

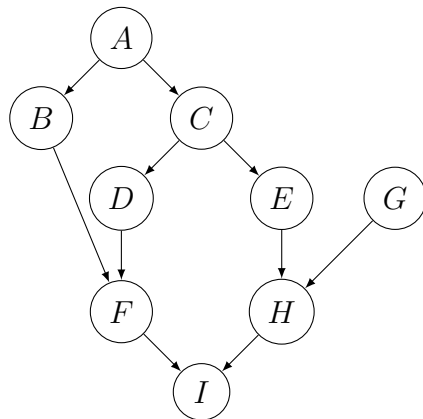
COMP0085 Summative Assignment

Jan 4, 2023

Question 1

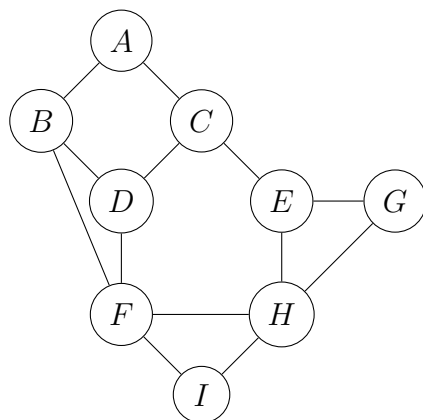
(a)

The directed acyclic graph:

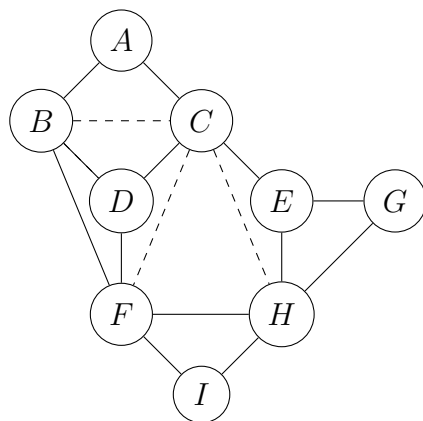


(b)

The moralised graph:

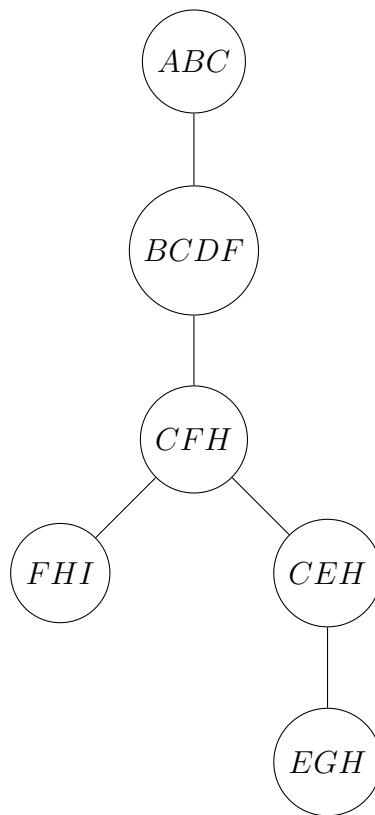


An effective triangulation:



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

The resulting junction tree:



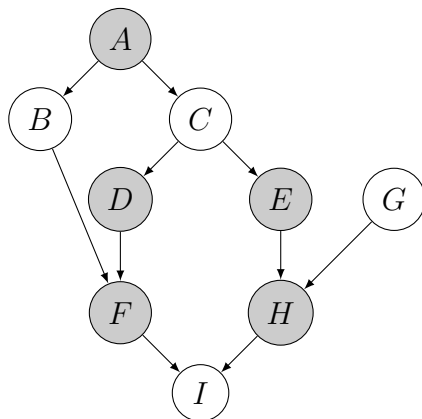
where the circular nodes are cliques.

