ATE computations from Baysian Networks in RCTs

This notebook aims to study the capabilities of Bayesian Networks for computing Average Treatment Effects (ATE) in Randomized Control Trials (RCT) under the Neyman-Rubin potential outcome framework.

Consider a set of n independent and identically distributed subjects. an observation on the i-th subject is given by the tuple (T_i, X_i, Y_i) where:

- T_i taking values in $\{0, 1\}$ is a binary random variable representing the treatment.
- X_i is the covariate vector.
- $Y_i = T_i Y_i(1) + (1 - T_i) Y_i(0)$ is the outcome of the treatment on the i-th subject, with $Y_i(1)$ and $Y_i(0)$ representing the treated and untreated outcomes, respectively.

We are interested in quantifying the effect of a given treatment on the population, namely the quantity $\Delta_i = Y_i(1) - Y_i(0)$. Althought this number cannot be directed calculated due to the presence of counterfactuals, there exists methods for approximating its expected value, the Avereage Treatment Effect:

$$\tau = \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\Delta_i\right] = \mathbb{E}[Y_(1)] - \mathbb{E}[Y_(0)]$$

To achive this, we suppose the Stable-Unit-Treatment-Value Assumption (SUTVA) is verified and further assume ignorability between the observations:

- $Y_i = Y_i(T_i)$ (SUTVA)
- $T_i \perp\!\!\!\perp \{Y_i(0), Y_i(1)\}$ (Ignorability)

We will proceed to present estimators of τ using Baysian Networks through three different methods:

- "Exact" Computation
- Parametric Learning
- Structural Learning

In [ ]:
```
import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
from scipy.stats import norm
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Consider a linear generative model described by the equation:

$$Y = 3X_1 + 2X_2 - 2X_3 - 0.8X_4 + T(2X_1 + 5X_3 + 3X_4)$$

Where $(X_1, X_2, X_3, X_4) \sim \mathcal{N}((1,1,1,1), I_4), T \sim \mathcal{Ber}(1/2)$ and

(X_1, X_2, X_3, X_4, T) are jointly independent.

Data from this model can be generated by the function given below.

In [ ]:
```
def linear_simulation(n,sigma,p):
  X1 = np.random.normal(1,1, n)
  X2 = np.random.normal(1,1, n)
  X3 = np.random.normal(1,1, n)
  X4 = np.random.normal(1,1, n)
  epsilon = np.random.normal(0,sigma, n)
  T=np.random.binomial(1, p, n)
  Y= 3*X1+ 2*X2-2*X3-0.8*X4+T*(2*X1+ 5*X3+ 3*X4) +epsilon
  d=np.array([T,X1,X2,X3,X4,Y])
  df_data = pd.DataFrame(data=d.T,columns=['T','X1','X2','X3','X4','Y'])
  df_data["T"] = df_data["T"].astype(int)
  return df_data
```

The expected values of $Y(0)$ and $Y(1)$ can be explicitly calculated, providing us the theoretical ATE which enables performance evaluations of the estimators.

$$\mathbb{E}[Y(0)] = \mathbb{E}[3X_1 + 2X_2 - 2X_3 - 0.8X_4] = 2.2$$

$$\mathbb{E}[Y(1)] = \mathbb{E}[5X_1 + 2X_2 + 3X_3 + 2.2X_4] = 12.2$$

$$ \tau = \mathbb{E}[Y(1)] - \mathbb{E}[Y(0)] = 10

1 - "Exact" Computation

Exact theoretical expected values can be calculated using Bayesian Networks by inputting the data-generating distribution directly into the network. However, since pyAgrum does not support continuous variables as of July 2024, a discretization of continuous distributions is necessary. Consequently, the calculated value will not be exact in a strict sense, but with a sufficient number of discrete states, a close approximation can be achieved.

In [ ]:
```
def getIntervals(start : float, end : float, num : int) -> list[list[floa
    """
    Returns list containing num intervals that paritions the interval [st
    """
    arr = (end-start)*np.arange(num+1)/(num)+start
    res = [[arr[i], arr[i+1]] for i in range(len(arr)-1)]
    return res

def getMeans(start : float, end : float, num : int) -> str:
    """
    Returns means of intervals of getIntervals under string format
    """
    arr = (end-start)*np.arange(num+1)/(num)+start
    means = [(arr[i]+arr[i+1])/2 for i in range(len(arr)-1)]
    res = "{"
    for n in means:
        res += str(n)+"|"
    return res[:len(res)-1] + "}"
```

In [ ]:
```
# Covariate parameters
```

