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
import pyAgrum.skbn as skbn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from scipy.stats import norm
from scipy.integrate import quad
```

We will consider two generative models in this notebook:

- A linear generative model described by the equation:

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

- And a non-linear generative model described by the equation:

$$Y = 3X_1 + 2X_2^2 - 2X_3 - 0.8X_4 + 10T$$

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 in both of the models.

Data from the models can be generated by the functions 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

def non_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**2-2*X3-0.8*X4+10*T +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.

Both models have an ATE of $\tau = \mathbb{E}[Y(1)] - \mathbb{E}[Y(0)] = 10$

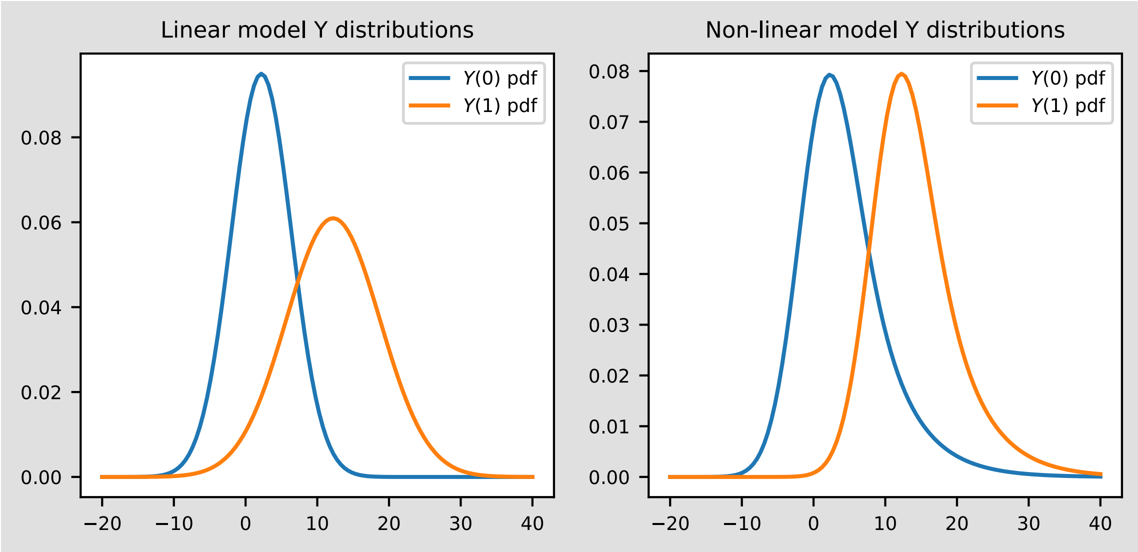
In [ ]:
```
# Computations of the theoretical distributions of Y0 and Y1 given by
# the equations of Y

X = np.linspace(-20, 40, 120)
dx = X[1] - X[0]

# Linear model

lin_y0_mean = 2.2
lin_y0_var = 17.64
lin_y1_mean = 12.2
lin_y1_var = 42.84
```

```
lin_y0 = norm(loc=lin_y0_mean, scale=np.sqrt(lin_y0_var)).pdf(X)
lin_y1 = norm(loc=lin_y1_mean, scale=np.sqrt(lin_y1_var)).pdf(X)

lin_pdf_df = pd.DataFrame(data={"y0": lin_y0, "y1": lin_y1}, index=X)

# Non Linear model

def twoX2squared_func(x):
    return 0 if x <= 0 else \
    (norm(1, 1).pdf(np.sqrt(x/2.0)) + norm(1, 1).pdf(-np.sqrt(x/2.0))) \
    / (4.0*np.sqrt(x))

def convolve(f, g):
    return (lambda t: quad((lambda x: f(t-x)*g(x)), -np.inf, np.inf))

nl_y0_norm_mean = 0.2
nl_y0_norm_var = 13.64
nl_y1_norm_mean = 10.2
nl_y1_norm_var = 13.64

nl_y0_norm = norm(loc=nl_y0_norm_mean, scale=np.sqrt(nl_y0_norm_var)).pdf
nl_y1_norm = norm(loc=nl_y1_norm_mean, scale=np.sqrt(nl_y1_norm_var)).pdf

nl_y0_func = convolve(nl_y0_norm, twoX2squared_func)
nl_y1_func = convolve(nl_y1_norm, twoX2squared_func)

nl_y0 = list()
nl_y1 = list()
for x in X:
    nl_y0.append(nl_y0_func(x)[0])
    nl_y1.append(nl_y1_func(x)[0])

nl_y0 = np.array(nl_y0)
nl_y0 = nl_y0/(nl_y0.sum()*dx)

nl_y1 = np.array(nl_y1)
nl_y1 = nl_y1/(nl_y1.sum()*dx)

nl_pdf_df = pd.DataFrame(data={"y0": nl_y0, "y1": nl_y1}, index=X)

# Runtime ~ 1 min
```

In [ ]:

```
# Plotting the distributions

plt.rcParams.update({'font.size': 7})
plt.subplots(figsize=(7, 3))

plt.subplot(1, 2, 1)

plt.plot(X, lin_y0, color="tab:blue", label="$Y(0)$ pdf")
plt.plot(X, lin_y1, color="tab:orange", label="$Y(1)$ pdf")
plt.legend()
plt.title("Linear model Y distributions")

plt.subplot(1, 2, 2)

plt.plot(X, nl_y0, color="tab:blue", label="$Y(0)$ pdf")
plt.plot(X, nl_y1, color="tab:orange", label="$Y(1)$ pdf")
plt.legend()
plt.title("Non-linear model Y distributions")
```

```
plt.show()
```



In [ ]:
```
# Y Expressions

lin_expr = "3*X1 + 2*X2 - 2*X3 - 0.8*X4 + T*(2*X1 + 5*X3 + 3*X4)"
nl_expr = "3*X1 + 2*(X2*X2) - 2*X3 - 0.8*X4 + 10*T"
```

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 getSplits(start : float, end : float, num : int) -> list[float]:
    """
    Returns list containing num intervals that paritions the interval [st
    """
    arr = (end-start)*np.arange(num+1)/(num)+start
    return arr.tolist()

def getStringIntervalMean(interval_string : str) -> float:
    """
    """
    separator = 0
    start = ""
    end = ""
    for c in interval_string:
        if str.isdecimal(c) or c in {"-", "."}:
            if separator == 1:
                start += c
            else:
                end += c
        else:
            separator += 1
    start = float(start)
    end = float(end)
```

```
    return (start + end)/2.0

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()

    var_labels = list()
    var = ie.posterior("Y").variable(0)
    for i in range(var.domainSize()):
        var_labels.append(var.label(i))

    Y0 = pd.DataFrame({"T": 0, "interval": var_labels, \
                       "probability": ie.posterior("Y").tolist()})

    Y0["interval_mean"] = Y0["interval"].apply(getStringIntervalMean)

    ie.setEvidence({"T": 1})
    ie.makeInference()

    var_labels = list()
    var = ie.posterior("Y").variable(0)
    for i in range(var.domainSize()):
        var_labels.append(var.label(i))

    Y1 = pd.DataFrame({"T": 1, "interval": var_labels, \
                       "probability": ie.posterior("Y").tolist()})

    Y1["interval_mean"] = Y1["interval"].apply(getStringIntervalMean)

    return [Y0, Y1]

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 [ ]:

```
# Covariate parameters

# WARNING : With bigger intervals, the pdf approaches 0 and may cause
# computational errors when filling the Y distribution

covariate_start = -3.0
covariate_end = 5.0
covariate_num_split = 10
covariate_domain = getSplits(covariate_start, covariate_end, \
                             covariate_num_split)
covariate_distribution = norm(loc=1, scale=1)

# Outcome parameters

outcome_start = -20.0
```

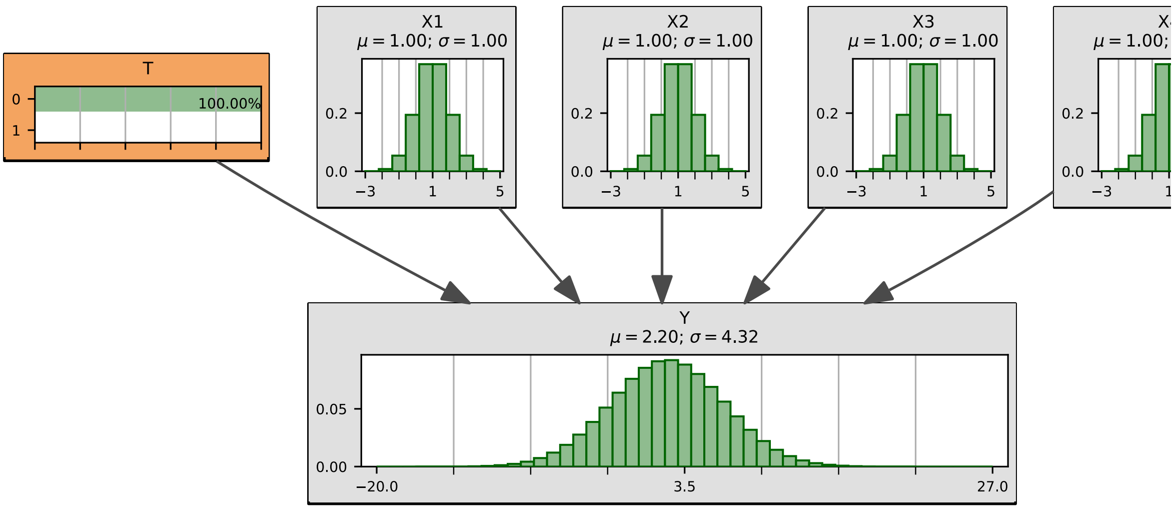
```
outcome_end = 40.0
outcome_num_split = 60
outcome_domain = getSplits(outcome_start, outcome_end, \
                           outcome_num_split)
```

In [ ]:
```
def getBN(outcome_domain, covariate_domain, expr):
    """
    """

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

    bn.cpt("X1").fillFromDistribution(covariate_distribution)
    bn.cpt("X2").fillFromDistribution(covariate_distribution)
    bn.cpt("X3").fillFromDistribution(covariate_distribution)
    bn.cpt("X4").fillFromDistribution(covariate_distribution)
    bn.cpt("T").fillWith([0.5, 0.5])
    bn.cpt("Y").fillFromDistribution(norm, loc=expr, scale=1)

    return bn
```

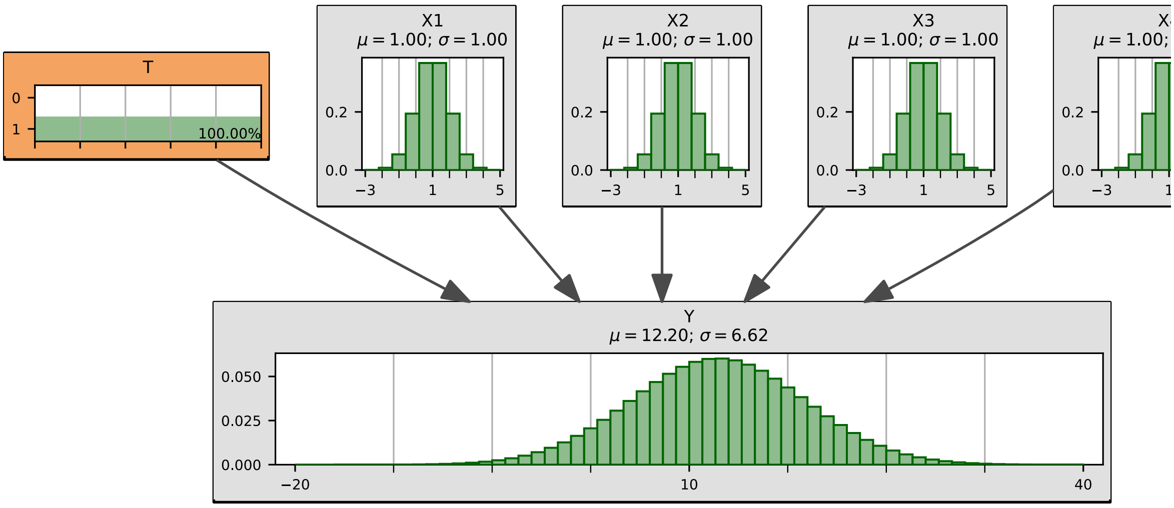
In [ ]:
```
def plotResults(Y_hat, Y, plot_title):
    """
    """
    plt.scatter(x=Y_hat[0]["interval_mean"] ,y=Y_hat[0]["probability"], \
                color="tab:blue", label="$\hat{Y}(0)$", s=10)
    plt.scatter(x=Y_hat[1]["interval_mean"] ,y=Y_hat[1]["probability"], \
                color="tab:orange", label="$\hat{Y}(1)$", s=10)
    plt.plot(Y["y0"], color="tab:blue", label="Y(0)")
    plt.plot(Y["y1"], color="tab:orange", label="Y(1)")
    plt.title(plot_title)
    plt.legend()
```

In [ ]:
```
lin_exbn = getBN(outcome_domain, covariate_domain, lin_expr)
nl_exbn = getBN(outcome_domain, covariate_domain, nl_expr)

# Runtime ~ 10 s
```

In [ ]:
```
gnb.sideBySide(gnb.showInference(lin_exbn,evs={"T":0},size="10"),
               gnb.showInference(lin_exbn,evs={"T":1},size="10"))
```



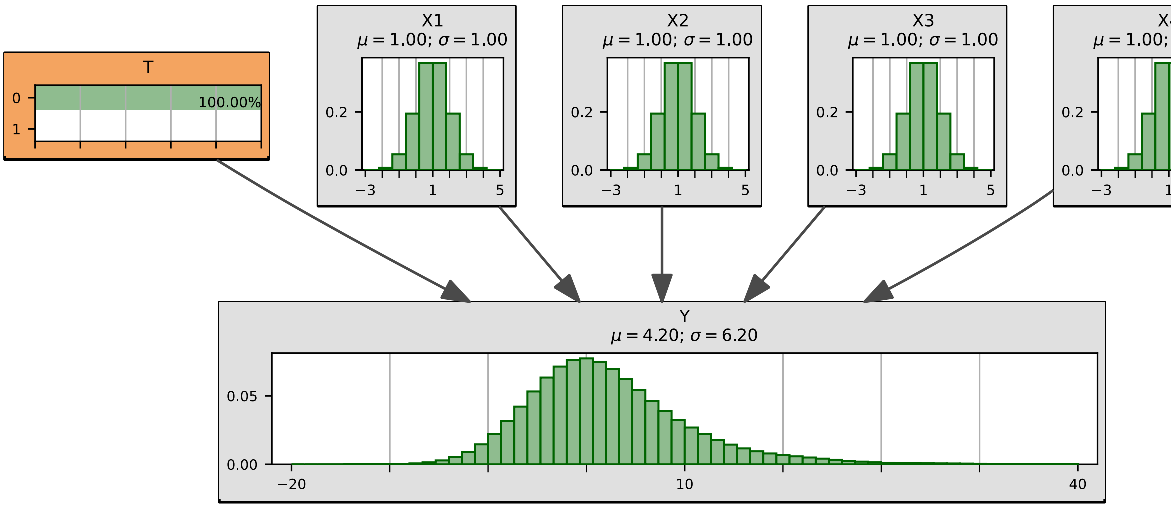
Inference in 5.52ms



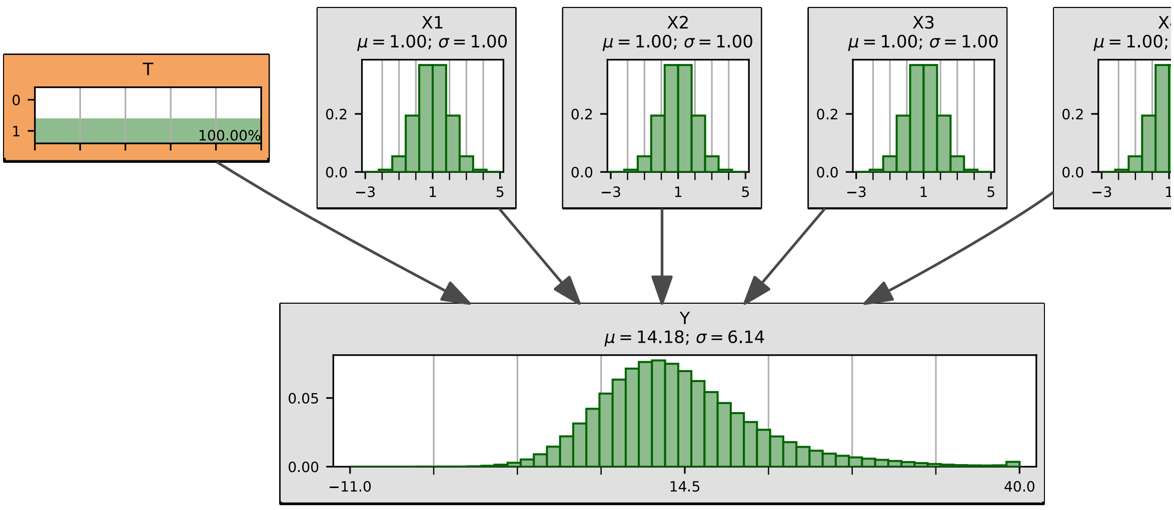
Inference in 5.89ms

None None

```
In [ ]: gnb.sideBySide(gnb.showInference(nl_exbn,evs={"T":0},size="10"),
                       gnb.showInference(nl_exbn,evs={"T":1},size="10"))
```



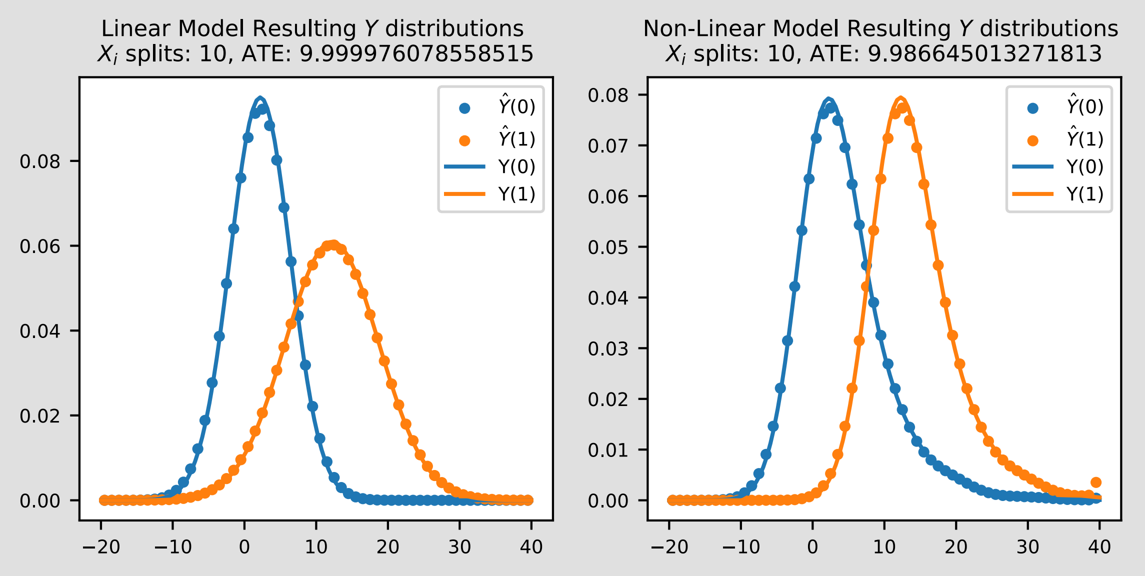
Inference in 10.52ms



Inference in 7.77ms

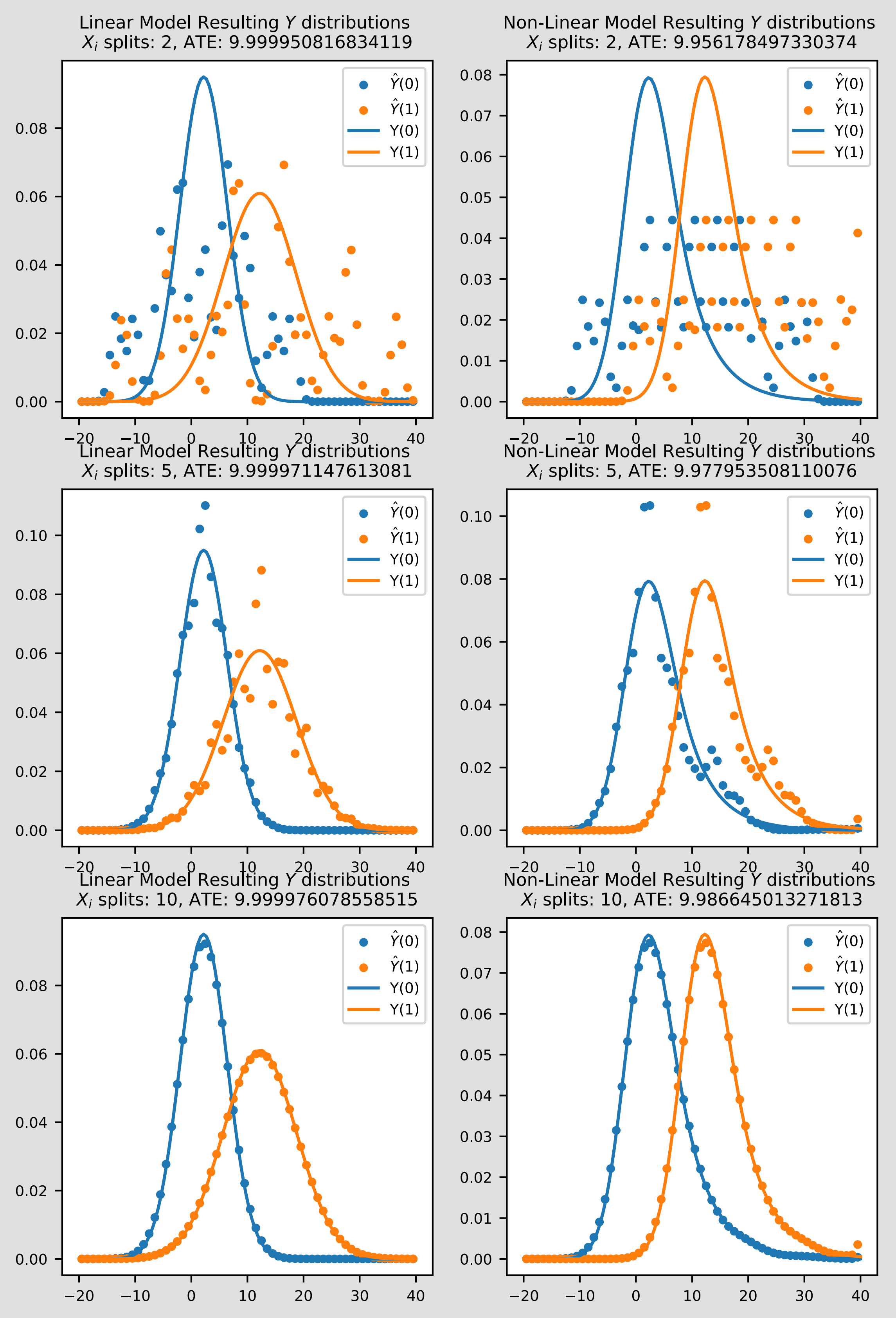
None None

In [ ]:
```
plt.subplots(figsize=(7, 3))

plt.subplot(1, 2, 1)

lin_Y = getY(lin_exbn)
plotResults(lin_Y, lin_pdf_df,
            f"Linear Model Resulting $Y$ distributions \n" \
            f"$X_i$ splits: {covariate_num_split}, ATE: {getTau(lin_Y)}")

plt.subplot(1, 2, 2)

nl_Y = getY(nl_exbn)
plotResults(nl_Y, nl_pdf_df,
            f"Non-Linear Model Resulting $Y$ distributions \n" \
            f"$X_i$ splits: {covariate_num_split}, ATE: {getTau(nl_Y)}")
```



In [ ]:
```
num_split_list = [2, 5, 10]
num_shots = 1

plt.subplots(figsize=(7, 3.5*len(num_split_list)))


for i in range(len(num_split_list)):

    covariate_num_split = num_split_list[i]
    covariate_domain = getSplits(covariate_start, covariate_end, \
                                 covariate_num_split)

    plt.subplot(len(num_split_list), 2, 2*i+1)

    lin_ex_bn = getBN(outcome_domain, covariate_domain, lin_expr)
    lin_Y_hat = getY(lin_ex_bn)
    plotResults(lin_Y_hat, lin_pdf_df,
                f"Linear Model Resulting $Y$ distributions \n" \
                f"$X_i$ splits: {covariate_num_split}, " \
                f"ATE: {getTau(lin_Y_hat)}")

    plt.subplot(len(num_split_list), 2, 2*i+2)

    nl_ex_bn = getBN(outcome_domain, covariate_domain, nl_expr)
    nl_Y_hat = getY(nl_ex_bn)
    plotResults(nl_Y_hat, nl_pdf_df,
```

```
                f"Non-Linear Model Resulting $Y$ distributions \n" \
                f"$X_i$ splits: {covariate_num_split}, " \
                f"ATE: {getTau(nl_Y_hat)}")

plt.show()
```

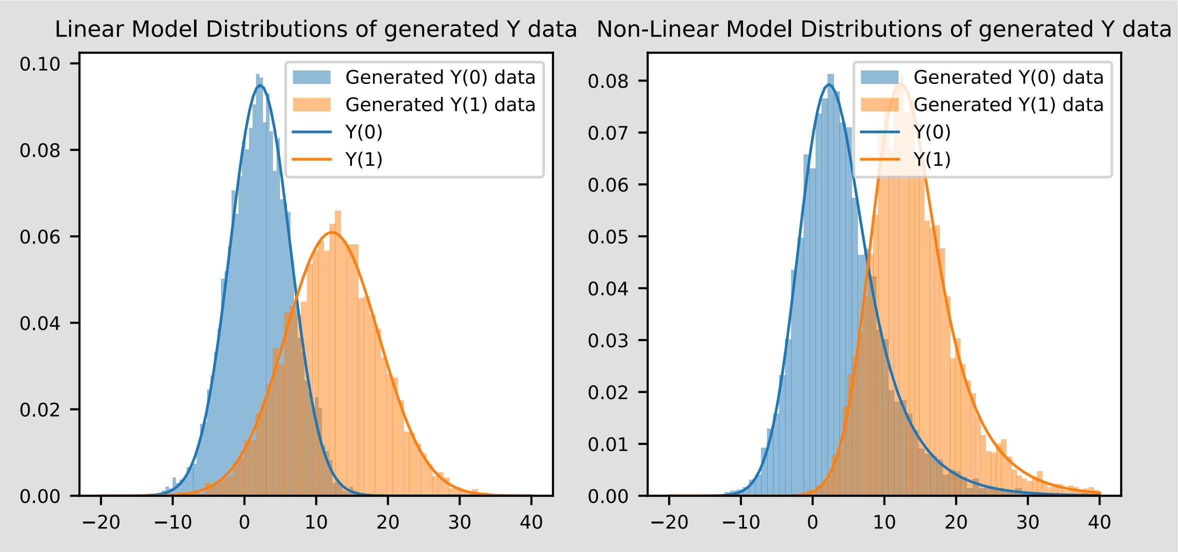


2 - Parametric Learning

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.

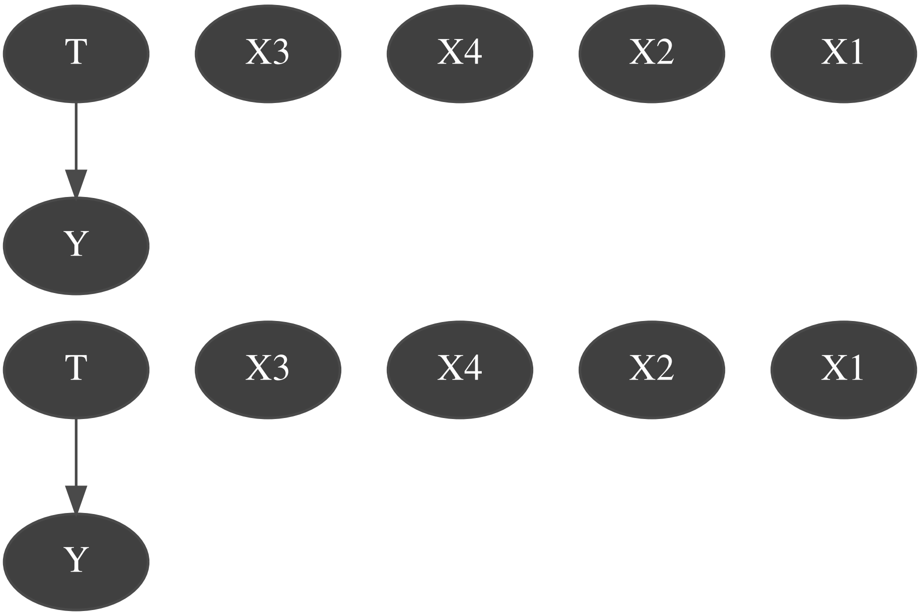
In [ ]:
```
def trimDataFrame(df, covariate_start, covariate_end, \
                  outcome_start, outcome_end):
    """
    """
    res = df.copy()
    res = res[(covariate_start<=res["X1"]) & (res["X1"]<=covariate_end)]
    res = res[(covariate_start<=res["X2"]) & (res["X2"]<=covariate_end)]
    res = res[(covariate_start<=res["X3"]) & (res["X3"]<=covariate_end)]
    res = res[(covariate_start<=res["X4"]) & (res["X4"]<=covariate_end)]
    res = res[(outcome_start<=res["Y"]) & (res["Y"]<=outcome_end)]
    return res
```

In [ ]:
```
lin_df = linear_simulation(10000, 1.0, 0.5)
nl_df = non_linear_simulation(10000, 1.0, 0.5)
lin_df = trimDataFrame(lin_df, covariate_start, covariate_end, \
                               outcome_start, outcome_end)
nl_df = trimDataFrame(nl_df, covariate_start, covariate_end, \
                             outcome_start, outcome_end)

plt.subplots(figsize=(7, 3))

plt.subplot(1, 2, 1)

plt.hist(lin_df[lin_df["T"] == 0]["Y"], bins=60, density=True, \
         alpha=0.5, edgecolor=None, label="Generated Y(0) data")
plt.hist(lin_df[lin_df["T"] == 1]["Y"], bins=60, density=True, \
         alpha=0.5, edgecolor=None, label="Generated Y(1) data")
plt.plot(lin_pdf_df["y0"], color="tab:blue", label="Y(0)", linewidth=1)
plt.plot(lin_pdf_df["y1"], color="tab:orange", label="Y(1)", linewidth=1)
plt.title("Linear Model Distributions of generated Y data")
plt.legend()

plt.subplot(1, 2, 2)

plt.hist(nl_df[nl_df["T"] == 0]["Y"], bins=60, density=True, \
         alpha=0.5, edgecolor=None, label="Generated Y(0) data")
plt.hist(nl_df[nl_df["T"] == 1]["Y"], bins=60, density=True, \
         alpha=0.5, edgecolor=None, label="Generated Y(1) data")
plt.plot(nl_pdf_df["y0"], color="tab:blue", label="Y(0)", linewidth=1)
plt.plot(nl_pdf_df["y1"], color="tab:orange", label="Y(1)", linewidth=1)
plt.title("Non-Linear Model Distributions of generated Y data")
plt.legend()

plt.show()
```



In [ ]:
```
lin_param_learner = gum.BNLearner(lin_df, lin_exbn)
lin_param_learner.useSmoothingPrior(1)
lin_param_learner.learnParameters(lin_exbn.dag())
lin_plbn = lin_param_learner.learnBN()

nl_param_learner = gum.BNLearner(nl_df, nl_exbn)
nl_param_learner.useSmoothingPrior(1)
nl_param_learner.learnParameters(nl_exbn.dag())
nl_plbn = nl_param_learner.learnBN()
```

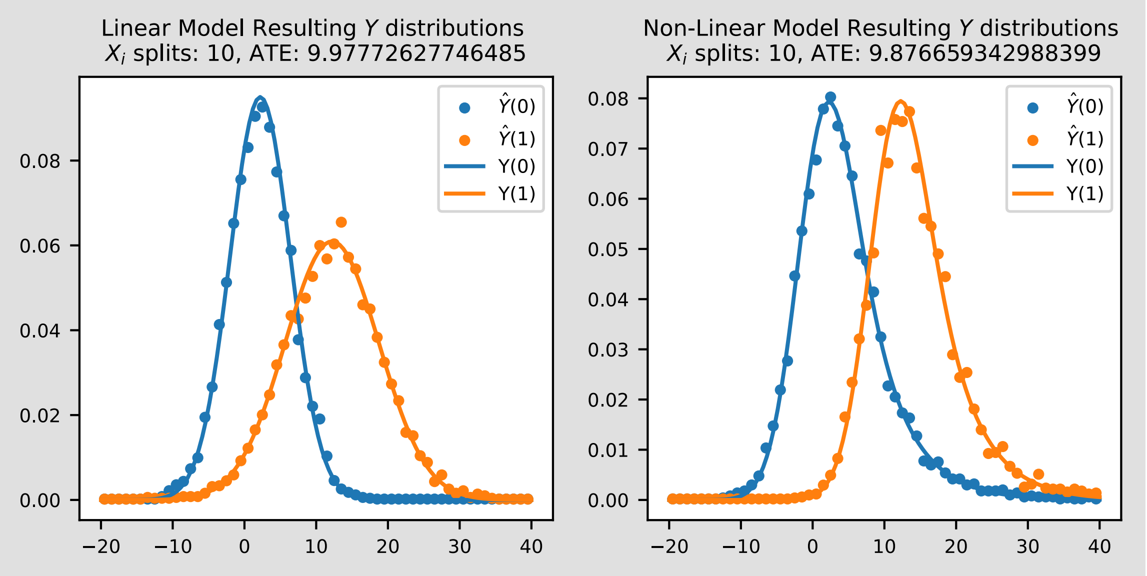
In [ ]:
```
gnb.sideBySide(gnb.show(lin_plbn), gnb.show(nl_plbn))
```



None None

In [ ]:
```
plt.subplots(figsize=(7, 3))

plt.subplot(1, 2, 1)

