Overview

The R package R6causal implements an R6 class called SCM. The class aims to simplify working with structural causal models. The missing data mechanism can be defined as a part of the structural model.

The class contains methods for

- defining a structural causal model via functions, text or conditional probability tables
- printing basic information on the model
- plotting the graph for the model using packages igraph or qgraph
- simulating data from the model
- applying an intervention
- checking the identifiability of a query using the R packages causaleffect and dosearch
- defining the missing data mechanism
- simulating incomplete data from the model according to the specified missing data mechanism
- checking the identifiability in a missing data problem using the R package dosearch
- checking the identifiability of a counterfactual query using the R package cfid

In addition, there are functions for

- running experiments
- counterfactual inference using simulation
- evaluating fairness of a prediction model

The class ParallelWorld inherits SCM and defines a structural causal model that describes parallel worlds for counterfactual inference.

The class LinearGaussianSCM inherits SCM and defines a structural causal model where all functions are linear and all background variables follow Gaussian distribution.

Setup

```r
library(R6causal)
library(data.table)
library(stats)
data.table::.setDTthreads(2)
```

Defining the model

Structural causal model (SCM) for a backdoor situation can be defined as follows

```r
backdoor <- SCM$new("backdoor",
  uflist = list(}
```
uz = function(n) {return(runif(n))},
ux = function(n) {return(runif(n))},
uy = function(n) {return(runif(n))}
),
vflist = list(
  z = function(uz) {
    return(as.numeric(uz < 0.4)),
  x = function(ux, z) {
    return(as.numeric(ux < 0.2 + 0.5*z)),
  y = function(uy, z, x) {
    return(as.numeric(uy < 0.1 + 0.4*z + 0.4*x))
  }
)
)

A shortcut notation for this is

```r
backdoor_text <- SCM$new("backdoor",
  uflist = list(
    uz = "n : runif(n)",
    ux = "n : runif(n)",
    uy = "n : runif(n)"
  ),
  vflist = list(
    z = "uz : as.numeric(uz < 0.4)",
    x = "ux, z : as.numeric(ux < 0.2 + 0.5*z)",
    y = "uy, z, x : as.numeric(uy < 0.1 + 0.4*z + 0.4*x)"
  )
)
```

Alternatively the functions of SCM can be specified via conditional probability tables

```r
backdoor_condprob <- SCM$new("backdoor",
  uflist = list(
    uz = function(n) {return(runif(n))},
    ux = function(n) {return(runif(n))},
    uy = function(n) {return(runif(n))}
  ),
  vflist = list(
    z = function(uz) {
      return( generate_condprob( ycondx = data.table(z = c(0,1),
        prob = c(0.6,0.4)),
        x = data.table(uz = uz),
        Umerge_expr = "uz")),
    x = function(ux, z) {
      return( generate_condprob( ycondx = data.table(x = c(0,1,0,1),
        z = c(0,0,1,1),
        prob = c(0.8,0.2,0.3,0.7)),
        x = data.table(z = z, ux = ux),
        Umerge_expr = "ux")),
    y = function(uy, z, x) {
      return( generate_condprob( ycondx = data.table(y= rep(c(0,1), 4),
        z = c(0,0,1,1,0,0,1,1),
        x = c(0,0,0,0,1,1,1,1),
        prob = c(0.9,0.1,0.5,0.5, 0.5,0.5,0.1,0.9)),
      )
    }
  )
```
x = data.table(z = z, x = x, uy = uy),
Umerge_expr = "uy"
)
)}

It is possible to mix the styles and define some elements of a function list as functions, some as text and some as conditional probability tables.