```
covariate_start = -4.0
covariate_end = 6.0
covariate_num_split = 20
covariate_intervals = getIntervals(covariate_start, covariate_end, covari
covariate_domain = getMeans(covariate_start, covariate_end, covariate_num
covariate_distribution = norm(loc=1, scale=1)

# Outcome parameters

outcome_start = -20.0
outcome_end = 40.0
outcome_num_split = 60
outcome_intervals = getIntervals(outcome_start, outcome_end, outcome_num_
outcome_domain = getMeans(outcome_start, outcome_end, outcome_num_split)
```

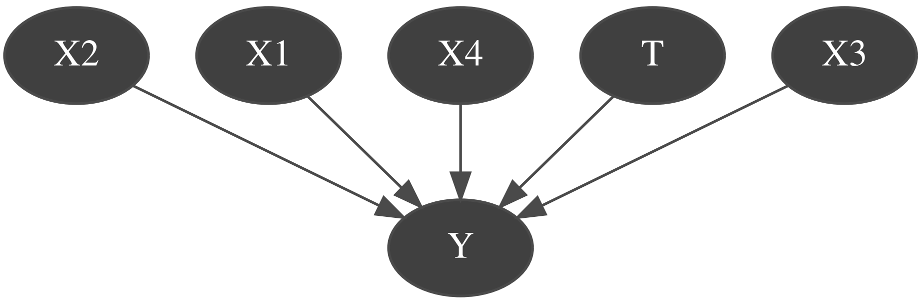
In [ ]:

```
# Theoretical distributions of Y0 and Y1 calculated from the equation of

y0_mean = 2.2
y0_var = 17.64
y1_mean = 12.2
y1_var = 42.84

x = np.linspace(-20, 40, 600)
y0 = norm(loc=y0_mean, scale=np.sqrt(y0_var)).pdf(x)
y1 = norm(loc=y1_mean, scale=np.sqrt(y1_var)).pdf(x)

pdf_df = pd.DataFrame(data={"y0": y0, "y1": y1}, index=x)
```

In [ ]:

```
exbn = gum.fastBN(f"Y{outcome_domain}; \
                  X1{covariate_domain}->Y<-X2{covariate_domain}; \
                  X3{covariate_domain}->Y<-X4{covariate_domain}; \
                  T[0,1]->Y")

exbn.cpt("X1").fillFromDistribution(covariate_distribution)
exbn.cpt("X2").fillFromDistribution(covariate_distribution)
exbn.cpt("X3").fillFromDistribution(covariate_distribution)
exbn.cpt("X4").fillFromDistribution(covariate_distribution)
exbn.cpt("T").fillWith([0.5, 0.5])
exbn.cpt("Y").fillFromFunction("3*X1 + 2*X2 - 2*X3 - 0.8*X4 + T*(2*X1 + 5

exbn
```

Out[ ]:



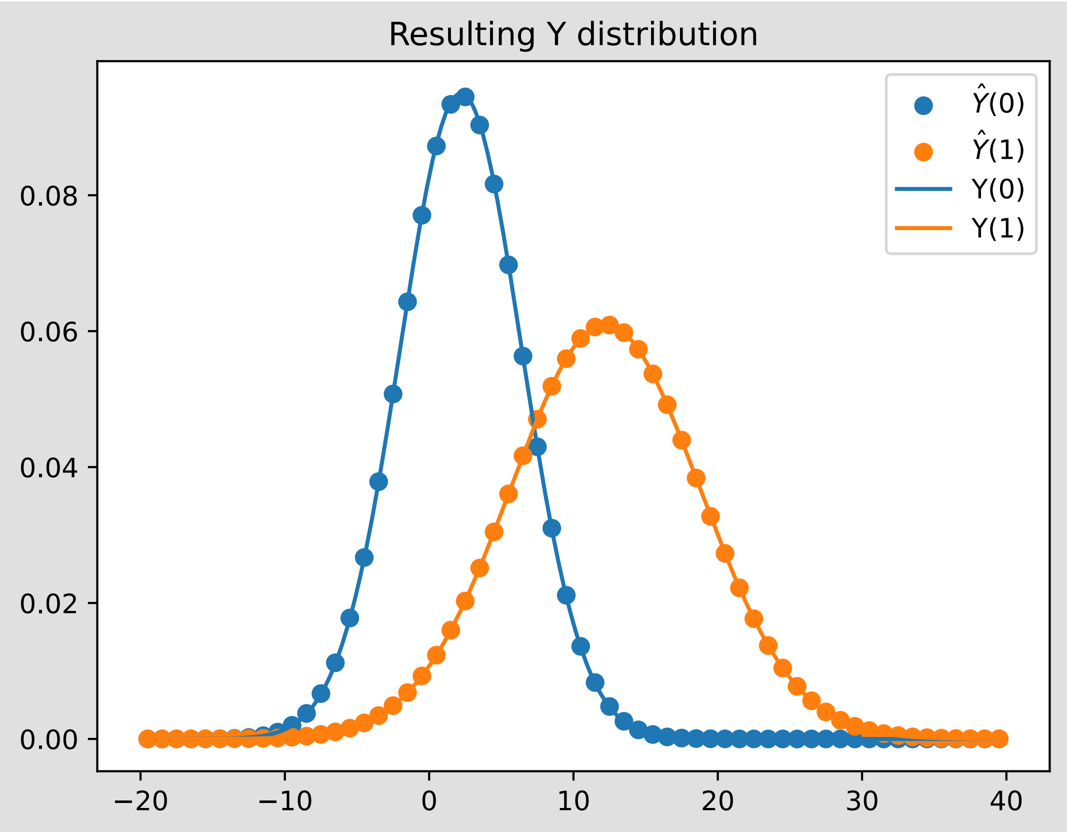
In [ ]:

