

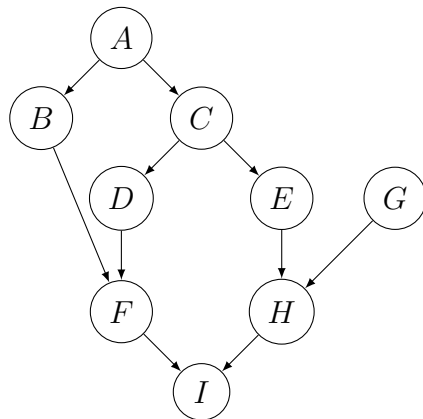
COMP0085 Summative Assignment

Jan 4, 2023

Question 1

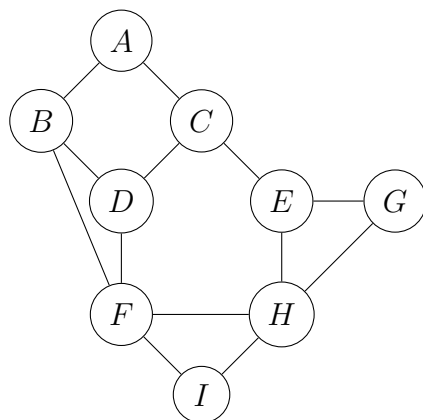
(a)

The directed acyclic graph:

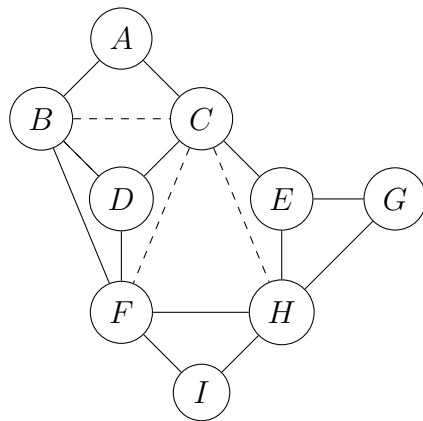


(b)

The moralised graph:

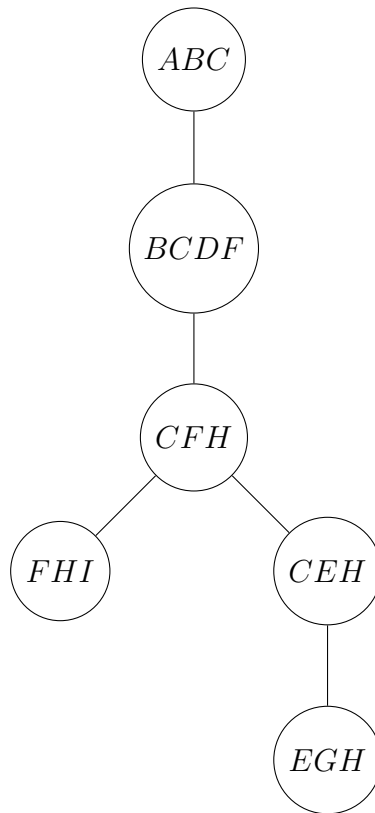


An effective triangulation:



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

The resulting junction tree:



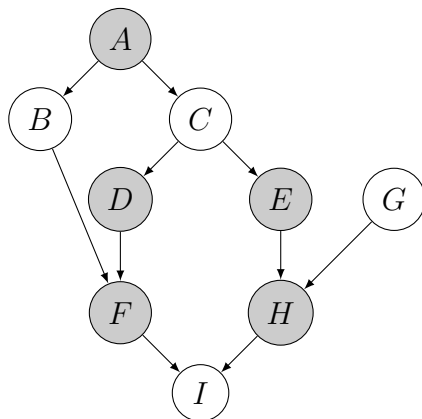
where the circular nodes are cliques.

