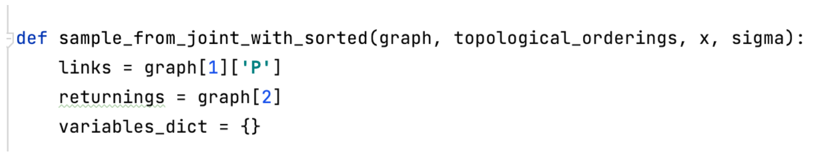
CPSC 532W - Homework 4

Xiaoxuan Liang - 48131163

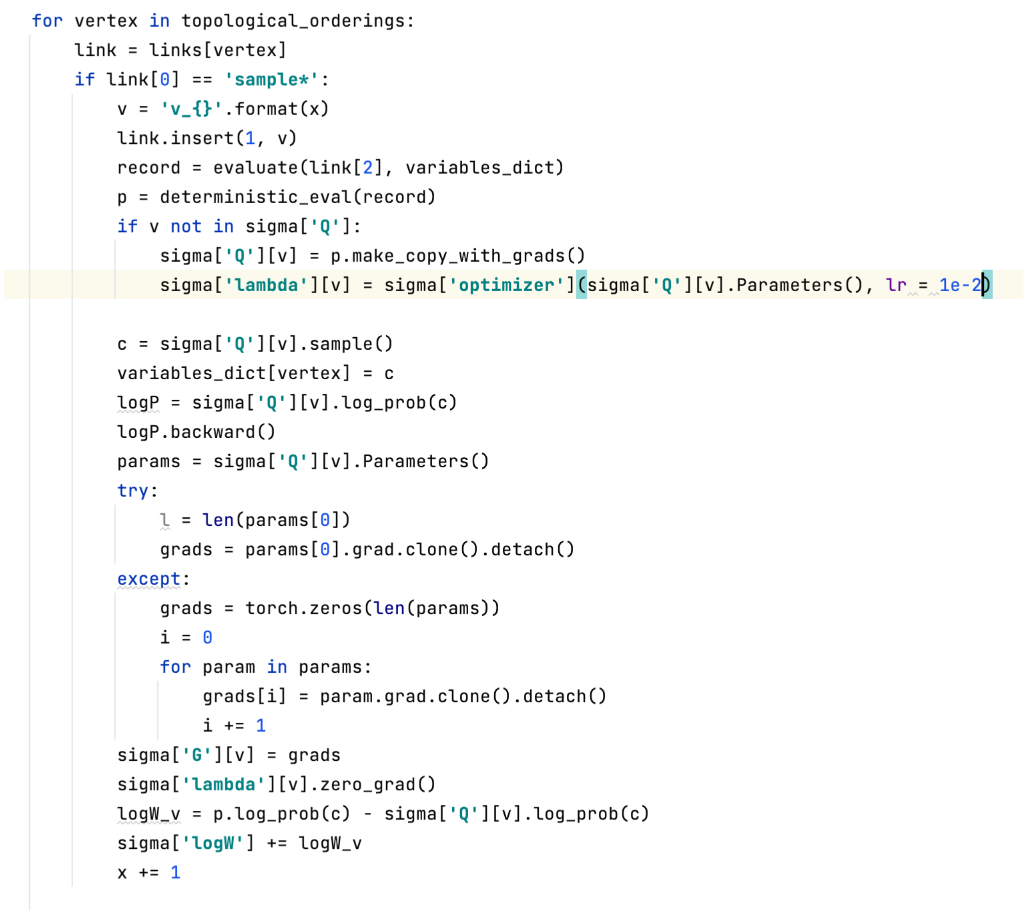
Public GitHub repo: https://github.com/Xiaoxuan1121/CPSC532W/tree/main/a4

1. Code snippets:



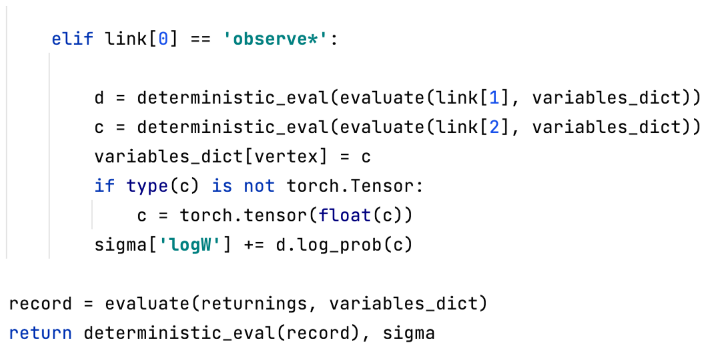
```
def sample_from_joint_with_sorted(graph, topological_orderings, x, sigma):
    links = graph[1]['P']
    returnings = graph[2]
    variables_dict = {}
```

Figure 1: code for graph evaluator part I



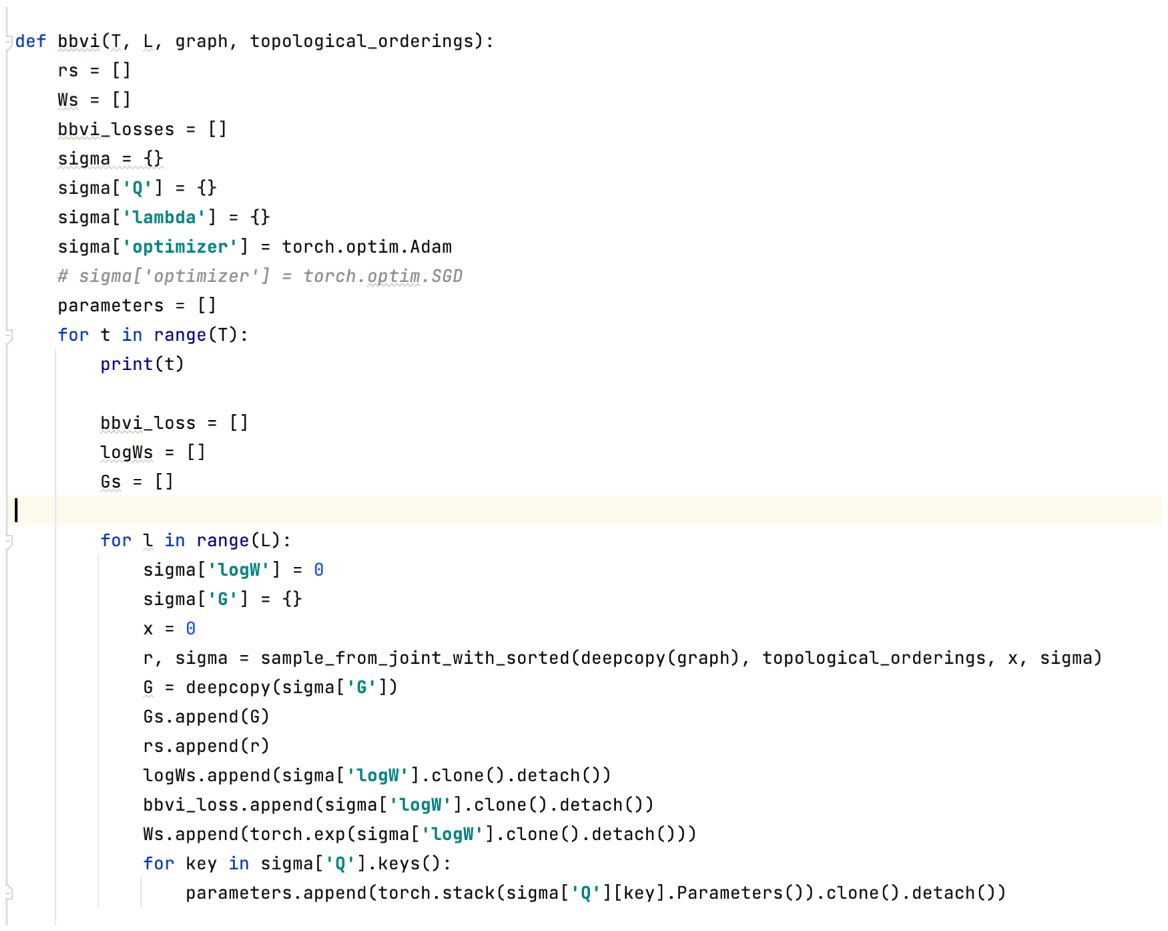
```
for vertex in topological_orderings:
    link = links[vertex]
    if link[0] == 'sample*':
        v = 'v_{}'.format(x)
        link.insert(1, v)
        record = evaluate(link[2], variables_dict)
        p = deterministic_eval(record)
        if v not in sigma['Q']:
            sigma['Q'][v] = p.make_copy_with_grads()
            sigma['lambda'][v] = sigma['optimizer'](sigma['Q'][v].Parameters(), lr = 1e-2)

        c = sigma['Q'][v].sample()
        variables_dict[vertex] = c
        logP = sigma['Q'][v].log_prob(c)
        logP.backward()
        params = sigma['Q'][v].Parameters()
        try:
            l = len(params[0])
            grads = params[0].grad.clone().detach()
        except:
            grads = torch.zeros(len(params))
            i = 0
            for param in params:
                grads[i] = param.grad.clone().detach()
                i += 1
        sigma['G'][v] = grads
        sigma['lambda'][v].zero_grad()
        logW_v = p.log_prob(c) - sigma['Q'][v].log_prob(c)
        sigma['logW'] += logW_v
        x += 1
```

Figure 2: code for graph evaluator part II



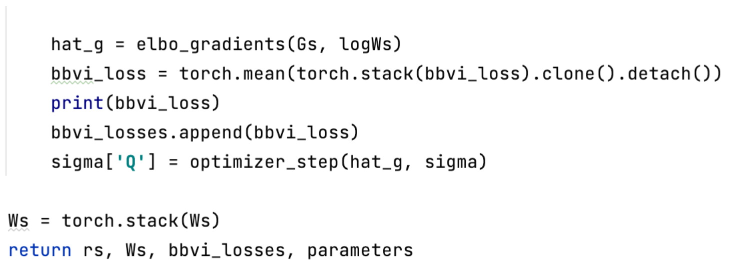
```python
    elif link[0] == 'observe*':

        d = deterministic_eval(evaluate(link[1], variables_dict))
        c = deterministic_eval(evaluate(link[2], variables_dict))
        variables_dict[vertex] = c
        if type(c) is not torch.Tensor:
            c = torch.tensor(float(c))
        sigma['logW'] += d.log_prob(c)

record = evaluate(returnings, variables_dict)
return deterministic_eval(record), sigma
```

Figure 3: code for graph evaluator part III



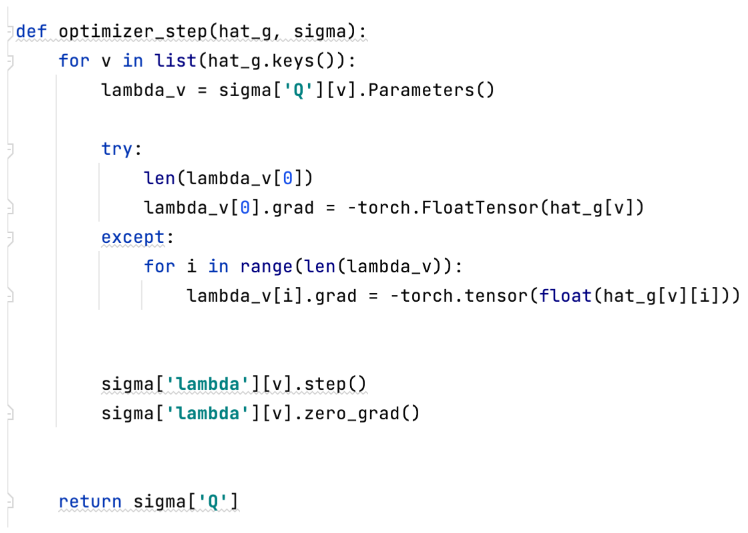
```python
def bbvi(T, L, graph, topological_orderings):
    rs = []
    Ws = []
    bbvi_losses = []
    sigma = {}
    sigma['Q'] = {}
    sigma['lambda'] = {}
    sigma['optimizer'] = torch.optim.Adam
    # sigma['optimizer'] = torch.optim.SGD
    parameters = []
    for t in range(T):
        print(t)

        bbvi_loss = []
        logWs = []
        Gs = []

        for l in range(L):
            sigma['logW'] = 0
            sigma['G'] = {}
            x = 0
            r, sigma = sample_from_joint_with_sorted(deepcopy(graph), topological_orderings, x, sigma)
            G = deepcopy(sigma['G'])
            Gs.append(G)
            rs.append(r)
            logWs.append(sigma['logW'].clone().detach())
            bbvi_loss.append(sigma['logW'].clone().detach())
            Ws.append(torch.exp(sigma['logW'].clone().detach()))
            for key in sigma['Q'].keys():
                parameters.append(torch.stack(sigma['Q'][key].Parameters()).clone().detach())
```

Figure 4: code for BBVI part I



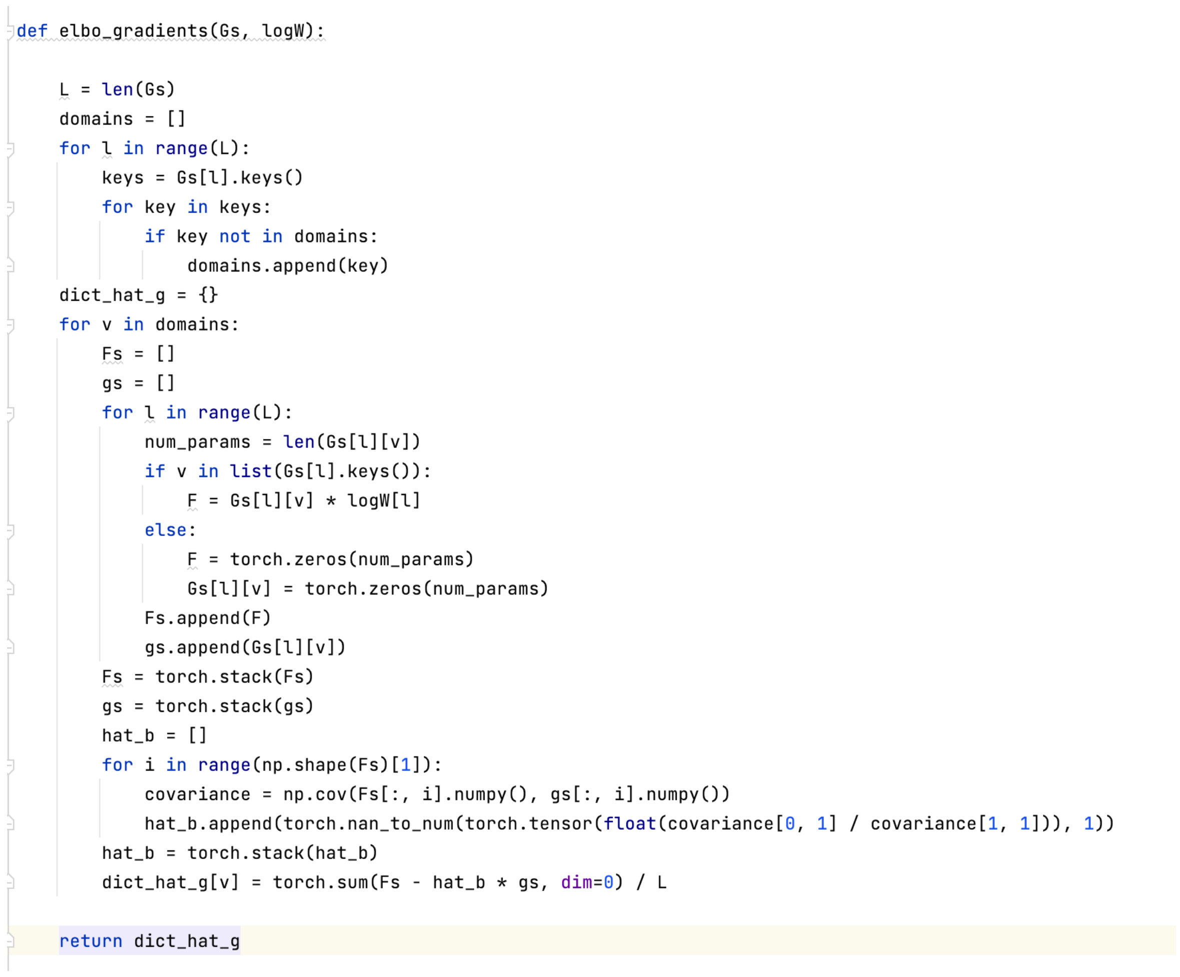
```
    hat_g = elbo_gradients(Gs, logWs)
    bbvi_loss = torch.mean(torch.stack(bbvi_loss).clone().detach())
    print(bbvi_loss)
    bbvi_losses.append(bbvi_loss)
    sigma['Q'] = optimizer_step(hat_g, sigma)

Ws = torch.stack(Ws)
return rs, Ws, bbvi_losses, parameters
```

Figure 5: code for BBVI part II



```
def optimizer_step(hat_g, sigma):
    for v in list(hat_g.keys()):
        lambda_v = sigma['Q'][v].Parameters()

        try:
            len(lambda_v[0])
            lambda_v[0].grad = -torch.FloatTensor(hat_g[v])
        except:
            for i in range(len(lambda_v)):
                lambda_v[i].grad = -torch.tensor(float(hat_g[v][i]))


        sigma['lambda'][v].step()
        sigma['lambda'][v].zero_grad()


    return sigma['Q']
```

Figure 6: code for BBVI part III



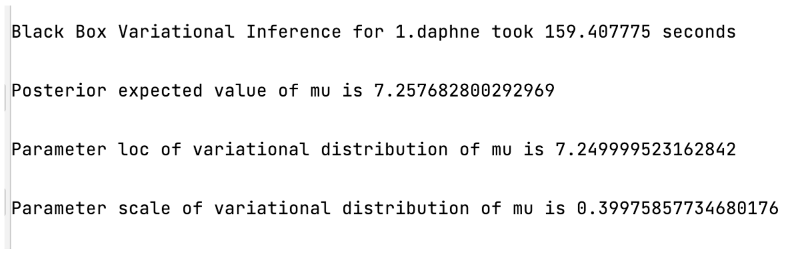
```python
def elbo_gradients(Gs, logW):

    L = len(Gs)
    domains = []
    for l in range(L):
        keys = Gs[l].keys()
        for key in keys:
            if key not in domains:
                domains.append(key)
    dict_hat_g = {}
    for v in domains:
        Fs = []
        gs = []
        for l in range(L):
            num_params = len(Gs[l][v])
            if v in list(Gs[l].keys()):
                F = Gs[l][v] * logW[l]
            else:
                F = torch.zeros(num_params)
                Gs[l][v] = torch.zeros(num_params)
            Fs.append(F)
            gs.append(Gs[l][v])
        Fs = torch.stack(Fs)
        gs = torch.stack(gs)
        hat_b = []
        for i in range(np.shape(Fs)[1]):
            covariance = np.cov(Fs[:, i].numpy(), gs[:, i].numpy())
            hat_b.append(torch.nan_to_num(torch.tensor(float(covariance[0, 1] / covariance[1, 1])), 1))
        hat_b = torch.stack(hat_b)
        dict_hat_g[v] = torch.sum(Fs - hat_b * gs, dim=0) / L

    return dict_hat_g
```

Figure 7: code for BBVI part IV

2. Program 1:

   The posterior expected value of mu is 7.2577.

   The variational distribution of μ is a Normal distribution with mean 7.25 and 0.3998.



```
Black Box Variational Inference for 1.daphne took 159.407775 seconds

Posterior expected value of mu is 7.257682800292969

Parameter loc of variational distribution of mu is 7.249999523162842

Parameter scale of variational distribution of mu is 0.39975857734680176
```

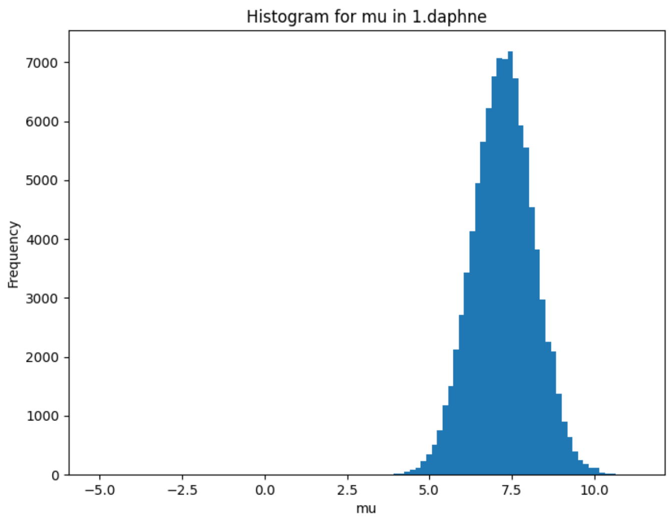


Figure 8: Posterior distribution for mu for 1.daphne using Black Box Variational Inference

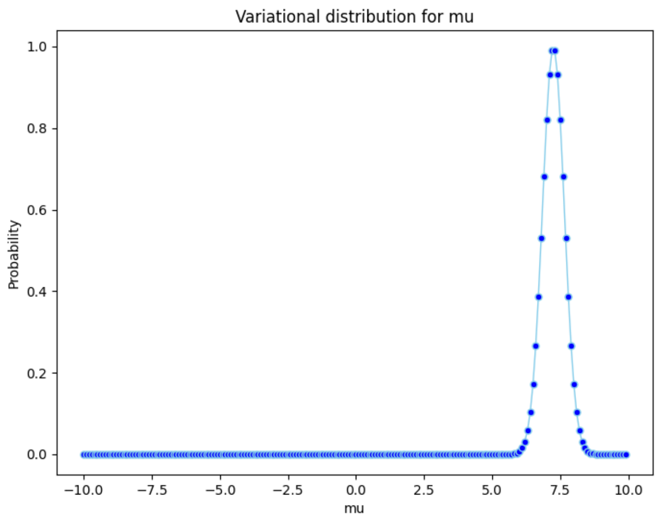


Figure 9: Variational distribution for mu for 1.daphne using Black Box Variational Inference

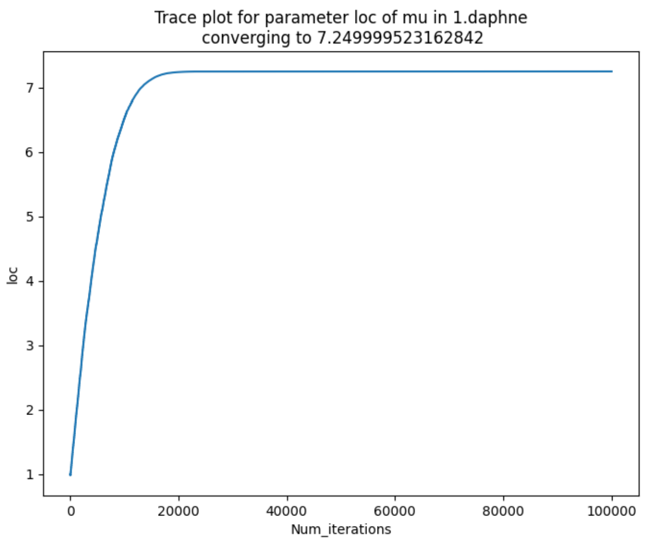


Figure 10: Trace plot for parameter loc of mu in 1.daphne

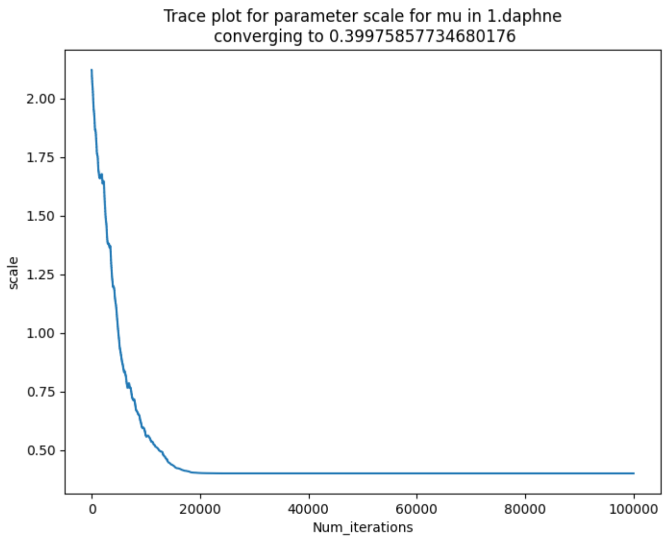


Figure 11: Trace plot for parameter scale of mu in 1.daphne

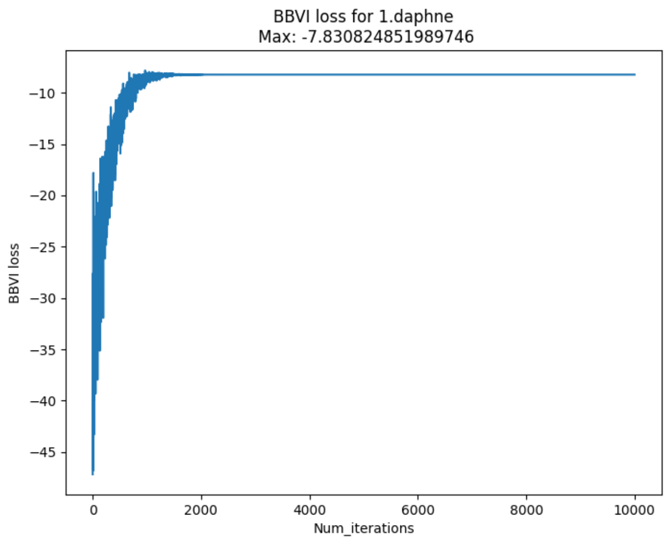
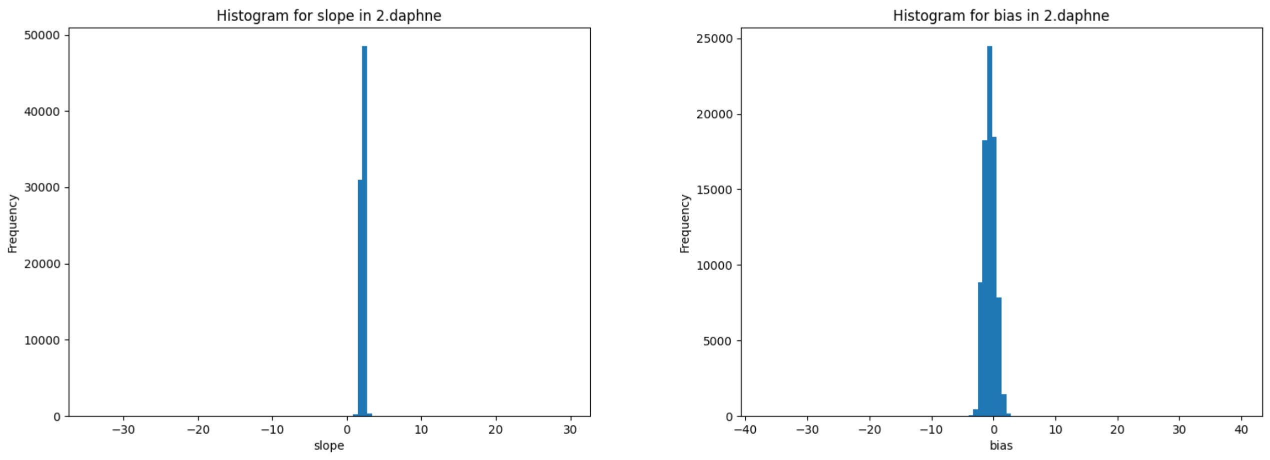


Figure 12: ELBO for 1.daphne

3. Program 2:

```
Black Box Variational Inference for 2.daphne took 314.351933 seconds

Posterior means for slope and bias for 2.daphne are 2.170492172241211, -0.5949399471282959
```



(a) Samples from the posterior for slope

(b) Samples from the posterior for bias

Figure 13: Posterior distribution for slope and bias for 2.daphne using Black Box Variational Inference

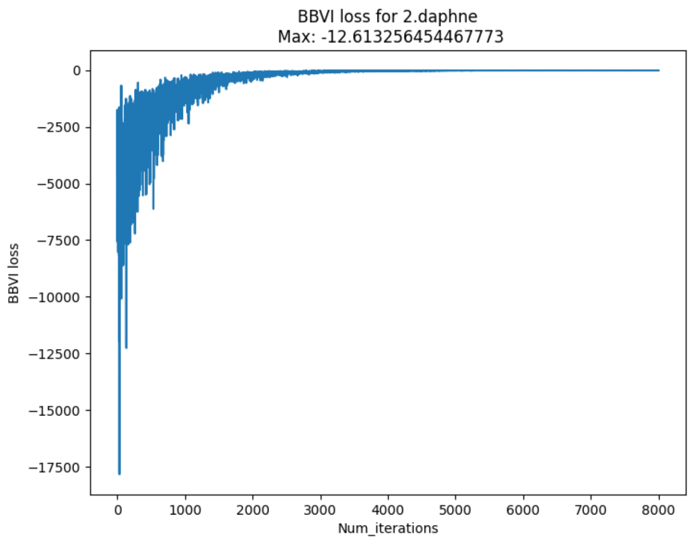
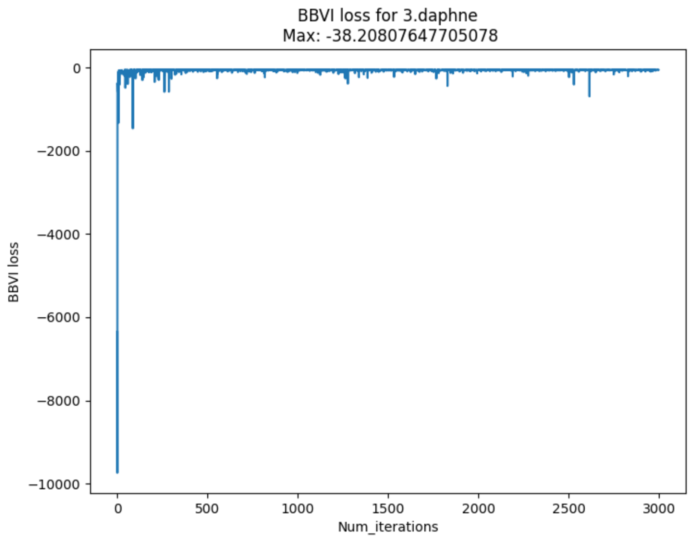


Figure 14: ELBO for 2.daphne

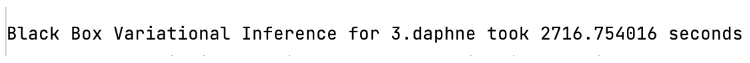
4. Program 3:

```
Black Box Variational Inference for 3.daphne took 419.316216 seconds

The posterior probability that the first and second datapoint are in the same cluster is 0.6049062609672546
```



The mode-seeking behaviour of VI on models would fall into different modes every time when repeating running the code with internal symmetries. Internal symmetries indicates the label switching problems, such that the class would be labelled (latent variables) by different notation since the labels are permuted randomly each time. This always happens in the mixture models, however, the model density is not changing even the labels are keeping switching among the classes. This might result in the inference analysis fall off sometimes and it will be back on the track after that, which canbe seen from the ELBO plot that the BBVI loss could suddenly drop down to a value and it will bounce back later.

5. Program 4:



```
Black Box Variational Inference for 3.daphne took 2716.754016 seconds
```

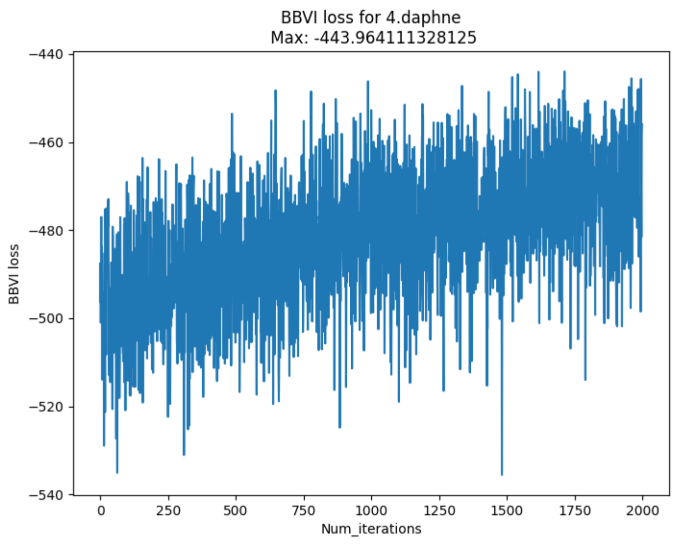
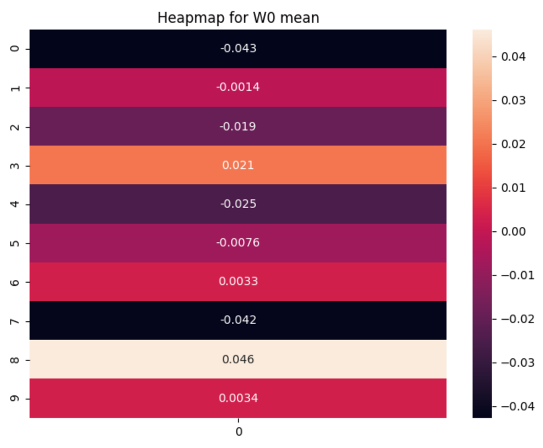
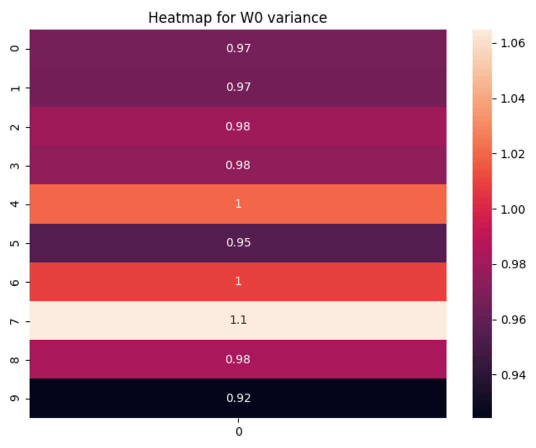


Figure 15: ELBO for 4.daphne

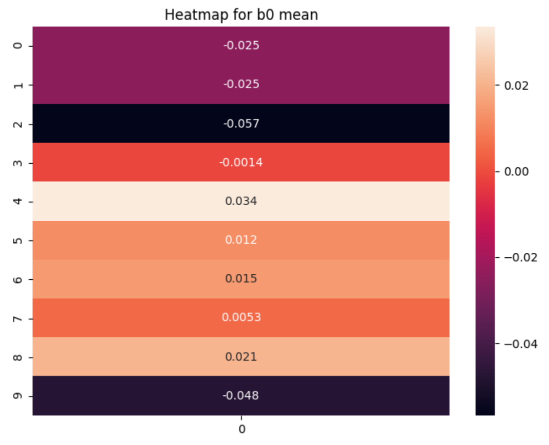


(a) Samples from the posterior for W0

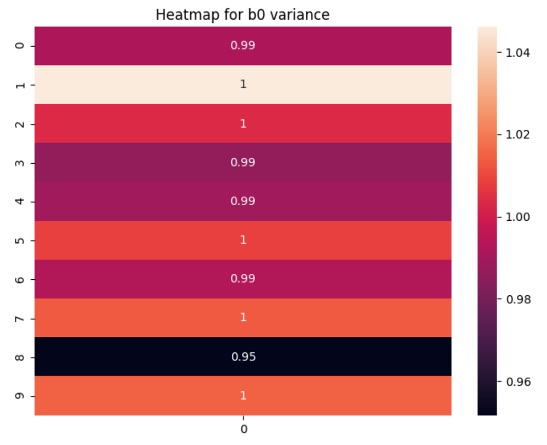


(b) Samples from the posterior for W0

Figure 16: Posterior distribution for slope and bias for 4.daphne using Black Box Variational Inference

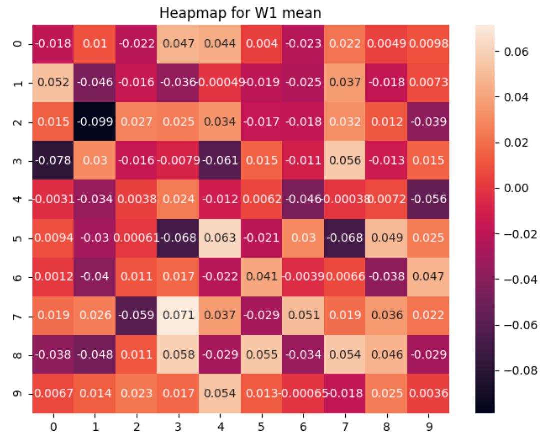


(a) Samples from the posterior for b0

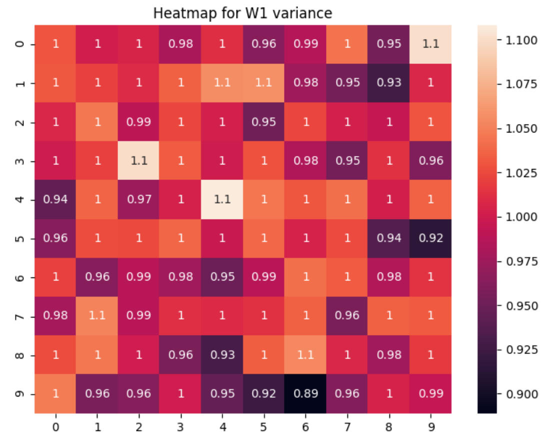


(b) Samples from the posterior for b0

Figure 17: Posterior distribution for slope and bias for 4.daphne using Black Box Variational Inference

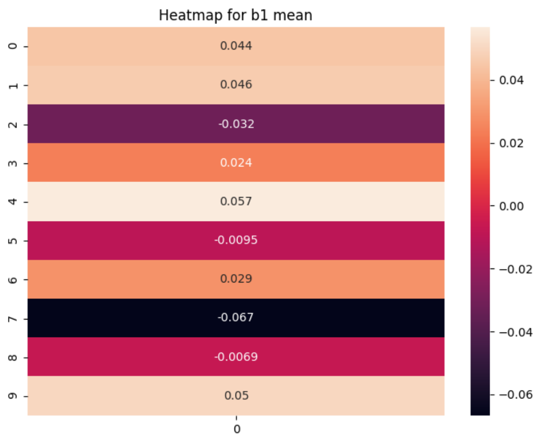


(a) Samples from the posterior for W1

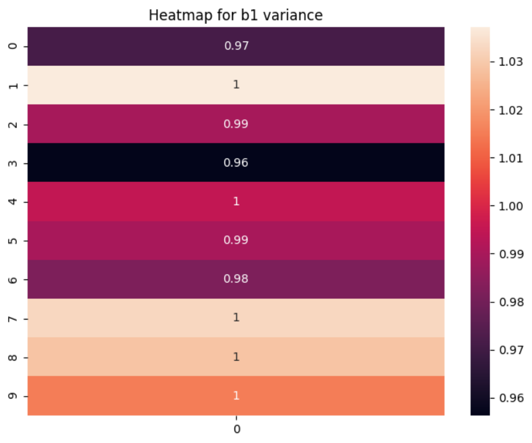


(b) Samples from the posterior for W1

Figure 18: Posterior distribution for slope and bias for 4.daphne using Black Box Variational Inference



(a) Samples from the posterior for b1



(b) Samples from the posterior for b1

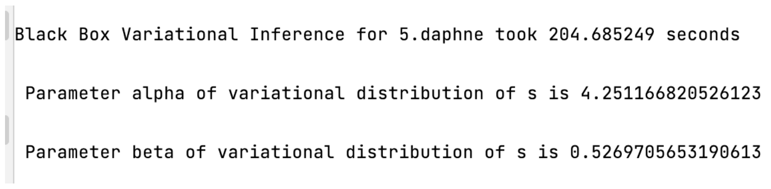
Figure 19: Posterior distribution for slope and bias for 4.daphne using Black Box Variational Inference

Mean-field black-box variational inference can be more widely used since we start from sampling from a prior distribution and keep updating the estimated parameters in posterior distribution by optimizing the evidence lower bound (ELBO), while parameter estimation via gradient descent keeps computing gradient so the function has to be differentiable, and more preferably, gradient is easy to obtain and compute with. Therefore, mean-field black-box variational inference can be applied to more complicated questions and a more generalizable method to use.

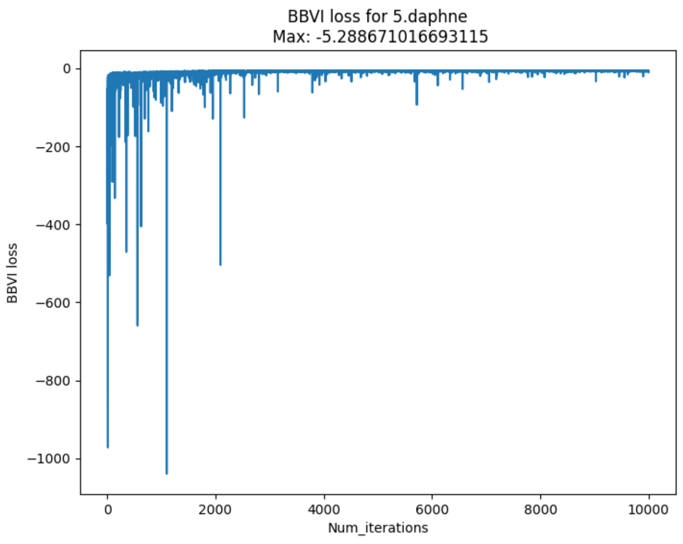
6. Program 5:

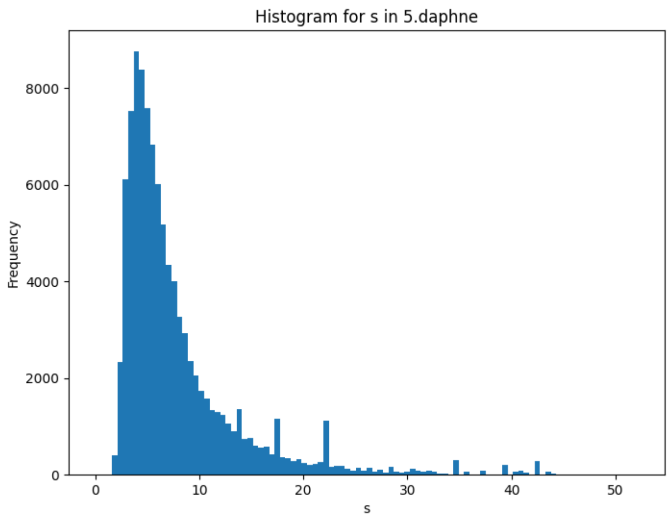
   To make the program run, we propose the distribution for continuous-uniform distribution with Gamma distribution with parameters 2 and m sampled from its distribution as the domain of Gamma distribution is the positive real line, satisfying the variance represented by m in normal distribution is always positive. The parameters of learned variational distribution are fluctuating, and we choose the last updated parameters in the gamma distribution as the final decision.

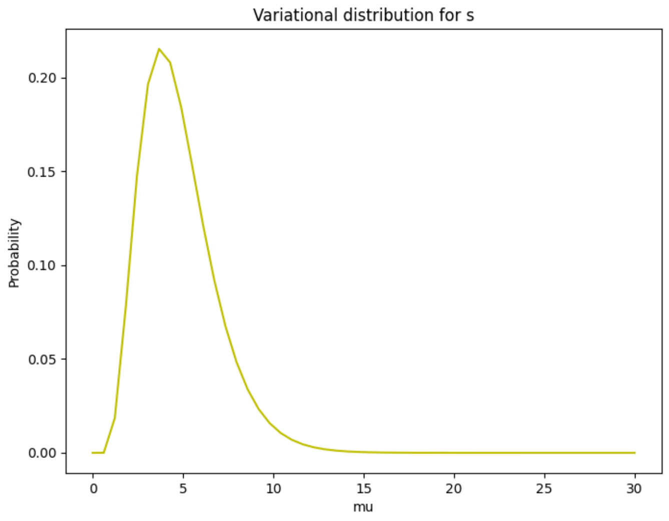
   The learned variational distribution for s is Gamma distribution with parameters 4.2512 and 0.5270.



```
Black Box Variational Inference for 5.daphne took 204.685249 seconds

 Parameter alpha of variational distribution of s is 4.251166820526123

 Parameter beta of variational distribution of s is 0.5269705653190613
```


BBVI loss for 5.daphne
Max: -5.288671016693115
BBVI loss
Num_iterations
0
−200
−400
−600
−800
−1000
0
2000
4000
6000
8000
10000


Histogram for s in 5.daphne
Frequency
s
0
2000
4000
6000
8000
0
10
20
30
40
50


Variational distribution for s
Probability
mu
0.00
0.05
0.10
0.15
0.20
0
5
10
15
20
25
30