Sheet 1

Group 20

November 30, 2024

1 Introduction Probability Theory

Bishop 2.8 Given random variables X and Y, where $\mathbb{E}_{X|Y}[X]$ is what Bishop refers to as $\mathbb{E}_X[X|Y]$ and $\mathbb{V} = \text{var}$,

(a)

$$\begin{aligned}
\mathbb{E}_Y[\mathbb{E}_{X|Y}[X]] &= \int_y \left(\int_x (x)p(X=x|Y=y)dx \right) p(Y=y)dy \\
&= \int_y \left(\int_x (x)p(X=x|Y=y)p(Y=y)dx \right) dy \\
&= \int_y \int_x (x)p(X=x, Y=y)dxdy \\
&= \int_x \int_y (x)p(Y=y, X=x)dydx \\
&= \int_x xp(X=x)dx \\
&= \mathbb{E}_X[X]
\end{aligned}$$

(b) Suppose Z = X^2, from above we know
$\mathbb{E}_Y[\mathbb{E}_{X|Y}[X^2]] = \mathbb{E}_Y[\mathbb{E}_{Z|Y}[Z]] = \mathbb{E}_Z[Z] = \mathbb{E}_X[X^2]$, so:

$$\begin{aligned}
\mathbb{E}_Y[\mathbb{V}_{X|Y}[X]] + \mathbb{V}_Y[\mathbb{E}_{X|Y}[X]] &= \mathbb{E}_Y\left[\mathbb{E}_{X|Y}[X^2] - \mathbb{E}_{X|Y}[X]^2\right] + \mathbb{E}_Y\left[\mathbb{E}_{X|Y}[X]^2\right] - \mathbb{E}_Y\left[\mathbb{E}_{X|Y}[X]\right]^2 \\
&= \mathbb{E}_Y\left[\mathbb{E}_{X|Y}[X^2]\right] - \mathbb{E}_Y\left[\mathbb{E}_{X|Y}[X]^2\right] + \mathbb{E}_Y\left[\mathbb{E}_{X|Y}[X]^2\right] - \mathbb{E}_Y\left[\mathbb{E}_{X|Y}[X]\right]^2 \\
&= \mathbb{E}_X[X^2] - \mathbb{E}_X[X]^2 \\
&= \mathbb{V}_X[X]
\end{aligned}$$

Bishop 8.9 Suppose we have a model similar to Figure 1, if we define A to be node x, C to be the Markov blanket of x and B to be all other nodes, we can see that A is independent of B when conditioned on C, as any path from a node in B to A must either:

(a) Be a parent of parent of x, meaning it is blocked by the head-tail link in the parent of x (C_1)

(b) Be a different child of a parent of x, meaning it is blocked by the tail-tail link in the parent of x (C_1)

(c) Be a parent of a co-parent of x, meaning it is blocked by the head-tail link in the co-parent of x (C_2)

(d) Be a child of a co-parent of x that is not a child of x, meaning it is blocked by the tail-tail link in the co-parent of x (C_2)

(e) Be a child of a child of x, meaning it is blocked by the head-tail link in the child of x (C_3)

We can see that any other path would be invalid, as a path:

(x) Directly from a node to x would imply that this node is a parent, and therefore must be part of C

(y) Directly from x to a node would imply that this node is a child, and therefore must be part of C

(z) Directly from a node to a child of x would imply that this node is a co-parent, and therefore must be part of C