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:

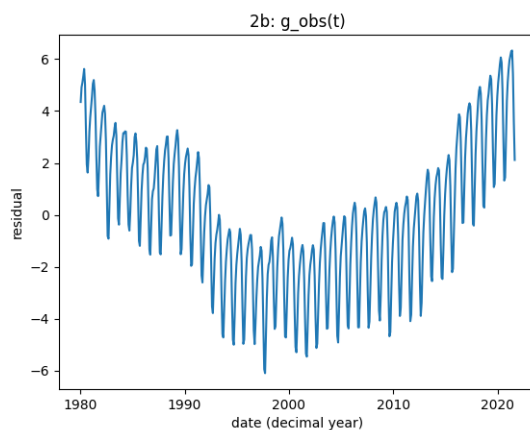


Figure 3: $g_{obs}(t)$

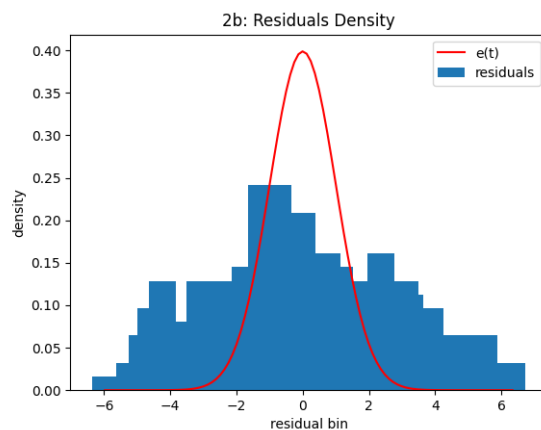


Figure 4: Density Estimation of Residuals vs $e(t) \sim \mathcal{N}(0, 1)$

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:

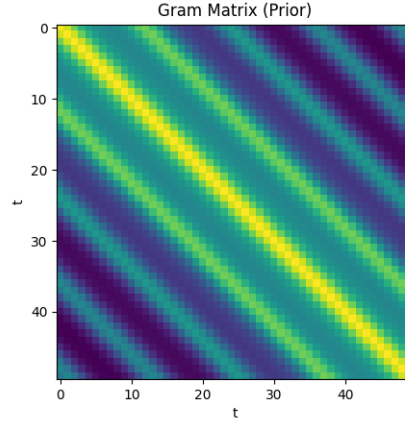


Figure 5: Covariance Matrix

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.

