

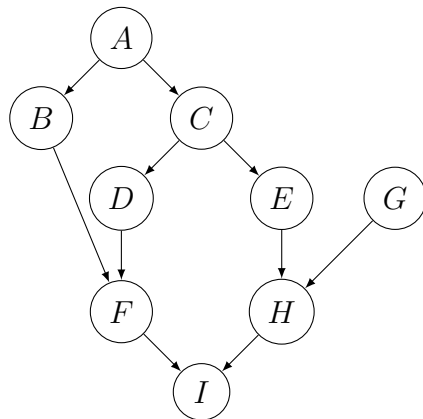
COMP0085 Summative Assignment

Jan 4, 2023

Question 1

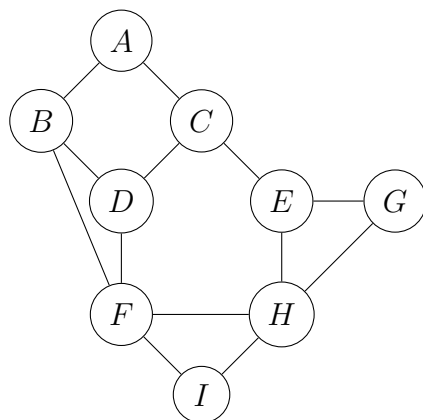
(a)

The directed acyclic graph:

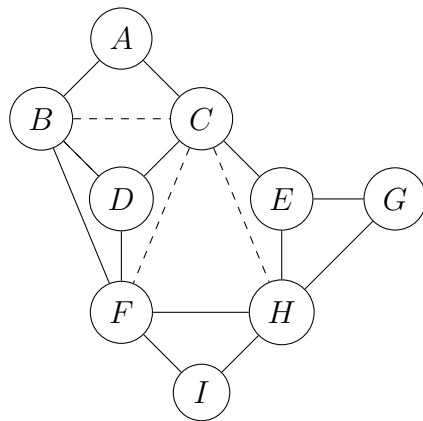


(b)

The moralised graph:

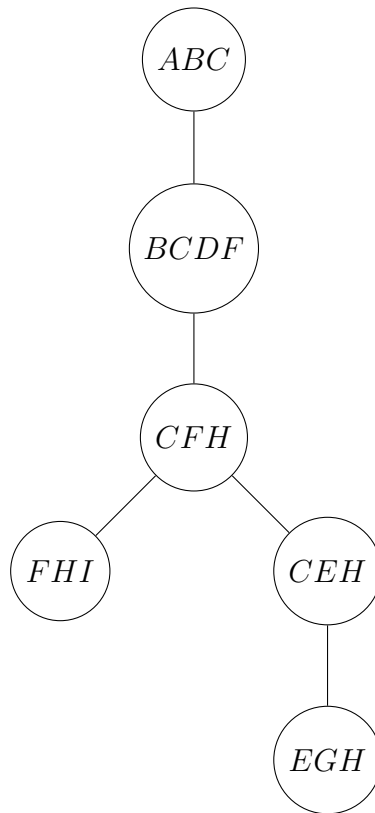


An effective triangulation:



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

The resulting junction tree:



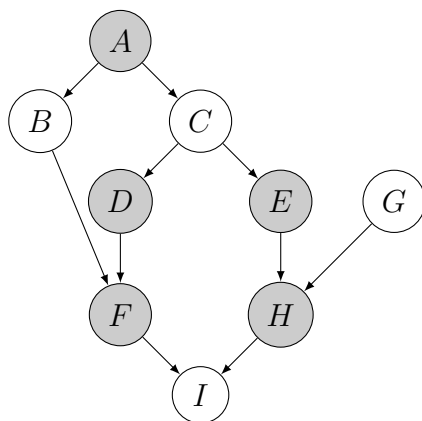
where the circular nodes are cliques.

The junction tree redrawn as a factor graph:



where the circular nodes are cliques and the square nodes are separators/factors.

(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)

(e)

Question 2

(a)

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

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

Our distribution for \mathbf{w} :

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

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

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

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

Knowing:

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

we can substitute the above distributions:

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

expanding:

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

collecting \mathbf{w} terms:

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

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

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

Thus we can identify the posterior covariance:

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

and the posterior mean:

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

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

value		
parameters	a	1.828457
	b	334.203782

Figure 1: The Posterior Mean

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

Figure 2: The Posterior Covariance

(b)

Plotting the residuals:

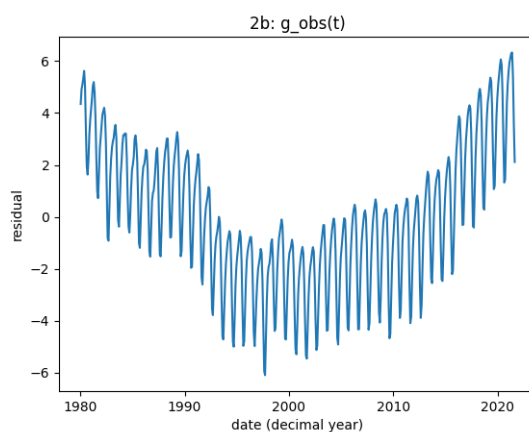


Figure 3: $g_{obs}(t)$

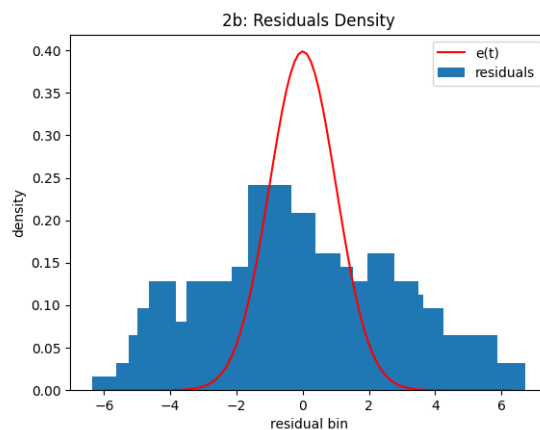


Figure 4: Density Estimation of Residuals vs $e(t) \sim \mathcal{N}(0, 1)$

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

(c & d)

We are considering the kernel:

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

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

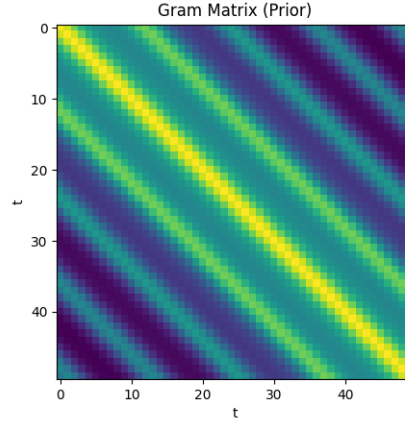


Figure 5: Covariance Matrix

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

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

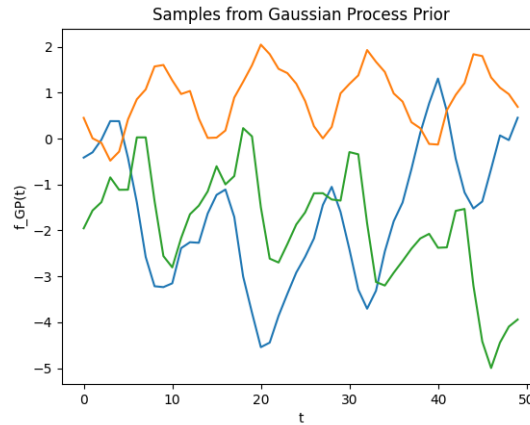


Figure 6: Samples from a zero mean GP with the provided covariance kernel

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

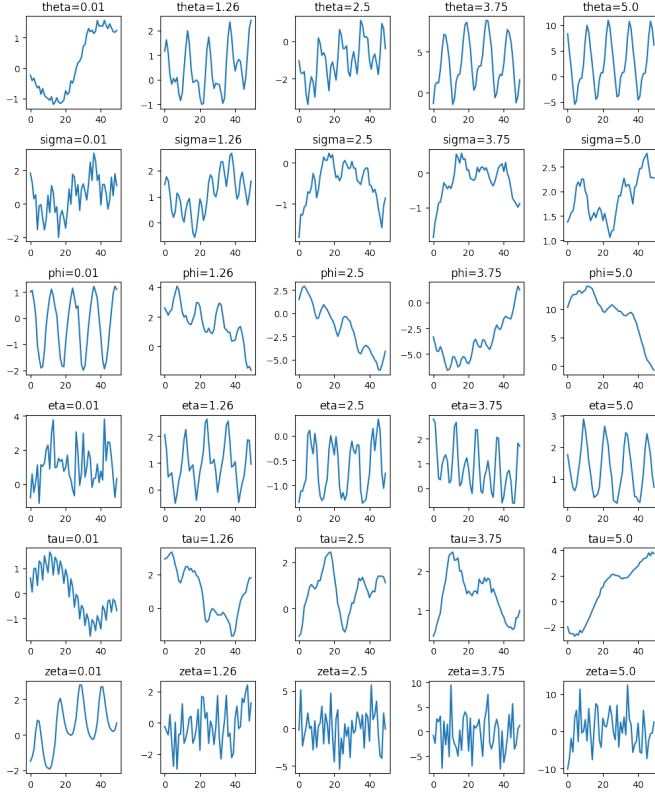


Figure 7: Samples for different parameters

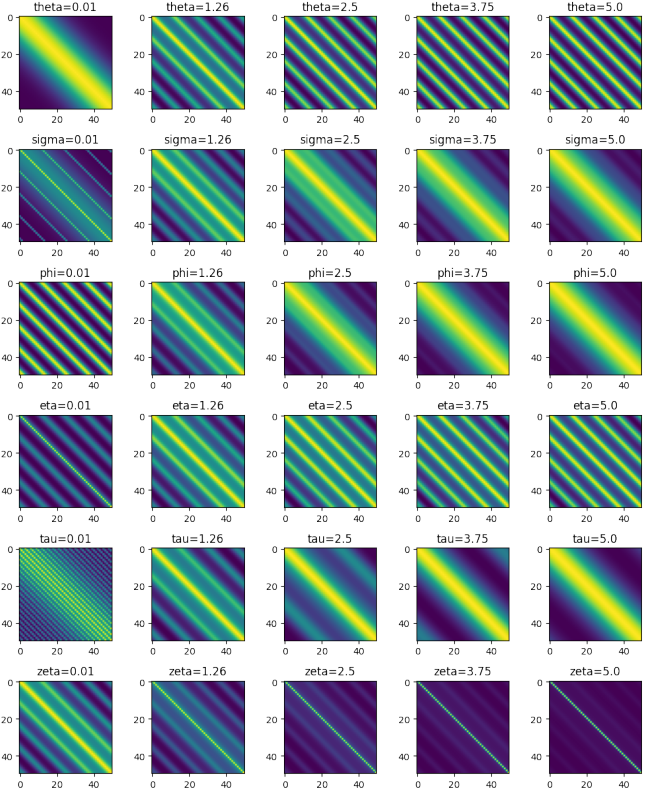


Figure 8: Covariances for different parameters

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

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

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

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

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

(e)

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

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

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

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

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

Figure 9: Untrained hyperparameters

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

Figure 10: Trained Hyperparameters

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

(f)

Extrapolating the CO_2 concentration levels:

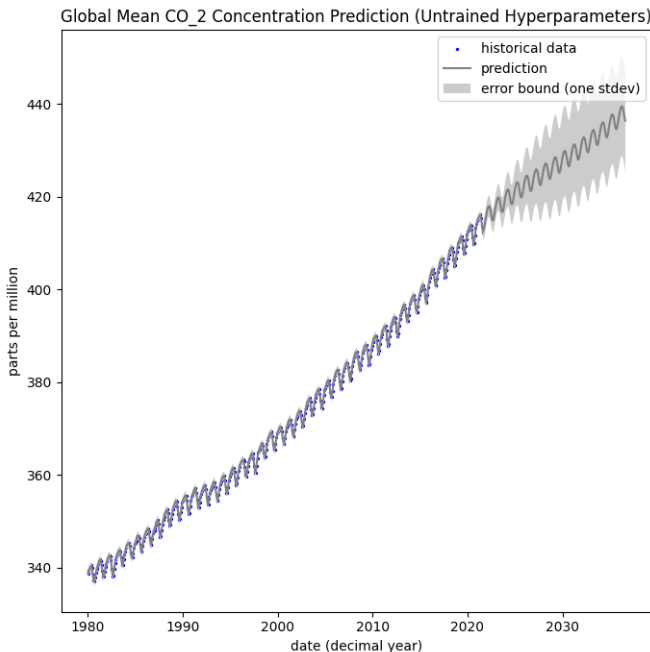


Figure 11: Untrained extrapolation

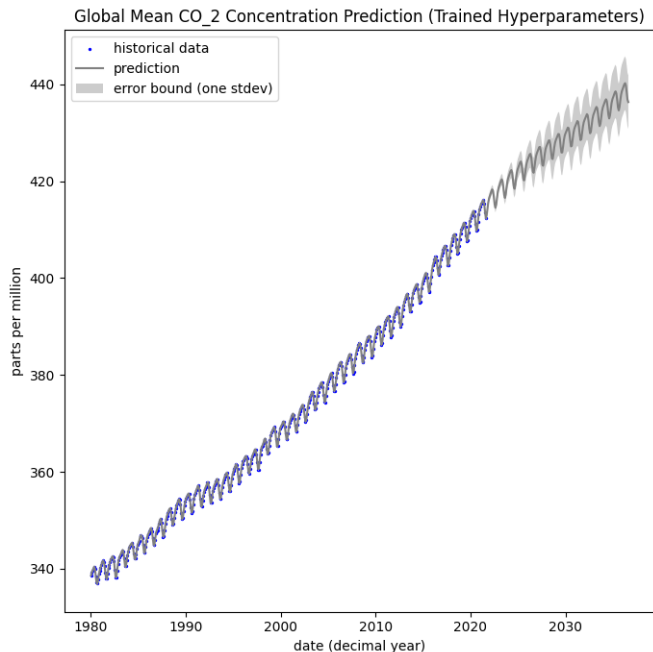


Figure 12: Trained extrapolation

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

(g)

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

The Python code for Bayesian Linear Regression:

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

src/models/bayesian_linear_regression.py

The Python code for kernels:

```

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

```

```

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

```



```

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

```

src/models/kernels.py

The Python code for Gaussian Process Regression:

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

```

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

```

src/models/gaussian_process_regression.py

The rest of the Python code for question 2:

```

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

```

```

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

```

```

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

```

```

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

```

src/solutions/q2.py

Question 3

(a)

The free energy is can be calculated as:

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

Knowing,

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

we can write:

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

Moreover, our mean field approximation:

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

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

To compute the first term:

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

substituting the appropriate terms:

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

with d being the number of dimensions.

Taking the logarithm:

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

The expectation distributed to the relevant terms:

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

Evaluating the expectations:

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

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

To compute the second term:

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

Taking the logarithm:

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

The expectation distributed to the relevant terms:

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

Evaluating the expectations:

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

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

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

Thus,

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

Substituting the appropriate values:

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

Combining, we have our free energy expression:

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

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

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

where λ^{LG} is the Lagrange multiplier.

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

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

Similar to our free energy derivation:

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

and

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

We can write:

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

Substituting the relevant terms:

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

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

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

Thus,

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

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

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

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

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

Solving for λ_i :

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

we have our partial update.

(b)

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

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

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

This mimics the least squares solution:

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

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

(c)

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

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

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

(d)

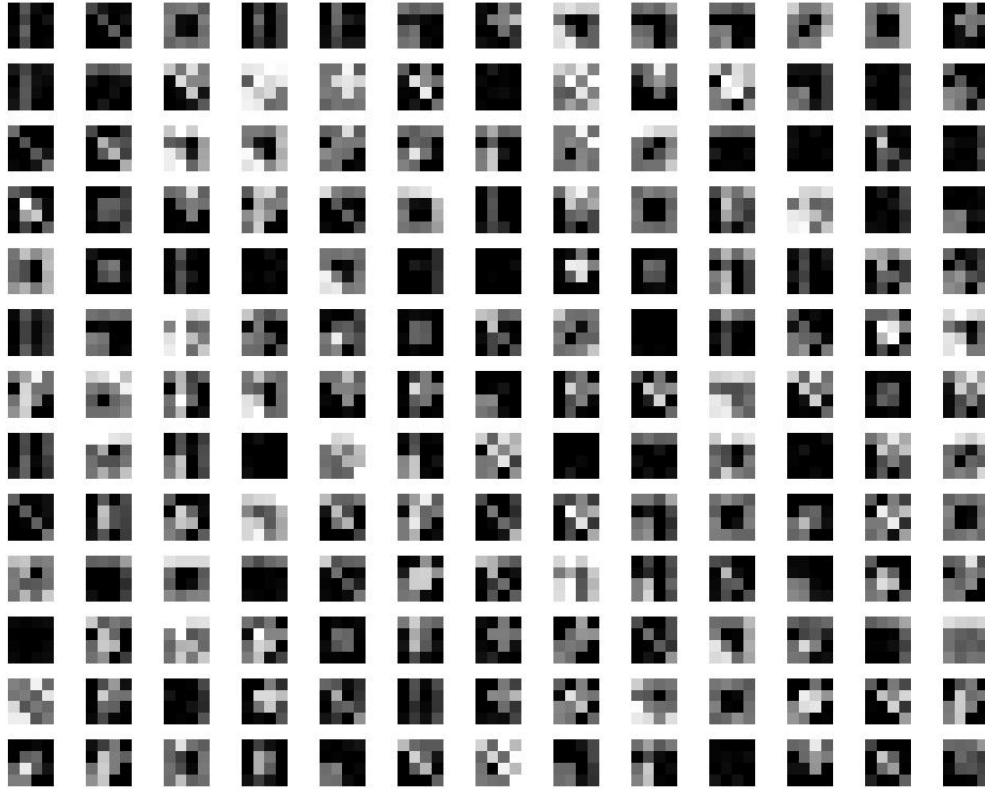


Figure 13: Images generated by randomly combined features with noise

Examining the generated images, we can see eight features:

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

Factor analysis assumes a model:

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

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

A mixture of Gaussians assumes as model:

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

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

The independent component analysis assumes a model:

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

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

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

(e)

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

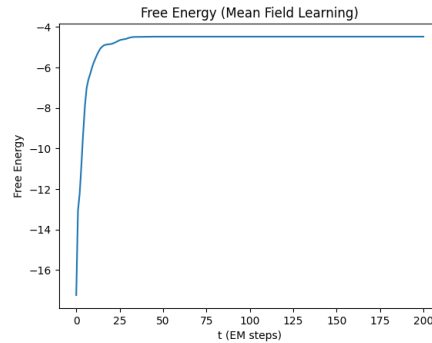


Figure 14: Free Energy

(f)

The initialised features:

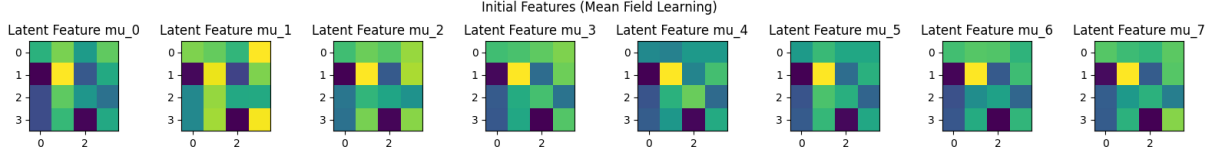


Figure 15: Initial Latent Factors

The features learned by the algorithm:

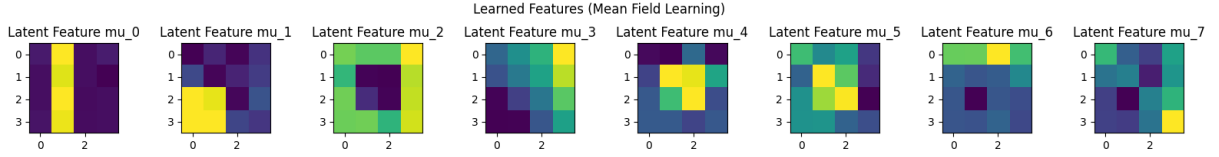


Figure 16: Learned Latent Factors

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

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

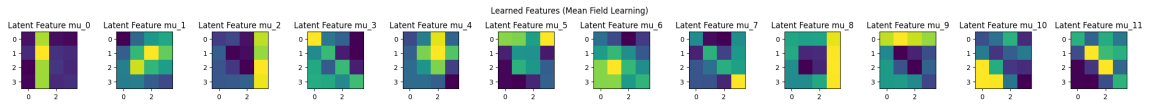


Figure 17: Increasing Number of Latent Factors

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

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

(g)

Plotting the convergence of the variational approximation for different σ 's:

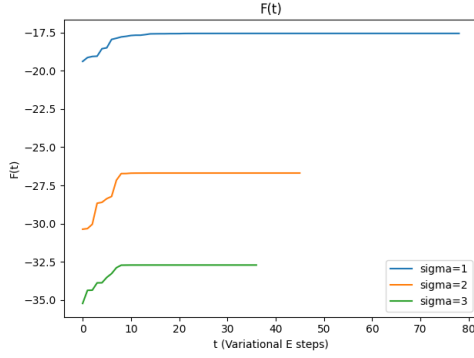


Figure 18: Free energy vs σ

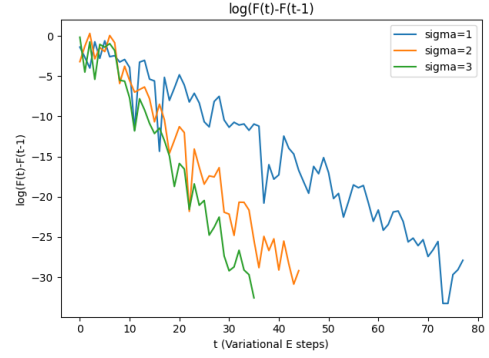


Figure 19: Free energy convergence vs σ

We can see that when σ is smaller, we are able to converge at a higher free energy. However, convergence is only reached after more steps, as seen in the plot of $\log(F(t) - F(t-1))$

The Python code for the binary latent factor model:

```

1  from __future__ import annotations
2
3  from abc import ABC, abstractmethod
4  from typing import List, Tuple
5
6  import numpy as np
7
8  from demo_code.MStep import m_step
9
10
11 class AbstractBinaryLatentFactorModel(ABC):
12     @property
13     @abstractmethod
14     def mu(self) -> np.ndarray:
15         pass
16
17     @property
18     @abstractmethod
19     def variance(self) -> float:
20         pass
21
22     @property
23     @abstractmethod
24     def pi(self) -> np.ndarray:
25         pass
26
27     @abstractmethod
28     def maximisation_step(
29         self,
30         x: np.ndarray,
31         binary_latent_factor_approximation: BinaryLatentFactorApproximation,
32     ) -> None:
33         pass
34
35     def mu_exclude(self, exclude_latent_index: int) -> np.ndarray:
36         return np.concatenate( # (number_of_dimensions, number_of_latent_variables-1)
37             (self.mu[:, :exclude_latent_index], self.mu[:, exclude_latent_index + 1 :]),
38             axis=1,
39         )
40
41     @property
42     def log_pi(self) -> np.ndarray:
43         return np.log(self.pi)
44
45     @property
46     def log_one_minus_pi(self) -> np.ndarray:
47         return np.log(1 - self.pi)
48
49     @property
50     def precision(self) -> float:
51         return 1 / self.variance
52
53     @property
54     def d(self) -> int:
55         return self.mu.shape[0]
56
57     @property
58     def k(self) -> int:
59         return self.mu.shape[1]
60
61
62 class BinaryLatentFactorModel(AbstractBinaryLatentFactorModel):
63     """
64     mu: matrix of means (number_of_dimensions, number_of_latent_variables)
65     sigma: gaussian noise parameter
66     pi: vector of priors (1, number_of_latent_variables)
67     """
68
69     def __init__(
70         self,
71         mu: np.ndarray,
72         sigma: float,
73         pi: np.ndarray,
74     ):
75         self._mu = mu
76         self._sigma = sigma
77         self._pi = pi
78
79     @property
80     def mu(self):
81         return self._mu
82
83     @mu.setter
84     def mu(self, value):
85         self._mu = value
86
87     @property
88     def sigma(self):
89         return self._sigma
90
91     @sigma.setter
92     def sigma(self, value):
93         self._sigma = value
94

```



```

95 @property
96 def pi(self):
97     return self._pi
98
99 @pi.setter
100 def pi(self, value):
101     self._pi = value
102
103 @property
104 def variance(self) -> float:
105     return self.sigma**2
106
107 @staticmethod
108 def calculate_maximisation_parameters(
109     x: np.ndarray,
110     binary_latent_factor_approximation: BinaryLatentFactorApproximation,
111 ) -> Tuple[np.ndarray, float, np.ndarray]:
112     return m_step(
113         x=x,
114         es=binary_latent_factor_approximation.expectation_s,
115         ess=binary_latent_factor_approximation.expectation_ss,
116     )
117
118 def maximisation_step(
119     self,
120     x: np.ndarray,
121     binary_latent_factor_approximation: BinaryLatentFactorApproximation,
122 ) -> None:
123     mu, sigma, pi = self.calculate_maximisation_parameters(
124         x, binary_latent_factor_approximation
125     )
126     self.mu = mu
127     self.sigma = sigma
128     self.pi = pi
129
130
131 def init_binary_latent_factor_model(
132     x: np.ndarray,
133     binary_latent_factor_approximation: BinaryLatentFactorApproximation,
134 ) -> BinaryLatentFactorModel:
135     mu, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
136         x, binary_latent_factor_approximation
137     )
138     return BinaryLatentFactorModel(mu, sigma, pi)
139
140
141 class BinaryLatentFactorApproximation(ABC):
142     @property
143     @abstractmethod
144     def lambda_matrix(self) -> np.ndarray:
145         pass
146
147     @abstractmethod
148     def variational_expectation_step(
149         self,
150         x: np.ndarray,
151         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
152     ) -> List[float]:
153         pass
154
155     @property
156     def expectation_s(self):
157         return self.lambda_matrix
158
159     @property
160     def expectation_ss(self):
161         ess = self.lambda_matrix.T @ self.lambda_matrix
162         np.fill_diagonal(ess, self.lambda_matrix.sum(axis=0))
163         return ess
164
165     @property
166     def log_lambda_matrix(self) -> np.ndarray:
167         return np.log(self.lambda_matrix)
168
169     @property
170     def log_one_minus_lambda_matrix(self) -> np.ndarray:
171         return np.log(1 - self.lambda_matrix)
172
173     @property
174     def n(self) -> int:
175         return self.lambda_matrix.shape[0]
176
177     @property
178     def k(self) -> int:
179         return self.lambda_matrix.shape[1]
180
181     def compute_free_energy(
182         self,
183         x: np.ndarray,
184         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
185     ) -> float:
186         """
187         free energy associated with current EM parameters and data x
188
189         :param x: data matrix (number_of_points, number_of_dimensions)
190         :param binary_latent_factor_model: a binary_latent_factor_model

```

```

191         :return: average free energy per data point
192         """
193         expectation_log_p_x_s_given_theta = (
194             self._compute_expectation_log_p_x_s_given_theta(
195                 x, binary_latent_factor_model
196             )
197         )
198         approximation_model_entropy = self._compute_approximation_model_entropy()
199         return (
200             expectation_log_p_x_s_given_theta + approximation_model_entropy
201         ) / self.n
202
203     def _compute_expectation_log_p_x_s_given_theta(
204         self,
205         x: np.ndarray,
206         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
207     ) -> float:
208         """
209         The first term of the free energy, the expectation of log P(X,S|theta)
210
211         :param x: data matrix (number_of_points, number_of_dimensions)
212         :param binary_latent_factor_model: a binary_latent_factor_model
213         :return: the expectation of log P(X,S|theta)
214         """
215         # (number_of_points, number_of_dimensions)
216         mu_lambda = self.lambda_matrix @ binary_latent_factor_model.mu.T
217
218         # (number_of_latent_variables, number_of_latent_variables)
219         expectation_s_i_s_j_mu_i_mu_j = np.multiply(
220             self.lambda_matrix.T @ self.lambda_matrix,
221             binary_latent_factor_model.mu.T @ binary_latent_factor_model.mu,
222         )
223
224         expectation_log_p_x_given_s_theta = -(
225             self.n * binary_latent_factor_model.d / 2
226             * np.log(2 * np.pi * binary_latent_factor_model.variance) - (
227                 0.5 * binary_latent_factor_model.precision
228             ) * (
229                 np.sum(np.multiply(x, x))
230                 - 2 * np.sum(np.multiply(x, mu_lambda))
231                 + np.sum(expectation_s_i_s_j_mu_i_mu_j)
232                 - np.trace(
233                     expectation_s_i_s_j_mu_i_mu_j
234                 ) # remove incorrect E[s_i s_i] = lambda_i * lambda_i
235                 + np.sum( # add correct E[s_i s_i] = lambda_i
236                     self.lambda_matrix
237                     @ np.multiply(
238                         binary_latent_factor_model.mu, binary_latent_factor_model.mu
239                     ).T
240                 )
241             )
242         )
243         expectation_log_p_s_given_theta = np.sum(
244             np.multiply(
245                 self.lambda_matrix,
246                 binary_latent_factor_model.log_pi,
247             )
248             + np.multiply(
249                 1 - self.lambda_matrix,
250                 binary_latent_factor_model.log_one_minus_pi,
251             )
252         )
253         return expectation_log_p_x_given_s_theta + expectation_log_p_s_given_theta
254
255     def _compute_approximation_model_entropy(self) -> float:
256         return -np.sum(
257             np.multiply(
258                 self.lambda_matrix,
259                 self.log_lambda_matrix,
260             )
261             + np.multiply(
262                 1 - self.lambda_matrix,
263                 self.log_one_minus_lambda_matrix,
264             )
265         )
266
267     def is_converge(
268         free_energies: List[float],
269         current_lambda_matrix: np.ndarray,
270         previous_lambda_matrix: np.ndarray,
271     ) -> bool:
272         return (abs(free_energies[-1] - free_energies[-2]) == 0) and np.linalg.norm(
273             current_lambda_matrix - previous_lambda_matrix
274         ) == 0
275
276
277     def learn_binary_factors(
278         x: np.ndarray,
279         em_iterations: int,
280         binary_latent_factor_model: AbstractBinaryLatentFactorModel,
281         binary_latent_factor_approximation: BinaryLatentFactorApproximation,
282     ) -> Tuple[
283         BinaryLatentFactorApproximation, AbstractBinaryLatentFactorModel, List[float]
284     ]:
285         free_energies: List[float] = [
286             binary_latent_factor_approximation.compute_free_energy(

```

```

287         x, binary_latent_factor_model
288     )
289 ]
290 for _ in range(em_iterations):
291     previous_lambda_matrix = np.copy(
292         binary_latent_factor_approximation.lambda_matrix
293     )
294     binary_latent_factor_approximation.variational_expectation_step(
295         x=x,
296         binary_latent_factor_model=binary_latent_factor_model,
297     )
298     binary_latent_factor_model.maximisation_step(
299         x,
300         binary_latent_factor_approximation,
301     )
302     free_energies.append(
303         binary_latent_factor_approximation.compute_free_energy(
304             x, binary_latent_factor_model
305         )
306     )
307     if is_converge(
308         free_energies,
309         binary_latent_factor_approximation.lambda_matrix,
310         previous_lambda_matrix,
311     ):
312         break
313 return binary_latent_factor_approximation, binary_latent_factor_model, free_energies

```

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

```

```

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

```

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

```

```

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

```

src/solutions/q3.py

Question 4

We begin with the log joint:

$$\log P(\mathbf{x}, \mathbf{s}, \pi, \mathbf{A}, \Psi | \eta) = \log P(\mathbf{x} | \mathbf{s}, \mathbf{A}, \Psi, \eta) + \log P(\mathbf{s} | \pi, \eta) + \log P(\pi | \eta) + \log P(\mathbf{A} | \eta) + \log P(\Psi | \eta)$$

where $\mathbf{A} \in \mathbb{R}^{D \times K}$ is a concatenation of μ_k for $k \in \{1, \dots, K\}$ and η is a collection of all hyperparameters.

We know:

$$P(\mathbf{x} | \mathbf{s}, \mathbf{A}, \Psi, \eta) = \frac{1}{(2\pi)^{\frac{D}{2}} |\Psi|^{\frac{1}{2}}} \exp \left(-\frac{1}{2} (\mathbf{x} - \mathbf{A}\mathbf{s})^T \Psi^{-1} (\mathbf{x} - \mathbf{A}\mathbf{s}) \right)$$

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

$$P(\pi | \eta) = \prod_{k=1}^K \frac{\pi_k^{\alpha-1} (1 - \pi_k)^{\beta-1}}{B(\alpha, \beta)}$$

For \mathbf{A} we choose a factorised conjugate prior for each column, μ_k :

$$P(\mathbf{A} | \eta) = \prod_{k=1}^K P(\mu_k | \eta)$$

For each column we choose:

$$P(\mu_k | \eta) = \mathcal{N}(\mu_k | \mu_{\mu_k}, \Sigma_{\mu_k}) = \frac{1}{(2\pi)^{\frac{D}{2}} |\Sigma_{\mu_k}|^{\frac{1}{2}}} \exp \left(-\frac{1}{2} (\mu_k - \mu_{\mu_k})^T \Sigma_{\mu_k}^{-1} (\mu_k - \mu_{\mu_k}) \right)$$

a Gaussian prior with diagonal covariance $\Sigma_{\mu_k} = \alpha_k^2 \mathbf{I}$ and mean zero, so we can simplify:

$$P(\mu_k | \alpha_k) = (2\pi\alpha_k^2)^{-\frac{D}{2}} \exp \left(-\frac{\mu_k^T \mu_k}{2\alpha_k^2} \right)$$

For Ψ we choose a conjugate prior:

$$P(\Psi | \eta) = \prod_{d=1}^D \text{InvGamma}(\sigma^2 | a, b) = \prod_{d=1}^D \frac{b^a}{\Gamma(a)} (\sigma^2)^{-a-1} \exp(-\frac{b}{\sigma^2})$$

a product of inverse gamma distributions, assuming Ψ is a diagonal matrix with elements σ^2 .
Combining, we have our expression:

$$\begin{aligned} \log P(\mathbf{x}, \mathbf{s}, \pi, \mathbf{A}, \Psi | \eta) = & -\frac{D}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \left(\mathbf{x}^T \mathbf{x} - 2\mathbf{x}^T \sum_{k=1}^K s_k \mu_k + \sum_{k=1}^K \sum_{k'=1, k' \neq k}^K s_k s_{k'} \mu_k^T \mu_{k'} + \sum_{k=1}^K s_k \mu_k^T \mu_k \right) \\ & + \sum_{k=1}^K s_k \log \pi_k + (1 - s_k) \log(1 - \pi_k) \\ & + \sum_{i=1}^K (\alpha - 1) \log \pi_k + (\beta - 1) \log(1 - \pi_k) - \log B(\alpha, \beta) \\ & + \sum_{i=1}^K -\frac{D}{2} \log(2\pi\alpha_k^2) - \frac{\mu_k^T \mu_k}{2\alpha_k^2} \\ & + \sum_{d=1}^D a \log b + (-a - 1) \log(\sigma^2) - \frac{b}{\sigma^2} - \log \Gamma(a) \end{aligned}$$

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:

$$q_s(\mathbf{s}) \propto \exp \langle \log P(\mathbf{x}, \mathbf{s}, \pi, \mathbf{A}, \Psi | \eta) \rangle_{q(\theta)}$$

where θ denotes the parameters $\pi, \mathbf{A}, \Psi, \eta$.

Substituting the relevant terms:

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

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

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

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

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

we have our partial E step update.

For the Variational Bayes maximisation step, we set:

$$q_\theta(\theta) \propto P(\theta) \exp \langle \log P(\mathbf{x}, \mathbf{s}, \pi, \mathbf{A}, \Psi | \eta) \rangle_{q(\mathbf{s})}$$

assuming the factorisation:

$$q_\theta(\theta) = q_\pi(\pi) q_\Psi(\Psi) q_{\mathbf{A}}(\mathbf{A})$$

we can calculate each factor independently.

For $q_\pi(\pi)$:

$$q_\pi(\pi) \propto P(\pi) \exp \langle \log P(\mathbf{x}, \mathbf{s}, \pi, \mathbf{A}, \Psi | \eta) \rangle_{q_{\mathbf{s}}(\mathbf{s}) q_{-\pi}(\theta)}$$

Substituting the appropriate terms:

$$q_\pi(\pi) \propto \left(\prod_{k=1}^K \frac{\pi_k^{\alpha-1} (1 - \pi_k)^{\beta-1}}{B(\alpha, \beta)} \right) \exp \left\langle \sum_{k=1}^K s_k \log \pi_k + (1 - s_k) \log (1 - \pi_k) \right\rangle_{q_{\mathbf{s}}(\mathbf{s}) q_{-\pi}(\theta)}$$

We see:

$$q_\pi(\pi) \propto \prod_{k=1}^K \frac{\pi_k^{\alpha + \frac{1}{D} (\sum_{k=1}^K \langle s_k \rangle_{q_{s_k}}) - 1} (1 - \pi_k)^{\beta - \frac{1}{D} (\sum_{k=1}^K \langle s_k \rangle_{q_{s_k}})}}{B(\alpha, \beta)}$$

$$q_\pi(\pi) = \prod_{k=1}^K \text{Beta} \left(\alpha + \frac{1}{D} \sum_{k=1}^K \langle s_k \rangle_{q_{s_k}}, \beta + 1 - \frac{1}{D} \sum_{k=1}^K \langle s_k \rangle_{q_{s_k}} \right)$$

For $q_\Psi(\Psi)$:

$$q_\Psi(\Psi) \propto P(\Psi) \exp \langle \log P(\mathbf{x}, \mathbf{s}, \pi, \mathbf{A}, \Psi | \eta) \rangle_{q_{\mathbf{s}}(\mathbf{s})q_{-\Psi}(\theta)}$$

Substituting the appropriate terms:

$$q_\Psi(\Psi) \propto \left(\prod_{d=1}^D \frac{b^a}{\Gamma(a)} (\sigma^2)^{-a-1} \exp \left(-\frac{b}{\sigma^2} \right) \right) \exp \left\langle -\frac{D}{2} \log(\sigma^2) - \frac{1}{2\sigma^2} (\mathbf{x} - \mathbf{A}\mathbf{s})^T (\mathbf{x} - \mathbf{A}\mathbf{s}) \right\rangle_{q_{\mathbf{s}}(\mathbf{s})q_{-\Psi}(\theta)}$$

We see:

$$q_\Psi(\Psi) \propto \prod_{d=1}^D \frac{b^a}{\Gamma(a)} (\sigma^2)^{-(a+\frac{1}{2})-1} \exp \left(-\frac{b + \frac{2}{D} \left\langle (\mathbf{x} - \mathbf{A}\mathbf{s})^T (\mathbf{x} - \mathbf{A}\mathbf{s}) \right\rangle_{q_{\mathbf{s}}(\mathbf{s})q_{\mathbf{A}}(\mathbf{A})}}{\sigma^2} \right)$$

where $\mathbf{A}_{d:} \in \mathbb{R}^{1 \times K}$ is the d^{th} row of \mathbf{A} .

Thus,

$$q_\Psi(\Psi) = \prod_{d=1}^D \text{InvGamma} \left(\sigma^2 \left| a + \frac{1}{2}, b + \frac{2}{D} \left\langle (\mathbf{x} - \mathbf{A}\mathbf{s})^T (\mathbf{x} - \mathbf{A}\mathbf{s}) \right\rangle_{q_{\mathbf{s}}(\mathbf{s})q_{\mathbf{A}}(\mathbf{A})} \right. \right)$$

For $q_{\mu_k}(\mu_k)$:

$$q_{\mu_k}(\mu_k) \propto P(\mu_k) \exp \langle \log P(\mathbf{x}, \mathbf{s}, \pi, \mathbf{A}, \Psi | \eta) \rangle_{q_{\mathbf{s}}(\mathbf{s})q_{-\mu_k}(\theta)}$$

Substituting the appropriate terms:

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

$$q_{\mu_k}(\mu_k) \propto \exp \left(-\frac{1}{2} \left(\mu_k^T \left(\frac{1}{\alpha_k^2} + \frac{\langle s_k \rangle_{q_{s_k}}}{\langle \sigma^2 \rangle_{q_\Psi}} \right) \mu_k + 2 \langle s_k \rangle_{q_{s_k}} \left(-\mathbf{x}^T + \sum_{k'=1, k' \neq k}^K \langle s_{k'} \rangle_{q_{s_{k'}}} \langle \mu_{k'} \rangle_{q_{\mu_{k'}}}^T \right) \mu_k \right) \right)$$

We see that

$$q_{\mu_k}(\mu_k) = \mathcal{N}(\mu_{\mu_k}, \Sigma_{\mu_k})$$

where:

$$\Sigma_{\mu_k} = \left(\frac{1}{\alpha_k^2} + \frac{\langle s_k \rangle_{q_{s_k}}}{\langle \sigma^2 \rangle_{q_\Psi}} \right)^{-1} \mathbf{I}$$

and

$$\mu_{\mu_k} = \Sigma_{\mu_k} \left(\mathbf{x}^T - \sum_{k'=1, k' \neq k}^K \langle s_{k'} \rangle_{q_{s_{k'}}} \langle \mu_{k'} \rangle_{q_{\mu_{k'}}}^T \right) \langle s_k \rangle_{q_{s_k}}$$

Optimising the parameters $\{a, b\}$, and α^2 in turn, will cause some α_i^2 to diverge, the number of remaining α_i^2 provide our determination for the value of K . This is automatic relevance determination through factor analysis.

The Python code for variational Bayes:

```

1 from abc import ABC, abstractmethod
2 from dataclasses import dataclass
3
4 import numpy as np
5
6 from src.models.binary_latent_factor_model import (
7     AbstractBinaryLatentFactorModel,
8     BinaryLatentFactorApproximation,
9 )
10
11
12 @dataclass
13 class Distribution(ABC):
14     @property
15     @abstractmethod
16     def mean(self):
17         pass
18
19
20 @dataclass
21 class Beta(Distribution):
22     alpha: np.ndarray
23     beta: np.ndarray
24
25     @property
26     def mean(self) -> np.ndarray:
27         return np.divide(self.alpha, (self.alpha + self.beta))
28
29
30 @dataclass
31 class InverseGamma(Distribution):
32     a: float
33     b: float
34
35     @property
36     def mean(self) -> float:
37         return self.b / (self.a - 1)
38
39
40 @dataclass
41 class Gaussian(Distribution):
42     mu: np.ndarray # (number_of_dimensions, number_of_latent_variables)
43     variance: np.ndarray # (number_of_latent_variables, )
44
45     @property
46     def precision(self) -> np.ndarray:
47         return 1 / self.variance
48
49     @property
50     def mean(self) -> np.ndarray:
51         return self.mu
52
53
54 class VariationalBayesBinaryLatentFactorModel(AbstractBinaryLatentFactorModel):
55     def __init__(self, mu: Gaussian, variance: InverseGamma, pi: Beta):
56         self._mu = mu
57         self._variance = variance
58         self._pi = pi
59
60     @property
61     def variance(self) -> float:
62         return self._variance.mean
63
64     @property
65     def pi(self) -> np.ndarray:
66         return self._pi.mean
67
68     @property
69     def mu(self) -> np.ndarray:
70         return self._mu.mean
71
72     def _update_pi(
73         self,
74         binary_latent_factor_approximation: BinaryLatentFactorApproximation,
75     ):
76         self._pi.alpha += np.sum(
77             binary_latent_factor_approximation.expectation_s, axis=0
78         ).reshape(1, -1)
79         self._pi.beta += binary_latent_factor_approximation.n - np.sum(
80             binary_latent_factor_approximation.expectation_s, axis=0
81         ).reshape(1, -1)
82
83     def _update_variance(
84         self,
85         x: np.ndarray, # (number_of_points, number_of_dimensions)
86         binary_latent_factor_approximation: BinaryLatentFactorApproximation,
87     ):
88         # expectation_s (number_of_points, number_of_latent_variables)
89         self._variance.a += (
90             0.5
91             * binary_latent_factor_approximation.n
92             * binary_latent_factor_approximation.k
93         )
94         # self._variance.b += 0.5 * np.mean(

```

```

95     # (x - binary_latent_factor_approximation.expectation_s @ self.mu.T) ** 2
96     # )
97     self._variance.b += 2 * np.sum(
98         x - binary_latent_factor_approximation.expectation_s @ self.mu.T
99     )
100
101 def _update_mu_k(
102     self,
103     x: np.ndarray, # (number_of_points, number_of_dimensions)
104     binary_latent_factor_approximation: BinaryLatentFactorApproximation,
105     k: int, # latent dimension
106 ):
107     # expectation_s (number_of_points, number_of_latent_variables)
108     # expectation_ss (number_of_latent_variables, number_of_latent_variables)
109     self._mu.variance[k] = 1 / (
110         self._mu.precision[k]
111         + np.mean(binary_latent_factor_approximation.expectation_s[:, k])
112         * self._mu.precision
113     )
114
115     # (number_of_points, number_of_latent_variables - 1)
116     es_except_k = np.concatenate(
117         (
118             binary_latent_factor_approximation.expectation_s[:, :k],
119             binary_latent_factor_approximation.expectation_s[:, k + 1 :],
120         ),
121         axis=1,
122     )
123
124     # (number_of_dimensions, number_of_latent_variables - 1)
125     mu_except_k = np.concatenate((self.mu[:, :k], self.mu[:, k + 1 :]), axis=1)
126
127     # (number_of_dimensions x 1)
128     self._mu.mu[:, k] = self._mu.variance[k] * (
129         (
130             x # (number_of_points, number_of_dimensions)
131             - es_except_k # (number_of_points, number_of_latent_variables - 1)
132             @ mu_except_k.T # (number_of_dimensions, number_of_latent_variables - 1)
133         ).T # (number_of_dimensions, number_of_points)
134         @ binary_latent_factor_approximation.expectation_s[
135             :, k
136         ] # (number_of_points, 1)
137     )
138
139 def maximisation_step(
140     self,
141     x: np.ndarray,
142     binary_latent_factor_approximation: BinaryLatentFactorApproximation,
143 ) -> None:
144     self._update_pi(binary_latent_factor_approximation)
145     self._update_variance(x, binary_latent_factor_approximation)
146     # es = binary_latent_factor_approximation.expectation_s
147     # ess = binary_latent_factor_approximation.expectation_ss
148     # n = binary_latent_factor_approximation.n
149     # self._pi = np.mean(ess, axis=0, keepdims=True)
150     # self._sigma = np.sqrt((np.trace(np.dot(x.T, x)) + np.trace(np.dot(np.dot(self.mu.T, self.mu), ess)))
151     # - 2 * np.trace(np.dot(np.dot(es.T, x), self.mu))) / (n * self.d))
152     for k in range(self.k):
153         self._update_mu_k(x, binary_latent_factor_approximation, k)

```

src/models/variational_bayes.py

The rest of the Python code for question 4:

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 from src.models.binary_latent_factor_model import (
5     BinaryLatentFactorModel,
6     learn_binary_factors,
7 )
8 from src.models.mean_field_approximation import init_mean_field_approximation
9 from src.models.variational_bayes import (
10     Beta,
11     Gaussian,
12     InverseGamma,
13     VariationalBayesBinaryLatentFactorModel,
14 )
15
16 def b(
17     x: np.ndarray,
18     k: int,
19     em_observations: int,
20     e_maximum_steps: int,
21     e_convergence_criterion: float,
22     # mu: Gaussian,
23     # variance: InverseGamma,
24     # pi: Beta,
25     save_path: str,
26 ) -> None:
27     n = x.shape[0]
28     mean_field_approximation = init_mean_field_approximation(
29         k, n, max_steps=e_maximum_steps, convergence_criterion=e_convergence_criterion
30     )
31     (
32         mu_max_step,
33         sigma,
34         pi_max_step,
35     ) = BinaryLatentFactorModel.calculate_maximisation_parameters(
36         x, mean_field_approximation
37     )
38     mu = Gaussian(
39         mu=mu_max_step,
40         variance=mu_max_step.std(axis=0) ** 2,
41     )
42     variance = InverseGamma(
43         a=2,
44         b=sigma**2,
45     )
46     pi = Beta(
47         alpha=np.ones((1, k)),
48         beta=np.divide(1 - pi_max_step, pi_max_step),
49     )
50     binary_latent_factor_model = VariationalBayesBinaryLatentFactorModel(
51         mu=mu,
52         variance=variance,
53         pi=pi,
54     )
55     fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
56     for i in range(k):
57         ax[i].imshow(binary_latent_factor_model.mu[:, i].reshape(4, 4))
58         ax[i].set_title(f"Latent Feature mu-{i}")
59     fig.suptitle("Initial Features (Variational Bayes)")
60     plt.tight_layout()
61     plt.savefig(save_path + "-init-latent-factors", bbox_inches="tight")
62     plt.close()
63     (
64         mean_field_approximation,
65         binary_latent_factor_model,
66         free_energy,
67     ) = learn_binary_factors(
68         x=x,
69         em_observations=em_observations,
70         binary_latent_factor_model=binary_latent_factor_model,
71         binary_latent_factor_approximation=mean_field_approximation,
72     )
73     fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
74     for i in range(k):
75         ax[i].imshow(binary_latent_factor_model.mu[:, i].reshape(4, 4))
76         ax[i].set_title(f"Latent Feature mu-{i}")
77     fig.suptitle("Learned Features (Variational Bayes)")
78     plt.tight_layout()
79     plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
80     plt.close()
81     plt.title("Free Energy (Variational Bayes)")
82     plt.xlabel("t (EM steps)")
83     plt.ylabel("Free Energy")
84     plt.plot(free_energy)
85     plt.savefig(save_path + "-free-energy", bbox_inches="tight")
86     plt.close()

```

src/solutions/q4.py

Question 5

(a)

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

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

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

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

Expanding:

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

Collecting terms pertaining to s_i :

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

where C are all other terms without s_i .

Knowing that $s_i^2 = s_i$:

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

Thus:

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

where the factors are defined:

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

and

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

as required.

The Boltzmann Machine can be defined:

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

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

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

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

and

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

and

$$\log Z = -C$$

(b)

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

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

Thus,

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

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

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

where

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

and

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

Thus,

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

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

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

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

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

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

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

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

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

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

Simplifying:

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

and,

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

Thus, the cavity distributions are:

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

and

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

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

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

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

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

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

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

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

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

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

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

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

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

We can write:

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

Simplifying:

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

Thus, the first moments:

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

and

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

Moreover:

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

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

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

Thus, the first moment:

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

Simplifying:

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

Similarly:

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

By setting:

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

and

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

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

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

Simplifying:

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

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

Our parameter update:

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

Similarly:

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

(c)

Using factored approximate messages, we see that:

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

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

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

and

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

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

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

(d)

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

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

Question 6

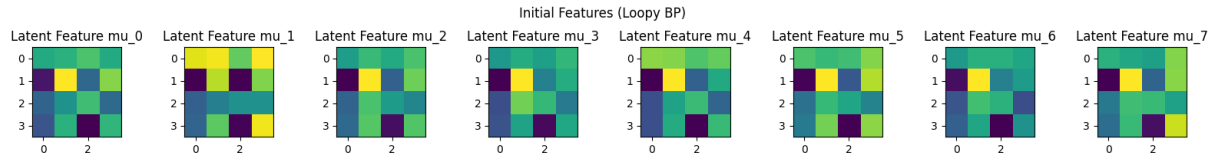


Figure 20: Initial Latent factors learned with EP/Loopy-BP

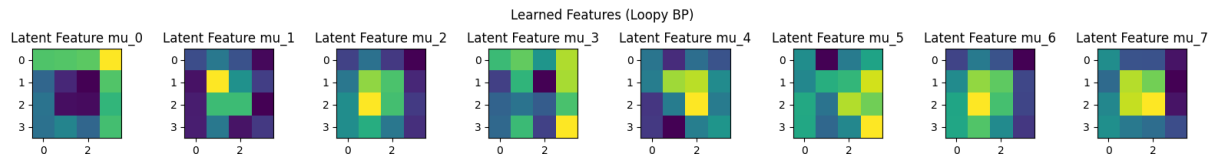


Figure 21: Learned Latent factors learned with EP/Loopy-BP

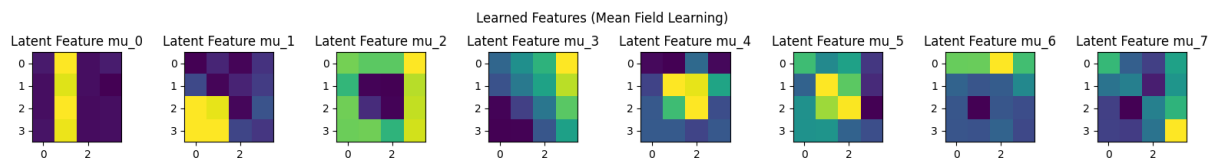


Figure 22: Learned Latent Factors with Mean Field Approximation

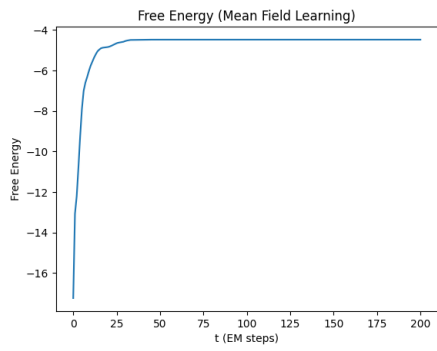


Figure 23: Mean Field Approximation

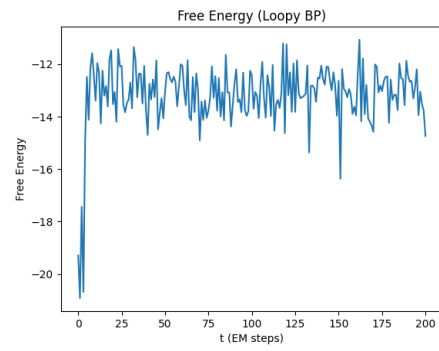


Figure 24: Loopy BP

The Python code for the Boltzmann machine:

```

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

```

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

```



```

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

```

src/models/message_passing.py

The rest of the Python code for question 6:

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

src/solutions/q6.py

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

```

93     log_tau=jnp.log(1),
94     log_zeta=jnp.log(2),
95 )
96 gaussian_process_parameters = GaussianProcessParameters(
97     kernel=asdict(init_kernel_parameters),
98     log_sigma=jnp.log(1),
99 )
100 years_to_predict = 15
101 t_new = t[-1] + np.linspace(0, years_to_predict, years_to_predict * 12)
102 t_test = np.concatenate((t, t_new))
103 q2.f(
104     t_train=t,
105     y_train=y,
106     t_test=t_test,
107     min_year=np.min(df_co2.decimal.values[:]),
108     prior_linear_regression_parameters=prior_linear_regression_parameters,
109     linear_regression_sigma=sigma,
110     kernel=kernel,
111     gaussian_process_parameters=gaussian_process_parameters,
112     learning_rate=1e-2,
113     number_of_iterations=100,
114     save_path=os.path.join(Q2.OUTPUT_FOLDER, "f"),
115 )
116
117 # Question 3
118 Q3.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q3")
119 if not os.path.exists(Q3.OUTPUT_FOLDER):
120     os.makedirs(Q3.OUTPUT_FOLDER)
121 number_of_images = 2000
122 x = generate_images(n=number_of_images)
123 k = 8
124 em_iterations = 200
125 e_maximum_steps = 100
126 e_convergence_criterion = 0
127
128 binary_latent_factor_model = q3.e_and_f(
129     x=x,
130     k=k,
131     em_iterations=em_iterations,
132     e_maximum_steps=e_maximum_steps,
133     e_convergence_criterion=e_convergence_criterion,
134     save_path=os.path.join(Q3.OUTPUT_FOLDER, "f"),
135 )
136 - = q3.e_and_f(
137     x=x,
138     k=int(k * 1.5),
139     em_iterations=em_iterations,
140     e_maximum_steps=e_maximum_steps,
141     e_convergence_criterion=e_convergence_criterion,
142     save_path=os.path.join(Q3.OUTPUT_FOLDER, "f-larger-k"),
143 )
144
145 q3.g(
146     x=x[:, 1, :],
147     binary_latent_factor_model=binary_latent_factor_model,
148     sigmas=[1, 2, 3],
149     k=k,
150     em_iterations=em_iterations,
151     e_maximum_steps=e_maximum_steps,
152     e_convergence_criterion=e_convergence_criterion,
153     save_path=os.path.join(Q3.OUTPUT_FOLDER, "g"),
154 )
155
156 # Question 4
157 Q4.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q4")
158 if not os.path.exists(Q4.OUTPUT_FOLDER):
159     os.makedirs(Q4.OUTPUT_FOLDER)
160 d = x.shape[1]
161 pi = Beta(
162     alpha=np.random.gamma(1, size=(1, k)),
163     beta=np.random.gamma(1, size=(1, k)),
164 )
165 variance = InverseGamma(
166     a=2,
167     b=1,
168 )
169 mu = Gaussian(
170     mu=np.zeros((d, k)),
171     variance=np.random.uniform(size=(k,)) + 1,
172 )
173
174 q4.b(
175     x=x,
176     k=k,
177     em_iterations=em_iterations,
178     e_maximum_steps=e_maximum_steps,
179     e_convergence_criterion=e_convergence_criterion,
180     # mu=mu,
181     # variance=variance,
182     # pi=pi,
183     save_path=os.path.join(Q4.OUTPUT_FOLDER, "b"),
184 )
185
186 # Question 6
187 Q6.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q6")
188 if not os.path.exists(Q6.OUTPUT_FOLDER):
189     os.makedirs(Q6.OUTPUT_FOLDER)
190 q6.run(x, k, em_iterations, save_path=os.path.join(Q6.OUTPUT_FOLDER, "all"))

```

main.py

Appendix 3: generate_images.py

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

src/generate_images.py

Appendix 4: MStep.py

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

demo_code/MStep.py