**Defining a linear Gaussian SCM**

A linear Gaussian SCM can be defined giving the coefficients for the structural equations:

```r
lgbackdoor <- LinearGaussianSCM$new("Linear Gaussian Backdoor",
linear_gaussian = list(
  uflist = list(ux = function(n) {rnorm(n)},
                uy = function(n) {rnorm(n)},
                uz = function(n) {rnorm(n)}),
  vnames = c("x","y","z"),
  vcoefmatrix = matrix(c(0,0.4,0,0,0,0.6,0.8,0),3,3),
  ucoefvector = c(1,1,1),
  ccoeffvector = c(0,0,0)))

print(lgbackdoor)
#> Name of the model: Linear Gaussian Backdoor
#>
#> Graph:
#> z -> x
#> x -> y
#> z -> y
#>
#> Functions of background (exogenous) variables:
#>
#> $ux
#> function(n) {rnorm(n)}
#>
#> $uy
#> function(n) {rnorm(n)}
#>
#> $uz
#> function(n) {rnorm(n)}
#>
#> Functions of endogenous variables:
#>
#> $x
#> function (z, ux)
#> { return(0 + 0.6 * z + 1 * ux)
#> }
#> <environment: 0x000001d0766a0928>
#>
#> $y
#> function (x, z, uy)
#> { return(0 + 0.4 * x + 0.8 * z + 1 * uy)
#> }
It is also possible to generate the underlying DAG and the coefficients randomly:

```r
randomlg <- LinearGaussianSCM$new(
  "Random Linear Gaussian",
  random_linear_gaussian = list(
    nv = 6,
    edgeprob = 0.5,
    vcoefdistr = function(n) { rnorm(n) },
    ccoefdistr = function(n) { rnorm(n) },
    ucoefdistr = function(n) { rnorm(n) }
  ))
print(randomlg)
```

```
#> Name of the model: Random Linear Gaussian
#>
#> Graph:
#> v3 -> v1
#> v4 -> v2
#> v3 -> v4
#> v2 -> v5
#> v1 -> v6
#> v4 -> v6
#>
#> Functions of background (exogenous) variables:
#>
#> $u1
#> function (n)
#> {
#>   return(rnorm(n))
#> }
#> <environment: 0x000001d076d789a0>
#>
#> $u2
#> function (n)
#> {
#>   return(rnorm(n))
#> }
#> <environment: 0x000001d076d83f60>
#>
#> $u3
#> function (n)
#> {
#>   return(rnorm(n))
#> }
#> <environment: 0x000001d076693898>
```
Functions of endogenous variables:

## $v1

```r
function (u1)
{
  return(-0.194452020142582 + 0.340822645556077 * u1)
}
```

## $v2

```r
function (u2)
{
  return(1.14436900375859 + 0.348478944040261 * u2)
}
```

## $v3

```r
function (u3)
{
  return(0.549220880109555 + -1.03735874103047 * u3)
}
```

## $v4

```r
function (v3, u4)
{
  return(-1.11379680798657 + -2.53563294243635 * v3 + -0.61458842511206 *
$v5$

```r
function (v2, u5)
{
  return(-0.930116389868676 + 0.626790759039911 * v2 + 0.0258670170501744 * u5)
}
```

$u4$

```r
}
```

$u5$

```r
}
```

$u6$

```r
}
```

```
Topological order of endogenous variables:
[1]  "v3"  "v1"  "v4"  "v2"  "v6"  "v5"
```

No missing data mechanism
Printing the model

The print method presents the basic information on the model

```
backdoor
#> Name of the model: backdoor
#>
#> Graph:
#> z -> x
#> z -> y
#> x -> y
#>
#> Functions of background (exogenous) variables:
#>
#> $uz
#> function(n) {return(runif(n))}
#>
#> $ux
#> function(n) {return(runif(n))}
#>
#> $uy
#> function(n) {return(runif(n))}
#>
#> Functions of endogenous variables:
#>
#> $z
#> function(uz) {
#>     return(as.numeric(uz < 0.4))
#> }
#>
#> $x
#> function(ux, z) {
#>     return(as.numeric(ux < 0.2 + 0.5*z))
#> }
#>
#> $y
#> function(uy, z, x) {
#>     return(as.numeric(uy < 0.1 + 0.4*z + 0.4*x))
#> }
#>
#> Topological order of endogenous variables:
#> [1] "z" "x" "y"
#>
#> No missing data mechanism
```
Plotting the graph

The plotting method of the package \texttt{igraph} is used by default. If \texttt{qgraph} is available, its plotting method can be used as well. The argument \texttt{subset} controls which variables are plotted. Plotting parameters are passed to the plotting method.

\begin{verbatim}
backdoor$plot(vertex.size = 25)  # with package 'igraph'
\end{verbatim}

\begin{verbatim}
backdoor$plot(subset = "v")  # only observed variables
\end{verbatim}

\begin{verbatim}
if (requireNamespace("qgraph", quietly = TRUE)) backdoor$plot(method = "qgraph")
\end{verbatim}
Simulating data

Calling method `simulate()` creates or updates data table `simdata`.

```r
backdoor$simulate(10)
```
```
# alternative look with package 'qgraph'
```
```r
backdoor$simdata
```
```
#>    uz    ux    uy   z   x   y
#> 1: 0.93706919 0.65222882 0.04931049 0 0 1
#> 2: 0.62029457 0.16501685 0.85497557 0 1 0
#> 3: 0.99339531 0.34660394 0.83191445 0 0 0
#> 4: 0.15499748 0.09027895 0.80703810 1 1 1
#> 5: 0.01558495 0.99805482 0.58878679 1 0 0
#> 6: 0.23510216 0.61778566 0.17032293 1 1 1
#> 7: 0.51754335 0.79296122 0.16970637 0 0 0
#> 8: 0.01453718 0.4206458 0.53460183 1 1 1
#> 9: 0.32152004 0.54656103 0.43992911 1 1 1
#>10: 0.53686887 0.24877851 0.74216326 0 0 0
```
```r
backdoor$simulate(8)
```
```
backdoor$simdata
```
```
#>    uz    ux    uy   z   x   y
#> 1: 0.2562507 0.9516747 0.6994977 1 0 0
#> 2: 0.1340398 0.3607472 0.9519103 1 1 1
#> 3: 0.3436197 0.3637142 0.5748896 1 1 1
#> 4: 0.3953156 0.4762140 0.2458309 1 1 1
#> 5: 0.7310678 0.7387591 0.7152488 0 0 0
```
Applying an intervention

In an intervention, the structural equation of the target variable is changed.

```r
backdoor_x1 <- backdoor$clone()  # making a copy
backdoor_x1$intervene("x", 1)  # applying the intervention
backdoor_x1$plot(method = "qgraph")  # to see that arrows incoming to x are cut
```

```r
backdoor_x1$simulate(10)  # simulating from the intervened model
```

```r
backdoor_x1 simuldata
```

```r
  #> 6: 0.8273890 0.1857632 0.4729871 0 1 1
  #> 7: 0.9219097 0.6306178 0.9444048 0 0 0
  #> 8: 0.3736709 0.9560987 0.5207959 1 0 0
```
An intervention can redefine a structural equation

backdoor_yz <- backdoor$clone()  # making a copy
backdoor_yz$intervene("y",  
  function(uy, z) {return(as.numeric(uy < 0.1 + 0.8*z ))})  # making y a function of z only
backdoor_yz$plot(method = "qgraph")  # to see that arrow x -> y is cut

Running an experiment (set of interventions)

The function run_experiment applies a set of interventions, simulates data and collects the results.

backdoor_experiment <- run_experiment(backdoor,  
  intervene = list(x = c(0,1)),  
  response = "y",  
  n = 10000)

str(backdoor_experiment)
#> List of 2
#> $ interventions:Classes 'data.table' and 'data.frame': 2 obs. of 1 variable:
#> ..$ x: num [1:2] 0 1
#> $ response_list:List of 1
#> ..$ y:Classes 'data.table' and 'data.frame': 10000 obs. of 2 variables:
#> ...$ V1: num [1:10000] 0 0 0 0 0 0 0 0 1 1 ...
#> ...$ V2: num [1:10000] 1 0 0 1 0 1 1 1 1 1 ...
#> ...$ attr(".internal.selfref")=externalptr

colMeans(backdoor_experiment$response_list$y)
#> V1   V2
#> 0.2614 0.6652
Applying the ID algorithm, Do-search and cfid

There are direct plugins to R packages `causaleffect`, `dosearch` and `cfid` that can be used to solve identifiability problems.

```r
backdoor$causal.effect(y = "y", x = "x")
#> [1] "\sum_{z}P(y/z,x)P(z)"
backdoor$dosearch(data = "p(x,y,z)", query = "p(y|do(x))")
#> \sum_{z}\left(p(z)p(y|x,z)\right)
backdoor$cfid(gamma = cfid::conj(cfid::cf("Y",0), cfid::cf("X",0, c(Z=1))))
#> The query P(y /\ x_{z'}) is not identifiable from P_*.
```

Counterfactual inference (a simple case)

Let us assume that intervention do(X=0) was applied and the response Y = 0 was recorded. What is the probability that in this situation the intervention do(X=1) would have led to the response Y = 1? We estimate this probability by means of simulation.

```r
cfdata <- counterfactual(backdoor, situation = list(do = list(target = "x", ifunction = 0), condition = data.table( x = 0, y = 0)), target = "x", ifunction = 1, n = 100000, method = "rejection")
mean(cfdata$y)
#> [1] 0.53843
```

The result differs from \(P(Y = 1 | do(X = 1))\)

```r
backdoor_x1$simulate(100000)
mean(backdoor_x1$simdata$y)
#> [1] 0.66093
```

Counterfactual inference (parallel worlds)

Parallel world graphs (a generalization of a twin graph) are used for counterfactual inference with several counterfactual interventions. The package implements class `ParallelWorld` which heritates class `SCM`. A `ParallelWorld` object is created from an `SCM` object by specifying the interventions for each world. By default the variables of the parallel worlds are named with suffixes "\_1", "\_2", ...

In the example below, we have the original world (variables x, z, y) and its two variants. In the variant 1 (variables x\_1, z\_1, y\_1), the value of x (variable x\_1 in the object) is set to be 0. In the variant 2 (variables x\_2, z\_2, y\_2), the value of x (variable x\_2 in the object) is set to be 0 and the value of z (variable z\_2 in the object) is set to be 1.

```r
backdoor_parallel <- ParallelWorld$new(
  backdoor,
  dolist=list(
    list(target = "x",
         ifunction = 0),
    list(target = list("z","x"),
         ifunction = list(1,0))
  )
)
backdoor_parallel
#> Name of the model: backdoor
```
Graph:
uz -> z
z -> x
uy -> y
z -> y
x -> y
uz -> z_1
uy -> y_1
z_1 -> y_1
x_1 -> y_1
uy -> y_2
z_2 -> y_2
x_2 -> y_2

Functions of background (exogenous) variables:

$uz
function(n) {return(runif(n))}
<bytecode: 0x000001d00769cba8>

$ux
function(n) {return(runif(n))}
<bytecode: 0x000001d00772d340>

$uy
function(n) {return(runif(n))}
<bytecode: 0x000001d0077bdad8>

Functions of endogenous variables:

$z
function(uz) {
  return(as.numeric(uz < 0.4))
}
<bytecode: 0x000001d007879a60>

$x
function(ux, z) {
  return(as.numeric(ux < 0.2 + 0.5*z))
}
<bytecode: 0x000001d0079948a0>

$y
function(uy, z, x) {
  return(as.numeric(uy < 0.1 + 0.4*z + 0.4*x))
}
<bytecode: 0x000001d007b10ee8>

$z_1
function (uz)
{
  return(as.numeric(uz < 0.4))
}

$x_1
function (…)
\[
\begin{align*}
\Rightarrow \{ \text{return}(\text{constant})
\Rightarrow \}\text{ <environment: 0x000001d007f06470> }
\Rightarrow \y_1 \Rightarrow \text{function (uy, z_1, x_1)}
\Rightarrow \{ \text{return}(\text{as.numeric}(uy < 0.1 + 0.4 \cdot z_1 + 0.4 \cdot x_1))
\Rightarrow \}\text{ <environment: 0x000001d007f06470> }
\Rightarrow \y_2 \Rightarrow \text{function (uy, z_2, x_2)}
\Rightarrow \{ \text{return}(\text{as.numeric}(uy < 0.1 + 0.4 \cdot z_2 + 0.4 \cdot x_2))
\Rightarrow \}\text{ <environment: 0x000001d007f06470> }
\Rightarrow \z_2 \Rightarrow \text{function (...)}
\Rightarrow \{ \text{return}(\text{constant})
\Rightarrow \}\text{ <environment: 0x000001d007f06470> }
\Rightarrow \z_1 \Rightarrow \text{function (...)}
\Rightarrow \{ \text{return}(\text{constant})
\Rightarrow \}\text{ <environment: 0x000001d007f06470> }
\Rightarrow \x_2 \Rightarrow \text{function (...)}
\Rightarrow \{ \text{return}(\text{constant})
\Rightarrow \}\text{ <environment: 0x000001d007f06470> }
\Rightarrow \x_1 \Rightarrow \text{Topological order of endogenous variables:}
\Rightarrow [1] "x_1" "z_2" "x_2" "z" "z_1" "y_2" "x" "y_1" "y"
\Rightarrow \text{No missing data mechanism}
\text{if (requireNamespace("qgraph", quietly = TRUE)) backdoor_parallel}$plot(method = "qgraph")
Counterfactual data can be simulated with function `counterfactual`. In the example below, we know that variable \( y \) obtained value 0 in the original world as well as variants 1 and 2. We are interested in the counterfactual distribution of \( y \) if \( x \) had been set to 1.

```r
cfdata <- counterfactual(backdoor_parallel,
    situation = list(
        do = NULL,
        condition = data.table::data.table( y = 0, y_1 = 0, y_2 = 0)),
        target = "x",
        ifunction = 1,
        n = 100000,
        method = "rejection"
)
mean(cfdata$y)
#> [1] 0.12464
```

The printed value is a simulation based estimate for the counterfactual probability \( P(Y = 1) \).

An alternative way for answering the same question defines the case of interest as one of the parallel worlds (here variant 3).

```r
backdoor_parallel2 <- ParallelWorld$new(
    backdoor,
    dolist=list(
        list(target = "x",
             ifunction = 0),
        list(target = list("z","x"),
             ifunction = list(1,0)),
        list(target = "x",
             ifunction = 1)
    )
)
```
The printed value is a simulation based estimate for the counterfactual probability $P(Y = 1)$.

A model with a missing data mechanism

The missing data mechanism is defined in similar manner as the other variables.

```r
backdoor_md <- SCM$new("backdoor_md",
  uflist = list(
    uz = "n : runif(n)",
    ux = "n : runif(n)",
    uy = "n : runif(n)",
    urz = "n : runif(n)",
    urx = "n : runif(n)",
    ury = "n : runif(n)"
  ),
  vflist = list(
    z = "uz : as.numeric(uz < 0.4)",
    x = "ux, z : as.numeric(ux < 0.2 + 0.5*z)",
    y = "uy, z, x : as.numeric(uy < 0.1 + 0.4*z + 0.4*x)"
  ),
  rflist = list(
    z = "urz : as.numeric( urz < 0.9)",
    x = "urx, z : as.numeric( (urx + z)/2 < 0.9)",
    y = "ury, z : as.numeric( (ury + z)/2 < 0.9)"
  ),
  rprefix = "r_"
)
```

Plotting the graph for a model with missing data mechanism

```r
backdoor_md$plot(vertex.size = 25, edge.arrow.size=0.5) # with package 'igraph'
```
Simulating incomplete data

By default both complete data and incomplete data are simulated. The incomplete dataset is named as $simdata.obs.

```
if (!requireNamespace("qgraph", quietly = TRUE)) backdoor_md$plot(method = "qgraph")
# alternative look with package 'qgraph'
```

```
summary(backdoor_md$simdata)
```
```r
table
```
<table>
<thead>
<tr>
<th></th>
<th>uz</th>
<th>ux</th>
<th>uy</th>
<th>urz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min.</td>
<td>0.007739</td>
<td>0.01717</td>
<td>0.04045</td>
<td>0.004756</td>
</tr>
<tr>
<td>1st Qu.</td>
<td>0.229987</td>
<td>0.34379</td>
<td>0.19353</td>
<td>0.222194</td>
</tr>
<tr>
<td>Median</td>
<td>0.470774</td>
<td>0.54481</td>
<td>0.47960</td>
<td>0.603977</td>
</tr>
<tr>
<td>Mean</td>
<td>0.498856</td>
<td>0.54419</td>
<td>0.49486</td>
<td>0.538054</td>
</tr>
<tr>
<td>3rd Qu.</td>
<td>0.788599</td>
<td>0.78421</td>
<td>0.74903</td>
<td>0.808703</td>
</tr>
<tr>
<td>Max.</td>
<td>0.997104</td>
<td>0.99674</td>
<td>0.99756</td>
<td>0.994521</td>
</tr>
</tbody>
</table>
```
By using the argument `fixedvars` one can keep the complete data unchanged and re-simulate the missing data mechanism.

`backdoor_md$simulate(100, fixedvars = c("x","y","z","ux","uy","uz"))`

```
#> Median :0.492273 Median :0.505153 Median :0.00 Median :0.00
#> Mean :0.502744 Mean :0.509611 Mean :0.43 Mean :0.38
#> 3rd Qu.:0.771682 3rd Qu.:0.742523 3rd Qu.:1.00 3rd Qu.:1.00
#> Max. :0.995291 Max. :0.995081 Max. :1.00 Max. :1.00
#>
```

```
#> y z_md x_md y_md
#> Min. :0.00 Min. :0.0000 Min. :0.0000 Min. :0.0000
#> 1st Qu.:0.00 1st Qu.:0.0000 1st Qu.:0.0000 1st Qu.:0.0000
#> Median :0.00 Median :0.0000 Median :0.0000 Median :0.0000
#> Mean :0.44 Mean :0.4157 Mean :0.3511 Mean :0.4066
#> 3rd Qu.:1.00 3rd Qu.:1.0000 3rd Qu.:1.0000 3rd Qu.:1.0000
#> Max. :1.00 Max. :1.0000 Max. :1.0000 Max. :1.0000
#> NA's :11 NA's :6 NA's :9
#> r_z r_x r_y
#> Min. :0.00 Min. :0.00 Min. :0.00
#> 1st Qu.:1.00 1st Qu.:1.00 1st Qu.:1.00
#> Median :1.00 Median :1.00 Median :1.00
#> Mean :0.89 Mean :0.94 Mean :0.91
#> 3rd Qu.:1.00 3rd Qu.:1.00 3rd Qu.:1.00
#> Max. :1.00 Max. :1.00 Max. :1.00
#>
```

```
#> r_z r_x
#> Min. :0.00 Min. :0.00
#> 1st Qu.:1.00 1st Qu.:1.00
#> Median :1.00 Median :1.00
#> Mean :0.94 Mean :0.91
#> 3rd Qu.:1.00 3rd Qu.:1.00
#> Max. :1.00 Max. :1.00

```
Applying Do-search to a missing data problem

\[
\sum_z \left( \frac{p(z, r_z = 1)}{p(r_z = 1)} p(y|z, r_z = 1, x, r_x = 1, r_y = 1) \right)
\]

It is automatically recognized that the problem is a missing data problem when `rflist != NULL`.