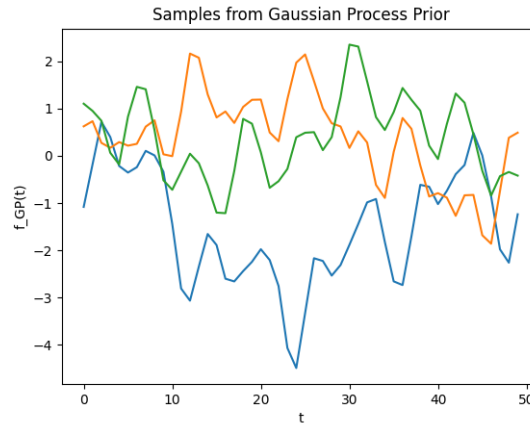


Figure 6: Samples from a zero mean GP with the provided covariance kernel

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

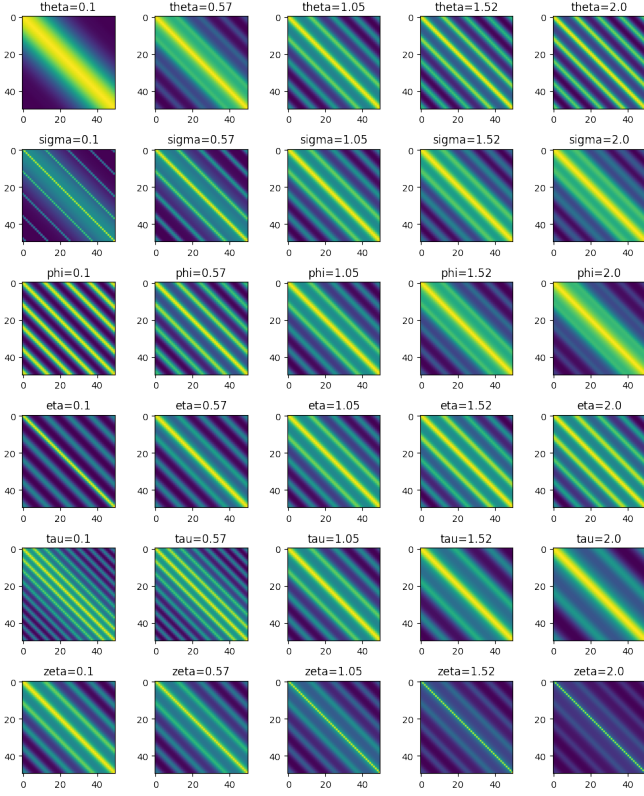


Figure 7: Covariances for different parameters

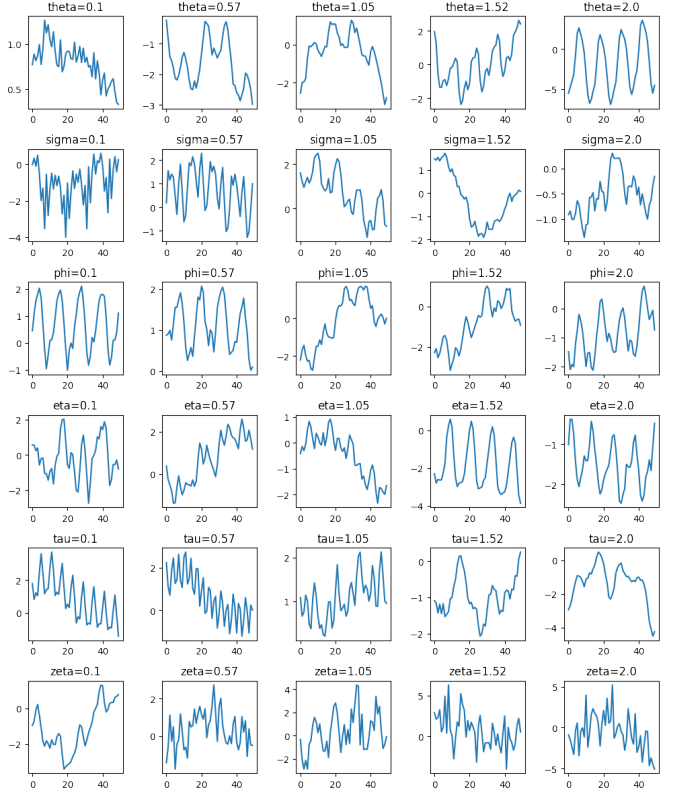


Figure 8: Samples for different parameters

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)