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: t=The Posterior Covariance

(b)

Plotting the residuals:

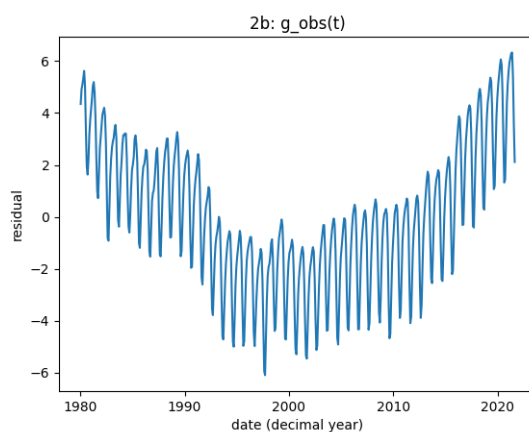


Figure 3: $g_{obs}(t)$

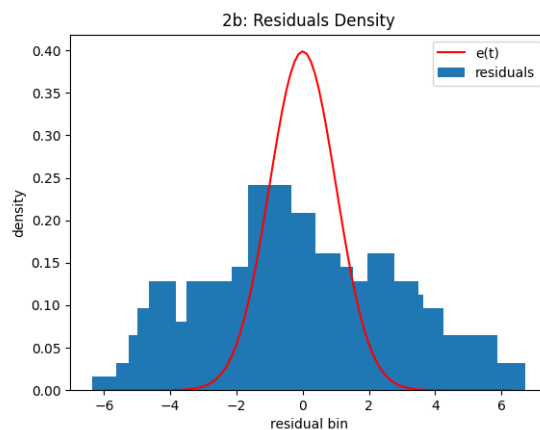


Figure 4: Density Estimation of Residuals vs $e(t) \sim \mathcal{N}(0, 1)$

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.

(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:

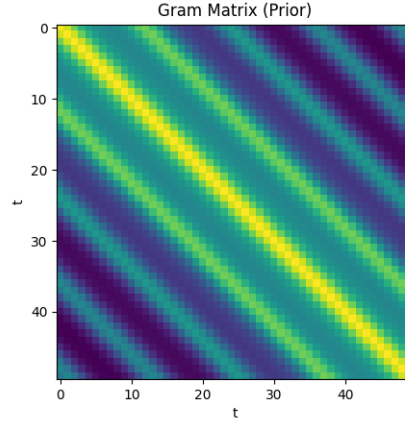


Figure 5: Gram Matrix

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.