lin_Y = getY(lin_plbn)
plotResults(lin_Y, lin_pdf_df,
            f"Linear Model Resulting $Y$ distributions \n" \
            f"$X_i$ splits: {covariate_num_split}, ATE: {getTau(lin_Y)}")
```

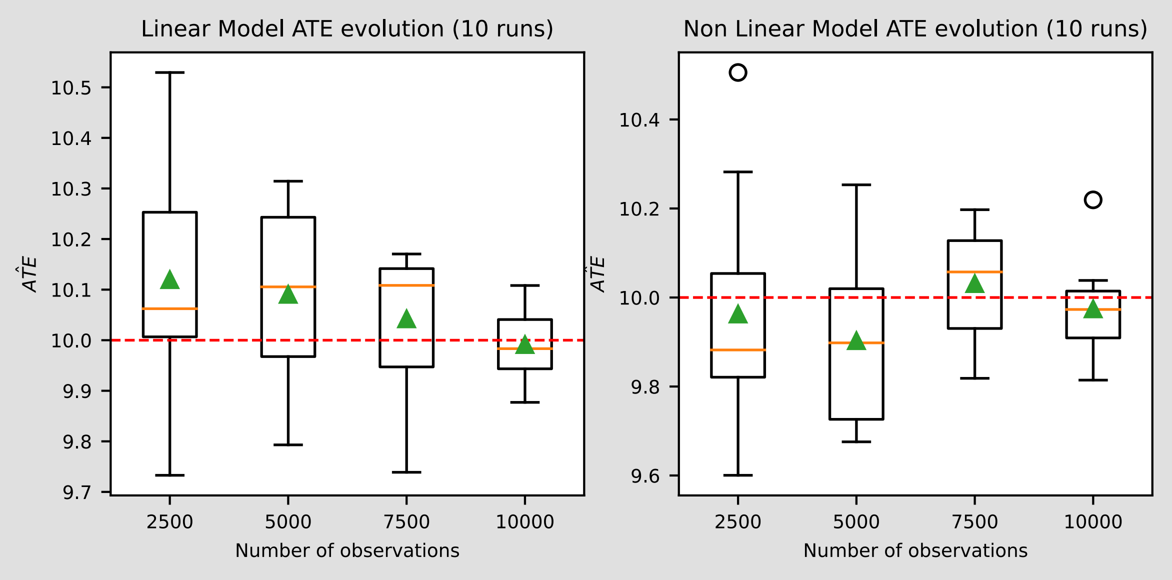
```
plt.subplot(1, 2, 2)

nl_Y = getY(nl_plbn)
plotResults(nl_Y, nl_pdf_df,
            f"Non-Linear Model Resulting $Y$ distributions \n" \
            f"$X_i$ splits: {covariate_num_split}, ATE: {getTau(nl_Y)}")
```



In [ ]:
```
lin_tau_hat_arr = []
nl_tau_hat_arr = []

num_obs_list = range(2500, 10001, 2500)
num_shots = 10

for i in num_obs_list:
    lin_tau_hat_arr.append(list())
    nl_tau_hat_arr.append(list())
    for j in range(num_shots):

        lin_df = linear_simulation(i, 1.0, 0.5)
        nl_df = non_linear_simulation(i, 1.0, 0.5)

        discretizer = skbn.BNDiscretizer("uniform", 30)

        lin_template = discretizer.discretizedBN(lin_df)
        lin_struct_learner = gum.BNLearner(lin_df, lin_template)
        lin_plbn = lin_struct_learner.learnBN()

        nl_template = discretizer.discretizedBN(nl_df)
        nl_struct_learner = gum.BNLearner(nl_df, nl_template)
        nl_plbn = nl_struct_learner.learnBN()

        lin_Y_hat = getY(lin_plbn)
        lin_tau_hat = getTau(lin_Y_hat)
        lin_tau_hat_arr[-1].append(lin_tau_hat)

        nl_Y_hat = getY(nl_plbn)
        nl_tau_hat = getTau(nl_Y_hat)
        nl_tau_hat_arr[-1].append(nl_tau_hat)
```

In [ ]:
```
plt.subplots(figsize=(7, 3))

plt.subplot(1, 2, 1)
```

```
plt.boxplot(lin_tau_hat_arr, labels=num_obs_list, meanline=False, \
            showmeans=True, showcaps=True)
plt.axhline(y=10, color='r', linestyle='--', linewidth=1)
plt.title(f"Linear Model ATE evolution ({num_shots} runs)")
plt.xlabel("Number of observations")
plt.ylabel("$\hat{ATE}$")

plt.subplot(1, 2, 2)

plt.boxplot(nl_tau_hat_arr, labels=num_obs_list, meanline=False, \
            showmeans=True, showcaps=True)
plt.axhline(y=10, color='r', linestyle='--', linewidth=1)
plt.title(f"Non Linear Model ATE evolution ({num_shots} runs)")
plt.xlabel("Number of observations")
plt.ylabel("$\hat{ATE}$")

plt.show()
```



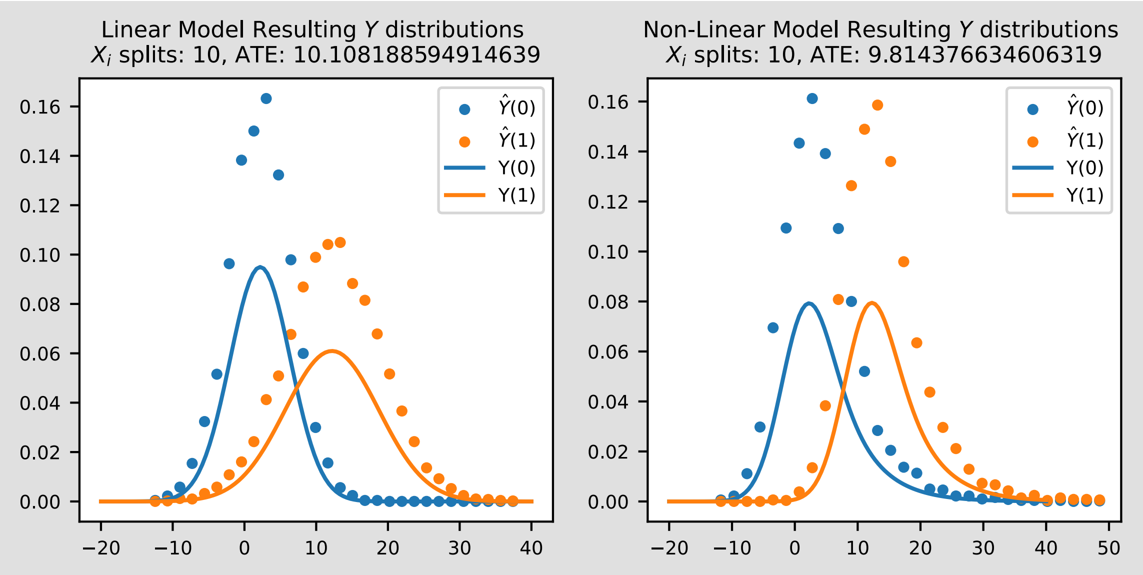
3 - Structural Learning

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

In [ ]:
```
discretizer = skbn.BNDiscretizer(defaultDiscretizationMethod="uniform", \
                                 defaultNumberOfBins=30)

lin_template = discretizer.discretizedBN(lin_df)
lin_struct_learner = gum.BNLearner(lin_df, lin_template)
lin_slbn = lin_struct_learner.learnBN()

nl_template = discretizer.discretizedBN(nl_df)
nl_struct_learner = gum.BNLearner(nl_df, nl_template)
nl_slbn = nl_struct_learner.learnBN()
```

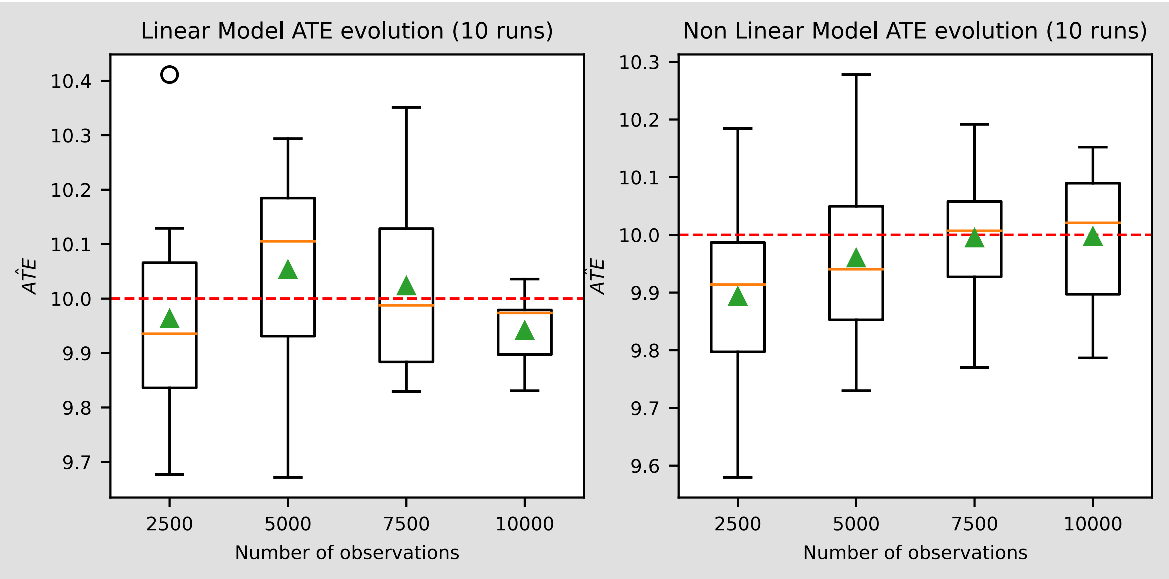
In [ ]:
```
plt.subplots(figsize=(7, 3))

plt.subplot(1, 2, 1)

lin_Y = getY(lin_slbn)
```

```
plotResults(lin_Y, lin_pdf_df,
            f"Linear Model Resulting $Y$ distributions \n" \
            f"$X_i$ splits: {covariate_num_split}, ATE: {getTau(lin_Y)}")

plt.subplot(1, 2, 2)

nl_Y = getY(nl_slbn)
plotResults(nl_Y, nl_pdf_df,
            f"Non-Linear Model Resulting $Y$ distributions \n" \
            f"$X_i$ splits: {covariate_num_split}, ATE: {getTau(nl_Y)}")
```



In [ ]:
```
lin_tau_hat_arr = []
nl_tau_hat_arr = []

num_obs_list = range(2500, 10001, 2500)
num_shots = 10

for i in num_obs_list:
    lin_tau_hat_arr.append(list())
    nl_tau_hat_arr.append(list())
    for j in range(num_shots):

        lin_df = linear_simulation(i, 1.0, 0.5)
        nl_df = non_linear_simulation(i, 1.0, 0.5)

        discretizer = skbn.BNDiscretizer("uniform", 30)

        lin_template = discretizer.discretizedBN(lin_df)
        lin_struct_learner = gum.BNLearner(lin_df, lin_template)
        lin_slbn = lin_struct_learner.learnBN()

        nl_template = discretizer.discretizedBN(nl_df)
        nl_struct_learner = gum.BNLearner(nl_df, nl_template)
        nl_slbn = nl_struct_learner.learnBN()

        lin_Y_hat = getY(lin_slbn)
        lin_tau_hat = getTau(lin_Y_hat)
        lin_tau_hat_arr[-1].append(lin_tau_hat)

        nl_Y_hat = getY(nl_slbn)
        nl_tau_hat = getTau(nl_Y_hat)
        nl_tau_hat_arr[-1].append(nl_tau_hat)
```

In [ ]:
```
plt.subplots(figsize=(7, 3))

plt.subplot(1, 2, 1)

plt.boxplot(lin_tau_hat_arr, labels=num_obs_list, meanline=False, \
            showmeans=True, showcaps=True)
plt.axhline(y=10, color='r', linestyle='--', linewidth=1)
plt.title(f"Linear Model ATE evolution ({num_shots} runs)")
plt.xlabel("Number of observations")
plt.ylabel("$\hat{ATE}$")

plt.subplot(1, 2, 2)

plt.boxplot(nl_tau_hat_arr, labels=num_obs_list, meanline=False, \
            showmeans=True, showcaps=True)
plt.axhline(y=10, color='r', linestyle='--', linewidth=1)
plt.title(f"Non Linear Model ATE evolution ({num_shots} runs)")
plt.xlabel("Number of observations")
plt.ylabel("$\hat{ATE}$")

plt.show()
```



In [ ]: