COMP0085 Summative Assignment

Jan 4, 2023

Question 1

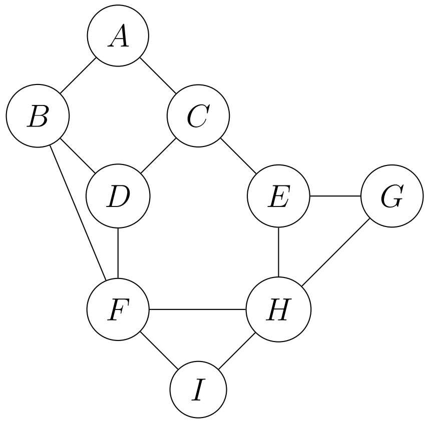
(a)

The directed acyclic graph:

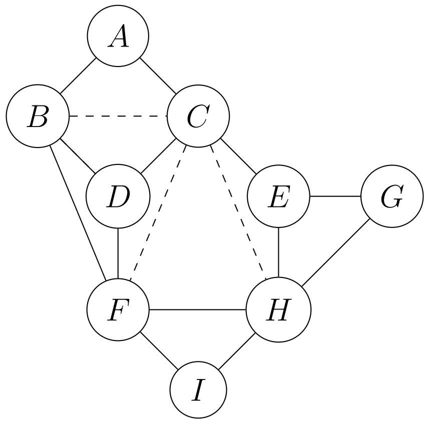
$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & & & \\
B \arrow[ddr] & & C \arrow[dl] \arrow[dr] & & \\
 & D \arrow[d] & & E \arrow[d] & G \arrow[dl] \\
 & F \arrow[dr] & & H \arrow[dl] & \\
 & & I & &
\end{tikzcd}$$

(b)

The moralised graph:

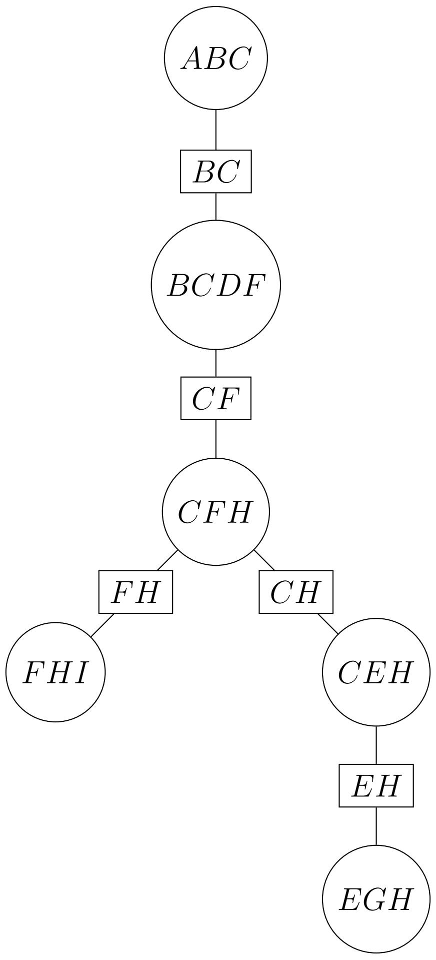


An effective triangulation:

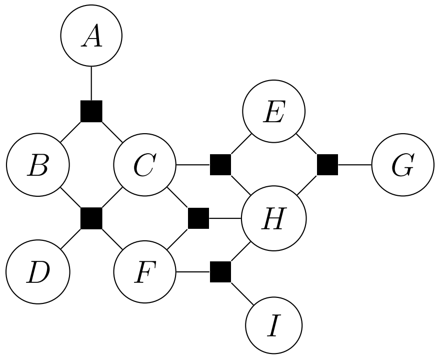


where the dashed lines are edges added to triangulate the moralised graph.

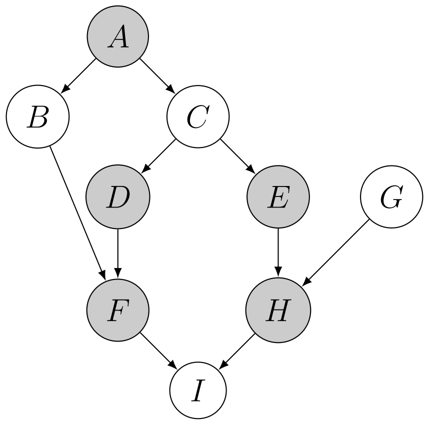
The resulting junction tree:



where the circular nodes are cliques and the square nodes are separators/factors.
The junction tree redrawn as a factor graph:



(c)



The set $\{A, D, E, F, H\}$ is a non-unique smallest set of molecules such that if the concentrations of the species within the set are known, the concentrations of the others $\{B, C, G, I\}$ would all be independent (conditioned on the measured ones).

(d)

Using our factor analysis model, we can describe the biochemical pathway as:

$$\delta[\mathbf{x}] = \Lambda\mathbf{z} + \epsilon$$

where $\delta[\mathbf{x}]$ are the concentration perturbations, $\epsilon \sim \mathcal{N}(0, \Psi)$, and the latent factors $z \sim \mathcal{N}(0, I)$. From the graph structure, we know that:

$$\Lambda = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \Lambda_{BA} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \Lambda_{CA} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Lambda_{DC} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Lambda_{EC} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \Lambda_{FB} & 0 & \Lambda_{FD} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \Lambda_{HE} & 0 & \Lambda_{HG} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \Lambda_{IF} & 0 & \Lambda_{IH} & 0 \end{bmatrix} \text{ and } \mathbf{z} = \begin{bmatrix} z_A \\ z_B \\ z_C \\ z_D \\ z_E \\ z_F \\ z_G \\ z_H \\ z_I \end{bmatrix}$$

Having observations for $\delta[B]$, $\delta[D]$, $\delta[E]$ and $\delta[G]$:

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

We can see that these simplify to the equations:

$$\begin{aligned}\delta[B] &= \Lambda_{BA} z_A + \epsilon_B \\ \delta[D] &= \Lambda_{DC} z_C + \epsilon_D \\ \delta[E] &= \Lambda_{EC} z_C + \epsilon_E \\ \delta[G] &= \epsilon_G\end{aligned}$$

Thus, we see that the only latent variables present are z_A and z_C, so would expect to recover the factors of A and C, the two parent nodes of the observations.

(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 $\mathcal{D} = \{\mathbf{X}, \mathbf{Y}\}$:

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

where $\mathbf{X} = \begin{bmatrix} t_1 & t_2 \cdots t_N \\ 1 & 1 \cdots 1 \end{bmatrix} \in \mathbb{R}^{2\times N}$ 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:

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

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:

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

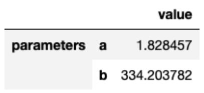
Thus we can identify the posterior covariance:

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

and the posterior mean:

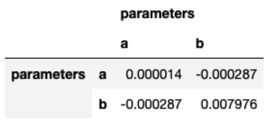
$$\bar{\mu}_w = \bar{\Sigma}_w\left(\frac{\mathbf{X}\mathbf{Y}^T}{\sigma^2} + \Sigma_{\mathbf{w}}^{-1}\mu_{\mathbf{w}}\right)$$

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



		value
parameters	a	1.828457
	b	334.203782

Figure 1: The Posterior Mean



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

Figure 2: The Posterior Covariance

(b)

Plotting the residuals:

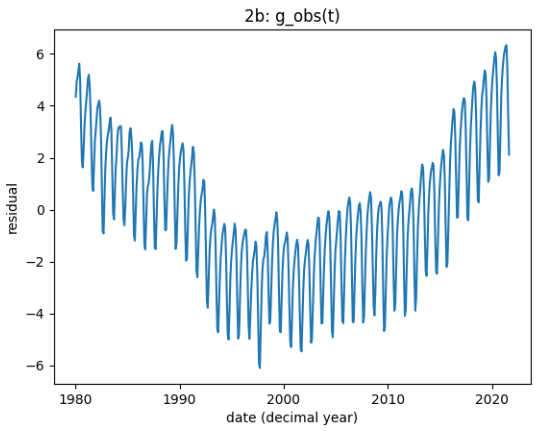


Figure 3: $g_{obs}(t)$

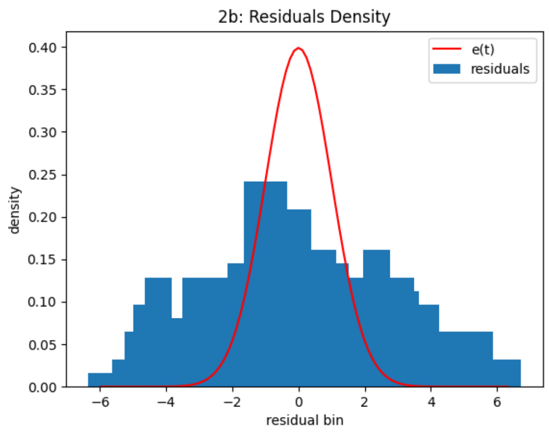


Figure 4: Density Estimation of Residuals vs $e(t) \sim \mathcal{N}(0,1)$

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

(c & d)

We are considering the kernel:

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

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

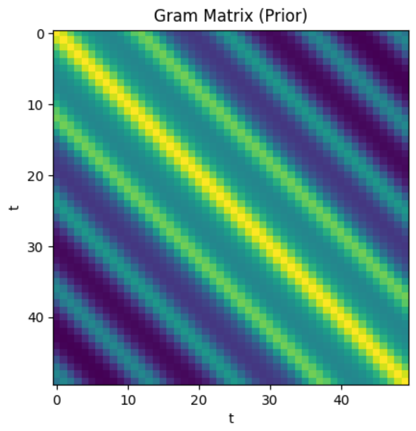


Figure 5: Covariance Matrix

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

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

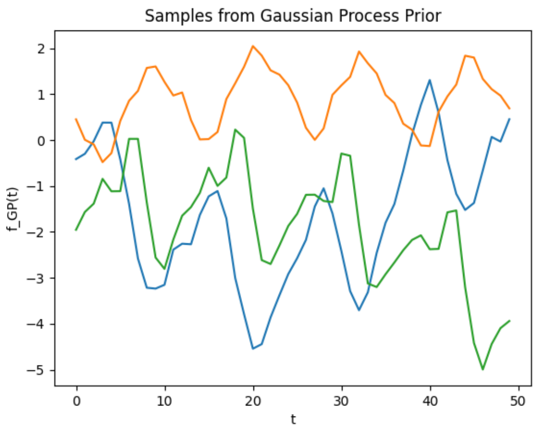


Figure 6: Samples from a zero mean GP with the provided covariance kernel

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

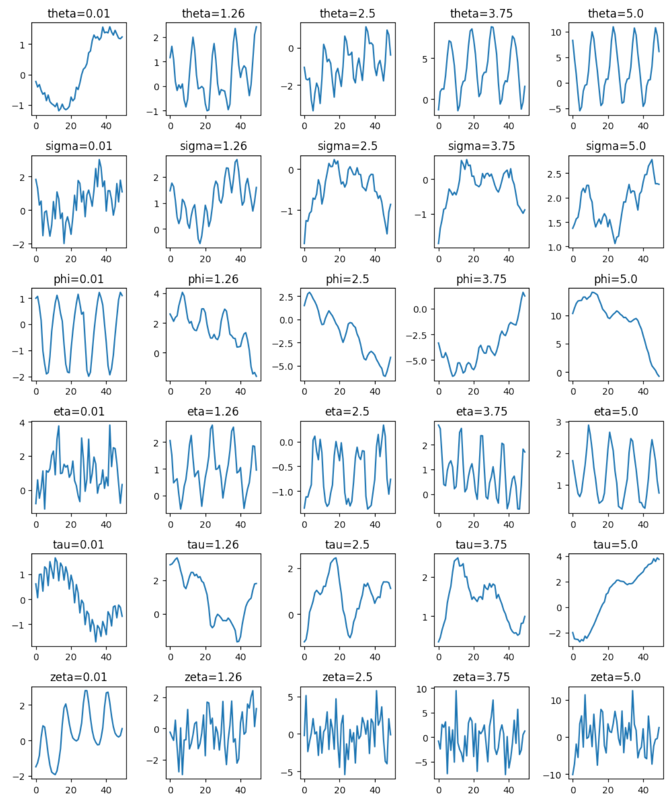


Figure 7: Samples for different parameters

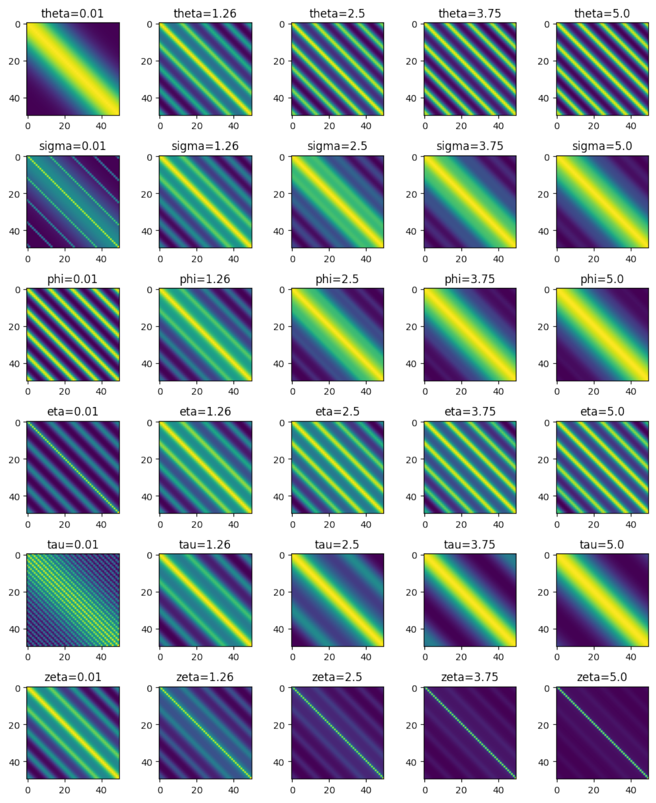


Figure 8: Covariances for different parameters

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

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

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

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

- τ: As τ increases, the period of the periodic function increases. We can see this reflected in the stripes in the gram matrix getting further apart. This makes sense because we are adjusting the period in the sinusoid function of the periodic term with τ.

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

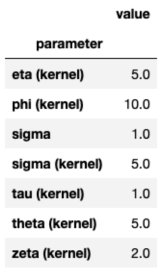
(e)

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

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

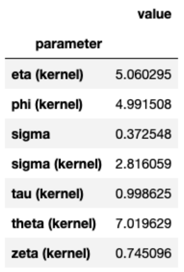
the log-likelihood of the posterior distribution with respect to the given data where $\mathbf{K}$ is the gram matrix for the kernel (equation 2.30 from http://gaussianprocess.org/gpml/chapters/RW2.pdf). We can define a loss function as the negative log-likelihood and employ gradient-based algorithms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

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



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

Figure 9: Untrained hyperparameters



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

Figure 10: Trained Hyperparmaeters

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

(f)

Extrapolating the CO_2 concentration levels:

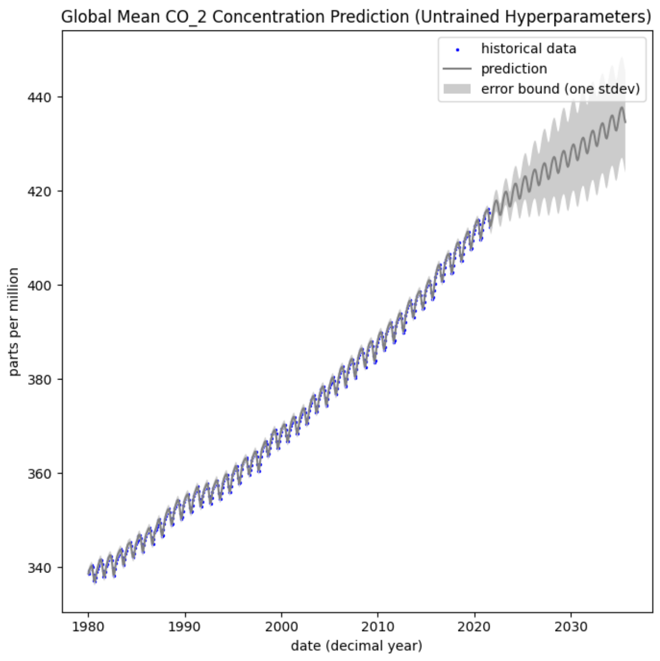


Figure 11: Untrained extrapolation

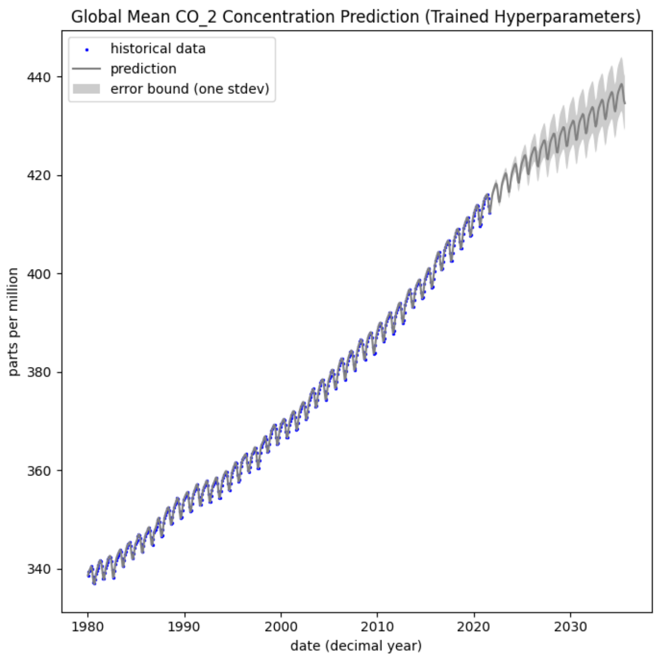


Figure 12: Trained extrapolation

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

(g)

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

The Python code for Bayesian Linear Regression:

```python
from dataclasses import dataclass

import numpy as np


@dataclass
class LinearRegressionParameters:
    mean: np.ndarray
    covariance: np.ndarray

    @property
    def precision(self) -> np.ndarray:
        return np.linalg.inv(self.covariance)

    def predict(self, x: np.ndarray) -> np.ndarray:
        return self.mean.T @ x


@dataclass
class Theta:
    linear_regression_parameters: LinearRegressionParameters
    sigma: float

    @property
    def variance(self) -> float:
        return self.sigma**2

    @property
    def precision(self) -> float:
        return 1 / self.variance


def compute_linear_regression_posterior(
    x: np.ndarray,
    y: np.ndarray,
    prior_linear_regression_parameters: LinearRegressionParameters,
    residuals_precision: float,
) -> LinearRegressionParameters:
    """
    Compute the parameters of the posterior distribution on the linear regression weights

    :param x: design matrix (number of features, number of data points)
    :param y: response matrix (1, number of data points)
    :param prior_linear_regression_parameters: parameters for the prior distribution on the linear regression
     weights
    :param residuals_precision: the precision of the residuals of the linear regression
    :return: parameters for the posterior distribution on the linear regression weights
    """
    posterior_covariance = np.linalg.inv(
        residuals_precision * x @ x.T + prior_linear_regression_parameters.precision
    )
    posterior_mean = posterior_covariance @ (
        residuals_precision * x @ y.T
        + prior_linear_regression_parameters.precision
        @ prior_linear_regression_parameters.mean
    )
    return LinearRegressionParameters(
        mean=posterior_mean, covariance=posterior_covariance
    )
```

src/models/bayesian_linear_regression.py

The Python code for kernels:

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass

import jax.numpy as jnp
from jax import vmap


@dataclass
class KernelParameters(ABC):
    """
    An abstract dataclass containing the parameters for a kernel.
    """


class Kernel(ABC):
    """
    An abstract kernel.
    """

    Parameters: KernelParameters = None

    @abstractmethod
    def _kernel(
        self, parameters: KernelParameters, x: jnp.ndarray, y: jnp.ndarray
    ) -> jnp.ndarray:
        """Kernel evaluation between a single feature x and a single feature y.

        Args:
            parameters: parameters dataclass for the kernel
            x: ndarray of shape (number_of_dimensions,)
            y: ndarray of shape (number_of_dimensions,)

        Returns:
            The kernel evaluation. (1, 1)
        """
        raise NotImplementedError

    def kernel(
        self, parameters: KernelParameters, x: jnp.ndarray, y: jnp.ndarray = None
    ) -> jnp.ndarray:
        """Kernel evaluation for an arbitrary number of x features and y features. Compute k(x, x) if y is None.
        This method requires the parameters dataclass and is better suited for parameter optimisation.

        Args:
            parameters: parameters dataclass for the kernel
            x: ndarray of shape (number_of_x_features, number_of_dimensions)
            y: ndarray of shape (number_of_y_features, number_of_dimensions)

        Returns:
            A gram matrix k(x, y), if y is None then k(x,x). (number_of_x_features, number_of_y_features)
        """
        # compute k(x, x) if y is None
        if y is None:
            y = x

        # add dimension when x is 1D, assume the vector is a single feature
        x = jnp.atleast_2d(x)
        y = jnp.atleast_2d(y)

        assert (
            x.shape[1] == y.shape[1]
        ), f"Dimension Mismatch: {x.shape[1]=} != {y.shape[1]=}"

        return vmap(
            lambda x_i: vmap(
                lambda y_i: self._kernel(parameters, x_i, y_i),
            )(y),
        )(x)

    def __call__(
        self, x: jnp.ndarray, y: jnp.ndarray = None, **parameter_args
    ) -> jnp.ndarray:
        """Kernel evaluation for an arbitrary number of x features and y features.
        This method is more user-friendly without the need for a parameter data class.
        It wraps the kernel computation with the initial step of constructing the parameter data class from the
        provided parameter arguments.

        Args:
            x: ndarray of shape (number_of_x_features, number_of_dimensions)
            y: ndarray of shape (number_of_y_features, number_of_dimensions)
            **parameter_args: parameter arguments for the kernel

        Returns:
            A gram matrix k(x, y), if y is None then k(x,x). (number_of_x_features, number_of_y_features).
        """
        parameters = self.Parameters(**parameter_args)
        return self.kernel(parameters, x, y)

    def diagonal(
        self,
        x: jnp.ndarray,
        y: jnp.ndarray = None,
        **parameter_args,
    ) -> jnp.ndarray:
        """Kernel evaluation of only the diagonal terms of the gram matrix.

        Args:
            x: ndarray of shape (number_of_x_features, number_of_dimensions)
            y: ndarray of shape (number_of_y_features, number_of_dimensions)
            **parameter_args: parameter arguments for the kernel

        Returns:
            A diagonal of gram matrix k(x, y), if y is None then trace(k(x,x)).
            (number_of_x_features, number_of_y_features)
        """
        # compute k(x, x) if y is None
        if y is None:
            y = x

        # add dimension when x is 1D, assume the vector is a single feature
        x = jnp.atleast_2d(x)
        y = jnp.atleast_2d(y)

        assert (
            x.shape[1] == y.shape[1]
        ), f"Dimension Mismatch: {x.shape[1]=} != {y.shape[1]=}"
        assert (
            x.shape[0] == y.shape[0]
        ), f"Must have same number of features for diagonal: {x.shape[0]=} != {y.shape[0]=}"

        return vmap(
            lambda x_i, y_i: self._kernel(
                parameters=self.Parameters(**parameter_args),
                x=x_i,
                y=y_i,
            ),
        )(x, y)

    def trace(
        self, x: jnp.ndarray, y: jnp.ndarray = None, **parameter_args
    ) -> jnp.ndarray:
        """Trace of the gram matrix, calculated by summation of the diagonal matrix.

        Args:
            x: ndarray of shape (number_of_x_features, number_of_dimensions)
            y: ndarray of shape (number_of_y_features, number_of_dimensions)
            **parameter_args: parameter arguments for the kernel

        Returns:
            The trace of the gram matrix k(x, y).
        """
        parameters = self.Parameters(**parameter_args)
        return jnp.trace(self.kernel(parameters, x, y))


@dataclass
class CombinedKernelParameters(KernelParameters):
    """
    Parameters for the Combined Kernel:
    """

    log_theta: float
    log_sigma: float
    log_phi: float
    log_eta: float
    log_tau: float
    log_zeta: float

    @property
    def theta(self) -> float:
        return jnp.exp(self.log_theta)

    @property
    def sigma(self) -> float:
        return jnp.exp(self.log_sigma)

    @property
    def phi(self) -> float:
        return jnp.exp(self.log_phi)

    @property
    def eta(self) -> float:
        return jnp.exp(self.log_eta)

    @property
    def tau(self) -> float:
        return jnp.exp(self.log_tau)

    @property
    def zeta(self) -> float:
        return jnp.exp(self.log_zeta)

    @theta.setter
    def theta(self, value: float) -> None:
        self.log_theta = jnp.log(value)

    @sigma.setter
    def sigma(self, value: float) -> None:
        self.log_sigma = jnp.log(value)

    @phi.setter
    def phi(self, value: float) -> None:
        self.log_phi = jnp.log(value)

    @eta.setter
    def eta(self, value: float) -> None:
        self.log_eta = jnp.log(value)

    @tau.setter
    def tau(self, value: float) -> None:
        self.log_tau = jnp.log(value)

    @zeta.setter
    def zeta(self, value: float) -> None:
        self.log_zeta = jnp.log(value)


class CombinedKernel(Kernel):
    """
    The  kernel defined as:
        k(x, y) = theta^2 * (exp(-(2sin^2(pi(x-y)/tau))/(sigma^2)) + phi^2 * exp(-(x-y)^2/(2 * eta^2)))
                    + zeta^2 * delta(x=y)
    """

    Parameters = CombinedKernelParameters

    def _kernel(
        self,
        parameters: CombinedKernelParameters,
        x: jnp.ndarray,
        y: jnp.ndarray,
    ) -> jnp.ndarray:
        """Kernel evaluation between a single feature x and a single feature y.

        Args:
            parameters: parameters dataclass for the Gaussian kernel
            x: ndarray of shape (1,)
            y: ndarray of shape (1,)

        Returns:
            The kernel evaluation.
        """
        return jnp.dot(
            jnp.ones(1),
            (
                (parameters.theta**2)
                * (
                    (
                        jnp.exp(
                            (-2 * jnp.sin(jnp.pi * (x - y) / parameters.tau) ** 2)
                            / (parameters.sigma**2)
                        )
                    )
                )
                + (parameters.phi**2)
                * (jnp.exp(-((x - y) ** 2) / (2 * parameters.eta**2)))
                + parameters.zeta**2 * (x == y)
            ),
        )
```
src/models/kernels.py

The Python code for Gaussian Process Regression:

```python
from dataclasses import dataclass
from typing import Any, Dict, Tuple

import jax
import jax.numpy as jnp
import optax
from jax import grad
from optax import GradientTransformation

from src.models.kernels import Kernel


@dataclass
class GaussianProcessParameters:
    """
    Parameters for a Gaussian Process:
        log_sigma: logarithm of the noise parameter
        kernel: parameters for the chosen kernel
    """

    log_sigma: float
    kernel: Dict[str, Any]

    @property
    def variance(self) -> float:
        return self.sigma**2

    @property
    def sigma(self) -> float:
        return jnp.exp(self.log_sigma)

    @sigma.setter
    def sigma(self, value: float) -> None:
        self.log_sigma = jnp.log(value)


class GaussianProcess:
    """
    A Gaussian measure defined with a kernel, better known as a Gaussian Process.
    """

    Parameters = GaussianProcessParameters

    def __init__(self, kernel: Kernel, x: jnp.ndarray, y: jnp.ndarray) -> None:
        """Initialising requires a kernel and data to condition the distribution.

        Args:
            kernel: kernel for the Gaussian Process
            x: design matrix (number_of_features, number_of_dimensions)
            y: response vector (number_of_features, )
        """
        self.number_of_train_points = x.shape[0]
        self.x = x
        self.y = y
        self.kernel = kernel

    def _compute_kxx_shifted_cholesky_decomposition(
        self, parameters
    ) -> Tuple[jnp.ndarray, bool]:
        """
        Cholesky decomposition of (kxx + (1/  ^2)*I)

        Args:
            parameters: parameters dataclass for the Gaussian Process

        Returns:
            cholesky_decomposition_kxx_shifted: the cholesky decomposition (number_of_features, number_of_features)
            lower_flag: flag indicating whether the factor is in the lower or upper triangle
        """
        kxx = self.kernel(self.x, **parameters.kernel)
        kxx_shifted = kxx + parameters.variance * jnp.eye(self.number_of_train_points)
        kxx_shifted_cholesky_decomposition, lower_flag = jax.scipy.linalg.cho_factor(
            a=kxx_shifted, lower=True
        )
        return kxx_shifted_cholesky_decomposition, lower_flag

    def posterior_distribution(
        self, x: jnp.ndarray, **parameter_args
    ) -> Tuple[jnp.ndarray, jnp.ndarray]:
        """Compute the posterior distribution for test points x.
        Reference: http://gaussianprocess.org/gpml/chapters/RW2.pdf

        Args:
            x: test points (number_of_features, number_of_dimensions)
            **parameter_args: parameter arguments for the Gaussian Process

        Returns:
            mean: the distribution mean (number_of_features, )
            covariance: the distribution covariance (number_of_features, number_of_features)
        """
        parameters = self.Parameters(**parameter_args)
        kxy = self.kernel(self.x, x, **parameters.kernel)
        kyy = self.kernel(x, **parameters.kernel)
        (
            kxx_shifted_cholesky_decomposition,
            lower_flag,
        ) = self._compute_kxx_shifted_cholesky_decomposition(parameters)

        mean = (
            kxy.T
            @ jax.scipy.linalg.cho_solve(
                c_and_lower=(kxx_shifted_cholesky_decomposition, lower_flag), b=self.y
            )
        ).reshape(
            -1,
        )
        covariance = kyy - kxy.T @ jax.scipy.linalg.cho_solve(
            (kxx_shifted_cholesky_decomposition, lower_flag), kxy
        )
        return mean, covariance

    def posterior_negative_log_likelihood(self, **parameter_args) -> jnp.float64:
        """The negative log likelihood of the posterior distribution for the training data (x, y).
        Reference: http://gaussianprocess.org/gpml/chapters/RW2.pdf

        Args:
            **parameter_args: parameter arguments for the Gaussian Process

        Returns:
            The negative log likelihood.
        """
        parameters = self.Parameters(**parameter_args)
        (
            kxx_shifted_cholesky_decomposition,
            lower_flag,
        ) = self._compute_kxx_shifted_cholesky_decomposition(parameters)

        negative_log_likelihood = -(
            -0.5
            * (
                self.y.T
                @ jax.scipy.linalg.cho_solve(
                    c_and_lower=(kxx_shifted_cholesky_decomposition, lower_flag),
                    b=self.y,
                )
            )
            - jnp.trace(jnp.log(kxx_shifted_cholesky_decomposition))
            - (self.number_of_train_points / 2) * jnp.log(2 * jnp.pi)
        )
        return negative_log_likelihood

    def _compute_gradient(self, **parameter_args) -> Dict[str, Any]:
        """Calculate the gradient of the posterior negative log likelihood with respect to the parameters.

        Args:
            **parameter_args: parameter arguments for the Gaussian Process

        Returns:
            A dictionary of the gradients for each parameter argument.
        """
        gradients = grad(
            lambda params: self.posterior_negative_log_likelihood(**params)
        )(parameter_args)
        return gradients

    def train(
        self,
        optimizer: GradientTransformation,
        number_of_training_iterations: int,
        **parameter_args,
    ) -> GaussianProcessParameters:
        """Train the parameters for a Gaussian Process by optimising the negative log likelihood.

        Args:
            optimizer: jax optimizer object
            number_of_training_iterations: number of iterations to perform the optimizer
            **parameter_args: parameter arguments for the Gaussian Process

        Returns:
            A parameters dataclass containing the optimised parameters.
        """
        opt_state = optimizer.init(parameter_args)
        for _ in range(number_of_training_iterations):
            gradients = self._compute_gradient(**parameter_args)
            updates, opt_state = optimizer.update(gradients, opt_state)
            parameter_args = optax.apply_updates(parameter_args, updates)
        return self.Parameters(**parameter_args)
```

src/models/gaussian_process_regression.py

The rest of the Python code for question 2:

```python
from dataclasses import asdict, fields

import dataframe_image as dfi
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import numpy as np
import optax
import pandas as pd
import scipy

from src.models.bayesian_linear_regression import (
    LinearRegressionParameters,
    Theta,
    compute_linear_regression_posterior,
)
from src.models.gaussian_process_regression import (
    GaussianProcess,
    GaussianProcessParameters,
)
from src.models.kernels import CombinedKernel, CombinedKernelParameters

jax.config.update("jax_enable_x64", True)


def construct_design_matrix(t: np.ndarray):
    return np.stack((t, np.ones(t.shape)), axis=1).T


def a(
    t: np.ndarray,
    y: np.ndarray,
    sigma: float,
    prior_linear_regression_parameters: LinearRegressionParameters,
    save_path: str,
) -> LinearRegressionParameters:
    x = construct_design_matrix(t)
    prior_theta = Theta(
        linear_regression_parameters=prior_linear_regression_parameters,
        sigma=sigma,
    )
    posterior_linear_regression_parameters = compute_linear_regression_posterior(
        x,
        y,
        prior_linear_regression_parameters,
        residuals_precision=prior_theta.precision,
    )
    df_mean = pd.DataFrame(
        posterior_linear_regression_parameters.mean, columns=["value"]
    )
    df_mean.index = ["a", "b"]
    df_mean = pd.concat([df_mean], keys=["parameters"])
    dfi.export(df_mean, save_path + "-mean.png")

    df_covariance = pd.DataFrame(
        posterior_linear_regression_parameters.covariance, columns=["a", "b"]
    )
    df_covariance.index = ["a", "b"]
    df_covariance = pd.concat([df_covariance], keys=["parameters"])
    df_covariance = pd.concat([df_covariance.T], keys=["parameters"])
    dfi.export(df_covariance, save_path + "-covariance.png")
    return posterior_linear_regression_parameters


def b(
    t_year: np.ndarray,
    t: np.ndarray,
    y: np.ndarray,
    linear_regression_parameters: LinearRegressionParameters,
    error_mean: float,
    error_variance: float,
    save_path,
) -> None:
    x = construct_design_matrix(t)
    residuals = y - linear_regression_parameters.predict(x)
    plt.plot(t_year.reshape(-1), residuals.reshape(-1))
    plt.xlabel("date (decimal year)")
    plt.ylabel("residual")
    plt.title("2b: g_obs(t)")
    plt.savefig(save_path + "-residuals-timeseries")
    plt.close()

    count, bins = np.histogram(residuals, bins=100, density=True)
    plt.bar(bins[1:], count, label="residuals")
    plt.plot(
        bins[1:],
        scipy.stats.norm.pdf(bins[1:], loc=error_mean, scale=error_variance),
        color="red",
        label="e(t)",
    )
    plt.xlabel("residual bin")
    plt.ylabel("density")
    plt.title("2b: Residuals Density")
    plt.legend()
    plt.savefig(save_path + "-residuals-density-estimation")
    plt.close()


def c(
    kernel: CombinedKernel,
    kernel_parameters: CombinedKernelParameters,
    log_theta_range: np.ndarray,
    t: np.ndarray,
    number_of_samples: int,
    save_path: str,
) -> None:
    gram = kernel(t, **asdict(kernel_parameters))
    plt.imshow(gram)
    plt.xlabel("t")
    plt.ylabel("t")
    plt.title("Gram Matrix (Prior)")
    plt.savefig(save_path + "-gram-matrix")
    plt.close()

    for _ in range(number_of_samples):
        plt.plot(
            np.random.multivariate_normal(
                jnp.zeros(gram.shape[0]), gram, size=1
            ).reshape(-1)
        )
    plt.xlabel("t")
    plt.ylabel("f_GP(t)")
    plt.title("Samples from Gaussian Process Prior")
    plt.savefig(save_path + "-samples")
    plt.close()

    fig_samples, ax_samples = plt.subplots(
        len(fields(kernel_parameters.__class__)),
        len(log_theta_range),
        figsize=(
            len(log_theta_range) * 2,
            len(fields(kernel_parameters.__class__)) * 2,
        ),
        frameon=False,
    )
    for i, field in enumerate(fields(kernel_parameters.__class__)):
        default_value = getattr(kernel_parameters, field.name)
        for j, log_value in enumerate(log_theta_range):
            setattr(kernel_parameters, field.name, log_value)
            gram = kernel(t, **asdict(kernel_parameters))
            ax_samples[i][j].plot(
                np.random.multivariate_normal(
                    jnp.zeros(gram.shape[0]), gram, size=1
                ).reshape(-1),
            )
            ax_samples[i][j].set_title(
                f"{field.name.strip('log_')}={np.round(np.exp(log_value), 2)}"
            )
        setattr(kernel_parameters, field.name, default_value)
    plt.tight_layout()
    plt.savefig(save_path + f"-parameter-samples", bbox_inches="tight")
    plt.close(fig_samples)

    fig_gram, ax_gram = plt.subplots(
        len(fields(kernel_parameters.__class__)),
        len(log_theta_range),
        figsize=(
            len(log_theta_range) * 2,
            len(fields(kernel_parameters.__class__)) * 2,
        ),
        frameon=False,
    )
    for i, field in enumerate(fields(kernel_parameters.__class__)):
        default_value = getattr(kernel_parameters, field.name)
        for j, log_value in enumerate(log_theta_range):
            setattr(kernel_parameters, field.name, log_value)
            gram = kernel(t, **asdict(kernel_parameters))
            ax_gram[i][j].imshow(gram)
            ax_gram[i][j].set_title(
                f"{field.name.strip('log_')}={np.round(np.exp(log_value), 2)}"
            )
        setattr(kernel_parameters, field.name, default_value)
    plt.tight_layout()
    plt.savefig(save_path + f"-parameter-grams", bbox_inches="tight")
    plt.close(fig_gram)


def f(
    t_train: np.ndarray,
    y_train: np.ndarray,
    t_test: np.ndarray,
    min_year: float,
    prior_linear_regression_parameters: LinearRegressionParameters,
    linear_regression_sigma: float,
    kernel: CombinedKernel,
    gaussian_process_parameters: GaussianProcessParameters,
    learning_rate: float,
    number_of_iterations: int,
    save_path: str,
) -> None:
    # Train Bayesian Linear Regression
    x_train = construct_design_matrix(t_train)
    prior_theta = Theta(
        linear_regression_parameters=prior_linear_regression_parameters,
        sigma=linear_regression_sigma,
    )
    posterior_linear_regression_parameters = compute_linear_regression_posterior(
        x_train,
        y_train,
        prior_linear_regression_parameters,
        residuals_precision=prior_theta.precision,
    )

    residuals = y_train - posterior_linear_regression_parameters.predict(x_train)
    gaussian_process = GaussianProcess(
        kernel, t_train.reshape(-1, 1), residuals.reshape(-1)
    )

    # Prediction
    x_test = construct_design_matrix(t_test)
    linear_prediction = posterior_linear_regression_parameters.predict(x_test).reshape(
        -1
    )
    mean_prediction, covariance_prediction = gaussian_process.posterior_distribution(
        t_test.reshape(-1, 1), **asdict(gaussian_process_parameters)
    )

    # Plot
    plt.figure(figsize=(7, 7))
    plt.scatter(
        t_train + min_year,
        y_train.reshape(-1),
        s=2,
        color="blue",
        label="historical data",
    )
    plt.plot(
        t_test + min_year,
        linear_prediction + mean_prediction,
        color="gray",
        label="prediction",
    )
    plt.fill_between(
        t_test + min_year,
        linear_prediction
        + mean_prediction
        - 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
        linear_prediction
        + mean_prediction
        + 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
        facecolor=(0.8, 0.8, 0.8),
        label="error bound (one stdev)",
    )
    plt.xlabel("date (decimal year)")
    plt.ylabel("parts per million")
    plt.title("Global Mean CO_2 Concentration Prediction (Untrained Hyperparameters)")
    plt.legend()
    plt.tight_layout()
    plt.savefig(save_path + "-extrapolation-untrained", bbox_inches="tight")
    plt.close()

    df_parameters = pd.DataFrame(
        [
            [
                x.strip("log_") + " (kernel)",
                np.exp(gaussian_process_parameters.kernel[x]),
            ]
            for x in gaussian_process_parameters.kernel.keys()
        ]
        + [["sigma", float(gaussian_process_parameters.sigma)]],
        columns=["parameter", "value"],
    )
    df_parameters = df_parameters.set_index("parameter").sort_values(by=["parameter"])
    dfi.export(df_parameters, save_path + "-untrained-parameters.png")

    # Train Gaussian Process Regression (Hyperparameter Tune)
    optimizer = optax.adam(learning_rate)
    gaussian_process_parameters = gaussian_process.train(
        optimizer, number_of_iterations, **asdict(gaussian_process_parameters)
    )
    df_parameters = pd.DataFrame(
        [
            [
                x.strip("log_") + " (kernel)",
                np.exp(gaussian_process_parameters.kernel[x]),
            ]
            for x in gaussian_process_parameters.kernel.keys()
        ]
        + [["sigma", float(gaussian_process_parameters.sigma)]],
        columns=["parameter", "value"],
    )
    df_parameters = df_parameters.set_index("parameter").sort_values(by=["parameter"])
    dfi.export(df_parameters, save_path + "-trained-parameters.png")

    # Prediction
    x_test = construct_design_matrix(t_test)
    linear_prediction = posterior_linear_regression_parameters.predict(x_test).reshape(
        -1
    )
    mean_prediction, covariance_prediction = gaussian_process.posterior_distribution(
        t_test.reshape(-1, 1), **asdict(gaussian_process_parameters)
    )

    # Plot
    plt.figure(figsize=(7, 7))
    plt.scatter(
        t_train + min_year,
        y_train.reshape(-1),
        s=2,
        color="blue",
        label="historical data",
    )
    plt.plot(
        t_test + min_year,
        linear_prediction + mean_prediction,
        color="gray",
        label="prediction",
    )
    plt.fill_between(
        t_test + min_year,
        linear_prediction
        + mean_prediction
        - 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
        linear_prediction
        + mean_prediction
        + 1 * jnp.sqrt(jnp.diagonal(covariance_prediction)),
        facecolor=(0.8, 0.8, 0.8),
        label="error bound (one stdev)",
    )
    plt.xlabel("date (decimal year)")
    plt.ylabel("parts per million")
    plt.title("Global Mean CO_2 Concentration Prediction (Trained Hyperparameters)")
    plt.legend()
    plt.tight_layout()
    plt.savefig(save_path + "-extrapolation-trained", bbox_inches="tight")
    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:

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

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:

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

The expectation distributed to the relevant terms:

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

Evaluating the expectations:

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

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

To compute the second term:

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

Taking the logarithm:

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

The expectation distributed to the relevant terms:

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

Evaluating the expectations:

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

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

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

Thus,

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

Substituting the appropriate values:

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

Combining, we have our free energy expression:

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

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:

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

where $\mu \in \mathbb{R}^{K \times D}$, $\mathbf{s} \in \mathbb{R}^{K \times N}$, and $\mathbf{x} \in \mathbb{R}^{N \times D}$.
This mimics the least squares solution:

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

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

(c)

