Name: Darshan D.Prabhu

Roll No.:86

Class: BE/AIML-C3

Experiment No.3

AIM: Design and implement a Bayesian Network for outcome prediction.

THEORY:

Designing and implementing a Bayesian Network for outcome prediction involves several steps. First, you need to define the variables, their dependencies, and the conditional probability distributions. Then, you can implement the Bayesian Network using a programming language like Python and libraries such as pomegranate or pgmpy. Here's a simplified example of how you might design and implement a Bayesian Network for outcome prediction:

Step 1: Define Variables and Dependencies

Let's consider a simple medical example where we want to predict whether a patient has a certain disease based on symptoms and test results. The variables we might consider are:

Disease (D): The presence or absence of the disease.

Symptom (S): The presence or absence of a particular symptom. ● Test Result (T): The result of a diagnostic test for the disease.

Dependencies:

Disease depends on both symptoms and test results.

Test results depend on the disease.

Step 2: Define Conditional Probability Distributions (CPDs)

We need to specify the conditional probability distributions for each variable given its parents in the network.

For example:

P(D): Prior probability of disease.

P(S | D): Probability of symptoms given disease.

P(T | D): Probability of test results given disease.

Step 3: Implement Bayesian Network in Python

You can implement the Bayesian Network using libraries such as pomegranate or pgmpy. pgmpy is used to define the structure of the Bayesian Network, specify the conditional probability distributions, and perform inference to predict outcomes.

You would need to adapt this example to your specific problem domain and incorporate more variables and complex dependencies as needed. Additionally, you may need to train your Bayesian Network on data to estimate the parameters of the conditional probability distributions.

PROGRAM:

```python
#Import the necessary libraries
import torch
import pyro
import pyro.distributions as dist
from pyro.infer import SVI, Trace_ELBO, Predictive
from pyro.optim import Adam
import matplotlib.pyplot as plt
import seaborn as sns


# Generate some sample data
torch.manual_seed(0)
X = torch.linspace(0, 10, 100)
true_slope = 2
true_intercept = 1
Y = true_intercept + true_slope * X + torch.randn(100)

# Define the Bayesian regression model
def model(X, Y):
  # Priors for the parameters
  slope = pyro.sample("slope", dist.Normal(0, 10))
  intercept = pyro.sample("intercept", dist.Normal(0, 10))
  sigma = pyro.sample("sigma", dist.HalfNormal(1))

  # Expected value of the outcome
  mu = intercept + slope * X

  # Likelihood (sampling distribution) of the observations
  with pyro.plate("data", len(X)):
    pyro.sample("obs", dist.Normal(mu, sigma), obs=Y)
```

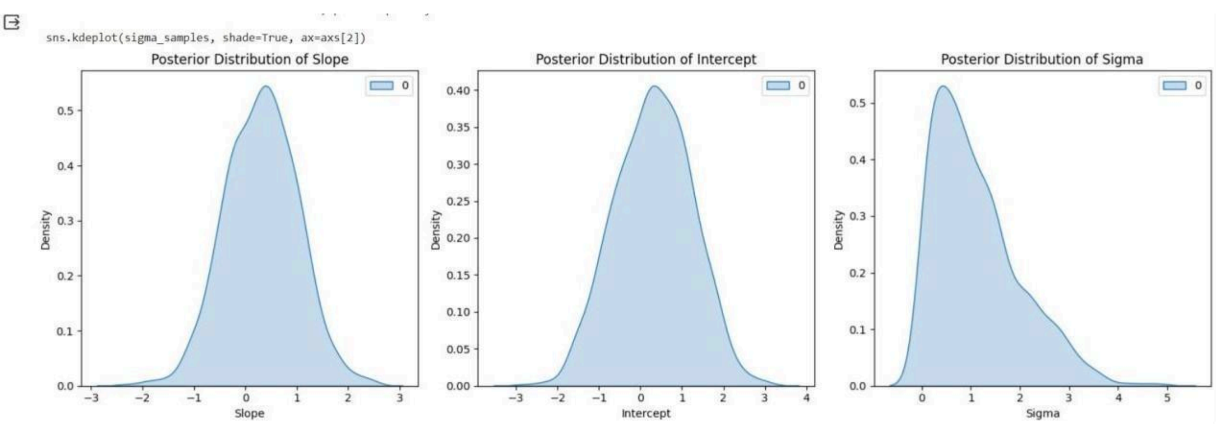
```python
# Run Bayesian inference using SVI (Stochastic Variational Inference)
def guide(X, Y):
  # Approximate posterior distributions for the parameters
  slope_loc = pyro.param("slope_loc", torch.tensor(0.0))
  slope_scale = pyro.param("slope_scale", torch.tensor(1.0),
              constraint=dist.constraints.positive)
  intercept_loc = pyro.param("intercept_loc", torch.tensor(0.0))
  intercept_scale = pyro.param("intercept_scale", torch.tensor(1.0),
              constraint=dist.constraints.positive)
  sigma_loc = pyro.param("sigma_loc", torch.tensor(1.0),
              constraint=dist.constraints.positive)

  # Sample from the approximate posterior distributions
  slope = pyro.sample("slope", dist.Normal(slope_loc, slope_scale))
  intercept = pyro.sample("intercept", dist.Normal(intercept_loc,
                          intercept_scale))
  sigma = pyro.sample("sigma", dist.HalfNormal(sigma_loc))

# Initialize the SVI and optimizer
optim = Adam({"lr": 0.01})
svi = SVI(model, guide, optim, loss=Trace_ELBO())

# Run the inference loop
num_iterations = 1000
for i in range(num_iterations):
  loss = svi.step(X, Y)
  if (i + 1) % 100 == 0:
    print(f"Iteration {i + 1}/{num_iterations} - Loss: {loss}")

# Obtain posterior samples using Predictive
predictive = Predictive(model, guide=guide, num_samples=1000)
posterior = predictive(X, Y)
```

```python
# Extract the parameter samples
slope_samples = posterior["slope"]
intercept_samples = posterior["intercept"]
sigma_samples = posterior["sigma"]

Tensor: sigma_mean
Tensor with shape torch.Size([]) ean()
sigma_mean = sigma_samples.mean()

# Print the estimated parameters
print("Estimated Slope:", slope_mean.item())
print("Estimated Intercept:", intercept_mean.item())
print("Estimated Sigma:", sigma_mean.item())


# Create subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Plot the posterior distribution of the slope
sns.kdeplot(slope_samples, shade=True, ax=axs[0])
axs[0].set_title("Posterior Distribution of Slope")
axs[0].set_xlabel("Slope")
axs[0].set_ylabel("Density")

# Plot the posterior distribution of the intercept
sns.kdeplot(intercept_samples, shade=True, ax=axs[1])
axs[1].set_title("Posterior Distribution of Intercept")
axs[1].set_xlabel("Intercept")
axs[1].set_ylabel("Density")
```

```python
# Plot the posterior distribution of sigma
sns.kdeplot(sigma_samples, shade=True, ax=axs[2])
axs[2].set_title("Posterior Distribution of Sigma")
axs[2].set_xlabel("Sigma")
axs[2].set_ylabel("Density")

# Adjust the layout
plt.tight_layout()

# Show the plot
plt.show()
```

OUTPUT:-

```
Iteration 100/1000 - Loss: 68686.4717028141
Iteration 200/1000 - Loss: 1957.55493080616
Iteration 300/1000 - Loss: 647.4665781259537
Iteration 400/1000 - Loss: 788.4604915380478
Iteration 500/1000 - Loss: 3308.1984667778015
Iteration 600/1000 - Loss: 20155.736622929573
Iteration 700/1000 - Loss: 2545.918571829796
Iteration 800/1000 - Loss: 515579.78982794285
Iteration 900/1000 - Loss: 1881.5490272045135
Iteration 1000/1000 - Loss: 50892.460729599
Estimated Slope: 0.30964675545692444
Estimated Intercept: 0.31471437215805054
Estimated Sigma: 1.1101857423782349
<ipython-input-3-475f8b8850cf>:85: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(slope_samples, shade=True, ax=axs[0])
<ipython-input-3-475f8b8850cf>:91: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(intercept_samples, shade=True, ax=axs[1])
<ipython-input-3-475f8b8850cf>:97: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(sigma_samples, shade=True, ax=axs[2])
```



CONCLUSION:

Hence, we Successfully implemented Design and implement a Bayesian Network for outcome prediction.