```
def getY(bn : gum.BayesNet) -> pd.DataFrame:
    """
    Returns the estimation of outcome Y from Lazy Propagation
    """
    ie = gum.LazyPropagation(bn)
```

```
    ie.setEvidence({"T":0})
    ie.makeInference()
    Y0 = ie.posterior("Y").topandas()
    Y0 = Y0.reset_index(level=[None,''])
    Y0["T"] = 0
    Y0["interval_mean"] = Y0[''].astype(float)
    Y0["probability"] = Y0[0]
    Y0 = Y0.drop(columns=["", 0, "level_0"])

    ie.setEvidence({"T":1})
    ie.makeInference()
    Y1 = ie.posterior("Y").topandas()
    Y1 = Y1.reset_index(level=[None,''])
    Y1["T"] = 1
    Y1["interval_mean"] = Y1[''].astype(float)
    Y1["probability"] = Y1[0]
    Y1 = Y1.drop(columns=["", 0, "level_0"])

    return [Y0, Y1]
```

In [ ]:

```
Y = getY(exbn)
plt.scatter(x=Y[0]["interval_mean"] ,y=Y[0]["probability"], color="tab:bl
plt.scatter(x=Y[1]["interval_mean"] ,y=Y[1]["probability"], color="tab:or
plt.plot(pdf_df["y0"], color="tab:blue", label="Y(0)")
plt.plot(pdf_df["y1"], color="tab:orange", label="Y(1)")
plt.title("Resulting Y distribution")
plt.legend()

plt.show()
```



In [ ]:

```
def getTau(Y : list[pd.DataFrame]) -> float:
    """
    Returns estimation of the ATE tau
    """
```

```
    E0 = (Y[0]["interval_mean"] * Y[0]["probability"]).sum()
    E1 = (Y[1]["interval_mean"] * Y[1]["probability"]).sum()
    tau = E1 - E0
    return tau
```

In [ ]:
```
print(f"Estination : {getTau(Y)} \nExpected Value : 10.0 \nBias : {getTau
```

