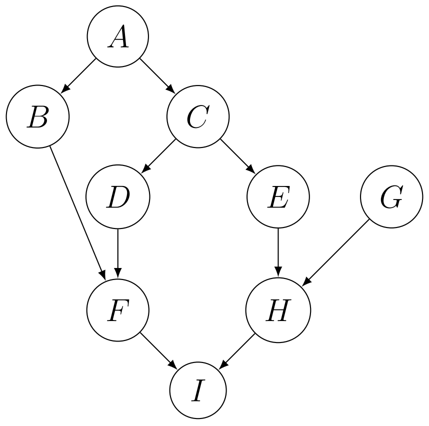
COMP0085 Summative Assignment

Jan 4, 2023

Question 1

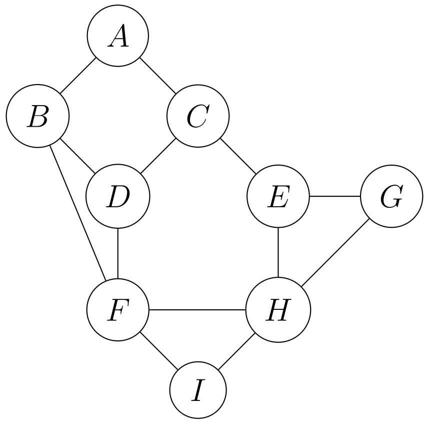
(a)

The directed acyclic graph:

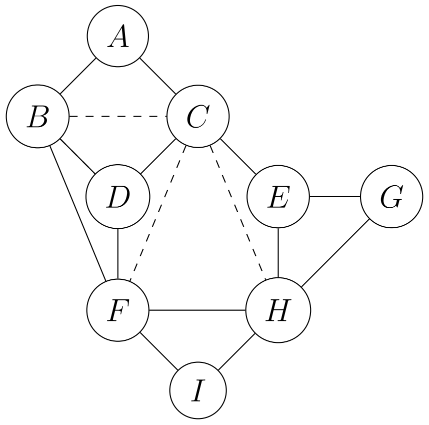


(b)

The moralised graph:

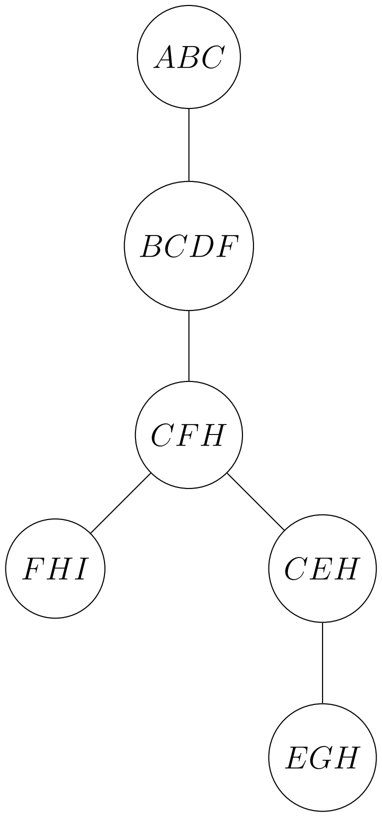


An effective triangulation:



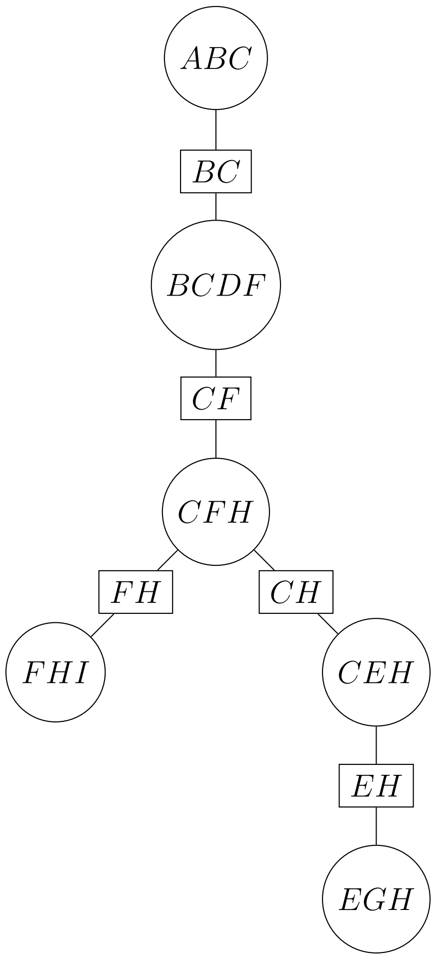
where the dashed lines are edges added to triangulate the moralised graph.