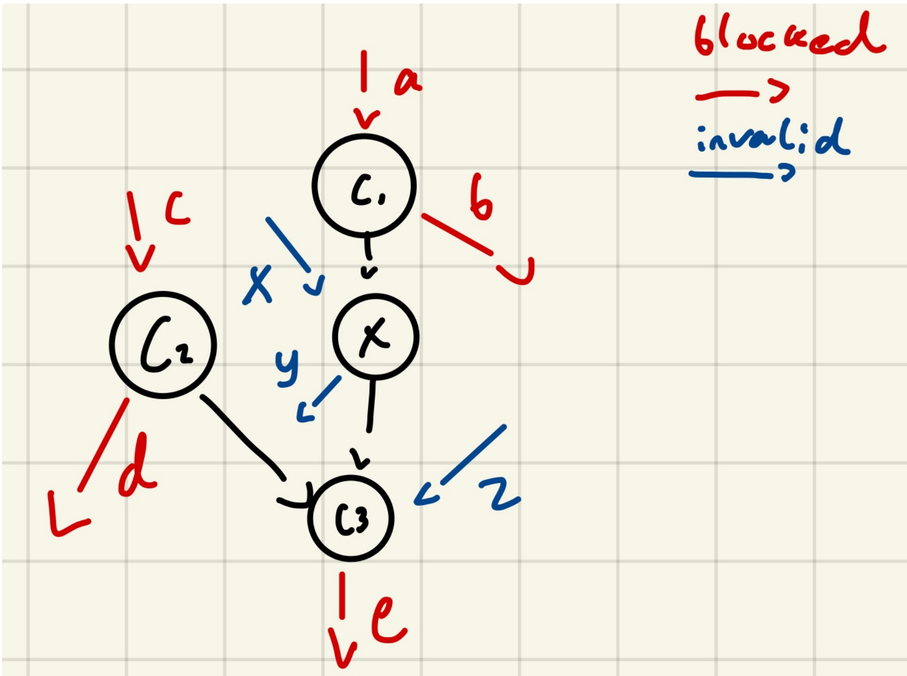


Figure 1: Small illustration of Markov blanket independence

Bishop 8.11 See equations and Figure 2

$$
\begin{aligned}
p(F=0|D=0) &= \frac{p(F=0,D=0)}{p(D=0)} \\
&= \frac{\sum_b \sum_g p(B=b,F=0,G=g,D=0)}{\sum_b \sum_f \sum_g p(B=b,F=f,G=g,D=0)} \\
&= \frac{\sum_b \sum_g p(B=b)p(F=0)p(G=g|B=b,F=0)p(D=0|G=g)}{\sum_b \sum_f \sum_g p(B=b)p(F=f)p(G=g|B=b,F=f)p(D=0|G=g)} \\
&\approx 0.2125 \\
p(F=0|D=0,B=0) &= \frac{p(F=0,D=0,B=0)}{p(D=0,B=0)} \\
&= \frac{\sum_g p(B=0,F=0,G=g,D=0)}{\sum_f \sum_g p(B=0,F=f,G=g,D=0)} \\
&= \frac{\sum_g p(B=0)p(F=0)p(G=g|B=0,F=0)p(D=0|G=g)}{\sum_f \sum_g p(B=0)p(F=f)p(G=g|B=0,F=f)p(D=0|G=g)} \\
&\approx 0.1096
\end{aligned}
$$

```
prob_b = np.array([0.1, 0.9])
prob_f = np.array([0.1, 0.9])
prob_g_given_bf = np.array([[[0.9, 0.1], [0.8, 0.2]], [[0.8, 0.2], [0.2, 0.8]]])
prob_d_given_g = np.array([[0.9, 0.1], [0.1, 0.9]])

prob_bfgd = (
    1  #            [B, F, G, D]
    * prob_b.reshape(2, 1, 1, 1)
    * prob_f.reshape(1, 2, 1, 1)
    * prob_g_given_bf.reshape(2, 2, 2, 1)
    * prob_d_given_g.reshape(1, 1, 2, 2)
)
✓ [14] < 10 ms
```

```
prob_fd = prob_bfgd.sum((0, 2), keepdims=True)
prob_f_given_d = prob_fd / prob_fd.sum(1, keepdims=True)
print(f"P(F=0 | D=0) = {prob_f_given_d[:, 0, :, 0].flat[0]:.2%}")
✓ [15] < 10 ms

P(F=0 | D=0) = 21.25%
```

```
prob_bfd = prob_bfgd.sum(2, keepdims=True)
prob_f_given_bd = prob_bfd / prob_bfd.sum(1, keepdims=True)
print(f"P(F=0 | B=0, D=0) = {prob_f_given_bd[0, 0, :, 0].flat[0]:.2%}")
✓ [16] < 10 ms

P(F=0 | B=0, D=0) = 10.96%
```

Figure 2: Calculation of probability values for 8.11

W1 Programming See Figure 3 for the graphs, see Sec. 3 (end of document) for the code.

