Statistical Models and Computing Methods **2020 Fall**

Problem Set 2. Solution

2020-12

1 Problem 1

(1) Let X have the standard Laplace distribution. Use *importance sampling* based on 100000 draws from the standard normal as the proposal density to estimate $\mathbb{E}(X)$, $\mathbb{V}\text{ar}(X)$ and $\mathbb{P}(X > 2)$.

Solution. Let $X \sim \text{Laplace}(0,1) \sim f(x)$, $q(x) \sim \mathcal{N}(0,1)$, and $w(x) \coloneqq \frac{f(x)}{q(x)} = \sqrt{\frac{\pi}{2}} \exp\left(\frac{1}{2}x^2 - |x|\right)$. Following $\mathbb{E}_f[h(X)] = \mathbb{E}_q\left[h(X)\frac{f(X)}{q(X)}\right] = \mathbb{E}_q[h(X)w(X)]$, we have

$$\widehat{\mathbb{E}}(X) = \frac{1}{n}\sum_{i=1}^{n} x_i\, w(x_i)$$

$$\widehat{\mathbb{V}\text{ar}}(X) = \frac{1}{n}\sum_{i=1}^{n} x_i^2\, w(x_i) - \left(\mathbb{E}(X)\right)^2$$

$$\widehat{\mathbb{P}}(X > 2) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(x_i > 2)\, w(x_i)$$

```python
import numpy as np
import pandas as pd
from scipy import stats
from IPython.display import display

def is_est(size=10**5, reverse=False):
    if reverse:
        x = np.random.laplace(size=size)
        w = stats.norm.pdf(x) / stats.laplace.pdf(x)
    else:
        x = np.random.normal(size=size)
        w = stats.laplace.pdf(x) / stats.norm.pdf(x)
    EX = (x * w).mean()
    VarX = ((x ** 2.0) * w).mean()
    P = ((x > 2.0) * w).mean()
    return x, w, EX, VarX, P

np.random.seed(1234)
x, w, EX, VarX, P = is_est()

df = pd.DataFrame({
    "E(X)"  :    ["0", f"{EX:.4f}"],
    "Var(X)":    ["2", f"{VarX:.4f}"],
    "P(X>2)":    [f"{1-stats.laplace.cdf(2):.4f}", f"{P:.4f}"],
    }, index=["Truth", "Estimate"]
)
```

```python
display(df)
print(f"Weights: [{w.min():.2f}, {w.max():.2f}]")
```

```
            E(X)     Var(X)     P(X>2)
Truth       0        2          0.0677
Estimate   -0.0043   1.5628     0.0583
Weights: [0.76, 127.48]
```

■

(2) Plot the logarithms of the importance weights as a histogram. Is the distribution of these log-importance-weights symmetric? Do you occasionally get extremely large importance weights? Extremely small ones? Which type of outliers is more worrisome?

Solution.

```python
import plotly.graph_objects as go

fig = go.Figure()
fig.add_trace(go.Histogram(
    x=w, xbins=dict(size=0.1), opacity=0.6
))
fig.update_xaxes(type="log")
fig.update_layout(
    xaxis_title_text='Importance Weights', # xaxis label
    yaxis_title_text='Count',              # yaxis label
    bargap=0.01,                           # gap between bars
    width=360, height=240, template="seaborn",
    margin=dict(l=5, r=5, t=5, b=5),
)
fig.show()
fig.write_image("01-02_importance-weight.pdf")
```

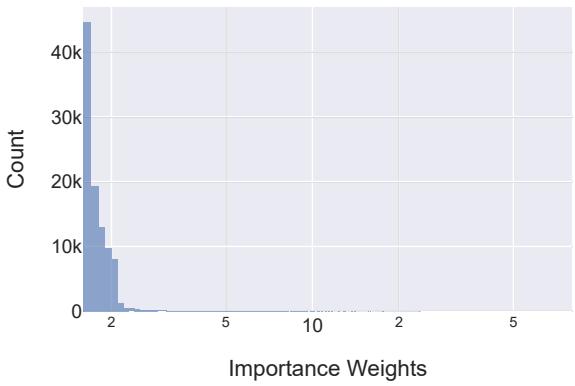

Figure 1.1: The distribution of log-importance-weights

The results show that the distribution of log-importance-weights is asymmetric. The outliers that have larger value, which lead to high variance, are more worrisome. ■

(3) Compare your Monte Carlo estimates with the true values. Are there biases in the Monte Carlo estimates? How large are the Monte Carlo standard derivations? (Is there a theoretical formula for the standard deviation of your Monte Carlo estimator?)

Solution. The Monte Carlo estimates are unbiased. Standard derivations are given as follows.

```
print(f"{(x * w).std():.2f}, {((x ** 2.0) * w).std():.2f}, {((x > 2.0) * w).std():.2f}")
```

```
3.77, 13.78, 0.76
```

Let $I_n^{\text{IS}} := \widehat{\mathbb{E}}_f\left[h(X)\right] = \widehat{\mathbb{E}}_q\left[h(X)w(X)\right] = \frac{1}{n}\sum_{i=1}^{n} h(x_i)w(x_i)$, we have

$$\begin{aligned}
\mathbb{E}_q\left[I_n^{\text{IS}}\right] &= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}_q\left[h(X_i)w(X_i)\right] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}_f\left[h(X_i)\right] = \mathbb{E}_f\left[h(X)\right] =: I \\
\mathbb{V}\text{ar}(I_n^{\text{IS}}) &= \frac{1}{n}\mathbb{V}\text{ar}_q\big(h(X)w(X)\big) = \frac{1}{n}\left(\sqrt{\tfrac{\pi}{8}}\int_{\mathbb{R}}(h(x))^2\exp\left(\tfrac{1}{2}x^2 - 2\left|x\right|\right)\mathrm{d}x - I^2\right).
\end{aligned}$$

Then

$$\begin{aligned}
\mathbb{V}\text{ar}(\widehat{\mathbb{E}}(X)) &= \frac{1}{n}\sqrt{\tfrac{\pi}{8}}\int_{\mathbb{R}} x^2\exp\left(\tfrac{1}{2}x^2 - 2\left|x\right|\right)\mathrm{d}x = \infty \\
\mathbb{V}\text{ar}(\widehat{\mathbb{V}\text{ar}}(X)) &= \frac{1}{n}\left(\sqrt{\tfrac{\pi}{8}}\int_{\mathbb{R}} x^4\exp\left(\tfrac{1}{2}x^2 - 2\left|x\right|\right)\mathrm{d}x - 4\right) = \infty \\
\mathbb{V}\text{ar}(\widehat{\mathbb{P}}(X>2)) &= \frac{1}{n}\left(\sqrt{\tfrac{\pi}{8}}\int_{x>2}\exp\left(\tfrac{1}{2}x^2 - 2\left|x\right|\right)\mathrm{d}x - \tfrac{1}{4}e^{-4}\right) = \infty.
\end{aligned}$$

■

(4) Let Y have the standard normal distribution. Use *importance sampling* based on 100000 draws from the standard Laplace as the proposal density to estimate $\mathbb{E}(Y)$, $\mathbb{V}ar(Y)$ and $\mathbb{P}(Y > 2)$. Repeat parts (2) and (3).

Solution.

```
np.random.seed(1234)
x, w, EX, VarX, P = is_est(reverse=True)

df = pd.DataFrame({
    "E(Y)"  :    ["0", f"{EX:.4f}"],
    "Var(Y)":    ["1", f"{VarX:.4f}"],
    "P(Y>2)":    [f"{1-stats.norm.cdf(2):.4f}", f"{P:.4f}"],
    }, index=["Truth", "Estimate"]
)
display(df)
print(f"Weights: [{w.min():.2f}, {w.max():.2f}]")

fig = go.Figure()
fig.add_trace(go.Histogram(
    x=np.log(w), xbins=dict(size=1), opacity=0.6
))
# fig.update_xaxes(type="log")
```

```python
fig.update_layout(
    xaxis_title_text='Importance Weights', # xaxis label
    yaxis_title_text='Count',              # yaxis label
    bargap=0.05,                           # gap between bars
    width=360, height=240, template="seaborn",
    margin=dict(l=5, r=5, t=5, b=5),
)
fig.show()
fig.write_image("01-02_importance-weight-reverse.pdf")
print(f"{(x * w).std():.2f}, {((x ** 2.0) * w).std():.2f}, {((x > 2.0) * w).std():.2f}")
```

```
            E(y)     Var(y)     P(y>2)
Truth       0        1          0.0228
Estimate   -0.0002   1.0007     0.0228
Weights: [0.00, 1.32]
0.99, 1.08, 0.11
```

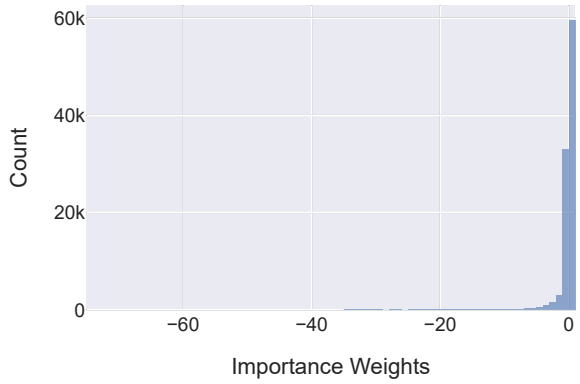

Figure 1.2: The distribution of log-importance-weights

$$\mathbb{V}\mathrm{ar}(\widehat{\mathbb{E}}(X)) = \tfrac{2}{n\pi}\int_{\mathbb{R}_+} x^2 \exp\left(-x^2+x\right)\mathrm{d}x \approx 0.0031^2 \approx \tfrac{0.99^2}{n}$$
$$\mathbb{V}\mathrm{ar}(\widehat{\mathbb{V}\mathrm{ar}}(X)) = \tfrac{1}{n}\left(\tfrac{2}{\pi}\int_{\mathbb{R}_+} x^4 \exp\left(-x^2+x\right)\mathrm{d}x - 1\right) \approx 0.0034^2 \approx \tfrac{1.08^2}{n}$$
$$\mathbb{V}\mathrm{ar}(\widehat{\mathbb{P}}(X>2)) = \tfrac{1}{n}\left(\tfrac{1}{\pi}\int_{x>2} \exp\left(-x^2+x\right)\mathrm{d}x - 0.0228^2\right) \approx 0.00035^2 \approx \tfrac{0.11^2}{n}.$$

■

2 Problem 2

Consider the following Restricted Boltzmann Machine (RBM) with energy function

$$p_\theta(v) = \frac{1}{Z_\theta}\sum_h \exp(-E(v,h)), \quad E(v,h) = -b^\mathsf{T}v - c^\mathsf{T}h - h^\mathsf{T}Wv$$

Here the model parameters are $\theta = \{b, c, W\}$

(1) Show that $p(v \mid h) = \prod_{i=1}^n p\left(v_i \mid h\right), p(h \mid v) = \prod_{i=1}^d p\left(h_i \mid v\right)$

Solution. It suffices to show the first identity since we have the second by symmetry.

$$\begin{aligned}
p(v,h) &= \frac{p(v,h)}{p(h)} \\
&= \frac{1}{p(h)} \frac{\exp(b^\mathsf{T} v + c^\mathsf{T} h + h^\mathsf{T} W v)}{\sum_u \exp(b^\mathsf{T} v + c^\mathsf{T} h + h^\mathsf{T} W u)} \\
&= \frac{1}{p(h)} \frac{\prod_i \exp(b_i v_i + c^\mathsf{T} h + h^\mathsf{T} W_{:,i} v_i)}{\sum_u \exp(b^\mathsf{T} v + c^\mathsf{T} h + h^\mathsf{T} W u)} \\
&= \frac{\prod_j \sum_{u_j} \exp(b_j u_j + c^\mathsf{T} h + h^\mathsf{T} W_{:,j} u_j)}{\sum_u \exp(b^\mathsf{T} v + c^\mathsf{T} h + h^\mathsf{T} W u)} \prod_i p(v_i \mid h) \\
&= \prod_i p(v_i \mid h)
\end{aligned}$$

■

(2) Derive the derivatives of the log-likelihood function w.r.t. the model parameters θ

Solution.

$$\begin{aligned}
L(\theta) &= \sum_i \log p(v^{(i)}) \\
\frac{\partial L}{\partial \theta} &= \sum_i \frac{\partial \log p(v^{(i)})}{\partial \theta} \\
&= \sum_i \tfrac{\partial}{\partial \theta} \Big(\log \sum_h \exp(-E(v^{(i)}, h)) - \log \sum_{v,h} \exp(-E(v,h)) \Big) \\
&= -\sum_i \frac{\sum_h \exp(-E(v^{(i)}, h) \tfrac{\partial}{\partial \theta} E(v^{(i)}, h)}{\sum_h \exp(-E(v^{(i)}, h))} + \sum_i \frac{\sum_{v,h} \exp(-E(v,h) \tfrac{\partial}{\partial \theta} E(v,h)}{\sum_{v,h} \exp(-E(v,h))} \\
&= -\sum_i \sum_h p(h \mid v^{(i)}) \tfrac{\partial}{\partial \theta} E(v^{(i)}, h) + \sum_i \sum_{v,h} p(v,h) \tfrac{\partial}{\partial \theta} E(v,h)
\end{aligned}$$

where

$$\begin{aligned}
\sum_h p(h \mid v) \tfrac{\partial E(v,h)}{\partial b_i} &= -\sum_h p(h \mid v) v_i = -v_i \\
\sum_h p(h \mid v) \tfrac{\partial E(v,h)}{\partial c_i} &= -\sum_h \sum_{k=1}^{d} p(h_k \mid v) h_i = -\sum_{h_i} p(h_i \mid v) h_i = -p(h_i = 1 \mid v) \\
\sum_h p(h \mid v) \tfrac{\partial E(v,h)}{\partial W_{i,j}} &= -\sum_h \sum_{k=1}^{d} p(h_k \mid v) h_i v_j = -\sum_{h_i} p(h_i \mid v) h_i v_j = -p(h_i = 1 \mid v) v_j,
\end{aligned}$$

then

$$\begin{aligned}
\frac{\partial L}{\partial b_i} &= \sum_k \Big(v_i^{(k)} - \sum_v p(v) v_i \Big) \\
\frac{\partial L}{\partial c_i} &= \sum_k \Big(p(h_i = 1 \mid v^{(k)}) - \sum_v p(v) p(h_i = 1 \mid v) \Big) \\
\frac{\partial L}{\partial W_{i,j}} &= \sum_k \Big(p(h_i = 1 \mid v^{(k)}) v_j^{(k)} - \sum_v p(v) p(h_i = 1 \mid v) v_j \Big)
\end{aligned}$$

■

(3) Use the following code to load the MNIST data set.

```
from sklearn.datasets import fetch_openml

X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
X = (X/255).astype('float32')
X_train , X_test = X[:60000,:], X[60000:,:]
```

Train your RBM on the training data set using contrastive divergence ($k = 1$), with Gibbs sampling for the energy induced distribution. Report the reconstruction error $\|v - \tilde{v}\|^2$ on the training data and the test data as a function of the number of iterations, where $\tilde{v}$ is the sample after $k = 1$ iteration of Gibbs sampling that starts at v. Do you have any interesting finding? Explain it.

Solution. Here we refer to the `Learnergy` framework proposed by Roder *et al.* (2020)[1].

```python
import time
import torch
import torchvision
from torch.utils.data import DataLoader
from tqdm.notebook import tqdm
from learnergy.models.bernoulli import RBM
# import learnergy.utils.logging as l
# logger = l.get_logger(__name__)

class MyRBM(RBM):
    def __init__(self, **kwargs):
        super(MyRBM, self).__init__(**kwargs)

    def train_func(
        self, train_dataset, test_dataset, batch_size=64, epochs=10
        ):
        batches = DataLoader(train_dataset, batch_size=batch_size,
                             shuffle=True, num_workers=0)
        for epoch in range(epochs):
            # logger.info('Epoch %d/%d', epoch+1, epochs)
            print(f'Epoch {epoch}')
            start = time.time()
            mse = 0
            pl = 0
            for samples, _ in tqdm(batches):
                samples = samples.reshape(len(samples), self.n_visible)
                if self.device == 'cuda':
                    samples = samples.cuda()
                _, _, _, _, visible_states = self.gibbs_sampling(samples)
                visible_states = visible_states.detach()
                cost = (
                    torch.mean(self.energy(samples))
                    - torch.mean(self.energy(visible_states))
                    )
                self.optimizer.zero_grad()
                cost.backward()
                self.optimizer.step()
                batch_size = samples.size(0)
                batch_mse = torch.div(
```

[1] See https://github.com/gugarosa/learnergy/blob/master/learnergy/models/bernoulli/rbm.py

```
                    torch.sum(torch.pow(samples - visible_states, 2)), batch_size).detach()
                batch_pl = self.pseudo_likelihood(samples).detach()
                mse += batch_mse
                pl += batch_pl
            mse /= len(batches)
            pl /= len(batches)
            mse_test, _ = self.reconstruct(test_dataset)
            end = time.time()
            self.dump(mse=mse.item(), mse_test=mse_test.item(),
                      pl=pl.item(), time=end-start)
            # logger.info('MSE: %f | MSE_test: %f| log-PL: %f', mse, mse_test, pl)

if __name__ == "__main__":
    import plotly.graph_objects as go
    # mnist data
    train = torchvision.datasets.MNIST(
        root='../dataset/mnist', train=True, download=True,
        transform=torchvision.transforms.ToTensor()
    )
    test = torchvision.datasets.MNIST(
        root='../dataset/mnist', train=False, download=True,
        transform=torchvision.transforms.ToTensor()
    )
    # RBM
    model = MyRBM(n_visible=784, n_hidden=128, steps=1, learning_rate=0.1,
                  momentum=0, decay=0, use_gpu=True)
    # Training
    n_epochs = 500
    model.train_func(train, test, batch_size=128, epochs=n_epochs)
    # Saving
    torch.save(model, 'model.pth')
    mse = model.history["mse"]
    mse_test = model.history["mse_test"]

    fig = go.Figure()
    fig.add_trace(go.Scatter(
        y=mse, name="Training Error",
        mode='lines+markers', opacity=0.6,
    ))
    fig.add_trace(go.Scatter(
        y=mse_test, name="Testing Error",
        mode='lines+markers', opacity=0.6,
    ))
    fig.update_layout(
        xaxis_title_text='epoch',
```

```
        yaxis_title_text='Reconstruction Error',
        width=700, height=240, template="seaborn",
        margin=dict(l=5, r=5, t=5, b=5),
    )
    fig.show()
    fig.write_image("02-03_reconstruction-error.pdf")
```

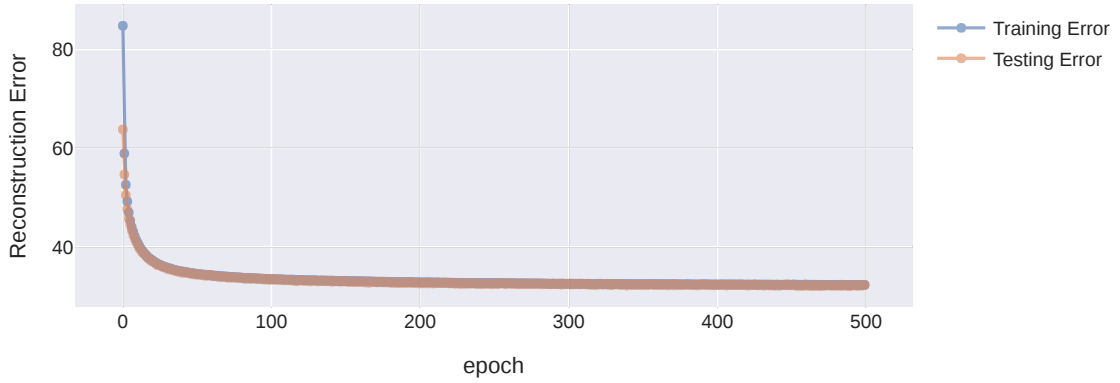

Figure 2.3: Reconstruction Errors

■

(4) Generate samples from your trained RBM using Gibbs sampling with 10 independent chains, and each chain runs 200 iterations. Show the results in a 10×11 grid plot where columns correspond to samples at every 20 iterations.

Solution.

```
import torch
import plotly.graph_objs as go
from plotly.subplots import make_subplots

torch.manual_seed(1234)
rows, cols = 10, 11
batches = torch.FloatTensor(rows, 1, 28, 28).normal_(0, 256)

fig = make_subplots(
    rows=rows, cols=cols,
    shared_xaxes="all", shared_yaxes="all",
)
for row, data in zip(range(1, rows+1), batches):
    v = data.to("cuda").reshape(1, model.n_visible)
    pos_hidden_probs, pos_hidden_states = model.hidden_sampling(v)
    neg_hidden_states = pos_hidden_states
    fig.add_trace(go.Heatmap(
        z=v.reshape(28, 28).detach().cpu().numpy(),
        colorscale='Viridis', showscale = False
        ), row=row, col=1,
```

```
    )
    for i in range(1, 201):
        visible_probs, visible_states = model.visible_sampling(
            neg_hidden_states, False
            )
        neg_hidden_probs, neg_hidden_states = model.hidden_sampling(
            visible_states, False
            )
        if i % 20 == 0:
            col = i//20+1
            fig.add_trace(go.Heatmap(
                z=visible_probs.reshape(28, 28).detach().cpu().numpy(),
                colorscale='Viridis', showscale = False
                ), row=row, col=col,
            )
            fig.update_xaxes(title_text=f"{i}", row=rows, col=col)
    # fig.update_yaxes(title_text=f"{label.item()}", row=row, col=1)

fig.update_xaxes(showticklabels=False)
fig.update_yaxes(autorange='reversed', showticklabels=False)
fig.update_layout(
    height=600, width=600,
    template = "seaborn",
    margin=dict(l=5, r=5, t=5, b=5),
)
fig.show()
fig.write_image("02-04_Gibbs-sampling.pdf")
```

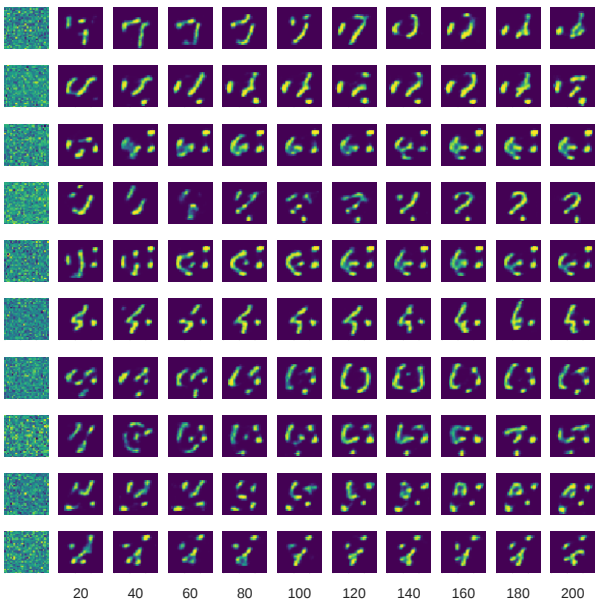

Figure 2.4: Gibbs sampling

■

3 Problem 3

Consider a logistic regression model with normal priors

$$y_i \sim \text{Bernoulli}\left(p_i\right), p_i = \frac{1}{1+\exp\left(-x_i^\mathsf{T}\beta\right)}, \quad i = 1, \ldots, n. \quad \beta \sim \mathcal{N}\left(0, \sigma_\beta^2\right)$$

where $\sigma_\beta = 1$. Download the data from the course website.

(1). Implement a Hamiltonian Monte Carlo sampler to collect 500 samples (with 500 discarded as burn-in), show the scatter plot. Test the following two strategies for the number of leapfrog steps L: (1) use a fixed L ;(2) use a random one, say Uniform $(1, L_{\max})$ Do you find any difference? Explain it.

Solution. The posterior distribution of β is

$$p(\beta \mid y, X) \propto p(y \mid \beta, X)p(\beta) \propto \exp\left(-\tfrac{1}{2}\left\|\beta\right\|_2^2\right) \prod_{i=1}^{n} p_i^{y_i}\left(1-p_i\right)^{1-y_i}.$$

The potential energy function is

$$\begin{aligned}
U(\beta) &= \tfrac{1}{2}\left\|\beta\right\|_2^2 - \log \prod_{i=1}^{n} p_i^{y_i}\left(1-p_i\right)^{1-y_i} + \text{constant} \\
&= \tfrac{1}{2}\left\|\beta\right\|_2^2 - \sum_{i=1}^{n}\left[y_i \log p_i + \left(1-y_i\right)\log\left(1-p_i\right)\right] + \text{constant} \\
&= \tfrac{1}{2}\left\|\beta\right\|_2^2 + \sum_{i=1}^{n}\left[\left(1-y_i\right)x_i^\mathsf{T}\beta + \log\left(1+\exp\left(-x_i^\mathsf{T}\beta\right)\right)\right] + \text{constant} \\
&= \tfrac{1}{2}\left\|\beta\right\|_2^2 + \left(X\beta\right)^\mathsf{T}\left(1-y\right) + 1^\mathsf{T}\log\left(1+\exp\left(-X\beta\right)\right) + \text{constant}.
\end{aligned}$$

The Hamilonian is $H(\beta, r) = U(\beta) + K(r)$ where $K(r) = \frac{1}{2}\left\|r\right\|_2^2$. Then,

$$\begin{aligned}
\frac{\partial H}{\partial \beta} &= \frac{\partial U}{\partial \beta} = \beta + \sum_{i=1}^{n}\left(p_i - y_i\right)x_i = \beta + X^\mathsf{T}\left(\tfrac{1}{1+\exp(-X\beta)} - y\right) \\
\frac{\partial H}{\partial r} &= \frac{\partial K}{\partial r} = r
\end{aligned}$$

```python
import numpy as np
from scipy.special import expit as sigmoid

data = np.load("./mcs_hw2_p3_data.npy")
# X, y = data[:, :2], data[:, -1]

class HamiltonianLogisticRegression:
    def __init__(self, data):
        self.X, self.y = data[:, :2], data[:, -1]

    def potential_energy(self, beta):
        """ U(\beta), potential energy
        """
        X, y = self.X, self.y
        Xbeta = X@beta
```

```
        return (
            0.5*(beta@beta) + Xbeta@(1-y) - np.sum(np.log(sigmoid(Xbeta)))
        )

    def kinetic_energy(self, r):
        """ K(r), Euclidean-Gaussain kinetic energy
        """
        return 0.5*(r@r)

    def potential_energy_grad(self, beta):
        X, y = self.X, self.y
        return beta + X.T@(sigmoid(X@beta)-y)

    def kinetic_energy_grad(self, r):
        return r

    def hamiltonian(self, beta, r):
        return self.potential_energy(beta) + self.kinetic_energy(r)

    def leapfrog(self, beta0, r0, n_leap_steps, leap_size):
        for i in range(n_leap_steps):
            r1 = r0 - 0.5 * leap_size * self.potential_energy_grad(beta0)
            beta2 = beta0 + leap_size * self.kinetic_energy_grad(r1)
            r2 = r1 - 0.5 * leap_size * self.potential_energy_grad(beta2)
            beta0, r0 = beta2.copy(), r2.copy()
        return beta0, r0

    def hamiltonian_monte_carlo(
        self, n_samples, n_discards, n_leapfrog_steps,
        leap_size=0.1, fixed=True,
        ):
        dim = self.X.shape[-1]
        n_leap_steps = n_leapfrog_steps
        samples = np.zeros((n_samples + n_discards, dim))
        beta = np.random.randn(dim)
        for i in range(n_samples + n_discards):
            if not fixed:
                n_leap_steps = np.random.randint(1, n_leapfrog_steps+1)
            r = np.random.normal(size=dim)
            beta0, r0 = self.leapfrog(
                beta0=beta, r0=r, n_leap_steps=n_leap_steps, leap_size=leap_size,
            )
            dH = self.hamiltonian(beta, r) - self.hamiltonian(beta0, r0)
            if np.random.random() < min(1, np.exp(dH)):
                beta = beta0.copy()
```

```
            samples[i] = beta
        return samples[n_discards:-1, :]

if __name__ == "__main__":
    np.random.seed(1234)
    hmc = HamiltonianLogisticRegression(data=data)
    n_samples = n_discards = 500
    sample_fix_steps = hmc.hamiltonian_monte_carlo(
        n_samples, n_discards, leap_size=0.005, n_leapfrog_steps=4,
    )
    sample_rand_steps = hmc.hamiltonian_monte_carlo(
        n_samples, n_discards, leap_size=0.005, n_leapfrog_steps=8, fixed=False
    )

    import plotly.graph_objects as go
    fig = go.Figure()
    fig.add_trace(go.Scatter(
        x=sample_fix_steps[:,0],y=sample_fix_steps[:,1], name="L = 4",
        mode='markers', opacity=0.9, marker_symbol="circle-open",
    ))
    fig.add_trace(go.Scatter(
        x=sample_rand_steps[:,0],y=sample_rand_steps[:,1], name="L ~ U(1,8)",
        mode='markers', opacity=0.9, marker_symbol="square-open",
    ))
    fig.update_layout(
        width=700, height=480, template="plotly_white",
        margin=dict(l=5, r=5, t=5, b=5),
    )
    fig.show()
    fig.write_image("03-01_hamiltonian-monte-carlo.pdf")
```

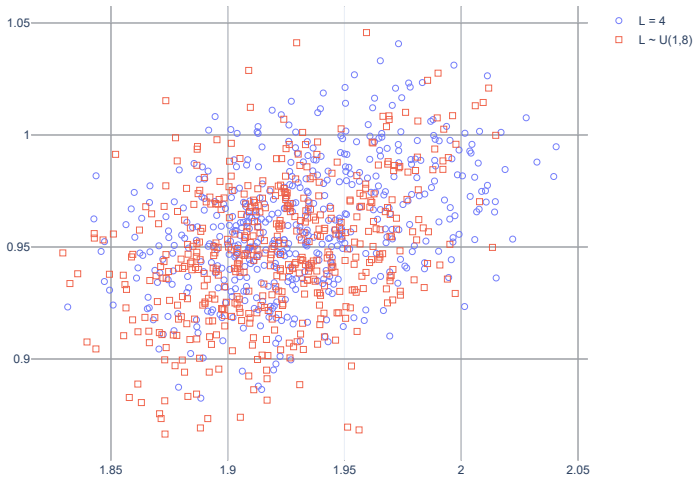

Figure 3.5: Hamiltonian Monte Carlo

■

(2). Run HMC for 100000 iterations and discard the first 50000 samples as burn-in to form the ground truth. Implement stochastic gradient MCMC algorithms including SGLD, SGHMC and SGNHT. Show the convergence rate of different SGMCMC algorithms in terms of KL divergence to the ground truth as a function of iterations. You may want to use the ITE package https://bitbucket.org/szzoli/ite-in-python/src/master/ to compute the KL divergence between two samples.

Solution.

```python
class SGMCMC(HamiltonianLogisticRegression):
    def __init__(self, **kwargs):
        super(SGMCMC, self).__init__(**kwargs)

    def potential_energy_sgrad(self, beta, batch_size):
        n_data = self.X.shape[0]
        indices = np.random.randint(0, n_data, size=batch_size)
        X, y = self.X[indices], self.y[indices]
        return beta + (n_data/batch_size)*X.T@(sigmoid(X@beta)-y)

    def sgld(
        self, n_samples, batch_size=64,
        truth=None, kl_divergence=None, interval=1,
        ):
        n_data, dim = self.X.shape[0], self.X.shape[1]
        beta = np.random.randn(dim)
        samples, kl_paths = [], {"i":[], "v":[]}

        for i in range(n_samples):
            eps = 1/(((i//100)+10)*20)
            grad = self.potential_energy_sgrad(beta, batch_size)
            beta1 = beta - 0.5*eps*grad + np.random.normal(0, eps, dim)
            dU = self.potential_energy(beta)-self.potential_energy(beta1)
            if dU >= 0 or np.random.rand() < np.exp(dU):
                beta = beta1
            samples.append(beta.copy())
            if (i > 10) and (i % interval == 0):
                samples_arr = np.array(samples)
                kl_distance = kl_divergence(truth, samples_arr[:i])
                print([i, kl_distance])
                kl_paths["i"].append(i)
                kl_paths["v"].append(kl_distance)
        return np.array(samples), kl_paths

    def sghmc(
        self, n_samples, n_leapfrog_steps=4, batch_size=64,
        truth=None, kl_divergence=None, interval=1,
```

```
        ):
        n_data, dim = self.X.shape[0], self.X.shape[1]
        beta = np.random.randn(dim)
        samples, kl_paths = [], {"i":[], "v":[]}

        for i in range(n_samples):
            eps = 1/(((i//1000)+10)*20)
            r = np.random.normal(0,1,2)
            for j in range(n_leapfrog_steps):
                beta += eps*r
                grad = self.potential_energy_sgrad(beta, batch_size)
                r += -eps*grad - eps*r+np.random.normal(0, 2*eps, 2)
            samples.append(beta.copy())
            if (i > 10) and (i % interval == 0):
                samples_arr = np.array(samples)
                kl_distance = kl_divergence(truth, samples_arr[:i])
                print([i, kl_distance])
                kl_paths["i"].append(i)
                kl_paths["v"].append(kl_distance)
        return np.array(samples), kl_paths

    def sgnht(
        self, n_samples, batch_size=64,
        truth=None, kl_divergence=None, interval=1,
        ):
        n_data, dim = self.X.shape[0], self.X.shape[1]
        beta, r = np.random.randn(dim), np.random.randn(dim)
        samples, kl_paths = [], {"i":[], "v":[]}
        A = 1
        xi = A
        for i in range(n_samples):
            eps = 1/(((i//1000)+10)*20)
            grad = self.potential_energy_sgrad(beta, batch_size)
            r = r - eps*grad - eps*xi*r + np.sqrt(2*A)*np.random.normal(0, eps)
            beta = beta + eps*r
            xi = xi + eps*(r@r/2-1)
            samples.append(beta)
            if (i > 10) and (i % interval == 0):
                samples_arr = np.array(samples)
                kl_distance = kl_divergence(truth, samples_arr[:i])
                print([i, kl_distance])
                kl_paths["i"].append(i)
                kl_paths["v"].append(kl_distance)
        return np.array(samples), kl_paths

```

```python

if __name__ == "__main__":
    import os
    import ite
    co = ite.cost.BDKL_KnnK()
    kl_divergence = co.estimation
    np.random.seed(1234)

    hmc = HamiltonianLogisticRegression(data=data)
    n_samples, n_discards = 100000, 50000
    if os.path.isfile(f"samples_hmc_{n_samples}.txt"):
        sample_hmc = np.loadtxt(f"samples_hmc_{n_samples}.txt")
    else:
        sample_hmc = hmc.hamiltonian_monte_carlo(
            n_samples, n_discards, leap_size=0.005, n_leapfrog_steps=4,
        )
        np.savetxt(f"samples_hmc_{n_samples}.txt", sample_fix_steps)

    sgmcmc = SGMCMC(data=data)
    batch_size = 64
    n_samples = 100000
    sample_sgld, kl_sgld = sgmcmc.sgld(
        n_samples,batch_size=batch_size,
        truth=sample_hmc, kl_divergence=kl_divergence, interval=5000,
    )
    sample_sghmc, kl_sghmc = sgmcmc.sghmc(
        n_samples, n_leapfrog_steps=3, batch_size=batch_size,
        truth=sample_hmc, kl_divergence=kl_divergence, interval=5000,
        )
    sample_sgnht, kl_sgnht = sgmcmc.sgnht(
        n_samples, batch_size=batch_size,
        truth=sample_hmc, kl_divergence=kl_divergence, interval=5000,
        )

    import plotly.graph_objects as go
    fig = go.Figure()
    fig.add_trace(go.Scatter(
        x=kl_sgld["i"], y=kl_sgld["v"], name="SGLD",
        mode='lines+markers', opacity=0.6,
    ))
    fig.add_trace(go.Scatter(
        x=kl_sghmc["i"], y=kl_sghmc["v"], name="SGHMC",
        mode='lines+markers', opacity=0.6,
    ))
    fig.add_trace(go.Scatter(
```

```
        x=kl_sgnht["i"], y=kl_sgnht["v"], name="SGNHT",
        mode='lines+markers', opacity=0.6,
    ))
    fig.update_layout(
        xaxis_title_text='Iterations',
        yaxis_title_text='KL-Divergence',
        width=700, height=240, template="seaborn",
        margin=dict(l=5, r=5, t=5, b=5),
    )
    fig.show()
    fig.write_image("03-02_KL-Divergence.pdf")
```

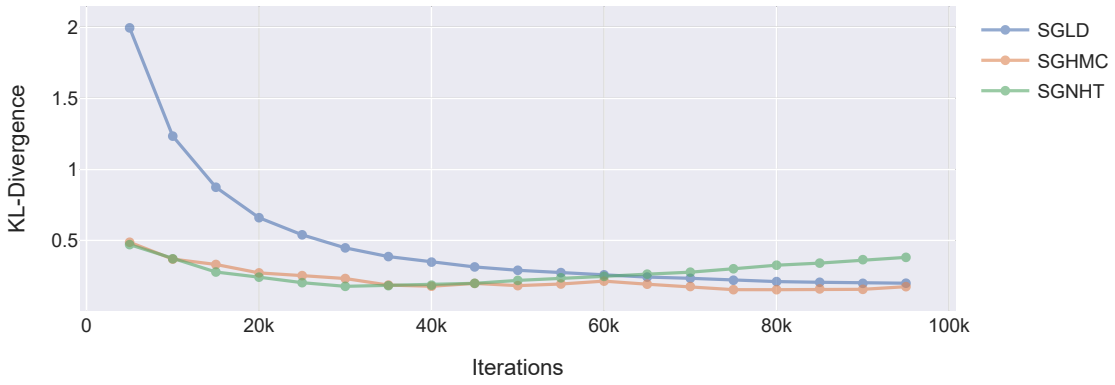

Figure 3.6: KL-Divergence

■

References

Roder, M., de Rosa, G. H. and Papa, J. P. (2020) Learnergy: Energy-based machine learners. 6