The resulting junction tree:



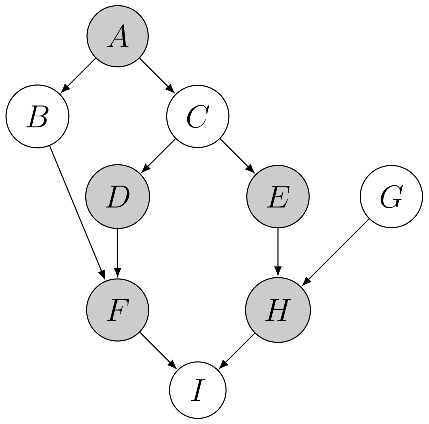
where the circular nodes are cliques.

The junction tree redrawn as a factor graph:



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

(c)



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

(d)

(e)

Question 2

(a)

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

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

Our distribution for $\mathbf{w}$:

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

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

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

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:

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

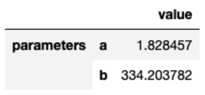
Thus we can identify the posterior covariance:

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

and the posterior mean:

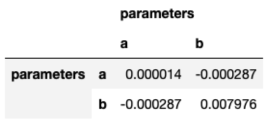
$$\bar{\mu}_w = \bar{\Sigma}_w\left(\frac{\mathbf{X}\mathbf{Y}^T}{\sigma^2} + \Sigma_{\mathbf{w}}^{-1}\mu_{\mathbf{w}}\right)$$

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:

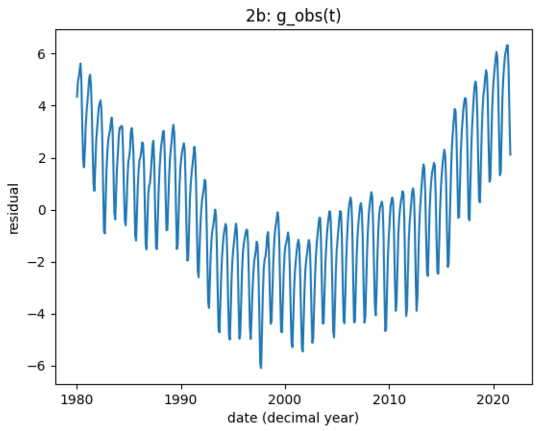


Figure 3: $g_{obs}(t)$

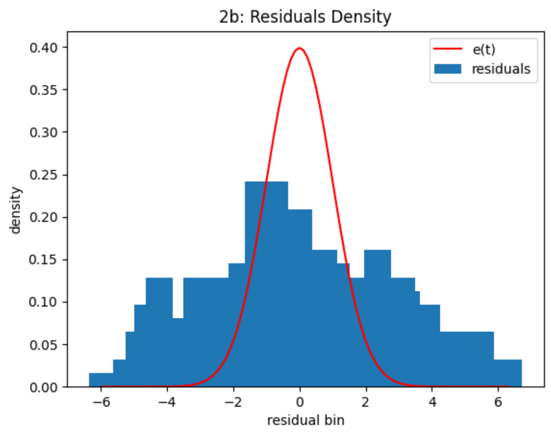


Figure 4: Density Estimation of Residuals vs $e(t) \sim \mathcal{N}(0, 1)$

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:

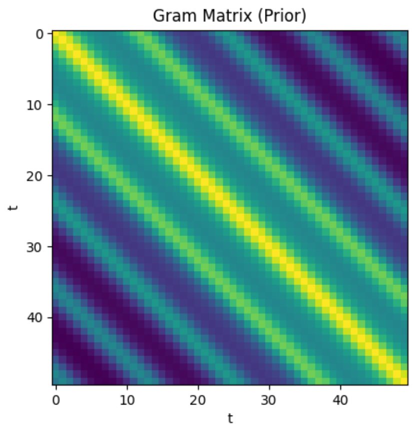


Figure 5: Covariance Matrix

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.

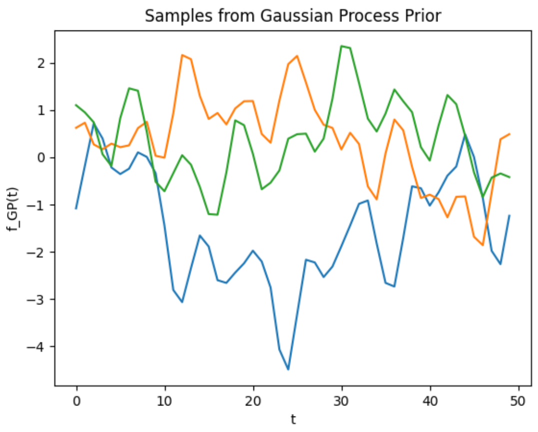


Figure 6: Samples from a zero mean GP with the provided covariance kernel

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

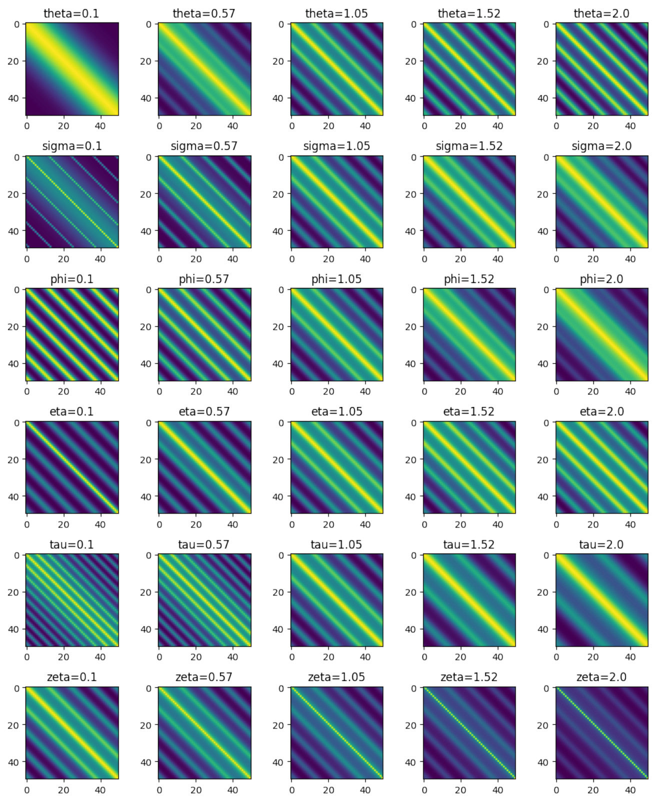


Figure 7: Covariances for different parameters

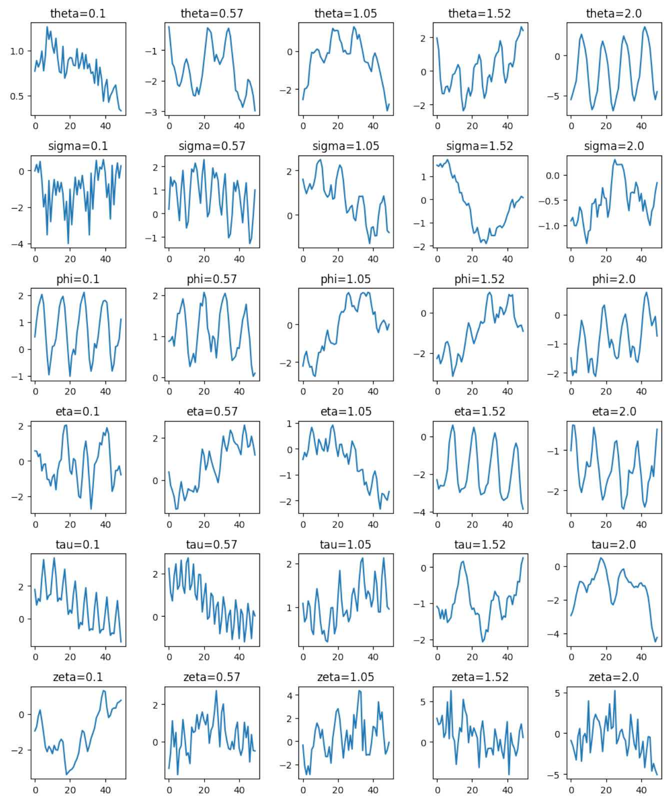


Figure 8: Samples for different parameters

θ: As θ increases, we see more pronounced periodic behavior in the sample function. The covariance matrix shows how increasing θ visually reveals the striped periodic component. This is expected because it is the parameter that adjusts the weight of $\exp\left(-\frac{2\sin^2(\pi(s-t)/\tau)}{\sigma^2}\right)$.

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

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

η:

τ:

ζ:

(e)

(f)

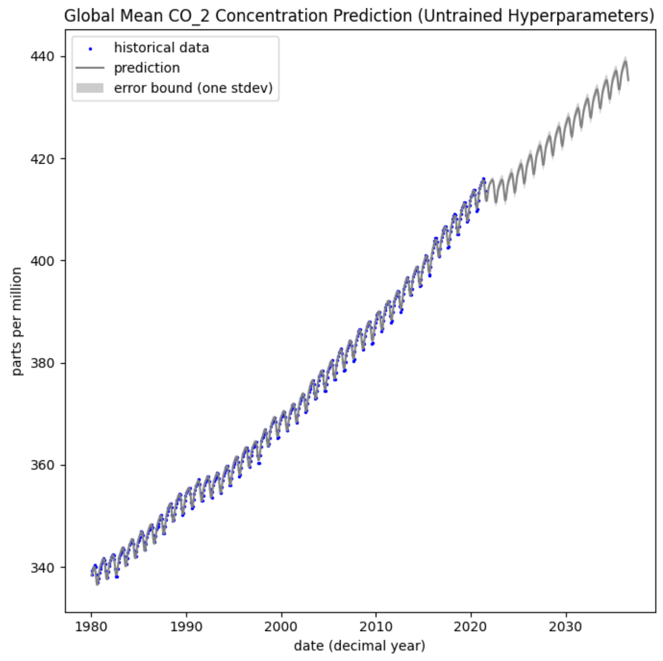


Figure 9: Without hyperparameter tuning

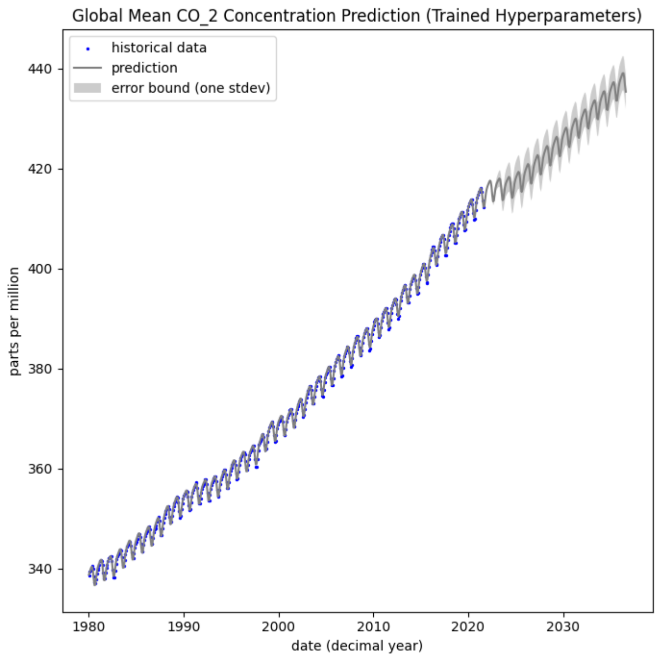


Figure 10: With hyperparameter tuning

(g)

