91     mu, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
92         x, binary_latent_factor_approximation
93     )
94     return BinaryLatentFactorModel(mu, sigma, pi)

```

src/models/binary_latent_factor_models/binary_latent_factor_model.py

The Python code for mean field learning:

```

1 from typing import List
2
3 import numpy as np
4
5 from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
6     AbstractBinaryLatentFactorApproximation,
7 )
8 from src.models.binary_latent_factor_models.binary_latent_factor_model import (
9     AbstractBinaryLatentFactorModel,
10 )
11
12
13 class MeanFieldApproximation(AbstractBinaryLatentFactorApproximation):
14     """
15     lambda_matrix: parameters variational approximation (number_of_points, number_of_latent_variables)
16     """
17
18     _lambda_matrix: np.ndarray
19
20     def __init__(self, lambda_matrix, max_steps, convergence_criterion):
21         self.lambda_matrix = lambda_matrix
22         self.max_steps = max_steps
23         self.convergence_criterion = convergence_criterion
24
25     @property
26     def lambda_matrix(self) -> np.ndarray:
27         return self._lambda_matrix
28
29     @lambda_matrix.setter
30     def lambda_matrix(self, value):
31         self._lambda_matrix = value
32
33     def lambda_matrix_exclude(self, exclude_latent_index: int) -> np.ndarray:
34         # (number_of_points, number_of_latent_variables-1)
35         return np.concatenate(
36             (
37                 self.lambda_matrix[:, :exclude_latent_index],
38                 self.lambda_matrix[:, exclude_latent_index + 1 :],
39             ),
40             axis=1,
41         )
42
43     def _partial_expectation_step(
44         self,
45         x: np.ndarray,
46         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
47         latent_factor: int,
48     ) -> np.ndarray:
49         """ Partial Variational E step for factor i for all data points
50
51         :param x: data matrix (number_of_points, number_of_dimensions)
52         :param binary_latent_factor_model: a binary_latent_factor_model
53         :param latent_factor: latent factor to compute partial update
54         :return: lambda_vector: new lambda parameters for the latent factor (number_of_points, 1)
55         """
56         lambda_matrix_excluded = self.lambda_matrix_exclude(latent_factor)
57         mu_excluded = binary_latent_factor_model.mu_exclude(latent_factor)
58
59         mu_latent = binary_latent_factor_model.mu[:, latent_factor]
60         # (number_of_points, 1)
61         partial_expectation_log_p_x_given_s_theta_proportion = (
62             binary_latent_factor_model.precision
63             * (
64                 x # (number_of_points, number_of_dimensions)
65                 - 0.5 * mu_latent.T # (1, number_of_dimensions)
66                 - lambda_matrix_excluded # (number_of_points, number_of_latent_variables-1)
67                 @ mu_excluded.T # (number_of_latent_variables-1, number_of_dimensions)
68             )
69             @ mu_latent # (number_of_dimensions, 1)
70         )
71
72         # (1, 1)
73         partial_expectation_log_p_s_given_theta_proportion = np.log(
74             binary_latent_factor_model.pi[0, latent_factor]
75             / (1 - binary_latent_factor_model.pi[0, latent_factor])
76         )
77
78         # (number_of_points, 1)
79         partial_expectation_log_p_x_s_given_theta_proportion = (
80             partial_expectation_log_p_x_given_s_theta_proportion
81             + partial_expectation_log_p_s_given_theta_proportion
82         )
83
84         # (number_of_points, 1)
85         lambda_vector = 1 / (
86             1 + np.exp(-partial_expectation_log_p_x_s_given_theta_proportion)
87         )
88         lambda_vector[lambda_vector == 0] = 1e-10
89         lambda_vector[lambda_vector == 1] = 1 - 1e-10
90         return lambda_vector
91
92     def variational_expectation_step(
93         self, x: np.ndarray, binary_latent_factor_model: AbstractBinaryLatentFactorModel
94     ) -> List[float]:

```

```

95     """Variational E step
96
97     :param binary_latent_factor_model: a binary_latent_factor_model
98     :param x: data matrix (number_of_points, number_of_dimensions)
99     """
100     free_energy = [self.compute_free_energy(x, binary_latent_factor_model)]
101     for i in range(self.max_steps):
102         for latent_factor in range(binary_latent_factor_model.k):
103             self.lambda_matrix[:, latent_factor] = self._partial_expectation_step(
104                 x, binary_latent_factor_model, latent_factor
105             )
106             free_energy.append(
107                 self.compute_free_energy(x, binary_latent_factor_model)
108             )
109             if free_energy[-1] - free_energy[-2] <= self.convergence_criterion:
110                 break
111         if free_energy[-1] - free_energy[-2] <= self.convergence_criterion:
112             break
113     return free_energy
114
115
116 def init_mean_field_approximation(
117     k: int, n: int, max_steps, convergence_criterion
118 ) -> MeanFieldApproximation:
119     return MeanFieldApproximation(
120         lambda_matrix=np.random.random(size=(n, k)),
121         max_steps=max_steps,
122         convergence_criterion=convergence_criterion,
123     )

```

src/models/binary_latent_factor_approximations/mean_field_approximation.py

The Python code for expectation maximisation:

```

1 from __future__ import annotations
2
3 from typing import TYPE_CHECKING, List, Tuple
4
5 import numpy as np
6
7 if TYPE_CHECKING:
8     from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
9         AbstractBinaryLatentFactorApproximation,
10     )
11     from src.models.binary_latent_factor_models.binary_latent_factor_model import (
12         AbstractBinaryLatentFactorModel,
13     )
14
15
16 def is_converge(
17     free_energies: List[float],
18     current_lambda_matrix: np.ndarray,
19     previous_lambda_matrix: np.ndarray,
20 ) -> bool:
21     return (abs(free_energies[-1] - free_energies[-2]) == 0) and np.linalg.norm(
22         current_lambda_matrix - previous_lambda_matrix
23     ) == 0
24
25
26 def learn_binary_factors(
27     x: np.ndarray,
28     em_iterations: int,
29     binary_latent_factor_model: AbstractBinaryLatentFactorModel,
30     binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
31 ) -> Tuple[
32     AbstractBinaryLatentFactorApproximation,
33     AbstractBinaryLatentFactorModel,
34     List[float],
35 ]:
36     free_energies: List[float] = [
37         binary_latent_factor_approximation.compute_free_energy(
38             x, binary_latent_factor_model
39         )
40     ]
41     for _ in range(em_iterations):
42         previous_lambda_matrix = np.copy(
43             binary_latent_factor_approximation.lambda_matrix
44         )
45         binary_latent_factor_approximation.variational_expectation_step(
46             x=x,
47             binary_latent_factor_model=binary_latent_factor_model,
48         )
49         binary_latent_factor_model.maximisation_step(
50             x,
51             binary_latent_factor_approximation,
52         )
53         free_energies.append(
54             binary_latent_factor_approximation.compute_free_energy(
55                 x, binary_latent_factor_model
56             )
57         )
58         if is_converge(
59             free_energies,
60             binary_latent_factor_approximation.lambda_matrix,
61             previous_lambda_matrix,
62         ):
63             break
64     return binary_latent_factor_approximation, binary_latent_factor_model, free_energies

```

src/expectation_maximisation.py

The rest of the Python code for question 3:

```

1 from typing import List
2
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 from src.expectation_maximisation import is_converge, learn_binary_factors
7 from src.models.binary_latent_factor_approximations.mean_field_approximation import (
8     init_mean_field_approximation,
9 )
10 from src.models.binary_latent_factor_models.binary_latent_factor_model import (
11     AbstractBinaryLatentFactorModel,
12     init_binary_latent_factor_model,
13 )
14
15
16 def e_and_f(
17     x: np.ndarray,
18     k: int,
19     em_iterations: int,
20     e_maximum_steps: int,
21     e_convergence_criterion: float,
22     save_path: str,
23 ) -> AbstractBinaryLatentFactorModel:
24     n = x.shape[0]
25     mean_field_approximation = init_mean_field_approximation(
26         k, n, max_steps=e_maximum_steps, convergence_criterion=e_convergence_criterion
27     )
28     binary_latent_factor_model = init_binary_latent_factor_model(
29         x, mean_field_approximation
30     )
31     fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
32     for i in range(k):
33         ax[i].imshow(binary_latent_factor_model.mu[:, i].reshape(4, 4))
34         ax[i].set_title(f"Latent Feature mu-{{i}}")
35     fig.suptitle("Initial Features (Mean Field Learning)")
36     plt.tight_layout()
37     plt.savefig(save_path + "-init-latent-factors", bbox_inches="tight")
38     plt.close()
39     _, binary_latent_factor_model, free_energy = learn_binary_factors(
40         x,
41         em_iterations,
42         binary_latent_factor_model,
43         binary_latent_factor_approximation=mean_field_approximation,
44     )
45     fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
46     for i in range(k):
47         ax[i].imshow(binary_latent_factor_model.mu[:, i].reshape(4, 4))
48         ax[i].set_title(f"Latent Feature mu-{{i}}")
49     fig.suptitle("Learned Features (Mean Field Learning)")
50     plt.tight_layout()
51     plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
52     plt.close()
53
54     plt.title("Free Energy (Mean Field Learning)")
55     plt.xlabel("t (EM steps)")
56     plt.ylabel("Free Energy")
57     plt.plot(free_energy)
58     plt.savefig(save_path + "-free-energy", bbox_inches="tight")
59     plt.close()
60     return binary_latent_factor_model
61
62
63 def g(
64     x: np.ndarray,
65     binary_latent_factor_model: AbstractBinaryLatentFactorModel,
66     sigmas: List[float],
67     k: int,
68     em_iterations: int,
69     e_maximum_steps: int,
70     e_convergence_criterion: float,
71     save_path: str,
72 ) -> None:
73     n = x.shape[0]
74     free_energies = []
75     for sigma in sigmas:
76         binary_latent_factor_model.sigma = sigma
77         mean_field_approximation = init_mean_field_approximation(
78             k,
79             n,
80             max_steps=e_maximum_steps,
81             convergence_criterion=e_convergence_criterion,
82         )
83         free_energy: List[float] = [
84             mean_field_approximation.compute_free_energy(x, binary_latent_factor_model)
85         ]
86         for _ in range(em_iterations):
87             free_energy.pop(-1)
88             previous_lambda_matrix = np.copy(mean_field_approximation.lambda_matrix)
89             new_free_energy = mean_field_approximation.variational_expectation_step(
90                 binary_latent_factor_model=binary_latent_factor_model,
91                 x=x,
92             )
93             free_energy.extend(new_free_energy)
94             if (

```

```

95         free_energy[-1] - free_energy[-2]
96         <= mean_field_approximation.convergence_criterion
97     ):
98         free_energy.pop(-1)
99         break
100     if is_converge(
101         free_energy,
102         mean_field_approximation.lambda_matrix,
103         previous_lambda_matrix,
104     ):
105         break
106     free_energies.append(free_energy)
107
108 for i, free_energy in enumerate(free_energies):
109     plt.plot(
110         free_energy,
111         label=f"sigma={sigmas[i]}",
112     )
113 plt.title(f"F(t)")
114 plt.xlabel("t (Variational E steps)")
115 plt.ylabel("F(t)")
116 plt.tight_layout()
117 plt.legend()
118 plt.savefig(save_path + f"-free-energy-sigma.png", bbox_inches="tight")
119 plt.close()
120
121 for i, free_energy in enumerate(free_energies):
122     diffs = np.log(np.diff(free_energy))
123     plt.plot(
124         diffs,
125         label=f"sigma={sigmas[i]}",
126     )
127 plt.title(f"log(F(t)-F(t-1))")
128 plt.xlabel("t (Variational E steps)")
129 plt.ylabel("log(F(t)-F(t-1))")
130 plt.tight_layout()
131 plt.legend()
132 plt.savefig(save_path + f"-free-energy-diff-sigma.png", bbox_inches="tight")
133 plt.close()

```

src/solutions/q3.py

Question 4

We begin by writing the expression for x_d :

$$P(x_d|s, \mathbf{w}_d, \sigma^2) = \mathcal{N}(\mathbf{s}^T \mathbf{w}_d, \sigma^2)$$

where we know from the diagonal covariance of $P(\mathbf{x}|\mathbf{s}, \mu, \sigma^2)$ that each dimension is independent. Moreover, $\mathbf{w}_d \in \mathbb{R}^{K \times 1}$, which is the d^{th} row of $\mu \in \mathbb{R}^{D \times K}$

Thus, we can write the posterior:

$$\log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha) = \log P(\mathbf{s}|\pi) + \sum_{d=1}^D \log P(x_d|s, \mathbf{w}_d, \sigma^2) + \log P(\mathbf{w}_d|\alpha)$$

where we introduce priors on each \mathbf{w}_k with $\alpha \in \mathbb{R}^{K \times 1}$.

We choose each prior to be:

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

where $\mathbf{A} = \text{diag}(\alpha)$, the precision matrix.

Combining, we have our expression:

$$\begin{aligned} \log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha) = & \\ & + \sum_{d=1}^D -\frac{1}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} (x_d^2 - 2x_d \mathbf{s}^T \mathbf{w}_d + \mathbf{w}_d^T \mathbf{s} \mathbf{s}^T \mathbf{w}_d) \\ & + \sum_{k=1}^K s_k \log \pi_k + (1 - s_k) \log(1 - \pi_k) \\ & + \sum_{d=1}^D -\frac{K}{2} \log(2\pi) + \frac{1}{2} \sum_{k=1}^K (\log \alpha_k) - \frac{1}{2} \mathbf{w}_d^T \mathbf{A} \mathbf{w}_d \end{aligned}$$

For the Variational Bayes expectation step, we minimise $\mathbf{KL}[q_s(\mathbf{s}|\text{everything else})||P(\mathbf{s}|\text{everything else})]$ by setting:

$$q_s(\mathbf{s}) \propto \exp \langle \log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha) \rangle_{q(\mu)}$$

Substituting the relevant terms:

$$q_s(\mathbf{s}) \propto \exp \left\langle -\frac{1}{2\sigma^2} \left(-2\mathbf{x}^T \sum_{k=1}^K s_k \mu_k + \sum_{k=1}^K \sum_{k'=1, k' \neq k}^K s_k s_{k'} \mu_k^T \mu_{k'} + \sum_{k=1}^K s_k \mu_k^T \mu_k \right) + \sum_{k=1}^K s_k \log \frac{\pi_k}{1 - \pi_k} \right\rangle_{q(\mu)}$$

Given our factored approximation $q(\mathbf{s}) = \prod_{i=1}^K q_i(s_i)$, we can see that we can derive a similar partial update for $q_i(s_i)$ as in Question 3, by taking the variation derivative of the Lagrangian to enforce the normalisation of q_i :

$$\frac{\partial}{\partial q_i} \left(\exp \langle \log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha) \rangle_{q(\mu)} + \lambda^{LG} \int q_i - 1 \right) \propto \exp \langle \log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha) \rangle_{q(\mu) \prod_{j \neq i} q_j(s_j)} - \log q_i(s_i)$$

Setting this to zero we can solve for λ_i where $q_i(s_i) = \lambda_i^{s_i} (1 - \lambda_i)^{(1-s_i)}$:

$$\lambda_i = \frac{1}{1 + \exp \left[- \left(\frac{\langle \mu_i \rangle_{q_{\mu_i}}^T}{\sigma^2} \left(\mathbf{x} - \frac{\langle \mu_i \rangle_{q_{\mu_i}}}{2} - \sum_{j=1, j \neq i}^K \lambda_j \langle \mu_j \rangle_{q_{\mu_j}} \right) + \log \frac{\pi_i}{1 - \pi_i} \right) \right]}$$

we have our partial E step update.

For the maximisation step, we perform maximisation steps for the parameters σ and π in the same way as question 3. However, having defined a prior on μ (through \mathbf{w}) so we will have to derive our expression for $\langle \mu_k \rangle_{q_{\mu_k}}$ the expectation of the posterior on μ_k . This involves deriving the posterior distribution of \mathbf{w}_d

$$q_{\mathbf{w}_d}(\mathbf{w}_d) \propto P(\mathbf{w}_d) \exp \langle \log P(\mathbf{x}, \mathbf{s}, \mu | \pi, \sigma^2, \alpha) \rangle_{q_{\mathbf{s}}(\mathbf{s}) q_{-\mathbf{w}_d}(\mathbf{w}_d)}$$

Substituting the appropriate terms:

$$q_{\mathbf{w}_d}(\mathbf{w}_d) \propto \exp \left(-\frac{1}{2} \mathbf{w}_d^T \mathbf{A} \mathbf{w}_d \right) \exp \left\langle -\frac{1}{2\sigma^2} (-2x_d \mathbf{s}^T \mathbf{w}_d + \mathbf{w}_d^T \mathbf{s} \mathbf{s}^T \mathbf{w}_d) \right\rangle_{q_{\mathbf{s}}(\mathbf{s}) q_{-\mathbf{w}_d}(\mathbf{w}_d)}$$

Simplifying:

$$q_{\mathbf{w}_d}(\mathbf{w}_d) \propto \exp \left(-\frac{1}{2} \left(\mathbf{w}_d^T \left(\mathbf{A} + \frac{\langle \mathbf{s} \mathbf{s}^T \rangle_{q_{\mathbf{s}}(\mathbf{s})}}{\sigma^2} \right) \mathbf{w}_d - 2 \left(\frac{x_d \langle \mathbf{s}^T \rangle_{q_{\mathbf{s}}(\mathbf{s})}}{\sigma^2} \right) \mathbf{w}_d \right) \right)$$

We see that the posterior:

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

where:

$$\Sigma_{\mathbf{w}_d} = \left(\frac{\langle \mathbf{s} \mathbf{s}^T \rangle_{q_{\mathbf{s}}(\mathbf{s})}}{\sigma^2} + \mathbf{A} \right)^{-1}$$

and

$$\mu_{\mathbf{w}_d} = \Sigma_{\mathbf{w}_d} \left(\frac{x_d \langle \mathbf{s}^T \rangle_{q_{\mathbf{s}}(\mathbf{s})}}{\sigma^2} \right)$$

Thus, $\langle \mu_k \rangle_{q_{\mu_k}} \in \mathbb{R}^{D \times 1}$ is simply the concatenation of the k^{th} elements of $\mu_{\mathbf{w}_d}$ for $d \in \{1, \dots, D\}$

For ARD, we must also optimise α with a hyper-M step. We start by choose $Ga(\alpha_k | a, b)$, a Gamma prior on α_k , with a and b being hyperparameters. Thus, to optimise α we want to maximise the penalised objective:

$$\alpha = \arg \max_{\alpha} \langle \log P(\mathbf{x}, \mathbf{s}, \mu | \pi, \sigma^2, \alpha) \rangle_{q(\mathbf{w})} + \sum_{k=1}^K \log P(\alpha_k | a, b)$$

Substituting the appropriate terms, we have our penalised objective \mathcal{Q} :

$$\mathcal{Q} = \left\langle \sum_{d=1}^D \frac{1}{2} \sum_{k=1}^K (\log \alpha_k) - \frac{1}{2} \mathbf{w}_d^T \mathbf{A} \mathbf{w}_d \right\rangle_{q(\mathbf{w})} + \sum_{k=1}^K (a-1) \log \alpha_k - b \alpha_k$$

Simplifying:

$$\mathcal{Q} = \frac{D}{2} \sum_{k=1}^K (\log \alpha_k) - \frac{1}{2} \sum_{d=1}^D \left(\text{tr} \left[\mathbf{A} \langle \mathbf{w}_d \mathbf{w}_d^T \rangle_{q(\mathbf{w}_d)} \right] \right) + \sum_{k=1}^K (a-1) \log \alpha_k - b \alpha_k$$

Setting $\frac{d\mathcal{Q}}{d\alpha_k} = 0$ we get:

$$\frac{D}{2\alpha_k} - \frac{1}{2} \sum_{d=1}^D \langle (w_{d,k})^2 \rangle_{q(\mathbf{w}_d)} + \frac{a-1}{\alpha_k} - b = 0$$

where $w_{d,k}$ is the k^{th} element of \mathbf{w}_d .

Knowing $\langle (w_{d,k})^2 \rangle_{q(\mathbf{w}_d)} = (\mu_{\mathbf{w}_{d,k}})^2 + \Sigma_{\mathbf{w}_{d,(k,k)}}$, we can solve for α_k :

$$\alpha_k = \frac{2a + D - 2}{2b + \sum_{d=1}^D \left((\mu_{\mathbf{w}_{d,k}})^2 + \Sigma_{\mathbf{w}_{d,(k,k)}} \right)}$$

we have our hyper-M steps for optimising α .

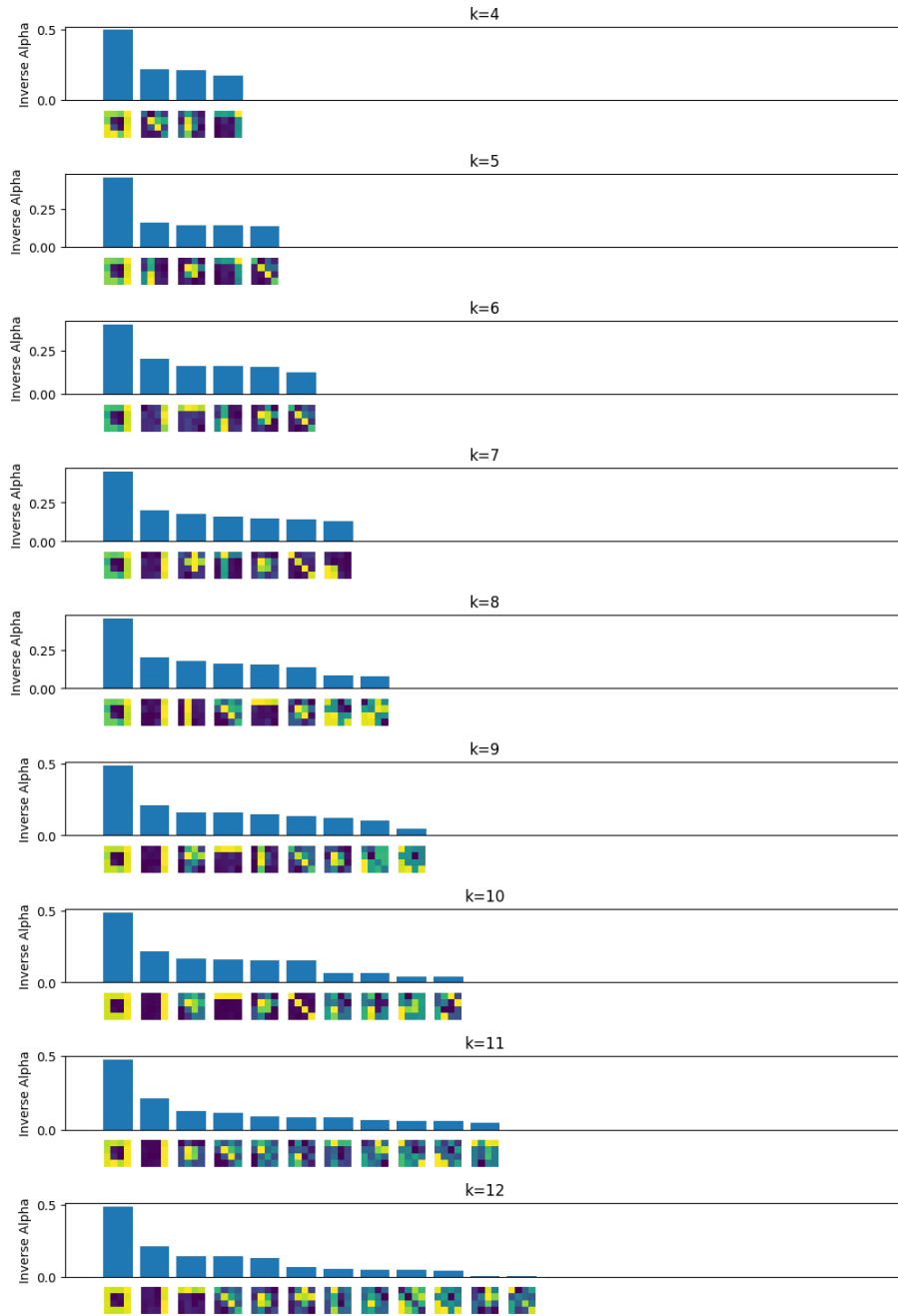


Figure 20: Learned Latent Factors vs Inverse Alpha

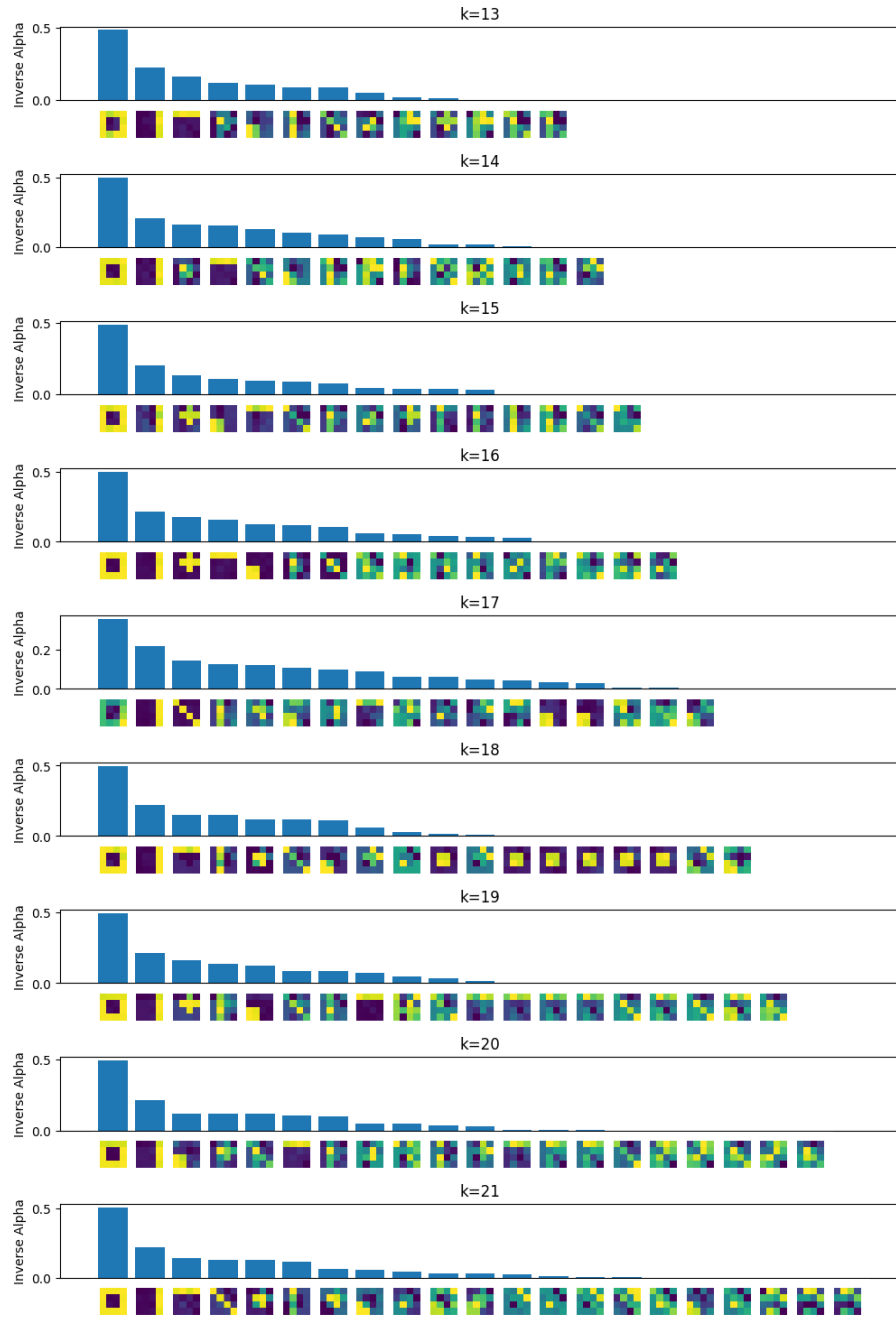


Figure 21: Learned Latent Factors vs Inverse Alpha

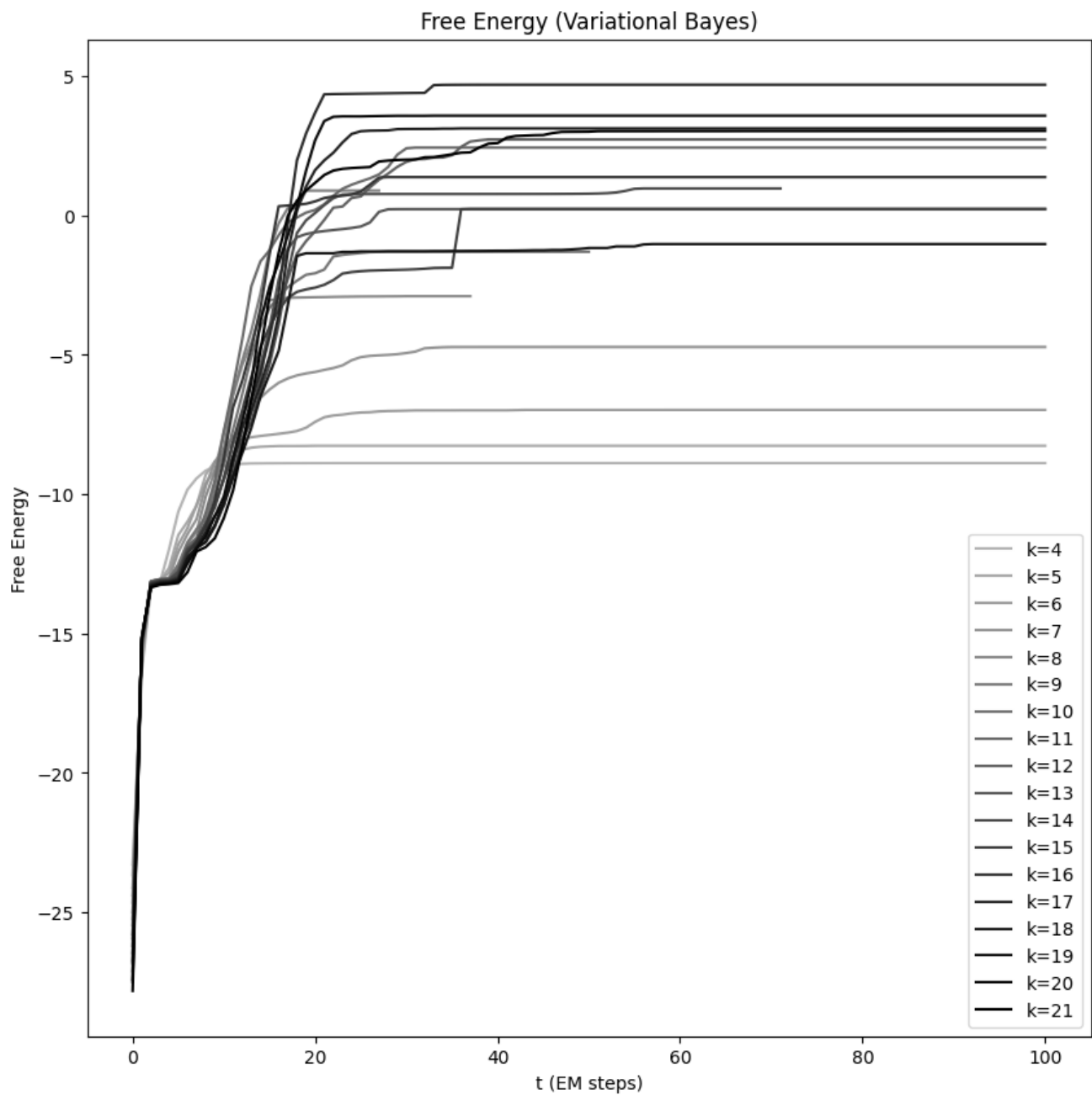


Figure 22: Free Energy for different values of k

The Python code for Variational Bayes:

```
1 import numpy as np
2
3 from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
4     AbstractBinaryLatentFactorApproximation,
5 )
6 from src.models.binary_latent_factor_models.binary_latent_factor_model import (
7     AbstractBinaryLatentFactorModel,
8     BinaryLatentFactorModel,
9 )
10
11
12 class GaussianPrior:
13     def __init__(self, a, b, d, k):
14         self.a = a
15         self.b = b
16         self.mu = np.zeros((d, k))
17         self.alpha = np.ones((k,)) # np.random.gamma(a, b, size=(k,))
18         self.w_covariance = np.zeros((k, k))
19
20     def mu_k(self, k): # (number_of_dimensions, 1)
21         return self.mu[:, k : k + 1]
22
23     def w_d(self, d): # (1, number_of_latent_variables)
24         return self.mu[d : d + 1, :]
25
26     @property
27     def a_matrix(self) -> np.ndarray:
28         # precision matrix for w_d
29         return np.diag(self.alpha)
30
31
32 class VariationalBayesBinaryLatentFactorModel(AbstractBinaryLatentFactorModel):
33     def __init__(self, mu: GaussianPrior, variance: float, pi: np.ndarray):
34         self.gaussian_prior = mu
35         self._variance = variance
36         self._pi = pi
37
38     @property
39     def variance(self) -> float:
40         return self._variance
41
42     @property
43     def pi(self) -> np.ndarray:
44         return self._pi
45
46     @property
47     def mu(self) -> np.ndarray:
48         return self.gaussian_prior.mu
49
50     def _update_w_d_covariance(
51         self,
52         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
53     ):
54         # expectation_s (number_of_points, number_of_latent_variables)
55         # expectation_ss (number_of_latent_variables, number_of_latent_variables)
56         self.gaussian_prior.w_covariance = np.linalg.inv(
57             self.gaussian_prior.a_matrix
58             + self.precision * binary_latent_factor_approximation.expectation_ss
59         )
60
61     def _update_w_d_mean(
62         self,
63         x: np.ndarray, # (number_of_points, number_of_dimensions)
64         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
65         d: int,
66     ):
67         # (number_of_latent_variables x 1)
68         self.gaussian_prior.mu[d : d + 1, :] = (
69             self.gaussian_prior.w_covariance
70             @ ( # (number_of_latent_variables, number_of_latent_variables)
71                 self.precision
72                 * binary_latent_factor_approximation.expectation_s.T # (number_of_latent_variables,
73                 number_of_points)
74                 @ x[:, d : d + 1] # (number_of_points, 1)
75             ).T
76         )
77
78     def _hyper_maximisation_step(self):
79         for k in range(self.k):
80             self.gaussian_prior.alpha[k] = (2 * self.gaussian_prior.a + self.d - 2) / (
81                 2 * self.gaussian_prior.b
82                 + np.sum(self.gaussian_prior.mu_k(k) ** 2)
83                 + self.d * self.gaussian_prior.w_covariance[k, k]
84             )
85
86     def maximisation_step(
87         self,
88         x: np.ndarray,
89         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
90     ) -> None:
91         _, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
92             x, binary_latent_factor_approximation
93         )
94         self._variance = sigma**2
```

```
94     self._pi = pi
95     self._update_w_d_covariance(binary_latent_factor_approximation)
96     for d in range(self.d):
97         self._update_w_d_mean(x, binary_latent_factor_approximation, d)
98     self._hyper_maximisation_step()
```

src/models/binary_latent_factor_models/variational_bayes.py

The rest of the Python code for question 4:

```

1 import os
2 from typing import List, Tuple
3
4 import matplotlib.pyplot as plt
5 import numpy as np
6 from matplotlib.offsetbox import AnnotationBbox, OffsetImage
7
8 from src.expectation_maximisation import learn_binary_factors
9 from src.models.binary_latent_factor_approximations.mean_field_approximation import (
10     init_mean_field_approximation,
11 )
12 from src.models.binary_latent_factor_models.binary_latent_factor_model import (
13     BinaryLatentFactorModel,
14 )
15 from src.models.binary_latent_factor_models.variational_bayes import (
16     GaussianPrior,
17     VariationalBayesBinaryLatentFactorModel,
18 )
19
20
21 def offset_image(coord, path, ax):
22     img = plt.imread(path)
23     im = OffsetImage(img, zoom=0.72)
24     im.image.axes = ax
25
26     ab = AnnotationBbox(
27         im,
28         (coord, 0),
29         xybox=(0.0, -19.0),
30         frameon=False,
31         xycoords="data",
32         boxcoords="offset points",
33         pad=0,
34     )
35
36     ax.add_artist(ab)
37
38
39 def _run_automatic_relevance_determination(
40     x: np.ndarray,
41     a_parameter: int,
42     b_parameter: int,
43     k: int,
44     em_iterations: int,
45     e_maximum_steps: int,
46     e_convergence_criterion: float,
47 ) -> Tuple[VariationalBayesBinaryLatentFactorModel, List[float]]:
48     n = x.shape[0]
49     mean_field_approximation = init_mean_field_approximation(
50         k, n, max_steps=e_maximum_steps, convergence_criterion=e_convergence_criterion
51     )
52     (_, sigma, pi,) = BinaryLatentFactorModel.calculate_maximisation_parameters(
53         x, mean_field_approximation
54     )
55     mu = GaussianPrior(
56         a=a_parameter,
57         b=b_parameter,
58         k=k,
59         d=x.shape[1],
60     )
61     binary_latent_factor_model: VariationalBayesBinaryLatentFactorModel = (
62         VariationalBayesBinaryLatentFactorModel(
63             mu=mu,
64             variance=sigma**2,
65             pi=pi,
66         )
67     )
68     (_, binary_latent_factor_model, free_energy,) = learn_binary_factors(
69         x=x,
70         em_iterations=em_iterations,
71         binary_latent_factor_model=binary_latent_factor_model,
72         binary_latent_factor_approximation=mean_field_approximation,
73     )
74     return binary_latent_factor_model, free_energy
75
76
77 def b(
78     x: np.ndarray,
79     a_parameter: int,
80     b_parameter: int,
81     ks: List[int],
82     max_k: int,
83     em_iterations: int,
84     e_maximum_steps: int,
85     e_convergence_criterion: float,
86     save_path: str,
87 ) -> None:
88
89     binary_latent_factor_models = []
90     free_energies = []
91     for i, k in enumerate(ks):
92         (
93             binary_latent_factor_model,
94             free_energy,

```

```

95         ) = _run_automatic_relevance_determination(
96             x,
97             a_parameter,
98             b_parameter,
99             k,
100             em_iterations,
101             e_maximum_steps,
102             e_convergence_criterion,
103         )
104         binary_latent_factor_models.append(binary_latent_factor_model)
105         free_energies.append(free_energy)
106
107     n = len(ks)
108     m = np.max(ks)
109     fig = plt.figure()
110     fig.set_figwidth(2 * n)
111     fig.set_figheight(2 * m)
112     for i, k in enumerate(ks):
113         sort_indices = np.argsort(binary_latent_factor_models[i].gaussian_prior.alpha)
114         for j, idx in enumerate(sort_indices):
115             ax = plt.subplot(n, m, m * i + j + 1)
116             ax.imshow(binary_latent_factor_models[i].mu[:, idx].reshape(4, 4))
117             ax.set_title(f"Latent Feature {idx+1}/{k}")
118     fig.suptitle("Learned Features (Variational Bayes)")
119     plt.tight_layout()
120     plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
121     plt.close()
122
123     for i, k in enumerate(ks):
124         sort_indices = np.argsort(binary_latent_factor_models[i].gaussian_prior.alpha)
125         for j, idx in enumerate(sort_indices):
126             fig = plt.figure(figsize=(0.3, 0.3))
127             ax = plt.Axes(fig, [0.0, 0.0, 1.0, 1.0])
128             ax.set_axis_off()
129             fig.add_axes(ax)
130             ax.imshow(binary_latent_factor_models[i].mu[:, idx].reshape(4, 4))
131             fig.savefig(save_path + f"-latent-factor-{i}-{j}.png", bbox_inches="tight")
132             plt.close()
133
134     fig, ax = plt.subplots(len(ks), 1, figsize=(12, 2 * len(ks)))
135     plt.subplots_adjust(hspace=1)
136     for i, k in enumerate(ks):
137         sort_indices = np.argsort(binary_latent_factor_models[i].gaussian_prior.alpha)
138         y = list(
139             1 / binary_latent_factor_models[i].gaussian_prior.alpha[sort_indices]
140         ) + [0] * (max_k - k)
141         ax[i].set_title(f"{k=}")
142         ax[i].bar(range(max_k), y)
143         ax[i].set_xticks([])
144         ax[i].set_ylabel("Inverse Alpha")
145     for i, k in enumerate(ks):
146         sort_indices = np.argsort(binary_latent_factor_models[i].gaussian_prior.alpha)
147         for j in range(len(sort_indices)):
148             path = save_path + f"-latent-factor-{i}-{j}.png"
149             offset_image(j, path, ax[i])
150             os.remove(path)
151     fig.savefig(save_path + f"-latent-factors-comparison", bbox_inches="tight")
152     plt.close()
153
154     fig = plt.figure()
155     fig.set_figwidth(10)
156     fig.set_figheight(10)
157     shades = np.flip(np.linspace(0, 0.7, len(ks)))
158     for i, k in enumerate(ks):
159         plt.plot(free_energies[i], label=f"{k=}", color=np.ones(3) * shades[i])
160     plt.title("Free Energy (Variational Bayes)")
161     plt.xlabel("t (EM steps)")
162     plt.ylabel("Free Energy")
163     plt.legend()
164     plt.savefig(save_path + "-free-energy", bbox_inches="tight")
165     plt.close()

```

src/solutions/q4.py

Question 5

(a)

The log-joint probability for a single observation-source pair:

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

Knowing $p(\mathbf{s}) = \prod_{i=1}^K p(s_i|\pi_i)$ and $p(\mathbf{x}|\mathbf{s}) = \mathcal{N}(\sum_{i=1}^K s_i \mu_i, \sigma^2 \mathbf{I})$:

$$\log p(\mathbf{s}, \mathbf{x}) \propto \frac{-1}{2} \left(\mathbf{x} - \sum_{i=1}^K s_i \mu_i \right)^T \frac{1}{\sigma^2} \mathbf{I} \left(\mathbf{x} - \sum_{i=1}^K s_i \mu_i \right) + \sum_{i=1}^K (s_i \log \pi_i + (1 - s_i) \log(1 - \pi_i))$$

Expanding:

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

Collecting terms pertaining to s_i :

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

where C are all other terms without s_i .

Knowing that $s_i^2 = s_i$:

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

Thus:

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

where the factors are defined:

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

and

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

as required.

The Boltzmann Machine can be defined:

$$P(\mathbf{s}|\mathbf{W}, \mathbf{b}) = \frac{1}{Z} \exp \left(\sum_{i=1}^K \sum_{j=1}^{i-1} W_{ij} s_i s_j - \sum_{i=1}^K b_i s_i \right)$$

where $s_i \in \{0, 1\}$, the same as our source variables.

From our factorisation, we can see that $p(\mathbf{s}, \mathbf{x})$ is a Boltzmann Machine with:

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

and

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

and

$$\log Z = -C$$

(b)

For $f_i(s_i)$, we will choose a Bernoulli approximation:

$$\tilde{f}_i(s_i) = \lambda_i^{s_i} + (1 - \lambda_i)^{1-s_i}$$

Thus,

$$\log \tilde{f}_i(s_i) \propto \log \left(\frac{\lambda_i}{1 - \lambda_i} \right) s_i$$

For $g_{ij}(s_i, s_j)$, we will choose a product of Bernoulli's approximation:

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

where

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

and

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

Thus,

$$\log \tilde{g}_{ij}(s_i, s_j) \propto \log \left(\frac{\theta_{ji}}{1 - \theta_{ji}} \right) s_i + \log \left(\frac{\theta_{ij}}{1 - \theta_{ij}} \right) s_j$$

we can define $\xi_{ji} = \log \left(\frac{\theta_{ji}}{1 - \theta_{ji}} \right)$ and $\xi_{ij} = \log \left(\frac{\theta_{ij}}{1 - \theta_{ij}} \right)$:

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

To derive the a message passing scheme, we first define the incoming message to node i from the singleton factor:

$$\mathcal{M}_i(s_i) = \tilde{f}_i(s_i)$$

and the message incoming message to node i from node j :

$$\mathcal{M}_{j \rightarrow i}(s_i) = \sum_{s_1 \in \{0,1\}} \cdots \sum_{s_{i-1} \in \{0,1\}} \sum_{s_{i+1} \in \{0,1\}} \cdots \sum_{s_1 \in \{0,1\}} \tilde{f}_j(s_j) \tilde{g}_{ji}(s_j, s_i) \prod_{k \in ne(j), k \neq i}^K \mathcal{M}_{k \rightarrow j}(s_j)$$

where $ne(j)$ are indices of neighbouring nodes of node j .

Because $\tilde{g}_{ji}(s_j, s_i)$ is a product:

$$\mathcal{M}_{j \rightarrow i}(s_i) = \tilde{g}_{ji, \neg s_j}(s_i) \sum_{s_1 \in \{0,1\}} \cdots \sum_{s_{i-1} \in \{0,1\}} \sum_{s_{i+1} \in \{0,1\}} \cdots \sum_{s_1 \in \{0,1\}} \tilde{f}_j(s_j) \tilde{g}_{ji, \neg s_i}(s_j) \prod_{k \in ne(j), k \neq i}^K \mathcal{M}_{k \rightarrow j}(s_j)$$

Simplifying:

$$\mathcal{M}_{j \rightarrow i}(s_i) = \tilde{g}_{ji, \neg s_j}(s_i)$$

and,

$$\mathcal{M}_{j \rightarrow i}(s_i) \propto \exp(\xi_{ji} s_i)$$

Thus, the cavity distributions are:

$$q_{\neg \tilde{f}_i(s_i)}(s_i) = \prod_{j \in ne(i)}^K \mathcal{M}_{j \rightarrow i}(s_i)$$

and

$$q_{\neg \tilde{g}_{ij}(s_i, s_j)}(s_i, s_j) = \left(\mathcal{M}_i(s_i) \prod_{k \in ne(i), k \neq j}^K \mathcal{M}_{k \rightarrow i}(s_i) \right) \left(\mathcal{M}_j(s_j) \prod_{k \in ne(j), k \neq i}^K \mathcal{M}_{k \rightarrow j}(s_j) \right)$$

For $\tilde{f}_i(s_i)$, we do not need to make an approximation step. This is because we are minimising:

$$\tilde{f}_i(s_i) = \arg \min_{\tilde{f}_i(s_i)} \mathbf{KL} \left[f_i(s_i) q_{\neg \tilde{f}_i(s_i)}(s_i) \parallel \tilde{f}_i(s_i) q_{\neg \tilde{f}_i(s_i)}(s_i) \right]$$

We know that the factor $\log f_i(s_i)$ is a Bernoulli of the form $b_i s_i$. Because our approximation for this site is also Bernoulli, we can simply solve for λ_i in $\log \tilde{f}_i(s_i)$:

$$\log \tilde{f}_i(s_i) = \log f_i(s_i)$$

$$\log \left(\frac{\lambda_i}{1 - \lambda_i} \right) s_i = b_i s_i$$

$$\lambda_i = \frac{1}{1 + \exp(-b_i)}$$

On the other hand, for $\tilde{g}_{ij}(s_i, s_j)$, we will approximate with:

$$\tilde{g}_{ij}(s_i, s_j) = \arg \min_{\tilde{g}_{ij}(s_i, s_j)} \mathbf{KL} \left[g_{ij}(s_i, s_j) q_{\neg \tilde{g}_{ij}(s_i, s_j)}(s_i, s_j) \parallel \tilde{g}_{ij}(s_i, s_j) q_{\neg \tilde{g}_{ij}(s_i, s_j)}(s_i, s_j) \right]$$

We can define natural parameters $\eta_{i,\neg s_j}$ and $\eta_{j,\neg s_i}$ for $q_{\tilde{g}_{ij}(s_i, s_j)}(s_i, s_j)$ such that:

$$\mathcal{M}_i(s_i) \prod_{k \in ne(i), k \neq j}^K \mathcal{M}_{k \rightarrow i}(s_i) \propto \exp(\eta_{i,\neg s_j} s_i)$$

$$\mathcal{M}_j(s_j) \prod_{k \in ne(j), k \neq i}^K \mathcal{M}_{k \rightarrow j}(s_j) \propto \exp(\eta_{j,\neg s_i} s_j)$$

Note that $\tilde{g}_{ij}(s_i, s_j)$ was chosen as the product of two Bernoulli distributions, updates to this site approximation involves updating the parameters ξ_{ij} and ξ_{ji} , for s_i and s_j respectively.

We can write:

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

Simplifying:

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

Thus, the first moments:

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

and

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

Moreover:

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

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

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

Thus, the first moment:

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

Simplifying:

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

Similarly:

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

By setting:

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

and

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

we can solve for the parameters of $\tilde{g}_{ij}(s_i, s_j)$ with moment matching:

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

Simplifying:

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

$$\frac{\exp(\eta_{j, \neg s_i}) + 1}{\exp(W_{ij} + \eta_{j, \neg s_i}) + 1} = \exp(-\xi_{ji})$$

Our parameter update:

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

Similarly:

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

(c)

Using factored approximate messages, we see that:

$$\eta_{i, \neg s_j} = \log \left(\frac{\lambda_i}{1 - \lambda_i} \right) + \sum_{k \in ne(i), k \neq j}^K \log \left(\frac{\theta_{ki}}{1 - \theta_{ki}} \right)$$

Knowing $b_i = \log \left(\frac{\lambda_i}{1 - \lambda_i} \right)$ and $\xi_{ki} = \log \left(\frac{\theta_{ki}}{1 - \theta_{ki}} \right)$:

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

and

$$\eta_{j,\neg s_i} = b_j + \sum_{k \in ne(j), k \neq i}^K \xi_{kj}$$

The summation of the natural parameters of the singleton factor for node i with the natural parameters of messages from all the neighbouring nodes.

This leads to a loopy BP algorithm because the nodes are fully connected (i.e. every node is the neighbour of all other nodes). Thus, we cannot simply move from one end of the graph to the other like BP for tree structured graphs.

(d)

We can use automatic relevance determination (ARD) as a hyperparameter method to select relevant features

Place prior on σ^2 and optimise with respect to the distributions would cause some to diverge and only relevant latent dimensions will remain. This gives us a value for K , the number of latent factors that haven't diverged.

Question 6

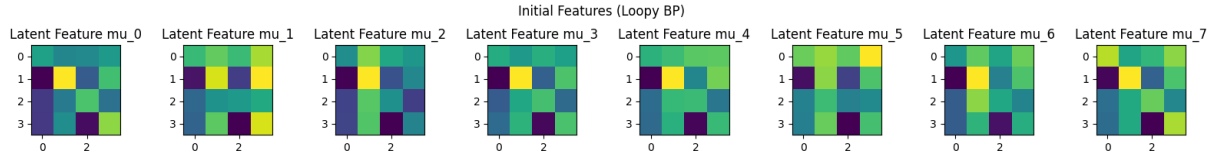


Figure 23: Initial Latent factors learned with EP/Loopy-BP

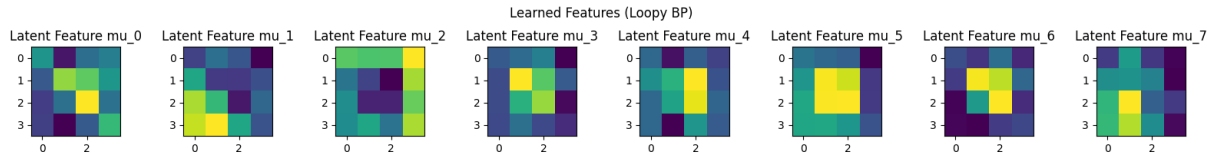


Figure 24: Learned Latent factors learned with EP/Loopy-BP

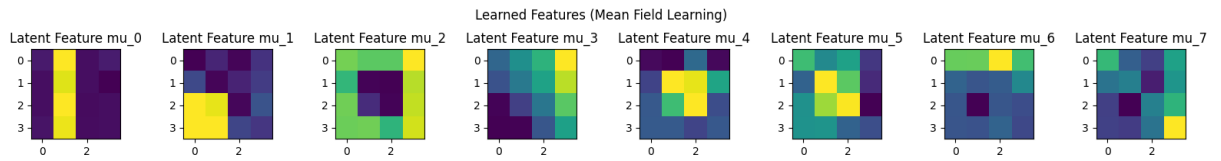


Figure 25: Learned Latent Factors with Mean Field Approximation

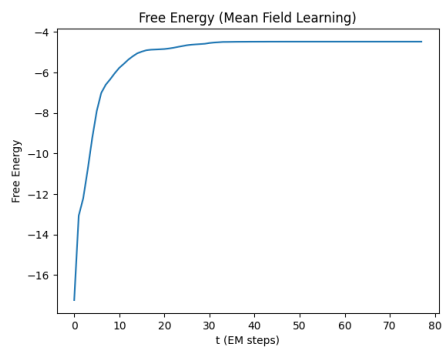


Figure 26: Mean Field Approximation

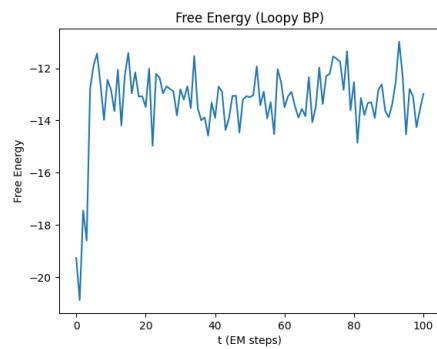


Figure 27: Loopy BP

The Python code for the Boltzmann machine:

```

1 import numpy as np
2
3 from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
4     AbstractBinaryLatentFactorApproximation,
5 )
6 from src.models.binary_latent_factor_models.binary_latent_factor_model import (
7     BinaryLatentFactorModel,
8 )
9
10
11 class BoltzmannMachine(BinaryLatentFactorModel):
12     """
13     mu: matrix of means (number_of_dimensions, number_of_latent_variables)
14     sigma: gaussian noise parameter
15     pi: vector of priors (1, number_of_latent_variables)
16     """
17
18     def __init__(
19         self,
20         mu: np.ndarray,
21         sigma: float,
22         pi: np.ndarray,
23     ):
24         super().__init__(mu, sigma, pi)
25
26     @property
27     def w_matrix(self) -> np.ndarray:
28         # (number_of_latent_variables, number_of_latent_variables)
29         return -self.precision * (self.mu.T @ self.mu)
30
31     def w_matrix_index(self, i, j) -> float:
32         return -self.precision * (self.mu[:, i] @ self.mu[:, j])
33
34     def b(self, x) -> np.ndarray:
35         """
36         :param x: design matrix (number_of_points, number_of_dimensions)
37         :return:
38         """
39         # (number_of_points, number_of_latent_variables)
40         return -(
41             self.precision * x @ self.mu
42             + self.log_pi_ratio
43             - 0.5 * self.precision * np.multiply(self.mu, self.mu).sum(axis=0)
44         )
45
46     def b_index(self, x, node_index) -> float:
47         # (number_of_points, 1)
48         return -(
49             self.precision * x @ self.mu[:, node_index]
50             + (self.log_pi[0, node_index] - self.log_one_minus_pi[0, node_index])
51             - 0.5 * self.precision * self.mu[:, node_index] @ self.mu[:, node_index]
52         ).reshape(
53             -1,
54         )
55
56     @property
57     def log_pi_ratio(self) -> np.ndarray:
58         return self.log_pi - self.log_one_minus_pi
59
60
61 def init_boltzmann_machine(
62     x: np.ndarray,
63     binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
64 ) -> BinaryLatentFactorModel:
65     mu, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
66         x, binary_latent_factor_approximation
67     )
68     return BoltzmannMachine(
69         mu=mu,
70         sigma=sigma,
71         pi=pi,
72     )
73

```

src/models/binary_latent_factor_models/boltzmann_machine.py

The Python code for message passing:

```

1 from typing import List
2
3 import numpy as np
4
5 from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
6     AbstractBinaryLatentFactorApproximation,
7 )
8 from src.models.binary_latent_factor_models.boltzmann_machine import BoltzmannMachine
9
10
11 class MessagePassing(AbstractBinaryLatentFactorApproximation):
12     """
13     eta_matrix: of parameters eta_matrix[n, i, j]
14                 off diagonals corresponds to  $\tilde{g}_{ij}$ ,  $\neg s_i(s_j)$  for data point n
15                 diagonals correspond to  $\tilde{f}_i(s_i)$ 
16                 (number_of_points, number_of_latent_variables, number_of_latent_variables)
17     """
18
19     def __init__(self, eta_matrix: np.ndarray):
20         self.eta_matrix = eta_matrix
21
22     @property
23     def lambda_matrix(self) -> np.ndarray:
24         lambda_matrix = 1 / (1 + np.exp(-self.xi.sum(axis=1)))
25         lambda_matrix[lambda_matrix == 0] = 1e-10
26         lambda_matrix[lambda_matrix == 1] = 1 - 1e-10
27         return lambda_matrix
28
29     @property
30     def xi(self) -> np.ndarray:
31         return np.log(np.divide(self.eta_matrix, 1 - self.eta_matrix))
32
33     def aggregate_incoming_binary_factor_messages(
34         self, node_index: int, excluded_node_index: int
35     ) -> np.ndarray:
36         # (number_of_points, )
37         # exclude message from excluded_node_index -> node_index
38         return (
39             np.sum(self.xi[:, :, excluded_node_index, node_index], axis=1)
40             + np.sum(self.xi[:, :, excluded_node_index + 1 :, node_index], axis=1)
41         ).reshape(
42             -1,
43         )
44
45     @staticmethod
46     def calculate_eta(xi: np.ndarray) -> np.ndarray:
47         eta = 1 / (1 + np.exp(-xi))
48         eta[eta == 0] = 1e-10
49         eta[eta == 1] = 1 - 1e-10
50         return eta
51
52     def variational_expectation_step(
53         self, x: np.ndarray, binary_latent_factor_model: BoltzmannMachine
54     ) -> List[float]:
55         free_energy = [self.compute_free_energy(x, binary_latent_factor_model)]
56         for i in range(self.k):
57             xi_new_ii = self.calculate_singleton_message_update(
58                 boltzmann_machine=binary_latent_factor_model,
59                 x=x,
60                 i=i,
61             )
62             self.eta_matrix[:, i, i] = self.calculate_eta(xi_new_ii)
63             free_energy.append(self.compute_free_energy(x, binary_latent_factor_model))
64
65             for j in range(i):
66                 xi_new_ij = self.calculate_binary_message_update(
67                     boltzmann_machine=binary_latent_factor_model,
68                     x=x,
69                     i=i,
70                     j=j,
71                 )
72                 self.eta_matrix[:, i, j] = self.calculate_eta(xi_new_ij)
73                 xi_new_ji = self.calculate_binary_message_update(
74                     boltzmann_machine=binary_latent_factor_model,
75                     x=x,
76                     i=j,
77                     j=i,
78                 )
79                 self.eta_matrix[:, j, i] = self.calculate_eta(xi_new_ji)
80                 free_energy.append(
81                     self.compute_free_energy(x, binary_latent_factor_model)
82                 )
83             return free_energy
84
85     def calculate_binary_message_update(
86         self,
87         x: np.ndarray,
88         boltzmann_machine: BoltzmannMachine,
89         i: int,
90         j: int,
91     ) -> float:
92         eta_i_not_j = boltzmann_machine.b.index(
93             x=x, node_index=i
94         ) + self.aggregate_incoming_binary_factor_messages(

```

```

95         node_index=i, excluded_node_index=j
96     )
97     w_i_j = boltzmann_machine.w_matrix_index(i, j)
98     return np.log(1 + np.exp(w_i_j + eta_i_not_j)) - np.log(1 + np.exp(eta_i_not_j))
99
100 @staticmethod
101 def calculate_singleton_message_update(
102     x: np.ndarray,
103     boltzmann_machine: BoltzmannMachine,
104     i: int,
105 ) -> float:
106     return boltzmann_machine.b_index(x=x, node_index=i)
107
108
109 def init_message_passing(k, n) -> MessagePassing:
110     eta_matrix = np.random.random(size=(n, k, k))
111     return MessagePassing(eta_matrix)

```

src/models/binary_latent_factor_approximations/message_passing.py

The rest of the Python code for question 6:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 from src.expectation_maximisation import learn_binary_factors
5 from src.models.binary_latent_factor_approximations.message_passing import (
6     init_message_passing,
7 )
8 from src.models.binary_latent_factor_models.boltzmann_machine import (
9     init_boltzmann_machine,
10 )
11
12
13 def run(x: np.ndarray, k: int, em_iterations: int, save_path: str) -> None:
14     n = x.shape[0]
15     message_passing = init_message_passing(k, n)
16     boltzmann_machine = init_boltzmann_machine(x, message_passing)
17     fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
18     for i in range(k):
19         ax[i].imshow(boltzmann_machine.mu[:, i].reshape(4, 4))
20         ax[i].set_title(f"Latent Feature mu_{i}")
21     fig.suptitle("Initial Features (Loopy BP)")
22     plt.tight_layout()
23     plt.savefig(save_path + "-init-latent-factors", bbox_inches="tight")
24     plt.close()
25     message_passing, boltzmann_machine, free_energy = learn_binary_factors(
26         x=x,
27         em_iterations=em_iterations,
28         binary_latent_factor_model=boltzmann_machine,
29         binary_latent_factor_approximation=message_passing,
30     )
31     fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
32     for i in range(k):
33         ax[i].imshow(boltzmann_machine.mu[:, i].reshape(4, 4))
34         ax[i].set_title(f"Latent Feature mu_{i}")
35     fig.suptitle("Learned Features (Loopy BP)")
36     plt.tight_layout()
37     plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
38     plt.close()
39
40     plt.title("Free Energy (Loopy BP)")
41     plt.xlabel("t (EM steps)")
42     plt.ylabel("Free Energy")
43     plt.plot(free_energy)
44     plt.savefig(save_path + "-free-energy", bbox_inches="tight")
45     plt.close()
```

src/solutions/q6.py

Appendix 1: abstract_binary_latent_factor_model.py

```
1 from __future__ import annotations
2
3 from abc import ABC, abstractmethod
4 from typing import TYPE_CHECKING
5
6 import numpy as np
7
8 if TYPE_CHECKING:
9     from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
10         AbstractBinaryLatentFactorApproximation,
11     )
12
13
14 class AbstractBinaryLatentFactorModel(ABC):
15     @property
16     @abstractmethod
17     def mu(self) -> np.ndarray:
18         pass
19
20     @property
21     @abstractmethod
22     def variance(self) -> float:
23         pass
24
25     @property
26     @abstractmethod
27     def pi(self) -> np.ndarray:
28         pass
29
30     @abstractmethod
31     def maximisation_step(
32         self,
33         x: np.ndarray,
34         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
35     ) -> None:
36         pass
37
38     def mu_exclude(self, exclude_latent_index: int) -> np.ndarray:
39         return np.concatenate( # (number_of_dimensions, number_of_latent_variables-1)
40             (self.mu[:, :exclude_latent_index], self.mu[:, exclude_latent_index + 1 :]),
41             axis=1,
42         )
43
44     @property
45     def log_pi(self) -> np.ndarray:
46         return np.log(self.pi)
47
48     @property
49     def log_one_minus_pi(self) -> np.ndarray:
50         return np.log(1 - self.pi)
51
52     @property
53     def precision(self) -> float:
54         return 1 / self.variance
55
56     @property
57     def d(self) -> int:
58         return self.mu.shape[0]
59
60     @property
61     def k(self) -> int:
62         return self.mu.shape[1]
```

src/models/binary_latent_factor_models/abstract_binary_latent_factor_model.py

Appendix 2: abstract_binary_latent_factor_approximation.py

```
1 from __future__ import annotations
2
3 from abc import ABC, abstractmethod
4 from typing import TYPE_CHECKING, List
5
6 if TYPE_CHECKING:
7     from src.models.binary_latent_factor_models.binary_latent_factor_model import (
8         AbstractBinaryLatentFactorModel,
9     )
10
11 import numpy as np
12
13
14 class AbstractBinaryLatentFactorApproximation(ABC):
15     @property
16     @abstractmethod
17     def lambda_matrix(self) -> np.ndarray:
18         pass
19
20     @abstractmethod
21     def variational_expectation_step(
22         self,
23         x: np.ndarray,
24         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
25     ) -> List[float]:
26         pass
27
28     @property
29     def expectation_s(self):
30         return self.lambda_matrix
31
32     @property
33     def expectation_ss(self):
34         ess = self.lambda_matrix.T @ self.lambda_matrix
35         np.fill_diagonal(ess, self.lambda_matrix.sum(axis=0))
36         return ess
37
38     @property
39     def log_lambda_matrix(self) -> np.ndarray:
40         return np.log(self.lambda_matrix)
41
42     @property
43     def log_one_minus_lambda_matrix(self) -> np.ndarray:
44         return np.log(1 - self.lambda_matrix)
45
46     @property
47     def n(self) -> int:
48         return self.lambda_matrix.shape[0]
49
50     @property
51     def k(self) -> int:
52         return self.lambda_matrix.shape[1]
53
54     def compute_free_energy(
55         self,
56         x: np.ndarray,
57         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
58     ) -> float:
59         """
60         free energy associated with current EM parameters and data x
61
62         :param x: data matrix (number_of_points, number_of_dimensions)
63         :param binary_latent_factor_model: a binary_latent_factor_model
64         :return: average free energy per data point
65         """
66         expectation_log_p_x_s_given_theta = (
67             self._compute_expectation_log_p_x_s_given_theta(
68                 x, binary_latent_factor_model
69             )
70         )
71         approximation_model_entropy = self._compute_approximation_model_entropy()
72         return (
73             expectation_log_p_x_s_given_theta + approximation_model_entropy
74         ) / self.n
75
76     def _compute_expectation_log_p_x_s_given_theta(
77         self,
78         x: np.ndarray,
79         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
80     ) -> float:
81         """
82         The first term of the free energy, the expectation of log P(X,S|theta)
83
84         :param x: data matrix (number_of_points, number_of_dimensions)
85         :param binary_latent_factor_model: a binary_latent_factor_model
86         :return: the expectation of log P(X,S|theta)
87         """
88         # (number_of_points, number_of_dimensions)
89         mu_lambda = self.lambda_matrix @ binary_latent_factor_model.mu.T
90
91         # (number_of_latent_variables, number_of_latent_variables)
92         expectation_s_i_s_j_mu_i_mu_j = np.multiply(
```

```

93         self.lambda_matrix.T @ self.lambda_matrix,
94         binary_latent_factor_model.mu.T @ binary_latent_factor_model.mu,
95     )
96
97     expectation_log_p_x_given_s_theta = -(
98         self.n * binary_latent_factor_model.d / 2
99     ) * np.log(2 * np.pi * binary_latent_factor_model.variance) - (
100         0.5 * binary_latent_factor_model.precision
101     ) * (
102         np.sum(np.multiply(x, x))
103         - 2 * np.sum(np.multiply(x, mu.lambda))
104         + np.sum(expectation_s_i_s_j_mu_i_mu_j)
105         - np.trace(
106             expectation_s_i_s_j_mu_i_mu_j
107         ) # remove incorrect E[s_i s_i] = lambda_i * lambda_i
108         + np.sum( # add correct E[s_i s_i] = lambda_i
109             self.lambda_matrix
110             @ np.multiply(
111                 binary_latent_factor_model.mu, binary_latent_factor_model.mu
112             ).T
113         )
114     )
115     expectation_log_p_s_given_theta = np.sum(
116         np.multiply(
117             self.lambda_matrix,
118             binary_latent_factor_model.log_pi,
119         )
120         + np.multiply(
121             1 - self.lambda_matrix,
122             binary_latent_factor_model.log_one_minus_pi,
123         )
124     )
125     return expectation_log_p_x_given_s_theta + expectation_log_p_s_given_theta
126
127 def _compute_approximation_model_entropy(self) -> float:
128     return -np.sum(
129         np.multiply(
130             self.lambda_matrix,
131             self.log_lambda_matrix,
132         )
133         + np.multiply(
134             1 - self.lambda_matrix,
135             self.log_one_minus_lambda_matrix,
136         )
137     )

```

src/models/binary_latent_factor_approximations/abstract_binary_latent_factor_approximation.py

Appendix 3: main.py

```
1 import os
2 from dataclasses import asdict
3
4 import jax
5 import jax.numpy as jnp
6 import numpy as np
7 import pandas as pd
8
9 from src.constants import CO2_FILE_PATH, DEFAULT_SEED, OUTPUTS_FOLDER
10 from src.generate_images import generate_images
11 from src.models.bayesian_linear_regression import LinearRegressionParameters
12 from src.models.gaussian_process_regression import GaussianProcessParameters
13 from src.models.kernels import CombinedKernel, CombinedKernelParameters
14 from src.solutions import q2, q3, q4, q6
15
16 jax.config.update("jax_enable_x64", True)
17
18 if __name__ == "__main__":
19     np.random.seed(DEFAULT_SEED)
20
21     if not os.path.exists(OUTPUTS_FOLDER):
22         os.makedirs(OUTPUTS_FOLDER)
23
24     # Question 2
25     Q2_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q2")
26     if not os.path.exists(Q2_OUTPUT_FOLDER):
27         os.makedirs(Q2_OUTPUT_FOLDER)
28     with open(CO2_FILE_PATH) as file:
29         lines = [line.rstrip().split() for line in file]
30
31     df_co2 = pd.DataFrame(
32         np.array([line for line in lines if line[0] != "#"]).astype(float)
33     )
34     column_names = lines[max([i for i, line in enumerate(lines) if line[0] == "#"])[1:]]
35     df_co2.columns = column_names
36     t = df_co2.decimal.values[:] - np.min(df_co2.decimal.values[:])
37     y = df_co2.average.values[:].reshape(1, -1)
38
39     sigma = 1
40     mean = np.array([0, 360]).reshape(-1, 1)
41     covariance = np.array(
42         [
43             [10**2, 0],
44             [0, 100**2],
45         ]
46     )
47     kernel = CombinedKernel()
48     kernel_parameters = CombinedKernelParameters(
49         log_theta=jnp.log(1),
50         log_sigma=jnp.log(1),
51         log_phi=jnp.log(1),
52         log_eta=jnp.log(1),
53         log_tau=jnp.log(1),
54         log_zeta=jnp.log(1e-1),
55     )
56
57     prior_linear_regression_parameters = LinearRegressionParameters(
58         mean=mean,
59         covariance=covariance,
60     )
61     posterior_linear_regression_parameters = q2.a(
62         t,
63         y,
64         sigma,
65         prior_linear_regression_parameters,
66         save_path=os.path.join(Q2_OUTPUT_FOLDER, "a"),
67     )
68     q2.b(
69         t_year=df_co2.decimal.values[:],
70         t=t,
71         y=y,
72         linear_regression_parameters=posterior_linear_regression_parameters,
73         error_mean=0,
74         error_variance=1,
75         save_path=os.path.join(Q2_OUTPUT_FOLDER, "b"),
76     )
77
78     q2.c(
79         kernel=kernel,
80         kernel_parameters=kernel_parameters,
81         log_theta_range=jnp.log(jnp.linspace(1e-2, 5, 5)),
82         t=t[:50].reshape(-1, 1),
83         number_of_samples=3,
84         save_path=os.path.join(Q2_OUTPUT_FOLDER, "c"),
85     )
86
87     init_kernel_parameters = CombinedKernelParameters(
88         log_theta=jnp.log(5),
89         log_sigma=jnp.log(5),
90         log_phi=jnp.log(10),
91         log_eta=jnp.log(5),
92         log_tau=jnp.log(1),
```

```

93     log_zeta=jnp.log(2),
94 )
95 gaussian_process_parameters = GaussianProcessParameters(
96     kernel=asdict(init_kernel_parameters),
97     log_sigma=jnp.log(1),
98 )
99 years_to_predict = 15
100 t_new = t[-1] + np.linspace(0, years_to_predict, years_to_predict * 12)
101 t_test = np.concatenate((t, t_new))
102 q2.f(
103     t_train=t,
104     y_train=y,
105     t_test=t_test,
106     min_year=np.min(df_co2.decimal.values[:]),
107     prior_linear_regression_parameters=prior_linear_regression_parameters,
108     linear_regression_sigma=sigma,
109     kernel=kernel,
110     gaussian_process_parameters=gaussian_process_parameters,
111     learning_rate=1e-2,
112     number_of_iterations=100,
113     save_path=os.path.join(Q2.OUTPUT_FOLDER, "f"),
114 )
115
116 # Question 3
117 Q3.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q3")
118 if not os.path.exists(Q3.OUTPUT_FOLDER):
119     os.makedirs(Q3.OUTPUT_FOLDER)
120 number_of_images = 2000
121 x = generate_images(n=number_of_images)
122 k = 8
123 em_iterations = 100
124 e_maximum_steps = 50
125 e_convergence_criterion = 0
126
127 binary_latent_factor_model = q3.e_and_f(
128     x=x,
129     k=k,
130     em_iterations=em_iterations,
131     e_maximum_steps=e_maximum_steps,
132     e_convergence_criterion=e_convergence_criterion,
133     save_path=os.path.join(Q3.OUTPUT_FOLDER, "f"),
134 )
135 - = q3.e_and_f(
136     x=x,
137     k=int(k * 1.5),
138     em_iterations=em_iterations,
139     e_maximum_steps=e_maximum_steps,
140     e_convergence_criterion=e_convergence_criterion,
141     save_path=os.path.join(Q3.OUTPUT_FOLDER, "f-larger-k"),
142 )
143 q3.g(
144     x=x[:1, :],
145     binary_latent_factor_model=binary_latent_factor_model,
146     sigmas=[1, 2, 3],
147     k=k,
148     em_iterations=em_iterations,
149     e_maximum_steps=e_maximum_steps,
150     e_convergence_criterion=e_convergence_criterion,
151     save_path=os.path.join(Q3.OUTPUT_FOLDER, "g"),
152 )
153 # Question 4
154 Q4.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q4")
155 if not os.path.exists(Q4.OUTPUT_FOLDER):
156     os.makedirs(Q4.OUTPUT_FOLDER)
157 max_k = 21
158 q4.b(
159     x=x,
160     a_parameter=1,
161     b_parameter=0,
162     ks=np.arange(4, 22),
163     max_k=max_k,
164     em_iterations=em_iterations,
165     e_maximum_steps=e_maximum_steps,
166     e_convergence_criterion=e_convergence_criterion,
167     save_path=os.path.join(Q4.OUTPUT_FOLDER, "b"),
168 )
169 q4.b(
170     x=x,
171     a_parameter=1,
172     b_parameter=0,
173     ks=np.arange(4, 13),
174     max_k=max_k,
175     em_iterations=em_iterations,
176     e_maximum_steps=e_maximum_steps,
177     e_convergence_criterion=e_convergence_criterion,
178     save_path=os.path.join(Q4.OUTPUT_FOLDER, "b-1"),
179 )
180 q4.b(
181     x=x,
182     a_parameter=1,
183     b_parameter=0,
184     ks=np.arange(13, 22),
185     max_k=max_k,
186     em_iterations=em_iterations,
187     e_maximum_steps=e_maximum_steps,
188     e_convergence_criterion=e_convergence_criterion,

```



```
189     save_path=os.path.join(Q4_OUTPUT_FOLDER, "b-2"),
190 )
191 # Question 6
192 Q6_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q6")
193 if not os.path.exists(Q6_OUTPUT_FOLDER):
194     os.makedirs(Q6_OUTPUT_FOLDER)
195 q6.run(x, k, em.iterations, save_path=os.path.join(Q6_OUTPUT_FOLDER, "all"))
```

main.py

Appendix 4: constants.py

```
1 import os
2
3 DATA_FOLDER = "data"
4
5 CO2_FILE_PATH = os.path.join(DATA_FOLDER, "co2.txt")
6 IMAGES_FILE_PATH = os.path.join(DATA_FOLDER, "images.jpg")
7
8 OUTPUTS_FOLDER = "outputs"
9
10 DEFAULT_SEED = 0
11
12 M1 = [0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0]
13
14 M2 = [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0]
15
16 M3 = [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
17
18 M4 = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]
19
20 M5 = [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0]
21
22 M6 = [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1]
23
24 M7 = [0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0]
25
26 M8 = [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]
```

src/constants.py

Appendix 5: generate_images.py

```
1 import numpy as np
2
3 from src.constants import DEFAULT_SEED, M1, M2, M3, M4, M5, M6, M7, M8
4
5
6 def generate_images(n: int = 400, seed: int = DEFAULT_SEED, sigma: float = 0.1):
7     """
8
9     :param n: number of data points
10    :param seed: random seed
11    :param sigma: Gaussian noise
12    :return:
13    """
14    d = 16 # dimensionality of the data
15    np.random.seed(seed)
16
17    # Define the basic shapes of the features
18    number_of_features = 8 # number of features
19    rr = (
20        0.5 + np.random.rand(number_of_features, 1) * 0.5
21    ) # weight of each feature between 0.5 and 1
22    mut = np.array(
23        [
24            rr[0] * M1,
25            rr[1] * M2,
26            rr[2] * M3,
27            rr[3] * M4,
28            rr[4] * M5,
29            rr[5] * M6,
30            rr[6] * M7,
31            rr[7] * M8,
32        ]
33    )
34    s = (
35        np.random.rand(n, number_of_features) < 0.3
36    ) # each feature occurs with prob 0.3 independently
37
38    # Generate Data — The Data is stored in Y
39
40    return (
41        np.dot(s, mut) + np.random.randn(n, d) * sigma
42    ) # some Gaussian noise is added
```

src/generate_images.py

Appendix 6: MStep.py

```
1 import numpy as np
2
3
4 def m_step(x, es, ess):
5     """
6     mu, sigma, pie = MStep(x, es, ess)
7
8     Inputs:
9
10         x: shape (n, d) data matrix
11         es: shape (n, k) E_q[s]
12         ess: shape (k, k) sum over data points of E_q[ss'] (n, k, k)
13             if E_q[ss'] is provided, the sum over n is done for you.
14
15     Outputs:
16
17         mu: shape (d, k) matrix of means in p(y|{s_i}, mu, sigma)
18         sigma: shape (,) standard deviation in same
19         pie: shape (1, k) vector of parameters specifying generative distribution for s
20     """
21     n, d = x.shape
22     if es.shape[0] != n:
23         raise TypeError('es must have the same number of rows as x')
24     k = es.shape[1]
25     if ess.shape == (n, k, k):
26         ess = np.sum(ess, axis=0)
27     if ess.shape != (k, k):
28         raise TypeError('ess must be square and have the same number of columns as es')
29
30     mu = np.dot(np.dot(np.linalg.inv(ess), es.T), x).T
31     sigma = np.sqrt((np.trace(np.dot(x.T, x)) + np.trace(np.dot(np.dot(mu.T, mu), ess))
32                     - 2 * np.trace(np.dot(np.dot(es.T, x), mu))) / (n * d))
33     pie = np.mean(es, axis=0, keepdims=True)
34
35     return mu, sigma, pie
```

demo_code/MStep.py