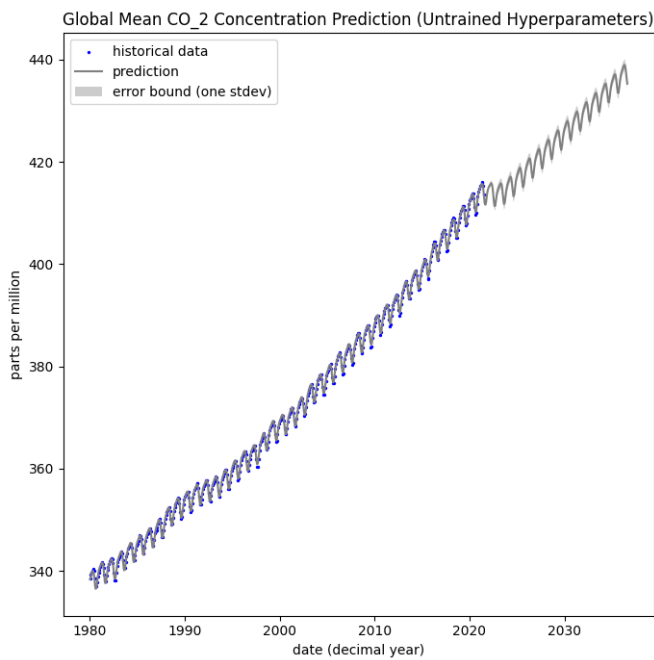


Figure 9: Without hyperparameter tuning

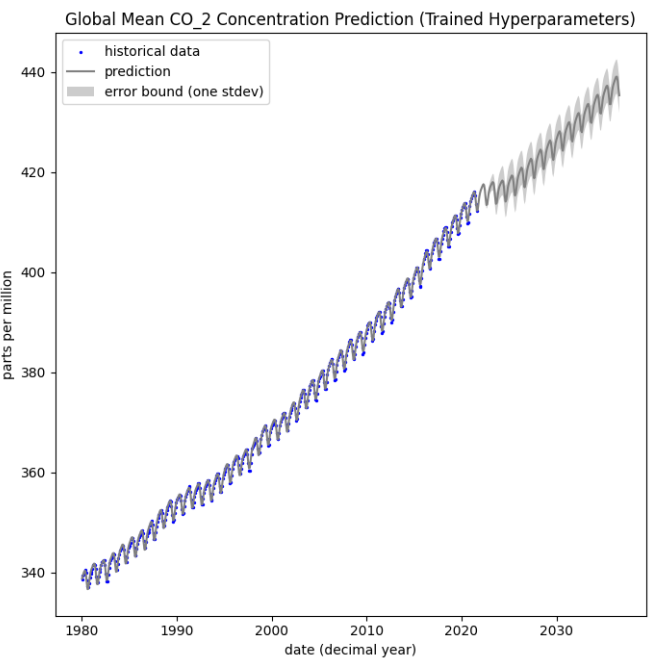


Figure 10: With hyperparameter tuning

(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:

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

and

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

We can write:

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

Substituting the relevant terms:

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

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

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

Thus,

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

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

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

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

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

Solving for λ_i :

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

we have our partial update.

(b)

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

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

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

This mimics the least squares solution:

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

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

(c)

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

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

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

(d)

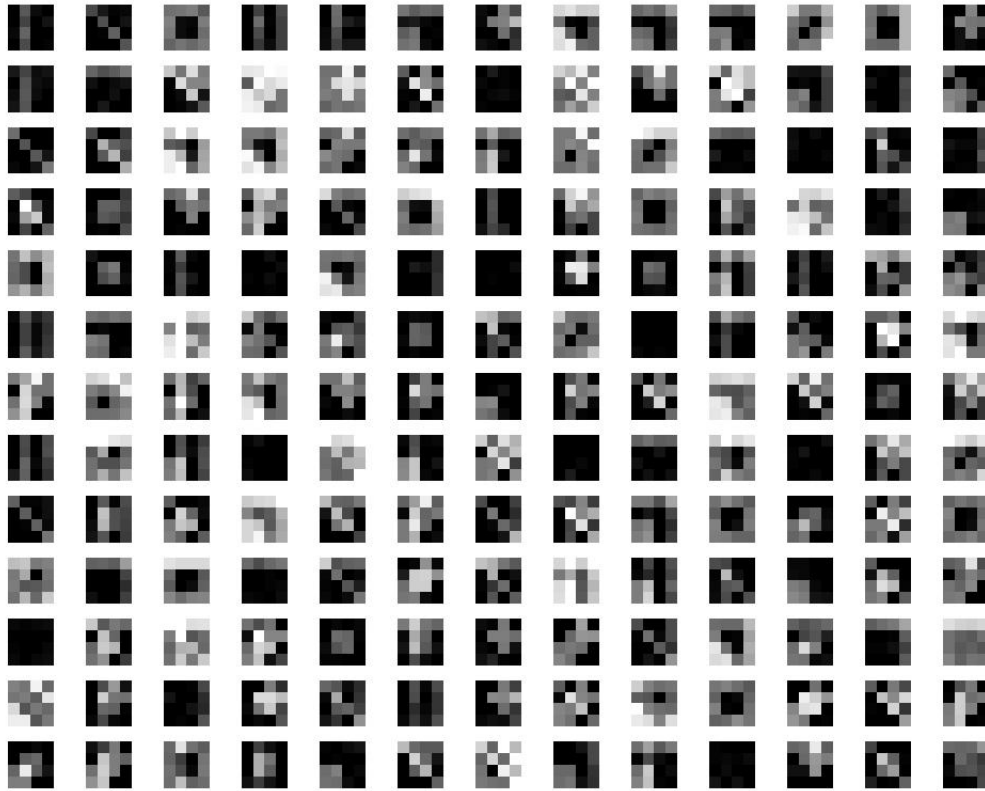


Figure 11: Images generated by randomly combined features with noise

(e)

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

(f)

(g)

The Python code for the binary latent factor model:

```

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

```

```

95
96
97 class BinaryLatentFactorApproximation(ABC):
98     @property
99     @abstractmethod
100     def lambda_matrix(self):
101         pass
102
103     @abstractmethod
104     def variational_expectation_step(
105         self,
106         x: np.ndarray,
107         binary_latent_factor_model: BinaryLatentFactorModel,
108     ) -> List[float]:
109         pass
110
111     @property
112     def log_lambda_matrix(self):
113         return np.log(self.lambda_matrix)
114
115     @property
116     def log_one_minus_lambda_matrix(self):
117         return np.log(1 - self.lambda_matrix)
118
119     @property
120     def n(self):
121         return self.lambda_matrix.shape[0]
122
123     @property
124     def k(self):
125         return self.lambda_matrix.shape[1]
126
127     def compute_free_energy(
128         self,
129         x: np.ndarray,
130         binary_latent_factor_model: BinaryLatentFactorModel,
131     ) -> float:
132         """
133         free energy associated with current EM parameters and data x
134
135         :param x: data matrix (number_of_points, number_of_dimensions)
136         :param binary_latent_factor_model: a binary_latent_factor_model
137         :return: average free energy per data point
138         """
139         expectation_log_p_x_s_given_theta = (
140             self._compute_expectation_log_p_x_s_given_theta(
141                 x, binary_latent_factor_model
142             )
143         )
144         approximation_model_entropy = self._compute_approximation_model_entropy()
145         return (
146             expectation_log_p_x_s_given_theta + approximation_model_entropy
147         ) / self.n
148
149     def _compute_expectation_log_p_x_s_given_theta(
150         self,
151         x: np.ndarray,
152         binary_latent_factor_model: BinaryLatentFactorModel,
153     ) -> float:
154         """
155         The first term of the free energy, the expectation of log P(X,S|theta)
156
157         :param x: data matrix (number_of_points, number_of_dimensions)
158         :param binary_latent_factor_model: a binary_latent_factor_model
159         :return: the expectation of log P(X,S|theta)
160         """
161         # (number_of_points, number_of_dimensions)
162         mu_lambda = self.lambda_matrix @ binary_latent_factor_model.mu.T
163
164         # (number_of_latent_variables, number_of_latent_variables)
165         expectation_s_i_s_j_mu_i_mu_j = np.multiply(
166             self.lambda_matrix.T @ self.lambda_matrix,
167             binary_latent_factor_model.mu.T @ binary_latent_factor_model.mu,
168         )
169
170         expectation_log_p_x_s_given_theta = -(
171             self.n * binary_latent_factor_model.d / 2
172         ) * np.log(2 * np.pi * binary_latent_factor_model.variance) - (
173             0.5 * binary_latent_factor_model.precision
174         ) * (
175             np.sum(np.multiply(x, x))
176             - 2 * np.sum(np.multiply(x, mu_lambda))
177             + np.sum(expectation_s_i_s_j_mu_i_mu_j)
178             - np.trace(
179                 expectation_s_i_s_j_mu_i_mu_j
180             ) # remove incorrect E[s_i s_i] = lambda_i * lambda_i
181             + np.sum( # add correct E[s_i s_i] = lambda_i
182                 self.lambda_matrix
183                 @ np.multiply(
184                     binary_latent_factor_model.mu, binary_latent_factor_model.mu
185                 ).T
186             )
187         )
188         expectation_log_p_s_given_theta = np.sum(
189             np.multiply(
190                 self.lambda_matrix,

```

```

191         binary_latent_factor_model.log_pi,
192     )
193     + np.multiply(
194         1 - self.lambda_matrix,
195         binary_latent_factor_model.log_one_minus_pi,
196     )
197 )
198 return expectation_log_p_x_given_s_theta + expectation_log_p_s_given_theta
199
200 def _compute_approximation_model_entropy(self) -> float:
201     return -np.sum(
202         np.multiply(
203             self.lambda_matrix,
204             self.log_lambda_matrix,
205         )
206         + np.multiply(
207             1 - self.lambda_matrix,
208             self.log_one_minus_lambda_matrix,
209         )
210     )
211
212
213 def is_converge(free_energies, current_lambda_matrix, previous_lambda_matrix):
214     return (abs(free_energies[-1] - free_energies[-2]) == 0) and np.linalg.norm(
215         current_lambda_matrix - previous_lambda_matrix
216     ) == 0
217
218
219 def learn_binary_factors(
220     x: np.ndarray,
221     em_iterations: int,
222     binary_latent_factor_model: BinaryLatentFactorModel,
223     binary_latent_factor_approximation: BinaryLatentFactorApproximation,
224 ):
225     free_energies: List[float] = [
226         binary_latent_factor_approximation.compute_free_energy(
227             x, binary_latent_factor_model
228         )
229     ]
230     for _ in range(em_iterations):
231         previous_lambda_matrix = np.copy(
232             binary_latent_factor_approximation.lambda_matrix
233         )
234         binary_latent_factor_approximation.variational_expectation_step(
235             x=x,
236             binary_latent_factor_model=binary_latent_factor_model,
237         )
238         binary_latent_factor_model.maximisation_step(
239             x,
240             binary_latent_factor_approximation,
241         )
242         free_energies.append(
243             binary_latent_factor_approximation.compute_free_energy(
244                 x, binary_latent_factor_model
245             )
246         )
247         if is_converge(
248             free_energies,
249             binary_latent_factor_approximation.lambda_matrix,
250             previous_lambda_matrix,
251         ):
252             break
253     return binary_latent_factor_approximation, binary_latent_factor_model, free_energies

```

src/models/binary_latent_factor_model.py

The Python code for mean field learning:

```

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