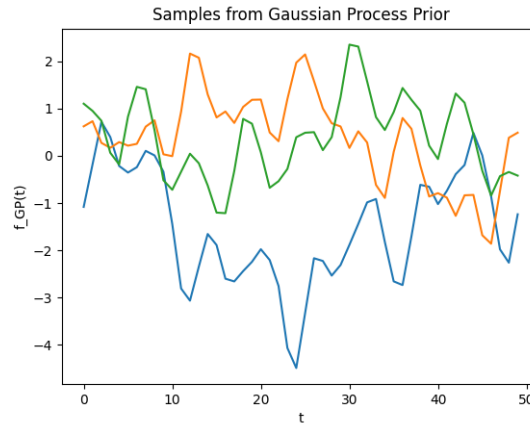


Figure 6: Samples from a zero mean GP with the provided covariance kernel

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

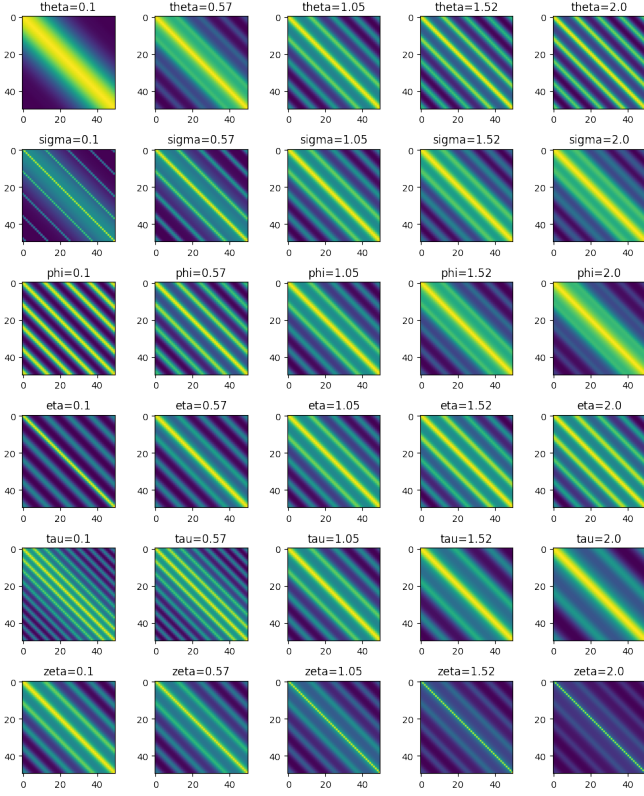


Figure 7: Covariances for different parameters

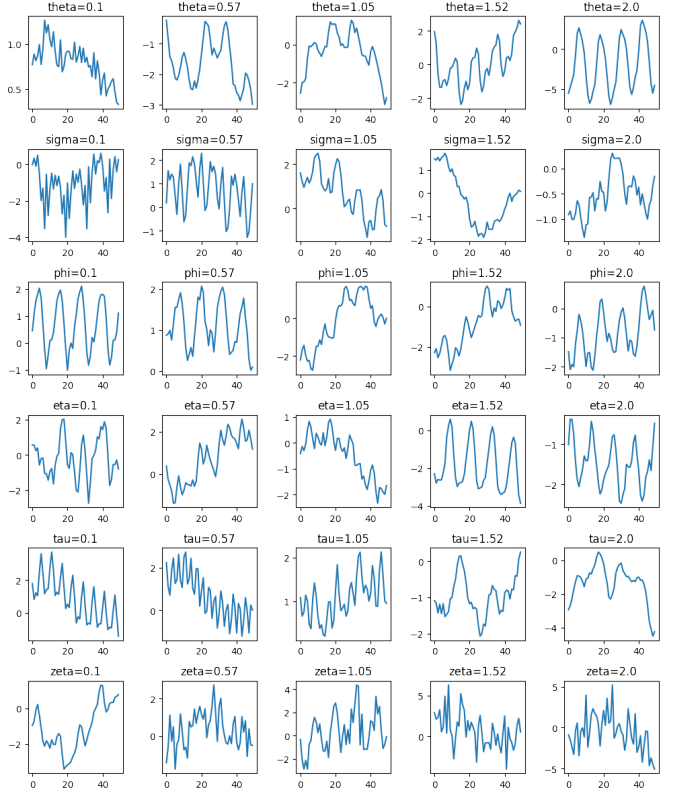


Figure 8: Samples for different parameters

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)

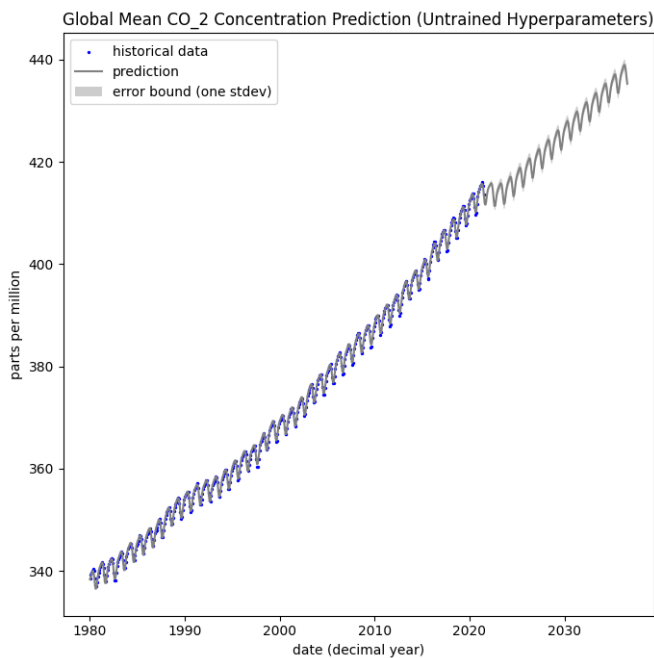


Figure 9: Without hyperparameter tuning

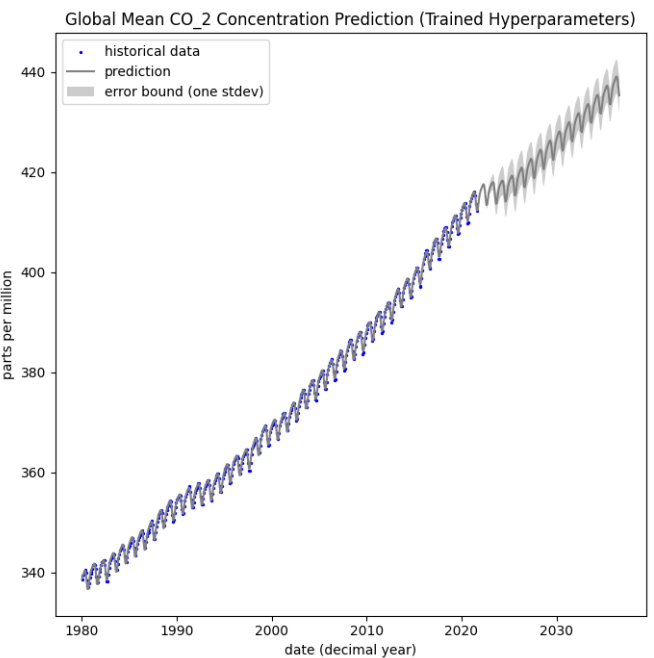


Figure 10: With hyperparameter tuning

(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_)), len(log_theta_range),
128         figsize=(len(log_theta_range) * 2, len(fields(kernel_parameters._.class_)) * 2),
129         frameon=False,
130     )
131     for i, field in enumerate(fields(kernel_parameters._.class_)):
132         default_value = getattr(kernel_parameters, field.name)
133         for j, log_value in enumerate(log_theta_range):
134             setattr(kernel_parameters, field.name, log_value)
135             gram = kernel(t, **asdict(kernel_parameters))
136             ax_samples[i][j].plot(
137                 np.random.multivariate_normal(
138                     jnp.zeros(gram.shape[0]), gram, size=1
139                 ).reshape(-1),
140             )
141             ax_samples[i][j].set_title(f"{field.name.strip('log-')}={np.round(np.exp(log_value), 2)}")
142             setattr(kernel_parameters, field.name, default_value)
143     plt.tight_layout()
144     plt.savefig(save_path + f"-parameter-samples", bbox_inches='tight')
145     plt.close(fig_samples)
146
147     fig_gram, ax_gram = plt.subplots(
148         len(fields(kernel_parameters._.class_)), len(log_theta_range),
149         figsize=(len(log_theta_range) * 2, len(fields(kernel_parameters._.class_)) * 2),
150         frameon=False,
151     )
152     for i, field in enumerate(fields(kernel_parameters._.class_)):
153         default_value = getattr(kernel_parameters, field.name)
154         for j, log_value in enumerate(log_theta_range):
155             setattr(kernel_parameters, field.name, log_value)
156             gram = kernel(t, **asdict(kernel_parameters))
157             ax_gram[i][j].imshow(gram)
158             ax_gram[i][j].set_title(f"{field.name.strip('log-')}={np.round(np.exp(log_value), 2)}")
159             setattr(kernel_parameters, field.name, default_value)
160     plt.tight_layout()
161     plt.savefig(save_path + f"-parameter-grams", bbox_inches='tight')
162     plt.close(fig_gram)
163
164
165 def f(
166     t_train: np.ndarray,
167     y_train: np.ndarray,
168     t_test: np.ndarray,
169     min_year: float,
170     prior_linear_regression_parameters: LinearRegressionParameters,
171     linear_regression_sigma: float,
172     kernel: CombinedKernel,
173     gaussian_process_parameters: GaussianProcessParameters,
174     learning_rate: float,
175     number_of_iterations: int,
176     save_path: str,
177 ):
178     # Train Bayesian Linear Regression
179     x_train = construct_design_matrix(t_train)
180     prior_theta = Theta(
181         linear_regression_parameters=prior_linear_regression_parameters,
182         sigma=linear_regression_sigma,
183     )
184     posterior_linear_regression_parameters = compute_linear_regression_posterior(
185         x_train,
186         y_train,
187         prior_linear_regression_parameters,
188         residuals_precision=prior_theta.precision,
189     )
190

```

```

191 residuals = y_train - posterior_linear_regression_parameters.predict(x_train)
192 gaussian_process = GaussianProcess(kernel, t_train.reshape(-1, 1), residuals.reshape(-1))
193
194 # Prediction
195 x_test = construct_design_matrix(t_test)
196 linear_prediction = posterior_linear_regression_parameters.predict(x_test).reshape(-1)
197 mean_prediction, covariance_prediction = gaussian_process.posterior_distribution(
198     t_test.reshape(-1, 1), **asdict(gaussian_process_parameters)
199 )
200
201 # Plot
202 plt.figure(figsize=(7, 7))
203 plt.scatter(t_train+min_year, y_train.reshape(-1), s=2, color='blue', label="historical data")
204 plt.plot(t_test+min_year, linear_prediction + mean_prediction, color="gray", label="prediction")
205 plt.fill_between(
206     t_test+min_year,
207     linear_prediction+mean_prediction-1*jnp.diagonal(covariance_prediction),
208     linear_prediction+mean_prediction+1*jnp.diagonal(covariance_prediction),
209     facecolor=(0.8, 0.8, 0.8),
210     label="error bound (one stdev)"
211 )
212 plt.xlabel("date (decimal year)")
213 plt.ylabel("parts per million")
214 plt.title("Global Mean CO2 Concentration Prediction (Untrained Hyperparameters)")
215 plt.legend()
216 plt.tight_layout()
217 plt.savefig(save_path+"-extrapolation-untrained", bbox_inches='tight')
218 plt.close()
219
220 # Train Gaussian Process Regression (Hyperparameter Tune)
221 optimizer = optax.adam(learning_rate)
222 gaussian_process_parameters = gaussian_process.train(
223     optimizer, number_of_iterations, **asdict(gaussian_process_parameters)
224 )
225
226 # Prediction
227 x_test = construct_design_matrix(t_test)
228 linear_prediction = posterior_linear_regression_parameters.predict(x_test).reshape(-1)
229 mean_prediction, covariance_prediction = gaussian_process.posterior_distribution(
230     t_test.reshape(-1, 1), **asdict(gaussian_process_parameters)
231 )
232
233 # Plot
234 plt.figure(figsize=(7, 7))
235 plt.scatter(t_train+min_year, y_train.reshape(-1), s=2, color='blue', label="historical data")
236 plt.plot(t_test+min_year, linear_prediction + mean_prediction, color="gray", label="prediction")
237 plt.fill_between(
238     t_test+min_year,
239     linear_prediction+mean_prediction-1*jnp.diagonal(covariance_prediction),
240     linear_prediction+mean_prediction+1*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 (Trained Hyperparameters)")
247 plt.legend()
248 plt.tight_layout()
249 plt.savefig(save_path+"-extrapolation-trained", bbox_inches='tight')
250 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.

The Python code:

```
src/solutions/q3.py
```


Question 4

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:

$$q(\mathbf{s}) = \left(\prod_{i=1}^K \tilde{f}_i(s_i) \right) \left(\prod_{i=1}^K \prod_{j=1}^{i-1} \tilde{g}_{ij}(s_i, s_j) \right)$$

Thus, we can derive cavity distributions:

$$q_{-\tilde{f}_i(s_i)}(\mathbf{s}) = \left(\prod_{i'=1, i' \neq i}^K \tilde{f}_{i'}(s_{i'}) \right) \left(\prod_{i=1}^K \prod_{j=1}^{i-1} \tilde{g}_{ij}(s_i, s_j) \right)$$

and

$$q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s}) = \left(\prod_{i'=1}^K \tilde{f}_{i'}(s_{i'}) \right) \left(\prod_{i'=1}^K \prod_{\substack{j'=1 \\ i', j' \neq i, j}}^{i'-1} \tilde{g}_{i'j'}(s_{i'}, s_{j'}) \right)$$

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

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

We know that the factor $\log f_i(s_i)$ is a Bernoulli of the form $b_i s_i$. Because our approximation 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:

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

Note that because $\tilde{g}_{ij}(s_i, s_j)$ is the product of two Bernoulli distributions, we only require the natural parameters:

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

the mean with respect to s_i and s_j respectively.

We can write:

$$\begin{aligned} \log \tilde{g}_{ij}(s_i, s_j) q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s}) &\propto \log \left(\frac{\theta_i}{1 - \theta_i} \right) s_i + \log \left(\frac{\theta_j}{1 - \theta_j} \right) s_j \\ &\quad + \sum_{i'=1}^K \log \left(\frac{\lambda_{i'}}{1 - \lambda_{i'}} \right) s_{i'} \\ &\quad + \sum_{i'=1}^K \sum_{\substack{j'=1 \\ j' \neq i, j}}^{i'-1} \log \left(\frac{\theta_{i'}}{1 - \theta_{i'}} \right) s_{i'} + \log \left(\frac{\theta_{j'}}{1 - \theta_{j'}} \right) s_{j'} \end{aligned}$$

Only including terms with s_i and s_j :

$$\begin{aligned} \log \tilde{g}_{ij}(s_i, s_j) q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s}) &\propto \left((K - 1) \log \left(\frac{\theta_i}{1 - \theta_i} \right) + \log \left(\frac{\lambda_i}{1 - \lambda_i} \right) \right) s_i \\ &\quad + \left((K - 1) \log \left(\frac{\theta_j}{1 - \theta_j} \right) + \log \left(\frac{\lambda_j}{1 - \lambda_j} \right) \right) s_j \end{aligned}$$

Moreover:

$$\begin{aligned} \log g_{ij}(s_i, s_j) q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s}) &\propto \frac{-\mu_i^T \mu_j}{\sigma^2} s_i s_j \\ &\quad + \sum_{i'=1}^K \log \left(\frac{\lambda_{i'}}{1 - \lambda_{i'}} \right) s_{i'} \\ &\quad + \sum_{i'=1}^K \sum_{\substack{j'=1 \\ j' \neq i, j}}^{i'-1} \log \left(\frac{\theta_{i'}}{1 - \theta_{i'}} \right) s_{i'} + \log \left(\frac{\theta_{j'}}{1 - \theta_{j'}} \right) s_{j'} \end{aligned}$$

Only including terms with s_i and s_j :

$$\begin{aligned} \log g_{ij}(s_i, s_j) q_{-\tilde{g}_{ij}(s_i, s_j)}(\mathbf{s}) &\propto \frac{-\mu_i^T \mu_j}{\sigma^2} s_i s_j \\ &\quad + \left((K - 2) \log \left(\frac{\theta_i}{1 - \theta_i} \right) + \log \left(\frac{\lambda_i}{1 - \lambda_i} \right) \right) s_i \\ &\quad + \left((K - 2) \log \left(\frac{\theta_j}{1 - \theta_j} \right) + \log \left(\frac{\lambda_j}{1 - \lambda_j} \right) \right) s_j \end{aligned}$$

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

```

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

```

main.py