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

μ:
- The inversion ESS^{-1} where $\text{ESS} \in \mathbb{R}^{K \times K}$ is $\mathcal{O}(K^3)$
- The dot product $\text{ESS}^{-1}\text{ES}^T$ where $\text{ESS}^{-1} \in \mathbb{R}^{K \times K}$ and $\text{ES} \in \mathbb{R}^{N \times K}$ is $\mathcal{O}(K^2N)$
- The dot product $(\text{ESS}^{-1}\text{ES}^T)\mathbf{x}$ where $(\text{ESS}^{-1}\text{ES}^T) \in \mathbb{R}^{K \times N}$ and $\mathbf{x} \in \mathbb{R}^{N \times D}$ is $\mathcal{O}(KND)$

σ:
- The dot product $(\mathbf{x}^T\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^{N \times D}$ is $\mathcal{O}(D^2N)$
- The dot product $\mu^T\mu$ where $\mu \in \mathbb{R}^{D \times K}$ is $\mathcal{O}(K^2D)$
- The dot product $(\mu^T\mu)\text{ESS}$ where $\mu^T\mu \in \mathbb{R}^{K \times K}$ and $\text{ESS} \in \mathbb{R}^{K \times K}$ is $\mathcal{O}(K^3)$

π:
- The mean operation for $\text{ES} \in \mathbb{R}^{N \times K}$ along the first dimension is $\mathcal{O}(NK)$

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

(d)

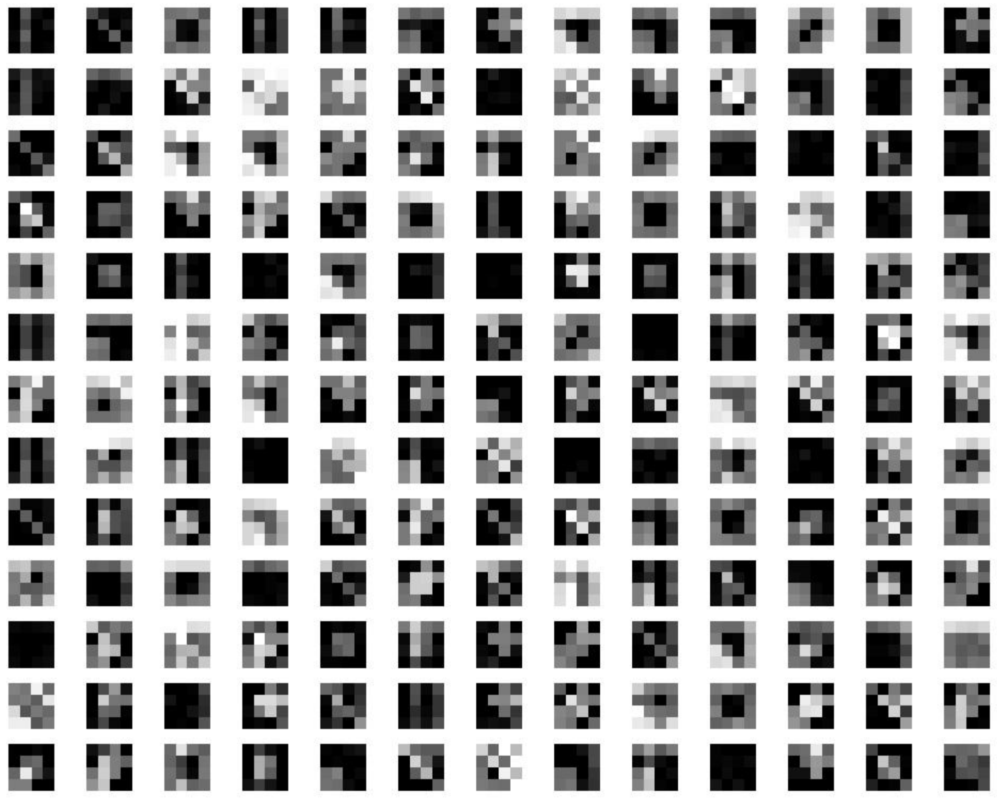


Figure 13: Images generated by randomly combined features with noise

Examining the generated images, we can see eight features:

(1) a cross

(2) a border

(3) a two by two square in the middle

(4) a two by two square in the bottom left corner

(5) a diagonal from top left to bottom right

(6) a vertical line in the second column

(7) a vertical line in the fourth column

(8) a a horizontal line in the first row

Factor analysis assumes a model:

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

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

A mixture of Gaussians assumes as model:

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

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

The independent component analysis assumes a model:

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

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

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

(e)

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

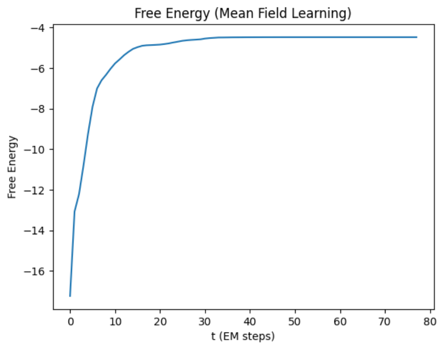


Figure 14: Free Energy

(f)

The initialised features:

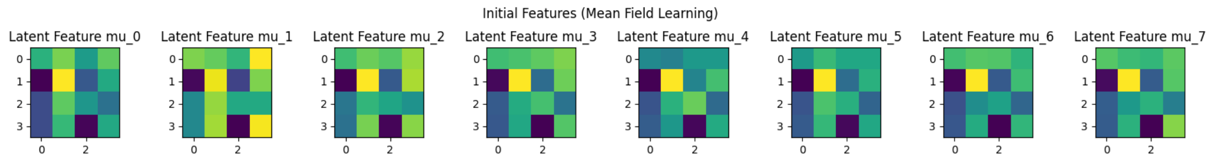


Figure 15: Initial Latent Factors

The features learned by the algorithm:

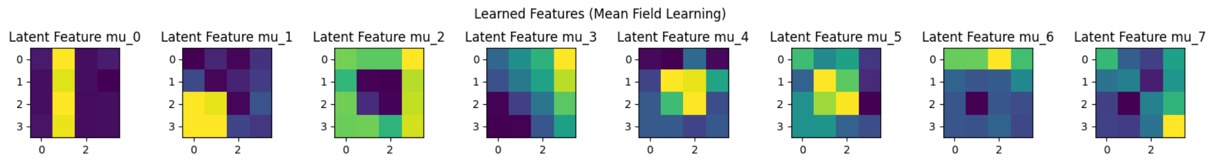


Figure 16: Learned Latent Factors

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

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

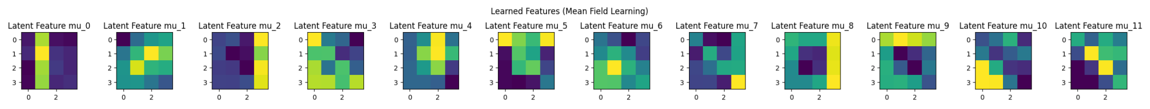


Figure 17: Increasing Number of Latent Factors

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

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

(g)

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

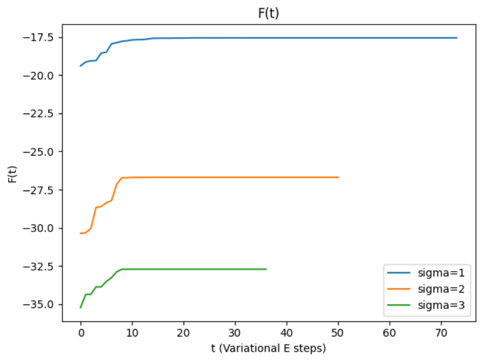


Figure 18: Free energy vs σ

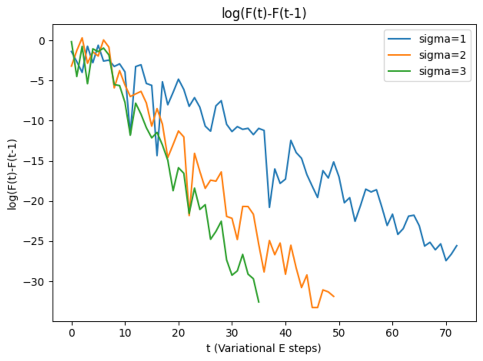


Figure 19: Free energy convergence vs σ

We know that our free energy is a upper bounded on the log likelihood:

$$\log P(\mathcal{X}|\theta) \geq \mathcal{F}(q, \theta)$$

In the variational expectation step, $\log P(\mathcal{X}|\theta)$ is fixed and we adjust our approximation q to reach this upper bound. We know that σ quantifies the noise of $\mathbf{x}$, thus a higher σ means a wider spread in our distribution $\log P(\mathcal{X}|\theta)$, meaning we are reducing our upper bound for $\mathcal{F}(q, \theta)$. As such, we can see in the plot for free energy above that when σ is increased, our free energy converges to a lower value, due to being bounded above by a lower log-likelihood. Moreover, by reducing the upper bound, we see in the plot of $\log(F(t) - F(t-1))$ that our free energy is able to converge faster. Because we have reduced the upper bound by increasing σ, our free energy can reach this upper bound faster.

The Python code for the binary latent factor model:

```python
from typing import TYPE_CHECKING, Tuple

import numpy as np

from demo_code.MStep import m_step
from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
    AbstractBinaryLatentFactorApproximation,
)
from src.models.binary_latent_factor_models.abstract_binary_latent_factor_model import (
    AbstractBinaryLatentFactorModel,
)


class BinaryLatentFactorModel(AbstractBinaryLatentFactorModel):
    """
    mu: matrix of means (number_of_dimensions, number_of_latent_variables)
    sigma: gaussian noise parameter
    pi: vector of priors (1, number_of_latent_variables)
    """

    def __init__(
        self,
        mu: np.ndarray,
        sigma: float,
        pi: np.ndarray,
    ):
        self._mu = mu
        self._sigma = sigma
        self._pi = pi

    @property
    def mu(self):
        return self._mu

    @mu.setter
    def mu(self, value):
        self._mu = value

    @property
    def sigma(self):
        return self._sigma

    @sigma.setter
    def sigma(self, value):
        self._sigma = value

    @property
    def pi(self):
        return self._pi

    @pi.setter
    def pi(self, value):
        self._pi = value

    @property
    def variance(self) -> float:
        return self.sigma**2

    @staticmethod
    def calculate_maximisation_parameters(
        x: np.ndarray,
        binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
    ) -> Tuple[np.ndarray, float, np.ndarray]:
        return m_step(
            x=x,
            es=binary_latent_factor_approximation.expectation_s,
            ess=binary_latent_factor_approximation.expectation_ss,
        )

    def maximisation_step(
        self,
        x: np.ndarray,
        binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
    ) -> None:
        mu, sigma, pi = self.calculate_maximisation_parameters(
            x, binary_latent_factor_approximation
        )
        self.mu = mu
        self.sigma = sigma
        self.pi = pi


def init_binary_latent_factor_model(
    x: np.ndarray,
    binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
) -> BinaryLatentFactorModel:
    mu, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
        x, binary_latent_factor_approximation
    )
    return BinaryLatentFactorModel(mu, sigma, pi)
```

src/models/binary_latent_factor_models/binary_latent_factor_model.py

The Python code for mean field learning:

```python
from typing import List

import numpy as np

from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
    AbstractBinaryLatentFactorApproximation,
)
from src.models.binary_latent_factor_models.binary_latent_factor_model import (
    AbstractBinaryLatentFactorModel,
)


class MeanFieldApproximation(AbstractBinaryLatentFactorApproximation):
    """
    lambda_matrix: parameters variational approximation (number_of_points, number_of_latent_variables)
    """

    _lambda_matrix: np.ndarray

    def __init__(self, lambda_matrix, max_steps, convergence_criterion):
        self.lambda_matrix = lambda_matrix
        self.max_steps = max_steps
        self.convergence_criterion = convergence_criterion

    @property
    def lambda_matrix(self) -> np.ndarray:
        return self._lambda_matrix

    @lambda_matrix.setter
    def lambda_matrix(self, value):
        self._lambda_matrix = value

    def lambda_matrix_exclude(self, exclude_latent_index: int) -> np.ndarray:
        # (number_of_points, number_of_latent_variables-1)
        return np.concatenate(
            (
                self.lambda_matrix[:, :exclude_latent_index],
                self.lambda_matrix[:, exclude_latent_index + 1 :],
            ),
            axis=1,
        )

    def _partial_expectation_step(
        self,
        x: np.ndarray,
        binary_latent_factor_model: AbstractBinaryLatentFactorModel,
        latent_factor: int,
    ) -> np.ndarray:
        """Partial Variational E step for factor i for all data points

        :param x: data matrix (number_of_points, number_of_dimensions)
        :param binary_latent_factor_model: a binary_latent_factor_model
        :param latent_factor: latent factor to compute partial update
        :return: lambda_vector: new lambda parameters for the latent factor (number_of_points, 1)
        """
        lambda_matrix_excluded = self.lambda_matrix_exclude(latent_factor)
        mu_excluded = binary_latent_factor_model.mu_exclude(latent_factor)

        mu_latent = binary_latent_factor_model.mu[:, latent_factor]
        # (number_of_points, 1)
        partial_expectation_log_p_x_given_s_theta_proportion = (
            binary_latent_factor_model.precision
            * (
                x  # (number_of_points, number_of_dimensions)
                - 0.5 * mu_latent.T  # (1, number_of_dimensions)
                - lambda_matrix_excluded  # (number_of_points, number_of_latent_variables-1)
                @ mu_excluded.T  # (number_of_latent_variables-1, number_of_dimensions)
            )
            @ mu_latent  # (number_of_dimensions, 1)
        )

        # (1, 1)
        partial_expectation_log_p_s_given_theta_proportion = np.log(
            binary_latent_factor_model.pi[0, latent_factor]
            / (1 - binary_latent_factor_model.pi[0, latent_factor])
        )

        # (number_of_points, 1)
        partial_expectation_log_p_x_s_given_theta_proportion = (
            partial_expectation_log_p_x_given_s_theta_proportion
            + partial_expectation_log_p_s_given_theta_proportion
        )

        # (number_of_points, 1)
        lambda_vector = 1 / (
            1 + np.exp(-partial_expectation_log_p_x_s_given_theta_proportion)
        )
        lambda_vector[lambda_vector == 0] = 1e-10
        lambda_vector[lambda_vector == 1] = 1 - 1e-10
        return lambda_vector

    def variational_expectation_step(
        self, x: np.ndarray, binary_latent_factor_model: AbstractBinaryLatentFactorModel
    ) -> List[float]:
        """Variational E step

        :param binary_latent_factor_model: a binary_latent_factor_model
        :param x: data matrix (number_of_points, number_of_dimensions)
        """
        free_energy = [self.compute_free_energy(x, binary_latent_factor_model)]
        for i in range(self.max_steps):
            for latent_factor in range(binary_latent_factor_model.k):
                self.lambda_matrix[:, latent_factor] = self._partial_expectation_step(
                    x, binary_latent_factor_model, latent_factor
                )
                free_energy.append(
                    self.compute_free_energy(x, binary_latent_factor_model)
                )
                if free_energy[-1] - free_energy[-2] <= self.convergence_criterion:
                    break
            if free_energy[-1] - free_energy[-2] <= self.convergence_criterion:
                break
        return free_energy


def init_mean_field_approximation(
    k: int, n: int, max_steps, convergence_criterion
) -> MeanFieldApproximation:
    return MeanFieldApproximation(
        lambda_matrix=np.random.random(size=(n, k)),
        max_steps=max_steps,
        convergence_criterion=convergence_criterion,
    )
```

src/models/binary_latent_factor_approximations/mean_field_approximation.py

The Python code for expectation maximisation:

```python
from __future__ import annotations

from typing import TYPE_CHECKING, List, Tuple

import numpy as np

if TYPE_CHECKING:
    from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
        AbstractBinaryLatentFactorApproximation,
    )
    from src.models.binary_latent_factor_models.binary_latent_factor_model import (
        AbstractBinaryLatentFactorModel,
    )


def is_converge(
    free_energies: List[float],
    current_lambda_matrix: np.ndarray,
    previous_lambda_matrix: np.ndarray,
) -> bool:
    return (abs(free_energies[-1] - free_energies[-2]) == 0) and np.linalg.norm(
        current_lambda_matrix - previous_lambda_matrix
    ) == 0


def learn_binary_factors(
    x: np.ndarray,
    em_iterations: int,
    binary_latent_factor_model: AbstractBinaryLatentFactorModel,
    binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
) -> Tuple[
    AbstractBinaryLatentFactorApproximation,
    AbstractBinaryLatentFactorModel,
    List[float],
]:
    free_energies: List[float] = [
        binary_latent_factor_approximation.compute_free_energy(
            x, binary_latent_factor_model
        )
    ]
    for _ in range(em_iterations):
        previous_lambda_matrix = np.copy(
            binary_latent_factor_approximation.lambda_matrix
        )
        binary_latent_factor_approximation.variational_expectation_step(
            x=x,
            binary_latent_factor_model=binary_latent_factor_model,
        )
        binary_latent_factor_model.maximisation_step(
            x,
            binary_latent_factor_approximation,
        )
        free_energies.append(
            binary_latent_factor_approximation.compute_free_energy(
                x, binary_latent_factor_model
            )
        )
        if is_converge(
            free_energies,
            binary_latent_factor_approximation.lambda_matrix,
            previous_lambda_matrix,
        ):
            break
    return binary_latent_factor_approximation, binary_latent_factor_model, free_energies
```

src/expectation_maximisation.py

The rest of the Python code for question 3:

```python
from typing import List

import matplotlib.pyplot as plt
import numpy as np

from src.expectation_maximisation import is_converge, learn_binary_factors
from src.models.binary_latent_factor_approximations.mean_field_approximation import (
    init_mean_field_approximation,
)
from src.models.binary_latent_factor_models.binary_latent_factor_model import (
    AbstractBinaryLatentFactorModel,
    init_binary_latent_factor_model,
)


def e_and_f(
    x: np.ndarray,
    k: int,
    em_iterations: int,
    e_maximum_steps: int,
    e_convergence_criterion: float,
    save_path: str,
) -> AbstractBinaryLatentFactorModel:
    n = x.shape[0]
    mean_field_approximation = init_mean_field_approximation(
        k, n, max_steps=e_maximum_steps, convergence_criterion=e_convergence_criterion
    )
    binary_latent_factor_model = init_binary_latent_factor_model(
        x, mean_field_approximation
    )
    fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
    for i in range(k):
        ax[i].imshow(binary_latent_factor_model.mu[:, i].reshape(4, 4))
        ax[i].set_title(f"Latent Feature mu_{i}")
    fig.suptitle("Initial Features (Mean Field Learning)")
    plt.tight_layout()
    plt.savefig(save_path + "-init-latent-factors", bbox_inches="tight")
    plt.close()
    _, binary_latent_factor_model, free_energy = learn_binary_factors(
        x,
        em_iterations,
        binary_latent_factor_model,
        binary_latent_factor_approximation=mean_field_approximation,
    )
    fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
    for i in range(k):
        ax[i].imshow(binary_latent_factor_model.mu[:, i].reshape(4, 4))
        ax[i].set_title(f"Latent Feature mu_{i}")
    fig.suptitle("Learned Features (Mean Field Learning)")
    plt.tight_layout()
    plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
    plt.close()

    plt.title("Free Energy (Mean Field Learning)")
    plt.xlabel("t (EM steps)")
    plt.ylabel("Free Energy")
    plt.plot(free_energy)
    plt.savefig(save_path + "-free-energy", bbox_inches="tight")
    plt.close()
    return binary_latent_factor_model


def g(
    x: np.ndarray,
    binary_latent_factor_model: AbstractBinaryLatentFactorModel,
    sigmas: List[float],
    k: int,
    em_iterations: int,
    e_maximum_steps: int,
    e_convergence_criterion: float,
    save_path: str,
) -> None:
    n = x.shape[0]
    free_energies = []
    for sigma in sigmas:
        binary_latent_factor_model.sigma = sigma
        mean_field_approximation = init_mean_field_approximation(
            k,
            n,
            max_steps=e_maximum_steps,
            convergence_criterion=e_convergence_criterion,
        )
        free_energy: List[float] = [
            mean_field_approximation.compute_free_energy(x, binary_latent_factor_model)
        ]
        for _ in range(em_iterations):
            free_energy.pop(-1)
            previous_lambda_matrix = np.copy(mean_field_approximation.lambda_matrix)
            new_free_energy = mean_field_approximation.variational_expectation_step(
                binary_latent_factor_model=binary_latent_factor_model,
                x=x,
            )
            free_energy.extend(new_free_energy)
            if (
                free_energy[-1] - free_energy[-2]
                <= mean_field_approximation.convergence_criterion
            ):
                free_energy.pop(-1)
                break
            if is_converge(
                free_energy,
                mean_field_approximation.lambda_matrix,
                previous_lambda_matrix,
            ):
                break
        free_energies.append(free_energy)

    for i, free_energy in enumerate(free_energies):
        plt.plot(
            free_energy,
            label=f"sigma={sigmas[i]}",
        )
    plt.title(f"F(t)")
    plt.xlabel("t (Variational E steps)")
    plt.ylabel("F(t)")
    plt.tight_layout()
    plt.legend()
    plt.savefig(save_path + f"-free-energy-sigma.png", bbox_inches="tight")
    plt.close()

    for i, free_energy in enumerate(free_energies):
        diffs = np.log(np.diff(free_energy))
        plt.plot(
            diffs,
            label=f"sigma={sigmas[i]}",
        )
    plt.title(f"log(F(t)-F(t-1)")
    plt.xlabel("t (Variational E steps)")
    plt.ylabel("log(F(t)-F(t-1)")
    plt.tight_layout()
    plt.legend()
    plt.savefig(save_path + f"-free-energy-diff-sigma.png", bbox_inches="tight")
    plt.close()
```

src/solutions/q3.py

Question 4

(a)

We begin by writing the expression for x_d:

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

where we know from the diagonal covariance of $P(\mathbf{x}|\mathbf{s}, \mu, \sigma^2)$ that each dimension is independent. Moreover, $\mathbf{w}_d \in \mathbb{R}^{K\times 1}$, which is the d^{th} row of $\mu \in \mathbb{R}^{D\times K}$

Thus, we can write the posterior:

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

where we introduce priors on each $\mathbf{w}_k$ with $\alpha \in \mathbb{R}^{K\times 1}$.

We choose each prior to be:

$$P(\mathbf{w}_d|\alpha) = \mathcal{N}(0, \mathbf{A}^{-1})$$

where $\mathbf{A} = diag(\alpha)$, the precision matrix.

Combining, we have our expression:

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

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:

$$q_s(\mathbf{s}) \propto \exp\left\langle \log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha)\right\rangle_{q(\mu)}$$

Substituting the relevant terms:

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

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

$$\frac{\partial}{\partial q_i}\left(\exp\left\langle \log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha)\right\rangle_{q(\mu)} + \lambda^{LG}\int q_i - 1)\right) \propto \exp\left\langle \log P(\mathbf{x}, \mathbf{s}, \mu|\pi, \sigma^2, \alpha)\right\rangle_{q(\mu)\prod_{j\neq i} q_j(s_j)} - \log q_i(s_i)$$

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

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

we have our partial E step update.

For the maximisation step, we perform maximisation steps for the parameters σ and π in the same way as question 3. However, having defined a prior on μ (through $\mathbf{w}$) so we will have to derive our expression for $\langle\mu_k\rangle_{q_{\mu_k}}$ the expectation of the posterior on μ_k. This involves deriving the posterior distribution of $\mathbf{w}_d$

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

Substituting the appropriate terms:

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

Simplifying:

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

We see that the posterior:

$$q_{\mathbf{w}_d}(\mathbf{w}_d) = \mathcal{N}\left(\mu_{\mathbf{w}_d},\Sigma_{\mathbf{w}_d}\right)$$

where:

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

and

$$\mu_{\mathbf{w}_d} = \Sigma_{\mathbf{w}_d}\left(\frac{x_d\langle\mathbf{s}^T\rangle_{q_{\mathbf{s}}(\mathbf{s})}}{\sigma^2}\right)$$

Thus, $\langle\mu_k\rangle_{q_{\mu_k}} \in \mathbb{R}^{D\times1}$ is simply the concatenation of the k^{th} elements of $\mu_{\mathbf{w}_d}$ for $d\in\{1,...,D\}$

For ARD, we must also optimise α with a hyper-M step. We start by choose $Ga(\alpha_k|a,b)$, a Gamma prior on α_k, with a and b being hyperparameters. Thus, to optimise α we want to maximise the penalised objective:

$$\alpha = \arg\max_{\alpha}\left\langle\log P(\mathbf{x},\mathbf{s},\mu|\pi,\sigma^2,\alpha)\right\rangle_{q(\mathbf{w})}+\sum_{k=1}^{K}\log P(\alpha_k|a,b)$$

Substituting the appropriate terms, we have our penalised objective $\mathcal{Q}$:

$$\mathcal{Q} = \left\langle\sum_{d=1}^{D}\frac{1}{2}\sum_{k=1}^{K}(\log\alpha_k)-\frac{1}{2}\mathbf{w}_d^T\mathbf{A}\mathbf{w}_d\right\rangle_{q(\mathbf{w})}+\sum_{k=1}^{K}(a-1)\log\alpha_k-b\alpha_k$$

Simplifying:

$$\mathcal{Q} = \frac{D}{2}\sum_{k=1}^{K}(\log \alpha_k) - \frac{1}{2}\sum_{d=1}^{D}\left(tr\left[\mathbf{A}\left\langle \mathbf{w}_d \mathbf{w}_d^T \right\rangle_{q(\mathbf{w}_d)}\right]\right) + \sum_{k=1}^{K}(a-1)\log \alpha_k - b\alpha_k$$

Setting $\frac{d\mathcal{Q}}{d\alpha_k} = 0$ we get:

$$\frac{D}{2\alpha_k} - \frac{1}{2}\sum_{d=1}^{D}\left\langle (w_{d,k})^2 \right\rangle_{q(\mathbf{w}_d)} + \frac{a-1}{\alpha_k} - b = 0$$

where $w_{d,k}$ is the k^{th} element of $\mathbf{w}_d$.
Knowing $\left\langle (w_{d,k})^2 \right\rangle_{q(\mathbf{w}_d)} = (\mu_{\mathbf{w}_{d,k}})^2 + \Sigma_{\mathbf{w}_{d,(k,k)}}$, we can solve for α_k:

$$\alpha_k = \frac{2a + D - 2}{2b + \sum_{d=1}^{D}\left((\mu_{\mathbf{w}_{d,k}})^2 + \Sigma_{\mathbf{w}_{d,(k,k)}}\right)}$$

we have our hyper-M steps for optimising α.

(b)

