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 on generated and real data through three different methods:

- "Exact" Computation
- Parameter Learning
- Structure Learning

In [ ]:
```
import pyAgrum as gum
import pyAgrum.skbn as skbn
import pyAgrum.lib.notebook as gnb
import pyAgrum.lib.explain as gexpl

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

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

1 - Generated Data

We will first 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 $(X_1, X_2, X_3, X_4) \sim \mathcal{N}_4((1,1,1,1), I_4)$, $T \sim \mathcal{B}er(1/2)$ 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 [ ]:
```python
def linear_simulation(n : int, sigma : float) -> pd.DataFrame:
  """
  Returns n observations from the linear model with normally distributed
  noise with expected value 0 and standard deviation sigma.
  """

  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, 0.5, 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 : int, sigma : float) -> pd.DataFrame:
  """
  Returns n observations from the non-linear model with normally distribu
  noise with expected value 0 and standard deviation sigma.
  """

  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, 0.5, 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
```

Furthermore, the expected values of $Y(0)$ and $Y(1)$ can be explicitly calculated,

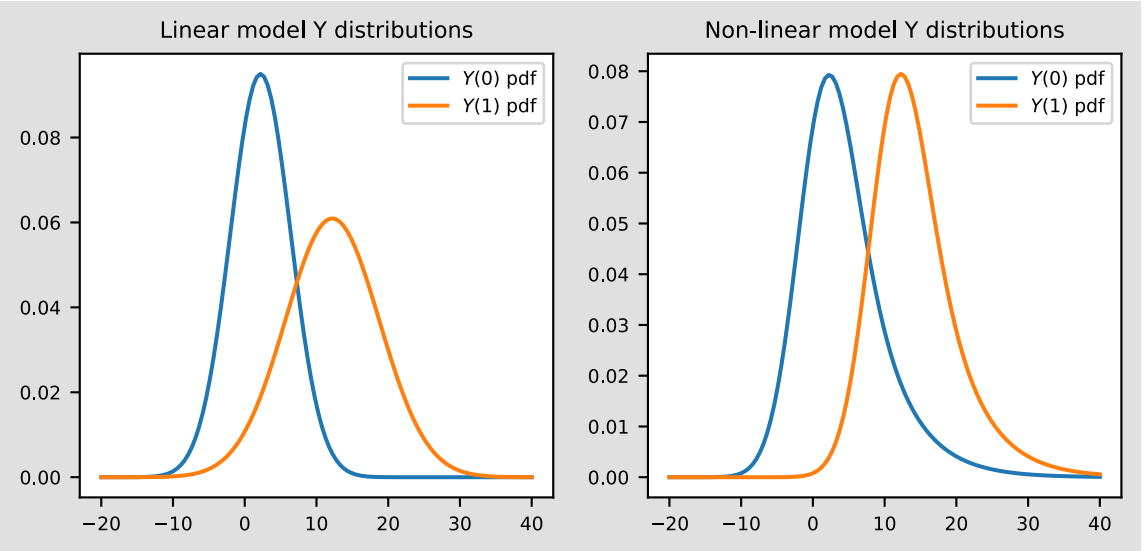
providing us the theoretical ATE which enables performance evaluations of the estimators.

Both models have an ATE of $\tau = 10$

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, lin_y0_var = (2.2, 17.64)
lin_y1_mean, lin_y1_var = (12.2, 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, nl_y0_norm_var = (0.2, 13.64)
nl_y1_norm_mean, nl_y1_norm_var = (10.2, 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, nl_y1 = (np.array(nl_y0), np.array(nl_y1))
nl_y0, nl_y1 = (nl_y0/(nl_y0.sum()*dx), nl_y1/(nl_y1.sum()*dx))

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

# Runtime ~ 1 min
```

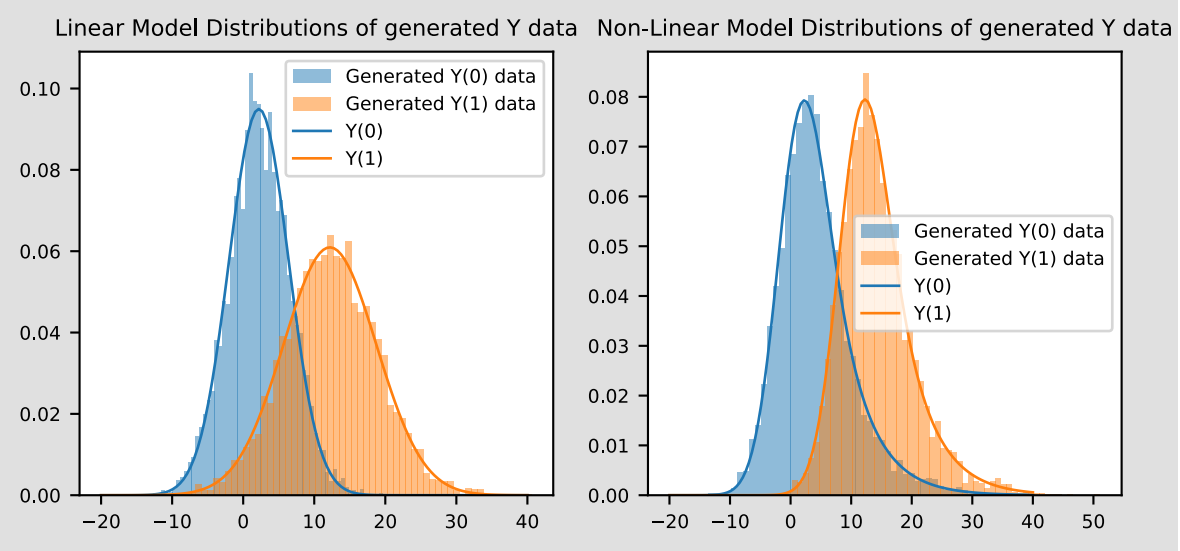
In [ ]:
```
# Plotting the distributions

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 [ ]:
```
# Visualisation of generated data used for Learning

lin_df = linear_simulation(10000, 1.0)
nl_df = non_linear_simulation(10000, 1.0)

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 [ ]:
```
# 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"
```

1.1 - "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 [ ]:
```
# Definitions of functions used in this section

def getStringIntervalMean(interval_string : str) -> float:
    """
    Returns the mean of a interval casted as string (e.g. [1.5, 2.9[).
    """

    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) -> tuple[pd.DataFrame]:
    """
    Returns the estimation of outcomes Y(0), Y(1) with Lazy Propagation
    from the inputed Baysian Network as a pandas Data Frame couple.
    """

    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 : tuple[pd.DataFrame]) -> float:
    """
    Returns estimation of the ATE tau from pandas Data Frame couple
    (Y(0), Y(1)).
    """

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

    return tau

def getBN(data : pd.DataFrame, covariate_num_split : int,
          outcome_num_split : int, covariate_distribution = None,
          expr : str = None, add_arcs : bool = True) -> gum.BayesNet:
    """
    Returns Baysian Network corresponding to the model by discretising
    countinous variables with given parameters.
    """

    disc = skbn.BNDiscretizer()

    disc.setDiscretizationParameters("X1", 'uniform', covariate_num_split
    disc.setDiscretizationParameters("X2", 'uniform', covariate_num_split
    disc.setDiscretizationParameters("X3", 'uniform', covariate_num_split
    disc.setDiscretizationParameters("X4", 'uniform', covariate_num_split
    disc.setDiscretizationParameters("T", 'NoDiscretization', [0, 1])
    disc.setDiscretizationParameters("Y", 'uniform', outcome_num_split)

    bn = disc.discretizedBN(data)
```

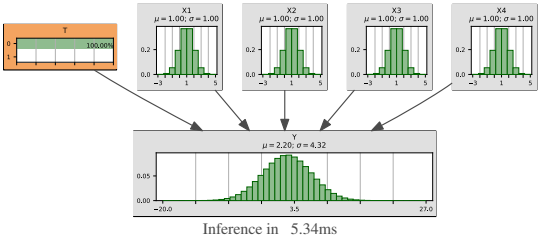
```
    if add_arcs :
        for _, name in bn:
            if name != "Y":
                bn.addArc(name, "Y")

    if covariate_distribution is not None :
        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])
    if expr is not None:
        bn.cpt("Y").fillFromDistribution(norm, loc=expr, scale=1)

    return bn

def plotResults(Y_hat : pd.DataFrame, Y : pd.DataFrame,
                plot_title : str) -> None:
    """
    Scatters Y_hat data and plots Y data in a plot titled 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 [ ]:
```
# Covariate parameters

covariate_start = -3.0
covariate_end = 5.0
covariate_num_split = 10

covariate_distribution = norm(loc=1, scale=1)

# Outcome parameters

outcome_start = -20.0
outcome_end = 40.0
outcome_num_split = 60
```

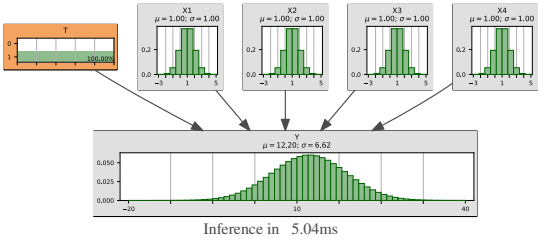
By employing a 10-bin discretization for the covariates and a 60-bin discretization for the outcome, the Bayesian Network estimator accurately approximates the true distribution for both the treated and untreated outcome.

In [ ]:
```
dummy_df = pd.DataFrame({"T": [0, 1], \
                         "X1": [covariate_start, covariate_end], \
                         "X2": [covariate_start, covariate_end], \
                         "X3": [covariate_start, covariate_end], \
                         "X4": [covariate_start, covariate_end], \
                         "Y" : [outcome_start, outcome_end]})
```

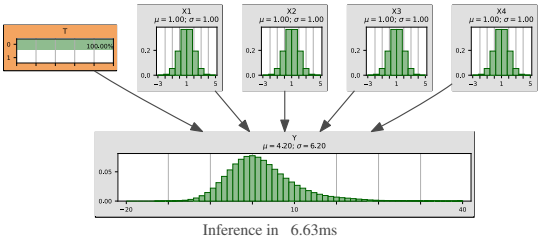
```
lin_exbn = getBN(dummy_df, covariate_num_split, outcome_num_split, \
                 covariate_distribution, lin_expr)

nl_exbn  = getBN(dummy_df, covariate_num_split, outcome_num_split, \
                 covariate_distribution, nl_expr)

gnb.sideBySide(gnb.getInference(lin_exbn, evs={"T":0}, size="10"), \
               gnb.getInference(lin_exbn, evs={"T":1}, size="10"), \
               captions=["Linear Y(0)", "Linear Y(1)"])
gnb.sideBySide(gnb.getInference(nl_exbn, evs={"T":0}, size="10"), \
               gnb.getInference(nl_exbn, evs={"T":1}, size="10"), \
               captions=["Non-Linear Y(0)", "Non-Linear Y(1)"])
```



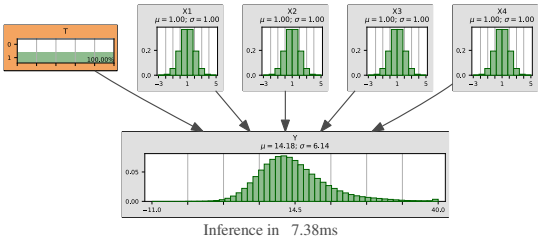
Linear Y(0)



Linear Y(1)



Non-Linear Y(0)

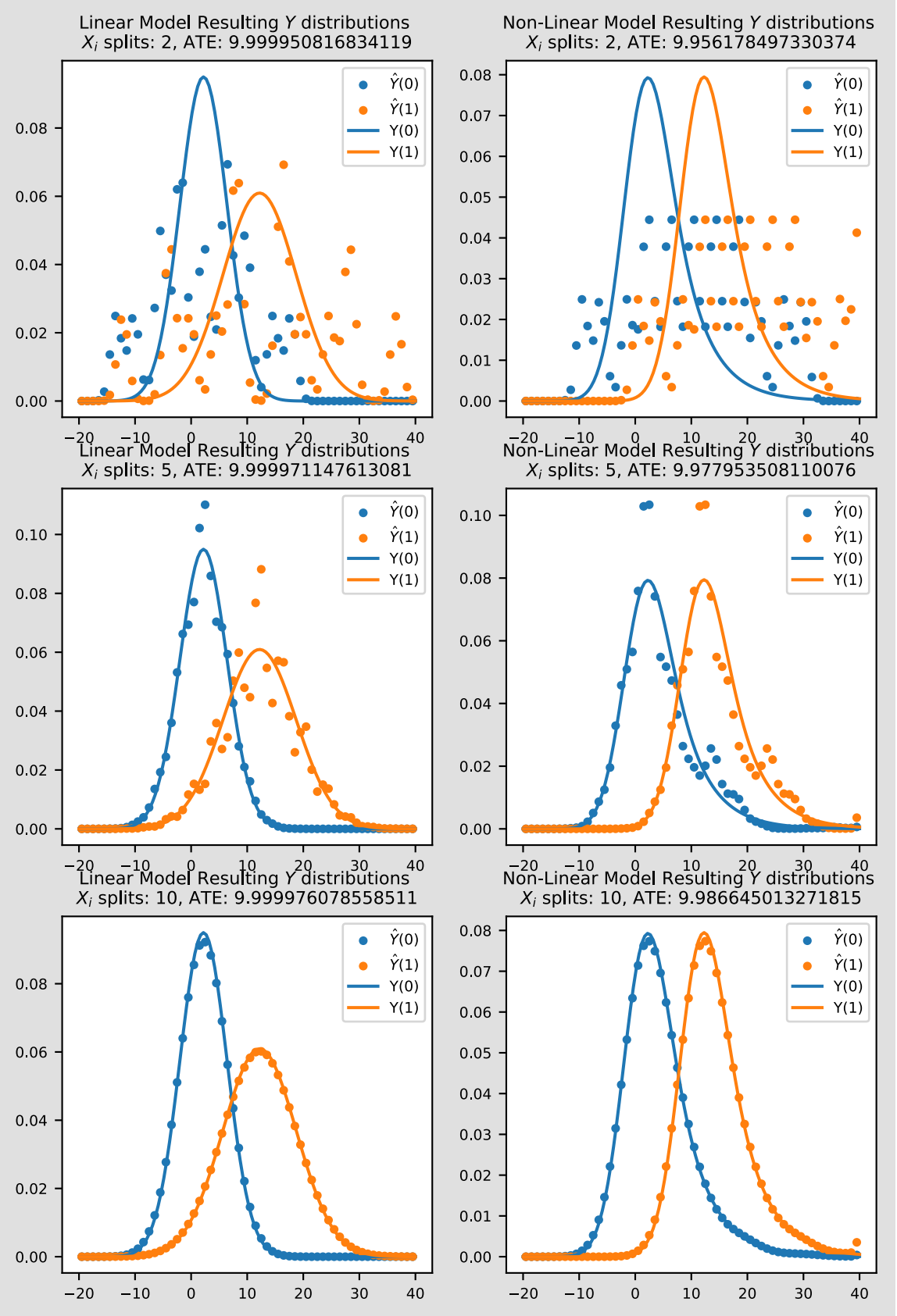


Non-Linear Y(1)

Let's examine how the fineness of covariate discretization impacts the outcome distribution.

In [ ]:
```
num_split_list = [2, 5, 10]
gnb.sideBySide(gnb.getInference(lin_slbn,evs={"T":0},size="10"), \
               gnb.getInference(lin_slbn,evs={"T":1},size="10"), \
               gnb.getInference(nl_slbn, evs={"T":0},size="10"), \
               gnb.getInference(nl_slbn, evs={"T":1},size="10"), \
               captions=["Linear Y(0)", "Linear Y(1)", \
                         "Non-Linear Y(0)", "Non-Linear Y(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]

    # Linear Model

    plt.subplot(len(num_split_list), 2, 2*i+1)
    lin_bn_params = [dummy_df, covariate_num_split, outcome_num_split, \
                     covariate_distribution, lin_expr]
    lin_ex_bn = getBN(*lin_bn_params)
    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)}")

    # Non Linear Model

    plt.subplot(len(num_split_list), 2, 2*i+2)
    nl_bn_params = [dummy_df, covariate_num_split, outcome_num_split, \
                    covariate_distribution, nl_expr]

    nl_ex_bn = getBN(*nl_bn_params)
    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()
```

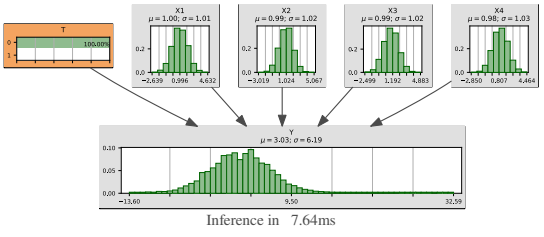


1.2 - Parameter Learning

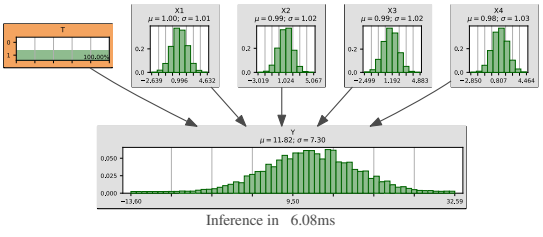
Given the data generating function defined above, parameter learning methods can be utilized to infer the underlying distribution based on the given structure of the Bayesian network.

In [ ]:
```
# Linear Model
```

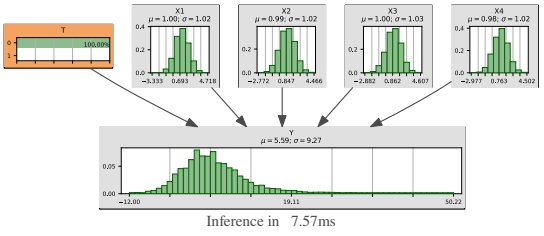
```
lin_template = getBN(lin_df, covariate_num_split, outcome_num_split)

lin_p_learner = gum.BNLearner(lin_df, lin_template)
lin_p_learner.useNMLCorrection()
lin_p_learner.useSmoothingPrior(1e-6)

lin_plbn = gum.BayesNet(lin_template)
lin_p_learner.fitParameters(lin_plbn)

# Non-Linear Model

nl_template = getBN(nl_df, covariate_num_split, outcome_num_split)

nl_p_learner = gum.BNLearner(nl_df, nl_template)
nl_p_learner.useNMLCorrection()
nl_p_learner.useSmoothingPrior(1e-6)

nl_plbn = gum.BayesNet(nl_template)
nl_p_learner.fitParameters(nl_plbn)