The Python code for Bayesian Linear Regression:

```python
from dataclasses import dataclass

import numpy as np


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

    @property
    def precision(self):
        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):
        return self.sigma**2

    @property
    def precision(self):
        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)

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

    @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 optax
import dataframe_image as dfi
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import numpy as np
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,
    t,
    y,
    linear_regression_parameters: LinearRegressionParameters,
    error_mean,
    error_variance,
    save_path,
):
    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,
):
    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,
):
    # 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.diagonal(covariance_prediction),
        linear_prediction + mean_prediction + 1 * 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()

    # 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)
    )

    # 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.diagonal(covariance_prediction),
        linear_prediction + mean_prediction + 1 * 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:

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

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)

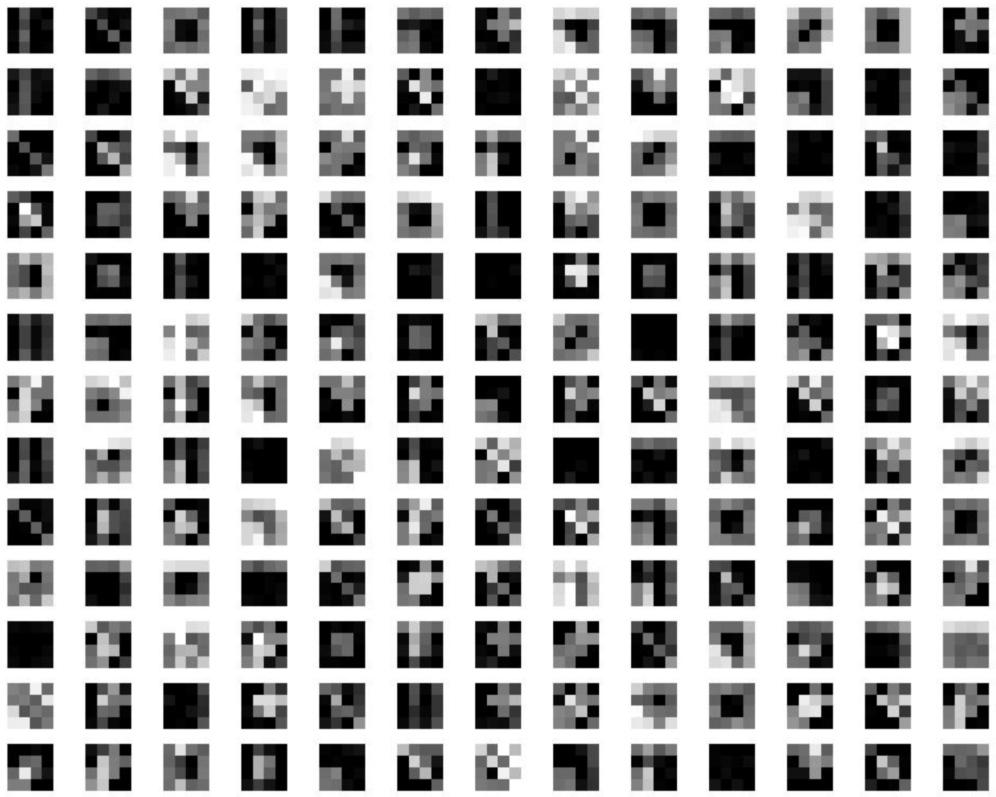


Figure 11: Images generated by randomly combined features with noise

(e)

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

(f)

(g)

The Python code for the binary latent factor model:

```python
from __future__ import annotations

import numpy as np

from demo_code.MStep import m_step
from typing import List
from abc import ABC, abstractmethod


class 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,
    ):
        self.mu = mu
        self.sigma = sigma
        self.pi = pi

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

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

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

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

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

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

    @property
    def k(self):
        return self.mu.shape[1]

    @staticmethod
    def calculate_maximisation_parameters(
        x: np.ndarray,
        binary_latent_factor_approximation: BinaryLatentFactorApproximation,
    ):

        expectation_s = binary_latent_factor_approximation.lambda_matrix
        expectation_ss = (
            binary_latent_factor_approximation.lambda_matrix.T
            @ binary_latent_factor_approximation.lambda_matrix
        )
        np.fill_diagonal(
            expectation_ss, binary_latent_factor_approximation.lambda_matrix.sum(axis=0)
        )
        return m_step(x, expectation_s, expectation_ss)

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

```python


class BinaryLatentFactorApproximation(ABC):
    @property
    @abstractmethod
    def lambda_matrix(self):
        pass

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

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

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

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

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

    def compute_free_energy(
        self,
        x: np.ndarray,
        binary_latent_factor_model: BinaryLatentFactorModel,
    ) -> 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: BinaryLatentFactorModel,
    ) -> 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,
            )
        )


def is_converge(free_energies, current_lambda_matrix, previous_lambda_matrix):
    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: BinaryLatentFactorModel,
    binary_latent_factor_approximation: BinaryLatentFactorApproximation,
):
    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/models/binary_latent_factor_model.py

The Python code for mean field learning:

```python
import numpy as np

from src.models.binary_latent_factor_model import (
    BinaryLatentFactorModel,
    BinaryLatentFactorApproximation,
)


class MeanFieldApproximation(BinaryLatentFactorApproximation):
    """
    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):
        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: BinaryLatentFactorModel,
        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: BinaryLatentFactorModel
    ):
        """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
        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/mean_field_learning.py

The rest of the Python code for question 3:

```python
import numpy as np
from src.models.mean_field_learning import (
    BinaryLatentFactorModel,
    init_mean_field_approximation,
)
from src.models.binary_latent_factor_model import (
    learn_binary_factors,
    init_binary_latent_factor_model,
    is_converge,
)
import matplotlib.pyplot as plt
from typing import List


def e_and_f(
    x: np.ndarray,
    k: int,
    em_iterations: int,
    e_maximum_steps: int,
    e_convergence_criterion: float,
    save_path: str,
):
    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
    )
    _, 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: BinaryLatentFactorModel,
    sigmas: List[float],
    k: int,
    em_iterations: int,
    e_maximum_steps: int,
    e_convergence_criterion: float,
    save_path: str,
):
    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):
            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 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(
            np.arange(len(free_energy) - 1),
            np.log(np.diff(np.array(free_energy))),
            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

We begin with the log joint:

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

where η is a collection of all hyperparameters.
We know:

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

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

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

We choose the conjugate priors:

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

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

a Gaussian prior on $\mathbf{A}$ and a product of inverse gamma distributions on Ψ where we assume Ψ is a diagonal matrix.
Combining, we have our expression:

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

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

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

where θ denotes the parameters π, $\mathbf{A}$, Ψ, η.
Substituting the relevant terms:

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

Simplifying:

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

By inspection, we can see:

$$q_s(\mathbf{s}) \propto \mathcal{N}(s|\mu^*_{\mathbf{s}}, \Sigma^*_{\mathbf{s}})$$

where

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

and

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

the E step updates.
For the Variational Bayes maximisation step, we set:

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

assuming the factorisation:

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

we can calculate each factor independently.
For $q_\pi(\pi)$:

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

Substituting the appropriate terms:

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

We see:

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

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

For $q_\Psi(\Psi)$:

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

Substituting the appropriate terms:

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

We see:

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

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

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

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

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

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

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

Question 5

(a)

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

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

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

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

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:

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

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

where:

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

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

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

and

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

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

We can write:

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

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)

Our message passing approximations:

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

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

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

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

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

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

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

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

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

(d)

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

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

1 Question 6

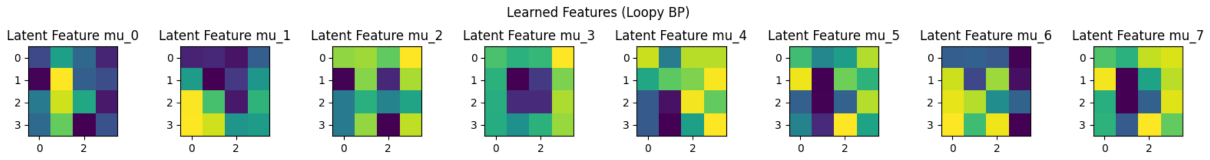


Figure 12: Latent factors learned with EP/Loopy-BP

The Python code for the Boltzmann machine:

```python
import numpy as np
from src.models.binary_latent_factor_model import (
    BinaryLatentFactorModel,
    BinaryLatentFactorApproximation,
)


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):
        # (number_of_latent_variables, number_of_latent_variables)
        return -self.precision * (self.mu.T @ self.mu)

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

    def b(self, x):
        """

        :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):
        return self.log_pi - self.log_one_minus_pi


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

src/models/boltzmann_machine.py

The Python code for message passing:

```python
import numpy as np
from src.models.binary_latent_factor_model import BinaryLatentFactorApproximation
from src.models.boltzmann_machine import BoltzmannMachine

from typing import List


class MessagePassing(BinaryLatentFactorApproximation):
    """
    eta_matrix: off diagonal matrix of parameters eta_matrix[n, i, j] corresponds to \tilda{g}_{ij, \neg s_i}(s_j
     ) for
                data point n
                (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):
        return 1 / (1 + np.exp(-self.xi.sum(axis=1)))

    @property
    def xi(self):
        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,
        )

    def set_xi(self, i, j, value):
        eta_values = 1 / (1 + np.exp(-value))
        eta_values[eta_values == 0] = 1e-10
        eta_values[eta_values == 1] = 1 - 1e-10
        self.eta_matrix[:, i, j] = eta_values

    def variational_expectation_step(
        self, x, binary_latent_factor_model: BoltzmannMachine
    ) -> List[float]:
        free_energy = [self.compute_free_energy(x, binary_latent_factor_model)]
        for i in range(self.k):
            for j in range(self.k):
                self.update_message(
                    binary_latent_factor_model,
                    x,
                    i,
                    j,
                )
            free_energy.append(self.compute_free_energy(x, binary_latent_factor_model))
        return free_energy

    def update_message(
        self,
        boltzmann_machine: BoltzmannMachine,
        x,
        start_node: int,
        end_node: int,
    ):
        if start_node != end_node:
            return self._update_binary_message(
                x, boltzmann_machine, start_node, end_node
            )
        else:
            return self._update_singleton_message(x, boltzmann_machine, start_node)

    def _update_binary_message(
        self,
        x,
        boltzmann_machine: BoltzmannMachine,
        i: int,
        j: int,
    ):
        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
        )
        eta_j_not_i = boltzmann_machine.b_index(
            x=x, node_index=j
        ) + self.aggregate_incoming_binary_factor_messages(
            node_index=j, excluded_node_index=i
        )
        w_i_j = boltzmann_machine.w_matrix_index(i, j)
        self.set_xi(
            i=i,
            j=j,
            value=np.log(1 + np.exp(w_i_j + eta_i_not_j))
            - np.log(1 + np.exp(eta_i_not_j)),
        )
        self.set_xi(
            i=j,
            j=i,
            value=np.log(1 + np.exp(w_i_j + eta_j_not_i))
            - np.log(1 + np.exp(eta_j_not_i)),
        )

    def _update_singleton_message(
        self,
        x,
        boltzmann_machine: BoltzmannMachine,
        i: int,
    ):
        b_i = boltzmann_machine.b_index(x=x, node_index=i)
        self.set_xi(i=i, j=i, value=b_i)


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

src/models/message_passing.py

The rest of the Python code for question 6:

```python
from src.generate_images import generate_images
import matplotlib.pyplot as plt
from src.models.binary_latent_factor_model import learn_binary_factors
from src.models.boltzmann_machine import init_boltzmann_machine
from src.models.message_passing import init_message_passing


def run(x, k, em_iterations, save_path):
    n = x.shape[0]
    message_passing = init_message_passing(k, n)
    boltzmann_machine = init_boltzmann_machine(x, message_passing)
    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: 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 2: main.py

```python
import os

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

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

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

if __name__ == "__main__":
    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-1, 2, 5)),
        t=t[:50].reshape(-1, 1),
        number_of_samples=3,
        save_path=os.path.join(Q2_OUTPUT_FOLDER, "c"),
    )

    gaussian_process_parameters = GaussianProcessParameters(
        kernel=asdict(kernel_parameters),
        log_sigma=jnp.log(1),
    )
    years_to_predict = 15
    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 = 1000
    x = generate_images(n=number_of_images)
    k = 8
    em_iterations = 1000
    e_maximum_steps = 200
    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.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 6
    Q6_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q6")
    if not os.path.exists(Q6_OUTPUT_FOLDER):
        os.makedirs(Q6_OUTPUT_FOLDER)
    em_iterations = 50
    q6.run(x, k, em_iterations, save_path=os.path.join(Q6_OUTPUT_FOLDER, "all"))
```

main.py