```

```

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

```

src/models/mean_field_learning.py

The rest of the Python code for question 3:

```

1 import numpy as np
2 from src.models.mean_field_learning import (
3     BinaryLatentFactorModel,
4     init_mean_field_approximation,
5 )
6 from src.models.binary_latent_factor_model import (
7     learn_binary_factors,
8     init_binary_latent_factor_model,
9     is_converge,
10 )
11 import matplotlib.pyplot as plt
12 from typing import List
13
14
15 def e_and_f(
16     x: np.ndarray,
17     k: int,
18     em_iterations: int,
19     e_maximum_steps: int,
20     e_convergence_criterion: float,
21     save_path: str,
22 ):
23     n = x.shape[0]
24     mean_field_approximation = init_mean_field_approximation(
25         k, n, max_steps=e_maximum_steps, convergence_criterion=e_convergence_criterion
26     )
27     binary_latent_factor_model = init_binary_latent_factor_model(
28         x, mean_field_approximation
29     )
30     _, binary_latent_factor_model, free_energy = learn_binary_factors(
31         x,
32         em_iterations,
33         binary_latent_factor_model,
34         binary_latent_factor_approximation=mean_field_approximation,
35     )
36     fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
37     for i in range(k):
38         ax[i].imshow(binary_latent_factor_model.mu[:, i].reshape(4, 4))
39         ax[i].set_title(f"Latent Feature mu.{i}")
40     fig.suptitle("Learned Features (Mean Field Learning)")
41     plt.tight_layout()
42     plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
43     plt.close()
44
45     plt.title("Free Energy (Mean Field Learning)")
46     plt.xlabel("t (EM steps)")
47     plt.ylabel("Free Energy")
48     plt.plot(free_energy)
49     plt.savefig(save_path + "-free-energy", bbox_inches="tight")
50     plt.close()
51     return binary_latent_factor_model
52
53
54 def g(
55     x: np.ndarray,
56     binary_latent_factor_model: BinaryLatentFactorModel,
57     sigmas: List[float],
58     k: int,
59     em_iterations: int,
60     e_maximum_steps: int,
61     e_convergence_criterion: float,
62     save_path: str,
63 ):
64     n = x.shape[0]
65     free_energies = []
66     for sigma in sigmas:
67         binary_latent_factor_model.sigma = sigma
68         mean_field_approximation = init_mean_field_approximation(
69             k,
70             n,
71             max_steps=e_maximum_steps,
72             convergence_criterion=e_convergence_criterion,
73         )
74         free_energy: List[float] = [
75             mean_field_approximation.compute_free_energy(x, binary_latent_factor_model)
76         ]
77         for _ in range(em_iterations):
78             previous_lambda_matrix = np.copy(mean_field_approximation.lambda_matrix)
79             new_free_energy = mean_field_approximation.variational_expectation_step(
80                 binary_latent_factor_model=binary_latent_factor_model,
81                 x=x,
82             )
83             free_energy.extend(new_free_energy)
84             if is_converge(
85                 free_energy,
86                 mean_field_approximation.lambda_matrix,
87                 previous_lambda_matrix,
88             ):
89                 break
90         free_energies.append(free_energy)
91
92     for i, free_energy in enumerate(free_energies):
93         plt.plot(
94             np.arange(len(free_energy) - 1),

```



```

95         np.log(np.diff(np.array(free_energy))),
96         label=f"sigma={sigmas[i]}" ,
97     )
98     plt.title(f"log(F(t)-F(t-1))")
99     plt.xlabel("t (Variational E steps)")
100    plt.ylabel("log(F(t)-F(t-1))")
101    plt.tight_layout()
102    plt.legend()
103    plt.savefig(save_path + f"-free-energy-diff-sigma.png", bbox_inches="tight")
104    plt.close()

```

src/solutions/q3.py

Question 4

We begin with the log joint:

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

where 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 $s_i^2 = s_i$:

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

Thus:

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

where the factors are defined:

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

and

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

as required.

The Boltzmann Machine can be defined:

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

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

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

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

and

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

and

$$\log Z = -C$$

(b)

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

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

Thus,

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

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

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

where

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

and

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

Thus,

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

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

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

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

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

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

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

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

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

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

Simplifying:

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

and,

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

Thus, the cavity distributions are:

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

and

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

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

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

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

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

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

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

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

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

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

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

where:

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

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

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

and

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

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

We can write:

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

Simplifying:

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

Thus, the first moments:

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

and

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

Moreover:

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

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

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

Thus, the first moment:

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

Simplifying:

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

Similarly:

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

By setting:

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

and

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

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

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

Simplifying:

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

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

Our parameter update:

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

Similarly:

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

(c)

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.

1 Question 6

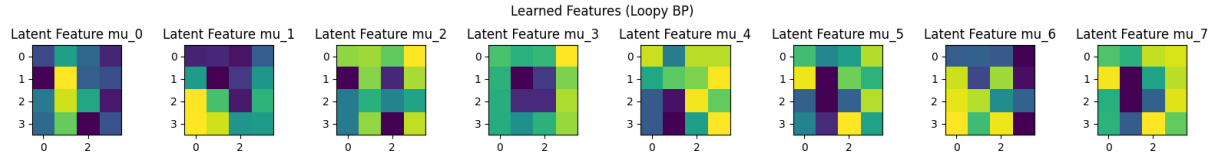


Figure 12: Latent factors learned with EP/Loopy-BP

The Python code for the Boltzmann machine:

```

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

