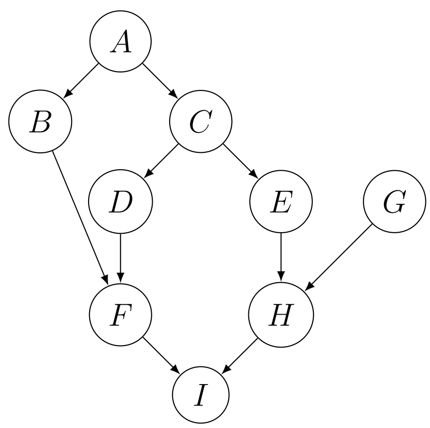
COMP0085 Summative Assignment

Jan 4, 2023

Question 1

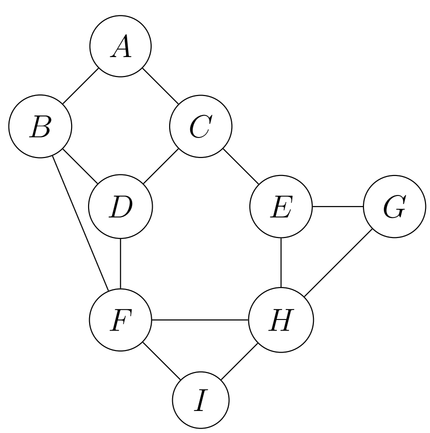
(a)

The directed acyclic graph:

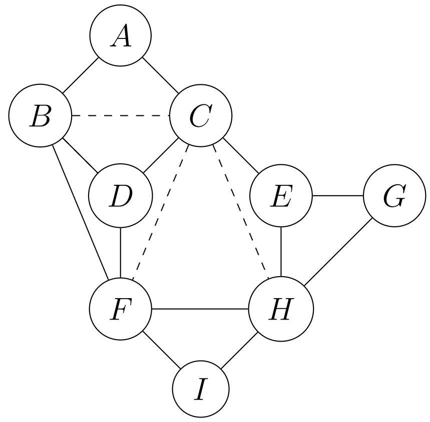


(b)

The moralised graph:

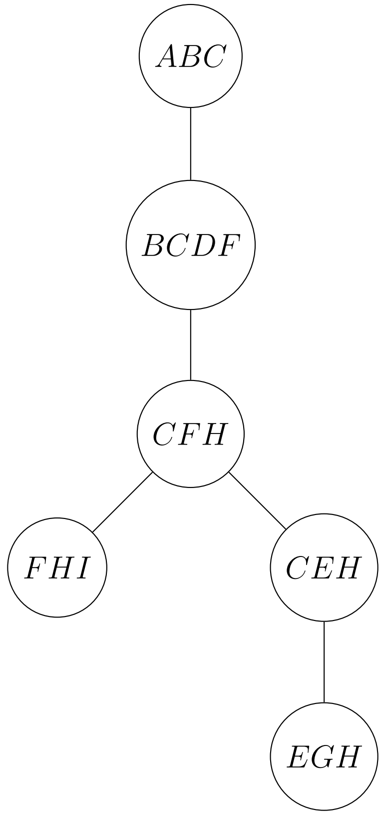


An effective triangulation:



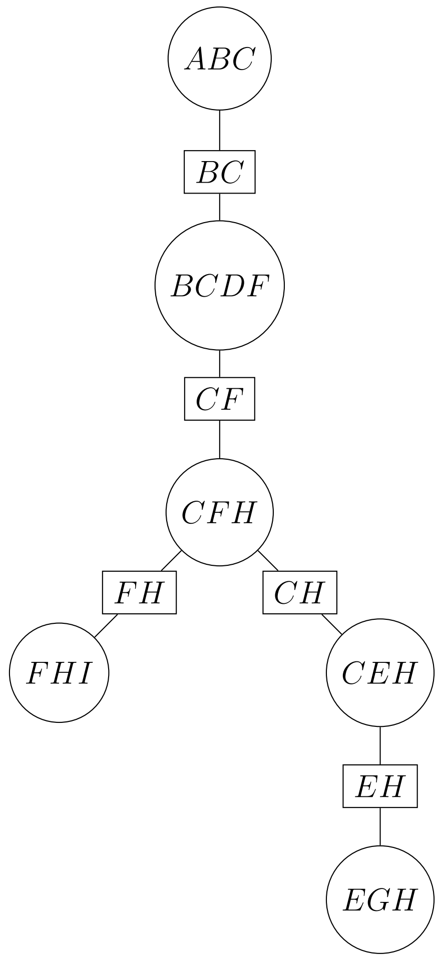
where the dashed lines are edges added to triangulate the moralised graph.

The resulting junction tree:



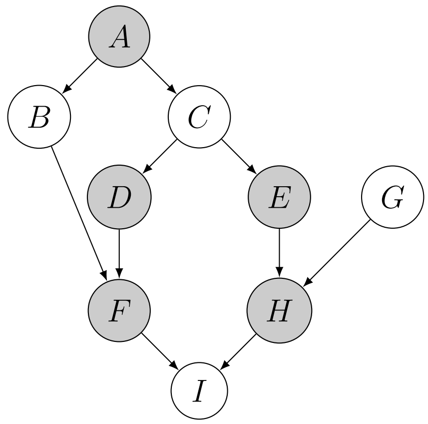
where the circular nodes are cliques.

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:

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

The Python code:

```python
import scipy
import numpy as np
from dataclasses import dataclass
import pandas as pd
import dataframe_image as dfi
import matplotlib.pyplot as plt

@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_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
    )


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_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, 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()
```

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:

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

The expectation distributed to the relevant terms:

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

Evaluating the expectations:

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

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:

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

Thus,

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

Substituting the appropriate values:

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

Combining, we have our free energy expression:

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

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:

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

Substituting the relevant terms:

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

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

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

Thus,

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

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

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

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

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

Solving for λ_i:

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

we have our partial update.

The Python code:

```python
from dataclasses import dataclass

import numpy as np

from demo_code.MStep import m_step


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

    lambda_matrix: np.ndarray

    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,
        )

    @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 init_mean_field_approximation(k: int, n: int) -> MeanFieldApproximation:
    return MeanFieldApproximation(
        lambda_matrix=np.random.random(size=(n, k)),
    )


@dataclass
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)
    """

    mu: np.ndarray
    sigma: float
    pi: np.ndarray

    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]


def init_binary_latent_factor_model(
    x: np.ndarray,
```

```python
    mean_field_approximation: MeanFieldApproximation,
) -> BinaryLatentFactorModel:
    return maximisation_step(x, mean_field_approximation)


def _compute_expectation_log_p_x_s_given_theta(
    x: np.ndarray,
    binary_latent_factor_model: BinaryLatentFactorModel,
    mean_field_approximation: MeanFieldApproximation,
) -> 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
    :param mean_field_approximation: a mean_field_approximation
    :return: the expectation of log P(X,S|theta)
    """
    # (number_of_points, number_of_dimensions)
    mu_lambda = mean_field_approximation.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(
        mean_field_approximation.lambda_matrix.T
        @ mean_field_approximation.lambda_matrix,
        binary_latent_factor_model.mu.T @ binary_latent_factor_model.mu,
    )

    expectation_log_p_x_given_s_theta = (
        (-binary_latent_factor_model.d / 2) * np.log(2 * np.pi)
        - np.log(binary_latent_factor_model.sigma)
        - (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
                mean_field_approximation.lambda_matrix
                @ np.multiply(
                    binary_latent_factor_model.mu, binary_latent_factor_model.mu
                ).T
            )
        )
    )
    expectation_log_p_s_given_theta = np.sum(
        mean_field_approximation.lambda_matrix @ binary_latent_factor_model.log_pi.T
        + (1 - mean_field_approximation.lambda_matrix)
        @ binary_latent_factor_model.log_one_minus_pi.T
    )
    return expectation_log_p_x_given_s_theta + expectation_log_p_s_given_theta


def _compute_mean_field_approximation_entropy(
    mean_field_approximation: MeanFieldApproximation,
) -> float:
    return -np.sum(
        np.multiply(
            mean_field_approximation.lambda_matrix,
            mean_field_approximation.log_lambda_matrix,
        )
        + np.multiply(
            1 - mean_field_approximation.lambda_matrix,
            mean_field_approximation.log_one_minus_lambda_matrix,
        )
    )


def compute_free_energy(
    x: np.ndarray,
    binary_latent_factor_model: BinaryLatentFactorModel,
    mean_field_approximation: MeanFieldApproximation,
) -> 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
    :param mean_field_approximation: a mean_field_approximation
    :return: free energy
    """
    expectation_log_p_x_s_given_theta = _compute_expectation_log_p_x_s_given_theta(
        x, binary_latent_factor_model, mean_field_approximation
    )
    mean_field_approximation_entropy = _compute_mean_field_approximation_entropy(
        mean_field_approximation
    )
    return expectation_log_p_x_s_given_theta + mean_field_approximation_entropy


def partial_expectation_step(
    x: np.ndarray,
    binary_latent_factor_model: BinaryLatentFactorModel,
    mean_field_approximation: MeanFieldApproximation,
```

```
    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 mean_field_approximation: a mean_field_approximation
    :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 = mean_field_approximation.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(
    x: np.ndarray,
    binary_latent_factor_model: BinaryLatentFactorModel,
    mean_field_approximation: MeanFieldApproximation,
    max_steps: int,
    convergence_criterion: float,
) -> MeanFieldApproximation:
    """Variational E step

    :param x: data matrix (number_of_points, number_of_dimensions)
    :param binary_latent_factor_model: a binary_latent_factor_model
    :param mean_field_approximation: a mean_field_approximation
    :param max_steps: maximum number of steps of fixed point equations
    :param convergence_criterion: early stopping if change in free energy <  convergence_criterion
    :return: mean field approximation
    """
    previous_free_energy = compute_free_energy(
        x, binary_latent_factor_model, mean_field_approximation
    )
    for i in range(max_steps):
        for latent_factor in range(binary_latent_factor_model.k):
            mean_field_approximation.lambda_matrix[
                :, latent_factor
            ] = partial_expectation_step(
                x, binary_latent_factor_model, mean_field_approximation, latent_factor
            )
        free_energy = compute_free_energy(
            x, binary_latent_factor_model, mean_field_approximation
        )
        if free_energy - previous_free_energy <= convergence_criterion:
            break
        previous_free_energy = free_energy
    return mean_field_approximation


def maximisation_step(
    x: np.ndarray,
    mean_field_approximation: MeanFieldApproximation,
) -> BinaryLatentFactorModel:
    expectation_s = mean_field_approximation.lambda_matrix
    expectation_ss = (
        mean_field_approximation.lambda_matrix.T
        @ mean_field_approximation.lambda_matrix
    )
    np.fill_diagonal(expectation_ss, mean_field_approximation.lambda_matrix.sum(axis=0))
    mu, sigma, pi = m_step(x, expectation_s, expectation_ss)
    return BinaryLatentFactorModel(
```

```python
        mu=mu,
        sigma=sigma,
        pi=pi,
    )


def learn_binary_factors(
    x: np.ndarray,
    k: int,
    em_maximum_iterations: int,
    e_maximum_steps: int,
    e_convergence_criterion: float,
):
    n = x.shape[0]
    mean_field_approximation = init_mean_field_approximation(k, n)
    binary_latent_factor_model = init_binary_latent_factor_model(
        x, mean_field_approximation
    )

    for _ in range(em_maximum_iterations):
        mean_field_approximation = variational_expectation_step(
            x=x,
            binary_latent_factor_model=binary_latent_factor_model,
            mean_field_approximation=mean_field_approximation,
            max_steps=e_maximum_steps,
            convergence_criterion=e_convergence_criterion,
        )
        binary_latent_factor_model = maximisation_step(
            x=x,
            mean_field_approximation=mean_field_approximation,
        )
    return mean_field_approximation, binary_latent_factor_model
```

src/solutions/q3.py

Question 4

Question 5

(a)

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

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

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

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

Expanding:

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

Collecting terms pertaining to s_i:

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

where C are all other terms without s_i.
Knowing that $s_i^2 = s_i$:

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

Thus:

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

where the factors are defined:

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

and

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

as required. Note that C can simply be absorbed into any one of these factors.
The Boltzmann Machine can be defined:

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

where $s_i \in \{0, 1\}$, the same as our source variables.
From our factorisation, we can see that $p(\mathbf{s}, \mathbf{x})$ is a Boltzmann Machine with:

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

and

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

and

$$\log Z = -C$$

(b)

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

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

Thus,

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

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

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

Thus,

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

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

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

Thus, we can derive cavity distributions:

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

and

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

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

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

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

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

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

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

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

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

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

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

the mean with respect to s_i and s_j respectively.

We can write:

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

Only including terms with s_i and s_j:

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

Moreover:

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

Only including terms with s_i and s_j:

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

Appendix 1: constants.py

```python
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 pandas as pd
import numpy as np
from src.constants import CO2_FILE_PATH, IMAGES_FILE_PATH, OUTPUTS_FOLDER
from src.solutions import q2, q3, q4, q5, q6
from src.solutions.q2 import LinearRegressionParameters
from src.generate_images import generate_images


if __name__ == "__main__":
    if not os.path.exists(OUTPUTS_FOLDER):
        os.makedirs(OUTPUTS_FOLDER)

    with open(CO2_FILE_PATH) as file:
        lines = [line.rstrip().split() for line in file]

    # Question 2
    Q2_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q2")
    if not os.path.exists(Q2_OUTPUT_FOLDER):
        os.makedirs(Q2_OUTPUT_FOLDER)
    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],
        ]
    )

    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")
    )

    # # Question 3
    # Q3_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q3")
    # if not os.path.exists(Q3_OUTPUT_FOLDER):
    #     os.makedirs(Q3_OUTPUT_FOLDER)
    #
    # q3.learn_binary_factors(
    #     x=generate_images(),
    #     k=8,
    #     em_maximum_iterations=100,
    #     e_maximum_steps=100,
    #     e_convergence_criterion=0,
    # )
```

main.py