gnb.sideBySide(gnb.getInference(lin_plbn, evs={"T":0}, size="10"), \
               gnb.getInference(lin_plbn, evs={"T":1}, size="10"), \
               captions=["Linear Y(0)", "Linear Y(1)"])
gnb.sideBySide(gnb.getInference(nl_plbn, evs={"T":0}, size="10"), \
               gnb.getInference(nl_plbn, evs={"T":1}, size="10"), \
               captions=["Non-Linear Y(0)", "Non-Linear Y(1)"])
```



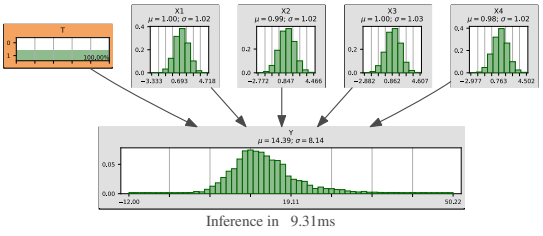
Linear Y(0)



Linear Y(1)



Non-Linear Y(0)

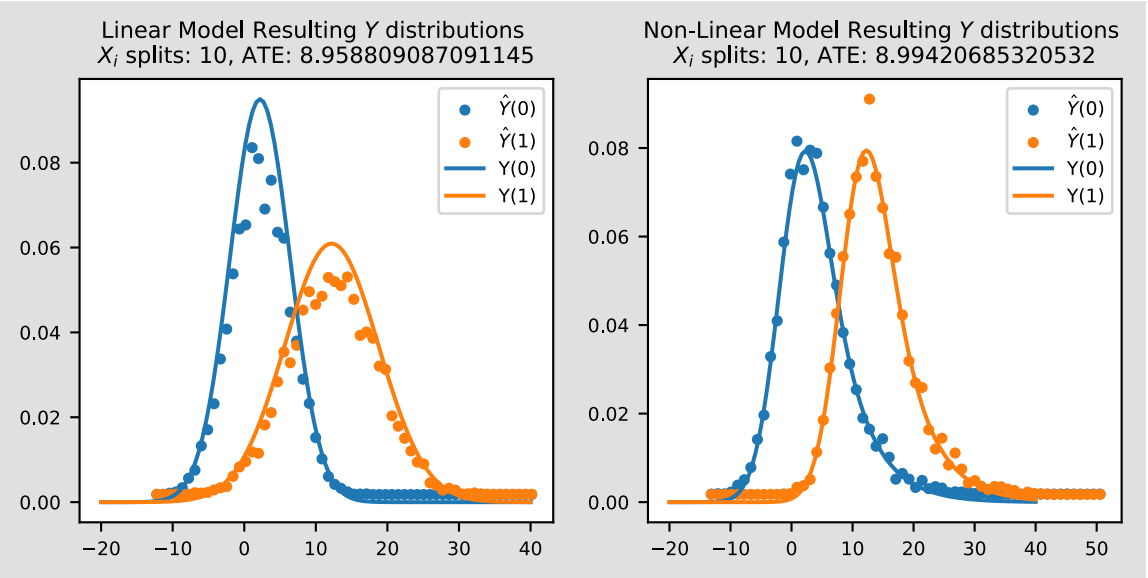


Non-Linear Y(1)

We observe that the inferred outcome distribution generally matches the exact distribution. However, the Average Treatment Effect (ATE) seems to be biased, as it is consistently smaller.

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)}")
```

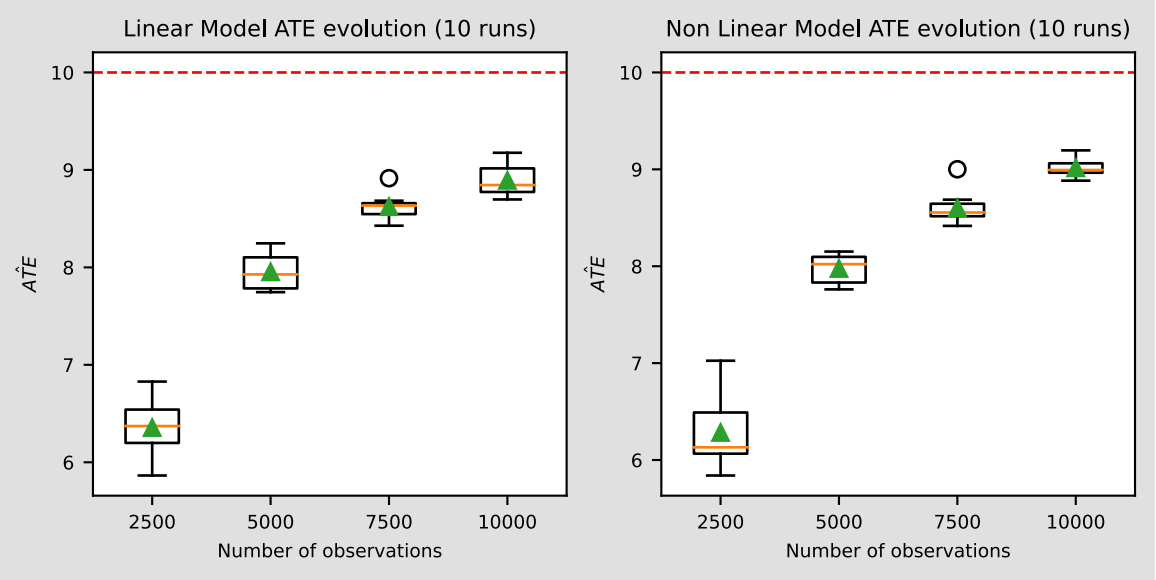


This underestimation can be further observed with varying numbers of observations in both models.

In [ ]:
```
lin_tau_hat_arr = list()
nl_tau_hat_arr = list()

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)
        nl_df = non_linear_simulation(i, 1.0)

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

        # Linear Model

        lin_template = getBN(lin_df, covariate_num_split, \
                             outcome_num_split, add_arcs=True)

        lin_p_learner = gum.BNLearner(lin_df, lin_template)
        lin_p_learner.useNMLCorrection()
        lin_p_learner.useSmoothingPrior(1e-6)
        lin_p_learner.setSliceOrder([["T"],["X1","X2","X3","X4"],["Y"]])

        lin_plbn = gum.BayesNet(lin_template)
        lin_p_learner.fitParameters(lin_plbn)
```

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

        # Non-Linear Model

        nl_template = getBN(nl_df, covariate_num_split, \
                            outcome_num_split, add_arcs=True)

        nl_p_learner = gum.BNLearner(nl_df, nl_template)
        nl_p_learner.useNMLCorrection()
        nl_p_learner.useSmoothingPrior(1e-6)
        nl_p_learner.setSliceOrder([["T"],["X1","X2","X3","X4"],["Y"]])

        nl_plbn = gum.BayesNet(nl_template)
        nl_p_learner.fitParameters(nl_plbn)

        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()
```

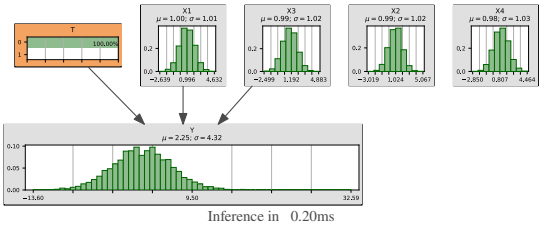


1.3 - Structure Learning

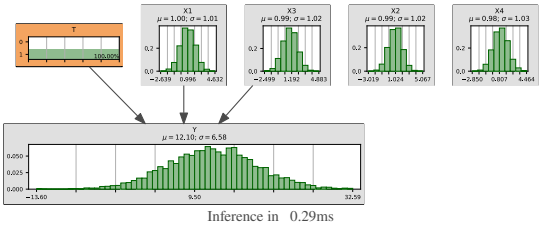
It is possible to derive the network's structure and distributions of the variables from a sufficiently large dataset through non-parametric learning methods. We will, however, impose a slice order on the learner to ensure the integrity of the process, prioritizing the outcome variable.

In [ ]:
```
lin_template = getBN(lin_df, covariate_num_split, \
                     outcome_num_split, add_arcs=False)
lin_s_learner = gum.BNLearner(lin_df, lin_template)
lin_s_learner.useNMLCorrection()
lin_s_learner.useSmoothingPrior(1e-6)
lin_s_learner.setSliceOrder([["T"],["X1","X2","X3","X4"],["Y"]])
lin_slbn = lin_s_learner.learnBN()

nl_template = getBN(nl_df, covariate_num_split, \
                    outcome_num_split, add_arcs=False)
```

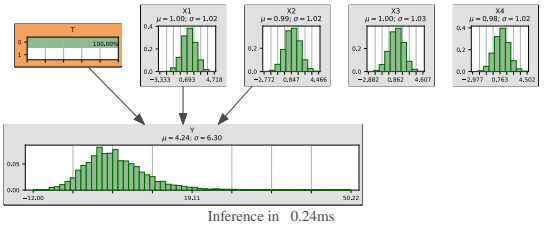
```
nl_s_learner = gum.BNLearner(nl_df, nl_template)
nl_s_learner.useNMLCorrection()
nl_s_learner.useSmoothingPrior(1e-6)
nl_s_learner.setSliceOrder([["T"],["X1","X2","X3","X4"],["Y"]])
nl_slbn = nl_s_learner.learnBN()

gnb.sideBySide(gnb.getInference(lin_slbn, evs={"T":0}, size="10"), \
               gnb.getInference(lin_slbn, evs={"T":1}, size="10"), \
               captions=["Linear Y(0)", "Linear Y(1)"])
gnb.sideBySide(gnb.getInference(nl_slbn, evs={"T":0}, size="10"), \
               gnb.getInference(nl_slbn, evs={"T":1}, size="10"), \
               captions=["Non-Linear Y(0)", "Non-Linear Y(1)"])
```



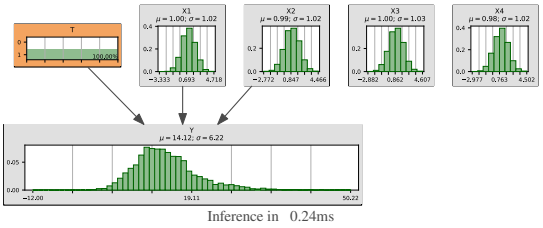
Linear Y(0)



Linear Y(1)



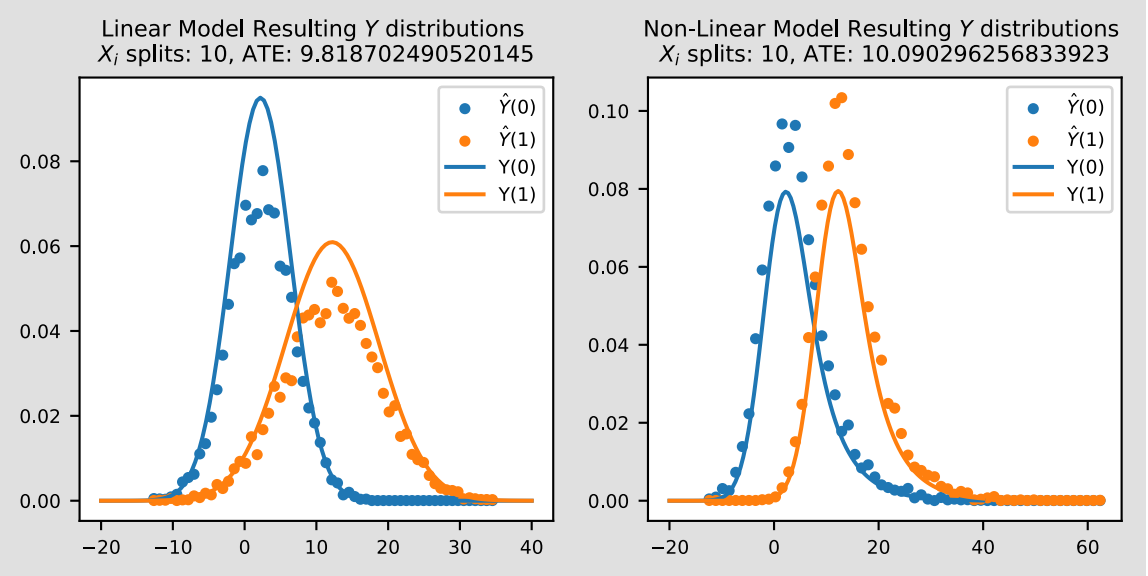
Non-Linear Y(0)



Non-Linear Y(1)

We achieve a more accurate estimation of the ATE using Structure Learning compared to Parameter Learning. This improvement can potentially be explained by the suboptimal structure imposed on the Bayesian Network for the generated dataset.

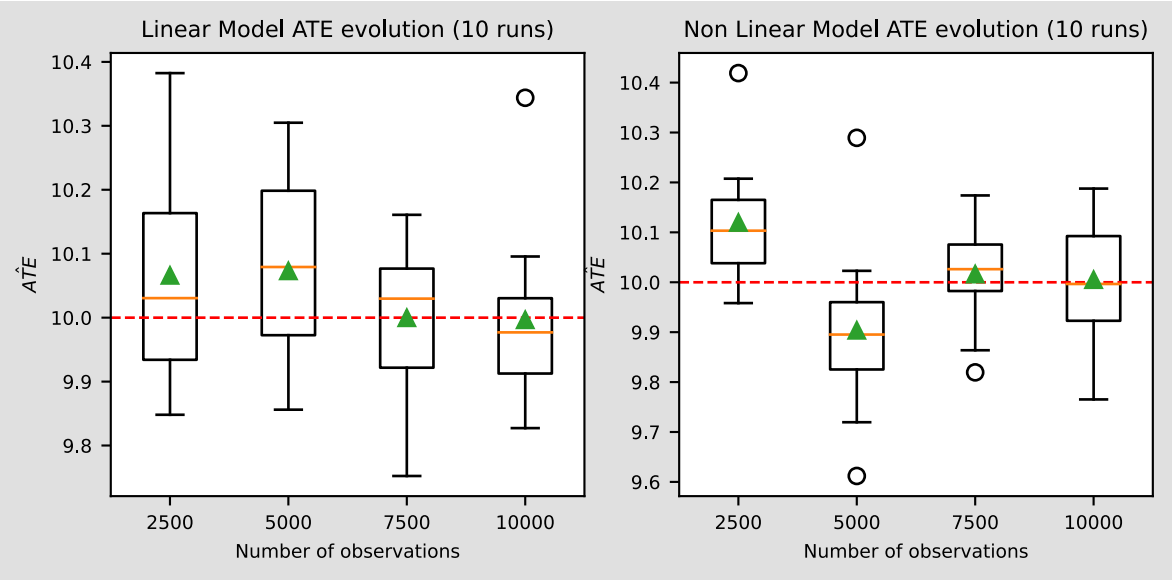
In [ ]:
```
plt.subplots(figsize=(7, 3))

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)
        nl_df = non_linear_simulation(i, 1.0)

        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()
```



2 - Real Data

After evaluating various estimation methods using generated data, we will now direct our attention to real data from the Tennessee Student/Teacher Achievement Ratio (STAR) trial. This randomized controlled trial, initiated in 1985, is a pioneering study in the field of education, designed to assess the effects of smaller class sizes in primary schools (T) on students' academic performance (Y).

The covariates in this study include:

- `gender`
- `age`
- `g1freelunch` being the number of lunchs provided to the child per day
- `g1surban` the localisation of the school (inner city or rural)
- `ethnicity`

In [ ]:
```
# Preprocessing

# Load data - read everything as a string and then cast
star_df = pd.read_csv("./STAR_data.csv", sep=",", dtype=str)
star_df = star_df.rename(columns={"race": "ethnicity"})

# Fill na
star_df = star_df.fillna({"g1freelunch": 0, "g1surban": 0})
drop_star_l = ["g1tlistss", "g1treadss", "g1tmathss", "g1classtype",
"birthyear", "birthmonth", "birthday", "gender",
```

```
"ethnicity", "g1freelunch", "g1surban"]
star_df = star_df.dropna(subset=drop_star_l, how='any')

# Cast value types before processing
star_df["gender"] = star_df["gender"].astype(int)
star_df["ethnicity"] = star_df["ethnicity"].astype(int)

star_df["g1freelunch"] = star_df["g1freelunch"].astype(int)
star_df["g1surban"] = star_df["g1surban"].astype(int)
star_df["g1classtype"] = star_df["g1classtype"].astype(int)

# Keep only class type 1 and 2 (in the initial trial,
# 3 class types where attributed and the third one was big classes
# but with a teaching assistant)
star_df = star_df[~(star_df["g1classtype"] == 3)].reset_index(drop=True)

# Compute the outcome
star_df["Y"] = (star_df["g1tlistss"].astype(int) + \
                star_df["g1treadss"].astype(int) + \
                star_df["g1tmathss"].astype(int)) / 3

# Compute the treatment
star_df["T"] = star_df["g1classtype"].apply(lambda x: 0 if x == 2 \
                                                    else 1)

# Transform date to obtain age (Notice: if na --> date is NaT)
star_df["date"] = pd.to_datetime(star_df["birthyear"] + "/"
+ star_df["birthmonth"] + "/"
+ star_df["birthday"], yearfirst=True, errors="coerce")
star_df["age"] = (np.datetime64("1985-01-01") - star_df["date"])
star_df["age"] = star_df["age"].dt.days / 365.25

# Keep only covariates we consider predictive of the outcome
star_covariates_l = ["gender", "ethnicity", "age", \
                     "g1freelunch", "g1surban"]
star_df = star_df[["Y", "T"] + star_covariates_l]

# Map numerical to categorical
star_df["gender"] = star_df["gender"].apply(lambda x: "Girl" if x == 2 \
                                                 else "Boy").astype("category"
star_df["ethnicity"] = star_df["ethnicity"].map( \
    {1:"White", 2:"Black", 3:"Asian", \
     4:"Hispanic",5:"Nat_American", 6:"Other"}).astype("category")
star_df["g1surban"] = star_df["g1surban"].map( \
    {1:"Inner_city", 2:"Suburban", \
     3:"Rural", 4:"Urban"}).astype("category")

star_df.head()
```

Out[ ]:

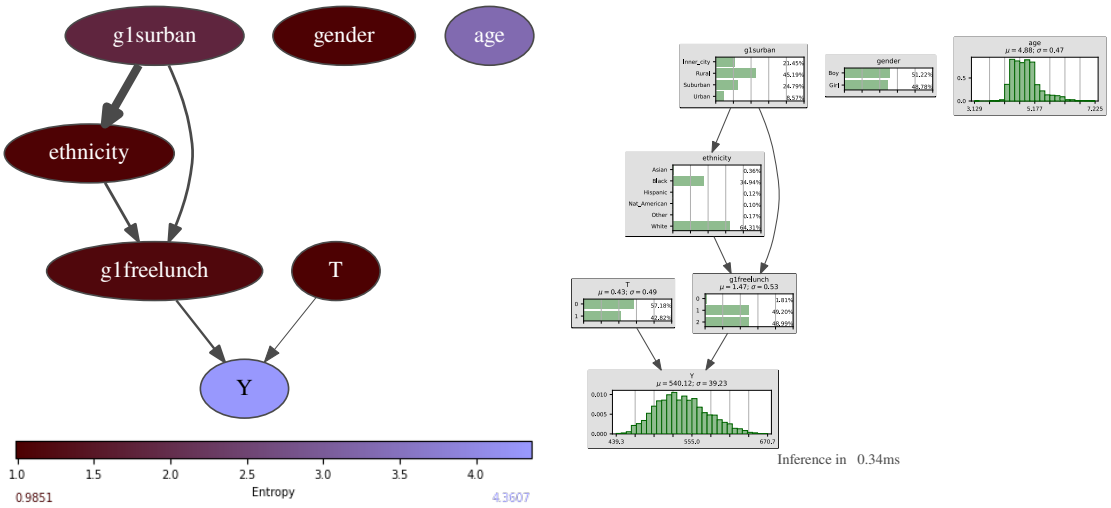
	Y	T	gender	ethnicity	age	g1freelunch	g1surban
0	514.000000	0	Boy	White	4.596851	2	Rural
1	512.666667	0	Girl	Black	5.694730	1	Inner_city
2	470.333333	1	Girl	Black	4.180698	1	Suburban
3	500.666667	1	Girl	White	5.963039	2	Urban
4	516.333333	0	Boy	Black	5.867214	1	Inner_city

2.1 - Structure Learning

In [ ]:
```
disc = skbn.BNDiscretizer(defaultDiscretizationMethod='uniform')
disc.setDiscretizationParameters("age", 'uniform', 24)
disc.setDiscretizationParameters("Y", 'uniform', 30)

template = disc.discretizedBN(star_df)

learner = gum.BNLearner(star_df, template)
learner.useNMLCorrection()
learner.useSmoothingPrior(1e-6)
learner.setSliceOrder([["T"], ["gender", "age", "g1surban", \
                        "g1freelunch", "ethnicity"], ["Y"]])
star_slbn = learner.learnBN()

print(learner)
```

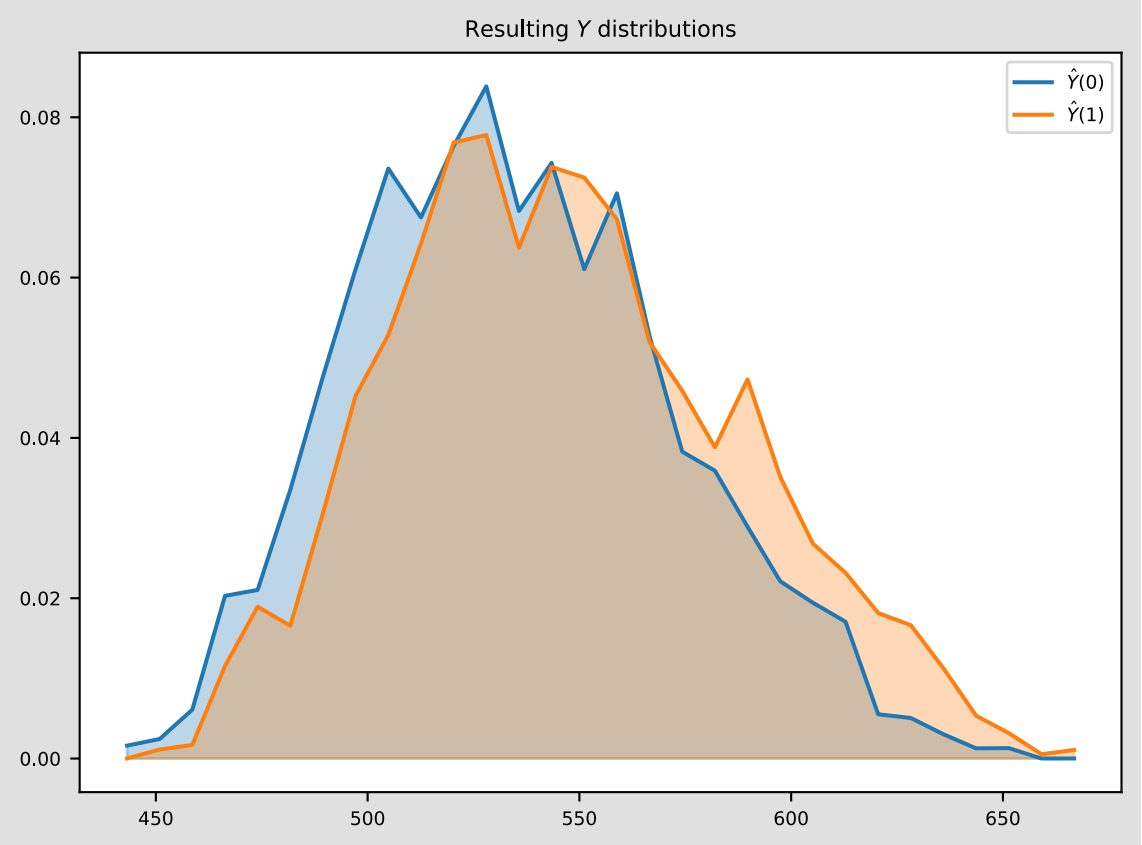
```
Filename               : /tmp/tmprzm2b5x6.csv
Size                   : (4215,7)
Variables              : Y[30], T[2], gender[2], ethnicity[6], age[24], g1
freelunch[3], g1surban[4]
Induced types          : False
Missing values         : False
Algorithm              : MIIC
Score                  : BDeu  (Not used for constraint-based algorithms)
Correction             : NML  (Not used for score-based algorithms)
Prior                  : Smoothing
Prior weight           : 0.000001
Constraint Slice Order : {ethnicity:1, T:0, g1surban:1, age:1, gender:1, g
1freelunch:1, Y:2}
```

In [ ]:
```
gnb.sideBySide(gexpl.getInformation(star_slbn, size="50"), gnb.getInferen
```



In [ ]:
```
Y_hat = getY(star_slbn)

x0, y0 = (Y_hat[0]["interval_mean"].to_numpy(), \
          Y_hat[0]["probability"].to_numpy())
x1, y1 = (Y_hat[1]["interval_mean"].to_numpy(), \
          Y_hat[1]["probability"].to_numpy())

plt.figure(figsize=(7, 5))
```

```
plt.fill_between(x0, y0, alpha=0.3, color='tab:blue')
plt.plot(x0, y0, color="tab:blue", label="$\hat{Y}(0)$")

plt.fill_between(x1, y1, alpha=0.3, color='tab:orange')
plt.plot(x1, y1, color="tab:orange", label="$\hat{Y}(1)$")
plt.legend()
plt.title("Resulting $Y$ distributions")
plt.show()

print(f"Estimated ATE : {getTau(Y_hat)}")
```




```
Estimated ATE : 11.513756264620497
```

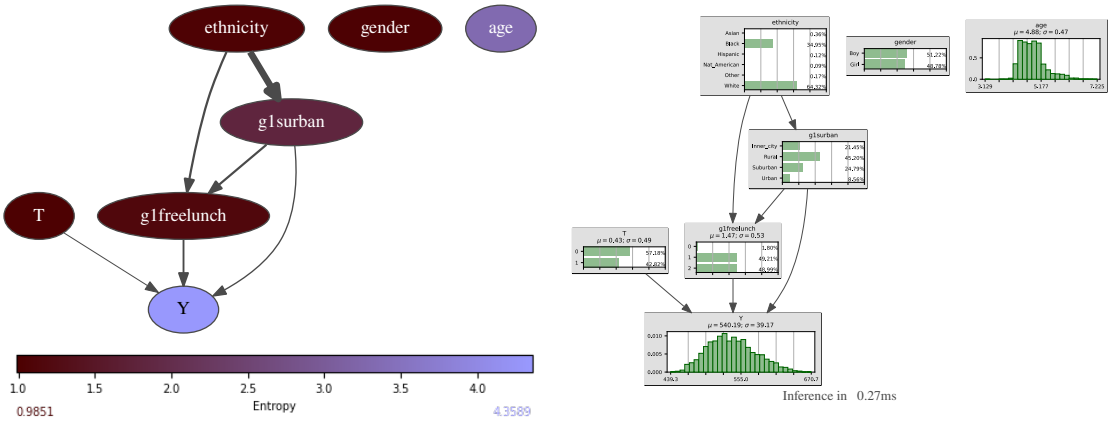
2.2 - Parameter Learning

In [ ]:
```
disc = skbn.BNDiscretizer(defaultDiscretizationMethod='uniform')
disc.setDiscretizationParameters("age", 'uniform', 24)
disc.setDiscretizationParameters("Y", 'uniform', 30)

template = disc.discretizedBN(star_df)

learner = gum.BNLearner(star_df, template)
learner.useNMLCorrection()
learner.useSmoothingPrior(1e-6)

star_plbn = gum.BayesNet(template)
star_plbn.addArc("T","Y")
star_plbn.addArc("ethnicity","g1surban")
star_plbn.addArc("ethnicity","g1freelunch")
star_plbn.addArc("g1surban","g1freelunch")
star_plbn.addArc("g1surban","Y")
star_plbn.addArc("g1freelunch","Y")
```

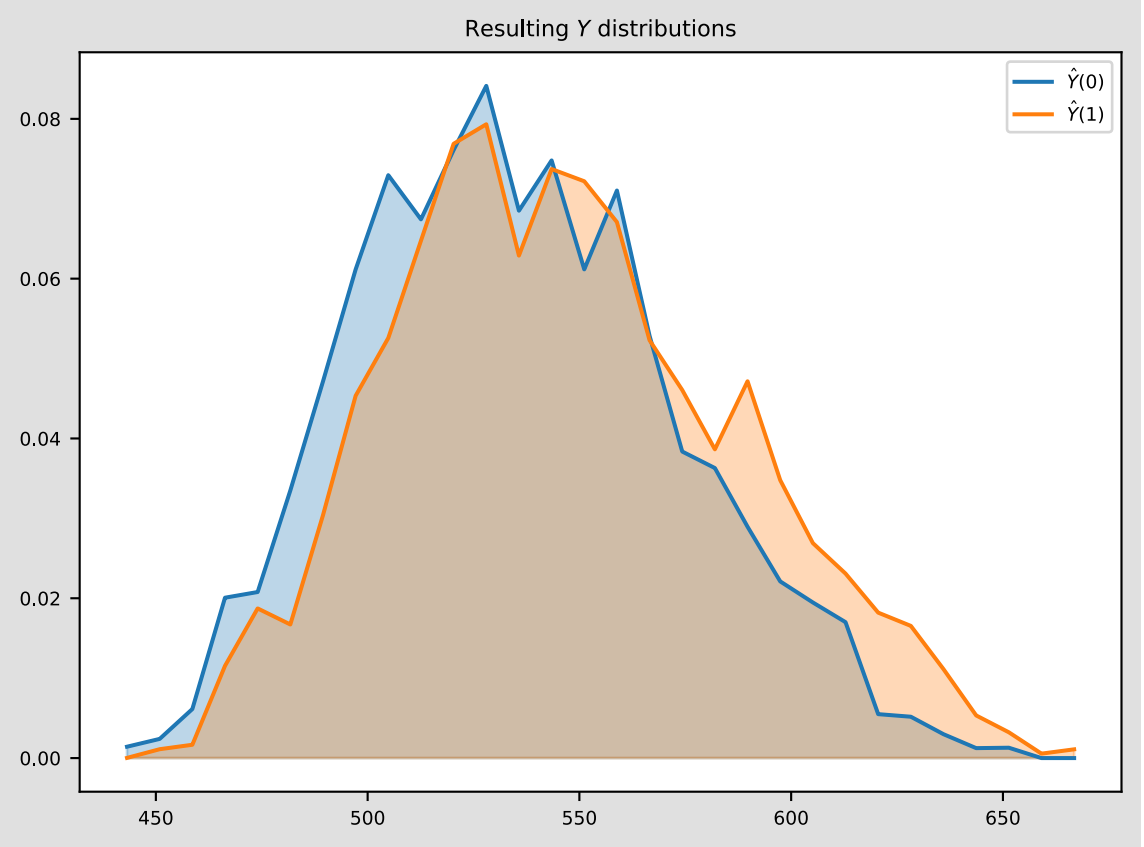
```
learner.fitParameters(star_plbn)

print(learner)
```

```
Filename       : /tmp/tmpruxk_1mx.csv
Size           : (4215,7)
Variables      : Y[30], T[2], gender[2], ethnicity[6], age[24], g1freelunc
h[3], g1surban[4]
Induced types  : False
Missing values : False
Algorithm      : MIIC
Score          : BDeu  (Not used for constraint-based algorithms)
Correction     : NML  (Not used for score-based algorithms)
Prior          : Smoothing
Prior weight   : 0.000001
```

In [ ]:
```
gnb.sideBySide(gexpl.getInformation(star_plbn, size="50"), gnb.getInferen
```



In [ ]:
```
Y_hat = getY(star_plbn)

x0, y0 = (Y_hat[0]["interval_mean"].to_numpy(), \
          Y_hat[0]["probability"].to_numpy())
x1, y1 = (Y_hat[1]["interval_mean"].to_numpy(), \
          Y_hat[1]["probability"].to_numpy())

plt.figure(figsize=(7, 5))

plt.fill_between(x0, y0, alpha=0.3, color='tab:blue')
plt.plot(x0, y0, color="tab:blue", label="$\hat{Y}(0)$")

plt.fill_between(x1, y1, alpha=0.3, color='tab:orange')
plt.plot(x1, y1, color="tab:orange", label="$\hat{Y}(1)$")
plt.legend()
plt.title("Resulting $Y$ distributions")
plt.show()

print(f"Estimated ATE : {getTau(Y_hat)}")
```




```
Estimated ATE : 11.337228734153769
```

We observe highly similar results between the parameter learning method and the structure learning method. By contrast, direct estimation methods such as the Difference in Means (DM) estimator and the Ordinary Least Squares (OLS) estimator yield average treatment effects of 12.81 and 10.77, respectively. These values are largely consistent with our findings.