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

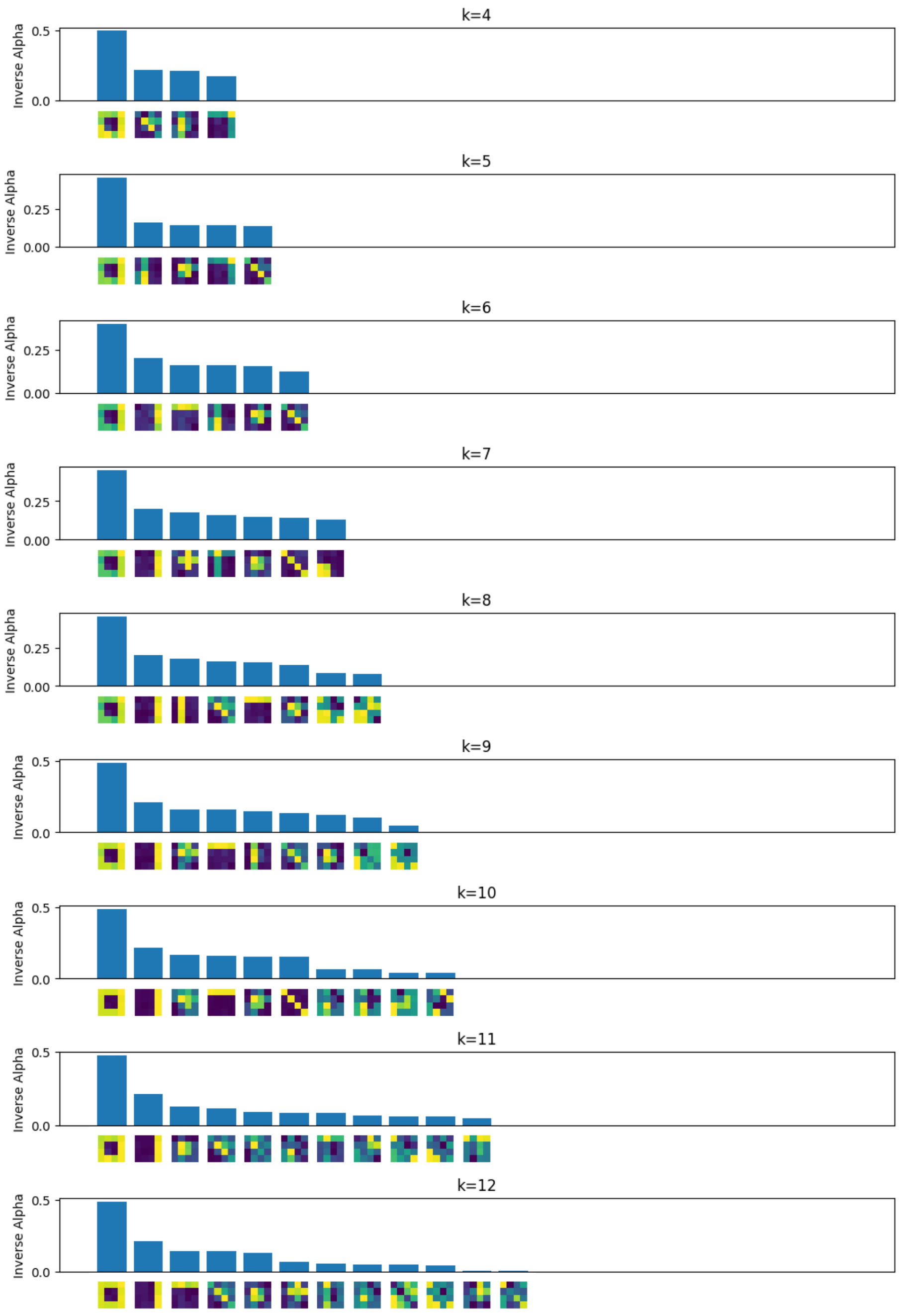


Figure 20: Learned Latent Factors vs Inverse Alpha

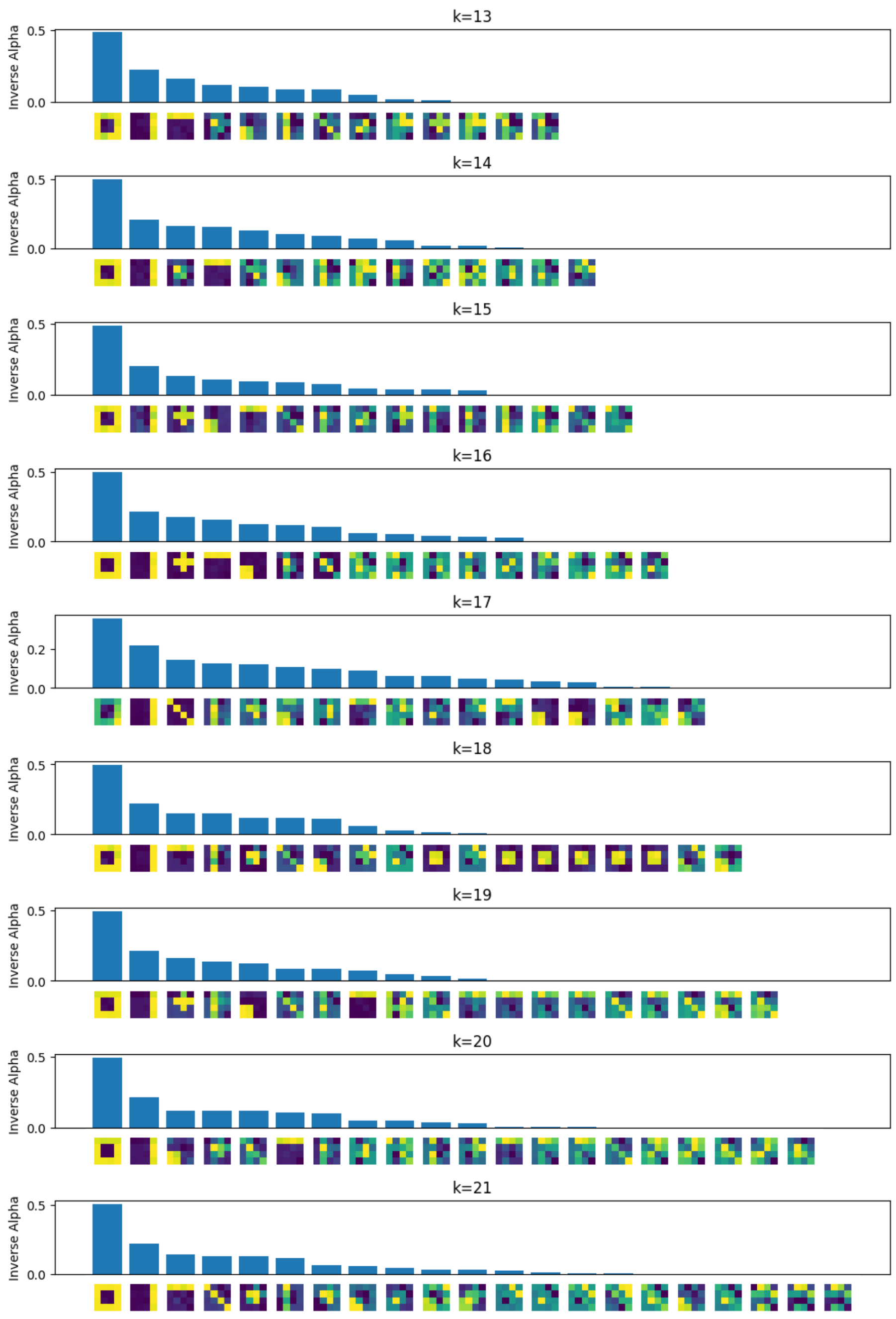


Figure 21: Learned Latent Factors vs Inverse Alpha

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

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

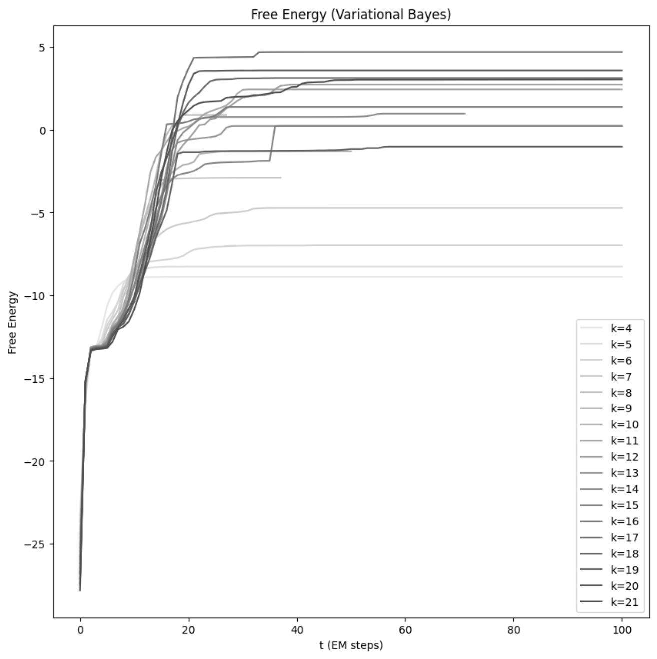


Figure 22: Free Energy for different values of k

We can see that initially, for $k = 4$ to $k = 8$, increasing k significantly increases the convergence value of the free energy. However, beyond $k = 8$ there is a no clear trend of k versus the free energy convergence value. We can see that this corresponds to the visualisation of α^{-1} where beyond $k = 11$, the effective number of features remains more or less the same. We know that there are only eight latent features, thus models with $k > 8$ should be learning duplicate or irrelevant features. As such, we wouldn't expect a model to be able to increase it's free energy significantly when provided with additional degrees of freedom by increasing the value of k beyond eight. We see that for models with $k >> 8$, there are typically ten or eleven features that might be deemed relevant (depending on how you threshold) and this is likely from slight overfitting, noise in the data, or duplicate features. Thus, the relationship between the free energy and the effective number of latent features for each model is as we would expect with ARD.

The Python code for Variational Bayes:

```python
import numpy as np

from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
    AbstractBinaryLatentFactorApproximation,
)
from src.models.binary_latent_factor_models.binary_latent_factor_model import (
    AbstractBinaryLatentFactorModel,
    BinaryLatentFactorModel,
)


class GaussianPrior:
    def __init__(self, a, b, d, k):
        self.a = a
        self.b = b
        self.mu = np.zeros((d, k))
        self.alpha = np.ones((k,))  # np.random.gamma(a, b, size=(k,))
        self.w_covariance = np.zeros((k, k))

    def mu_k(self, k):  # (number_of_dimensions,  1)
        return self.mu[:, k : k + 1]

    def w_d(self, d):  # (1,  number_of_latent_variables)
        return self.mu[d : d + 1, :]

    @property
    def a_matrix(self) -> np.ndarray:
        #  precision matrix for w_d
        return np.diag(self.alpha)


class VariationalBayesBinaryLatentFactorModel(AbstractBinaryLatentFactorModel):
    def __init__(self, mu: GaussianPrior, variance: float, pi: np.ndarray):
        self.gaussian_prior = mu
        self._variance = variance
        self._pi = pi

    @property
    def variance(self) -> float:
        return self._variance

    @property
    def pi(self) -> np.ndarray:
        return self._pi

    @property
    def mu(self) -> np.ndarray:
        return self.gaussian_prior.mu

    def _update_w_d_covariance(
        self,
        binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
    ):
        #  expectation_s (number_of_points, number_of_latent_variables)
        #  expectation_ss (number_of_latent_variables, number_of_latent_variables)
        self.gaussian_prior.w_covariance = np.linalg.inv(
            self.gaussian_prior.a_matrix
            + self.precision * binary_latent_factor_approximation.expectation_ss
        )

    def _update_w_d_mean(
        self,
        x: np.ndarray,  # (number_of_points, number_of_dimensions)
        binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
        d: int,
    ):
        # (number_of_latent_variables x 1)
        self.gaussian_prior.mu[d : d + 1, :] = (
            self.gaussian_prior.w_covariance
            @ (  # (number_of_latent_variables, number_of_latent_variables)
                self.precision
                * binary_latent_factor_approximation.expectation_s.T  # (number_of_latent_variables,
 number_of_points)
                @ x[:, d : d + 1]  # (number_of_points, 1)
            )
        ).T

    def _hyper_maximisation_step(self):
        for k in range(self.k):
            self.gaussian_prior.alpha[k] = (2 * self.gaussian_prior.a + self.d - 2) / (
                2 * self.gaussian_prior.b
                + np.sum(self.gaussian_prior.mu_k(k) ** 2)
                + self.d * self.gaussian_prior.w_covariance[k, k]
            )

    def maximisation_step(
        self,
        x: np.ndarray,
        binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
    ) -> None:
        _, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
            x, binary_latent_factor_approximation
        )
        self._variance = sigma**2
        self._pi = pi
        self._update_w_d_covariance(binary_latent_factor_approximation)
        for d in range(self.d):
            self._update_w_d_mean(x, binary_latent_factor_approximation, d)
        self._hyper_maximisation_step()
```

src/models/binary_latent_factor_models/variational_bayes.py

The rest of the Python code for question 4:

```
import os
from typing import List, Tuple

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.offsetbox import AnnotationBbox, OffsetImage

from src.expectation_maximisation import learn_binary_factors
from src.models.binary_latent_factor_approximations.mean_field_approximation import (
    init_mean_field_approximation,
)
from src.models.binary_latent_factor_models.binary_latent_factor_model import (
    BinaryLatentFactorModel,
)
from src.models.binary_latent_factor_models.variational_bayes import (
    GaussianPrior,
    VariationalBayesBinaryLatentFactorModel,
)


def offset_image(coord, path, ax):
    img = plt.imread(path)
    im = OffsetImage(img, zoom=0.72)
    im.image.axes = ax

    ab = AnnotationBbox(
        im,
        (coord, 0),
        xybox=(0.0, -19.0),
        frameon=False,
        xycoords="data",
        boxcoords="offset points",
        pad=0,
    )

    ax.add_artist(ab)


def _run_automatic_relevance_determination(
    x: np.ndarray,
    a_parameter: int,
    b_parameter: int,
    k: int,
    em_iterations: int,
    e_maximum_steps: int,
    e_convergence_criterion: float,
) -> Tuple[VariationalBayesBinaryLatentFactorModel, List[float]]:
    n = x.shape[0]
    mean_field_approximation = init_mean_field_approximation(
        k, n, max_steps=e_maximum_steps, convergence_criterion=e_convergence_criterion
    )
    (_, sigma, pi,) = BinaryLatentFactorModel.calculate_maximisation_parameters(
        x, mean_field_approximation
    )
    mu = GaussianPrior(
        a=a_parameter,
        b=b_parameter,
        k=k,
        d=x.shape[1],
    )
    binary_latent_factor_model: VariationalBayesBinaryLatentFactorModel = (
        VariationalBayesBinaryLatentFactorModel(
            mu=mu,
            variance=sigma**2,
            pi=pi,
        )
    )
    (_, binary_latent_factor_model, free_energy,) = learn_binary_factors(
        x=x,
        em_iterations=em_iterations,
        binary_latent_factor_model=binary_latent_factor_model,
        binary_latent_factor_approximation=mean_field_approximation,
    )
    return binary_latent_factor_model, free_energy


def b(
    x: np.ndarray,
    a_parameter: int,
    b_parameter: int,
    ks: List[int],
    max_k: int,
    em_iterations: int,
    e_maximum_steps: int,
    e_convergence_criterion: float,
    save_path: str,
) -> None:

    binary_latent_factor_models = []
    free_energies = []
    for i, k in enumerate(ks):
        (
            binary_latent_factor_model,
            free_energy,
        ) = _run_automatic_relevance_determination(
            x,
            a_parameter,
            b_parameter,
            k,
            em_iterations,
            e_maximum_steps,
            e_convergence_criterion,
        )
        binary_latent_factor_models.append(binary_latent_factor_model)
        free_energies.append(free_energy)

    n = len(ks)
    m = np.max(ks)
    fig = plt.figure()
    fig.set_figwidth(2 * n)
    fig.set_figheight(2 * m)
    for i, k in enumerate(ks):
        sort_indices = np.argsort(binary_latent_factor_models[i].gaussian_prior.alpha)
        for j, idx in enumerate(sort_indices):
            ax = plt.subplot(n, m, m * i + j + 1)
            ax.imshow(binary_latent_factor_models[i].mu[:, idx].reshape(4, 4))
            ax.set_title(f"Latent Feature {idx+1}/{k}")
    fig.suptitle("Learned Features (Variational Bayes)")
    plt.tight_layout()
    plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
    plt.close()

    for i, k in enumerate(ks):
        sort_indices = np.argsort(binary_latent_factor_models[i].gaussian_prior.alpha)
        for j, idx in enumerate(sort_indices):
            fig = plt.figure(figsize=(0.3, 0.3))
            ax = plt.Axes(fig, [0.0, 0.0, 1.0, 1.0])
            ax.set_axis_off()
            fig.add_axes(ax)
            ax.imshow(binary_latent_factor_models[i].mu[:, idx].reshape(4, 4))
            fig.savefig(save_path + f"-latent-factor-{i}-{j}", bbox_inches="tight")
            plt.close()

    fig, ax = plt.subplots(len(ks), 1, figsize=(12, 2 * len(ks)))
    plt.subplots_adjust(hspace=1)
    for i, k in enumerate(ks):
        sort_indices = np.argsort(binary_latent_factor_models[i].gaussian_prior.alpha)
        y = list(
            1 / binary_latent_factor_models[i].gaussian_prior.alpha[sort_indices]
        ) + [0] * (max_k - k)
        ax[i].set_title(f"{k=}")
        ax[i].bar(range(max_k), y)
        ax[i].set_xticks([])
        ax[i].set_ylabel("Inverse Alpha")
    for i, k in enumerate(ks):
        sort_indices = np.argsort(binary_latent_factor_models[i].gaussian_prior.alpha)
        for j in range(len(sort_indices)):
            path = save_path + f"-latent-factor-{i}-{j}.png"
            offset_image(j, path, ax[i])
            os.remove(path)
    fig.savefig(save_path + f"-latent-factors-comparison", bbox_inches="tight")
    plt.close()

    fig = plt.figure()
    fig.set_figwidth(10)
    fig.set_figheight(10)
    shades = np.flip(np.linspace(0.3, 0.9, len(ks)))
    for i, k in enumerate(ks):
        plt.plot(free_energies[i], label=f"{k=}", color=np.ones(3) * shades[i])
    plt.title("Free Energy (Variational Bayes)")
    plt.xlabel("t (EM steps)")
    plt.ylabel("Free Energy")
    plt.legend()
    plt.savefig(save_path + "-free-energy", bbox_inches="tight")
    plt.close()
```
src/solutions/q4.py

Question 5

(a)

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

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

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

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

Expanding:

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

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:

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

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

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

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

Simplifying:

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

and,

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

Thus, the cavity distributions are:

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

and

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

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

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

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:

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

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

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

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

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

We can write:

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

Simplifying:

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

Thus, the first moments:

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

and

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

Moreover:

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

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:

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

Thus, the first moment:

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

Simplifying:

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

Similarly:

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

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:

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

Simplifying:

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

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

Our parameter update:

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

Similarly:

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

(c)

Using factored approximate messages, we see that:

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

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

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

and

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

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

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

(d)

Similar to question 3, we can use automatic relevance determination (ARD) as a hyperparameter method to select relevant features by placing a prior on μ_i. With a hyper-M step, certain features will have diverging precision, indicating that they are not relevant to the model output. Thus, the number of remaining features will be our selection for K.

Question 6

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

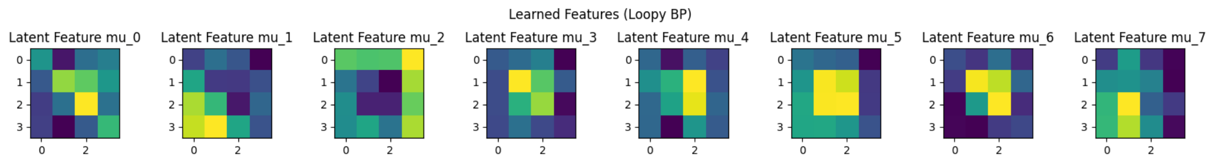


Figure 23: Learned Latent factors learned with EP/Loopy-BP

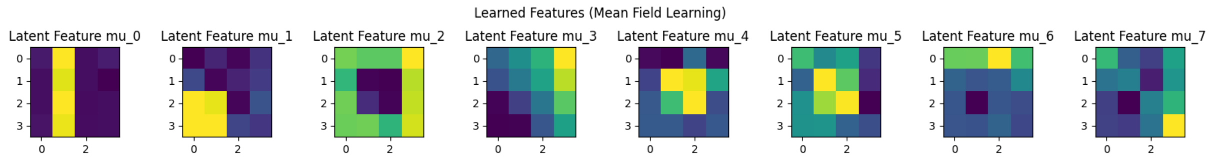


Figure 24: Learned Latent Factors with Mean Field Approximation

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

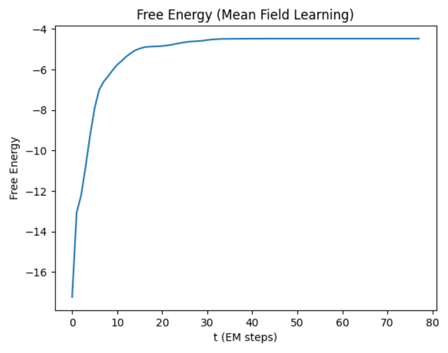


Figure 25: Mean Field Approximation

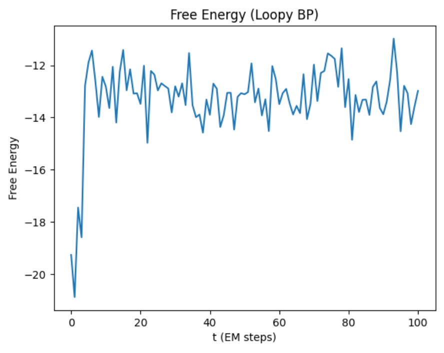


Figure 26: Loopy BP

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

The Python code for the Boltzmann machine:

```python
import numpy as np

from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
    AbstractBinaryLatentFactorApproximation,
)
from src.models.binary_latent_factor_models.binary_latent_factor_model import (
    BinaryLatentFactorModel,
)


class BoltzmannMachine(BinaryLatentFactorModel):
    """
    mu: matrix of means (number_of_dimensions, number_of_latent_variables)
    sigma: gaussian noise parameter
    pi: vector of priors (1, number_of_latent_variables)
    """

    def __init__(
        self,
        mu: np.ndarray,
        sigma: float,
        pi: np.ndarray,
    ):
        super().__init__(mu, sigma, pi)

    @property
    def w_matrix(self) -> np.ndarray:
        # (number_of_latent_variables, number_of_latent_variables)
        return -self.precision * (self.mu.T @ self.mu)

    def w_matrix_index(self, i, j) -> float:
        return -self.precision * (self.mu[:, i] @ self.mu[:, j])

    def b(self, x) -> np.ndarray:
        """

        :param x: design matrix (number_of_points, number_of_dimensions)
        :return:
        """
        # (number_of_points, number_of_latent_variables)
        return -(
            self.precision * x @ self.mu
            + self.log_pi_ratio
            - 0.5 * self.precision * np.multiply(self.mu, self.mu).sum(axis=0)
        )

    def b_index(self, x, node_index) -> float:
        # (number_of_points, 1)
        return -(
            self.precision * x @ self.mu[:, node_index]
            + (self.log_pi[0, node_index] - self.log_one_minus_pi[0, node_index])
            - 0.5 * self.precision * self.mu[:, node_index] @ self.mu[:, node_index]
        ).reshape(
            -1,
        )

    @property
    def log_pi_ratio(self) -> np.ndarray:
        return self.log_pi - self.log_one_minus_pi


def init_boltzmann_machine(
    x: np.ndarray,
    binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
) -> BinaryLatentFactorModel:
    mu, sigma, pi = BinaryLatentFactorModel.calculate_maximisation_parameters(
        x, binary_latent_factor_approximation
    )
    return BoltzmannMachine(
        mu=mu,
        sigma=sigma,
        pi=pi,
    )
```

src/models/binary_latent_factor_models/boltzmann_machine.py

The Python code for message passing:

```python
from typing import List

import numpy as np

from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
    AbstractBinaryLatentFactorApproximation,
)
from src.models.binary_latent_factor_models.boltzmann_machine import BoltzmannMachine


class MessagePassing(AbstractBinaryLatentFactorApproximation):
    """
    eta_matrix:  of parameters eta_matrix[n, i, j]
                 off diagonals corresponds to \tilda{g}_{ij, \neg s_i}(s_j) for data point n
                 diagonals correspond to \tilda{f}_{i}(s_i)
                 (number_of_points, number_of_latent_variables, number_of_latent_variables)
    """

    def __init__(self, eta_matrix: np.ndarray):
        self.eta_matrix = eta_matrix

    @property
    def lambda_matrix(self) -> np.ndarray:
        lambda_matrix = 1 / (1 + np.exp(-self.xi.sum(axis=1)))
        lambda_matrix[lambda_matrix == 0] = 1e-10
        lambda_matrix[lambda_matrix == 1] = 1 - 1e-10
        return lambda_matrix

    @property
    def xi(self) -> np.ndarray:
        return np.log(np.divide(self.eta_matrix, 1 - self.eta_matrix))

    def aggregate_incoming_binary_factor_messages(
        self, node_index: int, excluded_node_index: int
    ) -> np.ndarray:
        # (number_of_points, )
        #  exclude message from excluded_node_index -> node_index
        return (
            np.sum(self.xi[:, :excluded_node_index, node_index], axis=1)
            + np.sum(self.xi[:, excluded_node_index + 1 :, node_index], axis=1)
        ).reshape(
            -1,
        )

    @staticmethod
    def calculate_eta(xi: np.ndarray) -> np.ndarray:
        eta = 1 / (1 + np.exp(-xi))
        eta[eta == 0] = 1e-10
        eta[eta == 1] = 1 - 1e-10
        return eta

    def variational_expectation_step(
        self, x: np.ndarray, binary_latent_factor_model: BoltzmannMachine
    ) -> List[float]:
        free_energy = [self.compute_free_energy(x, binary_latent_factor_model)]
        for i in range(self.k):
            xi_new_ii = self.calculate_singleton_message_update(
                boltzmann_machine=binary_latent_factor_model,
                x=x,
                i=i,
            )
            self.eta_matrix[:, i, i] = self.calculate_eta(xi_new_ii)
            free_energy.append(self.compute_free_energy(x, binary_latent_factor_model))

            for j in range(i):
                xi_new_ij = self.calculate_binary_message_update(
                    boltzmann_machine=binary_latent_factor_model,
                    x=x,
                    i=i,
                    j=j,
                )
                self.eta_matrix[:, i, j] = self.calculate_eta(xi_new_ij)
                xi_new_ji = self.calculate_binary_message_update(
                    boltzmann_machine=binary_latent_factor_model,
                    x=x,
                    i=j,
                    j=i,
                )
                self.eta_matrix[:, j, i] = self.calculate_eta(xi_new_ji)
                free_energy.append(
                    self.compute_free_energy(x, binary_latent_factor_model)
                )
        return free_energy

    def calculate_binary_message_update(
        self,
        x: np.ndarray,
        boltzmann_machine: BoltzmannMachine,
        i: int,
        j: int,
    ) -> float:
        eta_i_not_j = boltzmann_machine.b_index(
            x=x, node_index=i
        ) + self.aggregate_incoming_binary_factor_messages(
            node_index=i, excluded_node_index=j
        )
        w_i_j = boltzmann_machine.w_matrix_index(i, j)
        return np.log(1 + np.exp(w_i_j + eta_i_not_j)) - np.log(1 + np.exp(eta_i_not_j))

    @staticmethod
    def calculate_singleton_message_update(
        x: np.ndarray,
        boltzmann_machine: BoltzmannMachine,
        i: int,
    ) -> float:
        return boltzmann_machine.b_index(x=x, node_index=i)


def init_message_passing(k, n) -> MessagePassing:
    eta_matrix = np.random.random(size=(n, k, k))
    return MessagePassing(eta_matrix)
```

src/models/binary_latent_factor_approximations/message_passing_approximation.py

The rest of the Python code for question 6:

```python
import matplotlib.pyplot as plt
import numpy as np

from src.expectation_maximisation import learn_binary_factors
from src.models.binary_latent_factor_approximations.message_passing_approximation import (
    init_message_passing,
)
from src.models.binary_latent_factor_models.boltzmann_machine import (
    init_boltzmann_machine,
)


def run(x: np.ndarray, k: int, em_iterations: int, save_path: str) -> None:
    n = x.shape[0]
    message_passing = init_message_passing(k, n)
    boltzmann_machine = init_boltzmann_machine(x, message_passing)
    fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
    for i in range(k):
        ax[i].imshow(boltzmann_machine.mu[:, i].reshape(4, 4))
        ax[i].set_title(f"Latent Feature mu_{i}")
    fig.suptitle("Initial Features (Loopy BP)")
    plt.tight_layout()
    plt.savefig(save_path + "-init-latent-factors", bbox_inches="tight")
    plt.close()
    message_passing, boltzmann_machine, free_energy = learn_binary_factors(
        x=x,
        em_iterations=em_iterations,
        binary_latent_factor_model=boltzmann_machine,
        binary_latent_factor_approximation=message_passing,
    )
    fig, ax = plt.subplots(1, k, figsize=(k * 2, 2))
    for i in range(k):
        ax[i].imshow(boltzmann_machine.mu[:, i].reshape(4, 4))
        ax[i].set_title(f"Latent Feature mu_{i}")
    fig.suptitle("Learned Features (Loopy BP)")
    plt.tight_layout()
    plt.savefig(save_path + "-latent-factors", bbox_inches="tight")
    plt.close()

    plt.title("Free Energy (Loopy BP)")
    plt.xlabel("t (EM steps)")
    plt.ylabel("Free Energy")
    plt.plot(free_energy)
    plt.savefig(save_path + "-free-energy", bbox_inches="tight")
    plt.close()
```

src/solutions/q6.py

Appendix 1: abstract_binary_latent_factor_model.py

```python
from __future__ import annotations

from abc import ABC, abstractmethod
from typing import TYPE_CHECKING

import numpy as np

if TYPE_CHECKING:
    from src.models.binary_latent_factor_approximations.abstract_binary_latent_factor_approximation import (
        AbstractBinaryLatentFactorApproximation,
    )


class AbstractBinaryLatentFactorModel(ABC):
    @property
    @abstractmethod
    def mu(self) -> np.ndarray:
        pass

    @property
    @abstractmethod
    def variance(self) -> float:
        pass

    @property
    @abstractmethod
    def pi(self) -> np.ndarray:
        pass

    @abstractmethod
    def maximisation_step(
        self,
        x: np.ndarray,
        binary_latent_factor_approximation: AbstractBinaryLatentFactorApproximation,
    ) -> None:
        pass

    def mu_exclude(self, exclude_latent_index: int) -> np.ndarray:
        return np.concatenate(  # (number_of_dimensions, number_of_latent_variables-1)
            (self.mu[:, :exclude_latent_index], self.mu[:, exclude_latent_index + 1 :]),
            axis=1,
        )

    @property
    def log_pi(self) -> np.ndarray:
        return np.log(self.pi)

    @property
    def log_one_minus_pi(self) -> np.ndarray:
        return np.log(1 - self.pi)

    @property
    def precision(self) -> float:
        return 1 / self.variance

    @property
    def d(self) -> int:
        return self.mu.shape[0]

    @property
    def k(self) -> int:
        return self.mu.shape[1]
```

src/models/binary_latent_factor_models/abstract_binary_latent_factor_model.py

Appendix 2: abstract_binary_latent_factor_approximation.py

```python
from __future__ import annotations

from abc import ABC, abstractmethod
from typing import TYPE_CHECKING, List

if TYPE_CHECKING:
    from src.models.binary_latent_factor_models.binary_latent_factor_model import (
        AbstractBinaryLatentFactorModel,
    )

import numpy as np


class AbstractBinaryLatentFactorApproximation(ABC):
    @property
    @abstractmethod
    def lambda_matrix(self) -> np.ndarray:
        pass

    @abstractmethod
    def variational_expectation_step(
        self,
        x: np.ndarray,
        binary_latent_factor_model: AbstractBinaryLatentFactorModel,
    ) -> List[float]:
        pass

    @property
    def expectation_s(self):
        return self.lambda_matrix

    @property
    def expectation_ss(self):
        ess = self.lambda_matrix.T @ self.lambda_matrix
        np.fill_diagonal(ess, self.lambda_matrix.sum(axis=0))
        return ess

    @property
    def log_lambda_matrix(self) -> np.ndarray:
        return np.log(self.lambda_matrix)

    @property
    def log_one_minus_lambda_matrix(self) -> np.ndarray:
        return np.log(1 - self.lambda_matrix)

    @property
    def n(self) -> int:
        return self.lambda_matrix.shape[0]

    @property
    def k(self) -> int:
        return self.lambda_matrix.shape[1]

    def compute_free_energy(
        self,
        x: np.ndarray,
        binary_latent_factor_model: AbstractBinaryLatentFactorModel,
    ) -> float:
        """
        free energy associated with current EM parameters and data x

        :param x: data matrix (number_of_points, number_of_dimensions)
        :param binary_latent_factor_model: a binary_latent_factor_model
        :return: average free energy per data point
        """
        expectation_log_p_x_s_given_theta = (
            self._compute_expectation_log_p_x_s_given_theta(
                x, binary_latent_factor_model
            )
        )
        approximation_model_entropy = self._compute_approximation_model_entropy()
        return (
            expectation_log_p_x_s_given_theta + approximation_model_entropy
        ) / self.n

    def _compute_expectation_log_p_x_s_given_theta(
        self,
        x: np.ndarray,
        binary_latent_factor_model: AbstractBinaryLatentFactorModel,
    ) -> float:
        """
        The first term of the free energy, the expectation of log P(X,S|theta)

        :param x: data matrix (number_of_points, number_of_dimensions)
        :param binary_latent_factor_model: a binary_latent_factor_model
        :return: the expectation of log P(X,S|theta)
        """
        # (number_of_points, number_of_dimensions)
        mu_lambda = self.lambda_matrix @ binary_latent_factor_model.mu.T

        # (number_of_latent_variables, number_of_latent_variables)
        expectation_s_i_s_j_mu_i_mu_j = np.multiply(
            self.lambda_matrix.T @ self.lambda_matrix,
            binary_latent_factor_model.mu.T @ binary_latent_factor_model.mu,
        )

        expectation_log_p_x_given_s_theta = -(
            self.n * binary_latent_factor_model.d / 2
        ) * np.log(2 * np.pi * binary_latent_factor_model.variance) - (
            0.5 * binary_latent_factor_model.precision
        ) * (
            np.sum(np.multiply(x, x))
            - 2 * np.sum(np.multiply(x, mu_lambda))
            + np.sum(expectation_s_i_s_j_mu_i_mu_j)
            - np.trace(
                expectation_s_i_s_j_mu_i_mu_j
            )  # remove incorrect E[s_i s_i] = lambda_i * lambda_i
            + np.sum(  # add correct E[s_i s_i] = lambda_i
                self.lambda_matrix
                @ np.multiply(
                    binary_latent_factor_model.mu, binary_latent_factor_model.mu
                ).T
            )
        )
        expectation_log_p_s_given_theta = np.sum(
            np.multiply(
                self.lambda_matrix,
                binary_latent_factor_model.log_pi,
            )
            + np.multiply(
                1 - self.lambda_matrix,
                binary_latent_factor_model.log_one_minus_pi,
            )
        )
        return expectation_log_p_x_given_s_theta + expectation_log_p_s_given_theta

    def _compute_approximation_model_entropy(self) -> float:
        return -np.sum(
            np.multiply(
                self.lambda_matrix,
                self.log_lambda_matrix,
            )
            + np.multiply(
                1 - self.lambda_matrix,
                self.log_one_minus_lambda_matrix,
            )
        )
```

src/models/binary_latent_factor_approximations/abstract_binary_latent_factor_approximation.py

Appendix 3: main.py

```python
import os
from dataclasses import asdict

import jax
import jax.numpy as jnp
import numpy as np
import pandas as pd

from src.constants import CO2_FILE_PATH, DEFAULT_SEED, OUTPUTS_FOLDER
from src.generate_images import generate_images
from src.models.bayesian_linear_regression import LinearRegressionParameters
from src.models.gaussian_process_regression import GaussianProcessParameters
from src.models.kernels import CombinedKernel, CombinedKernelParameters
from src.solutions import q2, q3, q4, q6

jax.config.update("jax_enable_x64", True)

if __name__ == "__main__":
    np.random.seed(DEFAULT_SEED)

    if not os.path.exists(OUTPUTS_FOLDER):
        os.makedirs(OUTPUTS_FOLDER)

    # Question 2
    Q2_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q2")
    if not os.path.exists(Q2_OUTPUT_FOLDER):
        os.makedirs(Q2_OUTPUT_FOLDER)
    with open(CO2_FILE_PATH) as file:
        lines = [line.rstrip().split() for line in file]

    df_co2 = pd.DataFrame(
        np.array([line for line in lines if line[0] != "#"]).astype(float)
    )
    column_names = lines[max([i for i, line in enumerate(lines) if line[0] == "#"])][1:]
    df_co2.columns = column_names
    t = df_co2.decimal.values[:] - np.min(df_co2.decimal.values[:])
    y = df_co2.average.values[:].reshape(1, -1)

    sigma = 1
    mean = np.array([0, 360]).reshape(-1, 1)
    covariance = np.array(
        [
            [10**2, 0],
            [0, 100**2],
        ]
    )
    kernel = CombinedKernel()
    kernel_parameters = CombinedKernelParameters(
        log_theta=jnp.log(1),
        log_sigma=jnp.log(1),
        log_phi=jnp.log(1),
        log_eta=jnp.log(1),
        log_tau=jnp.log(1),
        log_zeta=jnp.log(1e-1),
    )

    prior_linear_regression_parameters = LinearRegressionParameters(
        mean=mean,
        covariance=covariance,
    )
    posterior_linear_regression_parameters = q2.a(
        t,
        y,
        sigma,
        prior_linear_regression_parameters,
        save_path=os.path.join(Q2_OUTPUT_FOLDER, "a"),
    )
    q2.b(
        t_year=df_co2.decimal.values[:],
        t=t,
        y=y,
        linear_regression_parameters=posterior_linear_regression_parameters,
        error_mean=0,
        error_variance=1,
        save_path=os.path.join(Q2_OUTPUT_FOLDER, "b"),
    )

    q2.c(
        kernel=kernel,
        kernel_parameters=kernel_parameters,
        log_theta_range=jnp.log(jnp.linspace(1e-2, 5, 5)),
        t=t[:50].reshape(-1, 1),
        number_of_samples=3,
        save_path=os.path.join(Q2_OUTPUT_FOLDER, "c"),
    )

    init_kernel_parameters = CombinedKernelParameters(
        log_theta=jnp.log(5),
        log_sigma=jnp.log(5),
        log_phi=jnp.log(10),
        log_eta=jnp.log(5),
        log_tau=jnp.log(1),
        log_zeta=jnp.log(2),
    )
    gaussian_process_parameters = GaussianProcessParameters(
        kernel=asdict(init_kernel_parameters),
        log_sigma=jnp.log(1),
    )
    years_to_predict = 14
    t_new = t[-1] + np.linspace(0, years_to_predict, years_to_predict * 12)
    t_test = np.concatenate((t, t_new))
    q2.f(
        t_train=t,
        y_train=y,
        t_test=t_test,
        min_year=np.min(df_co2.decimal.values[:]),
        prior_linear_regression_parameters=prior_linear_regression_parameters,
        linear_regression_sigma=sigma,
        kernel=kernel,
        gaussian_process_parameters=gaussian_process_parameters,
        learning_rate=1e-2,
        number_of_iterations=100,
        save_path=os.path.join(Q2_OUTPUT_FOLDER, "f"),
    )

    # Question 3
    Q3_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q3")
    if not os.path.exists(Q3_OUTPUT_FOLDER):
        os.makedirs(Q3_OUTPUT_FOLDER)
    number_of_images = 2000
    x = generate_images(n=number_of_images)
    k = 8
    em_iterations = 100
    e_maximum_steps = 50
    e_convergence_criterion = 0

    binary_latent_factor_model = q3.e_and_f(
        x=x,
        k=k,
        em_iterations=em_iterations,
        e_maximum_steps=e_maximum_steps,
        e_convergence_criterion=e_convergence_criterion,
        save_path=os.path.join(Q3_OUTPUT_FOLDER, "f"),
    )
    _ = q3.e_and_f(
        x=x,
        k=int(k * 1.5),
        em_iterations=em_iterations,
        e_maximum_steps=e_maximum_steps,
        e_convergence_criterion=e_convergence_criterion,
        save_path=os.path.join(Q3_OUTPUT_FOLDER, "f-larger-k"),
    )
    q3.g(
        x=x[:1, :],
        binary_latent_factor_model=binary_latent_factor_model,
        sigmas=[1, 2, 3],
        k=k,
        em_iterations=em_iterations,
        e_maximum_steps=e_maximum_steps,
        e_convergence_criterion=e_convergence_criterion,
        save_path=os.path.join(Q3_OUTPUT_FOLDER, "g"),
    )
    # Question 4
    Q4_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q4")
    if not os.path.exists(Q4_OUTPUT_FOLDER):
        os.makedirs(Q4_OUTPUT_FOLDER)
    max_k = 21
    q4.b(
        x=x,
        a_parameter=1,
        b_parameter=0,
        ks=np.arange(4, 22),
        max_k=max_k,
        em_iterations=em_iterations,
        e_maximum_steps=e_maximum_steps,
        e_convergence_criterion=e_convergence_criterion,
        save_path=os.path.join(Q4_OUTPUT_FOLDER, "b"),
    )
    q4.b(
        x=x,
        a_parameter=1,
        b_parameter=0,
        ks=np.arange(4, 13),
        max_k=max_k,
        em_iterations=em_iterations,
        e_maximum_steps=e_maximum_steps,
        e_convergence_criterion=e_convergence_criterion,
        save_path=os.path.join(Q4_OUTPUT_FOLDER, "b-1"),
    )
    q4.b(
        x=x,
        a_parameter=1,
        b_parameter=0,
        ks=np.arange(13, 22),
        max_k=max_k,
        em_iterations=em_iterations,
        e_maximum_steps=e_maximum_steps,
        e_convergence_criterion=e_convergence_criterion,
        save_path=os.path.join(Q4_OUTPUT_FOLDER, "b-2"),
    )
    # Question 6
    Q6_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q6")
    if not os.path.exists(Q6_OUTPUT_FOLDER):
        os.makedirs(Q6_OUTPUT_FOLDER)
    q6.run(x, k, em_iterations, save_path=os.path.join(Q6_OUTPUT_FOLDER, "all"))
```

main.py

Appendix 4: constants.py

```
import os

DATA_FOLDER = "data"

CO2_FILE_PATH = os.path.join(DATA_FOLDER, "co2.txt")
IMAGES_FILE_PATH = os.path.join(DATA_FOLDER, "images.jpg")

OUTPUTS_FOLDER = "outputs"

DEFAULT_SEED = 0

M1 = [0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0]

M2 = [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0]

M3 = [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

M4 = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]

M5 = [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0]

M6 = [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1]

M7 = [0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0]

M8 = [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]
```

src/constants.py

Appendix 5: generate_images.py

```python
import numpy as np

from src.constants import DEFAULT_SEED, M1, M2, M3, M4, M5, M6, M7, M8


def generate_images(n: int = 400, seed: int = DEFAULT_SEED, sigma: float = 0.1):
    """

    :param n: number of data points
    :param seed: random seed
    :param sigma: Gaussian noise
    :return:
    """
    d = 16  # dimensionality of the data
    np.random.seed(seed)

    # Define the basic shapes of the features
    number_of_features = 8  # number of features
    rr = (
        0.5 + np.random.rand(number_of_features, 1) * 0.5
    )  # weight of each feature between 0.5 and 1
    mut = np.array(
        [
            rr[0] * M1,
            rr[1] * M2,
            rr[2] * M3,
            rr[3] * M4,
            rr[4] * M5,
            rr[5] * M6,
            rr[6] * M7,
            rr[7] * M8,
        ]
    )
    s = (
        np.random.rand(n, number_of_features) < 0.3
    )  # each feature occurs with prob 0.3 independently

    # Generate Data - The Data is stored in Y

    return (
        np.dot(s, mut) + np.random.randn(n, d) * sigma
    )  # some Gaussian noise is added
```

src/generate_images.py

Appendix 6: MStep.py

```python
import numpy as np


def m_step(x, es, ess):
    """
    mu, sigma, pie = MStep(x,es,ess)

    Inputs:
    -----------------
            x: shape (n, d) data matrix
           es: shape (n, k) E_q[s]
          ess: shape (k, k) sum over data points of E_q[ss'] (n, k, k)
                           if E_q[ss'] is provided, the sum over n is done for you.

    Outputs:
    --------
          mu: shape (d, k) matrix of means in p(y|{s_i},mu,sigma)
       sigma: shape (,)     standard deviation in same
         pie: shape (1, k) vector of parameters specifying generative distribution for s
    """
    n, d = x.shape
    if es.shape[0] != n:
        raise TypeError('es must have the same number of rows as x')
    k = es.shape[1]
    if ess.shape == (n, k, k):
        ess = np.sum(ess, axis=0)
    if ess.shape != (k, k):
        raise TypeError('ess must be square and have the same number of columns as es')

    mu = np.dot(np.dot(np.linalg.inv(ess), es.T), x).T
    sigma = np.sqrt((np.trace(np.dot(x.T, x)) + np.trace(np.dot(np.dot(mu.T, mu), ess))
                     - 2 * np.trace(np.dot(np.dot(es.T, x), mu))) / (n * d))
    pie = np.mean(es, axis=0, keepdims=True)

    return mu, sigma, pie
```

demo_code/MStep.py