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 [1]:
```
import pyAgrum as gum
import pyAgrum.skbn as skbn
import pyAgrum.lib.notebook as gnb
import pyAgrum.lib.explain as gexpl

from scipy.stats import norm, gaussian_kde
from scipy.integrate import quad

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
#plt.style.use("whitegrid")
plt.rcParams.update({'font.size': 6, 'font.family': 'DejaVu Sans'})
```

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 [2]:
```
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 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**2-2*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
```

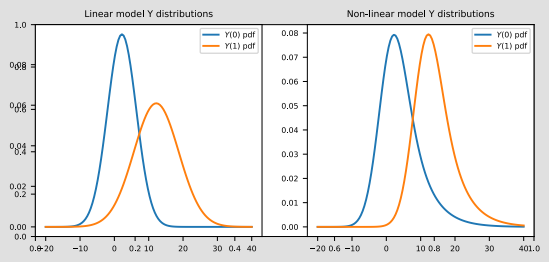
Furthermore, the expected values of $Y(0)$ and $Y(1)$ can be explicitly calculated, providing us the theoretical ATE which enables performance evaluations of the estimators.

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

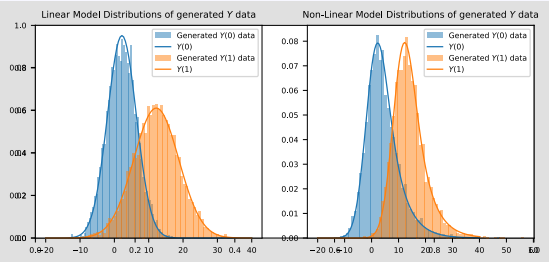
In [3]:
```
# Computations of the theoretical distributions of Y0 and Y1 given by
# the equations of Y

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):
    if x <= 0:
        return 0
    else:
        return (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 ~ 1m30
```

In [4]:
```
# Plotting the distributions

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 [5]:
```
# 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, color="tab:blue", label="Generated $Y(0)$ data")
plt.plot(lin_pdf_df["y0"], color="tab:blue", label="$Y(0)$", linewidth=1)
plt.hist(lin_df[lin_df["T"] == 1]["Y"], bins=60, density=True,
         alpha=0.5, color="tab:orange", label="Generated $Y(1)$ data")
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, color="tab:blue", label="Generated $Y(0)$ data")
plt.plot(nl_pdf_df["y0"], color="tab:blue", label="$Y(0)$", linewidth=1)
plt.hist(nl_df[nl_df["T"] == 1]["Y"], bins=60, density=True,
         alpha=0.5, color="tab:orange", label="Generated $Y(1)$ data")
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 [6]:
```
# 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 [7]:
```python
# 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 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=r"$\hat{Y}(0)$", s=10)
    plt.scatter(x=Y_hat[1]["interval_mean"] ,y=Y_hat[1]["probability"],
                color="tab:orange", label=r"$\hat{Y}(1)$", 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()
```

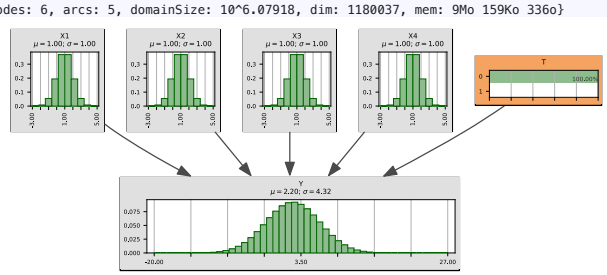
In [8]:
```python
def getBN(# Covariate parameters
          covariate_start : int = -3.0,
          covariate_end : int = 5.0 ,
          covariate_num_split : int = 10,
          covariate_distribution = None,
          # Outcome parameters
          outcome_start = -20.0 ,
          outcome_end = 40.0 ,
          outcome_num_split = 60,
          outcome_loc_expr : str | None = None,
          # Other
          data : pd.DataFrame | None = None,
          add_arcs : bool = True) -> gum.BayesNet:
    """
    Returns Baysian Network corresponding to the model by discretising
    countinous variables with given parameters.
    """
    if data is None:
        bn = gum.BayesNet()
        for i in range(1,5):
            bn.add(f"X{i}[{covariate_start}:{covariate_end}:{covariate_num_split}]")
        bn.add("T[2]")
        bn.add(f"Y[{outcome_start}:{outcome_end}:{outcome_num_split}]")

    else :
        disc = skbn.BNDiscretizer(defaultDiscretizationMethod="uniform",
                                  defaultNumberOfBins=covariate_num_split)
        disc.setDiscretizationParameters("T", 'NoDiscretization', [0, 1])
        disc.setDiscretizationParameters("Y", 'uniform', outcome_num_split)
        bn = disc.discretizedBN(data)

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

```
        bn.endTopologyTransformation()

    if covariate_distribution is not None :
        bn.cpt("T").fillWith([0.5, 0.5])
        for i in range(1,5):
            bn.cpt(f"X{i}").fillFromDistribution(covariate_distribution)

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

    return bn
```

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.

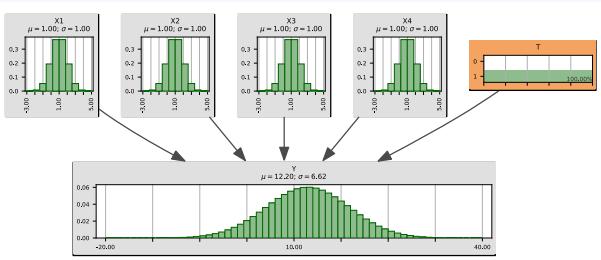
In [9]:
```
lin_exbn = getBN(covariate_distribution=norm(loc=1, scale=1),
                 outcome_loc_expr=lin_expr)
nl_exbn  = getBN(covariate_distribution=norm(loc=1, scale=1),
                 outcome_loc_expr=nl_expr)
print(lin_exbn)

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)"])
```

```
BN{nodes: 6, arcs: 5, domainSize: 10^6.07918, dim: 1180037, mem: 9Mo 159Ko 336o}
```



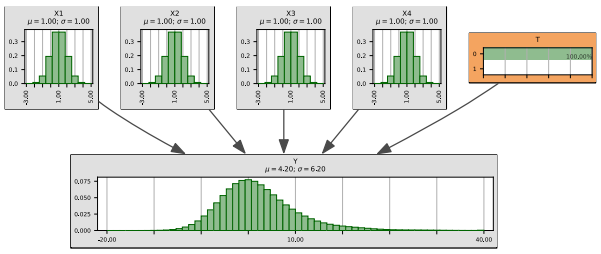
Inference in 6.13ms

Linear Y(0)



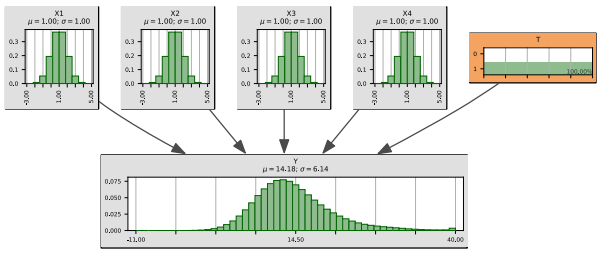
Inference in 4.76ms

Linear Y(1)



Inference in 5.10ms

Non-Linear Y(0)



Inference in 6.09ms

Non-Linear Y(1)

From now on, obtaining the average treatment effect is straightforward using pyAgrum's LazyPropagation exact inference. This involves setting the evidence of the treatment T to 0 and then to 1 to compute the respective posterior CPTs. The ATE is subsequently obtained by performing manipulations on the difference between the CPTs.