```
Estination : 9.999979754001421
Expected Value : 10.0
Bias : -2.024599857897158e-05
```

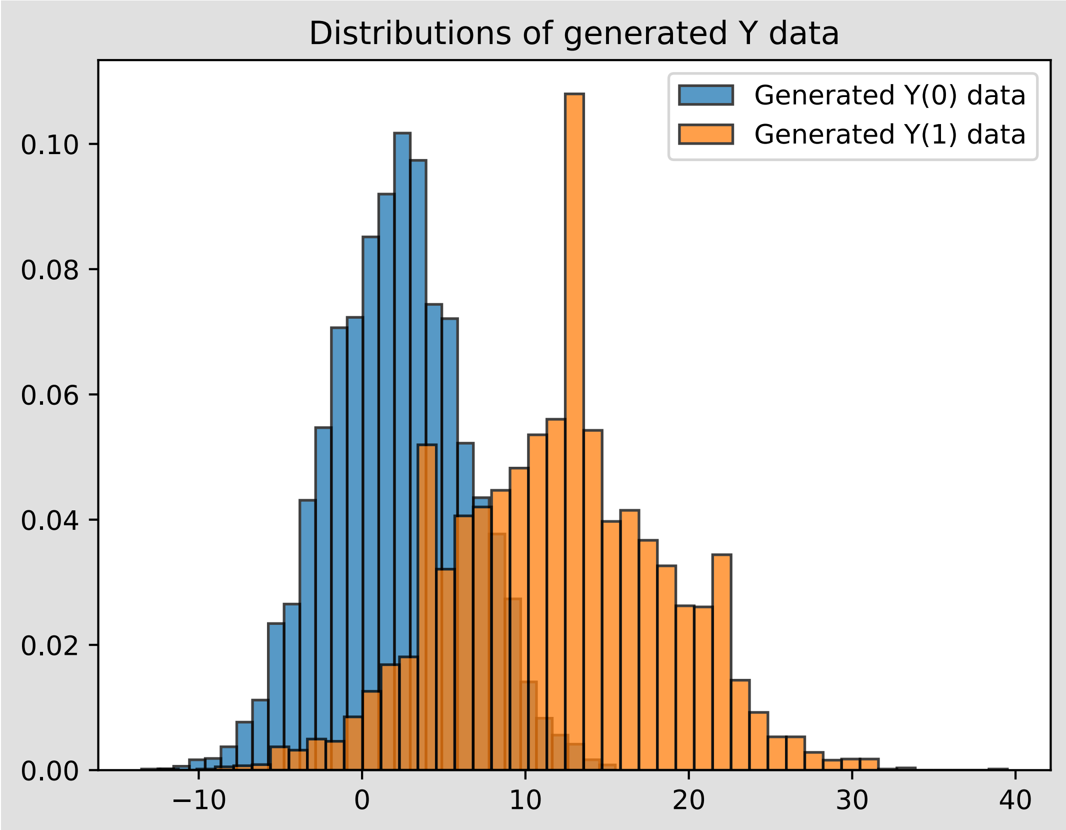
2 - Parameter Learning

Given the data generating function defined above, parameter learning methods can be utilized to infer the underlying distribution based on the structure of the Bayesian network. However, since the generated data is continuous, categorization will be necessary to reuse the previous network structure.

In [ ]:
```
def categorise(x : float, intervals_list : list[list[float]]) -> float:
    """
    Returns the mean of the interveral which contains x.
    If x isn't contained in any of the intervals,
    the mean of the smallest or largest interval will be returned.
    """
    if x < intervals_list[0][0]:
        return np.mean(intervals_list[0])
    for interval in intervals_list:
        if interval[0] <= x and x < interval[1]:
            return np.mean(interval)
    return np.mean(intervals_list[-1])
```

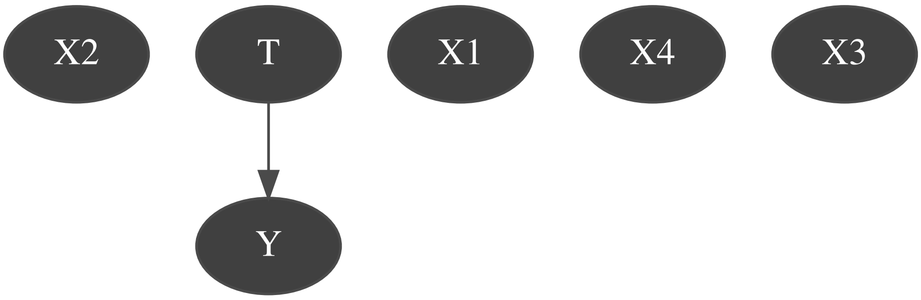
In [ ]:
```
df = linear_simulation(10000, 1.0, 0.5)
df["X1"] = df["X1"].apply(lambda x: categorise(x, covariate_intervals))
df["X2"] = df["X2"].apply(lambda x: categorise(x, covariate_intervals))
df["X3"] = df["X3"].apply(lambda x: categorise(x, covariate_intervals))
df["X4"] = df["X4"].apply(lambda x: categorise(x, covariate_intervals))
df["Y"] = df["Y"].apply(lambda x: categorise(x, outcome_intervals))
```

In [ ]:
```
Y0 = df[df["T"] == 0]["Y"]
Y1 = df[df["T"] == 1]["Y"]

plt.hist(Y0, bins=Y0.nunique(), density=True, alpha=0.75, edgecolor='blac
plt.hist(Y1, bins=Y1.nunique(), density=True, alpha=0.75, edgecolor='blac
plt.title("Distributions of generated Y data")
plt.legend()

plt.show()
```

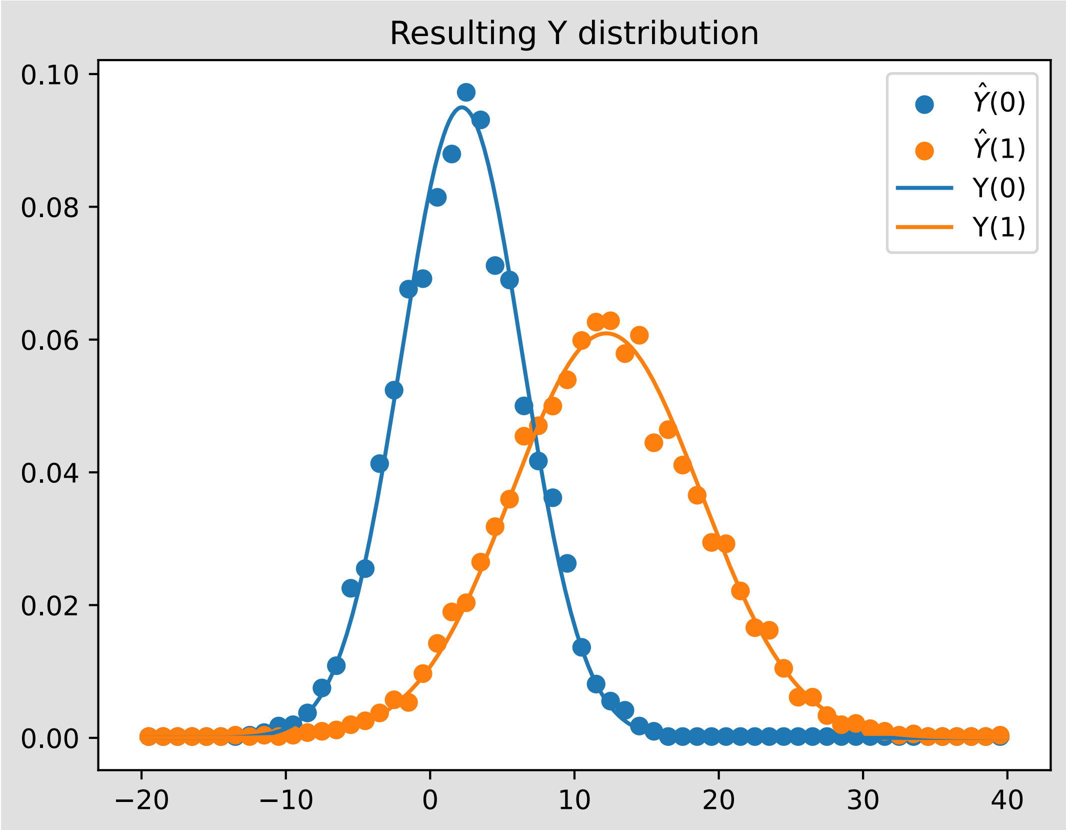


In [ ]:
```
param_learner = gum.BNLearner(df, exbn)
param_learner.useSmoothingPrior(1)
param_learner.learnParameters(exbn.dag())
plbn = param_learner.learnBN()
plbn
```

Out[ ]:



In [ ]:
```
Y = getY(plbn)
plt.scatter(x=Y[0]["interval_mean"] ,y=Y[0]["probability"], color="tab:bl
plt.scatter(x=Y[1]["interval_mean"] ,y=Y[1]["probability"], color="tab:or
plt.plot(pdf_df["y0"], color="tab:blue", label="Y(0)")
plt.plot(pdf_df["y1"], color="tab:orange", label="Y(1)")
plt.title("Resulting Y distribution")
plt.legend()

plt.show()
```



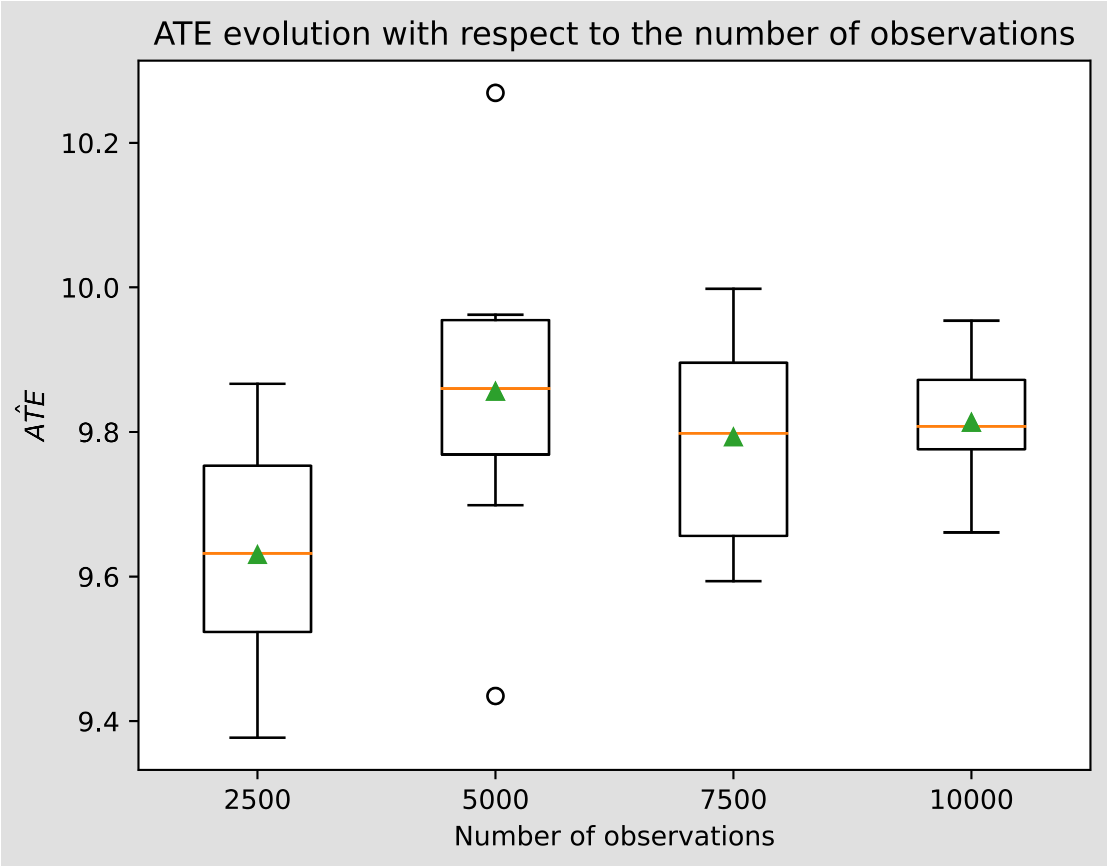
In [ ]:
```
print(f"Estination : {getTau(Y)} \nExpected Value : 10.0 \nBias : {getTau
```

```
Estination : 9.68851555553421
Expected Value : 10.0
Bias : -0.3114844444657905
```

In [ ]:
```
tau_arr = []
num_obs_list = range(2500, 10001, 2500)
num_shots = 10

for i in num_obs_list:
    tau_arr.append(list())
    for j in range(num_shots):
        df = linear_simulation(i, 1.0, 0.5)
        df["X1"] = df["X1"].apply(lambda x: categorise(x, covariate_inter
        df["X2"] = df["X2"].apply(lambda x: categorise(x, covariate_inter
        df["X3"] = df["X3"].apply(lambda x: categorise(x, covariate_inter
        df["X4"] = df["X4"].apply(lambda x: categorise(x, covariate_inter
        df["Y"] = df["Y"].apply(lambda x: categorise(x, outcome_intervals

        param_learner = gum.BNLearner(df, exbn)
        param_learner.useSmoothingPrior(1)
        param_learner.learnParameters(exbn.dag())
        plbn = param_learner.learnBN()

        Y = getY(plbn)
        tau = getTau(Y)
        tau_arr[-1].append(tau)
        print(f"{tau:.2f} ", end="")
    print("")
```

```
9.40 9.79 9.85 9.49 9.63 9.87 9.38 9.63 9.62 9.64
9.82 9.84 10.27 9.70 9.75 9.95 9.96 9.96 9.88 9.43
9.96 9.87 9.81 9.78 9.59 9.90 10.00 9.78 9.60 9.61
9.71 9.82 9.95 9.89 9.77 9.80 9.86 9.79 9.88 9.66
```

In [ ]:
```
plt.boxplot(tau_arr, labels=num_obs_list, meanline=False, showmeans=True,
plt.title(f"ATE evolution with respect to the number of observations")
plt.xlabel("Number of observations")
plt.ylabel("$\hat{ATE}$")
plt.show()
```



2 - Structure Learning

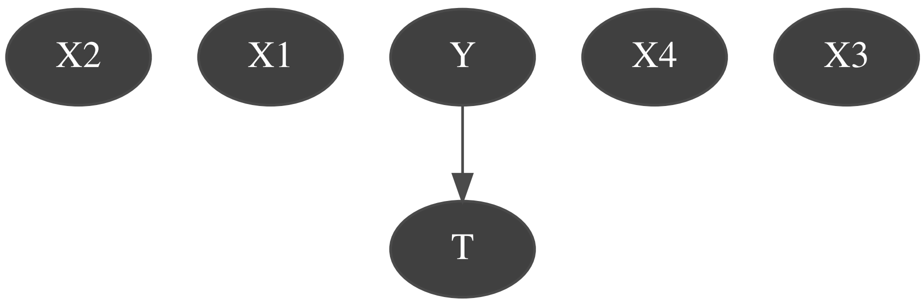
In certain cases, even without a given DAG, it is possible to derive a structure and distributions from a sufficiently large dataset.

In [ ]:
```
def columnsTypeToString(df : pd.DataFrame, columns : list[str] = None) ->
    """
    Returns dataframe with column value type converted to string
    """
    if columns == None:
        columns = df.columns
    for col in columns:
        df[col] = df[col].astype(str)
    return df
```

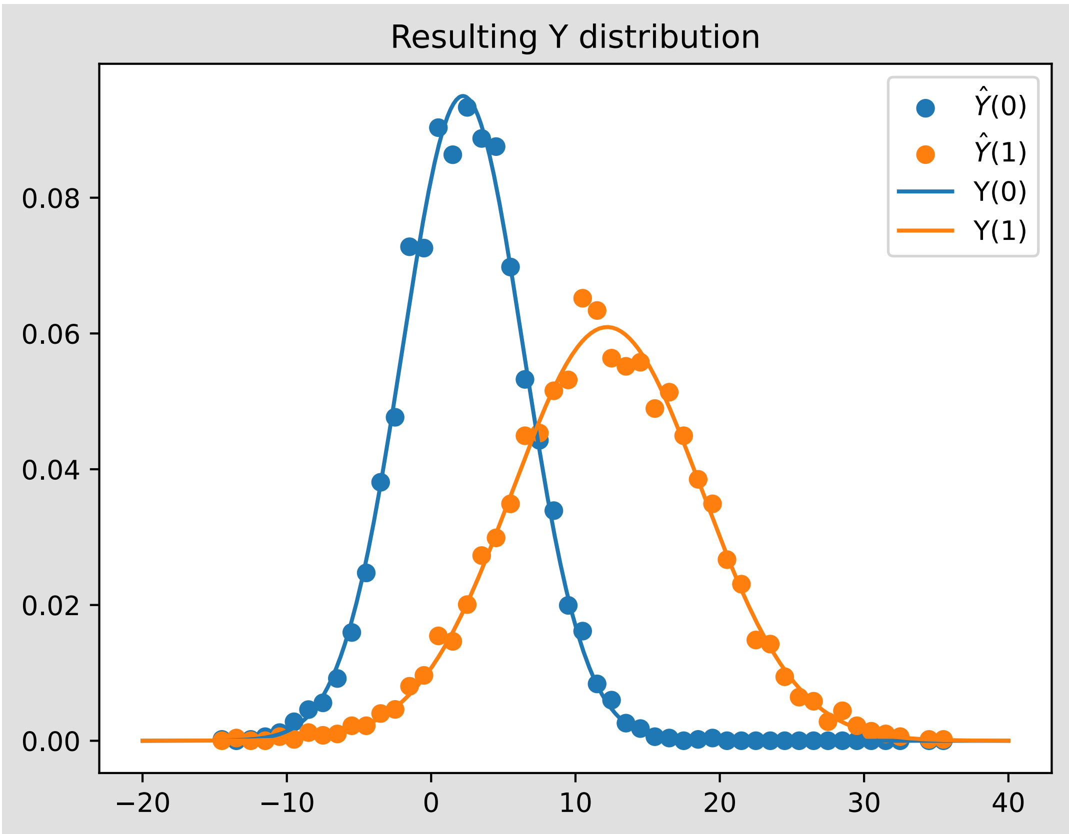
In [ ]:
```
sdf = columnsTypeToString(df)
```

In [ ]:
```
struct_learner = gum.BNLearner(columnsTypeToString(df), ["?"], False)
slbn = struct_learner.learnBN()
slbn
```

Out[ ]:



In [ ]:
```
Y = getY(slbn)
plt.scatter(x=Y[0]["interval_mean"] ,y=Y[0]["probability"], color="tab:bl
plt.scatter(x=Y[1]["interval_mean"] ,y=Y[1]["probability"], color="tab:or
plt.plot(pdf_df["y0"], color="tab:blue", label="Y(0)")
plt.plot(pdf_df["y1"], color="tab:orange", label="Y(1)")
plt.title("Resulting Y distribution")
plt.legend()

plt.show()
```



In [ ]:
```
print(f"Estination : {getTau(Y)} \nExpected Value : 10.0 \nBias : {getTau
```

```
Estination : 9.7767104345399
Expected Value : 10.0
Bias : -0.22328956546009948
```

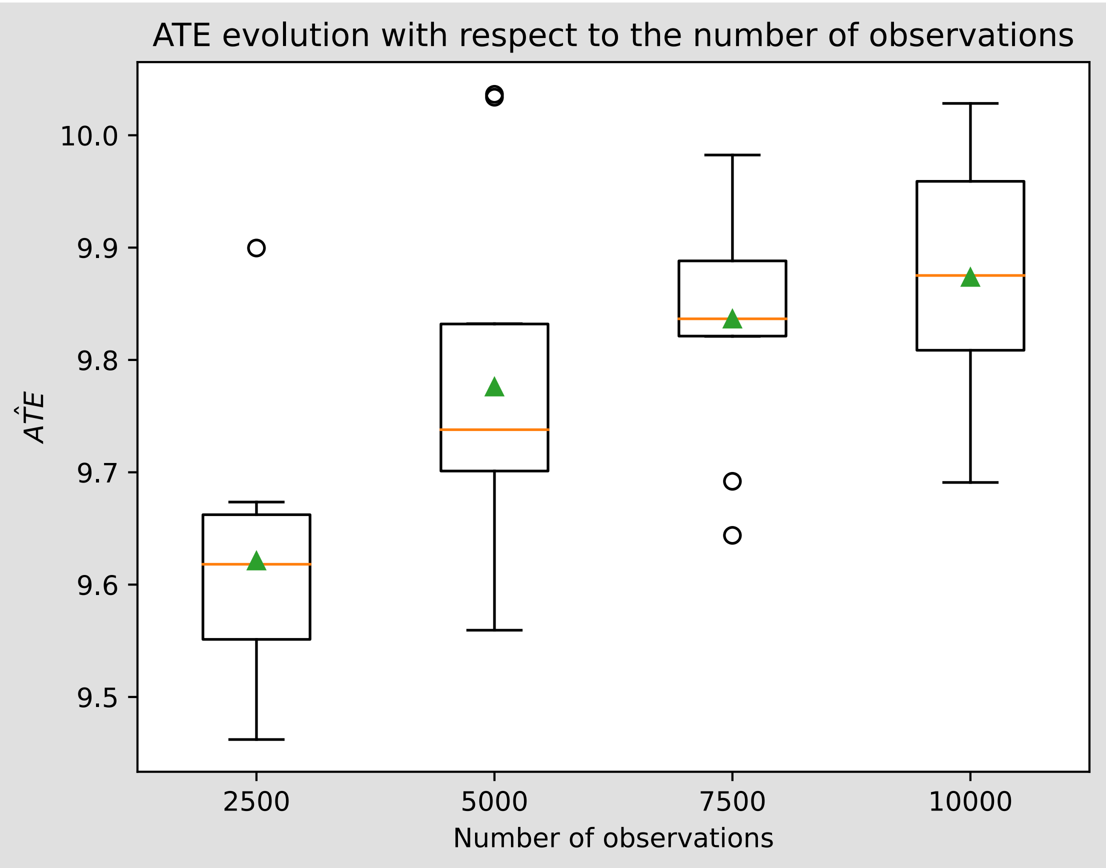
In [ ]:
```
tau_arr = []
num_obs_list = range(2500, 10001, 2500)
num_shots = 10

for i in num_obs_list:
    tau_arr.append(list())
    for j in range(num_shots):
        df = linear_simulation(i, 1.0, 0.5)
```

```
        df["X1"] = df["X1"].apply(lambda x: categorise(x, covariate_inter
        df["X2"] = df["X2"].apply(lambda x: categorise(x, covariate_inter
        df["X3"] = df["X3"].apply(lambda x: categorise(x, covariate_inter
        df["X4"] = df["X4"].apply(lambda x: categorise(x, covariate_inter
        df["Y"] = df["Y"].apply(lambda x: categorise(x, outcome_intervals

        param_learner = gum.BNLearner(df, exbn)
        param_learner.useSmoothingPrior(1)
        param_learner.learnParameters(exbn.dag())
        plbn = param_learner.learnBN()

        Y = getY(plbn)
        tau = getTau(Y)
        tau_arr[-1].append(tau)
        print(f"{tau:.2f} ", end="")
    print("")
```

```
9.46 9.50 9.67 9.63 9.58 9.67 9.54 9.65 9.90 9.60
10.03 9.58 9.83 9.83 9.71 9.71 9.76 10.04 9.56 9.70
9.87 9.82 9.97 9.85 9.90 9.98 9.82 9.82 9.69 9.64
9.99 9.73 9.69 9.91 10.03 9.87 9.79 9.97 9.88 9.87
```

In [ ]:

```
plt.boxplot(tau_arr, labels=num_obs_list, meanline=False, showmeans=True,
plt.title(f"ATE evolution with respect to the number of observations")
plt.xlabel("Number of observations")
plt.ylabel("$\hat{ATE}$")
plt.show()
```



In [ ]: