Homework 3
Statistical Learning for Decision Making 2023

Li Ju
li.ju@it.uu.se

March 21, 2023

1 Problem 1

a). K is fixed as 1000 and threshold α is fixed as 0.010. The code for the implementation is deferred to Appendix A.1.1. Results are shown in Table 1:

n	θ	α_θ	Reject?
5	30	0.020	False
5	40	0.193	False
50	30	0.000	True
50	40	0.052	False
500	30	0.000	True
500	40	0.236	False

Table 1: Results for data consistency test

b). If we use the suprisal loss $\ell(z) = -\ln p_\theta(z)$ as the loss function, the empirical risk function is given as follows:

$$\begin{aligned}\hat{R}(\theta)_n &= \mathbb{E}_n[-\ln p_\theta] \\ &= \frac{1}{n}\sum_{i=1}^{n} -\ln \frac{\theta^{z_i} e^{-\theta}}{z_i!}\end{aligned}$$

The gradient of the risk function w.r.t θ is derived as follows:

$$\begin{aligned}\nabla_\theta \hat{R}(\theta) &= -\frac{1}{n}\sum_{i=1}^{n} \frac{z_i!}{\theta^{z_i} e^{-\theta}} \frac{z_i \theta^{z_i - 1} e^{-\theta} - \theta^{z_i} e^{-\theta}}{z_i!} \\ &= -\frac{1}{n}\sum_{i=1}^{n} (z_i - \theta)\end{aligned}$$

let $\nabla_\theta \hat{R}(\hat{\theta}) \coloneqq 0$, we have

$$\begin{aligned}\nabla_\theta \hat{R}(\hat{\theta}) &= -\frac{1}{n}\sum_{i=1}^{n} (z_i - \hat{\theta}) = 0 \\ \hat{\theta} &= \sum_{i=1}^{n} \frac{z_i}{n}\end{aligned}$$

The code for the implementation is appended in Appendix A.1.2. Results are shown in Table 2.

n	$\hat{\theta}$	α_θ	Reject?
5	40.000	0.200	False
50	40.420	0.019	False
500	39.782	0.293	False

Table 2: Results for data consistency test for ERM estimation

c). The code for the implementation is appended in Appendix A.1.3. Results are shown in Table 3.

n	r	$\hat{\theta}$	α_θ	Reject?
5	10.0	1.6	0.260	False
5	10000.0	2509459.0	0.000	True
50	10.0	2.48	0.238	False
50	10000.0	2501609.18	0.000	True
500	10.0	2.556	0.003	True
500	10000.0	2498788.73	0.000	True

Table 3: Results for data consistency test for true data generated from different Negative Binomial distributions

2 Problem 2

a). Taken the solution from Homework 2 Problem 4, with the suprisal loss function, the empirical risk function and the minimizer for $\mathbf{s}_n$ are given as follows:

$$\begin{aligned} R_n(\theta) &= \mathbb{E}_n[\ell(\mathbf{z})_\theta] \\ &= \mathbb{E}_n[\frac{d\ln v}{2} + \frac{1}{2v}\|\mathbf{z} - 2c^{-1}\boldsymbol{\mu}(\mathbf{s})\|^2] \\ &= \frac{d\ln v}{2} + \frac{1}{2vn}\sum_{i=1}^{n}\|\mathbf{z}_i - 2c^{-1}\boldsymbol{\mu}(\mathbf{s})\|^2 \end{aligned}$$

$$\text{where } \boldsymbol{\mu}(\mathbf{s}) = \begin{bmatrix} \|\mathbf{s} - \mathbf{a}_1\| \\ \|\mathbf{s} - \mathbf{a}_2\| \\ \|\mathbf{s} - \mathbf{a}_3\| \end{bmatrix}$$

$$\hat{\mathbf{s}}_n = \arg\min_{\mathbf{s}} \mathbb{E}_n[\|\mathbf{z} - 2c^{-1}\boldsymbol{\mu}(\mathbf{s})\|^2]$$

The derivative of $\hat{R}_n$ w.r.t v is given as follows:

$$\frac{\partial \hat{R}_n}{\partial v} = \frac{d}{2v} - \frac{1}{2v^2 n}\sum_{i=1}^{n}\|\mathbf{z}_i - 2c^{-1}\boldsymbol{\mu}(\mathbf{s})\|^2$$

Thus, the minimizer for v is given when $\frac{\partial \hat{R}_n}{\partial v} = 0$ as follows:

$$\hat{v}_n = \frac{\sum_{i=1}^{n}\|\mathbf{z}_i - 2c^{-1}\boldsymbol{\mu}(\hat{\mathbf{s}}_\mathbf{n})\|^2}{dn}$$

With $\hat{s}_n$ and $\hat{v}_n$, we use the consistency test for different data generation process (Gaussian and Exponential distribution). Results are reported Table 4. All codes for the implementation are deferred in Appendix A.2.

DGP	n	$\hat{s}_n$	$\hat{v}_n$	α_θ	Reject?
gaussian	10	[196.777,198.499]	5.123e-15	0.159	False
gaussian	100	[198.311,201.279]	9.364e-15	0.471	False
gaussian	1000	[200.317,199.052]	9.668e-15	0.345	False
exp	10	[199.838,248.304]	2.761e-12	0.031	False
exp	100	[172.127,188.091]	2.915e-12	0.000	True
exp	1000	[196.187,200.010]	2.314e-12	0.000	True

Table 4: Results for data consistency test for Problem 2

3 Problem 3

a). The posterior belief $b(\boldsymbol{\theta}|\mathbf{z}^n)$ is given as follows by the Bayes' rule:

$$\begin{aligned} b(\boldsymbol{\theta}|\mathbf{z}^n) &= \frac{p(\mathbf{z}^n|\boldsymbol{\theta})b(\boldsymbol{\theta})}{p(\mathbf{z}^n)} \\ &\propto p(\mathbf{z}^n|\boldsymbol{\theta})b(\boldsymbol{\theta}) \end{aligned}$$

By taking logarithm of the equation, we have

$$\begin{aligned} \ln b(\boldsymbol{\theta}|\mathbf{z}^n) &= \ln p(\mathbf{z}^n|\boldsymbol{\theta}) + \ln b(\boldsymbol{\theta}) + \text{const.} \\ &= \sum_{i=1}^{n} \ln p(\mathbf{z}|\boldsymbol{\theta}) + \ln b(\boldsymbol{\theta}) + \text{const.} \\ &= n\mathbb{E}_n[-\ell(\mathbf{z};\boldsymbol{\theta})] + \ln \mathcal{N}(\mathbf{s};\mathbf{s}_0, v_0\mathbf{I}) + \text{const.} \\ &= -nL(\boldsymbol{\theta}, p_n) - \frac{1}{2v_0}(\mathbf{s}-\mathbf{s}_0)^\top(\mathbf{s}-\mathbf{s}_0) + \text{const.} \\ &= -nL(\boldsymbol{\theta}, p_n) - \frac{1}{2v_0}\|\mathbf{s}-\mathbf{s}_0\|^2 + \text{const.} \\ &\propto -L(\boldsymbol{\theta}, p_n) - \frac{1}{2nv_0}\|\mathbf{s}-\mathbf{s}_0\|^2 + \text{const.} \end{aligned}$$

The as-derived posterior belief is exactly proportional to the negative regularized empirical risk function. Thus, the maximizer of the posterior belief is also the minimizer of the regularized empirical risk function.

$v_0 \to \infty$ means that the prior distribution is almost uniform in $\mathbb{R}^2$. This leads to $-\frac{1}{2nv_0}\|\mathbf{s}-\mathbf{s}_0\|^2 \to 0$ and the Maximum Posterior Belief estimation is identical to the standard ERM estimation.

b). With the prior belief $\theta \sim \mathcal{N}(\mathbf{s}; [50, 50]^\top, 10^2\mathbf{I})$, we use the maximum posterior belief to do the inference with sample sizes of $n \in \{1, 10, 100, 1000\}$. Results are reported in Table 5. The code for the implementation is deferred in Appendix A.3. The pros of MAP include that prior

n	$\hat{s}_n$	$\hat{v}_n$	α_θ	Reject?
1	[50.243,50.227]	1.406e-12	0.000	True
10	[52.548,52.319]	1.309e-12	0.000	True
100	[80.491,78.205]	8.610e-13	0.000	True
1000	[198.021,199.212]	1.015e-14	0.040	False

Table 5: Results for data consistency test for Problem 3

information about the distribution of parameters can be used to estimate the target parameter with less computational cost. However, if the estimation will heavily rely on the prior belief if the number of samples are small.

4 Problem 4

a). Using the suprisal loss $\ell = -\ln p_{\boldsymbol{\theta}}(\mathbf{z})$, the risk function of the model is given as follows:

$$\begin{aligned} L(\boldsymbol{\theta}) &= \mathbb{E}[\sum_{k=1}^{d} \ln \mathcal{N}(z_k; \theta_k, \sigma)] \\ &= \mathbb{E}[d \ln(\sigma\sqrt{2\pi}) + \frac{1}{2}\sum_{k=1}^{d}(\frac{z_k - \theta_k}{\sigma})^2] \end{aligned}$$

The target parameter $\boldsymbol{\theta}_\circ$ is given when the gradient of the risk function $\nabla L(\boldsymbol{\theta}) \coloneqq \mathbf{0}$. For any $k \in \{0, \dots d\}$, we have

$$\begin{aligned} \frac{\partial L(\boldsymbol{\theta})}{\partial \theta_k} &= \mathbb{E}[\frac{z_k - \theta_k}{\sigma}] \\ 0 &= \mathbb{E}[\frac{z_k - \theta_{k,\circ}}{\sigma}] \\ \theta_{k,\circ} &= \mathbb{E}[z_k] \end{aligned}$$

Thus, we have $\boldsymbol{\theta}_\circ \coloneqq [\theta_1, \dots \theta_d]^\top = [\mathbb{E}[z_1], \dots \mathbb{E}[z_d]] = \mathbb{E}[\mathbf{z}]$.

When using empirical risk with $n = 1$, we have

$$\begin{aligned} \hat{L}_n(\boldsymbol{\theta}) &= \sum_{k=1}^{d} \ln \mathcal{N}(z_k; \theta_k, \sigma) \\ &= d \ln(\sigma\sqrt{2\pi}) + \frac{1}{2}\sum_{k=1}^{d}(\frac{z_k - \theta_k}{\sigma})^2 \\ \hat{\theta}_{n,k} &= z_k \text{ for } k \in \{1, \dots d\} \end{aligned}$$

Thus $\tau(\hat{\boldsymbol{\theta}}_n) = z_k$ for $k \in \{1, \dots d\}$

b). With the model $p_theta(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mu\mathbf{1}, (v + \sigma^2)\mathbf{I})$, using the suprisal loss, we have the risk function as follows:

$$\begin{aligned} L(\boldsymbol{\theta}) &= \mathbb{E}[\frac{d}{2}\ln 2\pi + \frac{1}{2}\ln \mathrm{tr}(\Sigma) + \frac{1}{2}(\mathbf{z} - \mu\mathbf{1})^\top \Sigma^{-1}(\mathbf{z} - \mu\mathbf{1})] \\ &= \mathbb{E}[\frac{d}{2}\ln 2\pi + \frac{\ln d(\sigma^2 + v)}{2} + \frac{1}{2}(\mathbf{z} - \mu\mathbf{1})^\top (\sigma^2 + v)^{-1}\mathbf{I}(\mathbf{z} - \mu\mathbf{1})] \end{aligned}$$

The empirical risk function with $n = 1$ is thus as follows:

$$\hat{L}_n(\boldsymbol{\theta}) = \frac{d}{2}\ln 2\pi + \frac{\ln d(\sigma^2 + v)}{2} + \frac{1}{2}(\mathbf{z} - \mu\mathbf{1})^\top (\sigma^2 + v)^{-1}\mathbf{I}(\mathbf{z} - \mu\mathbf{1})$$

The minimizer for μ and v is given when the gradient of the empirical risk function equals to 0 ($\nabla \hat{L}_n(\boldsymbol{\theta}) = \mathbf{0}$) as follows:

$$\begin{aligned} \frac{\partial \hat{L}_n}{\partial \mu} &= (\mu\mathbf{1} - \mathbf{z})^\top (\sigma^2 + v)^{-1}\mathbf{I}\mathbf{1}^\top \\ 0 &= (\hat{\mu}\mathbf{1} - \mathbf{z})^\top \mathbf{1}^\top \\ \hat{\mu} &= \frac{1}{d}\mathbf{1}^\top \mathbf{z} \end{aligned}$$

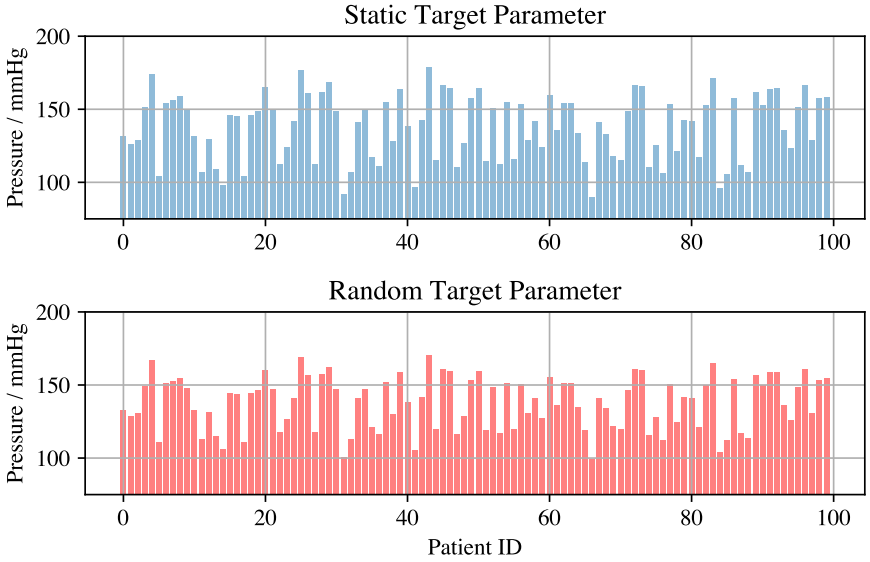

Figure 1: Contour of the empirical risk function and GD steps

and

$$\begin{aligned}
\frac{\partial \hat{L}_n}{\partial \mu} &= \frac{1}{2(\sigma^2+v)} - \frac{1}{2d}(\mathbf{z}-\mu\mathbf{1})^\top(\mathbf{z}-\mu\mathbf{1})(\sigma^2+v)^{-2} \\
0 &= \frac{1}{2(\sigma^2+\hat{v})} - \frac{1}{2d}(\mathbf{z}-\hat{\mu}\mathbf{1})^\top(\mathbf{z}-\hat{\mu}\mathbf{1})(\sigma^2+\hat{v})^{-2} \\
1 &= \frac{1}{d}(\mathbf{z}-\hat{\mu}\mathbf{1})^\top(\mathbf{z}-\hat{\mu}\mathbf{1})(\sigma^2+\hat{v})^{-1} \\
\hat{v} &= \frac{1}{d}(\mathbf{z}-\hat{\mu}\mathbf{1})^\top(\mathbf{z}-\hat{\mu}\mathbf{1}) - \sigma^2
\end{aligned}$$

Since we have $v \geq 0$ as a variance, $\hat{v} = \max(0, \frac{1}{d}\|\mathbf{z}-\hat{\mu}\mathbf{1}\|^2 - \sigma^2)$

c). When the sensor error $\sigma \to 0$, the target parameter for each observation z_k is exactly z_k. When $\sigma \to \infty$, the target parameter for all observations are fixed as $\mu_\circ$ and we expect all patients have exactly identical true blood pressure.

With a *random* target parameter, it is convenient to set up a model taking advantages of both the model in a) and b). We can make the trade-off between the learning and adaption by adjusting σ^2.

d). With both static and random target parameter settings, the resulting comparison between estimated blood pressures of patients is shown as Figure 1. The code for the implementation is appended in Appendix A.4. The random target parameter setting tends to have less variance of estimations from different patients while the static target parameter treat each patient independent and does use the information of average blood pressures of patients.

A Codes

A.1 Code for Problem 1

A.1.1 Problem 1.a

```
import jax.numpy as jnp
import jax


thetas = [30, 40]
ns = [5, 50, 500]

K = 1000

key = jax.random.PRNGKey(42)


def _ln_factorial(z):
    return jax.lax.fori_loop(1, z+1, lambda z, acc: jnp.log(z)+acc, 0)


def loss(z, theta):
    return z*jnp.log(theta) - theta - _ln_factorial(z)


for n in ns:
    # generate true samples
    _, key = jax.random.split(key)
    true_zs = jax.random.poisson(key, 40, shape=(n,))

    # estimate theta
    theta_hat = jnp.average(true_zs)

    # function to calculate T
    vecloss = jax.vmap(lambda z: loss(z, theta_hat))
    T_fn = jax.jit(lambda zs: vecloss(zs).std()**2)

    T_true = T_fn(true_zs)

    for theta in thetas:
        less_than = 0
        for _ in range(K):
            _, key = jax.random.split(key)
            gen_zs = jax.random.poisson(key, theta, shape=(n,))
            T_gen = T_fn(gen_zs)
            less_than = less_than + 1 if T_gen <= T_true else less_than

        P_theta = less_than/K
        alpha_theta = min(P_theta, 1-P_theta)
        print(f"n={n}, theta={theta}: alpha_theta={alpha_theta:.3f}")
        # print(f"{n} & {theta} & {alpha_theta:.3f} & "
        #       f"{True if alpha_theta < 0.01 else False} \\\\")
```

A.1.2 Problem 1.b

```
import jax.numpy as jnp
import jax


ns = [5, 50, 500]

K = 1000

key = jax.random.PRNGKey(0)


def _ln_factorial(z):
    return jax.lax.fori_loop(1, z+1, lambda z, acc: jnp.log(z)+acc, 0)


def loss(z, theta):
    return z*jnp.log(theta) - theta - _ln_factorial(z)


for n in ns:
    # generate true samples
    _, key = jax.random.split(key)
    true_zs = jax.random.poisson(key, 40, shape=(n,))

    # estimate theta
    theta_hat = jnp.average(true_zs)

    # function to calculate T
    vecloss = jax.vmap(lambda z: loss(z, theta_hat))
    T_fn = jax.jit(lambda zs: vecloss(zs).std()**2)

    T_true = T_fn(true_zs)

    less_than = 0
    for _ in range(K):
        _, key = jax.random.split(key)
        gen_zs = jax.random.poisson(key, theta_hat, shape=(n,))
        T_gen = T_fn(gen_zs)
        less_than = less_than + 1 if T_gen <= T_true else less_than

    P_theta = less_than/K
    alpha_theta = min(P_theta, 1-P_theta)
    print(f"n={n}, theta={theta_hat:.3f}: alpha_theta={alpha_theta:.3f}")
    # print(f"{n} & {theta_hat:.3f} & {alpha_theta:.3f} & "
    #       f"{True if alpha_theta < 0.01 else False} \\\\")
```

A.1.3 Problem 1.c

```
import jax.numpy as jnp
import numpy as np
import jax


ns = [5, 50, 500]
```

```
rs = [1e1, 1e4]

K = 1000

key = jax.random.PRNGKey(0)


def _ln_factorial(z):
    return jax.lax.fori_loop(1, z+1, lambda z, acc: jnp.log(z)+acc, 0)


def _stirling_ln_factorial(z):
    return z * jnp.log(z) - z + 1


def loss(z, theta):
    return -(z*jnp.log(theta) - theta - _ln_factorial(z))


def approx_loss(z, theta):
    return -(z*jnp.log(theta) - theta - _stirling_ln_factorial(z))


for n in ns:
    for r in rs:
        p = 40/(r+40)
        true_zs = np.random.negative_binomial(r, p, size=n)
        true_zs = jnp.array(true_zs)

        # estimate theta
        theta_hat = jnp.average(true_zs)

        # function to calculate T
        # Note: Stirling approximation is used for large z:
        # ln(z!) \approx nln(n)-n+1
        vecloss = jax.vmap(lambda z: loss(z, theta_hat)) if r <= 1e2 \
            else jax.vmap(lambda z: approx_loss(z, theta_hat))
        T_fn = jax.jit(lambda zs: vecloss(zs).std()**2)

        T_true = T_fn(true_zs)

        less_than = 0
        for _ in range(K):
            _, key = jax.random.split(key)
            gen_zs = jax.random.poisson(key, theta_hat, shape=(n,))
            T_gen = T_fn(gen_zs)
            less_than = less_than + 1 if T_gen <= T_true else less_than

        P_theta = less_than/K
        alpha_theta = min(P_theta, 1-P_theta)
        print(f"n={n}, r={r}, theta={theta_hat:.3f},"
              f"alpha_theta={alpha_theta:.3f}")
```

A.2 Code for Problem 2

```
import jax.numpy as jnp
import jax
import numpy as np


# settings
ns = [10, 100, 1000]

c = 3e8
a1 = jnp.array([0, 0])
a2 = jnp.array([350, 50])
a3 = jnp.array([250, 350])
s = jnp.array([200, 200])
eps = 1e-10
d = 3
K = 1000

key = jax.random.PRNGKey(42)


# functions for sn optimization: single sample
def _objective_sample(z, s):
    mu = jnp.array([
        jnp.linalg.norm(s - a1),
        jnp.linalg.norm(s - a2),
        jnp.linalg.norm(s - a3),
    ])
    values = (z - 2/c*mu) ** 2
    return values.sum()


# vectorized objective function
_objective_vec = jax.jit(jax.vmap(_objective_sample, (0, None)))


# gradient for the objective function
@jax.jit
def objective_grad(zs, s):
    E_z = zs.mean(axis=0)
    mu = jnp.array([
        jnp.linalg.norm(s - a1),
        jnp.linalg.norm(s - a2),
        jnp.linalg.norm(s - a3),
    ])
    dmuds = jnp.array([(s - a1)/(mu[0] + eps),
                       (s - a2)/(mu[1] + eps),
                       (s - a3)/(mu[2] + eps)])
    return 4/c * (2/c*mu - E_z) @ dmuds


# loss function: single sample
def loss_sample(z, s, v):
    return d/2 * jnp.log(v) + _objective_sample(z, s).sum()/(2*v)
```

```
# GD to estimate sn
def get_sn(start, num_steps, lr, zs):
    for _ in range(num_steps):
        gradient = objective_grad(zs, start)
        start -= lr * gradient
    return start


true_data_generators = {
    "gaussian": lambda key, mean, cov, num:
        jax.random.multivariate_normal(key, mean, cov, shape=(num, )),
    "exp": lambda key, mean, cov, num:
        jnp.array([np.random.exponential(mean[i], num)
                   for i in range(3)]).transpose()
}


mean = 2/c*jnp.linalg.norm(s - jnp.array([a1, a2, a3]), axis=1)
cov = 1e-14 * jnp.eye(3)

for generator_name in true_data_generators.keys():
    generator = true_data_generators[generator_name]
    for n in ns:
        # estimate thetas
        _, key = jax.random.split(key)
        true_zs = generator(key, mean, cov, n)

        # modeling
        sn = get_sn(start=jnp.array([0.0, 0.0]),
                    num_steps=2000,
                    lr=1e15,
                    zs=true_zs)
        vn = _objective_vec(true_zs, sn).mean() / d

        loss_vec = jax.vmap(lambda z: loss_sample(z, sn, vn))
        T_fn = jax.jit(lambda zs: loss_vec(zs).std() ** 2)

        T_true = T_fn(true_zs)

        less_than = 0
        est_mean = 2/c*jnp.linalg.norm(sn - jnp.array([a1, a2, a3]), axis=1)
        est_cov = vn*jnp.eye(3)
        for _ in range(K):
            _, key = jax.random.split(key)
            gen_zs = jax.random.multivariate_normal(key, est_mean,
                                                    est_cov, shape=(n, ))
            T_gen = T_fn(gen_zs)
            less_than = less_than + 1 if T_gen <= T_true else less_than

        P_theta = less_than/K
        alpha_theta = min(P_theta, 1-P_theta)
        print(f"{generator_name} & {n} & {sn} & {vn} & "
              f"{alpha_theta:.3f} & "
```

```
            f"{True if alpha_theta < 0.01 else False} \\\\")
```

A.3 Code for Problem 3

```
import jax.numpy as jnp
import jax


# settings
ns = [1, 10, 100, 1000]

c = 3e8
a1 = jnp.array([0, 0])
a2 = jnp.array([350, 50])
a3 = jnp.array([250, 350])
s = jnp.array([200, 200])
eps = 1e-10
d = 3
K = 1000
s0 = jnp.array([50., 50.])
v0 = 1e2

key = jax.random.PRNGKey(42)


# functions for sn optimization: single sample
def _objective_sample(z, s):
    mu = jnp.array([
        jnp.linalg.norm(s - a1),
        jnp.linalg.norm(s - a2),
        jnp.linalg.norm(s - a3),
    ])
    values = (z - 2/c*mu) ** 2
    return values.sum()


# vectorized objective function
_objective_vec = jax.jit(jax.vmap(_objective_sample, (0, None)))


@jax.jit
# gradient for the objective function
def objective_grad(zs, s):
    n = zs.shape[0]
    E_z = zs.mean(axis=0)
    mu = jnp.array([
        jnp.linalg.norm(s - a1),
        jnp.linalg.norm(s - a2),
        jnp.linalg.norm(s - a3),
    ])
    dmuds = jnp.array([(s - a1)/(mu[0] + eps),
                       (s - a2)/(mu[1] + eps),
                       (s - a3)/(mu[2] + eps)])
    return (4/c * (2/c*mu - E_z) @ dmuds /
            (_objective_vec(zs, s).mean(axis=0))/d) + (s-s0)/(n*v0)
```

```
# loss function: single sample
def loss_sample(z, s, v):
    return d/2 * jnp.log(v) + _objective_sample(z, s).sum()/(2*v)


# GD to estimate sn
def get_sn(start, num_steps, lr, zs):
    for _ in range(num_steps):
        # print(start)
        gradient = objective_grad(zs, start)
        start -= lr * gradient
    return start


mean = 2/c*jnp.linalg.norm(s - jnp.array([a1, a2, a3]), axis=1)
cov = 1e-14 * jnp.eye(3)


for n in ns:
    # estimate thetas
    _, key = jax.random.split(key)
    true_zs = jax.random.multivariate_normal(key, mean, cov, shape=(n, ))

    # modeling
    sn = get_sn(start=jnp.array([0.0, 0.0]),
                num_steps=10000,
                lr=1e2,
                zs=true_zs)
    vn = _objective_vec(true_zs, sn).mean() / d

    loss_vec = jax.vmap(lambda z: loss_sample(z, sn, vn))
    T_fn = jax.jit(lambda zs: loss_vec(zs).std() ** 2)

    T_true = T_fn(true_zs)

    less_than = 0
    est_mean = 2/c*jnp.linalg.norm(sn - jnp.array([a1, a2, a3]), axis=1)
    est_cov = vn*jnp.eye(3)
    for _ in range(K):
        _, key = jax.random.split(key)
        gen_zs = jax.random.multivariate_normal(key, est_mean,
                                                est_cov, shape=(n, ))
        T_gen = T_fn(gen_zs)
        less_than = less_than + 1 if T_gen <= T_true else less_than

    P_theta = less_than/K
    alpha_theta = min(P_theta, 1-P_theta)
    print(f"{n} & {sn} & {vn} & "
          f"{alpha_theta:.3f} & "
          f"{True if alpha_theta < 0.01 else False} \\\\")
```

A.4 Code for Problem 4

```
import jax.numpy as jnp
import jax
import matplotlib.pyplot as plt


key = jax.random.PRNGKey(42)

d = 100
sigma = 10
minval, maxval = 100, 170

keys = jax.random.split(key, num=d+1)
key, keys = keys[0], keys[1:d+1]
mus = jax.vmap(lambda key: jax.random.uniform(
    key, minval=minval, maxval=maxval))(keys)

keys = jax.random.split(key, num=d+1)
key, keys = keys[0], keys[1:d+1]
zs = jax.vmap(lambda key, m: jax.random.normal(key)*sigma+m, (0, 0))(keys, mus)

static_thetas = zs

hat_mu = zs.mean()
hat_v = jnp.maximum(((zs - hat_mu) ** 2).mean() - sigma**2, 0)

adapt_thetas = jax.vmap(lambda z: hat_v/(hat_v+sigma**2)*z +
                        sigma**2/(hat_v+sigma**2)*hat_mu)(zs)

print(static_thetas.shape, adapt_thetas.shape)

fig, ax = plt.subplots(figsize=(6, 4), nrows=2, ncols=1)

ax[0].bar(x=range(d), height=static_thetas, alpha=0.5)
ax[0].set_ylim((75, 200))
ax[0].set_title("Static Target Parameter")
ax[0].set_ylabel("Pressure / mmHg")

ax[1].bar(x=range(d), height=adapt_thetas, color='red', alpha=0.5)
ax[1].set_ylim((75, 200))
ax[1].set_title("Random Target Parameter")
ax[1].set_ylabel("Pressure / mmHg")
ax[1].set_xlabel("Patient ID")


# ax.set_title("Gradient Descent for optimization")
# ax.set_xlabel(r"£\theta_1£ / m")
# ax.set_ylabel(r"£\theta_2£ / m")

# ax.legend()
fig.tight_layout()
fig.savefig("hw3_4d.pdf", dpi=500)
```