In [10]:
```
def ATE(bn : gum.BayesNet) -> float:
    """
    Returns estimation of the ATE directly from Baysian Network.
    """
    ie = gum.LazyPropagation(bn)

    ie.setEvidence({"T": 0})
    ie.makeInference()
    p0 = ie.posterior("Y")

    ie.chgEvidence("T",1)
    ie.makeInference()
    p1 = ie.posterior("Y")

    dif = p1 - p0
    return dif.expectedValue(lambda d: dif.variable(0).numerical(d[dif.variable(0).name()]))
```

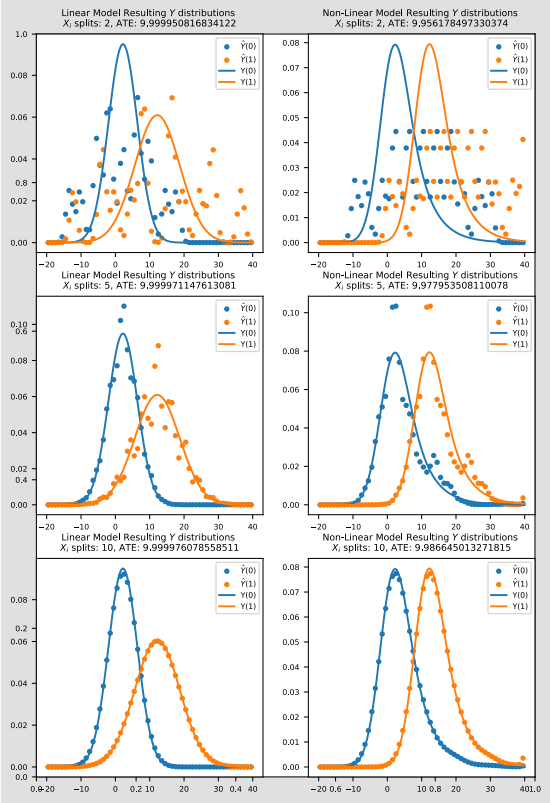
In [11]:
```
print(lin_exbn)
print(f"{ATE(lin_exbn) = }")
print(f"\n{nl_exbn}")
print(f"{ATE(nl_exbn) = }")
```

```
BN{nodes: 6, arcs: 5, domainSize: 10^6.07918, dim: 1180037, mem: 9Mo 159Ko 336o}
ATE(lin_exbn) = 9.999976078558515

BN{nodes: 6, arcs: 5, domainSize: 10^6.07918, dim: 1180037, mem: 9Mo 159Ko 336o}
ATE(nl_exbn) = 9.986645013271817
```

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

In [12]:
```
num_split_list = [2, 5, 10]

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_ex_bn = getBN(covariate_num_split=num_split_list[i],
                      covariate_distribution=norm(loc=1, scale=1),
                      outcome_loc_expr=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)}")
```

```
    # Non Linear Model

    plt.subplot(len(num_split_list), 2, 2*i+2)
    nl_ex_bn = getBN(covariate_num_split=num_split_list[i],
                     covariate_distribution=norm(loc=1, scale=1),
                     outcome_loc_expr=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()
```



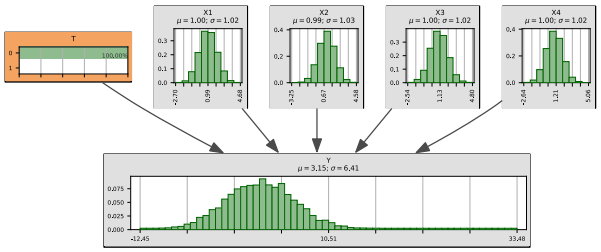
finer discretization with a greater number of bins results in improved approximations of the probability density functions. However, despite the use of rough discretization, the estimations of the ATE remain remarkably accurate.

1.2 - Parameter Learning

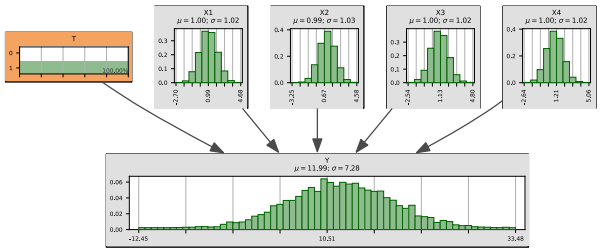
Given the data generating function defined above, parameter learning methods can be employed to infer the underlying distribution based on the given structure of the Bayesian network. The default Mutual Information based Inference of Causal Networks (MIIC) algorithm utilized in the `BNLearner` class effectively performs this task.

In [13]:
```
# Linear Model

lin_template = getBN(data=lin_df)

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(data=nl_df)

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)

print(lin_p_learner)
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)"])

print(nl_p_learner)
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)"])
```

```
Filename       : /var/folders/r1/pj4vdx_n4_d_xpsb04kzf97r0000gp/T/tmpefis6l6e.csv
Size           : (10000,6)
Variables      : T[2], X1[10], X2[10], X3[10], X4[10], Y[60]
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
```

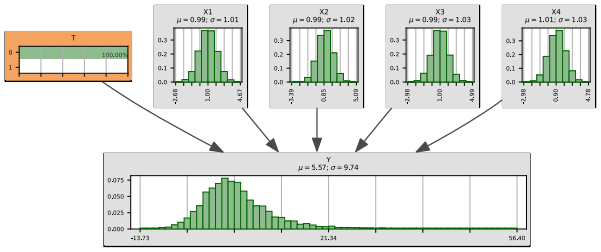


Linear Y(0)

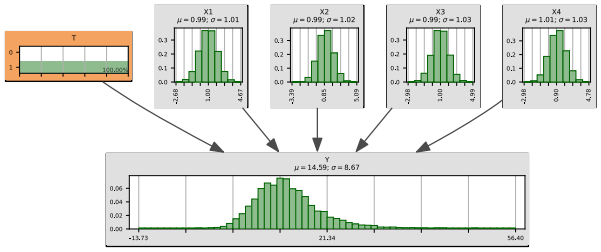


Linear Y(1)

```
Filename       : /var/folders/r1/pj4vdx_n4_d_xpsb04kzf97r0000gp/T/tmp8bxhctbx.csv
Size           : (10000,6)
Variables      : T[2], X1[10], X2[10], X3[10], X4[10], Y[60]
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
```



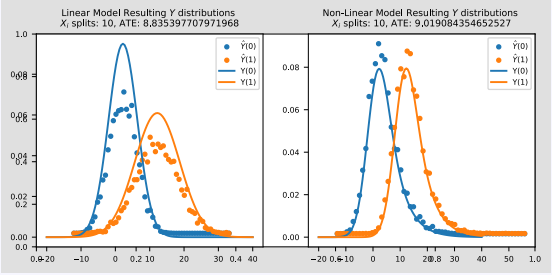
Non-Linear Y(0)



Non-Linear Y(1)

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

In [14]:
```
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)}")
```

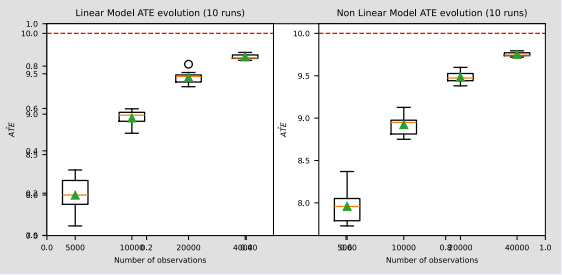


This underestimation can be further observed with varying numbers of observations in both models.

In [15]:
```
lin_tau_hat_arr = list()
nl_tau_hat_arr = list()

num_obs_list = [5000, 10000, 20000, 40000]
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(data=lin_df)

        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(data=nl_df)

        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 [16]:
```
plt.subplots(figsize=(7, 3))

plt.subplot(1, 2, 1)

plt.boxplot(lin_tau_hat_arr, tick_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(r"$\hat{ATE}$")

plt.subplot(1, 2, 2)

plt.boxplot(nl_tau_hat_arr, tick_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(r"$\hat{ATE}$")

plt.show()
```



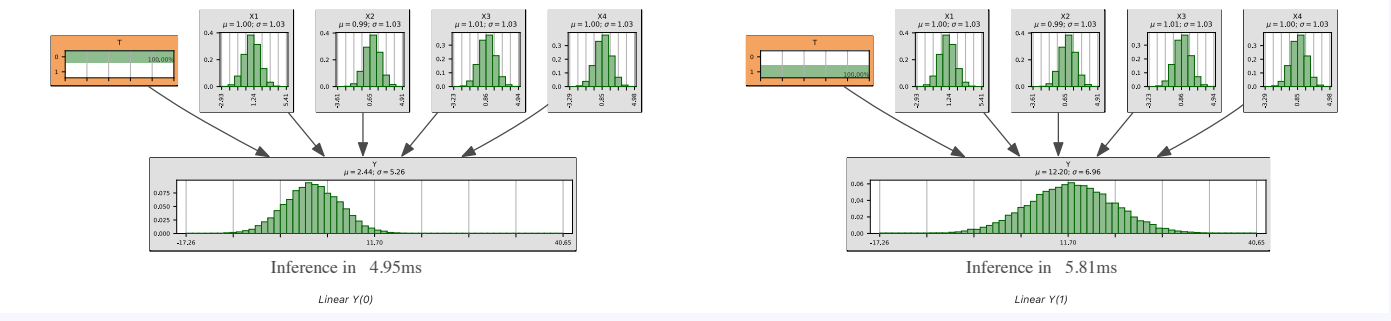
With an increasing number of observations, we observe a convergence of the estimation towards the true value of the ATE and a corresponding reduction in variance.

1.3 - Structure Learning

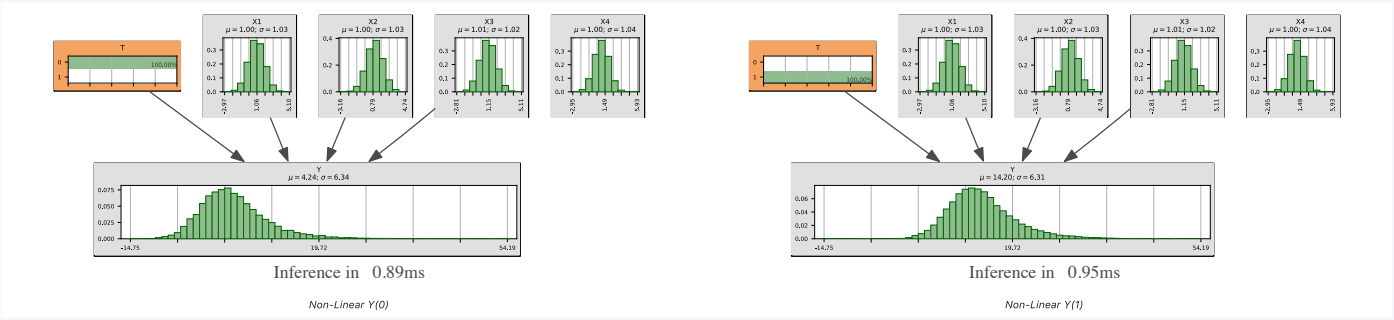
The network's structure and the distributions of the variables can be derived from a sufficiently large dataset through non-parametric learning methods. However, to ensure the integrity of the process, we will impose a slice order on the learner. This ensures that no node is an ancestor of the treatment variable, and no node is a descendant of the outcome variable.

In [17]:
```
lin_template = getBN(data=lin_df)
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(data=nl_df)
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()

print(lin_s_learner)
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)"])

print(nl_s_learner)
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)"])
```

```
Filename               : /var/folders/r1/pj4vdx_n4_d_xpsb04kzf97r0000gp/T/tmpuw7ro_nf.csv
Size                   : (40000,6)
Variables              : T[2], X1[10], X2[10], X3[10], X4[10], Y[60]
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 : {X3:1, X1:1, X4:1, T:0, X2:1, Y:2}
```

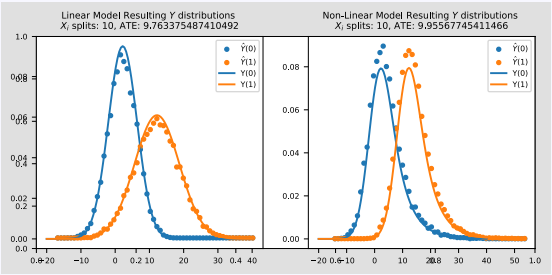


```
Filename               : /var/folders/r1/pj4vdx_n4_d_xpsb04kzf97r0000gp/T/tmp2djddqwq.csv
Size                   : (40000,6)
Variables              : T[2], X1[10], X2[10], X3[10], X4[10], Y[60]
Induced types          : False
Missing values         : False
Algorithm              : MIIC
Score                  : BDeu  (Not used for constraint-based algorithms)
Correction             : NML  (Not used for score-based algorithms)
Prior                  : Smoothing
Prior weight           : 0.000001
Constraint Slice Order : {X3:1, X1:1, X4:1, T:0, X2:1, Y:2}
```



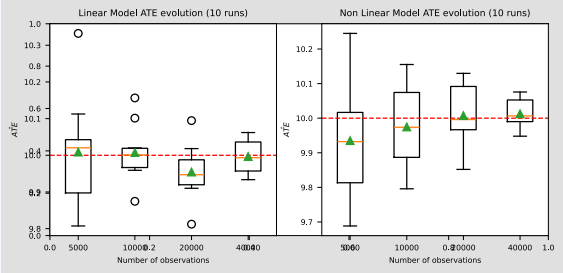
With 10,000 observations, structure learning yields a more accurate estimation of the Average Treatment Effect (ATE) compared to parameter learning. This improvement can be attributed to the use of a less complex structure, as opposed to the structure used previously, which features a higher in-degree on the outcome node. The increased number of parameters to be estimated in the outcome model due to this higher in-degree is suboptimal for smaller datasets.

In [18]:
```
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 [19]:
```
lin_tau_hat_arr = []
nl_tau_hat_arr = []

num_obs_list = [5000, 10000, 20000, 40000]
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 [20]:
```
plt.subplots(figsize=(7, 3))

plt.subplot(1, 2, 1)

plt.boxplot(lin_tau_hat_arr, tick_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(r"$\hat{ATE}$")

plt.subplot(1, 2, 2)

plt.boxplot(nl_tau_hat_arr, tick_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(r"$\hat{ATE}$")

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 [21]:
```
# 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[21]:

	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

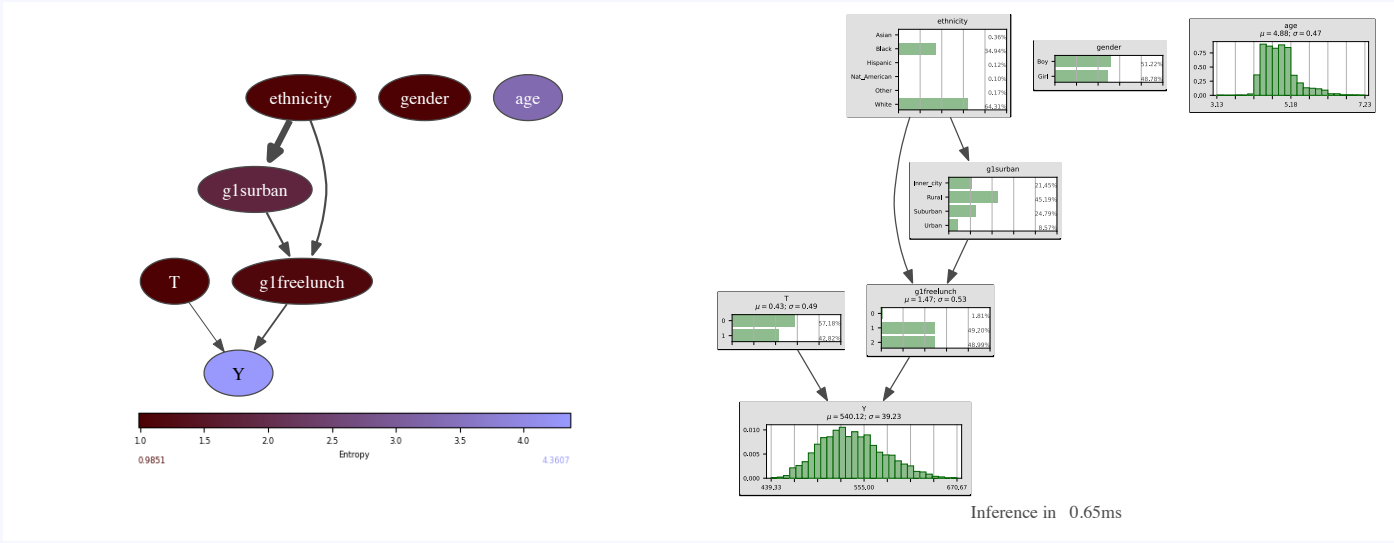
In [22]:
```
def toHist(X : list[float], Y : list[float],
           n : int = 1000) -> list[float]:
    """
    Transforms X, Y data from plotting to a
    histogram plotting format
    """
    res = list()
    for i in range(len(X)):
        res += [X[i]]*int(n*Y[i])
    return res
```

2.1 - Structure Learning

n the absence of prior knowledge regarding the underlying distributions of the variables and their relationships, causal inference can be challenging. Consequently, we will first utilize structure learning to automatically identify the network's underlying structure. To assist the learning process, we will impose a slice order on the variables once again.

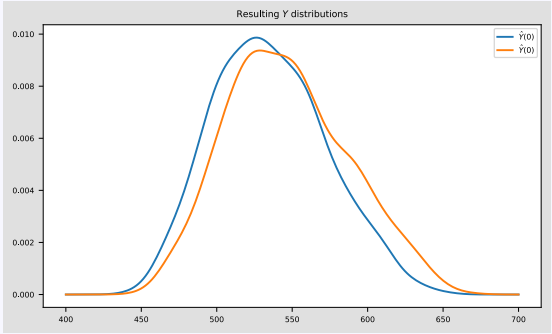
In [23]:
```
disc = skbn.BNDiscretizer(defaultDiscretizationMethod='uniform')
disc.setDiscretizationParameters("age", 'uniform', 24)
disc.setDiscretizationParameters("Y", 'uniform', 30)

template = disc.discretizedBN(star_df)

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

print(learner)

gnb.sideBySide(gexpl.getInformation(star_slbn, size="50"),
               gnb.getInference(star_slbn, size="50"))
```

```
Filename               : /var/folders/r1/pj4vdx_n4_d_xpsb04kzf97r0000gp/T/tmpm23i3zqq.csv
Size                   : (4215,7)
Variables              : Y[30], T[2], gender[2], ethnicity[6], age[24], g1freelunch[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:0, T:0, g1surban:1, age:0, gender:0, g1freelunch:1, Y:2}
```



This initial approach appears promising, as the inferred causal relationships are somewhat consistent with what might be expected from an non-expert perspective.

In [24]:
```
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())

density0 = gaussian_kde(toHist(x0, y0))
density1 = gaussian_kde(toHist(x1, y1))
xs = np.linspace(400,700,200)

plt.figure(figsize=(7, 4))

plt.plot(xs, density0(xs), color="tab:blue", label=r"$\hat{Y}(0)$")

plt.plot(xs, density1(xs), color="tab:orange", label=r"$\hat{Y}(0)$")


plt.legend()
plt.title("Resulting $Y$ distributions")
plt.show()

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




```
Estimated ATE : 11.51375626395145
```

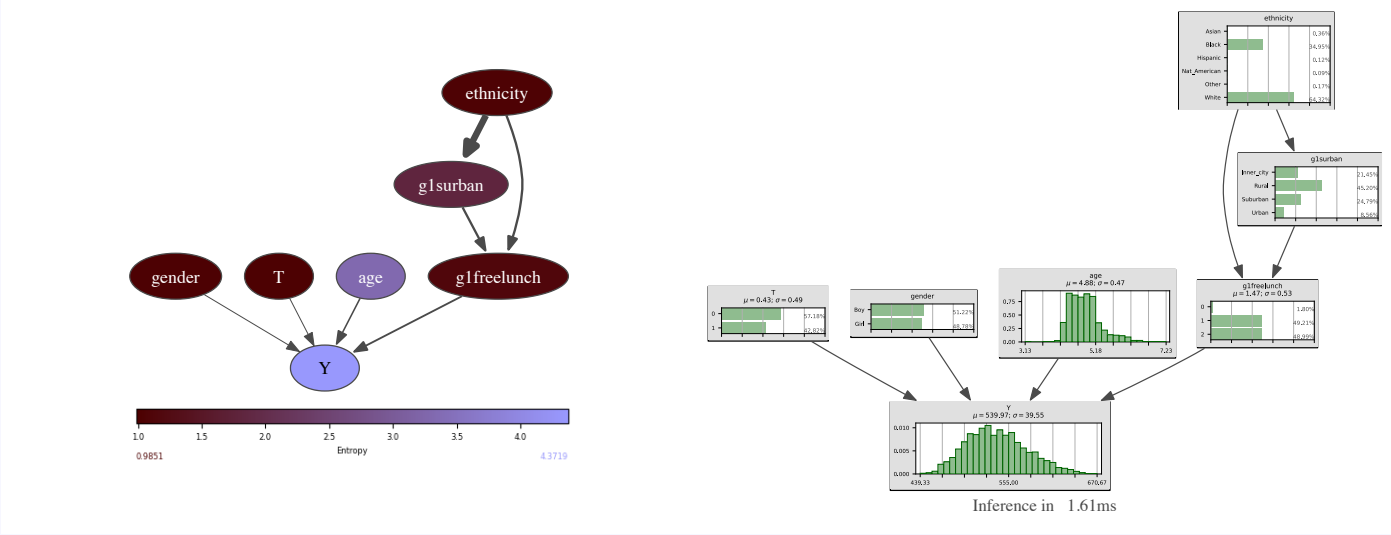
We observe a slight change in the outcome distribution. However, since the outcome takes values in the hundreds, this results in a non-negligeable impact on the treatment effect, given that the outcome is defined as the average of the students' three grades.

2.2 - Parameter Learning

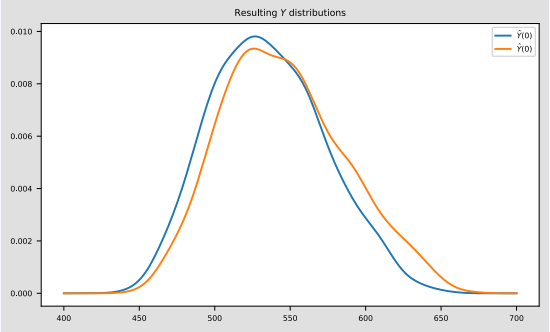
Using different structures when conducting parameter learning can yield varying results. For the sake of illustration, we will examine how the estimation performs when arcs from the `age` and `gender` covariates are added to the outcome.

In [25]:
```
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("g1freelunch","Y")
star_plbn.addArc("gender","Y")
star_plbn.addArc("age","Y")

learner.fitParameters(star_plbn)

print(learner)

gnb.sideBySide(gexpl.getInformation(star_plbn, size="50"),
               gnb.getInference(star_plbn, size="50"))
```

```
Filename       : /var/folders/r1/pj4vdx_n4_d_xpsb04kzf97r0000gp/T/tmp3a_80_au.csv
Size           : (4215,7)
Variables      : Y[30], T[2], gender[2], ethnicity[6], age[24], g1freelunch[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 [26]:
```
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())

density0 = gaussian_kde(toHist(x0, y0))
density1 = gaussian_kde(toHist(x1, y1))
xs = np.linspace(400,700,200)

plt.figure(figsize=(7, 4))

plt.plot(xs, density0(xs), color="tab:blue", label=r"$\hat{Y}(0)$")

plt.plot(xs, density1(xs), color="tab:orange", label=r"$\hat{Y}(0)$")


plt.legend()
plt.title("Resulting $Y$ distributions")
plt.show()

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




```
Estimated ATE : 10.344241933416356
```

As anticipated, there are observable differences between the parameter learning method and the structure learning method. When compared to direct estimation methods, such as the Difference in Means (DM) estimator and the Ordinary Least Squares (OLS) estimator, which yield average treatment effects of 12.81 and 10.77, respectively, our findings remain largely consistent. These results suggest that incorporating age and gender variables into the outcome model may deteriorate the final estimation accuracy.

In [ ]: