

COMP0085 Summative Assignment

Jan 6, 2023

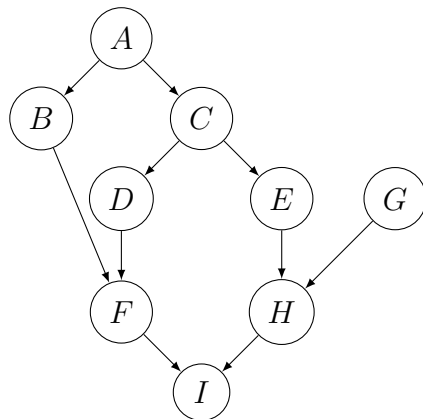
Question 1: A biochemical pathway

(a)

We are given that:

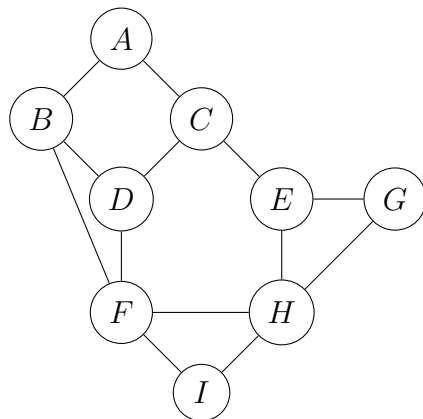
- The concentrations of both species B and C depend on that of A.
- Molecule C seems to redirect the process that produces enzyme D to produce enzyme E instead.
- D catalyses the production of F from B, while E catalyses the production of H from G.
- F and H then combine to form the end product I.

The corresponding directed acyclic graph:

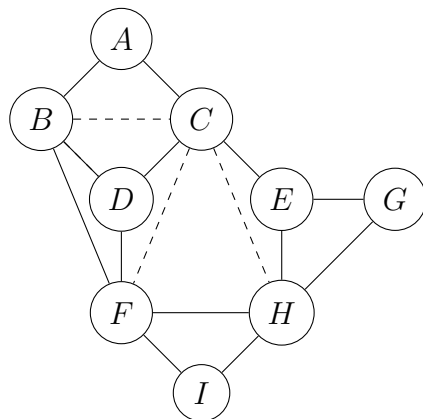


(b)

The moralised graph:

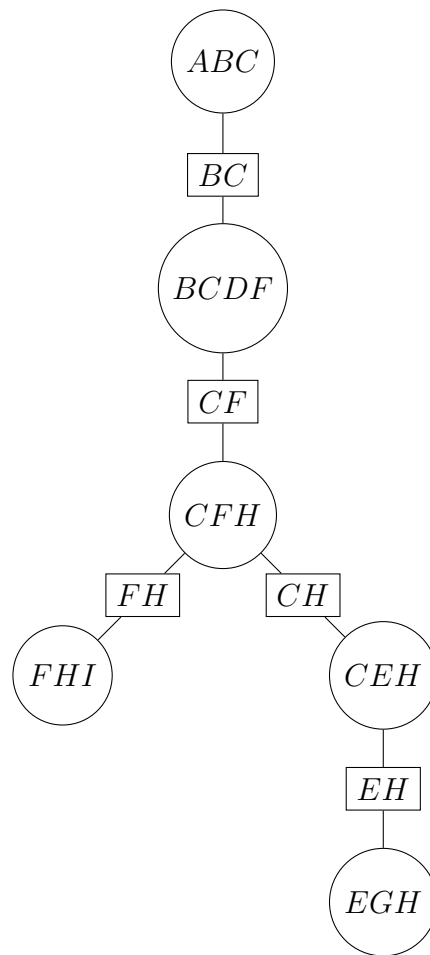


An effective triangulation:

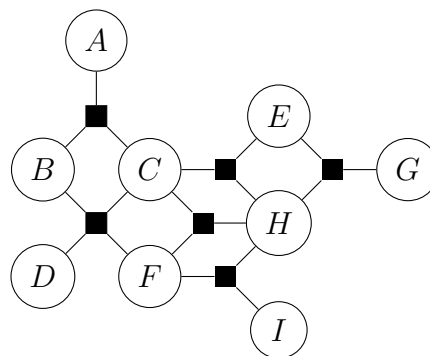


where the dashed lines are edges added to triangulate the moralised graph.

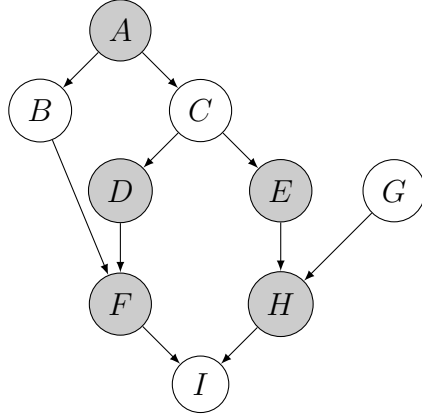
The resulting junction tree:



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, $\epsilon \sim \mathcal{N}(0, \Psi)$ are the reaction specific Gaussian noise variables, Λ is the factor loading matrix, and $\mathbf{z} \sim \mathcal{N}(0, I)$ are the latent factors, the parent concentration perturbations.

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 $\delta[G]$:

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

Knowing \mathbf{z} are the parent concentration perturbations, we can see that these simplify to a factor analysis set up:

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

Thus, we would expect to recover the factors of A and C , $\delta[A]$ and $\delta[C]$, the two parent nodes of the observations.

(e)

From factor analysis, we would recover $\delta[A]$ and $\delta[C]$. In theory, if these concentration perturbations had unknown parent factors, we could use the results to recover the concentration perturbations of those parent features. However, in our DAG this is not the case, so we would not be able to recover any other species in the cascade because they are all downstream from the known concentration perturbations and those recovered from factor analysis. Similarly, the downstream weights in the DAG from the known measurements would also be unidentifiable. Using factor analysis, we would only be able to identify the upstream weights from the measurements, up to an unknown scale factor. These are the weights Λ_{BA} , Λ_{DC} , and Λ_{EC} . Moreover, knowing $\delta[C]$ from factor analysis, we would also be able to determine Λ_{CA} , up to an unknown scale factor.

Question 2: Bayesian linear and Gaussian process regression

(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} :

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

For our data $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$:

$$P(\mathcal{D}|\mathbf{w}) = \prod_{t=1}^T \mathcal{N}(y_t - \mathbf{w}^T \mathbf{x}_t, \sigma^2)$$

where $\mathbf{x}_t = \begin{bmatrix} t_1 \\ 1 \end{bmatrix} \in \mathbb{R}^{2 \times 1}$ and $y_t \in \mathbb{R}^{1 \times 1}$ our response $f_{obs}(t_1)$.

Knowing:

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

we can substitute the above distributions:

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

where $\mathbf{X} = \begin{bmatrix} t_1 & t_2 \cdots t_T \\ 1 & 1 \cdots 1 \end{bmatrix} \in \mathbb{R}^{2 \times T}$ and $\mathbf{y} \in \mathbb{R}^{1 \times T}$. 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} :

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

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

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

Thus we can identify the posterior covariance:

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

and the posterior mean:

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

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

	value
parameters a	1.83E+00
b	3.34E+02

Figure 1: The Posterior Mean

	parameters	
	a	b
parameters a	1.38E-05	-2.87E-04
b	-2.87E-04	7.98E-03

Figure 2: The Posterior Covariance

(b)

Plotting the residuals $g_{obs}(t) = f_{obs}(t) - (a_{MAP}t + b_{MAP})$:

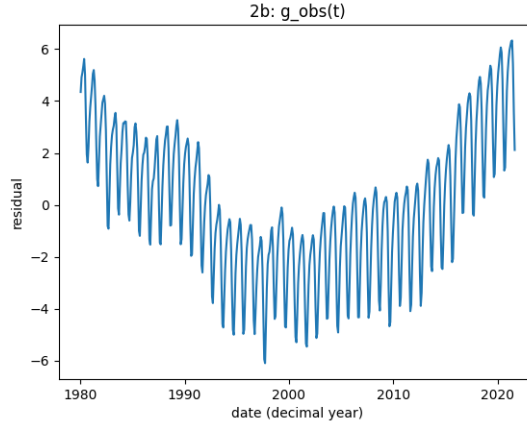


Figure 3: $g_{obs}(t)$

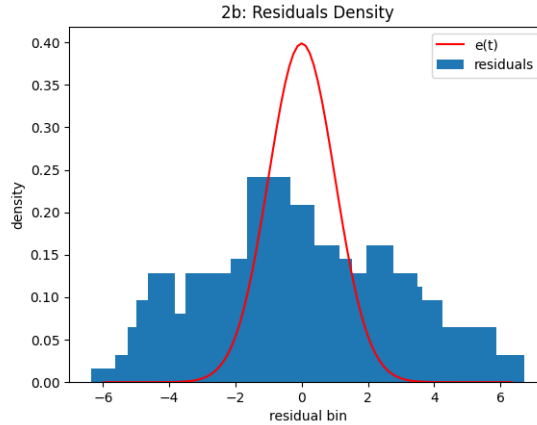


Figure 4: Density Estimation of Residuals vs $e(t) \sim \mathcal{N}(0, 1)$

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, observing $g_{obs}(t)$ we see that there is periodic structure in the timeseries meaning that the data is not independently and identically distributed as $e(t) \sim \mathcal{N}(0, 1)$ would suggest. If this were the case, we would expect $g(t)$ to look like random noise.

(c & d)

We are considering the kernel:

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

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

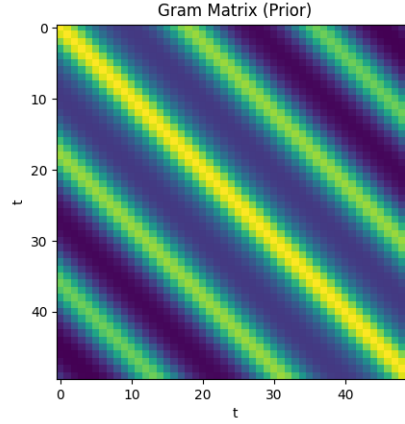


Figure 5: Covariance Matrix

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 periodic structure in 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 visualising $g(t)$, we would want to model this with a class of functions which exhibit both of these behaviours as $g(t)$ looks periodic (seasonal with respect to each year) and locally correlated (smooth).

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.

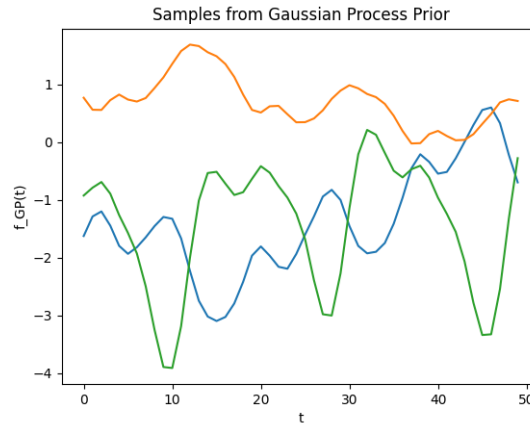


Figure 6: Samples from a zero mean GP with the provided covariance kernel

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

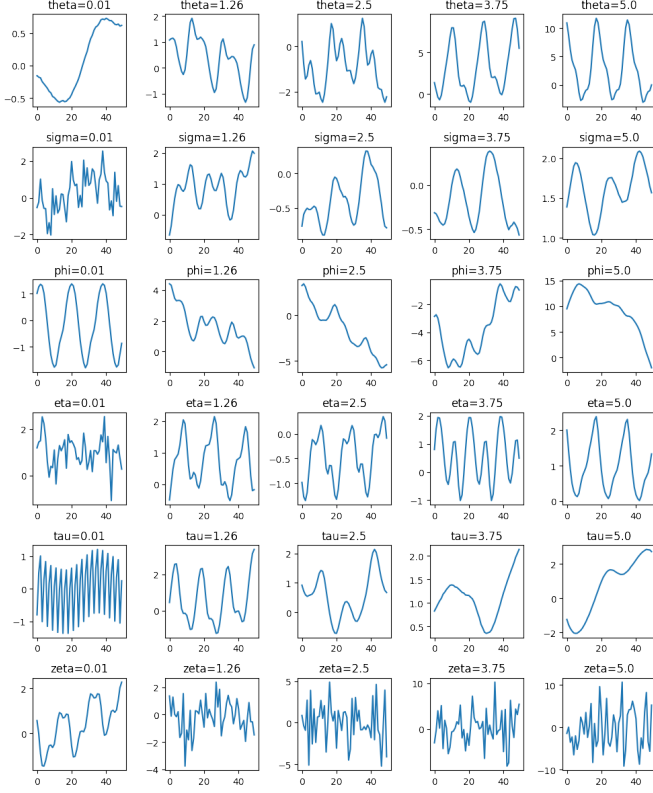


Figure 7: Samples for different parameters

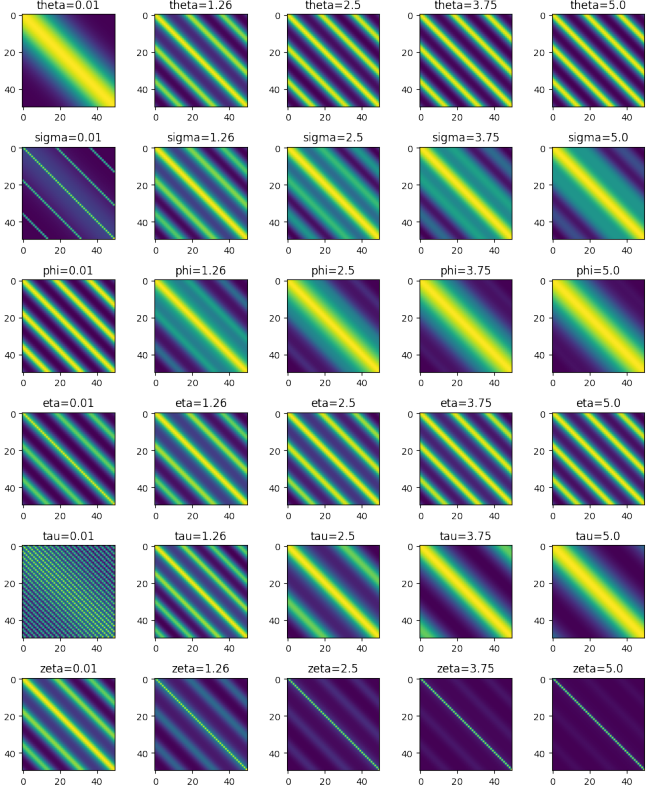


Figure 8: Covariances for different parameters

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

σ : As σ increases, we see smoother periodic behaviour in the sample function. The covariance matrix shows how increasing σ will increase covariance values around each periodic stripe of the covariance matrix. This is expected because it adjusts the lengthscale of the periodic portion of the kernel.

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 masking the periodic stripes. This is expected because it adjusts the weight of the squared exponential portion of the kernel. 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 squared exponential 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 weighting between 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 and the period of the samples. This makes sense because we are adjusting the period parameter in the sinusoid function of the periodic term.

ζ : As ζ increases, the function becomes less smooth. This is because the ζ parameter adjusts the weight of the $\delta_{s=t}$ kernel. 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. This masks the periodic and squared-exponential components 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. This behaviour cannot be modelled by our Bayesian linear regression and is reflected in our residuals $g(t)$. Thus we can choose $\tau = 1$ to ensure functions with a period of one year to reflect this prior 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 define a loss function as the negative log-likelihood and employ gradient-based algorithms to find optimal values for these parameters.

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

parameter	value
eta (kernel)	5.00E+00
phi (kernel)	1.00E+01
sigma	1.00E+00
sigma (kernel)	5.00E+00
tau (kernel)	1.00E+00
theta (kernel)	5.00E+00
zeta (kernel)	2.00E+00

Figure 9: Untrained hyperparameters

parameter	value
eta (kernel)	5.06E+00
phi (kernel)	4.99E+00
sigma	3.73E-01
sigma (kernel)	2.82E+00
tau (kernel)	9.99E-01
theta (kernel)	7.02E+00
zeta (kernel)	7.45E-01

Figure 10: Trained Hyperparameters

We can analyse some of the changes in these parameters from optimisation to gain some insights. We can see that τ remains the same as we would expect, given the yearly seasonality we knew apriori. 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 and are not independently and identically distributed.

(f)

Extrapolating the CO_2 concentration levels using both the untrained and trained hyperparameters:

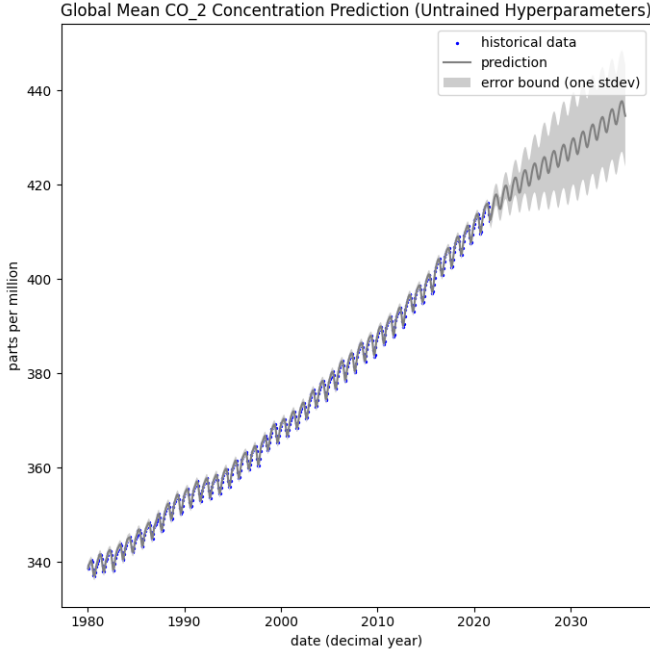


Figure 11: Untrained extrapolation

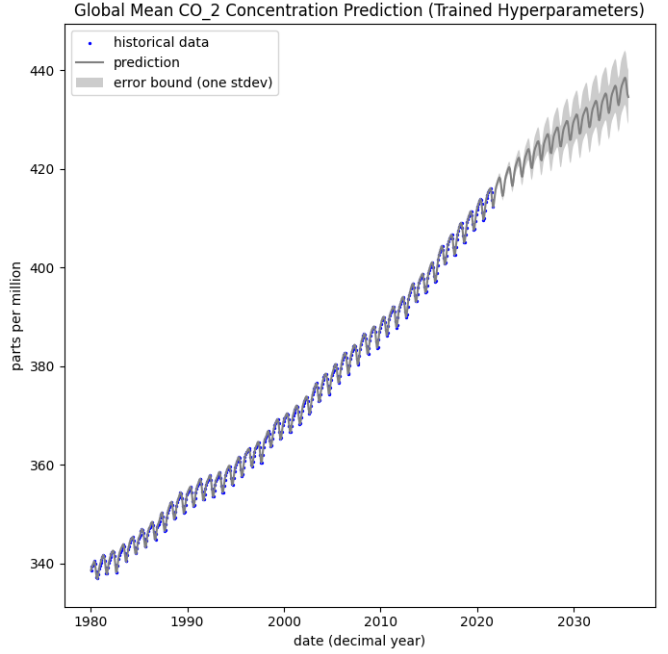


Figure 12: Trained extrapolation

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 and our Bayesian linear regression model assumption. Moreover, the concentration continues to exhibit yearly seasonality (for the trained extrapolation) as we would expect through the choice of our kernel for modelling the residuals $g(t)$. We can see that the conclusions can be quite sensitive to kernel hyperparameters when comparing the extrapolations from before and after training. Prior to training, the extrapolated prediction is not representative of the given data, with very large uncertainties. 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 MAP 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     """
9     Parameters for linear regression
10    """
11
12    mean: np.ndarray # weight vector (1, number of features)
13    covariance: np.ndarray # covariance matrix on mean (number of features, number of features)
14
15    @property
16    def precision(self) -> np.ndarray:
17        return np.linalg.inv(self.covariance)
18
19    def predict(self, x: np.ndarray) -> np.ndarray:
20        """
21        Linear regression prediction.
22
23        :param x: design matrix (number of features, number of data points)
24        :return: predicted response matrix (1, number of data points)
25        """
26        return self.mean.T @ x
27
28
29 @dataclass
30 class Theta:
31     linear_regression_parameters: LinearRegressionParameters
32     sigma: float
33
34     @property
35     def variance(self) -> float:
36         return self.sigma**2
37
38     @property
39     def precision(self) -> float:
40         return 1 / self.variance
41
42
43 def compute_linear_regression_posterior(
44     x: np.ndarray,
45     y: np.ndarray,
46     prior_linear_regression_parameters: LinearRegressionParameters,
47     residuals_precision: float,
48 ) -> LinearRegressionParameters:
49     """
50     Compute the parameters of the posterior distribution on the linear regression weights
51
52     :param x: design matrix (number of features, number of data points)
53     :param y: response matrix (1, number of data points)
54     :param prior_linear_regression_parameters: parameters for the prior distribution on the linear regression
55     weights
56     :param residuals_precision: the precision of the residuals of the linear regression
57     :return: parameters for the posterior distribution on the linear regression weights
58     """
59     posterior_covariance = np.linalg.inv(
60         residuals_precision * x @ x.T + prior_linear_regression_parameters.precision
61     )
62     posterior_mean = posterior_covariance @ (
63         residuals_precision * x @ y.T
64         + prior_linear_regression_parameters.precision
65         @ prior_linear_regression_parameters.mean
66     )
67     return LinearRegressionParameters(
68         mean=posterior_mean, covariance=posterior_covariance
69     )
```

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

```

```

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

```

```

191 ) -> None:
192 # Train Bayesian Linear Regression
193 x_train = construct_design_matrix(t_train)
194 prior_theta = Theta(
195     linear_regression_parameters=prior_linear_regression_parameters,
196     sigma=linear_regression_sigma,
197 )
198 posterior_linear_regression_parameters = compute_linear_regression_posterior(
199     x_train,
200     y_train,
201     prior_linear_regression_parameters,
202     residuals_precision=prior_theta.precision,
203 )
204
205 residuals = y_train - posterior_linear_regression_parameters.predict(x_train)
206 gaussian_process = GaussianProcess(
207     kernel, t_train.reshape(-1, 1), residuals.reshape(-1)
208 )
209
210 # Prediction
211 x_test = construct_design_matrix(t_test)
212 linear_prediction = posterior_linear_regression_parameters.predict(x_test).reshape(
213     -1
214 )
215 mean_prediction, covariance_prediction = gaussian_process.posterior_distribution(
216     t_test.reshape(-1, 1), **asdict(gaussian_process.parameters)
217 )
218
219 # Plot
220 plt.figure(figsize=(7, 7))
221 plt.scatter(
222     t_train + min_year,
223     y_train.reshape(-1),
224     s=2,
225     color="blue",
226     label="historical data",
227 )
228 plt.plot(
229     t_test + min_year,
230     linear_prediction + mean_prediction,
231     color="gray",
232     label="prediction",
233 )
234 plt.fill_between(
235     t_test + min_year,
236     linear_prediction
237     + mean_prediction
238     - 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
239     linear_prediction
240     + mean_prediction
241     + 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
242     facecolor=(0.8, 0.8, 0.8),
243     label="error bound (one stdev)",
244 )
245 plt.xlabel("date (decimal year)")
246 plt.ylabel("parts per million")
247 plt.title("Global Mean CO2 Concentration Prediction (Untrained Hyperparameters)")
248 plt.legend()
249 plt.tight_layout()
250 plt.savefig(save_path + "--extrapolation-untrained", bbox_inches="tight")
251 plt.close()
252
253 df_parameters = pd.DataFrame(
254     [
255         [
256             x.strip("log-") + " (kernel)",
257             "%2E" % Decimal(np.exp(gaussian_process.parameters.kernel[x])),
258         ]
259         for x in gaussian_process.parameters.kernel.keys()
260     ]
261     + [[ "sigma", "%2E" % Decimal(float(gaussian_process.parameters.sigma)) ]],
262     columns=["parameter", "value"],
263 )
264 df_parameters = df_parameters.set_index("parameter").sort_values(by=["parameter"])
265 dfi.export(df_parameters, save_path + "--untrained-parameters.png")
266
267 # Train Gaussian Process Regression (Hyperparameter Tune)
268 optimizer = optax.adam(learning_rate)
269 gaussian_process.parameters = gaussian_process.train(
270     optimizer, number_of_iterations, **asdict(gaussian_process.parameters)
271 )
272 df_parameters = pd.DataFrame(
273     [
274         [
275             x.strip("log-") + " (kernel)",
276             "%2E" % Decimal(np.exp(gaussian_process.parameters.kernel[x])),
277         ]
278         for x in gaussian_process.parameters.kernel.keys()
279     ]
280     + [[ "sigma", "%2E" % Decimal(float(gaussian_process.parameters.sigma)) ]],
281     columns=["parameter", "value"],
282 )
283 df_parameters = df_parameters.set_index("parameter").sort_values(by=["parameter"])
284 dfi.export(df_parameters, save_path + "--trained-parameters.png")
285
286 # Prediction

```

```

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

```

src/solutions/q2.py

Question 3: Mean-field learning

(a)

The free energy is can be expressed 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 is:

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

where $q_k(s_k) = (\lambda_k)^{s_k} (1 - \lambda_k)^{(1-s_k)}$.

Given that:

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

we can substitute the appropriate terms:

$$P(\mathbf{x}|\mathbf{s}, \theta) = 2\pi^{-\frac{D}{2}} |\sigma^2 \mathbf{I}|^{-\frac{1}{2}} \exp\left(-\frac{1}{2} \left(\mathbf{x} - \sum_{k=1}^K s_k \mu_k\right)^T \frac{1}{\sigma^2} \mathbf{I} \left(\mathbf{x} - \sum_{k=1}^K s_k \mu_k\right)\right)$$

with d being the number of dimensions.

Taking the logarithm:

$$\log P(\mathbf{x}|\mathbf{s}, \theta) = -\frac{D}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{k=1}^K s_k \mu_k + \sum_{k=1}^K \sum_{k'=1}^K s_k s_{k'} \mu_k^T \mu_{k'} \right)$$

The expectation distributed to the relevant terms:

$$\langle \log P(\mathbf{x}|\mathbf{s}, \theta) \rangle_{q(\mathbf{s})} = -\frac{D}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{k=1}^K \langle s_k \rangle_{q_k(s_k)} \mu_k + \sum_{k=1}^K \sum_{k'=1}^K \langle s_k s_{k'} \rangle_{q_k(s_k) q_{k'}(s_{k'})} \mu_k^T \mu_{k'} \right)$$

Evaluating the expectations:

$$\langle \log P(\mathbf{x}|\mathbf{s}, \theta) \rangle_{q(\mathbf{s})} = -\frac{D}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{k=1}^K \lambda_k \mu_k + \sum_{k=1}^K \sum_{k'=1, k' \neq k}^K \lambda_k \lambda_{k'} \mu_k^T \mu_{k'} + \sum_{k=1}^K \lambda_k \mu_k^T \mu_k \right)$$

where $\langle s_k s_{k'} \rangle_{q_k(s_k)} = \langle s_k \rangle_{q_k(s_k)}$ because $s_k \in \{0, 1\}$.

Given that:

$$P(\mathbf{s}|\theta) = \prod_{k=1}^K \pi_k^{s_k} (1 - \pi_k)^{(1-s_k)}$$

Taking the logarithm:

$$\log P(\mathbf{s}|\theta) = \sum_{k=1}^K s_k \log \pi_k + (1 - s_k) \log(1 - \pi_k)$$

The expectation distributed to the relevant terms:

$$\langle \log P(\mathbf{s}|\theta) \rangle_{q(\mathbf{s})} = \sum_{k=1}^K \langle s_k \rangle_{q_k(s_k)} \log \pi_k + (1 - \langle s_k \rangle_{q_k(s_k)}) \log(1 - \pi_k)$$

Evaluating the expectations with respect to $q(\mathbf{s})$:

$$\langle \log P(\mathbf{s}|\theta) \rangle_{q(\mathbf{s})} = \sum_{k=1}^K \lambda_k \log \pi_k + (1 - \lambda_k) \log(1 - \pi_k)$$

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

$$H[q(\mathbf{s})] = \sum_{k=1}^K H[q_k(s_k)]$$

Thus,

$$H[q(\mathbf{s})] = - \sum_{k=1}^K \sum_{s_k \in \{0,1\}} q_k(s_k) \log q_k(s_k)$$

Substituting the appropriate values:

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

Combining, we have our free energy expression:

$$\begin{aligned} \mathcal{F}(q, \theta) = & \frac{-D}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{k=1}^K \lambda_k \mu_k + \sum_{k=1}^K \sum_{k'=1, k' \neq k}^K \lambda_k \lambda_{k'} \mu_k^T \mu_{k'} + \sum_{k=1}^K \lambda_k \mu_k^T \mu_k \right) \\ & + \sum_{k=1}^K \lambda_k \log \pi_k + (1 - \lambda_k) \log(1 - \pi_k) \\ & - \sum_{k=1}^K \lambda_k \log \lambda_k + (1 - \lambda_k) \log(1 - \lambda_k) \end{aligned}$$

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

$$\frac{\partial}{\partial q_k} \left(\mathcal{F}(q, \theta) + \lambda^{LG} \int q_k - 1 \right) = \langle \log P(\mathbf{x}, \mathbf{s}|\theta) \rangle_{\prod_{k' \neq k} q_{k'}(s_{k'})} - \log q_k(s_k) - 1 + \lambda^{LG}$$

where λ^{LG} is the Lagrange multiplier.

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

$$\log q_k(s_k) = \langle \log P(\mathbf{x}, \mathbf{s} | \theta) \rangle_{\prod_{k' \neq k} q_{k'}(s_{k'})} - 1 + \lambda^{LG}$$

Similar to our free energy derivation:

$$\langle \log P(\mathbf{x} | \mathbf{s}, \theta) \rangle_{\prod_{k' \neq k} q_{k'}(s_{k'})} \propto -\frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{k=1}^K \langle s_k \rangle_{\prod_{k' \neq k} q_{k'}(s_{k'})} \mu_k + \sum_{k=1}^K \sum_{k'=1}^K \langle s_k s_{k'} \rangle_{\prod_{k' \neq k} q_{k'}(s_{k'})} \right)$$

and

$$\langle \log P(\mathbf{s} | \theta) \rangle_{\prod_{k' \neq k} q_{k'}(s_{k'})} = \sum_{k=1}^K \langle s_k \rangle_{\prod_{k' \neq k} q_{k'}(s_{k'})} \log \pi_k + (1 - \langle s_k \rangle_{\prod_{k' \neq k} q_{k'}(s_{k'})}) \log(1 - \pi_k)$$

We can write:

$$\log q_k(s_k) \propto \log P(\mathbf{x} | \mathbf{s}, \theta)_{\prod_{k' \neq k} q_{k'}(s_{k'})} + \langle \log P(\mathbf{s} | \theta) \rangle_{\prod_{k' \neq k} q_{k'}(s_{k'})}$$

Substituting the relevant terms:

$$\log q_k(s_k) \propto -\frac{1}{2\sigma^2} \left(-2s_k \mathbf{x}^T \mu_k + s_k s_k \mu_k^T \mu_k + 2 \sum_{k'=1, k' \neq k}^K s_k \lambda_{k'} \mu_k^T \mu_{k'} \right) + s_k \log \pi_k + (1 - s_k) \log(1 - \pi_k)$$

Knowing $\log q_k(s_k) = s_k \log \lambda_k + (1 - s_k) \log(1 - \lambda_k)$:

$$\log q_k(s_k) \propto s_k \log \frac{\lambda_k}{1 - \lambda_k}$$

Thus,

$$s_k \log \frac{\lambda_k}{1 - \lambda_k} \propto -\frac{1}{2\sigma^2} \left(-2s_k \mathbf{x}^T \mu_k + s_k s_k \mu_k^T \mu_k + 2 \sum_{k'=1, k' \neq k}^K s_k \lambda_{k'} \mu_k^T \mu_{k'} \right) + s_k \log \frac{\pi_k}{1 - \pi_k}$$

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

$$s_k \log \frac{\lambda_k}{1 - \lambda_k} \propto -\frac{1}{2\sigma^2} \left(-2s_k \mathbf{x}^T \mu_k + s_k \mu_k^T \mu_k + 2 \sum_{k'=1, k' \neq k}^K s_k \lambda_{k'} \mu_k^T \mu_{k'} \right) + s_k \log \frac{\pi_k}{1 - \pi_k}$$

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

$$s_k \log \frac{\lambda_k}{1 - \lambda_k} = \frac{s_k \mu_k^T}{2\sigma^2} \left(2\mathbf{x} - \mu_k - 2 \sum_{k'=1, k' \neq k}^K \lambda_{k'} \mu_{k'} \right) + s_k \log \frac{\pi_k}{1 - \pi_k}$$

Solving for λ_k :

$$\lambda_k = \frac{1}{1 + \exp \left[- \left(\frac{\mu_k^T}{\sigma^2} \left(\mathbf{x} - \frac{\mu_k}{2} - \sum_{k'=1, k' \neq k}^K \lambda_{k'} \mu_{k'} \right) + \log \frac{\pi_k}{1 - \pi_k} \right) \right]}$$

we have our partial update for $q_k(s_k)$

(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:

$$\hat{\beta} = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{Y}$$

for the linear regression problem $\mathbf{Y} = \mathbf{X}^T \beta$ where β corresponds to our mean parameters μ , the design matrix \mathbf{X} corresponds to our input \mathbf{s} and the response Y corresponds to the image pixels we denoted as \mathbf{x} . This makes sense because our resulting images \mathbf{x} are modeled as linear combinations of features μ , weighted by \mathbf{s} , $\mathbf{x} = \mu \mathbf{s}$ where $\mathbf{x} \in \mathbb{R}^{D \times 1}$, $\mu \in \mathbb{R}^{D \times K}$, and $\mathbf{s} \in \mathbb{R}^{K \times 1}$.

(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 $\mathcal{O}(K^2 N)$
 - 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 $\mathcal{O}(D^2 N)$
 - The dot product $\mu^T \mu$ where $\mu \in \mathbb{R}^{D \times K}$ is $\mathcal{O}(K^2 D)$
 - 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 $\mathcal{O}(K^3)$
 - The dot product $\text{ES}^T \mathbf{x}$ where $\text{ES} \in \mathbb{R}^{N \times K}$ and $\mathbf{x} \in \mathbb{R}^{N \times D}$ is $\mathcal{O}(KND)$
 - The dot product $(\text{ES}^T \mathbf{x}) \mu$ where $\text{ES}^T \mathbf{x} \in \mathbb{R}^{K \times D}$ and $\mu \in \mathbb{R}^{D \times K}$ is $\mathcal{O}(K^2 D)$
- 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 $\mathcal{O}(K^3 + K^2 N + KND + D^2 N + K^2 D)$ where we do not assume that any of N , K , or D is large compared to the others.

(d)

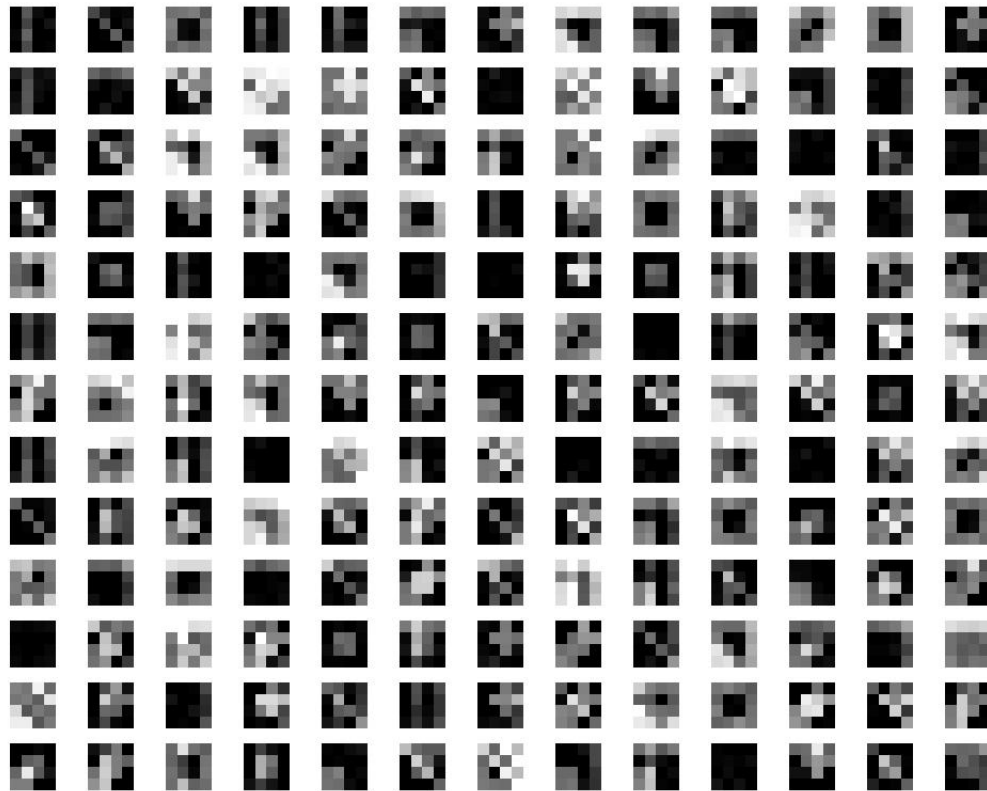


Figure 13: Images generated by randomly combined features with noise

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 $\epsilon \sim \mathcal{N}(0, \Psi)$ and $\mathbf{s} \sim \mathcal{N}(0, \mathbf{I})$. 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.

A mixture of Gaussians assumes as model:

$$\mathbf{x} = \sum_{k=1}^K s_k \mu_k + \epsilon \text{ and } \sum_{k=1}^K s_k = 1$$

where $\epsilon \sim \mathcal{N}(0, \Sigma_\epsilon)$. This also wouldn't be appropriate because it assumes a dirichlet distribution on the vector \mathbf{s} . Our data is constructed by having each s_k independently sampled from a Bernoulli distribution indicating the presence of feature μ_k .

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 $P(s_k)$ are our independent Bernoulli distributions and we are linearly combining different features with $\mathbf{W} \in \mathbb{R}^{D \times K}$ and then adding noise ϵ .

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

(e)

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

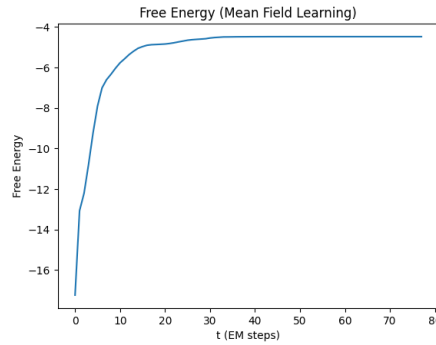


Figure 14: Free Energy

(f)

The initialised features:

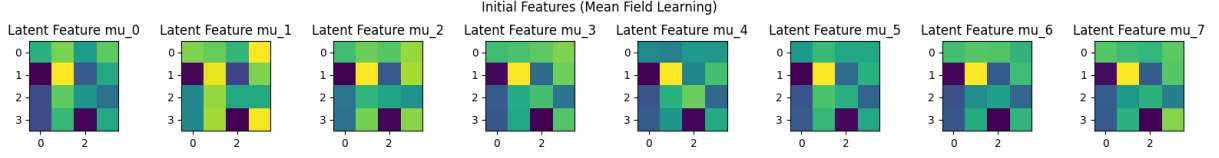


Figure 15: Initial Latent Factors

The features learned by the algorithm:

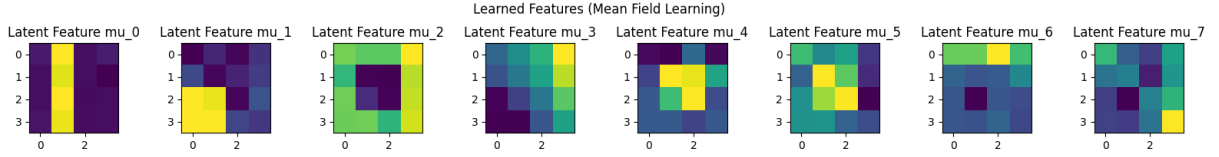


Figure 16: Learned Latent Factors

We can see that this approach has learned some of the previously identified features, such as the vertical line in the second column, the two by two square in the bottom left corner, the border, the horizontal line in the first row, 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 the model with the highest free energy convergence). We can also increase the number of data samples, with the hopes of learning better features. Finally, we can perform Variational Bayes by setting priors on π , σ^2 , and μ for better estimation of these parameters.

When implementing the algorithm, the mean field parameters were initialised randomly, each independently from a uniform distribution on $[0, 1]$. π , σ , and μ were initialised by running a maximisation step using these randomly initialised mean field parameters. K was set to eight, after visually identifying eight features in part d. Moreover, for computational stability, when calculating $\log(\lambda_i)$ and $\log(1 - \lambda_i)$, we manually shifted λ_i by $1e^{-10}$ to prevent $\lambda_i = 0$ or $\lambda_i = 1$.

(g)

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

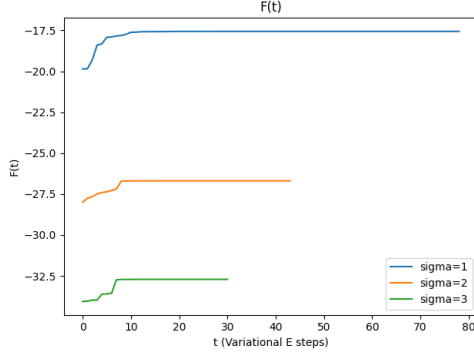


Figure 17: Free energy vs σ

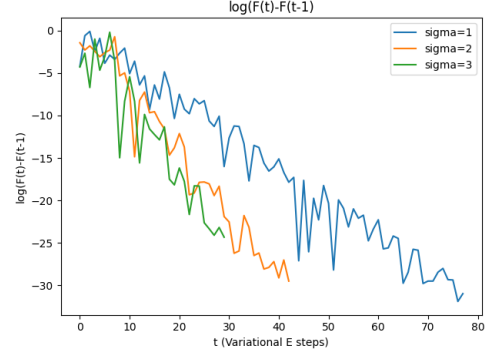


Figure 18: Free energy convergence vs σ

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 by the parameters π , σ , and μ and we adjust our approximation q with parameter λ to try to reach this upper bound by increasing $\mathcal{F}(q, \theta)$. 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 reaches the upper bound faster.

The Python code for the binary latent factor model:

```

1  from typing import Tuple
2
3  import numpy as np
4
5  from demo_code.m_step import m_step
6  from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
7      AbstractBinaryLatentFactorApproximation,
8  )
9  from src.models.binary_latent_factor_models.abstract_binary_latent_factor_model import (
10     AbstractBinaryLatentFactorModel,
11 )
12
13
14 class BinaryLatentFactorModel(AbstractBinaryLatentFactorModel):
15     def __init__(
16         self,
17         mu: np.ndarray,
18         sigma: float,
19         pi: np.ndarray,
20     ):
21         self._mu = mu # (number_of_dimensions, number_of_latent_variables)
22         self._sigma = sigma
23         self._pi = pi # (1, number_of_latent_variables)
24
25     @property
26     def mu(self):
27         return self._mu
28
29     @mu.setter
30     def mu(self, value):
31         self._mu = value
32
33     @property
34     def sigma(self):
35         return self._sigma
36
37     @sigma.setter
38     def sigma(self, value):
39         self._sigma = value
40
41     @property
42     def pi(self):
43         return self._pi
44
45     @pi.setter
46     def pi(self, value):
47         self._pi = value
48
49     @property
50     def variance(self) -> float:
51         return self._sigma**2
52
53     @staticmethod
54     def calculate_maximisation_parameters(
55         x: np.ndarray,
56         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
57     ) -> Tuple[np.ndarray, float, np.ndarray]:
58         return m_step(
59             x=x,
60             es=binary_latent_factor_approximation.expectation_s,
61             ess=binary_latent_factor_approximation.expectation_ss,
62         )
63
64     def maximisation_step(
65         self,
66         x: np.ndarray,
67         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
68     ) -> None:
69         mu, sigma, pi = self.calculate_maximisation_parameters(
70             x, binary_latent_factor_approximation
71         )
72         self.mu = mu
73         self.sigma = sigma
74         self.pi = pi
75
76
77 def init_binary_latent_factor_model(
78     x: np.ndarray,
79     binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
80 ) -> BinaryLatentFactorModel:
81     """
82     Initialise by running a maximisation step with the parameters of the binary latent factor approximation
83
84     :param x: data matrix (number_of_points, number_of_dimensions)
85     :param binary_latent_factor_approximation: a binary_latent_factor_approximation
86     :return: an initialised binary latent factor model
87     """
88     mu, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
89         x, binary_latent_factor_approximation
90     )
91     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     def __init__(
15         self, lambda_matrix: np.ndarray, max_steps: int, convergence_criterion: float
16     ):
17         self._lambda_matrix = lambda_matrix
18         self.max_steps = max_steps
19         self.convergence_criterion = convergence_criterion
20
21     @property
22     def lambda_matrix(self) -> np.ndarray:
23         """
24         lambda_matrix: parameters variational approximation (number_of_points, number_of_latent_variables)
25         """
26         return self._lambda_matrix
27
28     @lambda_matrix.setter
29     def lambda_matrix(self, value):
30         self._lambda_matrix = value
31
32     def lambda_matrix_exclude(self, exclude_latent_index: int) -> np.ndarray:
33         # (number_of_points, number_of_latent_variables-1)
34         return np.concatenate(
35             (
36                 self.lambda_matrix[:, :exclude_latent_index],
37                 self.lambda_matrix[:, exclude_latent_index + 1 :],
38             ),
39             axis=1,
40         )
41
42     def _partial_expectation_step(
43         self,
44         x: np.ndarray,
45         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
46         latent_factor: int,
47     ) -> np.ndarray:
48         """Partial Variational E step for factor i for all data points
49
50         :param x: data matrix (number_of_points, number_of_dimensions)
51         :param binary_latent_factor_model: a binary latent factor model
52         :param latent_factor: latent factor to compute partial update
53         :return: lambda_vector: new lambda parameters for the latent factor (number_of_points, 1)
54         """
55         lambda_matrix_excluded = self.lambda_matrix_exclude(latent_factor)
56         mu_excluded = binary_latent_factor_model.mu_excluded(latent_factor)
57
58         mu_latent = binary_latent_factor_model.mu[:, latent_factor]
59         # (number_of_points, 1)
60         partial_expectation_log_p_x_given_s_theta_proportion = (
61             binary_latent_factor_model.precision
62             * (
63                 x # (number_of_points, number_of_dimensions)
64                 - 0.5 * mu_latent.T # (1, number_of_dimensions)
65                 - lambda_matrix_excluded # (number_of_points, number_of_latent_variables-1)
66                 @ mu_excluded.T # (number_of_latent_variables-1, number_of_dimensions)
67             )
68             @ mu_latent # (number_of_dimensions, 1)
69         )
70
71         # (1, 1)
72         partial_expectation_log_p_s_given_theta_proportion = np.log(
73             binary_latent_factor_model.pi[0, latent_factor]
74             / (1 - binary_latent_factor_model.pi[0, latent_factor])
75         )
76
77         # (number_of_points, 1)
78         partial_expectation_log_p_x_s_given_theta_proportion = (
79             partial_expectation_log_p_x_given_s_theta_proportion
80             + partial_expectation_log_p_s_given_theta_proportion
81         )
82
83         # (number_of_points, 1)
84         lambda_vector = 1 / (
85             1 + np.exp(-partial_expectation_log_p_x_s_given_theta_proportion)
86         )
87         lambda_vector[lambda_vector == 0] = 1e-10
88         lambda_vector[lambda_vector == 1] = 1 - 1e-10
89         return lambda_vector
90
91     def variational_expectation_step(
92         self, x: np.ndarray, binary_latent_factor_model: AbstractBinaryLatentFactorModel
93     ) -> List[float]:
94         """Variational E step

```

```

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

```

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 List, Tuple
4
5 import numpy as np
6
7 from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
8     AbstractBinaryLatentFactorApproximation,
9 )
10 from src.models.binary_latent_factor_models.binary_latent_factor_model import (
11     AbstractBinaryLatentFactorModel,
12 )
13
14
15 def is_converge(
16     free_energies: List[float],
17     current_lambda_matrix: np.ndarray,
18     previous_lambda_matrix: np.ndarray,
19 ) -> bool:
20     """
21     Check for convergence of free energy and lambda matrix
22
23     :param free_energies: list of free energies
24     :param current_lambda_matrix: current lambda matrix
25     :param previous_lambda_matrix: previous lambda matrix
26     :return: boolean indicating convergence
27     """
28     return (abs(free_energies[-1] - free_energies[-2]) == 0) and np.linalg.norm(
29         current_lambda_matrix - previous_lambda_matrix
30     ) == 0
31
32
33 def learn_binary_factors(
34     x: np.ndarray,
35     em_iterations: int,
36     binary_latent_factor_model: AbstractBinaryLatentFactorModel,
37     binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
38 ) -> Tuple[
39     AbstractBinaryLatentFactorApproximation,
40     AbstractBinaryLatentFactorModel,
41     List[float],
42 ]:
43     """
44     Expectation maximisation algorithm to learn binary factors.
45
46     :param x: data matrix (number-of-points, number-of-dimensions)
47     :param em_iterations: number of iterations to run EM
48     :param binary_latent_factor_model: a binary_latent_factor_model
49     :param binary_latent_factor_approximation: a binary_latent_factor_approximation
50     :return: a Tuple containing the updated binary_latent_factor_model, updated
51             binary_latent_factor_approximation,
52             and free energies during each step of EM
53     """
54     free_energies: List[float] = [
55         binary_latent_factor_approximation.compute_free_energy(
56             x, binary_latent_factor_model
57         )
58     ]
59     for _ in range(em_iterations):
60         previous_lambda_matrix = np.copy(
61             binary_latent_factor_approximation.lambda_matrix
62         )
63
64         # E step
65         binary_latent_factor_approximation.variational_expectation_step(
66             x=x,
67             binary_latent_factor_model=binary_latent_factor_model,
68         )
69
70         # M step
71         binary_latent_factor_model.maximisation_step(
72             x,
73             binary_latent_factor_approximation,
74         )
75
76         free_energies.append(
77             binary_latent_factor_approximation.compute_free_energy(
78                 x, binary_latent_factor_model
79             )
80         )
81         if is_converge(
82             free_energies,
83             binary_latent_factor_approximation.lambda_matrix,
84             previous_lambda_matrix,
85         ):
86             break
87     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: Variational Bayes for binary factors

(a)

To derive a Variational Bayesian hyperparameter optimisation algorithm to automatically determine K , the number of hidden binary variables in this model, we begin by writing the expression for x_d :

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

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 express the posterior as:

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

where we introduce priors on each \mathbf{w}_d 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:

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

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

$$\frac{\partial}{\partial q_k} \left(\exp \langle \log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha) \rangle_{q(\mu)} + \lambda^{LG} \int q_k - 1 \right) \propto \exp \langle \log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha) \rangle_{q(\mu)} \prod_{k' \neq k} q_{k'}(s_{k'}) - \log q_k(s_k)$$

Setting this to zero we can solve for λ_i where $q_k(s_k) = \lambda_k^{s_k} (1 - \lambda_k)^{(1-s_k)}$:

$$\lambda_k = \frac{1}{1 + \exp \left[- \left(\frac{\langle \mu_k \rangle_{q_{\mu_k}}^T}{\sigma^2} \left(\mathbf{x} - \frac{\langle \mu_k \rangle_{q_{\mu_k}}}{2} - \sum_{k'=1, k' \neq k}^K \lambda_{k'} \langle \mu_{k'} \rangle_{q_{\mu_{k'}}} \right) + \log \frac{\pi_k}{1 - \pi_k} \right) \right]}$$

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}) 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

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

Substituting the appropriate terms:

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

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 the concatenation of the k^{th} elements of $\mu_{\mathbf{w}_d} \in \mathbb{R}^{K \times 1}$ for $d \in \{1, \dots, D\}$

For ARD, we must also optimise α with a hyper-M step. We start by choosing $Ga(\alpha_k | a, b)$, 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 $\langle (w_{d,k})^2 \rangle_{q(\mathbf{w}_d)} = (\mu_{\mathbf{w}_{d,k}})^2 + \Sigma_{\mathbf{w}_{d,(k,k)}}$, the k^{th} element of $\mu_{\mathbf{w}_d}$ and element (k, k) of $\Sigma_{\mathbf{w}_d}$ respectively, 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 .

(b)

Running variational Bayes for different values of k , we can visualise the learned features μ_k and corresponding α_k^{-1} :

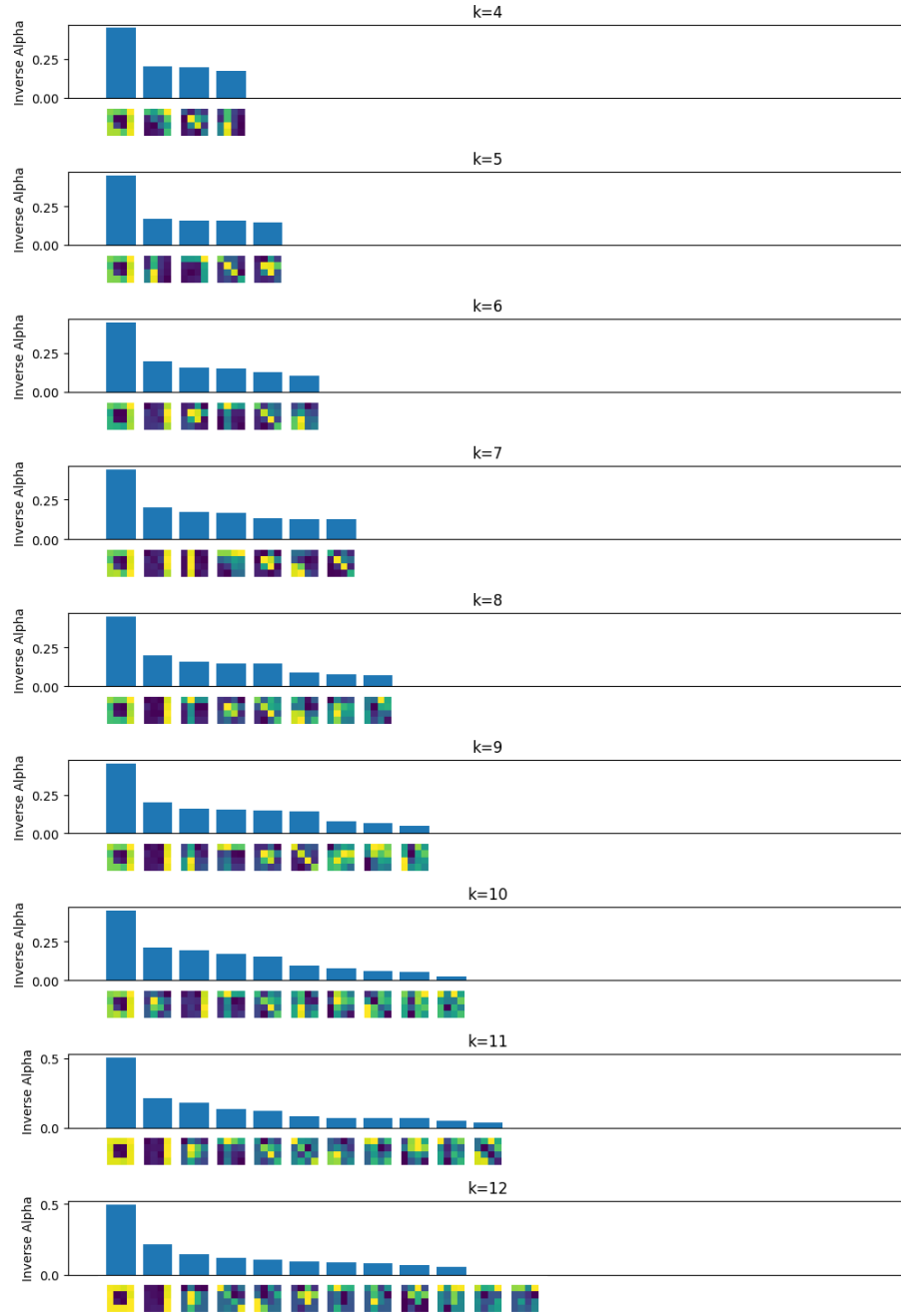


Figure 19: Learned Latent Factors vs Inverse Alpha



Figure 20: Learned Latent Factors vs Inverse Alpha

As we expect, when running the algorithm for higher K values, many of the features have $\alpha_k \rightarrow \infty$, depicted as $\alpha_k^{-1} \rightarrow 0$ for visual convenience. Moreover, visualising the learned features, we can see the clearest features often have the highest α_k^{-1} while the features deemed irrelevant are often noisy or duplicates.

Comparing the free energy plots of models trained on different K values:

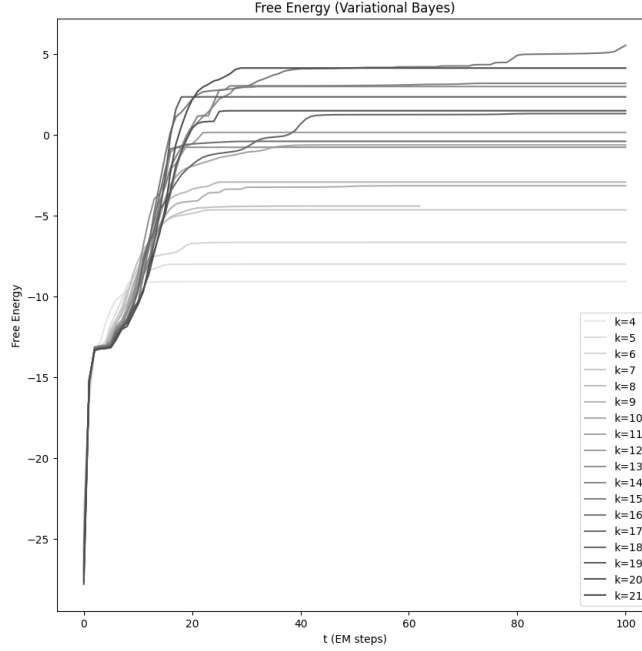


Figure 21: Free Energy for different values of k

We can see that initially for $K = 4$ to $K = 8$, increasing K significantly increases the convergence value of the free energy. However, beyond $K = 8$, the trend of K versus the free energy convergence value is not as clear. We can see that this corresponds to the visualisation of α^{-1} where beyond $K = 11$, the number of relevant features remains more or less the same. We know that there are only eight latent features, thus models with $K > 8$ should be learning duplicate or irrelevant features. As such, we wouldn't expect a model to be able to increase its free energy significantly when provided with additional degrees of freedom by increasing the value of K beyond eight. We see that for models with $K \gg 8$, there are typically ten or eleven features that might be deemed relevant (depending on how you threshold) and this is likely from slight overfitting, noise in the data, or duplicate features. Thus, the relationship between the free energy and the effective number of latent features for each model is as we would expect with ARD.

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     BinaryLatentFactorModel,
8 )
9 from src.models.binary_latent_factor_models.boltzmann_machine import BoltzmannMachine
10
11
12 class GaussianPrior:
13     def __init__(self, a: float, b: float, d: int, k: int):
14         """
15         Gaussian prior on mu matrix
16
17         :param a: alpha parameter of Gamma Prior
18         :param b: beta parameter of Gamma Prior
19         :param d: number of dimensions
20         :param k: number of latent variables
21         """
22         self.a = a
23         self.b = b
24         self.mu = np.zeros((d, k))
25         self.alpha = np.ones((k,))
26         self.w_covariance = np.zeros((k, k))
27
28     def mu_k(self, k: int) -> np.ndarray:
29         """
30         Column vector of mu matrix, the latent feature vector
31
32         :param k: latent factor index
33         :return: column vector (number_of_dimensions, 1)
34         """
35         return self.mu[:, k : k + 1]
36
37     def w_d(self, d: int) -> np.ndarray:
38         """
39         Row vector of mu matrix, the weight vector for a particular dimension (pixel) of the data
40
41         :param d: data dimension index
42         :return: row vector (1, number_of_latent_variables)
43         """
44         return self.mu[d : d + 1, :]
45
46     @property
47     def a_matrix(self) -> np.ndarray:
48         """
49         Precision matrix for a weight vector w_d
50         :return: matrix of shape (number_of_latent_variables, number_of_latent_variables)
51         """
52         return np.diag(self.alpha)
53
54
55 class VariationalBayes(BoltzmannMachine):
56     def __init__(self, mu: GaussianPrior, variance: float, pi: np.ndarray):
57         """
58         Variational Bayes implementation with prior on mu.
59         Note that we are inheriting from BoltzmannMachine for Question 5d only.
60
61         :param mu: Gaussian prior on latent features
62         :param variance: Gaussian noise parameter
63         :param pi: vector of priors (1, number_of_latent_variables)
64         """
65         super().__init__(mu=mu.mu, sigma=np.sqrt(variance), pi=pi)
66         self.gaussian_prior = mu
67         self._variance = variance
68         self._pi = pi
69
70     @property
71     def variance(self) -> float:
72         return self._variance
73
74     @property
75     def pi(self) -> np.ndarray:
76         return self._pi
77
78     @property
79     def mu(self) -> np.ndarray:
80         return self.gaussian_prior.mu
81
82     def _update_w_d_covariance(
83         self,
84         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
85     ):
86         # (number_of_latent_variables, number_of_latent_variables)
87         self.gaussian_prior.w_covariance = np.linalg.inv(
88             self.gaussian_prior.a_matrix
89             + self.precision * binary_latent_factor_approximation.expectation_ss
90         )
91
92     def _update_w_d_mean(
93         self,
94         x: np.ndarray, # (number_of_points, number_of_dimensions)

```

```

95     binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
96     d: int,
97 ) -> None:
98     """
99     Update mean vector for w_d.
100
101     :param x: data matrix (number_of_points, number_of_dimensions)
102     :param binary_latent_factor_approximation: a binary_latent_factor_approximation
103     :param d: index of data dimension to update
104     :return:
105     """
106
107     # (number_of_latent_variables, 1)
108     self.gaussian_prior.mu[d : d + 1, :] = (
109         self.gaussian_prior.w_covariance
110         @ (
111             self.precision
112             * binary_latent_factor_approximation.expectation_s.T
113             @ x[:, d : d + 1]
114         )
115     ).T
116
117 def _hyper_maximisation_step(self) -> None:
118     """
119     Hyper M step updating alpha, which parameterize the covariance matrix of the Gaussian prior on mu
120     """
121     for k in range(self.k):
122         self.gaussian_prior.alpha[k] = (2 * self.gaussian_prior.a + self.d - 2) / (
123             2 * self.gaussian_prior.b
124             + np.sum(self.gaussian_prior.mu_k(k) ** 2)
125             + self.d * self.gaussian_prior.w_covariance[k, k]
126         )
127
128 def maximisation_step(
129     self,
130     x: np.ndarray,
131     binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
132 ) -> None:
133     """
134     Maximisation step which runs the usual M-step followed by posterior updates to the
135     distribution of mu as well as a hyper M-step updating the prior parameters on mu, the alpha vector
136     :param x: data matrix (number_of_points, number_of_dimensions)
137     :param binary_latent_factor_approximation: a binary_latent_factor_approximation
138     """
139     _, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
140         x, binary_latent_factor_approximation
141     )
142     self._variance = sigma**2
143     self._pi = pi
144     self._update_w_d_covariance(binary_latent_factor_approximation)
145     for d in range(self.d):
146         self._update_w_d_mean(x, binary_latent_factor_approximation, d)
147     self._hyper_maximisation_step()

```

src/models/binary_latent_factor_models/variational_bayes.py

The Python code for Automatic Relevance Determination:

```
1 from typing import List, Tuple
2
3 import numpy as np
4
5 from src.expectation_maximisation import learn_binary_factors
6 from src.models.binary_latent_factor_approximations import AbstractBinaryLatentFactorApproximation,
7
8 )
9 from src.models.binary_latent_factor_models.binary_latent_factor_model import (
10     BinaryLatentFactorModel,
11 )
12 from src.models.binary_latent_factor_models.variational_bayes import (
13     GaussianPrior,
14     VariationalBayes,
15 )
16
17
18 def run_automatic_relevance_determination(
19     x: np.ndarray,
20     binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
21     a_parameter: float,
22     b_parameter: float,
23     k: int,
24     em_iterations: int,
25 ) -> Tuple[VariationalBayes, List[float]]:
26     """
27     Run automatic relevance determination with variational Bayes.
28
29     :param x: data matrix (number_of_points, number_of_dimensions)
30     :param binary_latent_factor_approximation: a binary_latent_factor_approximation
31     :param a_parameter: alpha parameter for gamma prior
32     :param b_parameter: beta parameter for gamma prior
33     :param k: number of latent variables
34     :param em_iterations: number of iterations to run EM
35     :return: a Tuple containing the optimised VB model and a list of free energies during each EM step
36     """
37     (_, sigma, pi) = BinaryLatentFactorModel.calculate_maximisation_parameters(
38         x, binary_latent_factor_approximation
39     )
40     mu = GaussianPrior(
41         a=a_parameter,
42         b=b_parameter,
43         k=k,
44         d=x.shape[1],
45     )
46     variational_bayes_model: VariationalBayes = VariationalBayes(
47         mu=mu,
48         variance=sigma**2,
49         pi=pi,
50     )
51     _, variational_bayes_model, free_energy = learn_binary_factors(
52         x=x,
53         em_iterations=em_iterations,
54         binary_latent_factor_model=variational_bayes_model,
55         binary_latent_factor_approximation=binary_latent_factor_approximation,
56     )
57     return variational_bayes_model, free_energy
```

src/automatic_relevance_determination.py

The rest of the Python code for question 4:

```
1 import os
2 from typing import List
3
4 import matplotlib.pyplot as plt
5 import numpy as np
6 from matplotlib.offsetbox import AnnotationBbox, OffsetImage
7
8 from src.automatic_relevance_determination import run_automatic_relevance_determination
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.variational_bayes import VariationalBayes
13
14
15 def offset_image(coord: int, path: str, ax: plt.Axes):
16     """
17     Add image to matplotlib axis.
18
19     :param coord: coordinate on axis
20     :param path: path to image
21     :param ax: plot axis
22     """
23     img = plt.imread(path)
24     im = OffsetImage(img, zoom=0.72)
25     im.image.axes = ax
26
27     ab = AnnotationBbox(
28         im,
29         (coord, 0),
30         xybox=(0.0, -19.0),
31         frameon=False,
32         xycoords="data",
33         boxcoords="offset points",
34         pad=0,
35     )
36     ax.add_artist(ab)
37
38
39 def plot_factors(
40     variational_bayes_binary_latent_factor_models: List[VariationalBayes],
41     ks: List[int],
42     max_k: int,
43     save_path: str,
44 ):
45     # store each feature as an image for later use
46     for i, k in enumerate(ks):
47         sort_indices = np.argsort(
48             variational_bayes_binary_latent_factor_models[i].gaussian_prior.alpha
49         )
50         for j, idx in enumerate(sort_indices):
51             fig = plt.figure(figsize=(0.3, 0.3))
52             ax = plt.Axes(fig, [0.0, 0.0, 1.0, 1.0])
53             ax.set_axis_off()
54             fig.add_axes(ax)
55             ax.imshow(
56                 variational_bayes_binary_latent_factor_models[i]
57                 .mu[:, idx]
58                 .reshape(4, 4)
59             )
60             fig.savefig(save_path + f"-latent-factor-{i}-{j}", bbox_inches="tight")
61             plt.close()
62
63     # bar plot of alphas
64     fig, ax = plt.subplots(len(ks), 1, figsize=(12, 2 * len(ks)))
65     plt.subplots_adjust(hspace=1)
66     for i, k in enumerate(ks):
67         sort_indices = np.argsort(
68             variational_bayes_binary_latent_factor_models[i].gaussian_prior.alpha
69         )
70         y = list(
71             1
72             /
73             variational_bayes_binary_latent_factor_models[i].gaussian_prior.alpha[
74                 sort_indices
75             ]
76             + [0] * (max_k - k)
77         )
78         ax[i].set_title(f"{k}")
79         ax[i].bar(range(max_k), y)
80         ax[i].set_xticks([])
81         ax[i].set_ylabel("Inverse Alpha")
82
83     # add feature image ticks
84     for i, k in enumerate(ks):
85         sort_indices = np.argsort(
86             variational_bayes_binary_latent_factor_models[i].gaussian_prior.alpha
87         )
88         for j in range(len(sort_indices)):
89             path = save_path + f"-latent-factor-{i}-{j}.png"
90             offset_image(j, path, ax[i])
91             os.remove(path)
92     fig.savefig(save_path + f"-latent-factors-comparison", bbox_inches="tight")
93     plt.close()
94
95 def b(
96     x: np.ndarray,
```

```

95     a_parameter: int,
96     b_parameter: int,
97     ks: List[int],
98     max_k: int,
99     em_iterations: int,
100     e_maximum_steps: int,
101     e_convergence_criterion: float,
102     save_path: str,
103 ) -> List[List[float]]:
104
105     variational_bayes_models: List[VariationalBayes] = []
106     free_energies = []
107     for i, k in enumerate(ks):
108         n = x.shape[0]
109         mean_field_approximation = init_mean_field_approximation(
110             k,
111             n,
112             max_steps=e_maximum_steps,
113             convergence_criterion=e_convergence_criterion,
114         )
115         (variational_bayes_model, free_energy) = run_automatic_relevance_determination(
116             x=x,
117             binary_latent_factor_approximation=mean_field_approximation,
118             a_parameter=a_parameter,
119             b_parameter=b_parameter,
120             k=k,
121             em_iterations=em_iterations,
122         )
123         variational_bayes_models.append(variational_bayes_model)
124         free_energies.append(free_energy)
125     plot_factors(
126         variational_bayes_models,
127         ks,
128         max_k,
129         save_path,
130     )
131     return free_energies
132
133
134 def free_energy_plot(
135     ks: List[int], free_energies: List[List[float]], model_name: str, save_path: str
136 ):
137     fig = plt.figure()
138     fig.set_figwidth(10)
139     fig.set_figheight(10)
140     shades = np.flip(np.linspace(0.3, 0.9, len(ks)))
141     for i, k in enumerate(ks):
142         plt.plot(free_energies[i], label=f"{k}", color=np.ones(3) * shades[i])
143     plt.title(f"Free Energy ({model_name})")
144     plt.xlabel("t (EM steps)")
145     plt.ylabel("Free Energy")
146     plt.legend()
147     plt.savefig(save_path + "-free-energy", bbox_inches="tight")
148     plt.close()

```

src/solutions/q4.py

Question 5: EP for the binary factor model

(a)

To derive an EP algorithm to infer marginals on the source variables in the binary latent factor model from Question 3, we first write 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 as:

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

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:

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

Thus,

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

we can define $\eta_{ii} = \log \left(\frac{\theta_{ii}}{1 - \theta_{ii}} \right)$:

$$\log \tilde{f}_i(s_i) \propto \eta_{ii} s_i$$

For $g_{ij}(s_i, s_j)$, we can choose a product of Bernoulli distributions for our 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 $\eta_{ji} = \log \left(\frac{\theta_{ji}}{1 - \theta_{ji}} \right)$ and $\eta_{ij} = \log \left(\frac{\theta_{ij}}{1 - \theta_{ij}} \right)$:

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

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

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

and the incoming message to node i from node j :

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

where $ne(j)$ are indices of neighbouring nodes of node j . This is the product of the incoming messages to node j from its neighbours, the singleton message from node j , and $\tilde{g}_{ji}(s_j, s_i)$ with all nodes except s_i marginalised out.

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

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

We are left with:

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

and

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

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

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

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 θ_{ii} in $\log \tilde{f}_i(s_i)$:

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

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

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

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

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

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:

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

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

We can write:

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

Simplifying:

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

Thus, the first moments:

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

and

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

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 :

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

Thus, the first moment:

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

Simplifying:

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

Similarly:

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

By setting:

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

and

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

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

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

Simplifying:

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

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

Our parameter update:

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

Similarly:

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

(c)

Using factored approximate messages, we see that:

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

Knowing $\eta_{ii} = \log \left(\frac{\theta_{ii}}{1 - \theta_{ii}} \right)$ and $\eta_{ki} = \log \left(\frac{\theta_{ki}}{1 - \theta_{ki}} \right)$:

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

and

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

The summation of the natural parameters of the singleton factor for node i with the natural parameters of messages from all the neighbouring nodes that aren't j .

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.

Moreover, our factored approximations:

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

and so:

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

our parameter for $q(s_i)$.

(d)

We can use automatic relevance determination (ARD) as a hyperparameter method to select relevant features by placing a prior on μ_k in the same way as Question 4. With a hyper-M step, certain features will have diverging precision, leaving us with the relevant features, from which we get our K . We can implement this by running the message passing algorithm (loopy BP) which was implemented for Question 6 for the expectation step and the VB algorithm for the maximisation step (along with its hyper-M Step) which was implemented for Question 4:

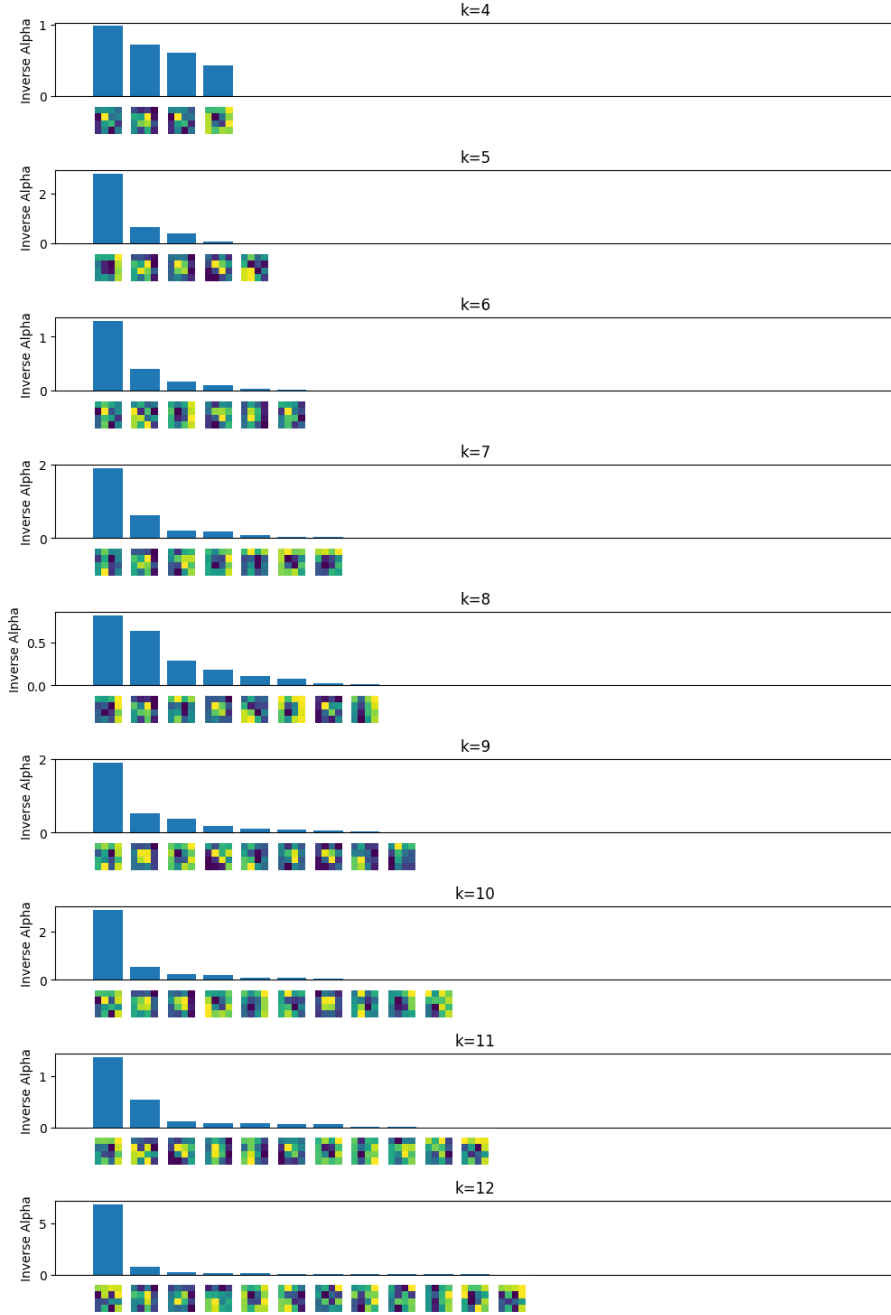


Figure 22: Learned Latent Factors vs Inverse Alpha

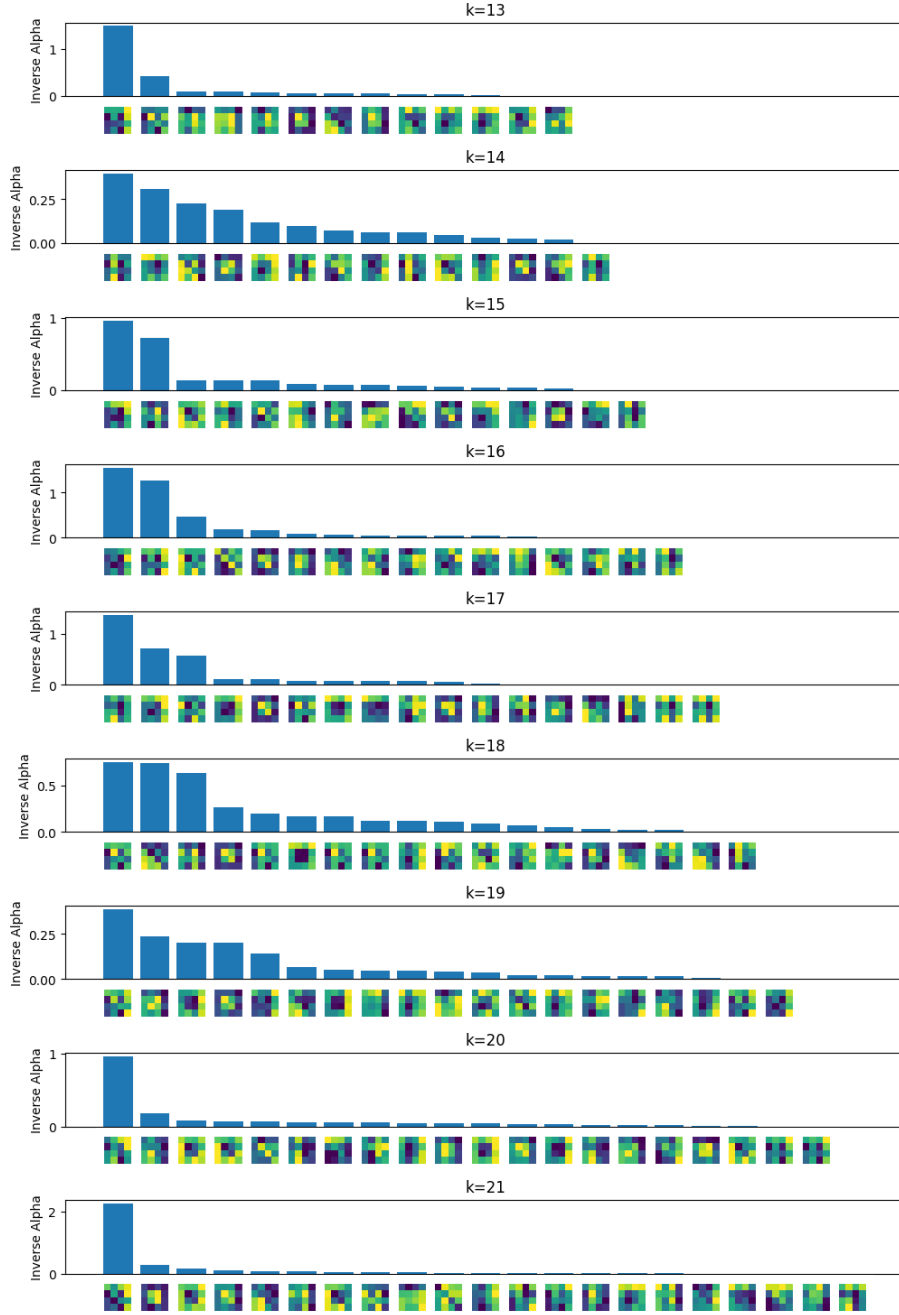


Figure 23: Learned Latent Factors vs Inverse Alpha

We can see that ARD has more difficulty finding relevant features when using loop BP. However, looking at the features themselves, we can see that the features are not very clear compared those from the mean field approximation or Variational Bayes in Question 4. This is probably why we do not consistently have the same number of α_k values that don't diverge as compared with Question 4. Moreover we usually have fewer than eight features that don't converge, likely because loopy BP seems to be unable to identify too many clear features that we would want to deem relevant in the first place. However, we can see for $K < 8$ that more of the features are deemed relevant as we would expect.

Comparing the free energy plots of models trained on different K values:

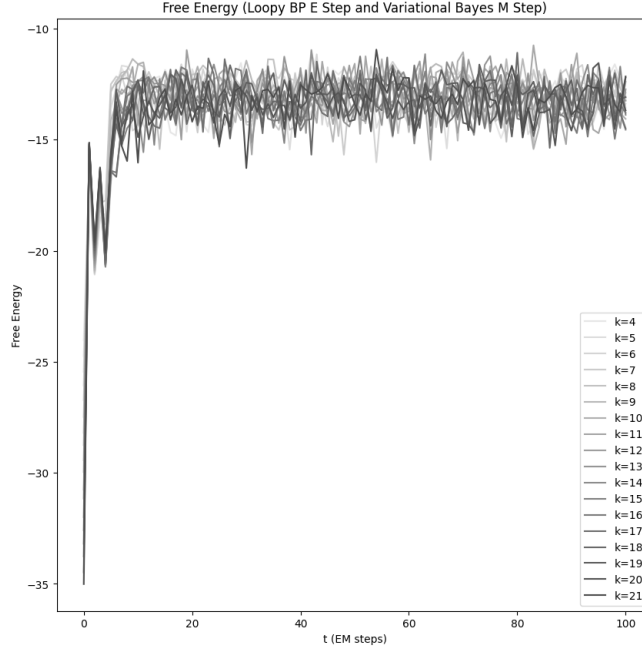


Figure 24: Free Energy for different values of k

We can see the difficulties of this approach due to the inability for loopy BP to converge for this particular problem. This likely contributes to the difficulty for ARD to find relevant features when the message passing algorithm is unable to converge on generating meaningful features. To improve this, we could also put priors on the other parameters σ^2 and π in the hopes that this will help stabilise the algorithm. Moreover, because loopy BP is unable to converge for this problem, it is quite computationally intensive as EM will always run the maximum number of iterations without improvement in the results. Knowing that it won't converge, employing early stopping or fewer iterations can also be helpful. Moreover, Boltzmann machines do not scale well with respect to the number of nodes, K , introducing computational difficulties. This was experienced when generating the above results, where for larger K values, loopy BP E steps took much longer to perform. To tackle this, a restricted Boltzmann Machine could be implemented to reduce the number of connections in the graph, alleviating these computational difficulties.

The Python code for question 5:

```
1 from typing import List
2
3 import numpy as np
4 from tqdm import tqdm
5
6 from src.automatic_relevance_determination import run_automatic_relevance_determination
7 from src.models.binary_latent_factor_approximations.message_passing_approximation import (
8     init_message_passing,
9 )
10 from src.models.binary_latent_factor_models.variational_bayes import VariationalBayes
11 from src.solutions.q4 import plot_factors
12
13
14 def d(
15     x: np.ndarray,
16     a_parameter: int,
17     b_parameter: int,
18     ks: List[int],
19     max_k: int,
20     em_iterations: int,
21     save_path: str,
22 ) -> List[List[float]]:
23
24     variational_bayes_models: List[VariationalBayes] = []
25     free_energies = []
26     for i, k in tqdm(enumerate(ks)):
27         n = x.shape[0]
28         message_passing_approximation = init_message_passing(k, n)
29         (variational_bayes_model, free_energy) = run_automatic_relevance_determination(
30             x=x,
31             binary_latent_factor_approximation=message_passing_approximation,
32             a_parameter=a_parameter,
33             b_parameter=b_parameter,
34             k=k,
35             em_iterations=em_iterations,
36         )
37         variational_bayes_models.append(variational_bayes_model)
38         free_energies.append(free_energy)
39     plot_factors(
40         variational_bayes_models,
41         ks,
42         max_k,
43         save_path,
44     )
45     return free_energies
```

src/solutions/q5.py

Question 6: EP/Loopy BP Implementation

Implementing the EP/loopy-BP algorithm, we can compare the learned latent factors with those of the variational mean-field algorithm:

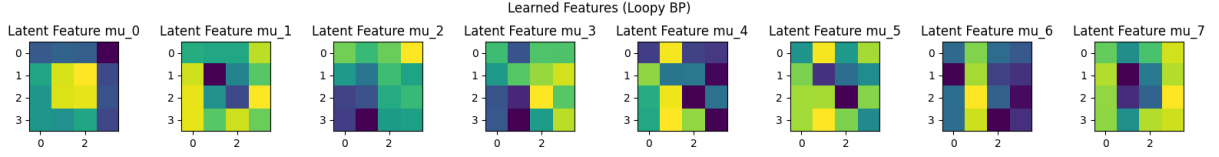


Figure 25: Learned Latent factors learned with EP/Loopy-BP

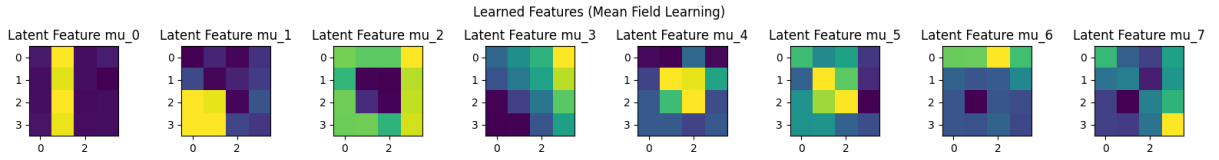


Figure 26: Learned Latent Factors with Mean Field Approximation

We can see that the mean field algorithm seems to learn better latent features. In particular there are fewer duplicates, unlike loopy BP that has a few duplicates. Moreover, the learned features of the mean field algorithm have less noise than those from loopy BP. For example μ_0 for the mean field algorithm looks almost like a binary image. We can understand the reason for this by comparing the free energies of the two algorithms:

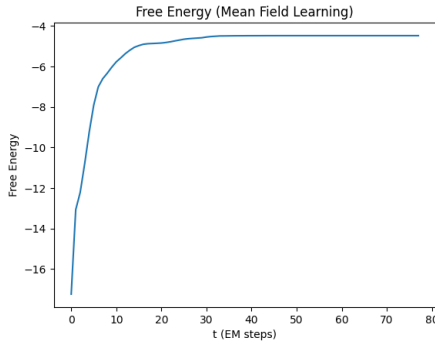


Figure 27: Mean Field Approximation

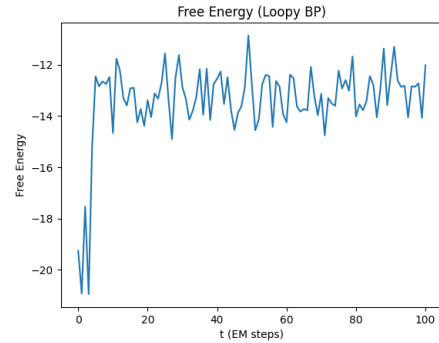


Figure 28: Loopy BP

We can observe that the free energy of the mean field algorithm converges while our loopy belief propagation is unable to converge to a free energy. Because loopy BP does not have convergence guarantees, one of the limitations of this approach, we can see that in this case loopy BP is unable to identify many latent factors.

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     def __init__(
13         self,
14         mu: np.ndarray,
15         sigma: float,
16         pi: np.ndarray,
17     ):
18         """
19         Binary latent factor model with Boltzmann Machine terms
20         """
21         super().__init__(mu, sigma, pi)
22
23     @property
24     def w_matrix(self) -> np.ndarray:
25         """
26         Weight matrix of the Boltzmann machine
27
28         :return: matrix of weights (number_of_latent_variables, number_of_latent_variables)
29         """
30         return -self.precision * (self.mu.T @ self.mu)
31
32     def w_matrix_index(self, i, j) -> float:
33         """
34         Weight matrix at a specific index
35
36         :param i: row index
37         :param j: column index
38         :return: weight value
39         """
40         return -self.precision * (self.mu[:, i] @ self.mu[:, j])
41
42     def b(self, x) -> np.ndarray:
43         """
44         b term in the Boltzmann machine for all data points
45
46         :param x: design matrix (number_of_points, number_of_dimensions)
47         :return: matrix of shape (number_of_points, number_of_latent_variables)
48         """
49         return -(
50             self.precision * x @ self.mu
51             + self.log_pi_ratio
52             - 0.5 * self.precision * np.multiply(self.mu, self.mu).sum(axis=0)
53         )
54
55     def b_index(self, x, node_index) -> float:
56         """
57         b term for a specific node in the Boltzmann machine for all data points
58
59         :param x: design matrix (number_of_points, number_of_dimensions)
60         :param node_index: node index
61         :return: vector of shape (number_of_points, 1)
62         """
63         return -(
64             self.precision * x @ self.mu[:, node_index]
65             + (self.log_pi[0, node_index] - self.log_one_minus_pi[0, node_index])
66             - 0.5 * self.precision * self.mu[:, node_index] @ self.mu[:, node_index]
67         ).reshape(-1, )
68
69
70
71     @property
72     def log_pi_ratio(self) -> np.ndarray:
73         return self.log_pi - self.log_one_minus_pi
74
75
76 def init_boltzmann_machine(
77     x: np.ndarray,
78     binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
79 ) -> BinaryLatentFactorModel:
80     """
81     Initialise by running a maximisation step with the parameters of the binary latent factor approximation
82
83     :param x: data matrix (number_of_points, number_of_dimensions)
84     :param binary_latent_factor_approximation: a binary_latent_factor_approximation
85     :return: an initialised Boltzmann machine model
86     """
87     mu, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
88         x, binary_latent_factor_approximation
89     )
90     return BoltzmannMachine(mu=mu, sigma=sigma, pi=pi)

```

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 MessagePassingApproximation(AbstractBinaryLatentFactorApproximation):
12     """
13     bernoulli_parameter_matrix (theta): matrix of parameters bernoulli_parameter_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, bernoulli_parameter_matrix: np.ndarray):
20         self.bernoulli_parameter_matrix = bernoulli_parameter_matrix
21
22     @property
23     def lambda_matrix(self) -> np.ndarray:
24         """
25         Aggregate messages and compute parameter for Bernoulli distribution
26         :return:
27         """
28         lambda_matrix = 1 / (1 + np.exp(-self.natural_parameter_matrix.sum(axis=1)))
29         lambda_matrix[lambda_matrix == 0] = 1e-10
30         lambda_matrix[lambda_matrix == 1] = 1 - 1e-10
31         return lambda_matrix
32
33     @property
34     def natural_parameter_matrix(self) -> np.ndarray:
35         """
36         The matrix containing natural parameters (eta) of each factor
37         off diagonals corresponds to  $\tilde{g}_{ij}$ ,  $\neg s_i(s_j)$  for data point n
38         diagonals correspond to  $\tilde{f}_{i}(s_i)$ 
39         (number_of_points, number_of_latent_variables, number_of_latent_variables)
40         :return:
41         """
42         return np.log(
43             np.divide(
44                 self.bernoulli_parameter_matrix, 1 - self.bernoulli_parameter_matrix
45             )
46         )
47
48     def aggregate_incoming_binary_factor_messages(
49         self, node_index: int, excluded_node_index: int
50     ) -> np.ndarray:
51         # (number_of_points, )
52         # exclude message from excluded_node_index -> node_index
53         return (
54             np.sum(
55                 self.natural_parameter_matrix[:, :excluded_node_index, node_index],
56                 axis=1,
57             )
58             + np.sum(
59                 self.natural_parameter_matrix[:, excluded_node_index + 1 :, node_index],
60                 axis=1,
61             )
62         ).reshape(
63             -1,
64         )
65
66     @staticmethod
67     def calculate_bernoulli_parameter(
68         natural_parameter_matrix: np.ndarray,
69     ) -> np.ndarray:
70         bernoulli_parameter = 1 / (1 + np.exp(-natural_parameter_matrix))
71         bernoulli_parameter[bernoulli_parameter == 0] = 1e-10
72         bernoulli_parameter[bernoulli_parameter == 1] = 1 - 1e-10
73         return bernoulli_parameter
74
75     def variational_expectation_step(
76         self, x: np.ndarray, binary_latent_factor_model: BoltzmannMachine
77     ) -> List[float]:
78         """
79         Iteratively update singleton and binary factors
80         :param x: data matrix (number_of_points, number_of_dimensions)
81         :param binary_latent_factor_model: a binary_latent_factor_model
82         :return: free energies after each update
83         """
84         free_energy = [self.compute_free_energy(x, binary_latent_factor_model)]
85         for i in range(self.k):
86             # singleton factor update
87             natural_parameter_ii = self.calculate_singleton_message_update(
88                 boltzmann_machine=binary_latent_factor_model,
89                 x=x,
90                 i=i,
91             )
92             self.bernoulli_parameter_matrix[
93                 :, i, i
94             ] = self.calculate_bernoulli_parameter(natural_parameter_ii)

```

```

95         free_energy.append(self.compute_free_energy(x, binary_latent_factor_model))
96
97     for j in range(i):
98         # binary factor update
99         natural_parameter_ij = self.calculate_binary_message_update(
100             boltzmann_machine=binary_latent_factor_model,
101             x=x,
102             i=i,
103             j=j,
104         )
105         self.bernoulli_parameter_matrix[
106             :, i, j
107         ] = self.calculate_bernoulli_parameter(natural_parameter_ij)
108         natural_parameter_ji = self.calculate_binary_message_update(
109             boltzmann_machine=binary_latent_factor_model,
110             x=x,
111             i=j,
112             j=i,
113         )
114         self.bernoulli_parameter_matrix[
115             :, j, i
116         ] = self.calculate_bernoulli_parameter(natural_parameter_ji)
117         free_energy.append(
118             self.compute_free_energy(x, binary_latent_factor_model)
119         )
120     return free_energy
121
122 def calculate_binary_message_update(
123     self,
124     x: np.ndarray,
125     boltzmann_machine: BoltzmannMachine,
126     i: int,
127     j: int,
128 ) -> float:
129     """
130     Calculate new parameters for a binary factored message.
131
132     :param x: data matrix (number_of_points, number_of_dimensions)
133     :param boltzmann_machine: Boltzmann machine model
134     :param i: starting node for the message
135     :param j: ending node for the message
136     :return: new parameter from aggregating incoming messages
137     """
138     natural_parameter_i_not_j = boltzmann_machine.b.index(
139         x=x, node_index=i
140     ) + self.aggregate_incoming_binary_factor_messages(
141         node_index=i, excluded_node_index=j
142     )
143     w_i_j = boltzmann_machine.w.matrix_index(i, j)
144     return np.log(1 + np.exp(w_i_j + natural_parameter_i_not_j)) - np.log(
145         1 + np.exp(natural_parameter_i_not_j)
146     )
147
148 @staticmethod
149 def calculate_singleton_message_update(
150     self,
151     x: np.ndarray,
152     boltzmann_machine: BoltzmannMachine,
153     i: int,
154 ) -> float:
155     """
156     Calculate the parameter update for the singleton message.
157     Note that this does not require any approximation.
158
159     :param x: data matrix (number_of_points, number_of_dimensions)
160     :param boltzmann_machine: Boltzmann machine model
161     :param i: node to update
162     :return: new parameter
163     """
164     return boltzmann_machine.b.index(x=x, node_index=i)
165
166 def init_message_passing(k: int, n: int) -> MessagePassingApproximation:
167     """
168     Message passing initialisation
169
170     :param k: number of latent variables
171     :param n: number of data points
172     :return: message passing
173     """
174     bernoulli_parameter_matrix = np.random.random(size=(n, k, k))
175     return MessagePassingApproximation(bernoulli_parameter_matrix)

```

src/models/binary_latent_factor_approximations/message_passing_approximation.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_approximation 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
18     # pre-training features plot
19     fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
20     for i in range(k):
21         ax[i].imshow(boltzmann_machine.mu[:, i].reshape(4, 4))
22         ax[i].set_title(f"Latent Feature mu_{i}")
23     fig.suptitle("Initial Features (Loopy BP)")
24     plt.tight_layout()
25     plt.savefig(save_path + "-init-latent-factors", bbox_inches="tight")
26     plt.close()
27
28     # EM
29     message_passing, boltzmann_machine, free_energy = learn_binary_factors(
30         x=x,
31         em_iterations=em_iterations,
32         binary_latent_factor_model=boltzmann_machine,
33         binary_latent_factor_approximation=message_passing,
34     )
35
36     # post training features plot
37     fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
38     for i in range(k):
39         ax[i].imshow(boltzmann_machine.mu[:, i].reshape(4, 4))
40         ax[i].set_title(f"Latent Feature mu_{i}")
41     fig.suptitle("Learned Features (Loopy BP)")
42     plt.tight_layout()
43     plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
44     plt.close()
45
46     # free energy plot
47     plt.title("Free Energy (Loopy BP)")
48     plt.xlabel("t (EM steps)")
49     plt.ylabel("Free Energy")
50     plt.plot(free_energy)
51     plt.savefig(save_path + "-free-energy", bbox_inches="tight")
52     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         """
19         matrix of means (number_of_dimensions, number_of_latent_variables)
20         """
21         pass
22
23     @property
24     @abstractmethod
25     def variance(self) -> float:
26         """
27         gaussian noise parameter
28         """
29         pass
30
31     @property
32     @abstractmethod
33     def pi(self) -> np.ndarray:
34         """
35         (1, number_of_latent_variables)
36         """
37         pass
38
39     @abstractmethod
40     def maximisation_step(
41         self,
42         x: np.ndarray,
43         binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
44     ) -> None:
45         pass
46
47     def mu_exclude(self, exclude_latent_index: int) -> np.ndarray:
48         return np.concatenate( # (number_of_dimensions, number_of_latent_variables-1)
49             (self.mu[:, :exclude_latent_index], self.mu[:, exclude_latent_index + 1 :]),
50             axis=1,
51         )
52
53     @property
54     def log_pi(self) -> np.ndarray:
55         return np.log(self.pi)
56
57     @property
58     def log_one_minus_pi(self) -> np.ndarray:
59         return np.log(1 - self.pi)
60
61     @property
62     def precision(self) -> float:
63         return 1 / self.variance
64
65     @property
66     def d(self) -> int:
67         return self.mu.shape[0]
68
69     @property
70     def k(self) -> int:
71         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         """
19         lambda_matrix: parameters variational approximation (number_of_points, number_of_latent_variables)
20         """
21         pass
22
23     @abstractmethod
24     def variational_expectation_step(
25         self,
26         x: np.ndarray,
27         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
28     ) -> List[float]:
29         pass
30
31     @property
32     def expectation_s(self):
33         return self.lambda_matrix
34
35     @property
36     def expectation_ss(self):
37         ess = self.lambda_matrix.T @ self.lambda_matrix
38         np.fill_diagonal(ess, self.lambda_matrix.sum(axis=0))
39         return ess
40
41     @property
42     def log_lambda_matrix(self) -> np.ndarray:
43         return np.log(self.lambda_matrix)
44
45     @property
46     def log_one_minus_lambda_matrix(self) -> np.ndarray:
47         return np.log(1 - self.lambda_matrix)
48
49     @property
50     def n(self) -> int:
51         """
52         Number of data points
53         """
54         return self.lambda_matrix.shape[0]
55
56     @property
57     def k(self) -> int:
58         """
59         Number of latent variables
60         """
61         return self.lambda_matrix.shape[1]
62
63     def compute_free_energy(
64         self,
65         x: np.ndarray,
66         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
67     ) -> float:
68         """
69         free energy associated with current EM parameters and data x
70
71         :param x: data matrix (number_of_points, number_of_dimensions)
72         :param binary_latent_factor_model: a binary_latent_factor_model
73         :return: average free energy per data point
74         """
75         expectation_log_p_x_s_given_theta = (
76             self._compute_expectation_log_p_x_s_given_theta(
77                 x, binary_latent_factor_model
78             )
79         )
80         approximation_model_entropy = self._compute_approximation_model_entropy()
81         return (
82             expectation_log_p_x_s_given_theta + approximation_model_entropy
83         ) / self.n
84
85     def _compute_expectation_log_p_x_s_given_theta(
86         self,
87         x: np.ndarray,
88         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
89     ) -> float:
90         """
91         The first term of the free energy, the expectation of log P(X,S|theta)
92         """
```

```

93 :param x: data matrix (number_of_points, number_of_dimensions)
94 :param binary_latent_factor_model: a binary_latent_factor_model
95 :return: the expectation of log P(X,S|theta)
96 """
97 # (number_of_points, number_of_dimensions)
98 mu_lambda = self.lambda_matrix @ binary_latent_factor_model.mu.T
99
100 # (number_of_latent_variables, number_of_latent_variables)
101 expectation_s_i_s_j_mu_i_mu_j = np.multiply(
102     self.lambda_matrix.T @ self.lambda_matrix,
103     binary_latent_factor_model.mu.T @ binary_latent_factor_model.mu,
104 )
105
106 expectation_log_p_x_given_s_theta = -(
107     self.n * binary_latent_factor_model.d / 2
108 ) * np.log(2 * np.pi * binary_latent_factor_model.variance) - (
109     0.5 * binary_latent_factor_model.precision
110 ) * (
111     np.sum(np.multiply(x, x))
112     - 2 * np.sum(np.multiply(x, mu_lambda))
113     + np.sum(expectation_s_i_s_j_mu_i_mu_j)
114     - np.trace(
115         expectation_s_i_s_j_mu_i_mu_j
116     ) # remove incorrect E[s_i s_i] = lambda_i * lambda_i
117 + np.sum( # add correct E[s_i s_i] = lambda_i
118     self.lambda_matrix
119     @ np.multiply(
120         binary_latent_factor_model.mu, binary_latent_factor_model.mu
121     ).T
122 )
123 )
124 expectation_log_p_s_given_theta = np.sum(
125     np.multiply(
126         self.lambda_matrix,
127         binary_latent_factor_model.log_pi,
128     )
129 + np.multiply(
130     1 - self.lambda_matrix,
131     binary_latent_factor_model.log_one_minus_pi,
132 )
133 )
134 return expectation_log_p_x_given_s_theta + expectation_log_p_s_given_theta
135
136 def _compute_approximation_model_entropy(self) -> float:
137     """
138     Compute the model entropy
139
140     :return: model entropy
141     """
142     return -np.sum(
143         np.multiply(
144             self.lambda_matrix,
145             self.log_lambda_matrix,
146         )
147 + np.multiply(
148     1 - self.lambda_matrix,
149     self.log_one_minus_lambda_matrix,
150 )
151 )

```

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, q5, 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(5e-1),
52         log_eta=jnp.log(1),
53         log_tau=jnp.log(1.5),
54         log_zeta=jnp.log(1e-2),
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 = 14
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.g(
136     x=x[:, 1, :],
137     binary_latent_factor_model=binary_latent_factor_model,
138     sigmas=[1, 2, 3],
139     k=k,
140     em_iterations=em_iterations,
141     e_maximum_steps=e_maximum_steps,
142     e_convergence_criterion=e_convergence_criterion,
143     save_path=os.path.join(Q3.OUTPUT_FOLDER, "g"),
144 )
145
146 # Question 4
147 Q4.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q4")
148 if not os.path.exists(Q4.OUTPUT_FOLDER):
149     os.makedirs(Q4.OUTPUT_FOLDER)
150 max_k = 21
151 free_energies_1 = q4.b(
152     x=x,
153     a_parameter=1,
154     b_parameter=0,
155     ks=np.arange(4, 13),
156     max_k=max_k,
157     em_iterations=em_iterations,
158     e_maximum_steps=e_maximum_steps,
159     e_convergence_criterion=e_convergence_criterion,
160     save_path=os.path.join(Q4.OUTPUT_FOLDER, "b-1"),
161 )
162 free_energies_2 = q4.b(
163     x=x,
164     a_parameter=1,
165     b_parameter=0,
166     ks=np.arange(13, 22),
167     max_k=max_k,
168     em_iterations=em_iterations,
169     e_maximum_steps=e_maximum_steps,
170     e_convergence_criterion=e_convergence_criterion,
171     save_path=os.path.join(Q4.OUTPUT_FOLDER, "b-2"),
172 )
173 q4.free_energy_plot(
174     ks=np.arange(4, 22),
175     free_energies=free_energies_1 + free_energies_2,
176     model_name="Variational Bayes",
177     save_path=os.path.join(Q4.OUTPUT_FOLDER, "b"),
178 )
179
180 # Question 5
181 Q5.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q5")
182 if not os.path.exists(Q5.OUTPUT_FOLDER):
183     os.makedirs(Q5.OUTPUT_FOLDER)
184 max_k = 21
185 free_energies_1 = q5.d(
186     x=x,
187     a_parameter=1,
188     b_parameter=0,

```

```

189         ks=np.arange(4, 13),
190         max_k=max_k,
191         em_iterations=em_iterations,
192         save_path=os.path.join(Q5.OUTPUT_FOLDER, "d-1"),
193     )
194     free_energies_2 = q5.d(
195         x=x,
196         a_parameter=1,
197         b_parameter=0,
198         ks=np.arange(13, 22),
199         max_k=max_k,
200         em_iterations=em_iterations,
201         save_path=os.path.join(Q5.OUTPUT_FOLDER, "d-2"),
202     )
203     q4.free_energy_plot(
204         ks=np.arange(4, 22),
205         free_energies=free_energies_1 + free_energies_2,
206         model_name="Loopy BP E Step and Variational Bayes M Step",
207         save_path=os.path.join(Q5.OUTPUT_FOLDER, "d"),
208     )
209
210 # Question 6
211 Q6.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q6")
212 if not os.path.exists(Q6.OUTPUT_FOLDER):
213     os.makedirs(Q6.OUTPUT_FOLDER)
214 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(
7     n: int = 400, seed: int = DEFAULT_SEED, sigma: float = 0.1
8 ) -> np.ndarray:
9     """
10     Image generation, adapted from provided demo code
11
12     :param n: number of data points
13     :param seed: random seed
14     :param sigma: Gaussian noise
15     :return: images as a data matrix (number_of_points, number_of_dimensions)
16     """
17     d = 16 # dimensionality of the data
18     np.random.seed(seed)
19
20     # Define the basic shapes of the features
21     number_of_features = 8 # number of features
22     rr = (
23         0.5 + np.random.rand(number_of_features, 1) * 0.5
24     ) # weight of each feature between 0.5 and 1
25     mut = np.array(
26         [
27             rr[0] * M1,
28             rr[1] * M2,
29             rr[2] * M3,
30             rr[3] * M4,
31             rr[4] * M5,
32             rr[5] * M6,
33             rr[6] * M7,
34             rr[7] * M8,
35         ]
36     )
37     s = (
38         np.random.rand(n, number_of_features) < 0.3
39     ) # each feature occurs with prob 0.3 independently
40
41     # Generate Data - The Data is stored in Y
42
43     return (
44         np.dot(s, mut) + np.random.randn(n, d) * sigma
45     ) # some Gaussian noise is added
```

src/generate_images.py

Appendix 6: m_step.py

```
1 import numpy as np
2
3 from typing import Tuple
4
5
6 def m_step(x: np.ndarray, es: np.ndarray, ess: np.ndarray) -> Tuple[np.ndarray, float, np.ndarray]:
7     """
8     mu, sigma, pie = m_step(x, es, ess)
9
10    Inputs:
11    -----
12        x: shape (n, d) data matrix
13        es: shape (n, k) E-q[s]
14        ess: shape (k, k) sum over data points of E-q[ss'] (n, k, k)
15                if E-q[ss'] is provided, the sum over n is done for you.
16
17    Outputs:
18    -----
19        mu: shape (d, k) matrix of means in p(y|{s_i}, mu, sigma)
20        sigma: shape (,) standard deviation in same
21        pie: shape (1, k) vector of parameters specifying generative distribution for s
22    """
23    n, d = x.shape
24    if es.shape[0] != n:
25        raise TypeError('es must have the same number of rows as x')
26    k = es.shape[1]
27    if ess.shape == (n, k, k):
28        ess = np.sum(ess, axis=0)
29    if ess.shape != (k, k):
30        raise TypeError('ess must be square and have the same number of columns as es')
31
32    mu = np.dot(np.dot(np.linalg.inv(ess), es.T), x).T
33    sigma = np.sqrt((np.trace(np.dot(x.T, x)) + np.trace(np.dot(np.dot(mu.T, mu), ess))
34                    - 2 * np.trace(np.dot(np.dot(es.T, x), mu))) / (n * d))
35    pie = np.mean(es, axis=0, keepdims=True)
36
37    return mu, sigma, pie
```

demo_code/m_step.py