```

src/models/boltzmann_machine.py

The Python code for message passing:

```

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

```

```

94         value=np.log(1 + np.exp(w_i_j + eta_i_not_j))
95         - np.log(1 + np.exp(eta_i_not_j)),
96     )
97     self.set_xi(
98         i=j,
99         j=i,
100         value=np.log(1 + np.exp(w_i_j + eta_j_not_i))
101         - np.log(1 + np.exp(eta_j_not_i)),
102     )
103
104     def _update_singleton_message(
105         self,
106         x,
107         boltzmann_machine: BoltzmannMachine,
108         i: int,
109     ):
110         b_i = boltzmann_machine.b_index(x=x, node_index=i)
111         self.set_xi(i=i, j=i, value=b_i)
112
113
114     def init_message_passing(k, n) -> MessagePassing:
115         eta_matrix = np.random.random(size=(n, k, k))
116         return MessagePassing(eta_matrix)

```

src/models/message_passing.py

The rest of the Python code for question 6:

```
1 from src.generate_images import generate_images
2 import matplotlib.pyplot as plt
3 from src.models.binary_latent_factor_model import learn_binary_factors
4 from src.models.boltzmann_machine import init_boltzmann_machine
5 from src.models.message_passing import init_message_passing
6
7
8 def run(x, k, em_iterations, save_path):
9     n = x.shape[0]
10    message_passing = init_message_passing(k, n)
11    boltzmann_machine = init_boltzmann_machine(x, message_passing)
12    message_passing, boltzmann_machine, free_energy = learn_binary_factors(
13        x=x,
14        em_iterations=em_iterations,
15        binary_latent_factor_model=boltzmann_machine,
16        binary_latent_factor_approximation=message_passing,
17    )
18    fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
19    for i in range(k):
20        ax[i].imshow(boltzmann_machine.mu[:, i].reshape(4, 4))
21        ax[i].set_title(f"Latent Feature mu-{{i}}")
22    fig.suptitle("Learned Features (Loopy BP)")
23    plt.tight_layout()
24    plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
25    plt.close()
26
27    plt.title("Free Energy (Loopy BP)")
28    plt.xlabel("t (EM steps)")
29    plt.ylabel("Free Energy")
30    plt.plot(free_energy)
31    plt.savefig(save_path + "-free-energy", bbox_inches="tight")
32    plt.close()
```

src/solutions/q6.py

Appendix 1: constants.py

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

src/constants.py

Appendix 2: main.py

```
1 import os
2
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, 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 number_of_images = 1000
111 x = generate_images(n=number_of_images)
112 k = 8
113 em_iterations = 1000
114 e_maximum_steps = 200
115 e_convergence_criterion = 0
116
117 binary_latent_factor_model = q3.e_and_f(
118     x=x,
119     k=k,
120     em_iterations=em_iterations,
121     e_maximum_steps=e_maximum_steps,
122     e_convergence_criterion=e_convergence_criterion,
123     save_path=os.path.join(Q3.OUTPUT_FOLDER, "f"),
124 )
125 q3.g(
126     x=x[:,1, :],
127     binary_latent_factor_model=binary_latent_factor_model,
128     sigmas=[1, 2, 3],
129     k=k,
130     em_iterations=em_iterations,
131     e_maximum_steps=e_maximum_steps,
132     e_convergence_criterion=e_convergence_criterion,
133     save_path=os.path.join(Q3.OUTPUT_FOLDER, "g"),
134 )
135
136 # Question 6
137 Q6.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q6")
138 if not os.path.exists(Q6.OUTPUT_FOLDER):
139     os.makedirs(Q6.OUTPUT_FOLDER)
140 em_iterations = 50
141 q6.run(x, k, em_iterations, save_path=os.path.join(Q6.OUTPUT_FOLDER, "all"))

```

main.py