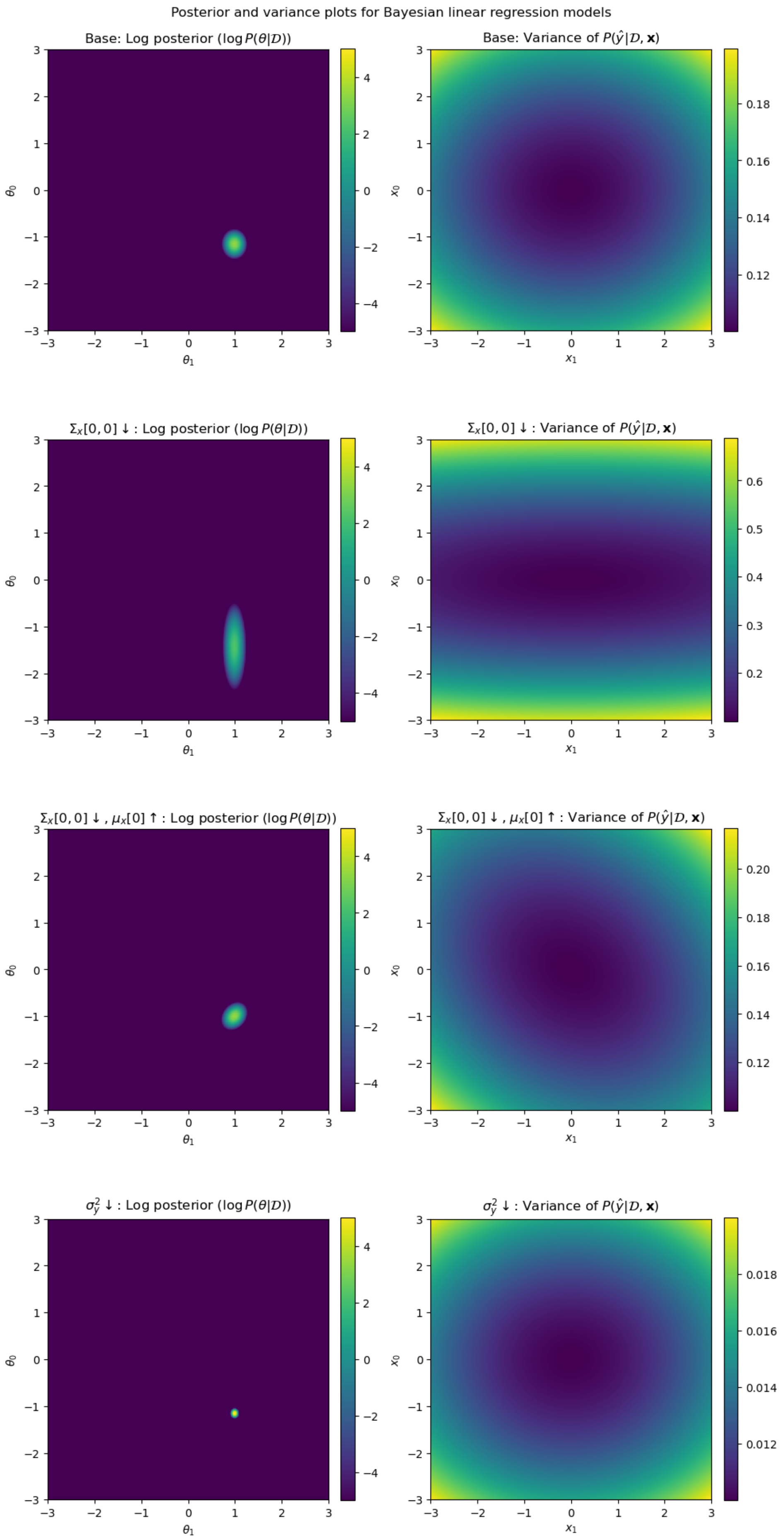


Figure 3: Parameter posterior and posterior-predictive variance plots for Bayesian linear regression using generated datasets. The first row is from the base dataset, the second row has variance 0.1 for x_0, the third row has variance 0.1 and mean 1 for x_0, while the final row has variance 0.01 for the target distribution. Note that, while the left column has identical hue scales for the heatmaps, the right column does not.

We can see that, compared to the base distribution, reducing the variance of x_0 makes the variance for θ_0 increase significantly. This somewhat counter-intuitive result comes from the fact that the distributions are zero-mean, meaning that if a component has low variance, then it will have little impact on the target variable: if all the elements of x_0 are ≈ 0, the value for θ_0 has a lower impact on y, making it harder to estimate. If we use the same lowered variance with a nonzero mean, we see that the parameter posterior variance shrinks significantly (as x_0 now always impacts y), and that the posterior becomes diagonal (reflecting that, with the non-zero means, it is possible to trade one variable off for another).

For the posterior predictive variance plots, we see that compared to the base distribution, the version with lower x_0 variance shows higher overall variance, as well as a clear increase in variance when $|x|$ increases. This is because the direction of θ_0 is uncertain, meaning that having larger values of x in this direction will make our prediction less certain. Increasing the mean of x_0 again lowers the variance, while also changing the direction of increasing predictive posterior variance to align with the direction of parameter posterior uncertainty.

Finally, decreasing σ_y^2 does not change the direction of uncertainty in either of the two plots, but significantly reduces the overall variance: the primary source of uncertainty in the base version was not in the estimation of θ but due to the randomness in y, so reducing this randomness greatly reduces the uncertainty of the overall system.

2 Mixture Models and PPCA

Bishop 9.10 If your component distribution(s) allow for tractable inference of $p(x_b|x_a, k)$ and $p(x_a|k')$, then

$$\begin{aligned} p(x_b|x_a) &= \sum_{k=1}^{K} p(x_b, k|x_a) \\ &= \sum_{k=1}^{K} p(x_b|x_a, k)p(k|x_a) \\ &= \sum_{k=1}^{K} p(x_b|x_a, k)\frac{p(x_a|k)p(k)}{p(x_a)} \\ &= \sum_{k=1}^{K} \pi_k \frac{p(x_a|k)}{p(x_a)} p(x_b|x_a, k) \\ &= \sum_{k=1}^{K} \pi_k \frac{p(x_a|k)}{\sum_{k'} \pi_{k'} p(x_a|k')} p(x_b|x_a, k) \end{aligned}$$

would give you a mixture distribution with coefficients $\pi_k^{cond} = \pi_k \frac{p(x_a|k)}{\sum_{k'} \pi_{k'} p(x_a|k')}$ and component densities $C_{k,x_a}(x_b) = p(x_b|x_a, k)$.

Bishop 10.4 $\mathbb{E}$ is expectation, $\mathbb{V}$ is variance, H is entropy, Σ is symmetric positive definite:

$$
\begin{aligned}
\text{KL}[q||p] &= -\int_x p(x)\ln\left(\frac{q(x)}{p(x)}\right)dx \\
&= -\int_x p(x)\ln\mathcal{N}(x;\mu,\Sigma)dx - -\int_x p(x)\ln p(x) \\
&= -\mathbb{E}_{x\sim p(x)}\left[-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu) - \ln\left(\sqrt{\det 2\pi\Sigma}\right)\right] - H_{x\sim p(x)}[x] \\
&= \frac{1}{2}\mathbb{E}_{x\sim p(x)}\left[(x-\mu)^T\Sigma^{-1}(x-\mu)\right] + \mathbb{E}_{x\sim p(x)}\left[\ln\left(\sqrt{\det 2\pi\Sigma}\right)\right] - H_{x\sim p(x)}[x] \\
&= \frac{1}{2}\mathbb{E}_{x\sim p(x)}\left[(x-\mu)^T\Sigma^{-1}(x-\mu)\right] + \ln\left(\sqrt{\det 2\pi\Sigma}\right) - H_{x\sim p(x)}[x] \\
\nabla_\mu\text{KL}[q||p] &= \nabla_\mu\left(\frac{1}{2}\mathbb{E}_{x\sim p(x)}\left[(x-\mu)^T\Sigma^{-1}(x-\mu)\right] + \ln\left(\sqrt{\det 2\pi\Sigma}\right) - H_{x\sim p(x)}[x]\right) \\
&= \frac{1}{2}\nabla_\mu\mathbb{E}_{x\sim p(x)}\left[(x-\mu)^T\Sigma^{-1}(x-\mu)\right] \\
&= \frac{1}{2}\mathbb{E}_{x\sim p(x)}\left[\nabla_\mu\left((x-\mu)^T\Sigma^{-1}(x-\mu)\right)\right] \\
&= \frac{1}{2}\mathbb{E}_{x\sim p(x)}\left[-2\Sigma^{-1}(x-\mu)\right] && (86) \\
&= -\Sigma^{-1}(\mathbb{E}_{x\sim p(x)}[x]-\mu) \\
&\text{Suppose } \nabla_\mu\text{KL}[q||p]=0\text{, then} \\
0 &= -\Sigma^{-1}(\mathbb{E}_{x\sim p(x)}[x]-\mu) \\
0 &= \mathbb{E}_{x\sim p(x)}[x]-\mu \\
\mu &= \mathbb{E}_{x\sim p(x)}[x] \\
\nabla_\Sigma\text{KL}[q||p] &= \nabla_\Sigma\left(\frac{1}{2}\mathbb{E}_{x\sim p(x)}\left[(x-\mu)^T\Sigma^{-1}(x-\mu)\right] + \ln\left(\sqrt{\det 2\pi\Sigma}\right) - H_{x\sim p(x)}[x]\right) \\
&= \frac{1}{2}\nabla_\Sigma\mathbb{E}_{x\sim p(x)}\left[(x-\mu)^T\Sigma^{-1}(x-\mu)\right] + \nabla_\Sigma\ln\left(\sqrt{\det 2\pi\Sigma}\right) \\
&= \frac{1}{2}\mathbb{E}_{x\sim p(x)}\left[\nabla_\Sigma\left((x-\mu)^T\Sigma^{-1}(x-\mu)\right)\right] + \frac{1}{2}\nabla_\Sigma\ln(\det\Sigma) + \nabla_\Sigma\, const \\
&= -\frac{1}{2}\mathbb{E}_{x\sim p(x)}\left[\Sigma^{-1}(x-\mu)(x-\mu)^T\Sigma^{-1}\right] + \frac{1}{2}\Sigma^{-1} && (61,\ 57) \\
&\text{Suppose } \nabla_\Sigma\text{KL}[q||p]=0\text{, then} \\
0 &= -\frac{1}{2}\mathbb{E}_{x\sim p(x)}\left[\Sigma^{-1}(x-\mu)(x-\mu)^T\Sigma^{-1}\right] + \frac{1}{2}\Sigma^{-1} \\
\Sigma^{-1} &= \Sigma^{-1}\mathbb{E}_{x\sim p(x)}\left[(x-\mu)(x-\mu)^T\right]\Sigma^{-1} \\
\Sigma\Sigma^{-1}\Sigma &= \Sigma\Sigma^{-1}\mathbb{V}_{x\sim p(x)}[x]\,\Sigma^{-1}\Sigma \\
\Sigma &= \mathbb{V}_{x\sim p(x)}[x] = \text{Cov}_{x\sim p(x)}[x,x]
\end{aligned}
$$

So, at the stationary point (the minimum of the KL divergence), the mean is the sample mean, and the covariance is the sample covariance.

W2 Programming See Figure 4 for samples from the Gaussian mixture model, and Figure 5 for an approximation of the conditional distribution for the same mixture. See Sec. 4 for the code that generated these figures.

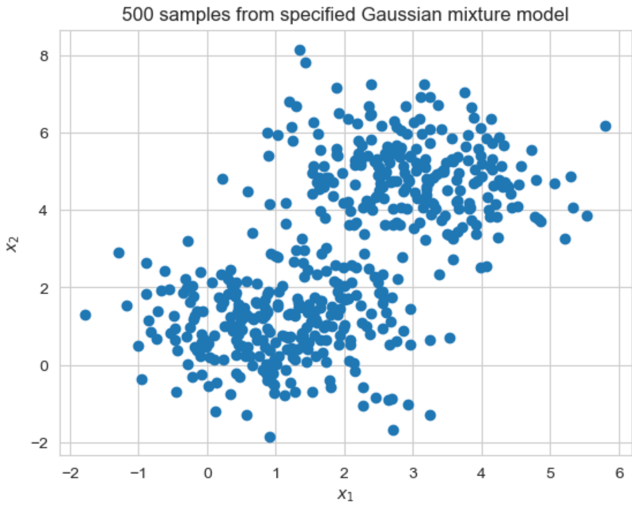


Figure 4: Samples for Gaussian mixture model of two components, with identity covariance and means (1, 1) and (3, 5).

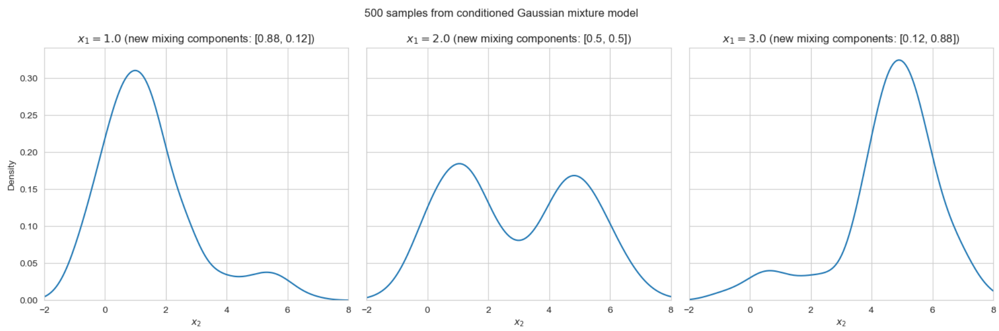


Figure 5: Conditional distributions of x_2 given x_1 from same Gaussian mixture model, for $x_1 = 1$, $x_1 = 2$ and $x_1 = 3$ (500 samples, smoothed by `sns.kdeplot`).

See Figure 6 for the distribution of latents for three sizes of VAE: a small one with a single transformation layer of size 10 (img $\rightarrow$ 10 $\rightarrow$ 2 $\rightarrow$ 10 $\rightarrow$ img), a medium one with transformation layers of sizes 100 and 10 (img $\rightarrow$ 100 $\rightarrow$ 10 $\rightarrow$ 2 $\rightarrow$ 10 $\rightarrow$ 100 $\rightarrow$ img) and a large one with transformation layers 300, 100 and 10. The small and medium VAEs were trained for 10 epochs, while the large one was trained for 20 epochs on the MNIST training set. Note that increasing the size of the VAE increases the ability of the model to separate the digits in the latent space.

See Figure 7 for a decoding of the latent space for the three VAEs. Note that increasing the size of the model, increases the number of clearly legible digits from 4 in the small model (0, 1, 7, 9) to all 10 in the large model latent space.

Again, see Sec. 4 for the code that generated these figures.

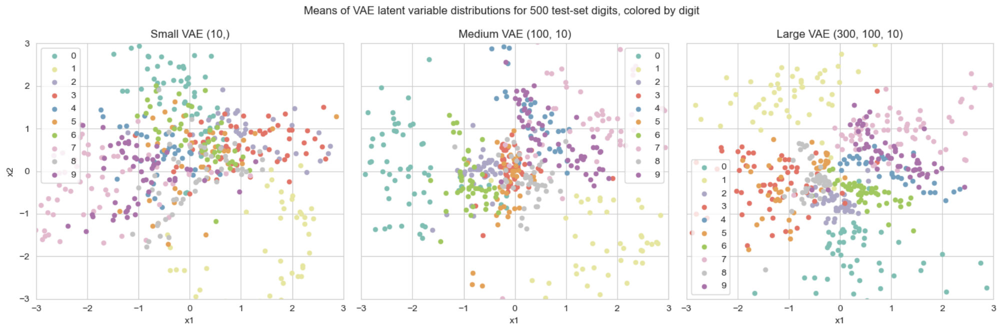
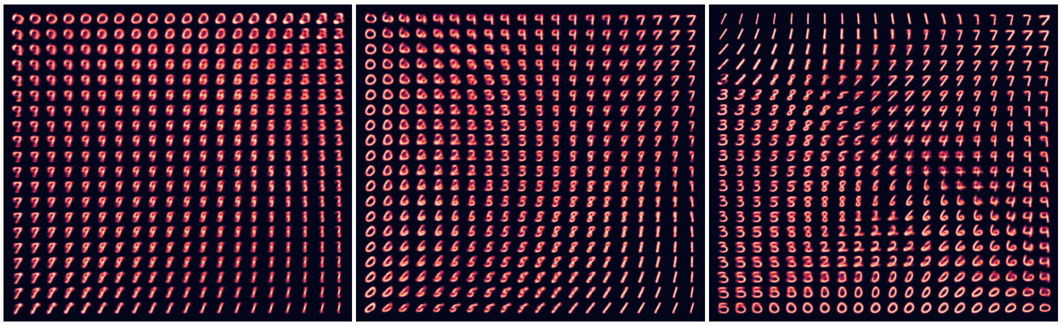


Figure 6: Distribution of latent variables for small-, medium- and large-sized VAEs, with datapoints sourced from test-set and coloured by the true label.



(a) Small (10,) VAE (b) Medium (100, 10) VAE (c) Large (300, 100, 10) VAE

Figure 7: Decoded representations of latent variables, where latents were sampled evenly using the inverse cumulative density function, for three different sizes of VAEs.

3 Code W1 Programming

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


# I'm not sure if we're allowed to use library functions for the distributions,
# so here are some multivariate distribution functions, re-implemented.
@np.vectorize(signature="(d),(d,d)->(r)")
def custom_multivariate_normal(mean: np.ndarray, a_transform: np.ndarray) -> np.
    ndarray:
    n_dim = a_transform.shape[1]
    e_noise = np.random.normal(0, 1, n_dim)
    return mean + a_transform @ e_noise


@np.vectorize
def custom_single_normal(mean: float, var: float) -> float:
    return np.random.normal(0, 1) * np.sqrt(var) + mean


def posterior_distribution_params(
    data: np.ndarray, targets: np.ndarray, target_variance: float
) -> tuple[np.ndarray, np.ndarray]:
    num, features = data.shape[:2]
    means = (
        data.T
        @ np.linalg.inv(target_variance * np.identity(num) + data @ data.T)
        @ targets
    ).flatten()
    covariance = (
        np.identity(features)
```

```python
        - data.T
        @ np.linalg.inv(target_variance * np.identity(num) + data @ data.T)
        @ data
    )
    return means, covariance


@np.vectorize(signature="(),(),(2),(2,2)->()")
def custom_2d_pdf(
    x0: float, x1: float, means: np.ndarray, covariance: np.ndarray
) -> float:
    difference = (np.array([x0, x1]) - means).reshape((-1, 1))
    d_dim = 2
    power = -0.5 * difference.T @ np.linalg.inv(covariance) @ difference
    exp_part = float(np.exp(power)[0, 0])
    normalize_part = (2 * np.pi) ** (d_dim / 2) * np.sqrt(np.linalg.det(covariance))
    return exp_part / normalize_part


def plot_param_posterior(
    means: np.ndarray,
    covariance: np.ndarray,
    res: int = 200,
    extra_title: str = "",
    ax: plt.Axes = None,
):
    axis = np.linspace(-3, 3, axis=0, num=res).reshape((-1, 1))
    pdf = custom_2d_pdf(axis, axis.T, means.flatten(), covariance)
    pdf[pdf == 0] = 1e-10
    log_pdf = np.log(pdf)
    if ax is None:
        _, ax = plt.subplots()
    img = ax.imshow(log_pdf[::-1, :], vmin=-5, vmax=5)
    plt.colorbar(img, fraction=0.046, pad=0.04)
    img.set_extent((-3, 3, -3, 3))
    ax.autoscale(False)
    ax.set_ylabel(r"$\theta_0$")
    ax.set_xlabel(r"$\theta_1$")
    if extra_title:
        extra_title += ": "
    ax.set_title(extra_title + r"Log posterior ($\log P(\theta|\mathcal{D})$)")


@np.vectorize(signature="(),(),(),(2,2)->()")
def custom_2d_posterior_predictive_variance(
    x0: float, x1: float, target_var: float, post_covar: np.ndarray
) -> float:
    x_vec = np.array([x0, x1]).reshape((2, 1))
    return target_var + x_vec.T @ post_covar @ x_vec


def plot_pp_var(
    target_var: float,
    post_covar: np.ndarray,
    res: int = 200,
    extra_title: str = "",
    ax=None,
):
    axis = np.linspace(-3, 3, axis=0, num=res).reshape((-1, 1))
    test_pdf = custom_2d_posterior_predictive_variance(
        axis, axis.T, target_var, post_covar
    )
    if ax is None:
        _, ax = plt.subplots()
    img = ax.imshow(test_pdf[::-1])
    plt.colorbar(img, fraction=0.046, pad=0.04)
    img.set_extent((-3, 3, -3, 3))
```

```python
    ax.autoscale(False)
    ax.set_ylabel(r"$x_0$")
    ax.set_xlabel(r"$x_1$")
    if extra_title:
        extra_title += ": "
    ax.set_title(extra_title + r"Variance of $P(\hat y|\mathcal{D}, \mathbf{x})$")


def plot_experiment(
    x_mean: np.ndarray | list,
    x_var_diag: np.ndarray | list,
    target_var: float,
    axs=None,
    seed: int = 42,
    t: str = "",
):
    np.random.seed(seed)
    if axs is None:
        _, axs = plt.subplots(ncols=2, figsize=(10, 5))
    x_vals = custom_multivariate_normal(
        np.repeat(np.atleast_2d(x_mean), 20, axis=0), np.diag(x_var_diag) ** 0.5
    )
    theta = np.array([-1, 1]).reshape((2, 1))
    y_vals = np.vectorize(custom_single_normal)(x_vals @ theta, 0.1)
    post_mean, post_covar = posterior_distribution_params(x_vals, y_vals, target_var)
    plot_param_posterior(post_mean, post_covar, ax=axs[0], extra_title=t)
    plot_pp_var(target_var, post_covar, ax=axs[1], extra_title=t)


_, axss = plt.subplots(nrows=4, ncols=2, figsize=(10, 20))
plot_experiment([0, 0], [1, 1], 0.1, axs=axss[0], t="Base")
plot_experiment([0, 0], [0.1, 1], 0.1, axs=axss[1], t=r"$\Sigma_x[0,0]\downarrow$")
plot_experiment(
    [1, 0],
    [0.1, 1],
    0.1,
    axs=axss[2],
    t=r"$\Sigma_x[0,0]\downarrow$, $\mu_x[0]\uparrow$",
)
plot_experiment([0, 0], [1, 1], 0.01, axs=axss[3], t=r"$\sigma^2_y\downarrow$")
plt.suptitle("Posterior and variance plots for Bayesian linear regression models")
plt.tight_layout()
plt.show()
```

4 Code W2 Programming

```python
from itertools import islice

import matplotlib.pyplot as plt
import numpy as np
import polars as pl
import seaborn as sns
import torch
import torch.distributions as dist
import torch.nn as nn
import torch.nn.functional as func
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from tqdm import tqdm, trange

torch.random.manual_seed(42)
mixture = dist.MixtureSameFamily(
    dist.Categorical(torch.tensor([0.5, 0.5], dtype=torch.float32)),
    dist.MultivariateNormal(
        torch.tensor([[1, 1], [3, 5]], dtype=torch.float32),
```

```python
        torch.stack([torch.eye(2), torch.eye(2)]),
    ),
)

sample = np.asarray(mixture.sample((500,)))
plt.scatter(sample[:, 0], sample[:, 1])
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.title("500 samples from specified Gaussian mixture model")
plt.show()


def condition_first(
    gmm: dist.MixtureSameFamily, cond: torch.Tensor
) -> dist.MixtureSameFamily:
    (c,) = cond.shape
    cat: dist.Categorical = gmm.mixture_distribution
    norm: dist.MultivariateNormal = gmm.component_distribution
    s_11 = norm.covariance_matrix[:, :c, :c]
    s_21 = norm.covariance_matrix[:, c:, :c]
    s_22 = norm.covariance_matrix[:, c:, c:]
    m_1 = norm.mean[:, :c]
    m_2 = norm.mean[:, c:]
    s_21_inv_s11 = s_21 @ torch.linalg.inv(s_11)
    cond_cat = dist.Categorical(
        logits=(
            cat.logits
            + dist.MultivariateNormal(loc=m_1, covariance_matrix=s_11).log_prob(cond)
        )
    )
    cond_norm = dist.MultivariateNormal(
        loc=m_2 + torch.einsum("knc,kc->kn", s_21_inv_s11, cond - m_1),
        covariance_matrix=s_22 - torch.einsum("knc,kNc->knN", s_21_inv_s11, s_21),
    )
    return dist.MixtureSameFamily(cond_cat, cond_norm)


def plot_conditioned(gmm: dist.MixtureSameFamily, cond: torch.Tensor, ax=None):
    gmm_cond = condition_first(gmm, cond)
    samples = np.asarray(gmm_cond.sample((500,)))
    if ax is None:
        _, ax = plt.subplots()
    sns.kdeplot(samples, ax=ax, legend=False)
    ax.set_xlabel("$x_2$")
    ax.set_xlim((-2, 8))
    ax.set_title(
        f"$x_1={cond[0]}$ "
        f"(new mixing components: {[round(x, 2) for x in gmm_cond.mixture_distribution.probs.tolist()]})"
    )


torch.random.manual_seed(42)
_, axs = plt.subplots(ncols=3, figsize=(15, 5), sharey=True)
plot_conditioned(mixture, torch.tensor([1], dtype=torch.float32), ax=axs[0])
plot_conditioned(mixture, torch.tensor([2], dtype=torch.float32), ax=axs[1])
plot_conditioned(mixture, torch.tensor([3], dtype=torch.float32), ax=axs[2])
plt.suptitle("500 samples from conditioned Gaussian mixture model")
plt.tight_layout()
plt.show()

train_loader = DataLoader(
    datasets.MNIST(
        "../data",
        train=True,
        download=True,
```

```python
        transform=transforms.Compose(
            [
                transforms.ToTensor(),
                transforms.Normalize((0.1307,), (0.3081,)),
                lambda x: x > 0,
                lambda x: x.float(),
            ]
        ),
    ),
    batch_size=50,
    shuffle=True,
)
test_loader = DataLoader(
    datasets.MNIST(
        "../data",
        train=False,
        transform=transforms.Compose(
            [
                transforms.ToTensor(),
                transforms.Normalize((0.1307,), (0.3081,)),
                lambda x: x > 0,
                lambda x: x.float(),
            ]
        ),
    ),
    batch_size=50,
    shuffle=True,
)


class VAE(nn.Module):
    def __init__(
        self,
        output_dim: int,
        transform_dims: list[int],
        latent_dim: int,
        # An experiment to see if multivariate encodings helped (not really)
        multivariate: bool = False,
    ):
        super().__init__()
        self.latent_dim = latent_dim
        trans_enc = []
        for dim in transform_dims:
            trans_enc.append(nn.LazyLinear(dim))
            trans_enc.append(nn.ReLU())
        self.trans_enc = nn.Sequential(*trans_enc)
        self.enc_mean = nn.LazyLinear(latent_dim)
        self.enc_log_var = nn.LazyLinear(
            latent_dim * (latent_dim + 1) // 2 if multivariate else latent_dim
        )
        dec = []
        for dim in reversed(transform_dims):
            dec.append(nn.LazyLinear(dim))
            dec.append(nn.ReLU())
        dec.append(nn.LazyLinear(output_dim))
        dec.append(nn.Sigmoid())
        self.dec = nn.Sequential(*dec)
        self.multivariate = multivariate

    def encode(self, x: torch.Tensor) -> dist.Distribution:
        trans = self.trans_enc(x.view(x.shape[0], -1))
        mean = self.enc_mean(trans)
        var_vals = torch.exp(self.enc_log_var(trans))
        if self.multivariate:
            var = torch.empty((x.shape[0], self.latent_dim, self.latent_dim))
            idx_u = torch.tril_indices(self.latent_dim, self.latent_dim)
            var[:, idx_u[0], idx_u[1]] = var_vals
```

```python
            var.mT[:, idx_u[0], idx_u[1]] = var_vals
            # hacky psd transform, the 0.1*eye providing a margin of error
            var = torch.einsum("bij,bik->bjk", var, var) + torch.eye(2) * 0.1
            return dist.MultivariateNormal(mean, var)
        else:
            return dist.Normal(mean, var_vals)

    def decode(self, x: torch.Tensor) -> torch.Tensor:
        return self.dec(x)

    def forward(self, x) -> tuple[torch.Tensor, dist.Distribution]:
        norm = self.encode(x)
        return self.decode(norm.rsample()), norm


def loss_fn(
    images: torch.Tensor,
    reconstructions: torch.Tensor,
    distributions: dist.Distribution,
) -> torch.Tensor:
    recon_loss = func.binary_cross_entropy(
        reconstructions, images.view_as(reconstructions), reduction="sum"
    )
    if isinstance(distributions, dist.Normal):
        diverg_loss = -0.5 * torch.sum(
            1
            + torch.log(distributions.variance)
            - distributions.mean.pow(2)
            - distributions.variance
        )
    elif isinstance(distributions, dist.MultivariateNormal):
        m_diff = 1 - distributions.mean
        batch_trace, batch_det = torch.vmap(torch.trace), torch.vmap(torch.det)
        diverg_loss = 0.5 * (
            torch.einsum("bd,bd->b", m_diff, m_diff)
            + batch_trace(distributions.covariance_matrix)
            + torch.log(batch_det(distributions.covariance_matrix))
            - distributions.mean.shape[1]
        ).sum(0)
    else:
        raise NotImplementedError

    return recon_loss + diverg_loss


test_model = VAE(784, [100, 10], 2, multivariate=False)
test_optim = torch.optim.Adam(test_model.parameters())
test_img = next(iter(train_loader))[0]
test_recon, test_dist = test_model(test_img)
print(test_dist)
print(test_recon.shape)
loss_fn(test_img, test_recon, test_dist)

device = "cpu"


def train_epoch(
    model: VAE, optimizer: torch.optim.Optimizer, epoch: int, show_bar: bool = True
) -> float:
    model.train()
    train_loss = 0
    bar = tqdm(
        train_loader,
        total=len(train_loader.dataset) // train_loader.batch_size,
        desc=f"Epoch {epoch}",
        leave=False,
        position=1,
```

```
        disable=not show_bar,
    )
    for batch_idx, (data, _) in enumerate(bar):
        data = data.to(device)
        optimizer.zero_grad()
        reconstructions, distribution = model(data)
        loss = loss_fn(data, reconstructions, distribution)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
        if batch_idx % 100 == 0:
            bar.set_postfix(loss=loss.item())
    return train_loss


def train_loop(
    epochs: int,
    model: VAE = None,
    optimizer: torch.optim.Optimizer = None,
    seed: int = 42,
    sub_bar: bool = False,
    transform_dims: list[int] = None,
    multivariate=False,
) -> VAE:
    torch.random.manual_seed(seed)
    if transform_dims is None:
        transform_dims = [10]
    if model is None:
        model = VAE(784, transform_dims, 2, multivariate=multivariate)
    model = model.to(device)
    model(next(iter(train_loader))[0].to(device))
    if device in ("cpu", "cuda"):
        model.compile()
    if optimizer is None:
        optimizer = torch.optim.Adam(model.parameters())
    bar = trange(epochs, unit="epoch", desc=f"Training {transform_dims}")
    for epoch in bar:
        train_loss = train_epoch(model, optimizer, epoch, show_bar=sub_bar)
        bar.set_postfix(epoch_loss=train_loss)
    return model


trained_model_s = train_loop(10, transform_dims=[10])
trained_model_m = train_loop(10, transform_dims=[100, 10])
trained_model_l = train_loop(20, transform_dims=[300, 60, 10])


def plot_model(model: VAE, seed: int = 42, grid_size: int = 10):
    with torch.no_grad():
        model.eval()
        torch.random.manual_seed(seed)
        space_1d = torch.linspace(0.01, 0.99, steps=grid_size)
        space_2d = torch.cartesian_prod(space_1d.flip(0), space_1d).flip(1)
        samples = dist.Normal(0, 1).icdf(space_2d).to(device)
        images = model.decode(samples).reshape(-1, 28, 28).cpu()
        _, axss = plt.subplots(
            grid_size,
            grid_size,
            gridspec_kw={
                "wspace": -0.8 if grid_size > 7 else -0.1,
                "hspace": 0,
                "bottom": 0,
                "top": 1,
                "left": 0,
                "right": 1,
            },
            facecolor="#020419",
```

```python
        )
        for ax, img in zip(axss.flat, images):
            ax.imshow(img)
            ax.axis("off")
        plt.show()


plot_model(trained_model_s, grid_size=20)
plot_model(trained_model_m, grid_size=20)
plot_model(trained_model_l, grid_size=20)


def plot_classes(
    model: VAE, batches: int = 1, title="", seed: int = 42, ax=None, limits: int = 3
):
    torch.random.manual_seed(seed)
    res = []
    if ax is None:
        _, ax = plt.subplots()
    with torch.no_grad():
        model.eval()
        for data, labels in islice(train_loader, batches):
            data = data.to(device)
            latents = model.encode(data).mean
            res.append(
                pl.DataFrame(
                    {
                        "x1": latents[:, 0].numpy(),
                        "x2": latents[:, 1].numpy(),
                        "Digit": labels.numpy(),
                    }
                )
            )
    for (dig, grp), col in zip(
        pl.concat(res).sort("Digit").group_by("Digit", maintain_order=True),
        plt.colormaps["Set3"].colors,
    ):
        grp.to_pandas().plot.scatter(
            x="x1", y="x2", label=dig[0], color=tuple(np.array(col) * 0.9), ax=ax
        )
    ax.set_xlim((-limits, limits))
    ax.set_ylim((-limits, limits))
    ax.set_title(title)


n_batches = 10
_, axs = plt.subplots(ncols=3, tight_layout=True, figsize=(15, 5), sharey=True)
plot_classes(trained_model_s, batches=n_batches, title="Small VAE (10,)", ax=axs[0])
plot_classes(
    trained_model_m, batches=n_batches, title="Medium VAE (100, 10)", ax=axs[1]
)
plot_classes(
    trained_model_l, batches=n_batches, title="Large VAE (300, 100, 10)", ax=axs[2]
)
plt.suptitle(
    "Means of VAE latent variable distributions for 500 test-set digits, colored by
    digit"
)
plt.show()

trained_model_multivar = train_loop(20, transform_dims=[300, 60, 10], multivariate=
    True)

plot_classes(
    trained_model_multivar,
    batches=n_batches,
    title="Multivariate VAE (300, 100, 10,)",
```

```
    limits=8,
)
plot_model(trained_model_multivar, grid_size=20)
```