

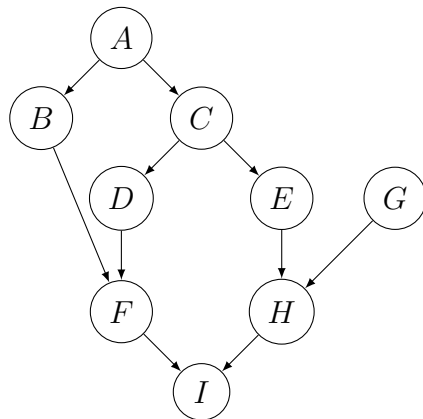
COMP0085 Summative Assignment

Jan 4, 2023

Question 1

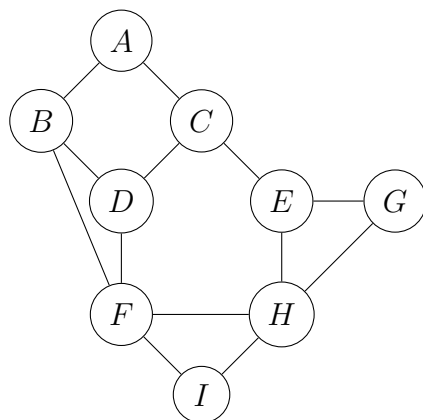
(a)

The directed acyclic graph:

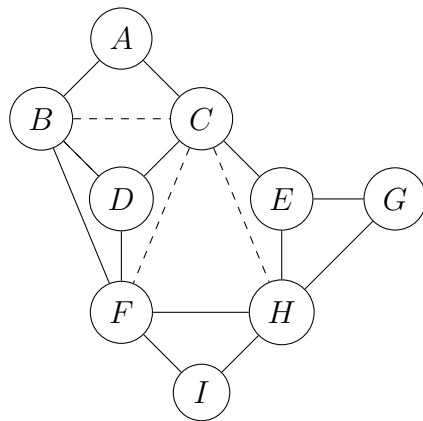


(b)

The moralised graph:

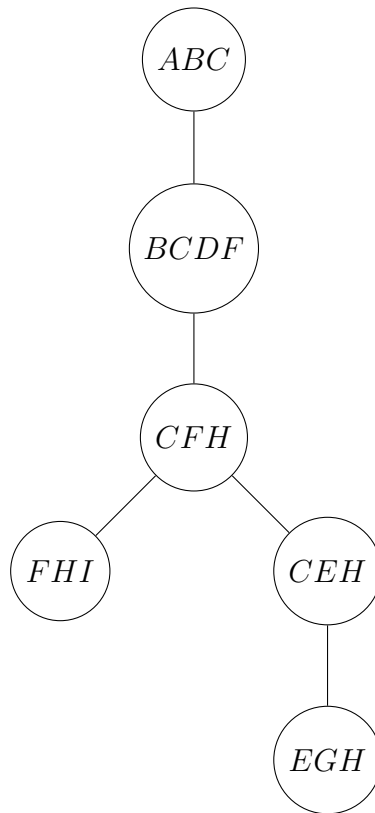


An effective triangulation:



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

The resulting junction tree:



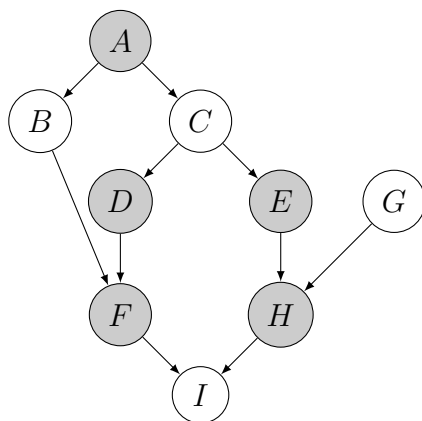
where the circular nodes are cliques.

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:

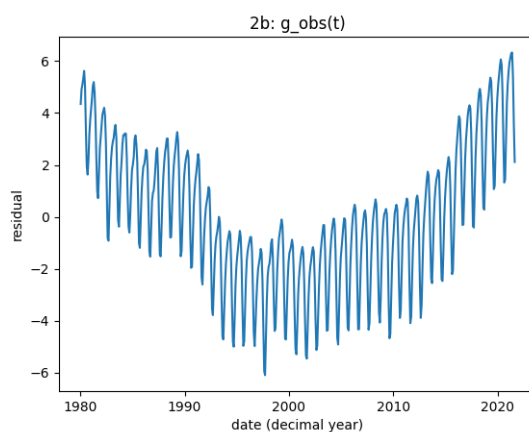


Figure 3: $g_{obs}(t)$

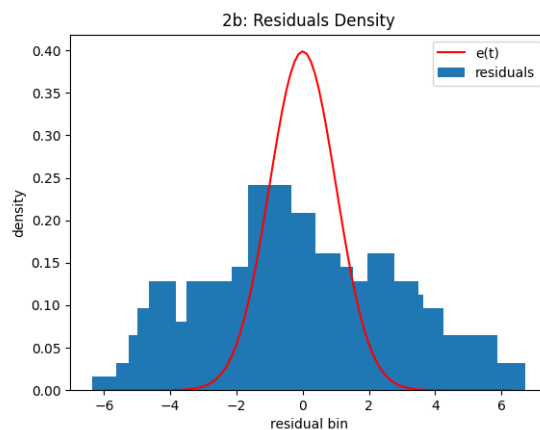


Figure 4: Density Estimation of Residuals vs $e(t) \sim \mathcal{N}(0, 1)$

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:

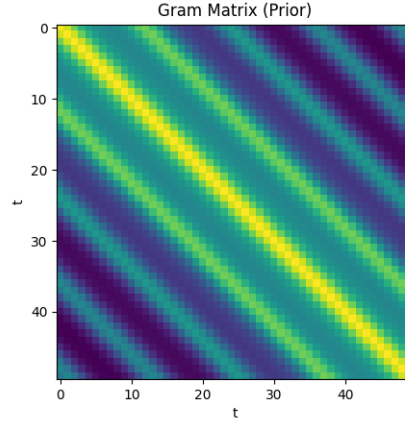


Figure 5: Covariance Matrix

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.

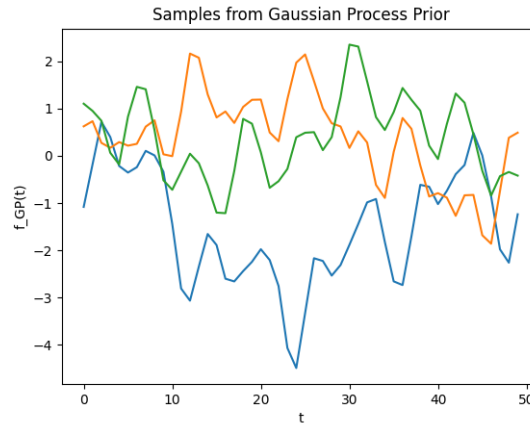


Figure 6: Samples from a zero mean GP with the provided covariance kernel

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

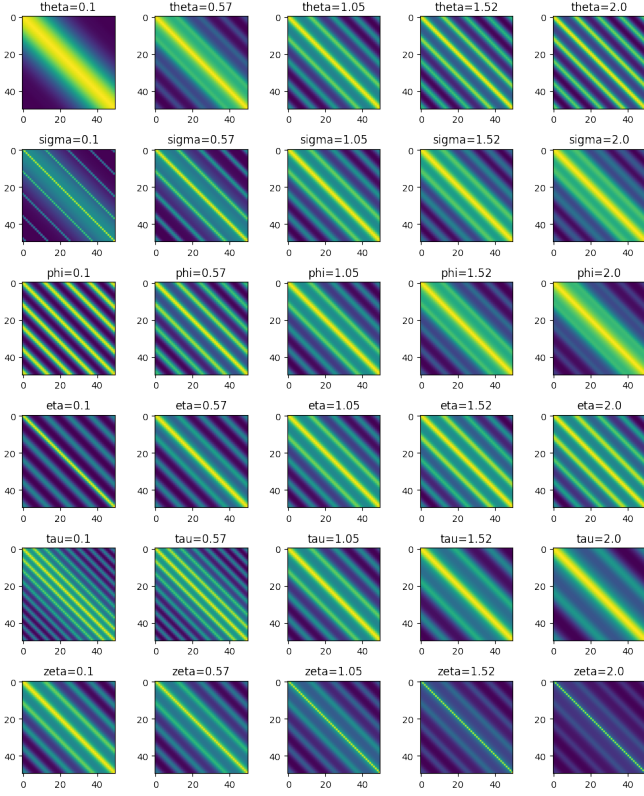


Figure 7: Covariances for different parameters

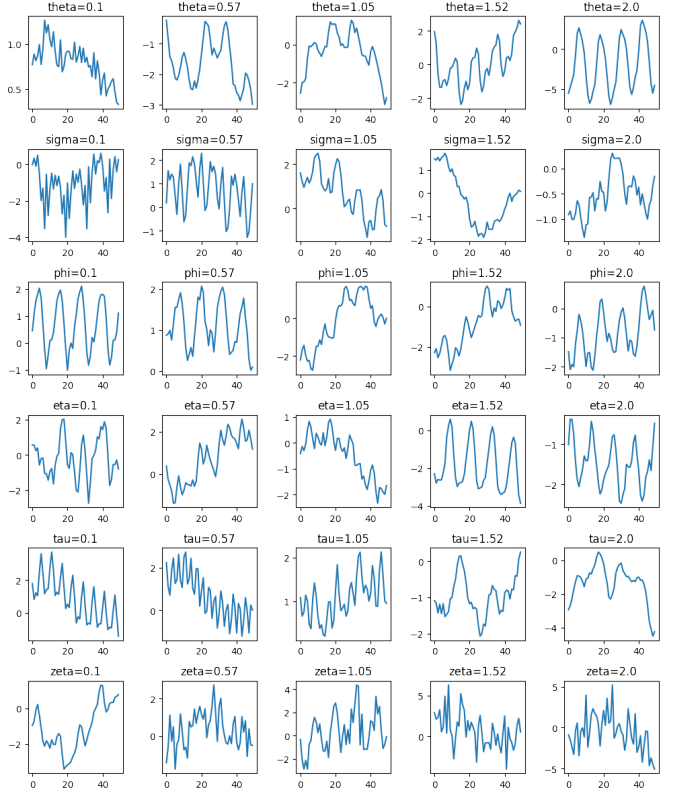


Figure 8: Samples for different parameters

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 $\exp\left(-\frac{2\sin^2(\pi(s-t)/\tau)}{\sigma^2}\right)$.

σ : As σ increases, we see smoother 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.

ϕ : As ϕ increases, we see less smooth 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.

η :

τ :

ζ :

(e)

(f)

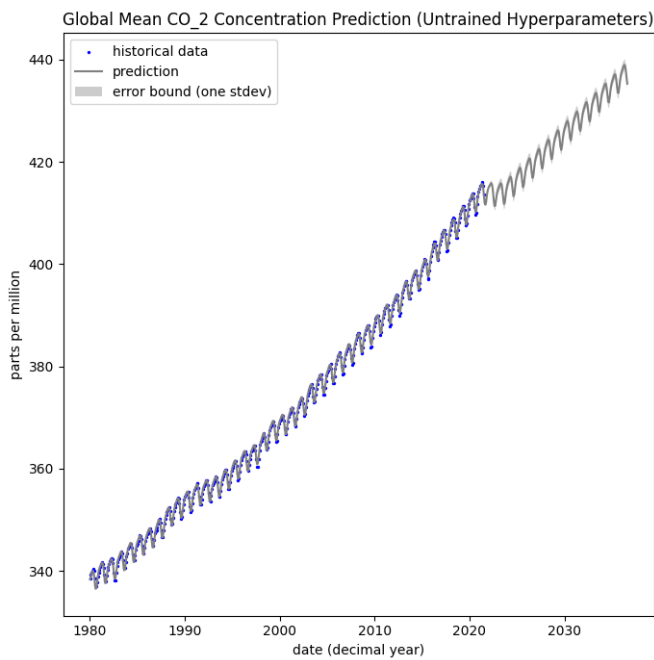


Figure 9: Without hyperparameter tuning

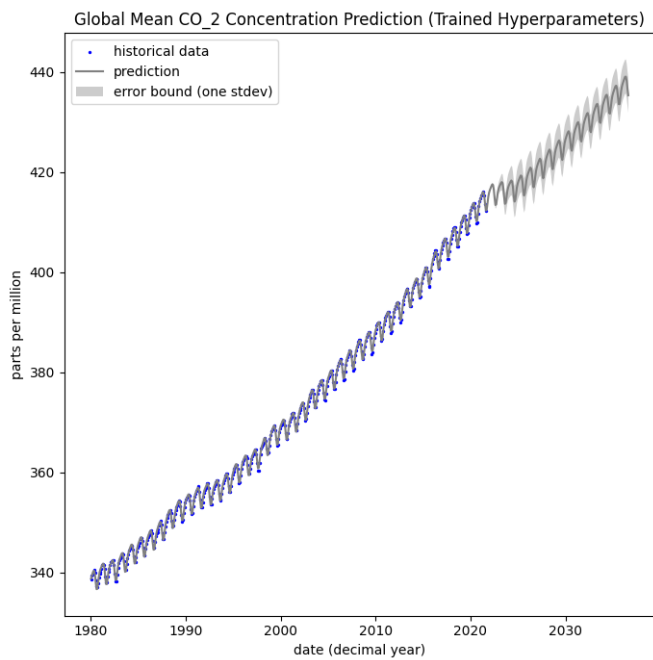


Figure 10: With hyperparameter tuning

(g)

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):
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):
26         return self.sigma**2
27
28     @property
29     def precision(self):
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     @property
184     def sigma(self) -> float:
185         return jnp.exp(self.log_sigma)
186
187     @theta.setter
188     def theta(self, value: float) -> None:
189         self.log_theta = jnp.log(value)
190

```

```

191 @sigma.setter
192 def sigma(self, value: float) -> None:
193     self.log_sigma = jnp.log(value)
194
195 @phi.setter
196 def phi(self, value: float) -> None:
197     self.log_phi = jnp.log(value)
198
199 @eta.setter
200 def eta(self, value: float) -> None:
201     self.log_eta = jnp.log(value)
202
203 @tau.setter
204 def tau(self, value: float) -> None:
205     self.log_tau = jnp.log(value)
206
207 @zeta.setter
208 def zeta(self, value: float) -> None:
209     self.log_zeta = jnp.log(value)
210
211
212 class CombinedKernel(Kernel):
213     """
214     The kernel defined as:
215      $k(x, y) = \theta^2 * (\exp(-(2\sin^2(\pi(x-y)/\tau))/(\sigma^2)) + \phi^2 * \exp(-(x-y)^2/(2 * \eta^2)))$ 
216      $+ \zeta^2 * \delta(x=y)$ 
217     """
218
219     Parameters = CombinedKernelParameters
220
221     def _kernel(
222         self,
223         parameters: CombinedKernelParameters,
224         x: jnp.ndarray,
225         y: jnp.ndarray,
226     ) -> jnp.ndarray:
227         """Kernel evaluation between a single feature x and a single feature y.
228
229         Args:
230             parameters: parameters dataclass for the Gaussian kernel
231             x: ndarray of shape (1,)
232             y: ndarray of shape (1,)
233
234         Returns:
235             The kernel evaluation.
236         """
237         return jnp.dot(
238             jnp.ones(1),
239             (
240                 (parameters.theta**2)
241                 * (
242                     (
243                         jnp.exp(
244                             (-2 * jnp.sin(jnp.pi * (x - y) / parameters.tau) ** 2)
245                             / (parameters.sigma**2)
246                         )
247                     )
248                     + (parameters.phi**2)
249                     * (jnp.exp(-((x - y) ** 2) / (2 * parameters.eta**2)))
250                     + parameters.zeta**2 * (x == y)
251                 )
252             ),
253         )

```

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 import optax
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 pandas as pd
9 import scipy
10
11 from src.models.bayesian_linear_regression import (
12     LinearRegressionParameters,
13     Theta,
14     compute_linear_regression_posterior,
15 )
16 from src.models.gaussian_process_regression import (
17     GaussianProcess,
18     GaussianProcessParameters,
19 )
20 from src.models.kernels import CombinedKernel, CombinedKernelParameters
21
22 jax.config.update("jax_enable_x64", True)
23
24
25 def construct_design_matrix(t: np.ndarray):
26     return np.stack((t, np.ones(t.shape)), axis=1).T
27
28
29 def a(
30     t: np.ndarray,
31     y: np.ndarray,
32     sigma: float,
33     prior_linear_regression_parameters: LinearRegressionParameters,
34     save_path: str,
35 ) -> LinearRegressionParameters:
36     x = construct_design_matrix(t)
37     prior_theta = Theta(
38         linear_regression_parameters=prior_linear_regression_parameters,
39         sigma=sigma,
40     )
41     posterior_linear_regression_parameters = compute_linear_regression_posterior(
42         x,
43         y,
44         prior_linear_regression_parameters,
45         residuals_precision=prior_theta.precision,
46     )
47     df_mean = pd.DataFrame(
48         posterior_linear_regression_parameters.mean, columns=["value"]
49     )
50     df_mean.index = ["a", "b"]
51     df_mean = pd.concat([df_mean], keys=["parameters"])
52     dfi.export(df_mean, save_path + "-mean.png")
53
54     df_covariance = pd.DataFrame(
55         posterior_linear_regression_parameters.covariance, columns=["a", "b"]
56     )
57     df_covariance.index = ["a", "b"]
58     df_covariance = pd.concat([df_covariance], keys=["parameters"])
59     df_covariance = pd.concat([df_covariance.T], keys=["parameters"])
60     dfi.export(df_covariance, save_path + "-covariance.png")
61     return posterior_linear_regression_parameters
62
63
64 def b(
65     t_year,
66     t,
67     y,
68     linear_regression_parameters: LinearRegressionParameters,
69     error_mean,
70     error_variance,
71     save_path,
72 ):
73     x = construct_design_matrix(t)
74     residuals = y - linear_regression_parameters.predict(x)
75     plt.plot(t_year.reshape(-1), residuals.reshape(-1))
76     plt.xlabel("date (decimal year)")
77     plt.ylabel("residual")
78     plt.title("2b: g-obs(t)")
79     plt.savefig(save_path + "-residuals-timeseries")
80     plt.close()
81
82     count, bins = np.histogram(residuals, bins=100, density=True)
83     plt.bar(bins[1:], count, label="residuals")
84     plt.plot(
85         bins[1:],
86         scipy.stats.norm.pdf(bins[1:], loc=error_mean, scale=error_variance),
87         color="red",
88         label="e(t)",
89     )
90     plt.xlabel("residual bin")
91     plt.ylabel("density")
92     plt.title("2b: Residuals Density")
93     plt.legend()
94     plt.savefig(save_path + "-residuals-density-estimation")

```

```

95 plt.close()
96
97
98 def c(
99     kernel: CombinedKernel,
100     kernel_parameters: CombinedKernelParameters,
101     log_theta_range: np.ndarray,
102     t: np.ndarray,
103     number_of_samples: int,
104     save_path: str,
105 ):
106     gram = kernel(t, **asdict(kernel_parameters))
107     plt.imshow(gram)
108     plt.xlabel("t")
109     plt.ylabel("t")
110     plt.title("Gram Matrix (Prior)")
111     plt.savefig(save_path + "-gram-matrix")
112     plt.close()
113
114     for _ in range(number_of_samples):
115         plt.plot(
116             np.random.multivariate_normal(
117                 jnp.zeros(gram.shape[0]), gram, size=1
118             ).reshape(-1)
119         )
120     plt.xlabel("t")
121     plt.ylabel("f.GP(t)")
122     plt.title("Samples from Gaussian Process Prior")
123     plt.savefig(save_path + "-samples")
124     plt.close()
125
126     fig_samples, ax_samples = plt.subplots(
127         len(fields(kernel_parameters.__class__)),
128         len(log_theta_range),
129         figsize=(
130             len(log_theta_range) * 2,
131             len(fields(kernel_parameters.__class__)) * 2,
132         ),
133         frameon=False,
134     )
135     for i, field in enumerate(fields(kernel_parameters.__class__)):
136         default_value = getattr(kernel_parameters, field.name)
137         for j, log_value in enumerate(log_theta_range):
138             setattr(kernel_parameters, field.name, log_value)
139             gram = kernel(t, **asdict(kernel_parameters))
140             ax_samples[i][j].plot(
141                 np.random.multivariate_normal(
142                     jnp.zeros(gram.shape[0]), gram, size=1
143                 ).reshape(-1),
144             )
145             ax_samples[i][j].set_title(
146                 f"{field.name.strip('log-')}={np.round(np.exp(log_value), 2)}"
147             )
148             setattr(kernel_parameters, field.name, default_value)
149     plt.tight_layout()
150     plt.savefig(save_path + f"-parameter-samples", bbox_inches="tight")
151     plt.close(fig_samples)
152
153     fig_gram, ax_gram = plt.subplots(
154         len(fields(kernel_parameters.__class__)),
155         len(log_theta_range),
156         figsize=(
157             len(log_theta_range) * 2,
158             len(fields(kernel_parameters.__class__)) * 2,
159         ),
160         frameon=False,
161     )
162     for i, field in enumerate(fields(kernel_parameters.__class__)):
163         default_value = getattr(kernel_parameters, field.name)
164         for j, log_value in enumerate(log_theta_range):
165             setattr(kernel_parameters, field.name, log_value)
166             gram = kernel(t, **asdict(kernel_parameters))
167             ax_gram[i][j].imshow(gram)
168             ax_gram[i][j].set_title(
169                 f"{field.name.strip('log-')}={np.round(np.exp(log_value), 2)}"
170             )
171             setattr(kernel_parameters, field.name, default_value)
172     plt.tight_layout()
173     plt.savefig(save_path + f"-parameter-grams", bbox_inches="tight")
174     plt.close(fig_gram)
175
176
177 def f(
178     t_train: np.ndarray,
179     y_train: np.ndarray,
180     t_test: np.ndarray,
181     min_year: float,
182     prior_linear_regression_parameters: LinearRegressionParameters,
183     linear_regression_sigma: float,
184     kernel: CombinedKernel,
185     gaussian_process_parameters: GaussianProcessParameters,
186     learning_rate: float,
187     number_of_iterations: int,
188     save_path: str,
189 ):
190     # Train Bayesian Linear Regression

```

```

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

```

```
287 plt.legend()
288 plt.tight_layout()
289 plt.savefig(save_path + "-extrapolation-trained", bbox_inches="tight")
290 plt.close()
```

src/solutions/q2.py

Question 3

(a)

The free energy is can be calculated as:

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

Knowing,

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

we can write:

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

Moreover, our mean field approximation:

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

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

To compute the first term:

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

substituting the appropriate terms:

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

with d being the number of dimensions.

Taking the logarithm:

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

The expectation distributed to the relevant terms:

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

Evaluating the expectations:

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

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

To compute the second term:

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

Taking the logarithm:

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

The expectation distributed to the relevant terms:

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

Evaluating the expectations:

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

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

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

Thus,

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

Substituting the appropriate values:

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

Combining, we have our free energy expression:

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

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

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

where λ^{LG} is the Lagrange multiplier.

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

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

Similar to our free energy derivation:

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

and

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

We can write:

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

Substituting the relevant terms:

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

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

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

Thus,

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

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

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

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

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

Solving for λ_i :

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

we have our partial update.

(b)

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

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

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

This mimics the least squares solution:

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

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

(c)

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

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

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

(d)

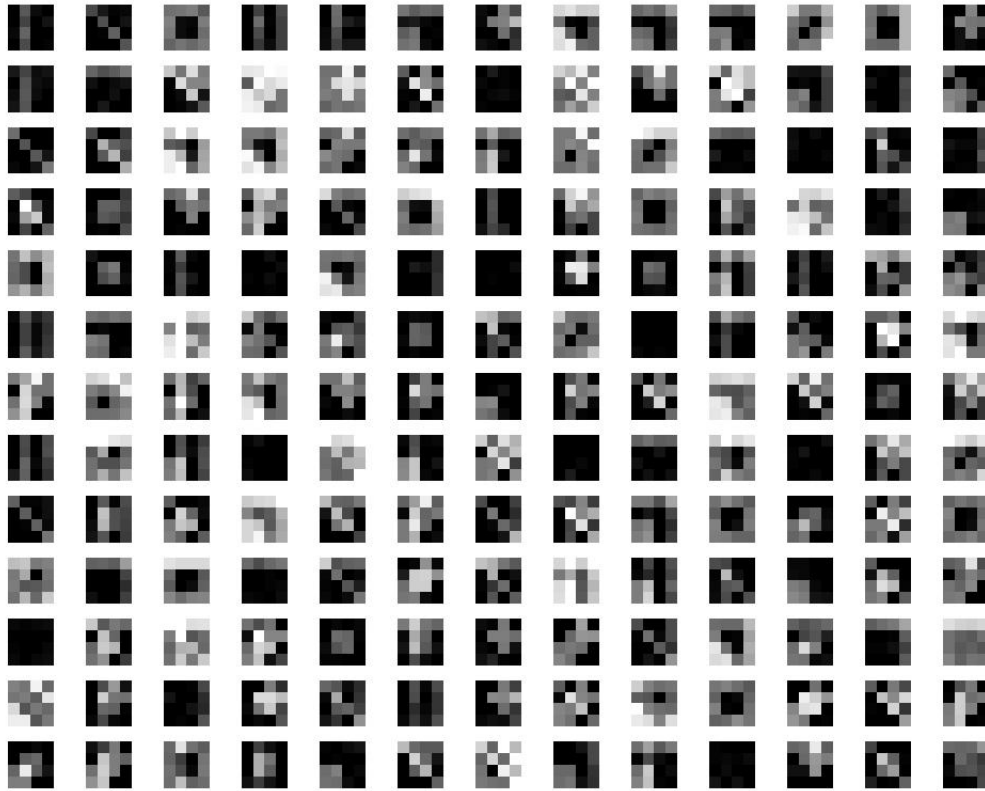


Figure 11: Images generated by randomly combined features with noise

(e)

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

(f)

(g)

The Python code for mean field learning:

```
1 from dataclasses import dataclass
2
3 import numpy as np
4
5 from demo_code.MStep import m_step
6
7
8 @dataclass
9 class MeanFieldApproximation:
10     """
11     lambda_matrix: parameters variational approximation (number_of_points, number_of_latent_variables)
12     """
13
14     lambda_matrix: np.ndarray
15
16     def lambda_matrix_exclude(self, exclude_latent_index: int) -> np.ndarray:
17         # (number_of_points, number_of_latent_variables-1)
18         return np.concatenate(
19             (
20                 self.lambda_matrix[:, :exclude_latent_index],
21                 self.lambda_matrix[:, exclude_latent_index + 1 :],
22             ),
23             axis=1,
24         )
25
26     @property
27     def log_lambda_matrix(self):
28         return np.log(self.lambda_matrix)
29
30     @property
31     def log_one_minus_lambda_matrix(self):
32         return np.log(1 - self.lambda_matrix)
33
34     @property
35     def n(self):
36         return self.lambda_matrix.shape[0]
37
38     @property
39     def k(self):
40         return self.lambda_matrix.shape[1]
41
42
43 def init_mean_field_approximation(k: int, n: int) -> MeanFieldApproximation:
44     return MeanFieldApproximation(
45         lambda_matrix=np.random.random(size=(n, k)),
46     )
47
48
49 @dataclass
50 class BinaryLatentFactorModel:
51     """
52     mu: matrix of means (number_of_dimensions, number_of_latent_variables)
53     sigma: gaussian noise parameter
54     pi: vector of priors (1, number_of_latent_variables)
55     """
56
57     mu: np.ndarray
58     sigma: float
59     pi: np.ndarray
60
61     def mu_exclude(self, exclude_latent_index: int) -> np.ndarray:
62         # (number_of_dimensions, number_of_latent_variables-1)
63         return np.concatenate(
64             (self.mu[:, :exclude_latent_index], self.mu[:, exclude_latent_index + 1 :]),
65             axis=1,
66         )
67
68     @property
69     def log_pi(self):
70         return np.log(self.pi)
71
72     @property
73     def log_one_minus_pi(self):
74         return np.log(1 - self.pi)
75
76     @property
77     def variance(self):
78         return self.sigma**2
79
80     @property
81     def precision(self):
82         return 1 / self.variance
83
84     @property
85     def d(self):
86         return self.mu.shape[0]
87
88     @property
89     def k(self):
90         return self.mu.shape[1]
91
92
93 def init_binary_latent_factor_model(
94     x: np.ndarray,
```

```

95     mean_field_approximation: MeanFieldApproximation,
96 ) -> BinaryLatentFactorModel:
97     return maximisation_step(x, mean_field_approximation)
98
99
100 def _compute_expectation_log_p_x_s_given_theta(
101     x: np.ndarray,
102     binary_latent_factor_model: BinaryLatentFactorModel,
103     mean_field_approximation: MeanFieldApproximation,
104 ) -> float:
105     """
106     The first term of the free energy, the expectation of log P(X,S|theta)
107
108     :param x: data matrix (number_of_points, number_of_dimensions)
109     :param binary_latent_factor_model: a binary_latent_factor_model
110     :param mean_field_approximation: a mean_field_approximation
111     :return: the expectation of log P(X,S|theta)
112     """
113     # (number_of_points, number_of_dimensions)
114     mu_lambda = mean_field_approximation.lambda_matrix @ binary_latent_factor_model.mu.T
115
116     # (number_of_latent_variables, number_of_latent_variables)
117     expectation_s_i_s_j_mu_i_mu_j = np.multiply(
118         mean_field_approximation.lambda_matrix.T
119         @ mean_field_approximation.lambda_matrix,
120         binary_latent_factor_model.mu.T @ binary_latent_factor_model.mu,
121     )
122
123     expectation_log_p_x_given_s_theta = -(
124         mean_field_approximation.n * binary_latent_factor_model.d / 2
125     ) * np.log(2 * np.pi * binary_latent_factor_model.variance) - (
126         0.5 * binary_latent_factor_model.precision
127     ) * (
128         np.sum(np.multiply(x, x))
129         - 2 * np.sum(np.multiply(x, mu_lambda))
130         + np.sum(expectation_s_i_s_j_mu_i_mu_j)
131         - np.trace(
132             expectation_s_i_s_j_mu_i_mu_j
133         ) # remove incorrect E[s_i s_i] = lambda_i * lambda_i
134     + np.sum( # add correct E[s_i s_i] = lambda_i
135         mean_field_approximation.lambda_matrix
136         @ np.multiply(
137             binary_latent_factor_model.mu, binary_latent_factor_model.mu
138         ).T
139     )
140 )
141 expectation_log_p_s_given_theta = np.sum(
142     np.multiply(
143         mean_field_approximation.lambda_matrix,
144         binary_latent_factor_model.log_pi,
145     )
146     + np.multiply(
147         1 - mean_field_approximation.lambda_matrix,
148         binary_latent_factor_model.log_one_minus_pi,
149     )
150 )
151 return expectation_log_p_x_given_s_theta + expectation_log_p_s_given_theta
152
153
154 def _compute_mean_field_approximation_entropy(
155     mean_field_approximation: MeanFieldApproximation,
156 ) -> float:
157     return -np.sum(
158         np.multiply(
159             mean_field_approximation.lambda_matrix,
160             mean_field_approximation.log_lambda_matrix,
161         )
162         + np.multiply(
163             1 - mean_field_approximation.lambda_matrix,
164             mean_field_approximation.log_one_minus_lambda_matrix,
165         )
166     )
167
168
169 def compute_free_energy(
170     x: np.ndarray,
171     binary_latent_factor_model: BinaryLatentFactorModel,
172     mean_field_approximation: MeanFieldApproximation,
173 ) -> float:
174     """
175     free energy associated with current EM parameters and data x
176
177     :param x: data matrix (number_of_points, number_of_dimensions)
178     :param binary_latent_factor_model: a binary_latent_factor_model
179     :param mean_field_approximation: a mean_field_approximation
180     :return: average free energy per data point
181     """
182     expectation_log_p_x_s_given_theta = _compute_expectation_log_p_x_s_given_theta(
183         x, binary_latent_factor_model, mean_field_approximation
184     )
185     mean_field_approximation_entropy = _compute_mean_field_approximation_entropy(
186         mean_field_approximation
187     )
188     return (
189         expectation_log_p_x_s_given_theta + mean_field_approximation_entropy
190     ) / mean_field_approximation.n

```

```

191
192
193 def partial_expectation_step(
194     x: np.ndarray,
195     binary_latent_factor_model: BinaryLatentFactorModel,
196     mean_field_approximation: MeanFieldApproximation,
197     latent_factor: int,
198 ) -> np.ndarray:
199     """ Partial Variational E step for factor i for all data points
200
201     :param x: data matrix (number_of_points, number_of_dimensions)
202     :param binary_latent_factor_model: a binary_latent_factor_model
203     :param mean_field_approximation: a mean_field_approximation
204     :param latent_factor: latent factor to compute partial update
205     :return: lambda_vector: new lambda parameters for the latent factor (number_of_points, 1)
206     """
207     lambda_matrix_excluded = mean_field_approximation.lambda_matrix.exclude(
208         latent_factor
209     )
210     mu_excluded = binary_latent_factor_model.mu_exclude(latent_factor)
211
212     mu_latent = binary_latent_factor_model.mu[:, latent_factor]
213     # (number_of_points, 1)
214     partial_expectation_log_p_x_given_s_theta_proportion = (
215         binary_latent_factor_model.precision
216         * (
217             x # (number_of_points, number_of_dimensions)
218             - 0.5 * mu_latent.T # (1, number_of_dimensions)
219             - lambda_matrix_excluded # (number_of_points, number_of_latent_variables-1)
220             @ mu_excluded.T # (number_of_latent_variables-1, number_of_dimensions)
221         )
222         @ mu_latent # (number_of_dimensions, 1)
223     )
224
225     # (1, 1)
226     partial_expectation_log_p_s_given_theta_proportion = np.log(
227         binary_latent_factor_model.pi[0, latent_factor]
228         / (1 - binary_latent_factor_model.pi[0, latent_factor])
229     )
230
231     # (number_of_points, 1)
232     partial_expectation_log_p_x_s_given_theta_proportion = (
233         partial_expectation_log_p_x_given_s_theta_proportion
234         + partial_expectation_log_p_s_given_theta_proportion
235     )
236
237     # (number_of_points, 1)
238     lambda_vector = 1 / (
239         1 + np.exp(-partial_expectation_log_p_x_s_given_theta_proportion)
240     )
241     lambda_vector[lambda_vector == 0] = 1e-10
242     lambda_vector[lambda_vector == 1] = 1 - 1e-10
243     return lambda_vector
244
245
246 def variational_expectation_step(
247     x: np.ndarray,
248     binary_latent_factor_model: BinaryLatentFactorModel,
249     mean_field_approximation: MeanFieldApproximation,
250     max_steps: int,
251     convergence_criterion: float,
252 ) -> MeanFieldApproximation:
253     """ Variational E step
254
255     :param x: data matrix (number_of_points, number_of_dimensions)
256     :param binary_latent_factor_model: a binary_latent_factor_model
257     :param mean_field_approximation: a mean_field_approximation
258     :param max_steps: maximum number of steps of fixed point equations
259     :param convergence_criterion: early stopping if change in free energy < convergence_criterion
260     :return: mean field approximation
261     """
262     previous_free_energy = compute_free_energy(
263         x, binary_latent_factor_model, mean_field_approximation
264     )
265     for i in range(max_steps):
266         for latent_factor in range(binary_latent_factor_model.k):
267             mean_field_approximation.lambda_matrix[
268                 :, latent_factor
269             ] = partial_expectation_step(
270                 x, binary_latent_factor_model, mean_field_approximation, latent_factor
271             )
272             free_energy = compute_free_energy(
273                 x, binary_latent_factor_model, mean_field_approximation
274             )
275             if free_energy - previous_free_energy <= convergence_criterion:
276                 break
277             previous_free_energy = free_energy
278     return mean_field_approximation
279
280
281 def maximisation_step(
282     x: np.ndarray,
283     mean_field_approximation: MeanFieldApproximation,
284 ) -> BinaryLatentFactorModel:
285     expectation_s = mean_field_approximation.lambda_matrix
286     expectation_ss = (

```

```

287     mean_field_approximation.lambda_matrix.T
288     @ mean_field_approximation.lambda_matrix
289 )
290 np.fill_diagonal(expectation_ss, mean_field_approximation.lambda_matrix.sum(axis=0))
291 mu, sigma, pi = m_step(x, expectation_s, expectation_ss)
292 return BinaryLatentFactorModel(
293     mu=mu,
294     sigma=sigma,
295     pi=pi,
296 )
297
298
299 def learn_binary_factors(
300     x: np.ndarray,
301     k: int,
302     em_maximum_iterations: int,
303     e_maximum_steps: int,
304     e_convergence_criterion: float,
305 ):
306     n = x.shape[0]
307     mean_field_approximation = init_mean_field_approximation(k, n)
308     binary_latent_factor_model = init_binary_latent_factor_model(
309         x, mean_field_approximation
310     )
311
312     for _ in range(em_maximum_iterations):
313         mean_field_approximation = variational_expectation_step(
314             x=x,
315             binary_latent_factor_model=binary_latent_factor_model,
316             mean_field_approximation=mean_field_approximation,
317             max_steps=e_maximum_steps,
318             convergence_criterion=e_convergence_criterion,
319         )
320         binary_latent_factor_model = maximisation_step(
321             x=x,
322             mean_field_approximation=mean_field_approximation,
323         )
324     return mean_field_approximation, binary_latent_factor_model

```

src/models/mean_field_learning.py

The rest of the Python code for question 3:

```
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 η is a collection of all hyperparameters.

We know:

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

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

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

We choose the conjugate priors:

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

$$P(\Psi | \eta) = \prod_{d=1}^D \text{InvGamma}(\Psi_{dd} | a_d, b_d) = \prod_{d=1}^D \frac{b_d^{a_d}}{\Gamma(a_d)} \Psi_{dd}^{-a_d-1} \exp(-\frac{b_d}{\Psi_{dd}})$$

a Gaussian prior on \mathbf{A} and a product of inverse gamma distributions on Ψ where we assume Ψ is a diagonal matrix.

Combining, we have our expression:

$$\begin{aligned} \log P(\mathbf{x}, \mathbf{s}, \pi, \mathbf{A}, \Psi | \eta) = & -\frac{D}{2} \log(2\pi) - \frac{1}{2} \sum_{d=1}^D \log \Psi_{dd} - \frac{1}{2} (\mathbf{x} - \mathbf{A}\mathbf{s})^T \Psi^{-1} (\mathbf{x} - \mathbf{A}\mathbf{s}) \\ & + \sum_{k=1}^K s_k \log \pi_k + (1 - s_k) \log(1 - \pi_k) \\ & + \sum_{i=1}^K (\alpha - 1) \log \pi_k + (\beta - 1) \log(1 - \pi_k) - \log B(\alpha, \beta) \\ & - \frac{DK}{2} \log(2\pi) - \frac{1}{2} \log(|\Sigma_{\mathbf{A}}|) - \frac{1}{2} (\mathbf{A} - \mu_{\mathbf{A}})^T \Sigma_{\mathbf{A}}^{-1} (\mathbf{A} - \mu_{\mathbf{A}}) \\ & + \sum_{d=1}^D a_d \log b_d + (-a_d - 1) \log \Psi_{dd} - \frac{b_d}{\Psi_{dd}} - \log \Gamma(a_d) \end{aligned}$$

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

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

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

Substituting the relevant terms:

$$q_s(\mathbf{s}) \propto \exp \left\langle -\frac{1}{2} (\mathbf{x} - \mathbf{A}\mathbf{s})^T \Psi^{-1} (\mathbf{x} - \mathbf{A}\mathbf{s}) + \sum_{k=1}^K s_k \log \pi_k + (1 - s_k) \log(1 - \pi_k) \right\rangle_{q(\theta)}$$

Simplifying:

$$q_s(\mathbf{s}) \propto \exp \left\langle -\frac{1}{2} \left(\mathbf{s}^T \mathbf{A}^T \Psi^{-1} \mathbf{A} \mathbf{s} - 2 \mathbf{s}^T \left(\mathbf{A}^T \Psi^{-1} \mathbf{x} + 2 \log \frac{\pi}{1 - \pi} \right) \right) \right\rangle_{q(\theta)}$$

By inspection, we can see:

$$q_s(\mathbf{s}) \propto \mathcal{N}(\mathbf{s} | \mu_{\mathbf{s}}^*, \Sigma_{\mathbf{s}}^*)$$

where

$$\Sigma_{\mathbf{s}}^* = \left\langle (\mathbf{A}^T \Psi^{-1} \mathbf{A})^{-1} \right\rangle_{q(\theta)}$$

and

$$\mu_{\mathbf{s}}^* = \left\langle (\mathbf{A}^T \Psi^{-1} \mathbf{A})^{-1} \left(\mathbf{A}^T \Psi^{-1} \mathbf{x} + 2 \log \frac{\pi}{1 - \pi} \right) \right\rangle_{q(\theta)}$$

the E step updates.

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 $q_{\pi}(\pi)$:

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

Substituting the appropriate terms:

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

We see:

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

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

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

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

Substituting the appropriate terms:

$$q_{\Psi}(\Psi) \propto \left(\prod_{d=1}^D \frac{b_d^{a_d}}{\Gamma(a_d)} \Psi_{dd}^{-a_d-1} \exp\left(-\frac{b_d}{\Psi_{dd}}\right) \right) \exp \left\langle -\frac{1}{2} \sum_{d=1}^D \log \Psi_{dd} - \frac{1}{2} (\mathbf{x} - \mathbf{A}\mathbf{s})^T \Psi^{-1} (\mathbf{x} - \mathbf{A}\mathbf{s}) \right\rangle_{q_{\mathbf{s}}(\mathbf{s})q_{-\Psi}(\theta)}$$

We see:

$$q_{\Psi}(\Psi) \propto \prod_{d=1}^D \frac{b_d^{a_d}}{\Gamma(a_d)} \Psi_{dd}^{-(a_d+\frac{1}{2})-1} \exp \left(-\frac{b_d + \frac{1}{2} \langle (x_d - \mathbf{A}_d \mathbf{s})^2 \rangle_{q_{\mathbf{s}}(\mathbf{s})q_{\mathbf{A}_d}(\mathbf{A}_d)}}{\Psi_{dd}} \right)$$

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

Thus,

$$q_{\Psi}(\Psi) = \prod_{d=1}^D \text{InvGamma} \left(\Psi_{dd} \left| a_d + \frac{1}{2}, b_d + \frac{1}{2} \langle (x_d - \mathbf{A}_d \mathbf{s})^2 \rangle_{q_{\mathbf{s}}(\mathbf{s})q_{\mathbf{A}_d}(\mathbf{A}_d)} \right. \right)$$

For $q_{\mathbf{A}}(\mathbf{A})$:

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

$$q_{\mathbf{A}}(\mathbf{A}) \propto \exp \left(-\frac{1}{2} (\mathbf{A} - \mu_{\mathbf{A}})^T \Sigma_{\mathbf{A}}^{-1} (\mathbf{A} - \mu_{\mathbf{A}}) \right) \exp \left\langle -\frac{1}{2} (\mathbf{x} - \mathbf{A}\mathbf{s})^T \Psi^{-1} (\mathbf{x} - \mathbf{A}\mathbf{s}) \right\rangle_{q_{\mathbf{s}}(\mathbf{s})q_{-\mathbf{A}}(\theta)}$$

$$q_{\mathbf{A}}(\mathbf{A}) \propto \exp \left(-\frac{1}{2} (\mathbf{A}^T \Sigma_{\mathbf{A}}^{-1} \mathbf{A} - 2\mathbf{A}^T \Sigma_{\mathbf{A}}^{-1} \mu_{\mathbf{A}}) \right) \exp \left\langle -\frac{1}{2} (\mathbf{s}^T \mathbf{A}^T \Psi^{-1} \mathbf{A} \mathbf{s} - 2\mathbf{s}^T \mathbf{A}^T \Psi^{-1} \mathbf{x}) \right\rangle_{q_{\mathbf{s}}(\mathbf{s})q_{-\mathbf{A}}(\theta)}$$

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 $s_i^2 = s_i$:

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

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:

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

and

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

as required. Note that C can simply be absorbed into any one of these factors.

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

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

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:

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

Thus,

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

To derive the a message passing scheme, we first define:

$$q(\mathbf{s}) = \prod_{i'=1}^K \left(\tilde{f}_{i'}(s_{i'}) \prod_{j \in ne(i)} \mathcal{M}_{j \rightarrow i} \right)$$

where $ne(i)$ are indices of neighbouring nodes of node i and $\mathcal{M}_{j \rightarrow i}$ is the message from node j to i .

Thus, the cavity distributions:

$$q_{-\tilde{f}_i(s_i)}(\mathbf{s}) = \prod_{j \in ne(i)} \mathcal{M}_{j \rightarrow i}$$

and

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

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

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

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:

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

Defining the natural parameters η_{ij} and η_{ji} for the messages:

$$\begin{aligned} \mathcal{M}_{i \rightarrow j} &= \exp(\eta_{ij} s_i) = \tilde{f}_i(s_i) \prod_{k \in ne(i), k \neq j}^K \mathcal{M}_{k \rightarrow i} \\ \mathcal{M}_{j \rightarrow i} &= \exp(\eta_{ji} s_j) = \tilde{f}_j(s_j) \prod_{k \in ne(j), k \neq i}^K \mathcal{M}_{k \rightarrow j} \end{aligned}$$

Note that because $\tilde{g}_{ij}(s_i, s_j)$ is the product of two Bernoulli distributions, updates to this site approximation involves updating to the natural parameters:

$$\phi_{ij}(\theta) = \begin{bmatrix} \theta_i \\ \theta_j \end{bmatrix}$$

the mean of s_i and s_j respectively.

We can write:

$$\log \tilde{g}_{ij}(s_i, s_j) q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s}) \propto \log \left(\frac{\theta_i}{1 - \theta_i} \right) s_i + \log \left(\frac{\theta_j}{1 - \theta_j} \right) s_j + \eta_{ij} s_i + \eta_{ji} s_j$$

Simplifying:

$$\log \tilde{g}_{ij}(s_i, s_j) q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s}) \propto \log \left(\frac{\theta_i}{1 - \theta_i} + \eta_{ij} \right) s_i + \log \left(\frac{\theta_j}{1 - \theta_j} + \eta_{ji} \right) s_j$$

Thus, the first moments:

$$\mathbb{E}_{s_i} [\tilde{g}_{ij}(s_i, s_j) q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s})] = \frac{1}{1 + \exp(-(\theta_i + \eta_{ij}))}$$

and

$$\mathbb{E}_{s_j}[\tilde{g}_{ij}(s_i, s_j)q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s})] = \frac{1}{1 + \exp(-(\theta_j + \eta_{ji}))}$$

Moreover:

$$\log g_{ij}(s_i, s_j)q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s}) \propto W_{ij}s_i s_j + \eta_{ij}s_i + \eta_{ji}s_j$$

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

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

Thus, the first moment:

$$\mathbb{E}_{s_i}[g_{ij}(s_i, s_j)q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s})] = \frac{\exp(W_{ij} + \eta_{ij} + \eta_{ji}) + \exp(\eta_{ij})}{[\exp(W_{ij} + \eta_{ij} + \eta_{ji}) + \exp(\eta_{ij})] + [\exp(\eta_{ji}) + 1]}$$

Simplifying:

$$\mathbb{E}_{s_i}[g_{ij}(s_i, s_j)q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s})] = \frac{(\exp(W_{ij} + \eta_{ji}) + 1) \exp(\eta_{ij})}{[(\exp(W_{ij} + \eta_{ji}) + 1) \exp(\eta_{ij})] + [\exp(\eta_{ji}) + 1]}$$

similarly:

$$\mathbb{E}_{s_j}[g_{ij}(s_i, s_j)q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s})] = \frac{(\exp(W_{ij} + \eta_{ij}) + 1) \exp(\eta_{ji})}{[(\exp(W_{ij} + \eta_{ij}) + 1) \exp(\eta_{ji})] + [\exp(\eta_{ij}) + 1]}$$

By setting:

$$\mathbb{E}_{s_i}[\tilde{g}_{ij}(s_i, s_j)q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s})] = \mathbb{E}_{s_i}[g_{ij}(s_i, s_j)q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s})]$$

and

$$\mathbb{E}_{s_j}[\tilde{g}_{ij}(s_i, s_j)q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s})] = \mathbb{E}_{s_j}[g_{ij}(s_i, s_j)q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s})]$$

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

$$\frac{1}{1 + \exp(-(\theta_i + \eta_{ij}))} = \frac{(\exp(W_{ij} + \eta_{ji}) + 1) \exp(\eta_{ij})}{[(\exp(W_{ij} + \eta_{ji}) + 1) \exp(\eta_{ij})] + [\exp(\eta_{ji}) + 1]}$$

$$\exp(\eta_{ji}) + 1 = \exp(-(\theta_i + \eta_{ij})) (\exp(W_{ij} + \eta_{ji}) + 1) \exp(\eta_{ij})$$

$$\frac{\exp(\eta_{ji}) + 1}{\exp(W_{ij} + \eta_{ji}) + 1} = \exp(-\theta_i)$$

$$\theta_i = \log \left(\frac{1 + \exp(W_{ij} + \eta_{ji})}{1 + \exp(\eta_{ji})} \right)$$

Similarly:

$$\theta_j = \log \left(\frac{1 + \exp(W_{ij} + \eta_{ij})}{1 + \exp(\eta_{ij})} \right)$$

(c)

Our message passing approximations:

$$\exp(\eta_{ij}s_i) = \tilde{f}_i(s_i) \prod_{k \in ne(i), k \neq j}^K \mathcal{M}_{k \rightarrow i}$$

$$\exp(\eta_{ji}s_j) = \tilde{f}_j(s_j) \prod_{k \in ne(j), k \neq i}^K \mathcal{M}_{k \rightarrow j}$$

where each message $\mathcal{M}_{j \rightarrow i}$ has a factored approximation:

$$\mathcal{M}_{k \rightarrow i} = \exp(\eta_{ki}s_k)$$

because each site $\tilde{g}_{jk}(s_j s_k)$ is approximated as a product of two messages $\mathcal{M}_{j \rightarrow k} \mathcal{M}_{k \rightarrow j}$, each a Bernoulli.

Thus, the natural parameters of the messages are updated with:

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

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.

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
3 import jax
4 import jax.numpy as jnp
5 import numpy as np
6 import pandas as pd
7
8 from src.constants import CO2_FILE_PATH, IMAGES_FILE_PATH, OUTPUTS_FOLDER
9 from src.generate_images import generate_images
10 from src.models.bayesian_linear_regression import LinearRegressionParameters
11 from src.models.kernels import CombinedKernel, CombinedKernelParameters
12 from src.models.gaussian_process_regression import GaussianProcessParameters
13 from src.solutions import q2, q3, q4, q5, q6
14 from dataclasses import asdict
15
16 jax.config.update("jax_enable_x64", True)
17
18 if __name__ == "__main__":
19     if not os.path.exists(OUTPUTS_FOLDER):
20         os.makedirs(OUTPUTS_FOLDER)
21
22     # Question 2
23     Q2_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q2")
24     if not os.path.exists(Q2_OUTPUT_FOLDER):
25         os.makedirs(Q2_OUTPUT_FOLDER)
26     with open(CO2_FILE_PATH) as file:
27         lines = [line.rstrip().split() for line in file]
28
29     df_co2 = pd.DataFrame(
30         np.array([line for line in lines if line[0] != "#"]).astype(float)
31     )
32     column_names = lines[max([i for i, line in enumerate(lines) if line[0] == "#"])[1:]]
33     df_co2.columns = column_names
34     t = df_co2.decimal.values[:] - np.min(df_co2.decimal.values[:])
35     y = df_co2.average.values[:].reshape(1, -1)
36
37     sigma = 1
38     mean = np.array([0, 360]).reshape(-1, 1)
39     covariance = np.array(
40         [
41             [10**2, 0],
42             [0, 100**2],
43         ]
44     )
45     kernel = CombinedKernel()
46     kernel_parameters = CombinedKernelParameters(
47         log_theta=jnp.log(1),
48         log_sigma=jnp.log(1),
49         log_phi=jnp.log(1),
50         log_eta=jnp.log(1),
51         log_tau=jnp.log(1),
52         log_zeta=jnp.log(1e-1),
53     )
54
55     prior_linear_regression_parameters = LinearRegressionParameters(
56         mean=mean,
57         covariance=covariance,
58     )
59     posterior_linear_regression_parameters = q2.a(
60         t,
61         y,
62         sigma,
63         prior_linear_regression_parameters,
64         save_path=os.path.join(Q2_OUTPUT_FOLDER, "a"),
65     )
66     q2.b(
67         t_year=df_co2.decimal.values[:],
68         t=t,
69         y=y,
70         linear_regression_parameters=posterior_linear_regression_parameters,
71         error_mean=0,
72         error_variance=1,
73         save_path=os.path.join(Q2_OUTPUT_FOLDER, "b"),
74     )
75
76     q2.c(
77         kernel=kernel,
78         kernel_parameters=kernel_parameters,
79         log_theta_range=jnp.log(jnp.linspace(1e-1, 2, 5)),
80         t=t[:50].reshape(-1, 1),
81         number_of_samples=3,
82         save_path=os.path.join(Q2_OUTPUT_FOLDER, "c"),
83     )
84
85     gaussian_process_parameters = GaussianProcessParameters(
86         kernel=asdict(kernel_parameters),
87         log_sigma=jnp.log(1),
88     )
89     years_to_predict = 15
90     t_new = t[-1] + np.linspace(0, years_to_predict, years_to_predict * 12)
91     t_test = np.concatenate((t, t_new))
92     q2.f(
```

```

93     t_train=t,
94     y_train=y,
95     t_test=t_test,
96     min_year=np.min(df_co2.decimal.values[:]),
97     prior_linear_regression_parameters=prior_linear_regression_parameters,
98     linear_regression_sigma=sigma,
99     kernel=kernel,
100    gaussian_process_parameters=gaussian_process_parameters,
101    learning_rate=1e-2,
102    number_of_iterations=100,
103    save_path=os.path.join(Q2.OUTPUT_FOLDER, "f"),
104 )
105
106 # Question 3
107 Q3.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q3")
108 if not os.path.exists(Q3.OUTPUT_FOLDER):
109     os.makedirs(Q3.OUTPUT_FOLDER)
110
111 # q3.learn_binary_factors(
112 #     x=generate_images(),
113 #     k=8,
114 #     em_maximum_iterations=5,
115 #     e_maximum_steps=100,
116 #     e_convergence_criterion=0,
117 # )

```

main.py