hw2-pyro

April 4, 2020

[9]:
```
warning_status = "ignore" #@param ["ignore", "always", "module", "once",␣
↪"default", "error"]
import warnings
warnings.filterwarnings(warning_status)
with warnings.catch_warnings():
    warnings.filterwarnings(warning_status, category=DeprecationWarning)
    warnings.filterwarnings(warning_status, category=UserWarning)

import torch
import torch.nn.functional as F
import pyro
import pyro.distributions as dist
from pyro.infer import EmpiricalMarginal
```

[2]:
```
def model():
    A_probs = torch.tensor([0.5, 0.5])

    B_probs = torch.tensor([
            [0.8, 0.2],
            [0.1, 0.9]
    ])
    C_probs = torch.tensor([
            [[0.90, 0.10], [0.99, 0.01]],
            [[0.10, 0.9], [0.40, 0.60]]])


    A = pyro.sample("A", pyro.distributions.Categorical(probs=A_probs))
    B = pyro.sample("B",  pyro.distributions.Categorical(probs=B_probs[A]))
    C = pyro.sample("C", pyro.distributions.Categorical(probs=C_probs[B][A]))

    return{'A': A,
           'B': B,
           'C': C
          }
```

[3]:
```
intervened_model = pyro.condition( pyro.do( model, data={'B': torch.tensor(1) }␣
↪), data={'C': torch.tensor(1) } )
```

```
conditioned_model = pyro.condition(model,  data={"B": torch.tensor(1), "C":␣
↪torch.tensor(1)})
```

[4]:
```
posterior_intervened = pyro.infer.Importance(intervened_model,␣
↪num_samples=5000).run()
marginal_intervened = EmpiricalMarginal(posterior_intervened, "A")
samples_intervened = [marginal_intervened().item() for _ in range(5000)]
unique_intervened, counts_intervened = np.unique(samples_intervened,␣
↪return_counts=True)
p_a_doB = counts_intervened[1]/counts_intervened.sum()
p_a_doB
```

[4]:
```
0.4054
```

[7]:
```
posterior_conditioned = pyro.infer.Importance(conditioned_model,␣
↪num_samples=5000).run()
marginal_conditioned = EmpiricalMarginal(posterior_conditioned, "A")
samples_conditioned = [marginal_conditioned().item() for _ in range(5000)]
unique_conditioned, counts_conditioned = np.unique(samples_conditioned,␣
↪return_counts=True)
p_a_B = counts_conditioned[1]/counts_conditioned.sum()
p_a_B
```

[7]:
```
0.7564
```

[8]:
```
print(f"P (A = on | B = on, C = on) = {p_a_B} and P (A = on | do(B = on), C =␣
↪on) = {p_a_doB}")
```

```
P (A = on | B = on, C = on) = 0.7564 and P (A = on | do(B = on), C = on) =
0.4054
```

[ ]:
```

```