lingam Documentation

Release 1.7.0

Author

Jan 13, 2023

Contents:

CHAPTER 1

Installation Guide

To install lingam package, use *pip* as follows:

```
$ pip install lingam
```

You can also install the development version of lingam package from GitHub:

```
$ pip install git+https://github.com/cdt15/lingam.git
```

CHAPTER 2

Tutorial

In this tutorial, we will show you how to run LiNGAM algorithms and see the results. We will also show you how to run the bootstrap method and check the results.

The following packages must be installed in order to run this tutorial. And import if necessary:

- numpy
- pandas
- scikit-learn
- graphviz
- statsmodels

Contents:

2.1 DirectLiNGAM

2.1.1 Model

DirectLiNGAM[1] is a direct method for learning the basic LiNGAM model[2]. It uses an entropy-based measure[3] to evaluate independence between error variables. The basic LiNGAM model makes the following assumptions.

1. Linearity
2. Non-Gaussian continuous error variables (except at most one)
3. Acyclicity

[1] S. Shimizu, T. Inazumi, Y. Sogawa, A. Hyvärinen, Y. Kawahara, T. Washio, P. O. Hoyer and K. Bollen. DirectLiNGAM: A direct method for learning a linear non-Gaussian structural equation model. Journal of Machine Learning Research, 12(Apr): 1225–1248, 2011.

[2] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. J. Kerminen. A linear non-gaussian acyclic model for causal discovery. Journal of Machine Learning Research, 7:2003-2030, 2006.

[3] A. Hyvärinen and S. M. Smith. Pairwise likelihood ratios for estimation of non-Gaussian structural eauation models. Journal of Machine Learning Research 14:111-152, 2013.

4. No hidden common causes

Example applications are found here. For example,[4] uses the basic LiNGAM model to infer causal relations of health indice including LDL, HDL, and γGT.

References

2.1.2 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import make_dot

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
np.random.seed(100)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.1']
```

2.1.3 Test data

We create test data consisting of 6 variables.

```
x3 = np.random.uniform(size=1000)
x0 = 3.0*x3 + np.random.uniform(size=1000)
x2 = 6.0*x3 + np.random.uniform(size=1000)
x1 = 3.0*x0 + 2.0*x2 + np.random.uniform(size=1000)
x5 = 4.0*x0 + np.random.uniform(size=1000)
x4 = 8.0*x0 - 1.0*x2 + np.random.uniform(size=1000)
X = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T ,columns=['x0', 'x1', 'x2', 'x3
↪', 'x4', 'x5'])
X.head()
```

```
m = np.array([[0.0, 0.0, 0.0, 3.0, 0.0, 0.0],
              [3.0, 0.0, 2.0, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 6.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
              [8.0, 0.0,-1.0, 0.0, 0.0, 0.0],
              [4.0, 0.0, 0.0, 0.0, 0.0, 0.0]])

dot = make_dot(m)

# Save pdf
dot.render('dag')
```

(continues on next page)

[4] J. Kotoku, A. Oyama, K. Kitazumi, H. Toki, A. Haga, R. Yamamoto, M. Shinzawa, M. Yamakawa, S. Fukui, K. Yamamoto, T. Moriyama. Causal relations of health indices inferred statistically using the DirectLiNGAM algorithm from big data of Osaka prefecture health checkups. PLoS ONE,15(12): e0243229, 2020.

(continued from previous page)

```
# Save png
dot.format = 'png'
dot.render('dag')

dot
```

2.1.4 Causal Discovery

Then, if we want to run DirectLiNGAM algorithm, we create a *DirectLiNGAM* object and call the *fit()* method:

```
model = lingam.DirectLiNGAM()
model.fit(X)
```

```
<lingam.direct_lingam.DirectLiNGAM at 0x1f6afac2fd0>
```

- If you want to use the ICA-LiNGAM algorithm, replace *DirectLiNGAM* above with *ICALiNGAM*.

Using the *causal_order_* property, we can see the causal ordering as a result of the causal discovery.

```
model.causal_order_
```

```
[3, 0, 2, 1, 4, 5]
```

Also, using the *adjacency_matrix_* property, we can see the adjacency matrix as a result of the causal discovery.

```
model.adjacency_matrix_
```

```
array([[ 0.   ,  0.   ,  0.   ,  2.994,  0.   ,  0.   ],
       [ 2.995,  0.   ,  1.993,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  5.586,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 7.981,  0.   , -0.996,  0.   ,  0.   ,  0.   ],
       [ 3.795,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ]])
```

We can draw a causal graph by utility funciton.

```
make_dot(model.adjacency_matrix_)
```

2.1.5 Independence between error variables

To check if the LiNGAM assumption is broken, we can get p-values of independence between error variables. The value in the i-th row and j-th column of the obtained matrix shows the p-value of the independence of the error variables e_i and e_j.

```
p_values = model.get_error_independence_p_values(X)
print(p_values)
```

```
[[0.    0.925 0.443 0.978 0.834 0.   ]
 [0.925 0.    0.133 0.881 0.317 0.214]
 [0.443 0.133 0.    0.    0.64  0.001]
 [0.978 0.881 0.    0.    0.681 0.   ]
 [0.834 0.317 0.64  0.681 0.    0.742]
 [0.    0.214 0.001 0.    0.742 0.   ]]
```

2.2 Bootstrap

2.2.1 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import print_causal_directions, print_dagc, make_dot

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
np.random.seed(0)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.4']
```

2.2.2 Test data

We create test data consisting of 6 variables.

```
x3 = np.random.uniform(size=1000)
x0 = 3.0*x3 + np.random.uniform(size=1000)
x2 = 6.0*x3 + np.random.uniform(size=1000)
x1 = 3.0*x0 + 2.0*x2 + np.random.uniform(size=1000)
x5 = 4.0*x0 + np.random.uniform(size=1000)
x4 = 8.0*x0 - 1.0*x2 + np.random.uniform(size=1000)
X = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T ,columns=['x0', 'x1', 'x2', 'x3
↪', 'x4', 'x5'])
X.head()
```

```
m = np.array([[0.0, 0.0, 0.0, 3.0, 0.0, 0.0],
              [3.0, 0.0, 2.0, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 6.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
              [8.0, 0.0,-1.0, 0.0, 0.0, 0.0],
              [4.0, 0.0, 0.0, 0.0, 0.0, 0.0]])

make_dot(m)
```

2.2.3 Bootstrapping

We call *bootstrap()* method instead of *fit()*. Here, the second argument specifies the number of bootstrap sampling.

```
model = lingam.DirectLiNGAM()
result = model.bootstrap(X, n_sampling=100)
```

2.2.4 Causal Directions

Since *BootstrapResult* object is returned, we can get the ranking of the causal directions extracted by *get_causal_direction_counts()* method. In the following sample code, `n_directions` option is limited to the causal directions of the top 8 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
cdc = result.get_causal_direction_counts(n_directions=8, min_causal_effect=0.01,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_causal_directions(cdc, 100)
```

```
x1 <--- x0 (b>0) (100.0%)
x1 <--- x2 (b>0) (100.0%)
x5 <--- x0 (b>0) (100.0%)
x0 <--- x3 (b>0) (99.0%)
x4 <--- x0 (b>0) (98.0%)
x2 <--- x3 (b>0) (96.0%)
x4 <--- x2 (b<0) (94.0%)
x4 <--- x5 (b>0) (20.0%)
```

2.2.5 Directed Acyclic Graphs

Also, using the *get_directed_acyclic_graph_counts()* method, we can get the ranking of the DAGs extracted. In the following sample code, `n_dags` option is limited to the dags of the top 3 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
dagc = result.get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.01,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_dagc(dagc, 100)
```

```
DAG[0]: 54.0%
    x0 <--- x3 (b>0)
    x1 <--- x0 (b>0)
    x1 <--- x2 (b>0)
    x2 <--- x3 (b>0)
    x4 <--- x0 (b>0)
    x4 <--- x2 (b<0)
    x5 <--- x0 (b>0)
DAG[1]: 16.0%
```

(continues on next page)

(continued from previous page)

```
    x0 <--- x3 (b>0)
    x1 <--- x0 (b>0)
    x1 <--- x2 (b>0)
    x2 <--- x3 (b>0)
    x4 <--- x0 (b>0)
    x4 <--- x2 (b<0)
    x4 <--- x5 (b>0)
    x5 <--- x0 (b>0)
DAG[2]: 7.0%
    x0 <--- x3 (b>0)
    x1 <--- x0 (b>0)
    x1 <--- x2 (b>0)
    x1 <--- x3 (b>0)
    x2 <--- x3 (b>0)
    x4 <--- x0 (b>0)
    x4 <--- x2 (b<0)
    x5 <--- x0 (b>0)
```

2.2.6 Probability

Using the *get_probabilities()* method, we can get the probability of bootstrapping.

```
prob = result.get_probabilities(min_causal_effect=0.01)
print(prob)
```

```
[[0.   0.   0.1  0.99 0.02 0.  ]
 [1.   0.   1.   0.1  0.   0.05]
 [0.   0.   0.   0.96 0.   0.  ]
 [0.   0.   0.04 0.   0.   0.  ]
 [0.98 0.   0.94 0.02 0.   0.2 ]
 [1.   0.   0.   0.   0.   0.  ]]
```

2.2.7 Total Causal Effects

Using the `get_total_causal_effects()` method, we can get the list of total causal effect. The total causal effects we can get are dictionary type variable. We can display the list nicely by assigning it to pandas.DataFrame. Also, we have replaced the variable index with a label below.

```
causal_effects = result.get_total_causal_effects(min_causal_effect=0.01)

# Assign to pandas.DataFrame for pretty display
df = pd.DataFrame(causal_effects)
labels = [f'x{i}' for i in range(X.shape[1])]
df['from'] = df['from'].apply(lambda x : labels[x])
df['to'] = df['to'].apply(lambda x : labels[x])
df
```

We can easily perform sorting operations with pandas.DataFrame.

```
df.sort_values('effect', ascending=False).head()
```

```
df.sort_values('probability', ascending=True).head()
```

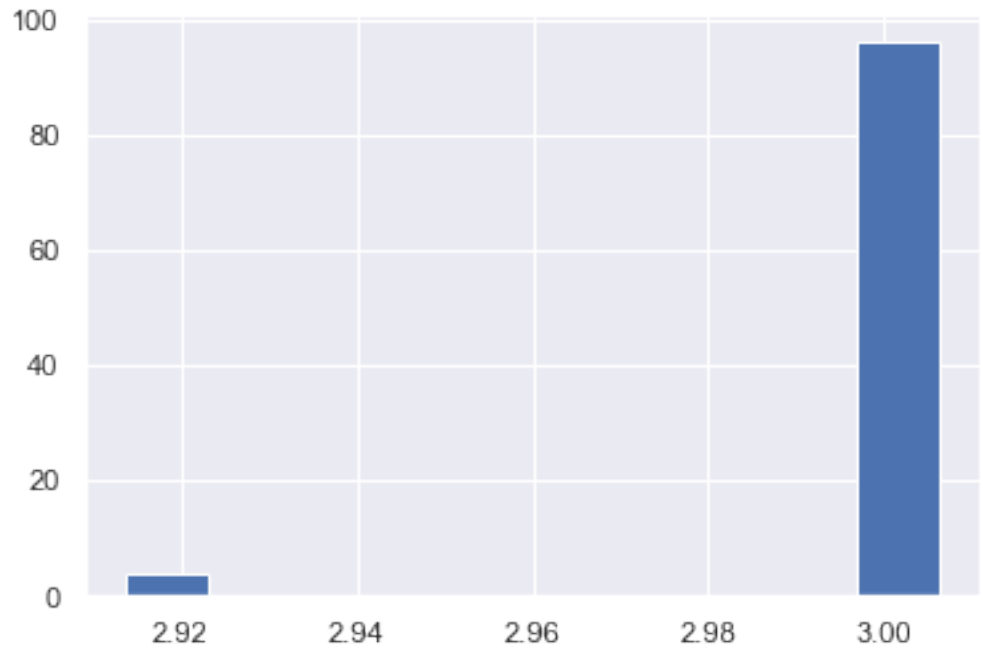
And with pandas.DataFrame, we can easily filter by keywords. The following code extracts the causal direction towards x1.

```
df[df['to']=='x1'].head()
```

Because it holds the raw data of the total causal effect (the original data for calculating the median), it is possible to draw a histogram of the values of the causal effect, as shown below.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline

from_index = 3 # index of x3
to_index = 0 # index of x0
plt.hist(result.total_effects_[:, to_index, from_index])
```



2.2.8 Bootstrap Probability of Path

Using the `get_paths()` method, we can explore all paths from any variable to any variable and calculate the bootstrap probability for each path. The path will be output as an array of variable indices. For example, the array `[3, 0, 1]` shows the path from variable X3 through variable X0 to variable X1.

```
from_index = 3 # index of x3
to_index = 1 # index of x0

pd.DataFrame(result.get_paths(from_index, to_index))
```

2.3 How to use prior knowledge in DirectLiNGAM

2.3.1 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import make_prior_knowledge, make_dot

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
np.random.seed(0)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.2']
```

2.3.2 Utility function

We define a utility function to draw the directed acyclic graph.

```
def make_prior_knowledge_graph(prior_knowledge_matrix):
    d = graphviz.Digraph(engine='dot')

    labels = [f'x{i}' for i in range(prior_knowledge_matrix.shape[0])]
    for label in labels:
        d.node(label, label)

    dirs = np.where(prior_knowledge_matrix > 0)
    for to, from_ in zip(dirs[0], dirs[1]):
        d.edge(labels[from_], labels[to])

    dirs = np.where(prior_knowledge_matrix < 0)
    for to, from_ in zip(dirs[0], dirs[1]):
        if to != from_:
            d.edge(labels[from_], labels[to], style='dashed')
    return d
```

2.3.3 Test data

We create test data consisting of 6 variables.

```
x3 = np.random.uniform(size=10000)
x0 = 3.0*x3 + np.random.uniform(size=10000)
x2 = 6.0*x3 + np.random.uniform(size=10000)
x1 = 3.0*x0 + 2.0*x2 + np.random.uniform(size=10000)
x5 = 4.0*x0 + np.random.uniform(size=10000)
x4 = 8.0*x0 - 1.0*x2 + np.random.uniform(size=10000)
X = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T ,columns=['x0', 'x1', 'x2', 'x3
↪', 'x4', 'x5'])
X.head()
```

```
m = np.array([[0.0, 0.0, 0.0, 3.0, 0.0, 0.0],
              [3.0, 0.0, 2.0, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 6.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
              [8.0, 0.0,-1.0, 0.0, 0.0, 0.0],
              [4.0, 0.0, 0.0, 0.0, 0.0, 0.0]])

make_dot(m)
```

2.3.4 Make Prior Knowledge Matrix

We create prior knowledge so that x0, x1 and x4 are sink variables.

The elements of prior knowledge matrix are defined as follows: * `0` : :math:`x_i` does not have a directed path to :math:`x_j` * `1` : :math:`x_i` has a directed path to :math:`x_j` * `-1` : No prior knowledge is available to know if either of the two cases above (0 or 1) is true.

```
prior_knowledge = make_prior_knowledge(
    n_variables=6,
    sink_variables=[0, 1, 4],
)
print(prior_knowledge)
```

```
[[-1  0 -1 -1  0 -1]
 [ 0 -1 -1 -1  0 -1]
 [ 0  0 -1 -1  0 -1]
 [ 0  0 -1 -1  0 -1]
 [ 0  0 -1 -1 -1 -1]
 [ 0  0 -1 -1  0 -1]]
```

```
# Draw a graph of prior knowledge
make_prior_knowledge_graph(prior_knowledge)
```

2.3.5 Causal Discovery

To run causal discovery using prior knowledge, we create a `DirectLiNGAM` object with the prior knowledge matrix.

```
model = lingam.DirectLiNGAM(prior_knowledge=prior_knowledge)
model.fit(X)
print(model.causal_order_)
print(model.adjacency_matrix_)
```

```
[3, 2, 5, 0, 1, 4]
[[ 0.     0.     0.     0.178  0.     0.235]
 [ 0.     0.     2.01   0.45   0.     0.707]
 [ 0.     0.     0.     6.001  0.     0.   ]
 [ 0.     0.     0.     0.     0.     0.   ]
 [ 0.     0.    -0.757  0.     0.     1.879]
 [ 0.     0.     0.    12.017  0.     0.   ]]
```

We can see that x0, x1, and x4 are output as sink variables, as specified in the prior knowledge.

```
make_dot(model.adjacency_matrix_)
```

Next, let's specify the prior knowledge so that x0 is an exogenous variable.

```
prior_knowledge = make_prior_knowledge(
    n_variables=6,
    exogenous_variables=[0],
)

model = lingam.DirectLiNGAM(prior_knowledge=prior_knowledge)
model.fit(X)

make_dot(model.adjacency_matrix_)
```

2.4 MultiGroupDirectLiNGAM

2.4.1 Model

This algorithm[3] simultaneously analyzes multiple datasets obtained from different sources, e.g., from groups of different ages. The algorithm is an extention of DirectLiNGAM[1] to multiple-group cases. The algorithm assumes that each dataset comes from a basic LiNGAM model[2], i.e., makes the following assumptions in each dataset:

1. Linearity
2. Non-Gaussian continuous error variables (except at most one)
3. Acyclicity
4. No hidden common causes

Further, it assumes the topological causal orders are common to the groups. The similarity in the topological causal orders would give a better performance than analyzing each dataset separatly if the assumption on the causal orders are reasonable.

References

2.4.2 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
```

(continues on next page)

[3] S. Shimizu. Joint estimation of linear non-Gaussian acyclic models. Neurocomputing, 81: 104-107, 2012.

[1] S. Shimizu, T. Inazumi, Y. Sogawa, A. Hyvärinen, Y. Kawahara, T. Washio, P. O. Hoyer and K. Bollen. DirectLiNGAM: A direct method for learning a linear non-Gaussian structural equation model. Journal of Machine Learning Research, 12(Apr): 1225–1248, 2011.

[2] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. J. Kerminen. A linear non-gaussian acyclic model for causal discovery. Journal of Machine Learning Research, 7:2003-2030, 2006.

(continued from previous page)

```
import lingam
from lingam.utils import print_causal_directions, print_dagc, make_dot

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
np.random.seed(0)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.4']
```

2.4.3 Test data

We generate two datasets consisting of 6 variables.

```
x3 = np.random.uniform(size=1000)
x0 = 3.0*x3 + np.random.uniform(size=1000)
x2 = 6.0*x3 + np.random.uniform(size=1000)
x1 = 3.0*x0 + 2.0*x2 + np.random.uniform(size=1000)
x5 = 4.0*x0 + np.random.uniform(size=1000)
x4 = 8.0*x0 - 1.0*x2 + np.random.uniform(size=1000)
X1 = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T ,columns=['x0', 'x1', 'x2', 'x3
↪', 'x4', 'x5'])
X1.head()
```

```
m = np.array([[0.0, 0.0, 0.0, 3.0, 0.0, 0.0],
              [3.0, 0.0, 2.0, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 6.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
              [8.0, 0.0,-1.0, 0.0, 0.0, 0.0],
              [4.0, 0.0, 0.0, 0.0, 0.0, 0.0]])

make_dot(m)
```

```
x3 = np.random.uniform(size=1000)
x0 = 3.5*x3 + np.random.uniform(size=1000)
x2 = 6.5*x3 + np.random.uniform(size=1000)
x1 = 3.5*x0 + 2.5*x2 + np.random.uniform(size=1000)
x5 = 4.5*x0 + np.random.uniform(size=1000)
x4 = 8.5*x0 - 1.5*x2 + np.random.uniform(size=1000)
X2 = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T ,columns=['x0', 'x1', 'x2', 'x3
↪', 'x4', 'x5'])
X2.head()
```

```
m = np.array([[0.0, 0.0, 0.0, 3.5, 0.0, 0.0],
              [3.5, 0.0, 2.5, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 6.5, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
              [8.5, 0.0,-1.5, 0.0, 0.0, 0.0],
              [4.5, 0.0, 0.0, 0.0, 0.0, 0.0]])

make_dot(m)
```

We create a list variable that contains two datasets.

```
X_list = [X1, X2]
```

2.4.4 Causal Discovery

To run causal discovery for multiple datasets, we create a *MultiGroupDirectLiNGAM* object and call the *fit()* method.

```
model = lingam.MultiGroupDirectLiNGAM()
model.fit(X_list)
```

```
<lingam.multi_group_direct_lingam.MultiGroupDirectLiNGAM at 0x21f895d0f60>
```

Using the *causal_order_* properties, we can see the causal ordering as a result of the causal discovery.

```
model.causal_order_
```

```
[3, 0, 5, 2, 1, 4]
```

Also, using the *adjacency_matrix_* properties, we can see the adjacency matrix as a result of the causal discovery. As you can see from the following, DAG in each dataset is correctly estimated.

```
print(model.adjacency_matrices_[0])
make_dot(model.adjacency_matrices_[0])
```

```
[[0.    0.    0.    3.006 0.    0.   ]
 [2.873 0.    1.969 0.    0.    0.   ]
 [0.    0.    0.    5.882 0.    0.   ]
 [0.    0.    0.    0.    0.    0.   ]
 [6.095 0.    0.    0.    0.    0.   ]
 [3.967 0.    0.    0.    0.    0.   ]]
```

```
print(model.adjacency_matrices_[1])
make_dot(model.adjacency_matrices_[1])
```

```
[[ 0.     0.     0.     3.483  0.     0.   ]
 [ 3.516  0.     2.466  0.165  0.     0.   ]
 [ 0.     0.     0.     6.383  0.     0.   ]
 [ 0.     0.     0.     0.     0.     0.   ]
 [ 8.456  0.    -1.471  0.     0.     0.   ]
 [ 4.446  0.     0.     0.     0.     0.   ]]
```

To compare, we run DirectLiNGAM with single dataset concatenating two datasets.

```
X_all = pd.concat([X1, X2])
print(X_all.shape)
```

```
(2000, 6)
```

```
model_all = lingam.DirectLiNGAM()
model_all.fit(X_all)

model_all.causal_order_
```

```
[1, 5, 2, 3, 0, 4]
```

You can see that the causal structure cannot be estimated correctly for a single dataset.

```
make_dot(model_all.adjacency_matrix_)
```

2.4.5 Independence between error variables

To check if the LiNGAM assumption is broken, we can get p-values of independence between error variables. The value in the i-th row and j-th column of the obtained matrix shows the p-value of the independence of the error variables e_i and e_j.

```
p_values = model.get_error_independence_p_values(X_list)
print(p_values[0])
```

```
[[0.    0.136 0.075 0.838 0.    0.832]
 [0.136 0.    0.008 0.    0.544 0.403]
 [0.075 0.008 0.    0.11  0.    0.511]
 [0.838 0.    0.11  0.    0.039 0.049]
 [0.    0.544 0.    0.039 0.    0.101]
 [0.832 0.403 0.511 0.049 0.101 0.   ]]
```

```
print(p_values[1])
```

```
[[0.    0.545 0.908 0.285 0.525 0.728]
 [0.545 0.    0.84  0.814 0.086 0.297]
 [0.908 0.84  0.    0.032 0.328 0.026]
 [0.285 0.814 0.032 0.    0.904 0.   ]
 [0.525 0.086 0.328 0.904 0.    0.237]
 [0.728 0.297 0.026 0.    0.237 0.   ]]
```

2.4.6 Bootstrapping

In *MultiGroupDirectLiNGAM*, bootstrap can be executed in the same way as normal *DirectLiNGAM*.

```
results = model.bootstrap(X_list, n_sampling=100)
```

2.4.7 Causal Directions

The *bootstrap()* method returns a list of multiple *BootstrapResult*, so we can get the result of bootstrapping from the list. We can get the same number of results as the number of datasets, so we specify an index when we access the results. We can get the ranking of the causal directions extracted by *get_causal_direction_counts()*.

```
cdc = results[0].get_causal_direction_counts(n_directions=8, min_causal_effect=0.01)
print_causal_directions(cdc, 100)
```

```
x0 <--- x3  (100.0%)
x1 <--- x0  (100.0%)
x1 <--- x2  (100.0%)
x2 <--- x3  (100.0%)
x4 <--- x0  (100.0%)
x5 <--- x0  (100.0%)
x4 <--- x2  (94.0%)
x4 <--- x5  (20.0%)
```

```
cdc = results[1].get_causal_direction_counts(n_directions=8, min_causal_effect=0.01)
print_causal_directions(cdc, 100)
```

```
x0 <--- x3  (100.0%)
x1 <--- x0  (100.0%)
x1 <--- x2  (100.0%)
x2 <--- x3  (100.0%)
x4 <--- x0  (100.0%)
x4 <--- x2  (100.0%)
x5 <--- x0  (100.0%)
x1 <--- x3  (72.0%)
```

2.4.8 Directed Acyclic Graphs

Also, using the *get_directed_acyclic_graph_counts()* method, we can get the ranking of the DAGs extracted. In the following sample code, `n_dags` option is limited to the dags of the top 3 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
dagc = results[0].get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.01)
print_dagc(dagc, 100)
```

```
DAG[0]: 61.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x5 <--- x0
DAG[1]: 13.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x4 <--- x5
    x5 <--- x0
DAG[2]: 6.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
```

(continues on next page)

(continued from previous page)

```
    x2 <--- x3
    x4 <--- x0
    x5 <--- x0
```

```
dagc = results[1].get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.01)
print_dagc(dagc, 100)
```

```
DAG[0]: 59.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x1 <--- x3
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x5 <--- x0
DAG[1]: 17.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x5 <--- x0
DAG[2]: 10.0%
    x0 <--- x2
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x1 <--- x3
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x5 <--- x0
```

2.4.9 Probability

Using the *get_probabilities()* method, we can get the probability of bootstrapping.

```
prob = results[0].get_probabilities(min_causal_effect=0.01)
print(prob)
```

```
[[0.   0.   0.08 1.   0.   0.  ]
 [1.   0.   1.   0.08 0.   0.05]
 [0.   0.   0.   1.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.  ]
 [1.   0.   0.94 0.   0.   0.2 ]
 [1.   0.   0.   0.   0.01 0.  ]]
```

2.4.10 Total Causal Effects

Using the `get_total_causal_effects()` method, we can get the list of total causal effect. The total causal effects we can get are dictionary type variable. We can display the list nicely by assigning it to pandas.DataFrame.

Also, we have replaced the variable index with a label below.

```
causal_effects = results[0].get_total_causal_effects(min_causal_effect=0.01)
df = pd.DataFrame(causal_effects)

labels = [f'x{i}' for i in range(X1.shape[1])]
df['from'] = df['from'].apply(lambda x : labels[x])
df['to'] = df['to'].apply(lambda x : labels[x])
df
```

We can easily perform sorting operations with pandas.DataFrame.

```
df.sort_values('effect', ascending=False).head()
```

And with pandas.DataFrame, we can easily filter by keywords. The following code extracts the causal direction towards x1.

```
df[df['to']=='x1'].head()
```

Because it holds the raw data of the causal effect (the original data for calculating the median), it is possible to draw a histogram of the values of the causal effect, as shown below.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline

from_index = 3
to_index = 0
plt.hist(results[0].total_effects_[:, to_index, from_index])
```

2.4.11 Bootstrap Probability of Path

Using the `get_paths()` method, we can explore all paths from any variable to any variable and calculate the bootstrap probability for each path. The path will be output as an array of variable indices. For example, the array `[3, 0, 1]` shows the path from variable X3 through variable X0 to variable X1.

```
from_index = 3 # index of x3
to_index = 1 # index of x0

pd.DataFrame(results[0].get_paths(from_index, to_index))
```

2.5 Total Effect

We use lingam package:

```
import lingam
```

Then, we create a *DirectLiNGAM* object and call the *fit()* method:

```
model = lingam.DirectLiNGAM()
model.fit(X)
```

To estimate the total effect, we can call *estimate_total_effect()* method. The following example estimates the total effect from x3 to x1.

```
te = model.estimate_total_effect(X, 3, 1)
print(f'total effect: {te:.3f}')
```

```
total effect: 21.002
```

For details, see also https://github.com/cdt15/lingam/blob/master/examples/TotalEffect.ipynb

2.6 Causal Effect on predicted variables

The following demonstrates a method[1] that analyzes the prediction mechanisms of constructed predictive models based on causality. This method estimates causal effects, i.e., intervention effects of features or explnatory variables used in constructed predictive models on the predicted variables. Users can use estimated causal structures, e.g., by a LiNGAM-type method or known causal structures based on domain knowledge.

References

In Proc. 2017 IEEE International Workshop on Machine Learning for Signal Processing (MLSP2017), pp. 1–6, Tokyo, Japan, 2017.

First, we use lingam package:

```
import lingam
```

Then, we create a *DirectLiNGAM* object and call the *fit()* method:

```
model = lingam.DirectLiNGAM()
model.fit(X)
```

Next, we create the prediction model. In the following example, linear regression model is created, but it is also possible to create logistic regression model or non-linear regression model.

```
from sklearn.linear_model import LinearRegression

target = 0
features = [i for i in range(X.shape[1]) if i != target]
reg = LinearRegression()
reg.fit(X.iloc[:, features], X.iloc[:, target])
```

2.6.1 Identification of Feature with Greatest Causal Influence on Prediction

We create a *CausalEffect* object and call the *estimate_effects_on_prediction()* method.

```
ce = lingam.CausalEffect(model)
effects = ce.estimate_effects_on_prediction(X, target, reg)
```

To identify of the feature having the greatest intervention effect on the prediction, we can get the feature that maximizes the value of the obtained list.

[1] P. Blöbaum and S. Shimizu. Estimation of interventional effects of features on prediction.

```
print(X.columns[np.argmax(effects)])
```

```
cylinders
```

2.6.2 Estimation of Optimal Intervention

To estimate of the intervention such that the expectation of the prediction of the post-intervention observations is equal or close to a specified value, we use *estimate_optimal_intervention()* method of *CausalEffect*. In the following example, we estimate the intervention value at variable index 1 so that the predicted value is close to 15.

```
c = ce.estimate_optimal_intervention(X, target, reg, 1, 15)
print(f'Optimal intervention: {c:.3f}')
```

```
Optimal intervention: 7.871
```

2.6.3 Use a known causal model

When using a known causal model, we can specify the adjacency matrix when we create *CausalEffect* object.

```
m = np.array([[0.0, 0.0, 0.0, 3.0, 0.0, 0.0],
              [3.0, 0.0, 2.0, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 6.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
              [8.0, 0.0,-1.0, 0.0, 0.0, 0.0],
              [4.0, 0.0, 0.0, 0.0, 0.0, 0.0]])

ce = lingam.CausalEffect(causal_model=m)
effects = ce.estimate_effects_on_prediction(X, target, reg)
```

For details, see also:

- https://github.com/cdt15/lingam/blob/master/examples/CausalEffect.ipynb
- https://github.com/cdt15/lingam/blob/master/examples/CausalEffect(LassoCV).ipynb
- https://github.com/cdt15/lingam/blob/master/examples/CausalEffect(LogisticRegression).ipynb
- https://github.com/cdt15/lingam/blob/master/examples/CausalEffect(LightGBM).ipynb

2.7 VARLiNGAM

2.7.1 Model

VARLiNGAM[2] is an extension of the basic LiNGAM model[1] to time series cases. It combines the basic LiNGAM model with the classic vector autoregressive models (VAR). It enables analyzing both lagged and contemporaneous (instantaneous) causal relations, whereas the classic VAR only analyzes lagged causal relations. This VARLiNGAM makes the following assumptions similarly to the basic LiNGAM model[1]: #. Linearity #. Non-Gaussian continuous error variables (except at most one) #. Acyclicity of contemporaneous causal relations #. No hidden common causes

[2] A. Hyvärinen, K. Zhang, S. Shimizu, and P. O. Hoyer. Estimation of a structural vector autoregression model using non-Gaussianity. Journal of Machine Learning Research, 11: 1709-1731, 2010.

[1] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. J. Kerminen. A linear non-gaussian acyclic model for causal discovery. Journal of Machine Learning Research, 7:2003-2030, 2006.

Example applications are found here, especially in Section. Economics/Finance/Marketing. For example,[3] uses the VARLiNGAM model to to study the processes of firm growth and firm performance using microeconomic data and to analyse the effects of monetary policy using macroeconomic data.

References

2.7.2 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import make_dot, print_causal_directions, print_dagc

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
np.random.seed(0)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.2']
```

2.7.3 Test data

We create test data consisting of 5 variables.

```
B0 = [
    [0,-0.12,0,0,0],
    [0,0,0,0,0],
    [-0.41,0.01,0,-0.02,0],
    [0.04,-0.22,0,0,0],
    [0.15,0,-0.03,0,0],
]
B1 = [
    [-0.32,0,0.12,0.32,0],
    [0,-0.35,-0.1,-0.46,0.4],
    [0,0,0.37,0,0.46],
    [-0.38,-0.1,-0.24,0,-0.13],
    [0,0,0,0,0],
]
causal_order = [1, 0, 3, 2, 4]

# data generated from B0 and B1
X = pd.read_csv('data/sample_data_var_lingam.csv')
```

2.7.4 Causal Discovery

To run causal discovery, we create a *VARLiNGAM* object and call the *fit()* method.

[3] A. Moneta, D. Entner, P. O. Hoyer and A. Coad. Causal inference by independent component analysis: Theory and applications. Oxford Bulletin of Economics and Statistics, 75(5): 705-730, 2013.

```
model = lingam.VARLiNGAM()
model.fit(X)
```

```
<lingam.var_lingam.VARLiNGAM at 0x20510e049b0>
```

Using the *causal_order_* properties, we can see the causal ordering as a result of the causal discovery.

```
model.causal_order_
```

```
[1, 0, 3, 2, 4]
```

Also, using the *adjacency_matrices_* properties, we can see the adjacency matrix as a result of the causal discovery.

```
# B0
model.adjacency_matrices_[0]
```

```
array([[ 0.   , -0.144,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ],
       [-0.372,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 0.069, -0.21 ,  0.   ,  0.   ,  0.   ],
       [ 0.083,  0.   , -0.033,  0.   ,  0.   ]])
```

```
# B1
model.adjacency_matrices_[1]
```

```
array([[-0.366, -0.011,  0.074,  0.297,  0.025],
       [-0.083, -0.349, -0.168, -0.327,  0.43 ],
       [ 0.077, -0.043,  0.427,  0.046,  0.49 ],
       [-0.389, -0.097, -0.263,  0.014, -0.159],
       [-0.018,  0.01 ,  0.001,  0.071,  0.003]])
```

```
model.residuals_
```

```
array([[-0.308,  0.911, -1.152, -1.159,  0.179],
       [ 1.364,  1.713, -1.389, -0.265, -0.192],
       [-0.861,  0.249,  0.479, -1.557, -0.462],
       ...,
       [-1.202,  1.819,  0.99 , -0.855, -0.127],
       [-0.133,  1.23 , -0.445, -0.753,  1.096],
       [-0.069,  0.558,  0.21 , -0.863, -0.189]])
```

Using `DirectLiNGAM` for the `residuals_` properties, we can calculate B0 matrix.

```
dlingam = lingam.DirectLiNGAM()
dlingam.fit(model.residuals_)
dlingam.adjacency_matrix_
```

```
array([[ 0.   , -0.144,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ],
       [-0.372,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 0.069, -0.21 ,  0.   ,  0.   ,  0.   ],
       [ 0.083,  0.   , -0.033,  0.   ,  0.   ]])
```

We can draw a causal graph by utility funciton.

```
labels = ['x0(t)', 'x1(t)', 'x2(t)', 'x3(t)', 'x4(t)', 'x0(t-1)', 'x1(t-1)', 'x2(t-1)
↪', 'x3(t-1)', 'x4(t-1)']
make_dot(np.hstack(model.adjacency_matrices_), ignore_shape=True, lower_limit=0.05,␣
↪labels=labels)
```

2.7.5 Independence between error variables

To check if the LiNGAM assumption is broken, we can get p-values of independence between error variables. The value in the i-th row and j-th column of the obtained matrix shows the p-value of the independence of the error variables e_i and e_j.

```
p_values = model.get_error_independence_p_values()
print(p_values)
```

```
[[0.    0.065 0.068 0.038 0.249]
 [0.065 0.    0.13  0.88  0.57 ]
 [0.068 0.13  0.    0.321 0.231]
 [0.038 0.88  0.321 0.    0.839]
 [0.249 0.57  0.231 0.839 0.   ]]
```

2.7.6 Bootstrap

2.7.6.1 Bootstrapping

We call *bootstrap()* method instead of *fit()*. Here, the second argument specifies the number of bootstrap sampling.

```
model = lingam.VARLiNGAM()
result = model.bootstrap(X, n_sampling=100)
```

2.7.7 Causal Directions

Since *BootstrapResult* object is returned, we can get the ranking of the causal directions extracted by *get_causal_direction_counts()* method. In the following sample code, `n_directions` option is limited to the causal directions of the top 8 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.3 or more.

```
cdc = result.get_causal_direction_counts(n_directions=8, min_causal_effect=0.3, split_
↪by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_causal_directions(cdc, 100, labels=labels)
```

```
x0(t) <--- x0(t-1) (b<0) (100.0%)
x1(t) <--- x1(t-1) (b<0) (100.0%)
x1(t) <--- x3(t-1) (b<0) (100.0%)
x1(t) <--- x4(t-1) (b>0) (100.0%)
x2(t) <--- x2(t-1) (b>0) (100.0%)
```

(continues on next page)

(continued from previous page)

```
x2(t) <--- x4(t-1) (b>0) (100.0%)
x3(t) <--- x0(t-1) (b<0) (100.0%)
x2(t) <--- x0(t) (b<0) (99.0%)
```

2.7.8 Directed Acyclic Graphs

Also, using the *get_directed_acyclic_graph_counts()* method, we can get the ranking of the DAGs extracted. In the following sample code, `n_dags` option is limited to the dags of the top 3 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.2 or more.

```
dagc = result.get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.2,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_dagc(dagc, 100, labels=labels)
```

```
DAG[0]: 57.0%
    x0(t) <--- x0(t-1) (b<0)
    x0(t) <--- x3(t-1) (b>0)
    x1(t) <--- x1(t-1) (b<0)
    x1(t) <--- x3(t-1) (b<0)
    x1(t) <--- x4(t-1) (b>0)
    x2(t) <--- x0(t) (b<0)
    x2(t) <--- x2(t-1) (b>0)
    x2(t) <--- x4(t-1) (b>0)
    x3(t) <--- x1(t) (b<0)
    x3(t) <--- x0(t-1) (b<0)
    x3(t) <--- x2(t-1) (b<0)
DAG[1]: 42.0%
    x0(t) <--- x0(t-1) (b<0)
    x0(t) <--- x3(t-1) (b>0)
    x1(t) <--- x1(t-1) (b<0)
    x1(t) <--- x3(t-1) (b<0)
    x1(t) <--- x4(t-1) (b>0)
    x2(t) <--- x0(t) (b<0)
    x2(t) <--- x2(t-1) (b>0)
    x2(t) <--- x4(t-1) (b>0)
    x3(t) <--- x0(t-1) (b<0)
    x3(t) <--- x2(t-1) (b<0)
DAG[2]: 1.0%
    x0(t) <--- x0(t-1) (b<0)
    x0(t) <--- x3(t-1) (b>0)
    x1(t) <--- x1(t-1) (b<0)
    x1(t) <--- x3(t-1) (b<0)
    x1(t) <--- x4(t-1) (b>0)
    x2(t) <--- x0(t) (b<0)
    x2(t) <--- x2(t-1) (b>0)
    x2(t) <--- x4(t-1) (b>0)
    x3(t) <--- x1(t) (b<0)
    x3(t) <--- x0(t-1) (b<0)
    x3(t) <--- x2(t-1) (b<0)
    x4(t) <--- x0(t) (b>0)
```

2.7.9 Probability

Using the *get_probabilities()* method, we can get the probability of bootstrapping.

```
prob = result.get_probabilities(min_causal_effect=0.1)
print('Probability of B0:\n', prob[0])
print('Probability of B1:\n', prob[1])
```

```
Probability of B0:
 [[0.   0.98 0.   0.02 0.  ]
 [0.   0.   0.   0.   0.  ]
 [1.   0.   0.   0.   0.01]
 [0.1  1.   0.   0.   0.  ]
 [0.51 0.   0.02 0.08 0.  ]]
Probability of B1:
 [[1.   0.   0.02 1.   0.  ]
 [0.   1.   1.   1.   1.  ]
 [0.03 0.   1.   0.05 1.  ]
 [1.   0.16 1.   0.   1.  ]
 [0.   0.   0.   0.25 0.  ]]
```

2.7.10 Total Causal Effects

Using the `get_causal_effects()` method, we can get the list of total causal effect. The total causal effects we can get are dictionary type variable. We can display the list nicely by assigning it to pandas.DataFrame. Also, we have replaced the variable index with a label below.

```
causal_effects = result.get_total_causal_effects(min_causal_effect=0.01)
df = pd.DataFrame(causal_effects)

df['from'] = df['from'].apply(lambda x : labels[x])
df['to'] = df['to'].apply(lambda x : labels[x])
df
```

We can easily perform sorting operations with pandas.DataFrame.

```
df.sort_values('effect', ascending=False).head()
```

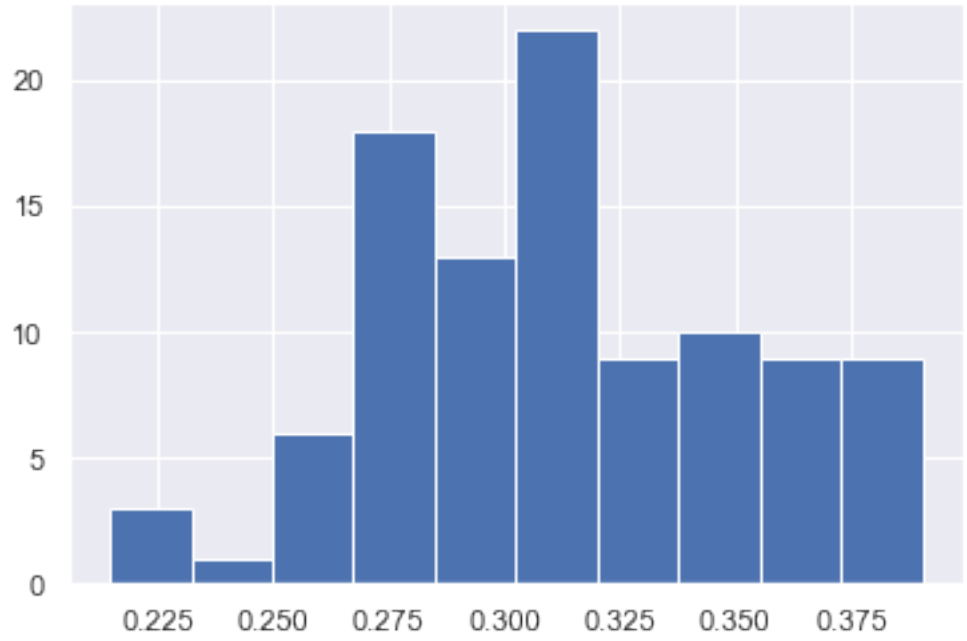
And with pandas.DataFrame, we can easily filter by keywords. The following code extracts the causal direction towards x1(t).

```
df[df['to']=='x1(t)'].head()
```

Because it holds the raw data of the causal effect (the original data for calculating the median), it is possible to draw a histogram of the values of the causal effect, as shown below.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline

from_index = 7 # index of x2(t-1). (index:2)+(n_features:5)*(lag:1) = 7
to_index = 2 # index of x2(t). (index:2)+(n_features:5)*(lag:0) = 2
plt.hist(result.total_effects_[:, to_index, from_index])
```



2.8 VARMALiNGAM

2.8.1 Model

VARMALiNGAM[3] is an extension of the basic LiNGAM model[1] to time series cases. It combines the basic LiNGAM model with the classic vector autoregressive moving average models (VARMA). It enables analyzing both lagged and contemporaneous (instantaneous) causal relations, whereas the classic VARMA only analyzes lagged causal relations. This VARMALiNGAM model also is an extension of the VARLiNGAM model[2]. It uses VARMA to analyze lagged causal relations instead of VAR. This VARMALiNGAM makes the following assumptions similarly to the basic LiNGAM model[1]: #. Linearity #. Non-Gaussian continuous error variables (except at most one) #. Acyclicity of contemporaneous causal relations #. No hidden common causes between contempraneous error variables

References

2.8.2 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
```

(continues on next page)

[3] Y. Kawahara, S. Shimizu and T. Washio. Analyzing relationships among ARMA processes based on non-Gaussianity of external influences. Neurocomputing, 74(12-13): 2212-2221, 2011. [PDF]

[1] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. J. Kerminen. A linear non-gaussian acyclic model for causal discovery. Journal of Machine Learning Research, 7:2003-2030, 2006.

[2] A. Hyvärinen, K. Zhang, S. Shimizu, and P. O. Hoyer. Estimation of a structural vector autoregression model using non-Gaussianity. Journal of Machine Learning Research, 11: 1709-1731, 2010.

(continued from previous page)

```
import lingam
from lingam.utils import make_dot, print_causal_directions, print_dagc

import warnings
warnings.filterwarnings('ignore')

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
np.random.seed(0)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.2']
```

2.8.3 Test data

We create test data consisting of 5 variables.

```
psi0 = np.array([
    [ 0.  ,  0.  , -0.25,  0.  ,  0.  ],
    [-0.38,  0.  ,  0.14,  0.  ,  0.  ],
    [ 0.  ,  0.  ,  0.  ,  0.  ,  0.  ],
    [ 0.44, -0.2 , -0.09,  0.  ,  0.  ],
    [ 0.07, -0.06,  0.  ,  0.07,  0.  ]
])
phi1 = np.array([
    [-0.04, -0.29, -0.26,  0.14,  0.47],
    [-0.42,  0.2 ,  0.1 ,  0.24,  0.25],
    [-0.25,  0.18, -0.06,  0.15,  0.18],
    [ 0.22,  0.39,  0.08,  0.12, -0.37],
    [-0.43,  0.09, -0.23,  0.16,  0.25]
])
theta1 = np.array([
    [ 0.15, -0.02, -0.3 , -0.2 ,  0.21],
    [ 0.32,  0.12, -0.11,  0.03,  0.42],
    [-0.07, -0.5 ,  0.03, -0.27, -0.21],
    [-0.17,  0.35,  0.25,  0.24, -0.25],
    [ 0.09,  0.4 ,  0.41,  0.24, -0.31]
])
causal_order = [2, 0, 1, 3, 4]

# data generated from psi0 and phi1 and theta1, causal_order
X = np.loadtxt('data/sample_data_varma_lingam.csv', delimiter=',')
```

2.8.4 Causal Discovery

To run causal discovery, we create a *VARMALiNGAM* object and call the *fit()* method.

```
model = lingam.VARMALiNGAM(order=(1, 1), criterion=None)
model.fit(X)
```

```
<lingam.varma_lingam.VARMALiNGAM at 0x1acfc3fa6d8>
```

Using the *causal_order_* properties, we can see the causal ordering as a result of the causal discovery.

```
model.causal_order_
```

```
[2, 0, 1, 3, 4]
```

Also, using the *adjacency_matrices_* properties, we can see the adjacency matrix as a result of the causal discovery.

```
# psi0
model.adjacency_matrices_[0][0]
```

```
array([[ 0.   ,  0.   , -0.238,  0.   ,  0.   ],
       [-0.392,  0.   ,  0.182,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 0.523, -0.149,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ]])
```

```
# psi1
model.adjacency_matrices_[0][1]
```

```
array([[-0.145, -0.288, -0.418,  0.041,  0.592],
       [-0.324,  0.027,  0.024,  0.231,  0.379],
       [-0.249,  0.191, -0.01 ,  0.136,  0.261],
       [ 0.182,  0.698,  0.21 ,  0.197, -0.815],
       [-0.486,  0.063, -0.263,  0.112,  0.26 ]])
```

```
# omega0
model.adjacency_matrices_[1][0]
```

```
array([[ 0.247, -0.12 , -0.128, -0.124,  0.037],
       [ 0.378,  0.319, -0.12 , -0.023,  0.573],
       [-0.107, -0.624,  0.012, -0.303, -0.246],
       [-0.22 ,  0.26 ,  0.313,  0.227, -0.057],
       [ 0.255,  0.405,  0.41 ,  0.256, -0.286]])
```

Using `DirectLiNGAM` for the `residuals_` properties, we can calculate psi0 matrix.

```
dlingam = lingam.DirectLiNGAM()
dlingam.fit(model.residuals_)
dlingam.adjacency_matrix_
```

```
array([[ 0.   ,  0.   , -0.238,  0.   ,  0.   ],
       [-0.392,  0.   ,  0.182,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 0.523, -0.149,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ]])
```

We can draw a causal graph by utility funciton

```
labels = ['y0(t)', 'y1(t)', 'y2(t)', 'y3(t)', 'y4(t)', 'y0(t-1)', 'y1(t-1)', 'y2(t-1)
↪', 'y3(t-1)', 'y4(t-1)']
make_dot(np.hstack(model.adjacency_matrices_[0]), lower_limit=0.3, ignore_shape=True,␣
↪labels=labels)
```

2.8.5 Independence between error variables

To check if the LiNGAM assumption is broken, we can get p-values of independence between error variables. The value in the i-th row and j-th column of the obtained matrix shows the p-value of the independence of the error variables e_i and e_j.

```
p_values = model.get_error_independence_p_values()
print(p_values)
```

```
[[0.    0.517 0.793 0.004 0.001]
 [0.517 0.    0.09  0.312 0.071]
 [0.793 0.09  0.    0.058 0.075]
 [0.004 0.312 0.058 0.    0.011]
 [0.001 0.071 0.075 0.011 0.   ]]
```

2.8.6 Bootstrap

2.8.6.1 Bootstrapping

We call *bootstrap()* method instead of *fit()*. Here, the second argument specifies the number of bootstrap sampling.

```
model = lingam.VARMALiNGAM()
result = model.bootstrap(X, n_sampling=100)
```

2.8.7 Causal Directions

Since *BootstrapResult* object is returned, we can get the ranking of the causal directions extracted by *get_causal_direction_counts()* method. In the following sample code, `n_directions` option is limited to the causal directions of the top 8 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.4 or more.

```
cdc = result.get_causal_direction_counts(n_directions=8, min_causal_effect=0.4, split_
↪by_causal_effect_sign=True)
```

We can check the result by utility function.

```
labels = ['y0(t)', 'y1(t)', 'y2(t)', 'y3(t)', 'y4(t)', 'y0(t-1)', 'y1(t-1)', 'y2(t-1)
↪', 'y3(t-1)', 'y4(t-1)', 'e0(t-1)', 'e1(t-1)', 'e2(t-1)', 'e3(t-1)', 'e4(t-1)']
print_causal_directions(cdc, 100, labels=labels)
```

```
y0(t) <--- y2(t-1) (b<0) (100.0%)
y0(t) <--- y4(t-1) (b>0) (100.0%)
y1(t) <--- e4(t-1) (b>0) (100.0%)
y2(t) <--- e1(t-1) (b<0) (100.0%)
y3(t) <--- y0(t) (b>0) (100.0%)
y3(t) <--- y1(t-1) (b>0) (100.0%)
y3(t) <--- y4(t-1) (b<0) (100.0%)
y4(t) <--- y0(t-1) (b<0) (100.0%)
```

2.8.8 Directed Acyclic Graphs

Also, using the *get_directed_acyclic_graph_counts()* method, we can get the ranking of the DAGs extracted. In the following sample code, `n_dags` option is limited to the dags of the top 3 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.3 or more.

```
dagc = result.get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.3,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_dagc(dagc, 100, labels=labels)
```

```
DAG[0]: 40.0%
    y0(t) <--- y2(t-1) (b<0)
    y0(t) <--- y4(t-1) (b>0)
    y1(t) <--- y0(t) (b<0)
    y1(t) <--- y0(t-1) (b<0)
    y1(t) <--- y4(t-1) (b>0)
    y1(t) <--- e0(t-1) (b>0)
    y1(t) <--- e1(t-1) (b>0)
    y1(t) <--- e4(t-1) (b>0)
    y2(t) <--- e1(t-1) (b<0)
    y2(t) <--- e3(t-1) (b<0)
    y3(t) <--- y0(t) (b>0)
    y3(t) <--- y1(t-1) (b>0)
    y3(t) <--- y4(t-1) (b<0)
    y3(t) <--- e2(t-1) (b>0)
    y4(t) <--- y0(t-1) (b<0)
    y4(t) <--- e1(t-1) (b>0)
    y4(t) <--- e2(t-1) (b>0)
DAG[1]: 19.0%
    y0(t) <--- y2(t-1) (b<0)
    y0(t) <--- y4(t-1) (b>0)
    y1(t) <--- y0(t) (b<0)
    y1(t) <--- y0(t-1) (b<0)
    y1(t) <--- y4(t-1) (b>0)
    y1(t) <--- e0(t-1) (b>0)
    y1(t) <--- e4(t-1) (b>0)
    y2(t) <--- e1(t-1) (b<0)
    y2(t) <--- e3(t-1) (b<0)
    y3(t) <--- y0(t) (b>0)
    y3(t) <--- y1(t-1) (b>0)
    y3(t) <--- y4(t-1) (b<0)
    y3(t) <--- e2(t-1) (b>0)
    y4(t) <--- y0(t-1) (b<0)
    y4(t) <--- e1(t-1) (b>0)
    y4(t) <--- e2(t-1) (b>0)
DAG[2]: 7.0%
    y0(t) <--- y2(t) (b<0)
    y0(t) <--- y2(t-1) (b<0)
    y0(t) <--- y4(t-1) (b>0)
    y1(t) <--- y0(t) (b<0)
    y1(t) <--- y0(t-1) (b<0)
    y1(t) <--- y4(t-1) (b>0)
    y1(t) <--- e0(t-1) (b>0)
    y1(t) <--- e1(t-1) (b>0)
```

(continues on next page)

(continued from previous page)

```
    y1(t) <--- e4(t-1) (b>0)
    y2(t) <--- e1(t-1) (b<0)
    y2(t) <--- e3(t-1) (b<0)
    y3(t) <--- y0(t) (b>0)
    y3(t) <--- y1(t-1) (b>0)
    y3(t) <--- y4(t-1) (b<0)
    y3(t) <--- e2(t-1) (b>0)
    y4(t) <--- y0(t-1) (b<0)
    y4(t) <--- e1(t-1) (b>0)
    y4(t) <--- e2(t-1) (b>0)
```

2.8.9 Probability

Using the *get_probabilities()* method, we can get the probability of bootstrapping.

```
prob = result.get_probabilities(min_causal_effect=0.1)
print('Probability of psi0:\n', prob[0])
print('Probability of psi1:\n', prob[1])
print('Probability of omega1:\n', prob[2])
```

```
Probability of psi0:
 [[0.   0.   1.   0.   0.  ]
 [1.   0.   0.95 0.   0.  ]
 [0.   0.   0.   0.   0.  ]
 [1.   0.96 0.24 0.   0.  ]
 [0.16 0.03 0.1  0.04 0.  ]]
Probability of psi1:
 [[1.   1.   1.   0.   1.  ]
 [1.   0.   0.   1.   1.  ]
 [1.   1.   0.   1.   1.  ]
 [1.   1.   1.   1.   1.  ]
 [1.   0.19 1.   0.96 1.  ]]
Probability of omega1:
 [[1.   0.77 1.   0.96 0.  ]
 [1.   1.   1.   0.   1.  ]
 [1.   1.   0.   1.   1.  ]
 [1.   1.   1.   1.   0.04]
 [1.   1.   1.   1.   1.  ]]
```

2.8.10 Total Causal Effects

Using the `get_total causal_effects()` method, we can get the list of total causal effect. The total causal effects we can get are dictionary type variable. We can display the list nicely by assigning it to pandas.DataFrame. Also, we have replaced the variable index with a label below.

```
causal_effects = result.get_total_causal_effects(min_causal_effect=0.01)
df = pd.DataFrame(causal_effects)

df['from'] = df['from'].apply(lambda x : labels[x])
df['to'] = df['to'].apply(lambda x : labels[x])
df
```

We can easily perform sorting operations with pandas.DataFrame.

```
df.sort_values('effect', ascending=False).head()
```

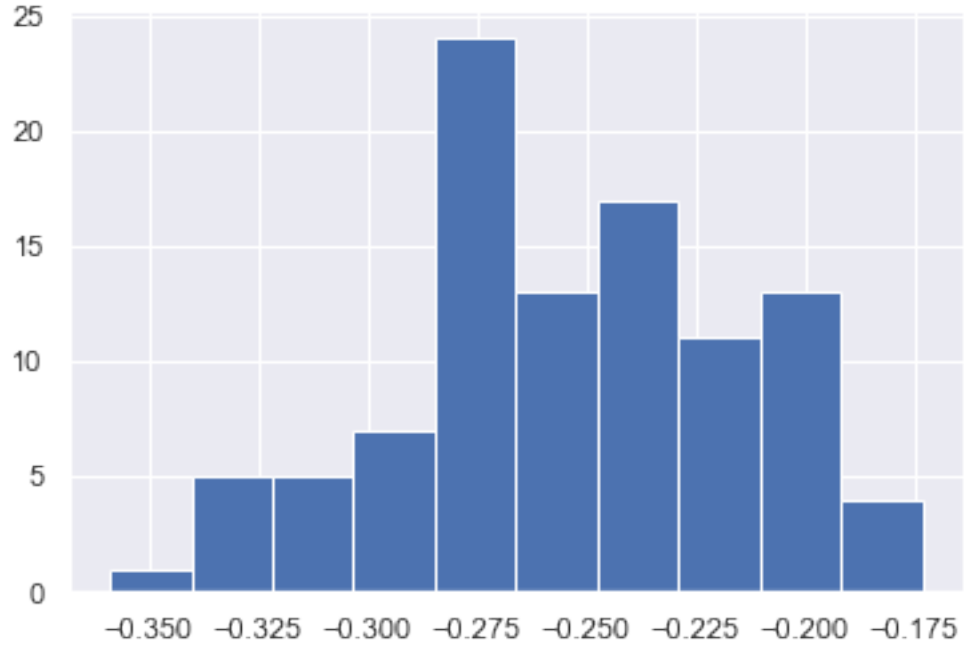
And with pandas.DataFrame, we can easily filter by keywords. The following code extracts the causal direction towards y2(t).

```
df[df['to']=='y2(t)'].head()
```

Because it holds the raw data of the causal effect (the original data for calculating the median), it is possible to draw a histogram of the values of the causal effect, as shown below.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline

from_index = 5 # index of y0(t-1). (index:0)+(n_features:5)*(lag:1) = 5
to_index = 2 # index of y2(t). (index:2)+(n_features:5)*(lag:0) = 2
plt.hist(result.total_effects_[:, to_index, from_index])
```



2.9 Longitudinal LiNGAM

2.9.1 Model

This method[2] performs causal discovery on paired samples based on longitudinal data that collects samples over time. Their algorithm can analyze causal structures, including topological causal orders, that may change over time. Similarly to the basic LiNGAM model[1], this method makes the following assumptions:

[2] K. Kadowaki, S. Shimizu, and T. Washio. Estimation of causal structures in longitudinal data using non-Gaussianity. In Proc. 23rd IEEE International Workshop on Machine Learning for Signal Processing (MLSP2013), pp. 1–6, Southampton, United Kingdom, 2013.

[1] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. J. Kerminen. A linear non-gaussian acyclic model for causal discovery. Journal of Machine Learning Research, 7:2003-2030, 2006.

1. Linearity
2. Non-Gaussian continuous error variables (except at most one)
3. Acyclicity
4. No hidden common causes

References

2.9.2 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import graphviz
import lingam
from lingam.utils import print_causal_directions, print_dagc, make_dot

import warnings
warnings.filterwarnings('ignore')

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
np.random.seed(0)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.2']
```

2.9.3 Test data

We create test data consisting of 5 variables. The causal model at each timepoint is as follows.

```
# setting
n_features = 5
n_samples = 200
n_lags = 1
n_timepoints = 3

causal_orders = []
B_t_true = np.empty((n_timepoints, n_features, n_features))
B_tau_true = np.empty((n_timepoints, n_lags, n_features, n_features))
X_t = np.empty((n_timepoints, n_samples, n_features))
```

```
# B(0,0)
B_t_true[0] = np.array([[0.0, 0.5,-0.3, 0.0, 0.0],
                        [0.0, 0.0,-0.3, 0.4, 0.0],
                        [0.0, 0.0, 0.0, 0.3, 0.0],
                        [0.0, 0.0, 0.0, 0.0, 0.0],
                        [0.1,-0.7, 0.0, 0.0, 0.0]])
causal_orders.append([3, 2, 1, 0, 4])
make_dot(B_t_true[0], labels=[f'x{i}(0)' for i in range(5)])
```

```
# B(1,1)
B_t_true[1] = np.array([[0.0, 0.2,-0.1, 0.0,-0.5],
                        [0.0, 0.0, 0.0, 0.4, 0.0],
                        [0.0, 0.3, 0.0, 0.0, 0.0],
                        [0.0, 0.0, 0.0, 0.0, 0.0],
                        [0.0,-0.4, 0.0, 0.0, 0.0]])
causal_orders.append([3, 1, 2, 4, 0])
make_dot(B_t_true[1], labels=[f'x{i}(1)' for i in range(5)])
```

```
# B(2,2)
B_t_true[2] = np.array([[0.0, 0.0, 0.0, 0.0, 0.0],
                        [0.0, 0.0,-0.7, 0.0, 0.5],
                        [0.2, 0.0, 0.0, 0.0, 0.0],
                        [0.0, 0.0,-0.4, 0.0, 0.0],
                        [0.3, 0.0, 0.0, 0.0, 0.0]])
causal_orders.append([0, 2, 4, 3, 1])
make_dot(B_t_true[2], labels=[f'x{i}(2)' for i in range(5)])
```

```
# create B(t,t-τ) and X
for t in range(n_timepoints):
    # external influence
    expon = 0.1
    ext = np.empty((n_features, n_samples))
    for i in range(n_features):
        ext[i, :] = np.random.normal(size=(1, n_samples));
        ext[i, :] = np.multiply(np.sign(ext[i, :]), abs(ext[i, :]) ** expon);
        ext[i, :] = ext[i, :] - np.mean(ext[i, :]);
        ext[i, :] = ext[i, :] / np.std(ext[i, :]);

    # create B(t,t-τ)
    for tau in range(n_lags):
        value = np.random.uniform(low=0.01, high=0.5, size=(n_features, n_features))
        sign = np.random.choice([-1, 1], size=(n_features, n_features))
        B_tau_true[t, tau] = np.multiply(value, sign)

    # create X(t)
    X = np.zeros((n_features, n_samples))
    for co in causal_orders[t]:
        X[co] = np.dot(B_t_true[t][co, :], X) + ext[co]
        if t > 0:
            for tau in range(n_lags):
                X[co] = X[co] + np.dot(B_tau_true[t, tau][co, :], X_t[t-(tau+1)].T)

    X_t[t] = X.T
```

2.9.4 Causal Discovery

To run causal discovery, we create a *LongitudinalLiNGAM* object by specifying the `n_lags` parameter. Then, we call the *fit()* method.

```
model = lingam.LongitudinalLiNGAM(n_lags=n_lags)
model = model.fit(X_t)
```

Using the *causal_orders_* property, we can see the causal ordering in time-points as a result of the causal discovery. All elements are nan because the causal order of B(t,t) at t=0 is not calculated. So access to the time points above t=1.

```
print(model.causal_orders_[0]) # nan at t=0
print(model.causal_orders_[1])
print(model.causal_orders_[2])
```

```
[nan, nan, nan, nan, nan]
[3, 1, 2, 4, 0]
[0, 4, 2, 3, 1]
```

Also, using the *adjacency_matrices_* property, we can see the adjacency matrix as a result of the causal discovery. As with the causal order, all elements are nan because the B(t,t) and B(t,t-τ) at t=0 is not calculated. So access to the time points above t=1. Also, if we run causal discovery with n_lags=2, B(t,t-τ) at t=1 is also not computed, so all the elements are nan.

```
t = 0 # nan at t=0
print('B(0,0):')
print(model.adjacency_matrices_[t, 0])
print('B(0,-1):')
print(model.adjacency_matrices_[t, 1])

t = 1
print('B(1,1):')
print(model.adjacency_matrices_[t, 0])
print('B(1,0):')
print(model.adjacency_matrices_[t, 1])

t = 2
print('B(2,2):')
print(model.adjacency_matrices_[t, 0])
print('B(2,1):')
print(model.adjacency_matrices_[t, 1])
```

```
B(0,0):
[[nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]]
B(0,-1):
[[nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]]
B(1,1):
[[ 0.     0.099  0.     0.    -0.52 ]
 [ 0.     0.     0.     0.398  0.   ]
 [ 0.     0.384  0.    -0.162  0.   ]
 [ 0.     0.     0.     0.     0.   ]
 [ 0.    -0.249 -0.074  0.     0.   ]]
```

(continues on next page)

(continued from previous page)

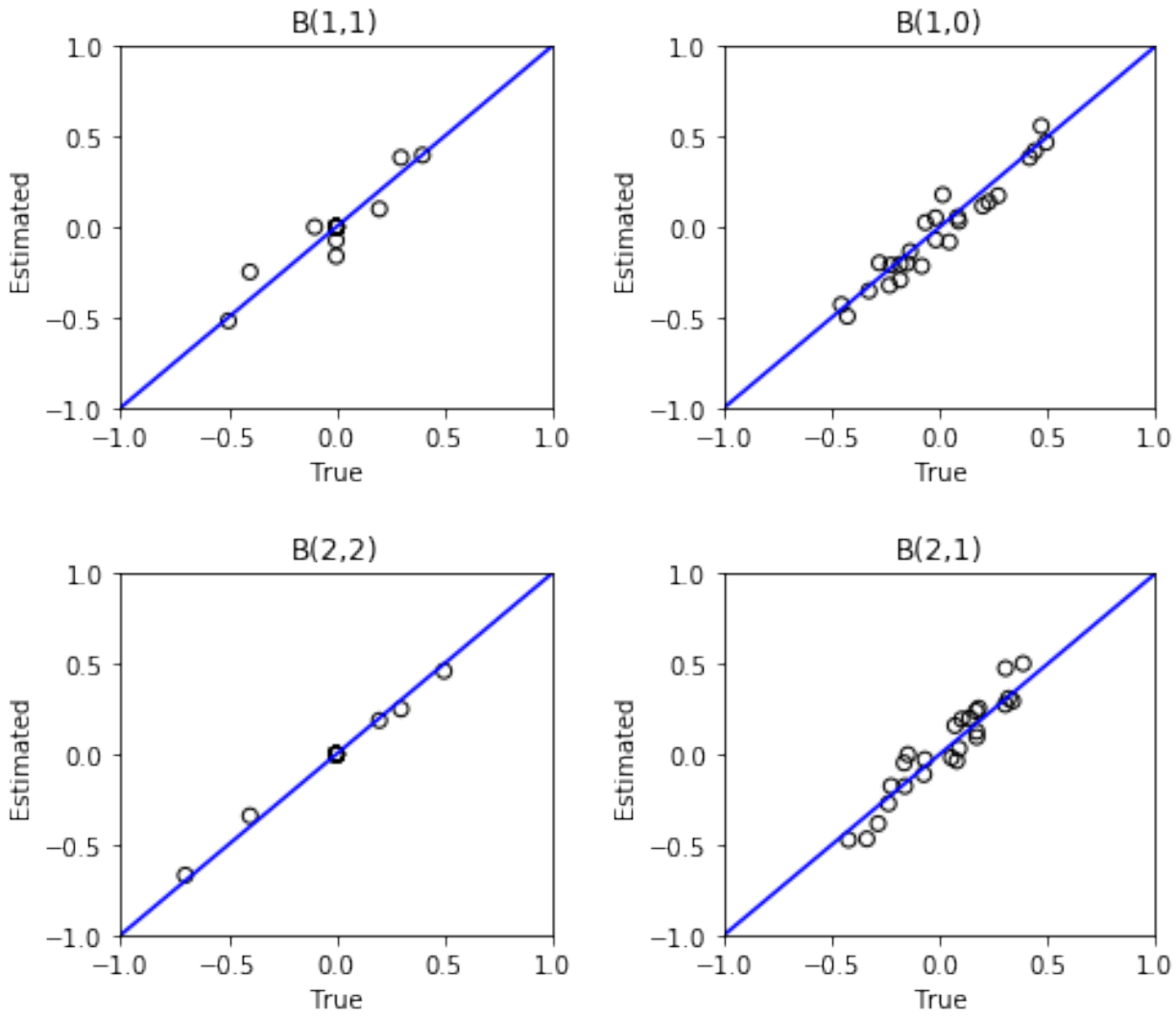
```
B(1,0):
[[ 0.025  0.116 -0.202  0.054 -0.216]
 [ 0.139 -0.211 -0.43   0.558  0.051]
 [-0.135  0.178  0.421  0.173  0.031]
 [ 0.384 -0.083 -0.495 -0.072 -0.323]
 [-0.206 -0.354 -0.199 -0.293  0.468]]
B(2,2):
[[ 0.     0.     0.     0.     0.   ]
 [ 0.     0.    -0.67   0.     0.46 ]
 [ 0.187  0.     0.     0.     0.   ]
 [ 0.     0.    -0.341  0.     0.   ]
 [ 0.25   0.     0.     0.     0.   ]]
B(2,1):
[[ 0.194  0.2    0.031 -0.473 -0.002]
 [-0.384 -0.037  0.158  0.255  0.095]
 [ 0.126  0.275 -0.048  0.502 -0.019]
 [ 0.238 -0.469  0.475 -0.029 -0.176]
 [-0.177  0.309 -0.112  0.295 -0.273]]
```

```
for t in range(1, n_timepoints):
    B_t, B_tau = model.adjacency_matrices_[t]
    plt.figure(figsize=(7, 3))

    plt.subplot(1,2,1)
    plt.plot([-1, 1],[-1, 1], marker="", color="blue", label="support")
    plt.scatter(B_t_true[t], B_t, facecolors='none', edgecolors='black')
    plt.xlim(-1, 1)
    plt.ylim(-1, 1)
    plt.xlabel('True')
    plt.ylabel('Estimated')
    plt.title(f'B({t},{t})')

    plt.subplot(1,2,2)
    plt.plot([-1, 1],[-1, 1], marker="", color="blue", label="support")
    plt.scatter(B_tau_true[t], B_tau, facecolors='none', edgecolors='black')
    plt.xlim(-1, 1)
    plt.ylim(-1, 1)
    plt.xlabel('True')
    plt.ylabel('Estimated')
    plt.title(f'B({t},{t-1})')

    plt.tight_layout()
    plt.show()
```



2.9.5 Independence between error variables

To check if the LiNGAM assumption is broken, we can get p-values of independence between error variables. The value in the i-th row and j-th column of the obtained matrix shows the p-value of the independence of the error variables e_i and e_j.

```
p_values_list = model.get_error_independence_p_values()
```

```
t = 1
print(p_values_list[t])
```

```
[[0.    0.167 0.107 0.534 0.313]
 [0.167 0.    0.195 0.821 0.204]
 [0.107 0.195 0.    0.005 0.105]
 [0.534 0.821 0.005 0.    0.049]
 [0.313 0.204 0.105 0.049 0.   ]]
```

```
t = 2
print(p_values_list[2])
```

```
[[0.    0.723 0.596 0.579 0.564]
 [0.723 0.    0.612 0.688 0.412]
 [0.596 0.612 0.    0.267 0.636]
 [0.579 0.688 0.267 0.    0.421]
 [0.564 0.412 0.636 0.421 0.   ]]
```

2.9.6 Bootstrapping

We call *bootstrap()* method instead of *fit()*. Here, the second argument specifies the number of bootstrap sampling.

```
model = lingam.LongitudinalLiNGAM()
result = model.bootstrap(X_t, n_sampling=100)
```

2.9.7 Causal Directions

Since *LongitudinalBootstrapResult* object is returned, we can get the ranking of the causal directions extracted by *get_causal_direction_counts()* method. In the following sample code, `n_directions` option is limited to the causal directions of the top 8 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
cdc_list = result.get_causal_direction_counts(n_directions=12, min_causal_effect=0.01,
↪ split_by_causal_effect_sign=True)
```

```
t = 1
labels = [f'x{i}({u})' for u in [t, t-1] for i in range(5)]
print_causal_directions(cdc_list[t], 100, labels=labels)
```

```
x4(1) <--- x4(0) (b>0) (100.0%)
x2(1) <--- x0(0) (b<0) (100.0%)
x3(1) <--- x0(0) (b>0) (100.0%)
x1(1) <--- x3(0) (b>0) (100.0%)
x1(1) <--- x2(0) (b<0) (100.0%)
x3(1) <--- x2(0) (b<0) (100.0%)
x3(1) <--- x4(0) (b<0) (100.0%)
x1(1) <--- x3(1) (b>0) (100.0%)
x0(1) <--- x4(1) (b<0) (100.0%)
x4(1) <--- x1(0) (b<0) (100.0%)
x4(1) <--- x1(1) (b<0) (100.0%)
x2(1) <--- x2(0) (b>0) (100.0%)
```

```
t = 2
labels = [f'x{i}({u})' for u in [t, t-1] for i in range(5)]
print_causal_directions(cdc_list[t], 100, labels=labels)
```

```
x0(2) <--- x0(1) (b>0) (100.0%)
x4(2) <--- x1(1) (b>0) (100.0%)
x3(2) <--- x2(1) (b>0) (100.0%)
```

(continues on next page)

(continued from previous page)

```
x3(2) <--- x1(1) (b<0) (100.0%)
x3(2) <--- x0(1) (b>0) (100.0%)
x3(2) <--- x2(2) (b<0) (100.0%)
x2(2) <--- x3(1) (b>0) (100.0%)
x2(2) <--- x1(1) (b>0) (100.0%)
x4(2) <--- x3(1) (b>0) (100.0%)
x1(2) <--- x3(1) (b>0) (100.0%)
x1(2) <--- x2(1) (b>0) (100.0%)
x1(2) <--- x0(1) (b<0) (100.0%)
```

2.9.8 Directed Acyclic Graphs

Also, using the *get_directed_acyclic_graph_counts()* method, we can get the ranking of the DAGs extracted. In the following sample code, `n_dags` option is limited to the dags of the top 3 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
dagc_list = result.get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.01,
↪ split_by_causal_effect_sign=True)
```

```
t = 1
labels = [f'x{i}({u})' for u in [t, t-1] for i in range(5)]
print_dagc(dagc_list[t], 100, labels=labels)
```

```
DAG[0]: 2.0%
    x0(1) <--- x4(1) (b<0)
    x0(1) <--- x0(0) (b>0)
    x0(1) <--- x1(0) (b>0)
    x0(1) <--- x2(0) (b<0)
    x0(1) <--- x3(0) (b>0)
    x0(1) <--- x4(0) (b<0)
    x1(1) <--- x3(1) (b>0)
    x1(1) <--- x0(0) (b>0)
    x1(1) <--- x1(0) (b<0)
    x1(1) <--- x2(0) (b<0)
    x1(1) <--- x3(0) (b>0)
    x1(1) <--- x4(0) (b>0)
    x2(1) <--- x1(1) (b>0)
    x2(1) <--- x0(0) (b<0)
    x2(1) <--- x1(0) (b>0)
    x2(1) <--- x2(0) (b>0)
    x2(1) <--- x3(0) (b>0)
    x2(1) <--- x4(0) (b>0)
    x3(1) <--- x0(0) (b>0)
    x3(1) <--- x1(0) (b<0)
    x3(1) <--- x2(0) (b<0)
    x3(1) <--- x4(0) (b<0)
    x4(1) <--- x1(1) (b<0)
    x4(1) <--- x0(0) (b<0)
    x4(1) <--- x1(0) (b<0)
    x4(1) <--- x2(0) (b<0)
    x4(1) <--- x3(0) (b<0)
    x4(1) <--- x4(0) (b>0)
DAG[1]: 1.0%
    x0(1) <--- x2(1) (b<0)
```

(continues on next page)

(continued from previous page)

```
    x0(1) <--- x4(1) (b<0)
    x0(1) <--- x0(0) (b>0)
    x0(1) <--- x1(0) (b<0)
    x0(1) <--- x2(0) (b<0)
    x0(1) <--- x3(0) (b>0)
    x0(1) <--- x4(0) (b<0)
    x1(1) <--- x3(1) (b>0)
    x1(1) <--- x0(0) (b>0)
    x1(1) <--- x1(0) (b<0)
    x1(1) <--- x2(0) (b<0)
    x1(1) <--- x3(0) (b>0)
    x1(1) <--- x4(0) (b>0)
    x2(1) <--- x1(1) (b>0)
    x2(1) <--- x0(0) (b<0)
    x2(1) <--- x2(0) (b>0)
    x2(1) <--- x3(0) (b>0)
    x2(1) <--- x4(0) (b>0)
    x3(1) <--- x0(0) (b>0)
    x3(1) <--- x1(0) (b>0)
    x3(1) <--- x2(0) (b<0)
    x3(1) <--- x3(0) (b<0)
    x3(1) <--- x4(0) (b<0)
    x4(1) <--- x1(1) (b<0)
    x4(1) <--- x2(1) (b<0)
    x4(1) <--- x3(1) (b>0)
    x4(1) <--- x0(0) (b<0)
    x4(1) <--- x1(0) (b<0)
    x4(1) <--- x2(0) (b>0)
    x4(1) <--- x3(0) (b>0)
    x4(1) <--- x4(0) (b>0)
DAG[2]: 1.0%
    x0(1) <--- x1(1) (b>0)
    x0(1) <--- x4(1) (b<0)
    x0(1) <--- x1(0) (b>0)
    x0(1) <--- x2(0) (b<0)
    x0(1) <--- x3(0) (b>0)
    x0(1) <--- x4(0) (b<0)
    x1(1) <--- x3(1) (b>0)
    x1(1) <--- x0(0) (b>0)
    x1(1) <--- x1(0) (b<0)
    x1(1) <--- x2(0) (b<0)
    x1(1) <--- x3(0) (b>0)
    x1(1) <--- x4(0) (b>0)
    x2(1) <--- x1(1) (b>0)
    x2(1) <--- x0(0) (b<0)
    x2(1) <--- x1(0) (b>0)
    x2(1) <--- x2(0) (b>0)
    x2(1) <--- x3(0) (b>0)
    x2(1) <--- x4(0) (b>0)
    x3(1) <--- x0(0) (b>0)
    x3(1) <--- x1(0) (b<0)
    x3(1) <--- x2(0) (b<0)
    x3(1) <--- x3(0) (b<0)
    x3(1) <--- x4(0) (b<0)
    x4(1) <--- x1(1) (b<0)
    x4(1) <--- x2(1) (b<0)
    x4(1) <--- x3(1) (b>0)
```

(continues on next page)

(continued from previous page)

```
    x4(1) <--- x0(0) (b<0)
    x4(1) <--- x1(0) (b<0)
    x4(1) <--- x2(0) (b<0)
    x4(1) <--- x3(0) (b<0)
    x4(1) <--- x4(0) (b>0)
```

```
t = 2
labels = [f'x{i}({u})' for u in [t, t-1] for i in range(5)]
print_dagc(dagc_list[t], 100, labels=labels)
```

```
DAG[0]: 3.0%
    x0(2) <--- x0(1) (b>0)
    x0(2) <--- x1(1) (b>0)
    x0(2) <--- x2(1) (b>0)
    x0(2) <--- x3(1) (b<0)
    x0(2) <--- x4(1) (b>0)
    x1(2) <--- x2(2) (b<0)
    x1(2) <--- x4(2) (b>0)
    x1(2) <--- x0(1) (b<0)
    x1(2) <--- x1(1) (b<0)
    x1(2) <--- x2(1) (b>0)
    x1(2) <--- x3(1) (b>0)
    x1(2) <--- x4(1) (b>0)
    x2(2) <--- x0(2) (b>0)
    x2(2) <--- x0(1) (b>0)
    x2(2) <--- x1(1) (b>0)
    x2(2) <--- x2(1) (b<0)
    x2(2) <--- x3(1) (b>0)
    x2(2) <--- x4(1) (b<0)
    x3(2) <--- x2(2) (b<0)
    x3(2) <--- x0(1) (b>0)
    x3(2) <--- x1(1) (b<0)
    x3(2) <--- x2(1) (b>0)
    x3(2) <--- x3(1) (b>0)
    x3(2) <--- x4(1) (b<0)
    x4(2) <--- x0(2) (b>0)
    x4(2) <--- x0(1) (b<0)
    x4(2) <--- x1(1) (b>0)
    x4(2) <--- x2(1) (b<0)
    x4(2) <--- x3(1) (b>0)
    x4(2) <--- x4(1) (b<0)
DAG[1]: 2.0%
    x0(2) <--- x0(1) (b>0)
    x0(2) <--- x1(1) (b>0)
    x0(2) <--- x2(1) (b>0)
    x0(2) <--- x3(1) (b<0)
    x0(2) <--- x4(1) (b>0)
    x1(2) <--- x2(2) (b<0)
    x1(2) <--- x4(2) (b>0)
    x1(2) <--- x0(1) (b<0)
    x1(2) <--- x1(1) (b<0)
    x1(2) <--- x2(1) (b>0)
    x1(2) <--- x3(1) (b>0)
    x1(2) <--- x4(1) (b<0)
    x2(2) <--- x0(2) (b>0)
    x2(2) <--- x0(1) (b>0)
```

(continues on next page)

(continued from previous page)

```
    x2(2) <--- x1(1) (b>0)
    x2(2) <--- x2(1) (b<0)
    x2(2) <--- x3(1) (b>0)
    x2(2) <--- x4(1) (b>0)
    x3(2) <--- x2(2) (b<0)
    x3(2) <--- x0(1) (b>0)
    x3(2) <--- x1(1) (b<0)
    x3(2) <--- x2(1) (b>0)
    x3(2) <--- x3(1) (b<0)
    x3(2) <--- x4(1) (b<0)
    x4(2) <--- x0(2) (b>0)
    x4(2) <--- x0(1) (b<0)
    x4(2) <--- x1(1) (b>0)
    x4(2) <--- x2(1) (b<0)
    x4(2) <--- x3(1) (b>0)
    x4(2) <--- x4(1) (b<0)
DAG[2]: 2.0%
    x0(2) <--- x0(1) (b>0)
    x0(2) <--- x1(1) (b>0)
    x0(2) <--- x2(1) (b<0)
    x0(2) <--- x3(1) (b<0)
    x0(2) <--- x4(1) (b<0)
    x1(2) <--- x2(2) (b<0)
    x1(2) <--- x4(2) (b>0)
    x1(2) <--- x0(1) (b<0)
    x1(2) <--- x1(1) (b<0)
    x1(2) <--- x2(1) (b>0)
    x1(2) <--- x3(1) (b>0)
    x1(2) <--- x4(1) (b>0)
    x2(2) <--- x0(1) (b>0)
    x2(2) <--- x1(1) (b>0)
    x2(2) <--- x2(1) (b<0)
    x2(2) <--- x3(1) (b>0)
    x2(2) <--- x4(1) (b<0)
    x3(2) <--- x2(2) (b<0)
    x3(2) <--- x0(1) (b>0)
    x3(2) <--- x1(1) (b<0)
    x3(2) <--- x2(1) (b>0)
    x3(2) <--- x3(1) (b<0)
    x3(2) <--- x4(1) (b<0)
    x4(2) <--- x0(2) (b>0)
    x4(2) <--- x0(1) (b<0)
    x4(2) <--- x1(1) (b>0)
    x4(2) <--- x2(1) (b<0)
    x4(2) <--- x3(1) (b>0)
    x4(2) <--- x4(1) (b<0)
```

2.9.9 Probability

Using the *get_probabilities()* method, we can get the probability of bootstrapping.

```
probs = result.get_probabilities(min_causal_effect=0.01)
print(probs[1])
```

```
[[[0.   0.51 0.09 0.15 1.  ]
  [0.   0.   0.   1.   0.  ]
  [0.02 0.99 0.   0.52 0.3 ]
  [0.   0.   0.   0.   0.  ]
  [0.   1.   0.23 0.3  0.  ]]

 [[0.92 0.97 1.   0.94 0.99]
  [0.99 0.99 1.   1.   0.94]
  [1.   0.97 1.   0.99 0.87]
  [1.   0.98 1.   0.92 1.  ]
  [1.   1.   1.   1.   1.  ]]]
```

```
t = 1
print('B(1,1):')
print(probs[t, 0])
print('B(1,0):')
print(probs[t, 1])

t = 2
print('B(2,2):')
print(probs[t, 0])
print('B(2,1):')
print(probs[t, 1])
```

```
B(1,1):
[[0.   0.51 0.09 0.15 1.  ]
 [0.   0.   0.   1.   0.  ]
 [0.02 0.99 0.   0.52 0.3 ]
 [0.   0.   0.   0.   0.  ]
 [0.   1.   0.23 0.3  0.  ]]
B(1,0):
[[0.92 0.97 1.   0.94 0.99]
 [0.99 0.99 1.   1.   0.94]
 [1.   0.97 1.   0.99 0.87]
 [1.   0.98 1.   0.92 1.  ]
 [1.   1.   1.   1.   1.  ]]
B(2,2):
[[0.   0.   0.   0.   0.  ]
 [0.1  0.   1.   0.06 1.  ]
 [0.78 0.   0.   0.   0.13]
 [0.13 0.   1.   0.   0.16]
 [0.88 0.   0.   0.   0.  ]]
B(2,1):
[[1.   1.   0.91 1.   0.92]
 [1.   0.86 1.   1.   0.95]
 [0.95 1.   0.96 1.   0.8 ]
 [1.   1.   1.   0.92 1.  ]
 [0.99 1.   0.96 1.   1.  ]]
```

2.9.10 Total Causal Effects

Using the `get_total_causal_effects()` method, we can get the list of total causal effect. The total causal effects we can get are dictionary type variable. We can display the list nicely by assigning it to pandas.DataFrame. Also, we have replaced the variable index with a label below.

```
causal_effects = result.get_total_causal_effects(min_causal_effect=0.01)

df = pd.DataFrame(causal_effects)

labels = [f'x{i}({t})' for t in range(3) for i in range(5)]
df['from'] = df['from'].apply(lambda x : labels[x])
df['to'] = df['to'].apply(lambda x : labels[x])
df
```

We can easily perform sorting operations with pandas.DataFrame.

```
df.sort_values('effect', ascending=False).head()
```

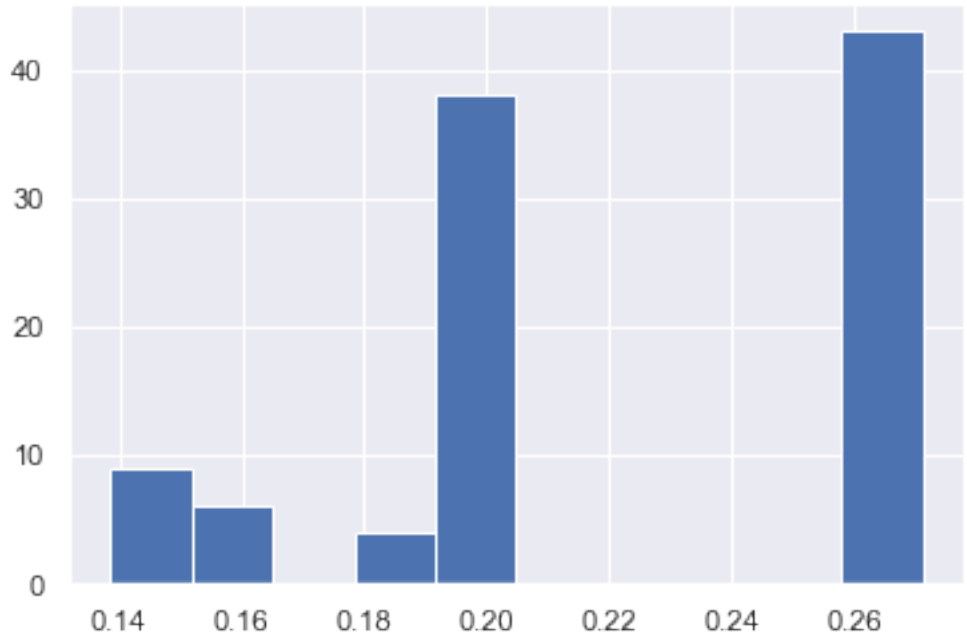
And with pandas.DataFrame, we can easily filter by keywords. The following code extracts the causal direction towards x0(2).

```
df[df['to']=='x0(2)'].head()
```

Because it holds the raw data of the total causal effect (the original data for calculating the median), it is possible to draw a histogram of the values of the causal effect, as shown below.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline

from_index = 5 # index of x0(1). (index:0)+(n_features:5)*(timepoint:1) = 5
to_index = 12 # index of x2(2). (index:2)+(n_features:5)*(timepoint:2) = 12
plt.hist(result.total_effects_[:, to_index, from_index])
```



2.10 BottomUpParceLiNGAM

2.10.1 Model

This method assumes an extension of the basic LiNGAM model[1] to hidden common cause cases. Specifically, this implements Algorithm 1 of[3] except the Step 2. Similarly to the basic LiNGAM model[1], this method makes the following assumptions:

1. Linearity
2. Non-Gaussian continuous error variables (except at most one)
3. Acyclicity

However, it allows the following hidden common causes:

1. Only exogenous observed variables may share hidden common causes.

This is a simpler version of the latent variable LiNGAM[2] that extends the basic LiNGAM model to hidden common causes. Note that the latent variable LiNGAM[2] allows the existence of hidden common causes between any observed variables. However, this kind of causal graph structures are often assumed in the classic structural equation modelling[4].

References

2.10.2 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import print_causal_directions, print_dagc, make_dot

import warnings
warnings.filterwarnings('ignore')

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.4']
```

[1] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. J. Kerminen. A linear non-gaussian acyclic model for causal discovery. Journal of Machine Learning Research, 7:2003-2030, 2006.

[3] T. Tashiro, S. Shimizu, A. Hyvärinen, T. Washio. ParceLiNGAM: a causal ordering method robust against latent confounders. Neural computation, 26(1): 57-83, 2014.

[2] P. O. Hoyer, S. Shimizu, A. Kerminen, and M. Palviainen. Estimation of causal effects using linear non-gaussian causal models with hidden variables. International Journal of Approximate Reasoning, 49(2): 362-378, 2008.

[4] K. A. Bollen. Structural Equations With Latent Variables, 1984, Wiley.

2.10.3 Test data

First, we generate a causal structure with 7 variables. Then we create a dataset with 6 variables from x0 to x5, with x6 being the latent variable for x2 and x3.

```
np.random.seed(1000)

x6 = np.random.uniform(size=1000)
x3 = 2.0*x6 + np.random.uniform(size=1000)
x0 = 0.5*x3 + np.random.uniform(size=1000)
x2 = 2.0*x6 + np.random.uniform(size=1000)
x1 = 0.5*x0 + 0.5*x2 + np.random.uniform(size=1000)
x5 = 0.5*x0 + np.random.uniform(size=1000)
x4 = 0.5*x0 - 0.5*x2 + np.random.uniform(size=1000)

# The latent variable x6 is not included.
X = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T, columns=['x0', 'x1', 'x2', 'x3
↪', 'x4', 'x5'])

X.head()
```

```
m = np.array([[0.0, 0.0, 0.0, 0.5, 0.0, 0.0, 0.0],
              [0.5, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2.0],
              [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2.0],
              [0.5, 0.0,-0.5, 0.0, 0.0, 0.0, 0.0],
              [0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]])

dot = make_dot(m)

# Save pdf
dot.render('dag')

# Save png
dot.format = 'png'
dot.render('dag')

dot
```

2.10.4 Causal Discovery

To run causal discovery, we create a `BottomUpParceLiNGAM` object and call the `fit` method.

```
model = lingam.BottomUpParceLiNGAM()
model.fit(X)
```

```
<lingam.bottom_up_parce_lingam.BottomUpParceLiNGAM at 0x2098ee24860>
```

Using the `causal_order_` properties, we can see the causal ordering as a result of the causal discovery. x2 and x3, which have latent confounders as parents, are stored in a list without causal ordering.

```
model.causal_order_
```

```
[[2, 3], 0, 5, 1, 4]
```

Also, using the `adjacency_matrix_` properties, we can see the adjacency matrix as a result of the causal discovery. The coefficients between variables with latent confounders are np.nan.

```
model.adjacency_matrix_
```

```
array([[ 0.   ,  0.   ,  0.   ,  0.506,  0.   ,  0.   ],
       [ 0.499,  0.   ,  0.495,  0.007,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,    nan,  0.   ,  0.   ],
       [ 0.   ,  0.   ,    nan,  0.   ,  0.   ,  0.   ],
       [ 0.448,  0.   , -0.451,  0.   ,  0.   ,  0.   ],
       [ 0.48 ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ]])
```

We can draw a causal graph by utility funciton.

```
make_dot(model.adjacency_matrix_)
```

2.10.5 Independence between error variables

To check if the LiNGAM assumption is broken, we can get p-values of independence between error variables. The value in the i-th row and j-th column of the obtained matrix shows the p-value of the independence of the error variables e_i and e_j.

```
p_values = model.get_error_independence_p_values(X)
print(p_values)
```

```
[[0.    0.491   nan   nan 0.763 0.2  ]
 [0.491 0.      nan   nan 0.473 0.684]
 [  nan   nan 0.      nan   nan   nan]
 [  nan   nan   nan 0.      nan   nan]
 [0.763 0.473   nan   nan 0.    0.427]
 [0.2   0.684   nan   nan 0.427 0.   ]]
```

2.10.6 Bootstrapping

We call `bootstrap()` method instead of `fit()`. Here, the second argument specifies the number of bootstrap sampling.

```
import warnings
warnings.filterwarnings('ignore', category=UserWarning)

model = lingam.BottomUpParceLiNGAM()
result = model.bootstrap(X, n_sampling=100)
```

2.10.7 Causal Directions

Since `BootstrapResult` object is returned, we can get the ranking of the causal directions extracted by `get_causal_direction_counts()` method. In the following sample code, `n_directions` option is limited

to the causal directions of the top 8 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
cdc = result.get_causal_direction_counts(n_directions=8, min_causal_effect=0.01,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_causal_directions(cdc, 100)
```

```
x4 <--- x0 (b>0) (45.0%)
x4 <--- x2 (b<0) (45.0%)
x1 <--- x0 (b>0) (41.0%)
x1 <--- x2 (b>0) (41.0%)
x5 <--- x0 (b>0) (26.0%)
x1 <--- x3 (b>0) (21.0%)
x0 <--- x3 (b>0) (12.0%)
x5 <--- x2 (b>0) (7.0%)
```

2.10.8 Directed Acyclic Graphs

Also, using the `get_directed_acyclic_graph_counts()` method, we can get the ranking of the DAGs extracted. In the following sample code, `n_dags` option is limited to the dags of the top 3 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
dagc = result.get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.01,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_dagc(dagc, 100)
```

```
DAG[0]: 33.0%
DAG[1]: 13.0%
    x4 <--- x0 (b>0)
    x4 <--- x2 (b<0)
DAG[2]: 7.0%
    x1 <--- x0 (b>0)
    x1 <--- x2 (b>0)
```

2.10.9 Probability

Using the `get_probabilities()` method, we can get the probability of bootstrapping.

```
prob = result.get_probabilities(min_causal_effect=0.01)
print(prob)
```

```
[[0.   0.01 0.   0.12 0.01 0.  ]
 [0.41 0.   0.41 0.21 0.   0.  ]
 [0.   0.   0.   0.02 0.   0.  ]
 [0.   0.   0.   0.   0.   0.  ]
 [0.45 0.03 0.45 0.02 0.   0.07]
 [0.26 0.01 0.07 0.02 0.   0.  ]]
```

2.10.10 Total Causal Effects

Using the `get_total_causal_effects()` method, we can get the list of total causal effect. The total causal effects we can get are dictionary type variable. We can display the list nicely by assigning it to pandas.DataFrame. Also, we have replaced the variable index with a label below.

```
causal_effects = result.get_total_causal_effects(min_causal_effect=0.01)

# Assign to pandas.DataFrame for pretty display
df = pd.DataFrame(causal_effects)
labels = [f'x{i}' for i in range(X.shape[1])]
df['from'] = df['from'].apply(lambda x : labels[x])
df['to'] = df['to'].apply(lambda x : labels[x])
df
```

We can easily perform sorting operations with pandas.DataFrame.

```
df.sort_values('effect', ascending=False).head()
```

```
df.sort_values('probability', ascending=True).head()
```

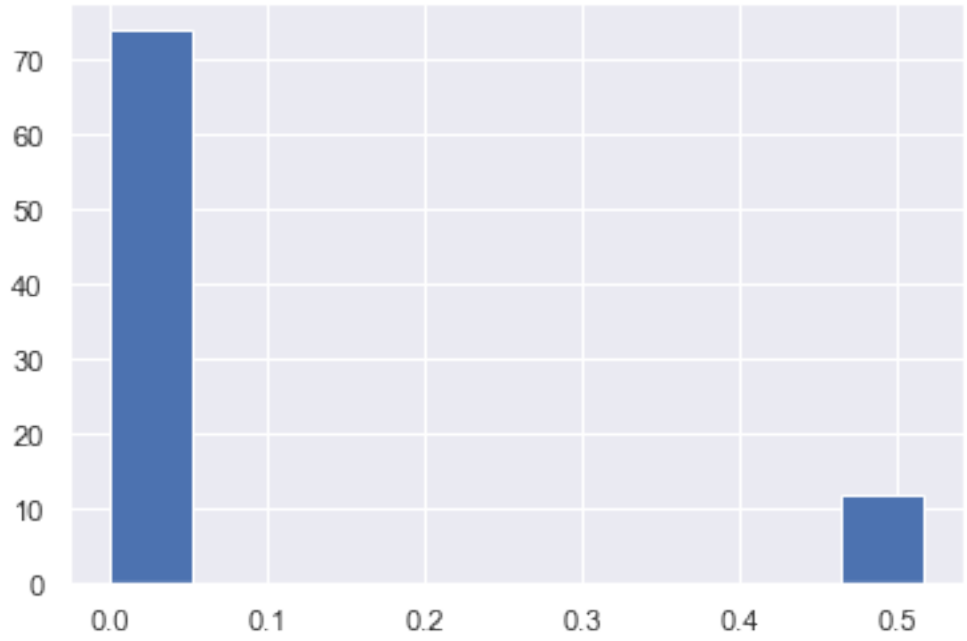
And with pandas.DataFrame, we can easily filter by keywords. The following code extracts the causal direction towards x1.

```
df[df['to']=='x1'].head()
```

Because it holds the raw data of the total causal effect (the original data for calculating the median), it is possible to draw a histogram of the values of the causal effect, as shown below.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline

from_index = 0 # index of x0
to_index = 5 # index of x5
plt.hist(result.total_effects_[:, to_index, from_index])
```



2.10.11 Bootstrap Probability of Path

Using the `get_paths()` method, we can explore all paths from any variable to any variable and calculate the bootstrap probability for each path. The path will be output as an array of variable indices. For example, the array `[3, 0, 1]` shows the path from variable X3 through variable X0 to variable X1.

```
from_index = 3 # index of x3
to_index = 1 # index of x0

pd.DataFrame(result.get_paths(from_index, to_index))
```

2.11 How to use prior knowledge in BottomUpParceLiNGAM

2.11.1 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import make_prior_knowledge, make_dot

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
np.random.seed(0)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.2']
```

2.11.2 Utility function

We define a utility function to draw the directed acyclic graph.

```
def make_prior_knowledge_graph(prior_knowledge_matrix):
    d = graphviz.Digraph(engine='dot')

    labels = [f'x{i}' for i in range(prior_knowledge_matrix.shape[0])]
    for label in labels:
        d.node(label, label)

    dirs = np.where(prior_knowledge_matrix > 0)
    for to, from_ in zip(dirs[0], dirs[1]):
        d.edge(labels[from_], labels[to])

    dirs = np.where(prior_knowledge_matrix < 0)
    for to, from_ in zip(dirs[0], dirs[1]):
        if to != from_:
            d.edge(labels[from_], labels[to], style='dashed')
    return d
```

2.11.3 Test data

We create test data consisting of 6 variables.

```
np.random.seed(1000)

x6 = np.random.uniform(size=1000)
x3 = 2.0*x6 + np.random.uniform(size=1000)
x0 = 0.5*x3 + np.random.uniform(size=1000)
x2 = 2.0*x6 + np.random.uniform(size=1000)
x1 = 0.5*x0 + 0.5*x2 + np.random.uniform(size=1000)
x5 = 0.5*x0 + np.random.uniform(size=1000)
x4 = 0.5*x0 - 0.5*x2 + np.random.uniform(size=1000)

# The latent variable x6 is not included.
X = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T, columns=['x0', 'x1', 'x2', 'x3
↪', 'x4', 'x5'])
```

```
m = np.array([[0.0, 0.0,    0.0,    0.5, 0.0, 0.0],
              [0.5, 0.0,    0.5,    0.0, 0.0, 0.0],
              [0.0, 0.0,    0.0, np.nan, 0.0, 0.0],
              [0.0, 0.0, np.nan,    0.0, 0.0, 0.0],
              [0.5, 0.0,   -0.5,    0.0, 0.0, 0.0],
              [0.5, 0.0,    0.0,    0.0, 0.0, 0.0]])

make_dot(m)
```

2.11.4 Make Prior Knowledge Matrix

We create prior knowledge so that x0, x1 and x4 are sink variables.

The elements of prior knowledge matrix are defined as follows: * `0` : :math:`x_i` does not have a directed path to :math:`x_j` * `1` : :math:`x_i` has a directed path to :math:`x_j` * `-1` : No prior knowledge is available to know if either of the two cases above (0 or 1) is true.

```
prior_knowledge = make_prior_knowledge(
    n_variables=6,
    sink_variables=[0, 1, 4],
)
print(prior_knowledge)
```

```
[[-1  0 -1 -1  0 -1]
 [ 0 -1 -1 -1  0 -1]
 [ 0  0 -1 -1  0 -1]
 [ 0  0 -1 -1  0 -1]
 [ 0  0 -1 -1 -1 -1]
 [ 0  0 -1 -1  0 -1]]
```

```
# Draw a graph of prior knowledge
make_prior_knowledge_graph(prior_knowledge)
```

2.11.5 Causal Discovery

To run causal discovery using prior knowledge, we create a `DirectLiNGAM` object with the prior knowledge matrix.

```
model = lingam.BottomUpParceLiNGAM(prior_knowledge=prior_knowledge)
model.fit(X)
print(model.causal_order_)
print(model.adjacency_matrix_)
```

```
[[0, 2, 3, 5], 4, 1]
[[ 0.     0.       nan    nan  0.       nan]
 [ 0.     0.     0.479  0.219  0.     0.212]
 [  nan  0.     0.       nan  0.       nan]
 [  nan  0.       nan  0.     0.       nan]
 [ 0.     0.    -0.494  0.212  0.     0.199]
 [  nan  0.       nan    nan  0.     0.   ]]
```

We can see that x0, x1, and x4 are output as sink variables, as specified in the prior knowledge.

```
make_dot(model.adjacency_matrix_)
```

Next, let's specify the prior knowledge so that x0 is an exogenous variable.

```
prior_knowledge = make_prior_knowledge(
    n_variables=6,
    exogenous_variables=[0],
)
```

(continues on next page)

(continued from previous page)

```
model = lingam.BottomUpParceLiNGAM(prior_knowledge=prior_knowledge)
model.fit(X)

make_dot(model.adjacency_matrix_)
```

2.12 RCD

2.12.1 Model

This method RCD (Repetitive Causal Discovery) assumes an extension of the basic LiNGAM model[1] to hidden common cause cases, i.e., the latent variable LiNGAM model[2]. Similarly to the basic LiNGAM model[1], this method makes the following assumptions:

1. Linearity
2. Non-Gaussian continuous error variables
3. Acyclicity

However, RCD allows the existence of hidden common causes. It outputs a causal graph where a bi-directed arc indicates the pair of variables that have the same hidden common causes, and a directed arrow indicates the causal direction of a pair of variables that are not affected by the same hidden common causes.

References

2.12.2 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import print_causal_directions, print_dagc, make_dot

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.5.4']
```

2.12.3 Test data

First, we generate a causal structure with 7 variables. Then we create a dataset with 5 variables from x0 to x4, with x5 and x6 being the latent variables.

[1] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. J. Kerminen. A linear non-gaussian acyclic model for causal discovery. Journal of Machine Learning Research, 7:2003-2030, 2006.

[2] P. O. Hoyer, S. Shimizu, A. Kerminen, and M. Palviainen. Estimation of causal effects using linear non-gaussian causal models with hidden variables. International Journal of Approximate Reasoning, 49(2): 362-378, 2008.

```
np.random.seed(0)

get_external_effect = lambda n: np.random.normal(0.0, 0.5, n) ** 3
n_samples = 300

x5 = get_external_effect(n_samples)
x6 = get_external_effect(n_samples)
x1 = 0.6*x5 + get_external_effect(n_samples)
x3 = 0.5*x5 + get_external_effect(n_samples)
x0 = 1.0*x1 + 1.0*x3 + get_external_effect(n_samples)
x2 = 0.8*x0 - 0.6*x6 + get_external_effect(n_samples)
x4 = 1.0*x0 - 0.5*x6 + get_external_effect(n_samples)

# The latent variable x6 is not included.
X = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T, columns=['x0', 'x1', 'x2', 'x3
↪', 'x4', 'x5'])

X.head()
```

```
m = np.array([[ 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0],
              [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.6, 0.0],
              [ 0.8, 0.0, 0.0, 0.0, 0.0, 0.0,-0.6],
              [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.5, 0.0],
              [ 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,-0.5],
              [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
              [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]])
dot = make_dot(m, labels=['x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'f1(x6)'])

# Save pdf
dot.render('dag')

# Save png
dot.format = 'png'
dot.render('dag')

dot
```

2.12.4 Causal Discovery

To run causal discovery, we create a `RCD` object and call the `fit` method.

```
model = lingam.RCD()
model.fit(X)
```

```
<lingam.rcd.RCD at 0x25e725a4dd8>
```

Using the `ancestors_list_` properties, we can see the list of ancestors sets as a result of the causal discovery.

```
ancestors_list = model.ancestors_list_

for i, ancestors in enumerate(ancestors_list):
    print(f'M{i}={ancestors}')
```

```
M0={1, 3, 5}
M1={5}
M2={0, 1, 3, 5}
M3={5}
M4={0, 1, 3, 5}
M5=set()
```

Also, using the `adjacency_matrix_` properties, we can see the adjacency matrix as a result of the causal discovery. The coefficients between variables with latent confounders are np.nan.

```
model.adjacency_matrix_
```

```
array([[0.   , 0.939, 0.   , 0.994, 0.   , 0.   ],
       [0.   , 0.   , 0.   , 0.   , 0.   , 0.556],
       [0.751, 0.   , 0.   , 0.   ,   nan, 0.   ],
       [0.   , 0.   , 0.   , 0.   , 0.   , 0.563],
       [1.016, 0.   ,   nan, 0.   , 0.   , 0.   ],
       [0.   , 0.   , 0.   , 0.   , 0.   , 0.   ]])
```

```
make_dot(model.adjacency_matrix_)
```

2.12.5 Independence between error variables

To check if the LiNGAM assumption is broken, we can get p-values of independence between error variables. The value in the i-th row and j-th column of the obtained matrix shows the p-value of the independence of the error variables e_i and e_j.

```
p_values = model.get_error_independence_p_values(X)
print(p_values)
```

```
[[0.    0.      nan 0.413   nan 0.68 ]
 [0.    0.      nan 0.732   nan 0.382]
 [  nan   nan 0.      nan   nan   nan]
 [0.413 0.732   nan 0.      nan 0.054]
 [  nan   nan   nan   nan 0.      nan]
 [0.68  0.382   nan 0.054   nan 0.   ]]
```

2.12.6 Bootstrapping

We call `bootstrap()` method instead of `fit()`. Here, the second argument specifies the number of bootstrap sampling.

```
import warnings
warnings.filterwarnings('ignore', category=UserWarning)

model = lingam.RCD()
result = model.bootstrap(X, n_sampling=100)
```

2.12.7 Causal Directions

Since `BootstrapResult` object is returned, we can get the ranking of the causal directions extracted by `get_causal_direction_counts()` method. In the following sample code, `n_directions` option is limited to the causal directions of the top 8 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
cdc = result.get_causal_direction_counts(n_directions=8, min_causal_effect=0.01,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_causal_directions(cdc, 100)
```

```
x0 <--- x1 (b>0) (100.0%)
x4 <--- x0 (b>0) (99.0%)
x1 <--- x5 (b>0) (97.0%)
x2 <--- x0 (b>0) (96.0%)
x0 <--- x3 (b>0) (92.0%)
x3 <--- x5 (b>0) (67.0%)
x2 <--- x4 (b>0) (13.0%)
x4 <--- x3 (b<0) (11.0%)
```

2.12.8 Directed Acyclic Graphs

Also, using the `get_directed_acyclic_graph_counts()` method, we can get the ranking of the DAGs extracted. In the following sample code, `n_dags` option is limited to the dags of the top 3 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
dagc = result.get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.01,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_dagc(dagc, 100)
```

```
DAG[0]: 47.0%
    x0 <--- x1 (b>0)
    x0 <--- x3 (b>0)
    x1 <--- x5 (b>0)
    x2 <--- x0 (b>0)
    x3 <--- x5 (b>0)
    x4 <--- x0 (b>0)
DAG[1]: 20.0%
    x0 <--- x1 (b>0)
    x0 <--- x3 (b>0)
    x1 <--- x5 (b>0)
    x2 <--- x0 (b>0)
    x4 <--- x0 (b>0)
DAG[2]: 10.0%
    x0 <--- x1 (b>0)
    x0 <--- x3 (b>0)
    x1 <--- x5 (b>0)
    x2 <--- x0 (b>0)
    x3 <--- x5 (b>0)
```

(continues on next page)

(continued from previous page)

```
    x4 <--- x0 (b>0)
    x4 <--- x3 (b<0)
```

2.12.9 Probability

Using the `get_probabilities()` method, we can get the probability of bootstrapping.

```
prob = result.get_probabilities(min_causal_effect=0.01)
print(prob)
```

```
[[0.   1.   0.   0.92 0.   0.08]
 [0.   0.   0.   0.   0.   0.97]
 [0.96 0.   0.   0.   0.13 0.  ]
 [0.   0.   0.   0.   0.   0.67]
 [0.99 0.01 0.02 0.12 0.   0.  ]
 [0.   0.   0.   0.   0.   0.  ]]
```

2.12.10 Total Causal Effects

Using the `get_total_causal_effects()` method, we can get the list of total causal effect. The total causal effects we can get are dictionary type variable. We can display the list nicely by assigning it to pandas.DataFrame. Also, we have replaced the variable index with a label below.

```
causal_effects = result.get_total_causal_effects(min_causal_effect=0.01)

# Assign to pandas.DataFrame for pretty display
df = pd.DataFrame(causal_effects)
labels = [f'x{i}' for i in range(X.shape[1])]
df['from'] = df['from'].apply(lambda x : labels[x])
df['to'] = df['to'].apply(lambda x : labels[x])
df
```

We can easily perform sorting operations with pandas.DataFrame.

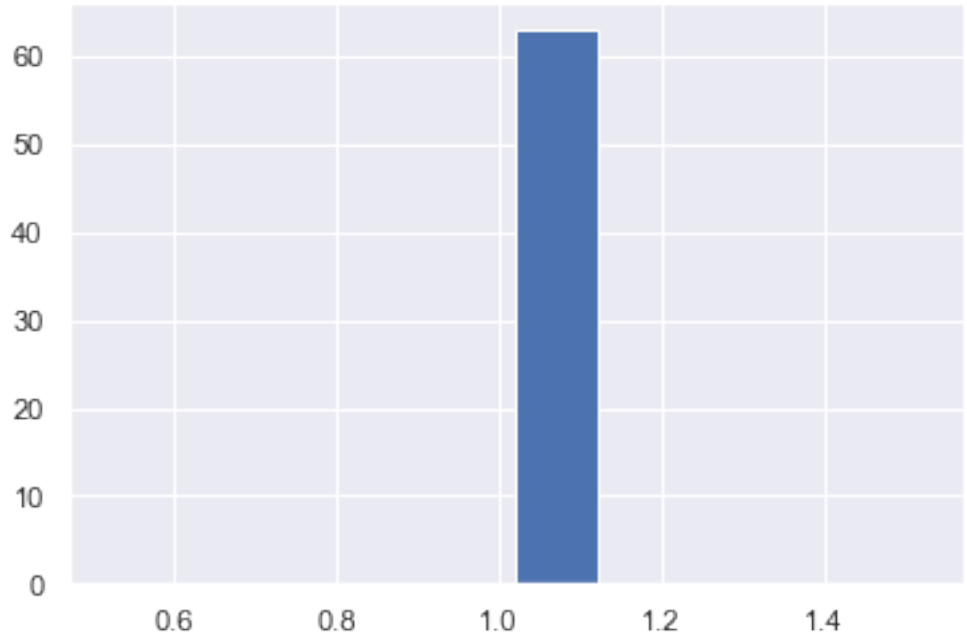
```
df.sort_values('effect', ascending=False).head()
```

```
df.sort_values('probability', ascending=True).head()
```

Because it holds the raw data of the causal effect (the original data for calculating the median), it is possible to draw a histogram of the values of the causal effect, as shown below.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline

from_index = 5 # index of x5
to_index = 0 # index of x0
plt.hist(result.total_effects_[:, to_index, from_index])
```



2.12.11 Bootstrap Probability of Path

Using the `get_paths()` method, we can explore all paths from any variable to any variable and calculate the bootstrap probability for each path. The path will be output as an array of variable indices. For example, the array `[3, 0, 1]` shows the path from variable X3 through variable X0 to variable X1.

```
from_index = 5 # index of x5
to_index = 4 # index of x4

pd.DataFrame(result.get_paths(from_index, to_index))
```

2.13 Draw Causal Graph

2.13.1 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`. And to draw the causal graph, we need to import `make_dot` method from `lingam.utils`.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import make_dot

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
np.random.seed(0)
```

```
['1.16.2', '0.24.2', '0.11.1', '1.3.1']
```

2.13.2 Draw the result of LiNGAM

First, we can draw a simple graph that is the result of LiNGAM.

```
x3 = np.random.uniform(size=10000)
x0 = 3.0*x3 + np.random.uniform(size=10000)
x2 = 6.0*x3 + np.random.uniform(size=10000)
x1 = 3.0*x0 + 2.0*x2 + np.random.uniform(size=10000)
x5 = 4.0*x0 + np.random.uniform(size=10000)
x4 = 8.0*x0 - 1.0*x2 + np.random.uniform(size=10000)
X = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T ,columns=['x0', 'x1', 'x2', 'x3
↪', 'x4', 'x5'])

model = lingam.DirectLiNGAM()
model.fit(X)
make_dot(model.adjacency_matrix_)
```

If we want to change the variable name, we can use `labels`.

```
labels = [f'var{i}' for i in range(X.shape[1])]
make_dot(model.adjacency_matrix_, labels=labels)
```

2.13.3 Save graph

The created dot data can be saved as an image file in addition to being displayed in Jupyter Notebook.

```
dot = make_dot(model.adjacency_matrix_, labels=labels)

# Save pdf
dot.render('dag')

# Save png
dot.format = 'png'
dot.render('dag')
```

```
'dag.png'
```

2.13.4 Draw the result of LiNGAM with prediction model

For example, we create a linear regression model with x0 as the target variable.

```
from sklearn.linear_model import LinearRegression

target = 0
features = [i for i in range(X.shape[1]) if i != target]
reg = LinearRegression()
```

(continues on next page)

(continued from previous page)

```
reg.fit(X.iloc[:, features], X.iloc[:, target])
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

By specify `prediction_feature_indices` and `prediction_coefs` that can be obtained from the prediction model, we can draw the prediction model with the causal structure.

```
make_dot(model.adjacency_matrix_, prediction_feature_indices=features, prediction_
↪coefs=reg.coef_)
```

Also, we can change the label of the target variable by `prediction_target_label`, omit the coefficient of prediction model without `prediction_coefs`, and change the color by `prediction_line_color`.

```
make_dot(model.adjacency_matrix_, prediction_feature_indices=features, prediction_
↪target_label='Target', prediction_line_color='#0000FF')
```

In addition to the above, we can use `prediction_feature_importance` to draw the importance of the prediction model as an edge label.

```
import lightgbm as lgb

target = 0
features = [i for i in range(X.shape[1]) if i != target]
reg = lgb.LGBMRegressor(random_state=0)
reg.fit(X.iloc[:, features], X.iloc[:, target])
reg.feature_importances_
```

```
array([619, 205, 310, 957, 909])
```

```
make_dot(model.adjacency_matrix_, prediction_feature_indices=features, prediction_
↪feature_importance=reg.feature_importances_)
```

2.14 LiNA

2.14.1 Model

LiNA[1] allows to locate the latent factors as well as uncover the causal structure between such latent factors of interests. Causal structure between latent factors can be ubiquitous in real-world applications, e.g., relations bewteen anxiety, depression, and coping in psychology[2][3] , etc.

[1] Y. Zeng, S. Shimizu, R. Cai, F. Xie, M. Yamamoto, and Z. Hao. Causal discovery with multi-domain LiNGAM for latent factors. In Proc. Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21), 2021.

[2] R. Silva, R. Scheines, C. Glymour, and P. Spirtes. Learning the structure of linear latent variable models. Journal of Machine Learning Research, 7(2):191-246, 2006.

[3] D. Bartholomew, F. Steele, I. Moustaki, and J. Galbraith. The analysis and interpretation of multivariate data for social scientists. Routledge (Second edition), 2008.

This method makes the following assumptions.

1. Linearity
2. Acyclicity
3. No causal relations between observed variables
4. Non-Gaussian continuous distubance variables (except at most one) for latent factors
5. Gaussian error variables (except at most one) for observed variables
6. Each latent factor has at lest 2 pure measurement variables.

References

2.14.2 Import and settings

In this example, we need to import `numpy`, and `random`, in addition to `lingam`.

```
import numpy as np
import random
import lingam
import lingam.utils as ut

print([np.__version__, lingam.__version__])
```

```
['1.20.3', '1.5.4']
```

2.14.3 Single-domain test data

First, we generate a causal structure with 10 measurement variables and 5 latent factors, where each latent variable has 2 pure measurement variables.

```
ut.set_random_seed(1)
noise_ratios = 0.1
n_features = 10  # number of measurement vars.
n_samples, n_features_latent, n_edges, graph_type, sem_type = 1000, 5, 5, 'ER',
↪'laplace'
B_true = ut.simulate_dag(n_features_latent, n_edges, graph_type)
W_true = ut.simulate_parameter(B_true)  # row to column

f, E, E_weight = ut.simulate_linear_sem(W_true, n_samples, sem_type)
f_nor = np.zeros([n_samples, n_features_latent])
scale = np.zeros([1, n_features_latent])
W_true_scale = np.zeros([n_features_latent, n_features_latent])
for j in range(n_features_latent):
    scale[0, j] = np.std(f[:, j])
    f_nor[:, j] = f[:, j] / np.std(f[:, j])
    W_true_scale[:, j] = W_true[:, j] / scale[0, j]  # scaled W_true

# generate noises ei of xi
e = np.random.random([n_features, n_samples])
for j in range(n_features):
    e[j, :] = e[j, :] - np.mean(e[j, :])
```

(continues on next page)

(continued from previous page)

```
    e[j, :] = e[j, :] / np.std(e[j, :])

G = np.zeros([n_features, n_features_latent])
G[0, 0] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G[1, 0] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G[2, 1] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G[3, 1] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G[4, 2] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G[5, 2] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G[6, 3] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G[7, 3] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G[8, 4] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G[9, 4] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G_sign = np.sign(G)

# normalize G
G_nor = np.zeros([n_features, n_features_latent])
for j in range(n_features):
    e[j, :] = e[j, :] / np.sqrt(np.square(np.sum(G[j, :])) + np.square(noise_ratios))
    G_nor[j, :] = G[j, :] / np.sqrt(np.square(np.sum(G[j, :])) + np.square(noise_
↪ratios))

X = G_nor @ f_nor.T + noise_ratios * e  # X:n_features*n_samples   "e is small or n_
↪features are large"
X = X.T

print('The true adjacency matrix is:\n', W_true)
```

```
The true adjacency matrix is:
[[ 0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.52905044 -1.87243368]
 [-1.94141783  0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          1.12108398]
 [ 0.          0.         -0.87478353  0.          0.        ]]
```

2.14.4 Causal Discovery for single-domain data

To run causal discovery, we create a `LiNA` object and call the `fit` method.

```
model = lingam.LiNA()
model.fit(X, G_sign, scale)
```

```
<lingam.lina.LiNA at 0x2130f482970>
```

Using the `_adjacency_matrix` properties, we can see the estimated adjacency matrix between latent factors.

```
print('The estimated adjacency matrix is:\n', model._adjacency_matrix)
```

```
The estimated adjacency matrix is:
[[ 0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.51703777 -1.75584025]
 [-1.75874721  0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.99860274]
 [ 0.          0.         -0.77518384  0.          0.        ]]
```

2.14.5 Multi-domain test data

We generate a causal structure with 2 domains where in each domain there are 6 measurement variables and 3 latent factors. Each latent factor has 2 pure measurement variables.

```
n_features = 6  # number of measurement vars. in each domain
noise_ratios = 0.1

ut.set_random_seed(1)

n_samples, n_features_latent, n_edges, graph_type, sem_type1, sem_type2 = 1000, 3, 3,
↪'ER', 'subGaussian', 'supGaussian'
# n_edges: number of edges btw. latent factors in a domain
# sem_type1/sem_type2: different distributions of noises from different domains
B_true = ut.simulate_dag(n_features_latent, n_edges, graph_type)  # skeleton btw.␣
↪latent factors
W_true = ut.simulate_parameter(B_true)  # causal effects matrix btw. latent factors

# 1 domain
f, E, E_weight = ut.simulate_linear_sem(W_true, n_samples, sem_type1)
f_nor1 = np.zeros([n_samples, n_features_latent])
scale1 = np.zeros([1, n_features_latent])
W_true_scale = np.zeros([n_features_latent, n_features_latent])
for j in range(n_features_latent):
    scale1[0, j] = np.std(f[:, j])
    f_nor1[:, j] = f[:, j] / np.std(f[:, j])
    W_true_scale[:, j] = W_true[:, j] / scale1[0, j]
e = np.random.random([n_features, n_samples])
for j in range(n_features):
    e[j, :] = e[j, :] - np.mean(e[j, :])
    e[j, :] = e[j, :] / np.std(e[j, :])

G1 = np.zeros([n_features, n_features_latent])
G1[0, 0] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G1[1, 0] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G1[2, 1] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G1[3, 1] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G1[4, 2] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G1[5, 2] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G_sign1 = np.sign(G1)
# normalize G
G_nor1 = np.zeros([n_features, n_features_latent])
for j in range(n_features):
    e[j, :] = e[j, :] / np.sqrt(np.square(np.sum(G1[j, :])) + np.square(noise_ratios))
    G_nor1[j, :] = G1[j, :] / np.sqrt(np.square(np.sum(G1[j, :])) + np.square(noise_
↪ratios))
X1 = G_nor1 @ f_nor1.T + noise_ratios * e  # "the noise ratio e is small or n_
↪features is large"
X1 = X1.T

# 2 domain
f2, E, E_weight = ut.simulate_linear_sem(W_true, n_samples, sem_type2)
f_nor2 = np.zeros([n_samples, n_features_latent])
scale2 = np.zeros([1, n_features_latent])
W_true_scale = np.zeros([n_features_latent, n_features_latent])
for j in range(n_features_latent):
    scale2[0, j] = np.std(f2[:, j])
    f_nor2[:, j] = f2[:, j] / np.std(f2[:, j])
```

(continues on next page)

(continued from previous page)

```
    W_true_scale[:, j] = W_true[:, j] / scale2[0, j]
e = np.random.random([n_features, n_samples])
for j in range(n_features):
    e[j, :] = e[j, :] - np.mean(e[j, :])
    e[j, :] = e[j, :] / np.std(e[j, :])
G2 = np.zeros([n_features, n_features_latent])
G2[0, 0] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G2[1, 0] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G2[2, 1] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G2[3, 1] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G2[4, 2] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G2[5, 2] = random.choice((-1, 1)) * (0.2 + 0.5 * np.random.rand(1))
G_sign2 = np.sign(G2)
# normalize G
G_nor2 = np.zeros([n_features, n_features_latent])
for j in range(n_features):
    e[j, :] = e[j, :] / np.sqrt(np.square(np.sum(G2[j, :])) + np.square(noise_ratios))
    G_nor2[j, :] = G2[j, :] / np.sqrt(np.square(np.sum(G2[j, :])) + np.square(noise_
↪ratios))
X2 = G_nor2 @ f_nor2.T + noise_ratios * e
X2 = X2.T  # X:n_samples * n_features

# augment the data X
X = scipy.linalg.block_diag(X1, X2)
G_sign = scipy.linalg.block_diag(G_sign1, G_sign2)
scale = scipy.linalg.block_diag(scale1, scale2)

print('The true adjacency matrix is:\n', W_true)
```

```
The true adjacency matrix is:
[[0.         1.18580721 1.14604785]
 [0.         0.         0.        ]
 [0.         0.63920121 0.        ]]
```

2.14.6 Causal Discovery for multi-domain data

To run causal discovery, we create a `MDLiNA` object and call the `fit` method.

```
model = lingam.MDLiNA()
model.fit(XX, G_sign, scale)
```

```
<lingam.lina.MDLiNA at 0x1812ee2fdf0>
```

Using the `_adjacency_matrix` properties, we can see the estimated adjacency matrix between latent factors of interest.

```
print('The estimated adjacency matrix is:\n', model._adjacency_matrix)
```

```
The estimated adjacency matrix is:
[[ 0.          0.34880702 -0.78706636]
 [ 0.          0.          0.61577239]
 [ 0.          0.          0.        ]]
```

2.15 RESIT

2.15.1 Model

RESIT[2] is an estimation algorithm for Additive Noise Model[1].

This method makes the following assumptions.

1. Continouos variables
2. Nonlinearity
3. Additive noise
4. Acyclicity
5. No hidden common causes

References

2.15.2 Import and settings

In this example, we need to import `numpy`, `pandas`, and `graphviz` in addition to `lingam`.

```
import numpy as np
import pandas as pd
import graphviz
import lingam
from lingam.utils import print_causal_directions, print_dagc, make_dot

import warnings
warnings.filterwarnings('ignore')

print([np.__version__, pd.__version__, graphviz.__version__, lingam.__version__])

np.set_printoptions(precision=3, suppress=True)
```

```
['1.21.5', '1.3.2', '0.17', '1.6.0']
```

2.15.3 Test data

First, we generate a causal structure with 7 variables. Then we create a dataset with 6 variables from x0 to x5, with x6 being the latent variable for x2 and x3.

```
X = pd.read_csv('nonlinear_data.csv')
```

```
m = np.array([
    [0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0],
```

(continues on next page)

[2] J. Peters, J. M. Mooij, D. Janzing, and B. Schölkopf. Causal discovery with continuous additive noise models Journal of Machine Learning Research, 15: 2009–2053, 2014.

[1] P. O. Hoyer, D. Janzing, J. M. Mooij, and J. Peters. and B. Schölkopf. Nonlinear causal discovery with additive noise models. Advances in Neural Information Processing Systems 21, pages 689–696. 2009.

(continued from previous page)

```
    [1, 1, 0, 0, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 1, 0]])

dot = make_dot(m)

# Save pdf
dot.render('dag')

# Save png
dot.format = 'png'
dot.render('dag')

dot
```

2.15.4 Causal Discovery

To run causal discovery, we create a `RESIT` object and call the `fit` method.

```
from sklearn.ensemble import RandomForestRegressor
reg = RandomForestRegressor(max_depth=4, random_state=0)

model = lingam.RESIT(regressor=reg)
model.fit(X)
```

```
<lingam.resit.RESIT at 0x201a773c548>
```

Using the `causal_order_` properties, we can see the causal ordering as a result of the causal discovery. x2 and x3, which have latent confounders as parents, are stored in a list without causal ordering.

```
model.causal_order_
```

```
[0, 1, 2, 3, 4]
```

Also, using the `adjacency_matrix_` properties, we can see the adjacency matrix as a result of the causal discovery. The coefficients between variables with latent confounders are np.nan.

```
model.adjacency_matrix_
```

```
array([[0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [1., 1., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

We can draw a causal graph by utility funciton.

```
make_dot(model.adjacency_matrix_)
```

2.15.5 Bootstrapping

We call `bootstrap()` method instead of `fit()`. Here, the second argument specifies the number of bootstrap sampling.

```
import warnings
warnings.filterwarnings('ignore', category=UserWarning)

n_sampling = 100
model = lingam.RESIT(regressor=reg)
result = model.bootstrap(X, n_sampling=n_sampling)
```

2.15.6 Causal Directions

Since `BootstrapResult` object is returned, we can get the ranking of the causal directions extracted by `get_causal_direction_counts()` method. In the following sample code, `n_directions` option is limited to the causal directions of the top 8 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
cdc = result.get_causal_direction_counts(n_directions=8, min_causal_effect=0.01,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_causal_directions(cdc, n_sampling)
```

```
x1 <--- x0 (b>0) (100.0%)
x2 <--- x1 (b>0) (71.0%)
x4 <--- x1 (b>0) (62.0%)
x2 <--- x0 (b>0) (62.0%)
x3 <--- x1 (b>0) (53.0%)
x3 <--- x4 (b>0) (52.0%)
x4 <--- x3 (b>0) (47.0%)
x3 <--- x0 (b>0) (44.0%)
```

2.15.7 Directed Acyclic Graphs

Also, using the `get_directed_acyclic_graph_counts()` method, we can get the ranking of the DAGs extracted. In the following sample code, `n_dags` option is limited to the dags of the top 3 rankings, and `min_causal_effect` option is limited to causal directions with a coefficient of 0.01 or more.

```
dagc = result.get_directed_acyclic_graph_counts(n_dags=3, min_causal_effect=0.01,␣
↪split_by_causal_effect_sign=True)
```

We can check the result by utility function.

```
print_dagc(dagc, n_sampling)
```

```
DAG[0]: 13.0%
    x1 <--- x0 (b>0)
    x2 <--- x1 (b>0)
    x3 <--- x4 (b>0)
    x4 <--- x0 (b>0)
```

(continues on next page)

(continued from previous page)

```
    x4 <--- x1 (b>0)
DAG[1]: 13.0%
    x1 <--- x0 (b>0)
    x2 <--- x0 (b>0)
    x2 <--- x1 (b>0)
    x3 <--- x4 (b>0)
    x4 <--- x1 (b>0)
DAG[2]: 11.0%
    x1 <--- x0 (b>0)
    x2 <--- x1 (b>0)
    x3 <--- x0 (b>0)
    x3 <--- x1 (b>0)
    x4 <--- x3 (b>0)
```

2.15.8 Probability

Using the `get_probabilities()` method, we can get the probability of bootstrapping.

```
prob = result.get_probabilities(min_causal_effect=0.01)
print(prob)
```

```
[[0.   0.   0.   0.02 0.  ]
 [1.   0.   0.07 0.05 0.01]
 [0.62 0.71 0.   0.06 0.03]
 [0.44 0.53 0.18 0.   0.52]
 [0.43 0.62 0.21 0.47 0.  ]]
```

2.15.9 Bootstrap Probability of Path

Using the `get_paths()` method, we can explore all paths from any variable to any variable and calculate the bootstrap probability for each path. The path will be output as an array of variable indices. For example, the array `[0, 1, 3]` shows the path from variable X0 through variable X1 to variable X3.

```
from_index = 0 # index of x0
to_index = 3 # index of x3

pd.DataFrame(result.get_paths(from_index, to_index))
```

2.16 LiM

2.16.1 Model

Linear Mixed (LiM) causal discovery algorithm[1] extends LiNGAM to handle the mixed data that consists of both continuous and discrete variables. The estimation is performed by first globally optimizing the log-likelihood function on the joint distribution of data with the acyclicity constraint, and then applying a local combinatorial search to output a causal graph.

This method makes the following assumptions.

[1] Y. Zeng, S. Shimizu, H. Matsui, F. Sun. Causal discovery for linear mixed data. In Proc. First Conference on Causal Learning and Reasoning (CLeaR2022). PMLR 177, pp. 994-1009, 2022.

1. Continous variables and binary variables.
2. Linearity
3. Acyclicity
4. No hidden common causes
5. Baselines are the same when predicting one binary variable from the other for every pair of binary variables.

References

2.16.2 Import and settings

In this example, we need to import `numpy`, and `random`, in addition to `lingam`.

```
import numpy as np
import random
import lingam
import lingam.utils as ut

print([np.__version__, lingam.__version__])
```

```
['1.20.3', '1.6.0']
```

2.16.3 Test data

First, we generate a causal structure with 2 variables, where one of them is randomly set to be a discrete variable.

```
ut.set_random_seed(1)
n_samples, n_features, n_edges, graph_type, sem_type = 1000, 2, 1, 'ER', 'mixed_
↪random_i_dis'
B_true = ut.simulate_dag(n_features, n_edges, graph_type)
W_true = ut.simulate_parameter(B_true)  # row to column

no_dis = np.random.randint(1, n_features)  # number of discrete vars.
print('There are %d discrete variable(s).' % (no_dis))
nodes = [iii for iii in range(n_features)]
dis_var = random.sample(nodes, no_dis) # randomly select no_dis discrete variables
dis_con = np.full((1, n_features), np.inf)
for iii in range(n_features):
    if iii in dis_var:
        dis_con[0, iii] = 0  # 1:continuous;   0:discrete
    else:
        dis_con[0, iii] = 1

X = ut.simulate_linear_mixed_sem(W_true, n_samples, sem_type, dis_con)

print('The true adjacency matrix is:\n', W_true)
```

```
There are 1 discrete variable(s).
The true adjacency matrix is:
[[0.        0.       ]
[1.3082251 0.       ]]
```

2.16.4 Causal Discovery for linear mixed data

To run causal discovery, we create a `LiM` object and call the `fit` method.

```
model = lingam.LiM()
model.fit(X, dis_con)
```

```
<lingam.lim.LiM at 0x174d475f850>
```

Using the `_adjacency_matrix` properties, we can see the estimated adjacency matrix between mixed variables.

```
print('The estimated adjacency matrix is:\n', model._adjacency_matrix)
```

```
The estimated adjacency matrix is:
[[ 0.         ,  0.         ],
 [-1.09938457,  0.         ]]
```

2.17 CAM-UV

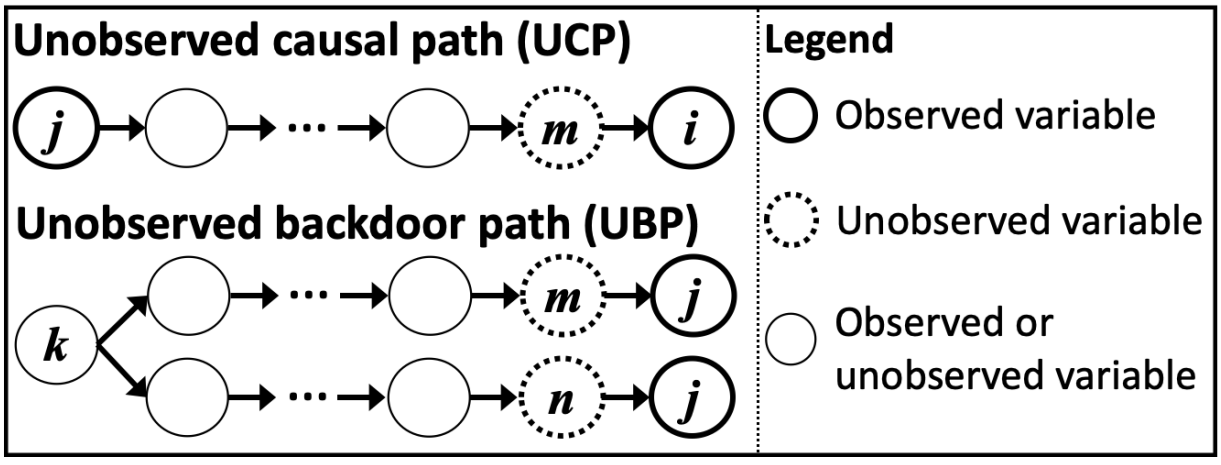
2.17.1 Model

This method CAM-UV (Causal Additive Models with Unobserved Variables) assumes an extension of the basic CAM model[1] to include unobserved variables. This method makes the following assumptions:

1. The effects of direct causes on a variable form generalized additive models (GAMs).
2. The causal structures form directed acyclic graphs (DAGs).

CAM-UV allows the existence of unobserved variables. It outputs a causal graph where a undirected edge indicates the pair of variables that have an unobserved causal path (UCP) or an unobserved backdoor path (UBP), and a directed edge indicates the causal direction of a pair of variables that do not have an UCP or UBP.

Definition of UCPs ans UBPs: As shown in the below figure, a causal path from x_j to x_i is called an UCP if it ends with the directed edge connecting x_i and its unobserved direct cause. A backdoor path between x_i and x_j is called an UBP if it starts with the edge connecting x_i and its unobserved direct cause, and ends with the edge connecting x_j and its unobserved direct cause.



[1] P. Bühlmann, J. Peters, and J. Ernest. CAM: Causal additive models, high-dimensional order search and penalized regression. Annals of Statistics, 42(6):2526–2556, 2014.

References

In Proc. 27th Conference on Uncertainty in Artificial Intelligence (UAI2021), PMLR 161:97-106, 2021.

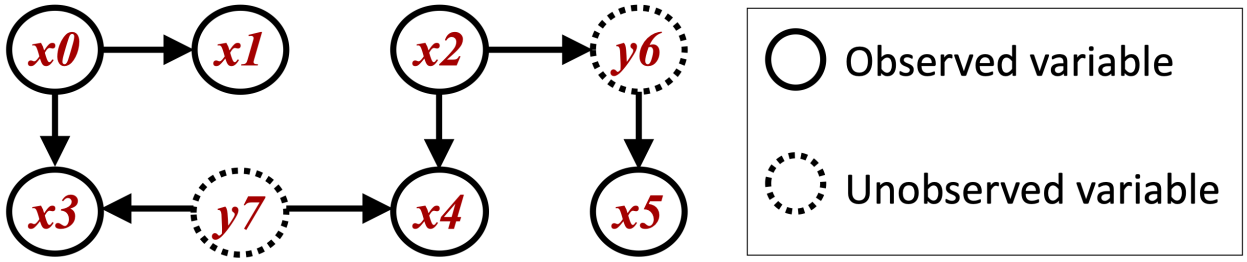
2.17.2 Import and settings

In this example, we need to import `numpy` in addition to `lingam`.

```
import numpy as np
import random
import lingam
```

2.17.3 Test data

First, we generate a causal structure with 2 unobserved variables (y6 and y7) and 6 observed variables (x0–x5) as shown in the below figure.




```
def get_noise(n):
    noise = ((np.random.rand(1, n)-0.5)*5).reshape(n)
    mean = get_random_constant(0.0,2.0)
    noise += mean
    return noise



def causal_func(cause):
    a = get_random_constant(-5.0,5.0)
    b = get_random_constant(-1.0,1.0)
    c = int(random.uniform(2,3))
    return ((cause+a)**(c))+b


def get_random_constant(s,b):
    constant = random.uniform(-1.0, 1.0)
    if constant>0:
        constant = random.uniform(s, b)
    else:
        constant = random.uniform(-b, -s)
    return constant


def create_data(n):
    causal_pairs = [[0,1],[0,3],[2,4]]
    intermediate_pairs = [[2,5]]
    confounder_pairs = [[3,4]]
```

(continues on next page)

(continued from previous page)

```
    n_variables = 6

    data = np.zeros((n, n_variables)) # observed data
    confounders = np.zeros((n, len(confounder_pairs))) # data of unobserced common␣
↪causes

    # Adding external effects
    for i in range(n_variables):
        data[:,i] = get_noise(n)
    for i in range(len(confounder_pairs)):
        confounders[:,i] = get_noise(n)
        confounders[:,i] = confounders[:,i] / np.std(confounders[:,i])

    # Adding the effects of unobserved common causes
    for i, cpair in enumerate(confounder_pairs):
        cpair = list(cpair)
        cpair.sort()
        data[:,cpair[0]] += causal_func(confounders[:,i])
        data[:,cpair[1]] += causal_func(confounders[:,i])

    for i1 in range(n_variables)[0:n_variables]:
        data[:,i1] = data[:,i1] / np.std(data[:,i1])
        for i2 in range(n_variables)[i1+1:n_variables+1]:
            # Adding direct effects between observed variables
            if [i1, i2] in causal_pairs:
                data[:,i2] += causal_func(data[:,i1])
            # Adding undirected effects between observed variables mediated through␣
↪unobserved variables
            if [i1, i2] in intermediate_pairs:
                interm = causal_func(data[:,i1])+get_noise(n)
                interm = interm / np.std(interm)
                data[:,i2] += causal_func(interm)

    return data


sample_size = 2000
X = create_data(sample_size)
```

2.17.4 Causal Discovery

To run causal discovery, we create a `CAMUV` object and call the `fit` method.

```
model = lingam.CAMUV()
model.fit(X)
```

Using the `adjacency_matrix_` properties, we can see the adjacency matrix as a result of the causal discovery. When the value of a variable pair is np.nan, the variables have a UCP or UBP.

```
model.adjacency_matrix_
```

```
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.,  0.],
```

(continues on next page)

(continued from previous page)

```
       [ 0.,  0.,  0.,  0.,  0., nan],
       [ 1.,  0.,  0.,  0., nan,  0.],
       [ 0.,  0.,  1., nan,  0.,  0.],
       [ 0.,  0., nan,  0.,  0.,  0.]])
```

CHAPTER 3

API Reference

3.1 ICA-LiNGAM

class lingam.**ICALiNGAM**(*random_state=None*, *max_iter=1000*)
Implementation of ICA-based LiNGAM Algorithm[1]

References

__init__(*random_state=None*, *max_iter=1000*)
Construct a ICA-based LiNGAM model.

Parameters

- **random_state** (*int, optional (default=None)*) – random_state is the seed used by the random number generator.
- **max_iter** (*int, optional (default=1000)*) – The maximum number of iterations of FastICA.

adjacency_matrix_
Estimated adjacency matrix.

Returns **adjacency_matrix_** – The adjacency matrix B of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features, n_features)

bootstrap(*X*, *n_sampling*)
Evaluate the statistical reliability of DAG based on the bootstrapping.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where n_samples is the number of samples and n_features is the number of features.

[1] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. J. Kerminen. A linear non-gaussian acyclic model for causal discovery. Journal of Machine Learning Research, 7:2003-2030, 2006.

- **n_sampling** (*int*) – Number of bootstrapping samples.

Returns **result** – Returns the result of bootstrapping.

Return type *BootstrapResult*

causal_order_
Estimated causal ordering.

Returns **causal_order_** – The causal order of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features)

estimate_total_effect(*X*, *from_index*, *to_index*)
Estimate total effect using causal model.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.
- **from_index** – Index of source variable to estimate total effect.
- **to_index** – Index of destination variable to estimate total effect.

Returns **total_effect** – Estimated total effect.

Return type float

fit(*X*)
Fit the model to X.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns **self** – Returns the instance of self.

Return type object

get_error_independence_p_values(*X*)
Calculate the p-value matrix of independence between error variables.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.

Returns **independence_p_values** – p-value matrix of independence between error variables.

Return type array-like, shape (n_features, n_features)

3.2 DirectLiNGAM

class lingam.**DirectLiNGAM**(*random_state=None*, *prior_knowledge=None*, *apply_prior_knowledge_softly=False*, *measure='pwling'*)
Implementation of DirectLiNGAM Algorithm[1,2]

[1] S. Shimizu, T. Inazumi, Y. Sogawa, A. Hyvärinen, Y. Kawahara, T. Washio, P. O. Hoyer and K. Bollen. DirectLiNGAM: A direct method for learning a linear non-Gaussian structural equation model. Journal of Machine Learning Research, 12(Apr): 1225–1248, 2011.

[2] A. Hyvärinen and S. M. Smith. Pairwise likelihood ratios for estimation of non-Gaussian structural eauation models. Journal of Machine Learning Research 14:111-152, 2013.

References

__init__(*random_state=None*, *prior_knowledge=None*, *apply_prior_knowledge_softly=False*, *measure='pwling'*)
Construct a DirectLiNGAM model.

Parameters

- **random_state** (*int, optional (default=None)*) – random_state is the seed used by the random number generator.
- **prior_knowledge** (*array-like, shape (n_features, n_features), optional (default=None)*) – Prior knowledge used for causal discovery, where n_features is the number of features.

  The elements of prior knowledge matrix are defined as follows[1]:

  - 0 : x_i does not have a directed path to x_j
  - 1 : x_i has a directed path to x_j
  - -1 : No prior knowledge is available to know if either of the two cases above (0 or 1) is true.
- **apply_prior_knowledge_softly** (*boolean, optional (default=False)*) – If True, apply prior knowledge softly.
- **measure** (*{'pwling', 'kernel'}, optional (default='pwling')*) – Measure to evaluate independence: 'pwling'[2] or 'kernel'[1].

adjacency_matrix_
Estimated adjacency matrix.

Returns **adjacency_matrix_** – The adjacency matrix B of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features, n_features)

bootstrap(*X*, *n_sampling*)
Evaluate the statistical reliability of DAG based on the bootstrapping.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where n_samples is the number of samples and n_features is the number of features.
- **n_sampling** (*int*) – Number of bootstrapping samples.

Returns **result** – Returns the result of bootstrapping.

Return type *BootstrapResult*

causal_order_
Estimated causal ordering.

Returns **causal_order_** – The causal order of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features)

estimate_total_effect(*X*, *from_index*, *to_index*)
Estimate total effect using causal model.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.
- **from_index** – Index of source variable to estimate total effect.
- **to_index** – Index of destination variable to estimate total effect.

Returns **total_effect** – Estimated total effect.

Return type float

fit(*X*)
Fit the model to X.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns **self** – Returns the instance itself.

Return type object

get_error_independence_p_values(*X*)
Calculate the p-value matrix of independence between error variables.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.

Returns **independence_p_values** – p-value matrix of independence between error variables.

Return type array-like, shape (n_features, n_features)

3.3 MultiGroupDirectLiNGAM

class lingam.**MultiGroupDirectLiNGAM**(*random_state=None, prior_knowledge=None, apply_prior_knowledge_softly=False*)
Implementation of DirectLiNGAM Algorithm with multiple groups[1]

References

__init__(*random_state=None*, *prior_knowledge=None*, *apply_prior_knowledge_softly=False*)
Construct a model.

Parameters

- **random_state** (*int, optional (default=None)*) – `random_state` is the seed used by the random number generator.
- **prior_knowledge** (*array-like, shape (n_features, n_features), optional (default=None)*) – Prior knowledge used for causal discovery, where `n_features` is the number of features.

  The elements of prior knowledge matrix are defined as follows[1]:

  – `0` : x_i does not have a directed path to x_j

  – `1` : x_i has a directed path to x_j

[1] S. Shimizu. Joint estimation of linear non-Gaussian acyclic models. Neurocomputing, 81: 104-107, 2012.

- `-1` : No prior knowledge is available to know if either of the two cases above (0 or 1) is true.

- **apply_prior_knowledge_softly** (*boolean, optional (default=False)*) – If True, apply prior knowledge softly.

adjacency_matrices_
Estimated adjacency matrices.

Returns **adjacency_matrices_** – The list of adjacency matrix B for multiple datasets. The shape of B is (n_features, n_features), where n_features is the number of features.

Return type array-like, shape (B, ..)

adjacency_matrix_
Estimated adjacency matrix.

Returns **adjacency_matrix_** – The adjacency matrix B of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features, n_features)

bootstrap(*X_list*, *n_sampling*)
Evaluate the statistical reliability of DAG based on the bootstrapping.

Parameters

- **X_list** (*array-like, shape (X, ..)*) – Multiple datasets for training, where `X` is an dataset. The shape of ''X" is (n_samples, n_features), where `n_samples` is the number of samples and `n_features` is the number of features.

- **n_sampling** (*int*) – Number of bootstrapping samples.

Returns **results** – Returns the results of bootstrapping for multiple datasets.

Return type array-like, shape (*BootstrapResult*, ..)

causal_order_
Estimated causal ordering.

Returns **causal_order_** – The causal order of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features)

estimate_total_effect(*X_list*, *from_index*, *to_index*)
Estimate total effect using causal model.

Parameters

- **X_list** (*array-like, shape (X, ..)*) – Multiple datasets for training, where `X` is an dataset. The shape of ''X" is (n_samples, n_features), where `n_samples` is the number of samples and `n_features` is the number of features.

- **from_index** – Index of source variable to estimate total effect.

- **to_index** – Index of destination variable to estimate total effect.

Returns **total_effect** – Estimated total effect.

Return type float

fit(*X_list*)
Fit the model to multiple datasets.

Parameters **X_list** (*list, shape [X, ..]*) – Multiple datasets for training, where X is an dataset. The shape of ''X" is (n_samples, n_features), where n_samples is the number of samples and n_features is the number of features.

Returns **self** – Returns the instance itself.

Return type object

get_error_independence_p_values(*X_list*)

Calculate the p-value matrix of independence between error variables.

Parameters **X_list** (*array-like, shape (X, ..)*) – Multiple datasets for training, where X is an dataset. The shape of ''X" is (n_samples, n_features), where n_samples is the number of samples and n_features is the number of features.

Returns **independence_p_values** – p-value matrix of independence between error variables.

Return type array-like, shape (n_datasets, n_features, n_features)

3.4 VAR-LiNGAM

class lingam.**VARLiNGAM**(*lags=1*, *criterion='bic'*, *prune=False*, *ar_coefs=None*, *lingam_model=None*, *random_state=None*)

Implementation of VAR-LiNGAM Algorithm[1]

References

__init__(*lags=1*, *criterion='bic'*, *prune=False*, *ar_coefs=None*, *lingam_model=None*, *random_state=None*)

Construct a VARLiNGAM model.

Parameters

- **lags** (*int, optional (default=1)*) – Number of lags.
- **criterion** (*{'aic', 'fpe', 'hqic', 'bic', None}, optional (default='bic')*) – Criterion to decide the best lags within lags. Searching the best lags is disabled if criterion is None.
- **prune** (*boolean, optional (default=False)*) – Whether to prune the adjacency matrix or not.
- **ar_coefs** (*array-like, optional (default=None)*) – Coefficients of AR model. Estimating AR model is skipped if specified ar_coefs. Shape must be (lags, n_features, n_features).
- **lingam_model** (*lingam object inherits 'lingam._BaseLiNGAM', optional (default=None)*) – LiNGAM model for causal discovery. If None, DirectLiNGAM algorithm is selected.
- **random_state** (*int, optional (default=None)*) – random_state is the seed used by the random number generator.

adjacency_matrices_

Estimated adjacency matrix.

[1] Aapo Hyvärinen, Kun Zhang, Shohei Shimizu, Patrik O. Hoyer. Estimation of a Structural Vector Autoregression Model Using Non-Gaussianity. Journal of Machine Learning Research, 11: 1709-1731, 2010.

Returns **adjacency_matrices_** – The adjacency matrix of fitted model, where n_features is the number of features.

Return type array-like, shape (lags, n_features, n_features)

bootstrap(*X*, *n_sampling*)
Evaluate the statistical reliability of DAG based on the bootstrapping.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.
- **n_sampling** (*int*) – Number of bootstrapping samples.

Returns **result** – Returns the result of bootstrapping.

Return type TimeseriesBootstrapResult

causal_order_
Estimated causal ordering.

Returns **causal_order_** – The causal order of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features)

estimate_total_effect(*X*, *from_index*, *to_index*, *from_lag=0*)
Estimate total effect using causal model.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.
- **from_index** – Index of source variable to estimate total effect.
- **to_index** – Index of destination variable to estimate total effect.

Returns **total_effect** – Estimated total effect.

Return type float

fit(*X*)
Fit the model to X.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns **self** – Returns the instance itself.

Return type object

get_error_independence_p_values()
Calculate the p-value matrix of independence between error variables.

Returns **independence_p_values** – p-value matrix of independence between error variables.

Return type array-like, shape (n_features, n_features)

residuals_
Residuals of regression.

Returns **residuals_** – Residuals of regression, where n_samples is the number of samples.

Return type array-like, shape (n_samples)

3.5 VARMA-LiNGAM

class lingam.**VARMALiNGAM**(*order=(1, 1), criterion='bic', prune=False, max_iter=100, ar_coefs=None, ma_coefs=None, lingam_model=None, random_state=None*)

Implementation of VARMA-LiNGAM Algorithm[1]

References

__init__(*order=(1, 1), criterion='bic', prune=False, max_iter=100, ar_coefs=None, ma_coefs=None, lingam_model=None, random_state=None*)

Construct a VARMALiNGAM model.

Parameters

- **order** (*turple, length = 2, optional (default=(1, 1))*) – Number of lags for AR and MA model.
- **criterion** (*{'aic', 'bic', 'hqic', None}, optional (default='bic')*) – Criterion to decide the best order in the all combinations of `order`. Searching the best order is disabled if `criterion` is `None`.
- **prune** (*boolean, optional (default=False)*) – Whether to prune the adjacency matrix or not.
- **max_iter** (*int, optional (default=100)*) – Maximm number of iterations to estimate VARMA model.
- **ar_coefs** (*array-like, optional (default=None)*) – Coefficients of AR of ARMA. Estimating ARMA model is skipped if specified `ar_coefs` and *ma_coefs*. Shape must be (`order[0]`, n_features, n_features).
- **ma_coefs** (*array-like, optional (default=None)*) – Coefficients of MA of ARMA. Estimating ARMA model is skipped if specified `ar_coefs` and *ma_coefs*. Shape must be (`order[1]`, n_features, n_features).
- **lingam_model** (*lingam object inherits 'lingam._BaseLiNGAM', optional (default=None)*) – LiNGAM model for causal discovery. If None, DirectLiNGAM algorithm is selected.
- **random_state** (*int, optional (default=None)*) – `random_state` is the seed used by the random number generator.

adjacency_matrices_

Estimated adjacency matrix.

Returns **adjacency_matrices_** – The adjacency matrix psi and omega of fitted model, where n_features is the number of features.

Return type array-like, shape ((p, n_features, n_features), (q, n_features, n_features))

bootstrap(*X, n_sampling*)

Evaluate the statistical reliability of DAG based on the bootstrapping.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

[1] Yoshinobu Kawahara, Shohei Shimizu, Takashi Washio. Analyzing relationships among ARMA processes based on non-Gaussianity of external influences. Neurocomputing, Volume 74: 2212-2221, 2011

- **n_sampling** (*int*) – Number of bootstrapping samples.

Returns **result** – Returns the result of bootstrapping.

Return type TimeseriesBootstrapResult

causal_order_
Estimated causal ordering.

Returns **causal_order_** – The causal order of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features)

estimate_total_effect (*X*, *E*, *from_index*, *to_index*, *from_lag=0*)
Estimate total effect using causal model.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.
- **E** (*array-like, shape (n_samples, n_features)*) – Original error data, where n_samples is the number of samples and n_features is the number of features.
- **from_index** – Index of source variable to estimate total effect.
- **to_index** – Index of destination variable to estimate total effect.

Returns **total_effect** – Estimated total effect.

Return type float

fit (*X*)
Fit the model to X.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns **self** – Returns the instance itself.

Return type object

get_error_independence_p_values ()
Calculate the p-value matrix of independence between error variables.

Returns **independence_p_values** – p-value matrix of independence between error variables.

Return type array-like, shape (n_features, n_features)

residuals_
Residuals of regression.

Returns **residuals_** – Residuals of regression, where n_samples is the number of samples.

Return type array-like, shape (n_samples)

3.6 LongitudinalLiNGAM

class lingam.**LongitudinalLiNGAM**(*n_lags=1*, *measure='pwling'*, *random_state=None*)
Implementation of Longitudinal LiNGAM algorithm[1]

[1] K. Kadowaki, S. Shimizu, and T. Washio. Estimation of causal structures in longitudinal data using non-Gaussianity. In Proc. 23rd IEEE International Workshop on Machine Learning for Signal Processing (MLSP2013), pp. 1–6, Southampton, United Kingdom, 2013.

References

__init__ (*n_lags=1*, *measure='pwling'*, *random_state=None*)
Construct a model.

Parameters

- **n_lags** (*int, optional (default=1)*) – Number of lags.
- **measure** (*{'pwling', 'kernel'}, default='pwling'*) – Measure to evaluate independence : 'pwling' or 'kernel'.
- **random_state** (*int, optional (default=None)*) – random_state is the seed used by the random number generator.

adjacency_matrices_
Estimated adjacency matrices.

Returns **adjacency_matrices_** – The list of adjacency matrix B(t,t) and B(t,t-τ) for longitudinal datasets. The shape of B(t,t) and B(t,t-τ) is (n_features, n_features), where n_features is the number of features. **If the previous data required for the calculation are not available, such as B(t,t) or B(t,t-τ) at t=0, all elements of the matrix are nan**.

Return type array-like, shape ((B(t,t), B(t,t-1), .., B(t,t-τ)), ..)

bootstrap (*X_list*, *n_sampling*, *start_from_t=1*)
Evaluate the statistical reliability of DAG based on the bootstrapping.

Parameters

- **X_list** (*array-like, shape (X, ..)*) – Longitudinal multiple datasets for training, where X is an dataset. The shape of ''X" is (n_samples, n_features), where n_samples is the number of samples and n_features is the number of features.
- **n_sampling** (*int*) – Number of bootstrapping samples.

Returns **results** – Returns the results of bootstrapping for multiple datasets.

Return type array-like, shape (*BootstrapResult*, ..)

causal_orders_
Estimated causal ordering.

Returns **causal_order_** – The causal order of fitted models for B(t,t). The shape of causal_order is (n_features), where n_features is the number of features.

Return type array-like, shape (causal_order, ..)

estimate_total_effect (*X_t*, *from_t*, *from_index*, *to_t*, *to_index*)
Estimate total effect using causal model.

Parameters

- **X_t** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.
- **_t** (*from*) – The timepoint of source variable.
- **from_index** – Index of source variable to estimate total effect.
- **to_t** – The timepoint of destination variable.
- **to_index** – Index of destination variable to estimate total effect.

Returns **total_effect** – Estimated total effect.

Return type float

fit(*X_list*)
Fit the model to datasets.

Parameters **X_list** (*list, shape [X, ..]*) – Longitudinal multiple datasets for training, where X is an dataset. The shape of X is (n_samples, n_features), where n_samples is the number of samples and n_features is the number of features.

Returns **self** – Returns the instance itself.

Return type object

get_error_independence_p_values()
Calculate the p-value matrix of independence between error variables.

Returns **independence_p_values** – p-value matrix of independence between error variables.

Return type array-like, shape (n_features, n_features)

residuals_
Residuals of regression.

Returns **residuals_** – Residuals of regression, where E is an dataset. The shape of E is (n_samples, n_features), where n_samples is the number of samples and n_features is the number of features.

Return type list, shape [E, ..]

3.7 BootstrapResult

class lingam.**BootstrapResult**(*adjacency_matrices*, *total_effects*)
The result of bootstrapping.

__init__(*adjacency_matrices*, *total_effects*)
Construct a BootstrapResult.

Parameters

- **adjacency_matrices** (*array-like, shape (n_sampling)*) – The adjacency matrix list by bootstrapping.
- **total_effects** (*array-like, shape (n_sampling)*) – The total effects list by bootstrapping.

adjacency_matrices_
The adjacency matrix list by bootstrapping.

Returns **adjacency_matrices_** – The adjacency matrix list, where n_sampling is the number of bootstrap sampling.

Return type array-like, shape (n_sampling)

get_causal_direction_counts(*n_directions=None*, *min_causal_effect=None*, *split_by_causal_effect_sign=False*)
Get causal direction count as a result of bootstrapping.

Parameters

- **n_directions** (*int, optional (default=None)*) – If int, then The top n_directions items are included in the result

- **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. If float, then causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.
- **split_by_causal_effect_sign** (*boolean, optional (default=False)*) – If True, then causal directions are split depending on the sign of the causal effect.

Returns

causal_direction_counts – List of causal directions sorted by count in descending order. The dictionary has the following format:

```
{'from': [n_directions], 'to': [n_directions], 'count': [n_
↪directions]}
```

where `n_directions` is the number of causal directions.

Return type dict

get_directed_acyclic_graph_counts(*n_dags=None*, *min_causal_effect=None*, *split_by_causal_effect_sign=False*)

Get DAGs count as a result of bootstrapping.

Parameters

- **n_dags** (*int, optional (default=None)*) – If int, then The top `n_dags` items are included in the result
- **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. If float, then causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.
- **split_by_causal_effect_sign** (*boolean, optional (default=False)*) – If True, then causal directions are split depending on the sign of the causal effect.

Returns

directed_acyclic_graph_counts – List of directed acyclic graphs sorted by count in descending order. The dictionary has the following format:

```
{'dag': [n_dags], 'count': [n_dags]}.
```

where `n_dags` is the number of directed acyclic graphs.

Return type dict

get_paths(*from_index*, *to_index*, *min_causal_effect=None*)

Get all paths from the start variable to the end variable and their bootstrap probabilities.

Parameters

- **from_index** (*int*) – Index of the variable at the start of the path.
- **to_index** (*int*) – Index of the variable at the end of the path.
- **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. Causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.

Returns

paths – List of path and bootstrap probability. The dictionary has the following format:

```
{'path': [n_paths], 'effect': [n_paths], 'probability': [n_paths]}
```

where `n_paths` is the number of paths.

Return type dict

get_probabilities(*min_causal_effect=None*)
Get bootstrap probability.

Parameters **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. If float, then causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.

Returns **probabilities** – List of bootstrap probability matrix.

Return type array-like

get_total_causal_effects(*min_causal_effect=None*)
Get total effects list.

Parameters **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. If float, then causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.

Returns

total_causal_effects – List of bootstrap total causal effect sorted by probability in descending order. The dictionary has the following format:

```
{'from': [n_directions], 'to': [n_directions], 'effect': [n_
↪directions], 'probability': [n_directions]}
```

where `n_directions` is the number of causal directions.

Return type dict

total_effects_
The total effect list by bootstrapping.

Returns **total_effects_** – The total effect list, where `n_sampling` is the number of bootstrap sampling.

Return type array-like, shape (n_sampling)

3.8 TimeseriesBootstrapResult

3.9 LongitudinalBootstrapResult

class lingam.**LongitudinalBootstrapResult**(*n_timepoints*, *adjacency_matrices*, *total_effects*)
The result of bootstrapping for LongitudinalLiNGAM.

__init__(*n_timepoints*, *adjacency_matrices*, *total_effects*)
Construct a BootstrapResult.

Parameters

- **adjacency_matrices** (*array-like, shape (n_sampling)*) – The adjacency matrix list by bootstrapping.

- **total_effects** (*array-like, shape (n_sampling)*) – The total effects list by bootstrapping.

adjacency_matrices_
The adjacency matrix list by bootstrapping.

Returns **adjacency_matrices_** – The adjacency matrix list, where `n_sampling` is the number of bootstrap sampling.

Return type array-like, shape (n_sampling)

get_causal_direction_counts(*n_directions=None*, *min_causal_effect=None*, *split_by_causal_effect_sign=False*)
Get causal direction count as a result of bootstrapping.

Parameters

- **n_directions** (*int, optional (default=None)*) – If int, then The top `n_directions` items are included in the result
- **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. If float, then causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.
- **split_by_causal_effect_sign** (*boolean, optional (default=False)*) – If True, then causal directions are split depending on the sign of the causal effect.

Returns

causal_direction_counts – List of causal directions sorted by count in descending order. The dictionary has the following format:

```
{'from': [n_directions], 'to': [n_directions], 'count': [n_
↪directions]}
```

where `n_directions` is the number of causal directions.

Return type dict

get_directed_acyclic_graph_counts(*n_dags=None*, *min_causal_effect=None*, *split_by_causal_effect_sign=False*)
Get DAGs count as a result of bootstrapping.

Parameters

- **n_dags** (*int, optional (default=None)*) – If int, then The top `n_dags` items are included in the result
- **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. If float, then causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.
- **split_by_causal_effect_sign** (*boolean, optional (default=False)*) – If True, then causal directions are split depending on the sign of the causal effect.

Returns

directed_acyclic_graph_counts – List of directed acyclic graphs sorted by count in descending order. The dictionary has the following format:

```
{'dag': [n_dags], 'count': [n_dags]}.
```

where `n_dags` is the number of directed acyclic graphs.

Return type dict

get_paths(*from_index*, *to_index*, *from_t*, *to_t*, *min_causal_effect=None*)
Get all paths from the start variable to the end variable and their bootstrap probabilities.

Parameters

- **from_index** (*int*) – Index of the variable at the start of the path.
- **to_index** (*int*) – Index of the variable at the end of the path.
- **from_t** (*int*) – The starting timepoint of the path.
- **to_t** (*int*) – The end timepoint of the path.
- **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. Causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.

Returns

paths – List of path and bootstrap probability. The dictionary has the following format:

```
{'path': [n_paths], 'effect': [n_paths], 'probability': [n_paths]}
```

where `n_paths` is the number of paths.

Return type dict

get_probabilities(*min_causal_effect=None*)
Get bootstrap probability.

Parameters **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. If float, then causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.

Returns **probabilities** – List of bootstrap probability matrix.

Return type array-like

get_total_causal_effects(*min_causal_effect=None*)
Get total effects list.

Parameters **min_causal_effect** (*float, optional (default=None)*) – Threshold for detecting causal direction. If float, then causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.

Returns

total_causal_effects – List of bootstrap total causal effect sorted by probability in descending order. The dictionary has the following format:

```
{'from': [n_directions], 'to': [n_directions], 'effect': [n_
↪directions], 'probability': [n_directions]}
```

where `n_directions` is the number of causal directions.

Return type dict

total_effects_
The total effect list by bootstrapping.

Returns **total_effects_** – The total effect list, where `n_sampling` is the number of bootstrap sampling.

Return type array-like, shape (n_sampling)

3.10 BottomUpParceLiNGAM

class lingam.**BottomUpParceLiNGAM**(*random_state=None*, *alpha=0.1*, *regressor=None*, *prior_knowledge=None*)

Implementation of ParceLiNGAM Algorithm[1]

References

__init__(*random_state=None*, *alpha=0.1*, *regressor=None*, *prior_knowledge=None*)

Construct a BottomUpParceLiNGAM model.

Parameters

- **random_state** (*int, optional (default=None)*) – random_state is the seed used by the random number generator.
- **alpha** (*float, optional (default=0.1)*) – Significant level of statistical test. If alpha=0.0, rejection does not occur in statistical tests.
- **regressor** (*regressor object implementing 'fit' and 'predict' function (default=None)*) – Regressor to compute residuals. This regressor object must have fit``method and ``predict function like scikit-learn's model.
- **prior_knowledge** (*array-like, shape (n_features, n_features), optional (default=None)*) – Prior knowledge used for causal discovery, where n_features is the number of features.

  The elements of prior knowledge matrix are defined as follows[1]:

  - 0 : x_i does not have a directed path to x_j
  - 1 : x_i has a directed path to x_j
  - -1 : No prior knowledge is available to know if either of the two cases above (0 or 1) is true.

adjacency_matrix_

Estimated adjacency matrix.

Returns **adjacency_matrix_** – The adjacency matrix B of fitted model, where n_features is the number of features. Set np.nan if order is unknown.

Return type array-like, shape (n_features, n_features)

bootstrap(*X*, *n_sampling*)

Evaluate the statistical reliability of DAG based on the bootstrapping.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where n_samples is the number of samples and n_features is the number of features.
- **n_sampling** (*int*) – Number of bootstrapping samples.

Returns **result** – Returns the result of bootstrapping.

[1] T. Tashiro, S. Shimizu, and A. Hyvärinen. ParceLiNGAM: a causal ordering method robust against latent confounders. Neural computation, 26.1: 57-83, 2014.

Return type *BootstrapResult*

causal_order_
Estimated causal ordering.

Returns **causal_order_** – The causal order of fitted model, where n_features is the number of features. Set the features as a list if order is unknown.

Return type array-like, shape (n_features)

estimate_total_effect(*X*, *from_index*, *to_index*)
Estimate total effect using causal model.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.
- **from_index** – Index of source variable to estimate total effect.
- **to_index** – Index of destination variable to estimate total effect.

Returns **total_effect** – Estimated total effect.

Return type float

fit(*X*)
Fit the model to X.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns **self** – Returns the instance itself.

Return type object

get_error_independence_p_values(*X*)
Calculate the p-value matrix of independence between error variables.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.

Returns **independence_p_values** – p-value matrix of independence between error variables.

Return type array-like, shape (n_features, n_features)

3.11 RCD

class lingam.**RCD**(*max_explanatory_num=2*, *cor_alpha=0.01*, *ind_alpha=0.01*, *shapiro_alpha=0.01*, *MLHSICR=False*, *bw_method='mdbs'*)
Implementation of RCD Algorithm[1]

References

__init__(*max_explanatory_num=2*, *cor_alpha=0.01*, *ind_alpha=0.01*, *shapiro_alpha=0.01*, *MLHSICR=False*, *bw_method='mdbs'*)
Construct a RCD model.

Parameters

[1] T.N.Maeda and S.Shimizu. RCD: Repetitive causal discovery of linear non-Gaussian acyclic models with latent confounders. In Proc. 23rd International Conference on Artificial Intelligence and Statistics (AISTATS2020), Palermo, Sicily, Italy. PMLR 108:735-745, 2020.

- **max_explanatory_num** (*int, optional (default=2)*) – Maximum number of explanatory variables.
- **cor_alpha** (*float, optional (default=0.01)*) – Alpha level for pearson correlation.
- **ind_alpha** (*float, optional (default=0.01)*) – Alpha level for HSIC.
- **shapiro_alpha** (*float, optional (default=0.01)*) – Alpha level for Shapiro-Wilk test.
- **MLHSICR** (*bool, optional (default=False)*) – If True, use MLHSICR for multiple regression, if False, use OLS for multiple regression.
- **bw_method** (*str, optional (default=``mdbs``)*) – The method used to calculate the bandwidth of the HSIC.
  - `mdbs` : Median distance between samples.
  - `scott` : Scott's Rule of Thumb.
  - `silverman` : Silverman's Rule of Thumb.

adjacency_matrix_
Estimated adjacency matrix.

Returns **adjacency_matrix_** – The adjacency matrix B of fitted model, where n_features is the number of features. Set np.nan if order is unknown.

Return type array-like, shape (n_features, n_features)

ancestors_list_
Estimated ancestors list.

Returns **ancestors_list_** – The list of causal ancestors sets, where n_features is the number of features.

Return type array-like, shape (n_features)

bootstrap(*X*, *n_sampling*)
Evaluate the statistical reliability of DAG based on the bootstrapping.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.
- **n_sampling** (*int*) – Number of bootstrapping samples.

Returns **result** – Returns the result of bootstrapping.

Return type *BootstrapResult*

fit(*X*)
Fit the model to X.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns **self** – Returns the instance itself.

Return type object

get_error_independence_p_values(*X*)
Calculate the p-value matrix of independence between error variables.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.

Returns **independence_p_values** – p-value matrix of independence between error variables.

Return type array-like, shape (n_features, n_features)

3.12 CausalEffect

class lingam.**CausalEffect**(*causal_model*)

Implementation of causality and prediction.[1]

References

__init__(*causal_model*)

Construct a CausalEffect.

Parameters **causal_model** (*lingam object inherits 'lingam._BaseLiNGAM' or array-like with shape (n_features, n_features)*) – Causal model for calculating causal effects. The lingam object is `lingam.DirectLiNGAM` or `lingam.ICALiNGAM`, and `fit` function needs to be executed already. For array-like, adjacency matrix to estimate causal effect, where `n_features` is the number of features.

estimate_effects_on_prediction(*X*, *target_index*, *pred_model*)

Estimate the intervention effect with the prediction model.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Original data, where `n_samples` is the number of samples and `n_features` is the number of features.
- **target_index** (*int*) – Index of target variable.
- **pred_model** (*model object implementing 'predict' or 'predict_proba'*) – Model to predict the expectation. For linear regression or non-linear reggression, model object must have `predict` method. For logistic regression, model object must have `predict_proba` method.

Returns **intervention_effects** – Estimated values of intervention effect. The first column of the list is the value of 'E[Y|do(Xi=mean)]-E[Y|do(Xi=mean+std)]', and the second column is the value of 'E[Y|do(Xi=mean)]–E[Y|do(Xi=mean-std)]'. The maximum value in this array is the feature having the greatest intervention effect.

Return type array-like, shape (n_features, 2)

estimate_optimal_intervention(*X*, *target_index*, *pred_model*, *intervention_index*, *desired_output*)

Estimate of the intervention such that the expectation of the prediction of the post-intervention observations is equal or close to a specified value.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Original data, where `n_samples` is the number of samples and `n_features` is the number of features.

[1] P. Blöbaum and S. Shimizu. Estimation of interventional effects of features on prediction. In Proc. 2017 IEEE International Workshop on Machine Learning for Signal Processing (MLSP2017), pp. 1–6, Tokyo, Japan, 2017.

- **target_index** (*int*) – Index of target variable.
- **pred_model** (*model object.*) – Model to predict the expectation. Only linear regression model can be specified. Model object musst have `coef_` and `intercept_` attributes.
- **intervention_index** (*int*) – Index of variable to apply intervention.
- **desired_output** – Desired expected post-intervention output of prediction.

Returns **optimal_intervention** – Optimal intervention on `intervention_index` variable.

Return type float

3.13 utils

lingam.utils.**print_causal_directions**(*cdc*, *n_sampling*, *labels=None*)

Print causal directions of bootstrap result to stdout.

Parameters

- **cdc** (*dict*) – List of causal directions sorted by count in descending order. This can be set the value returned by `BootstrapResult.get_causal_direction_counts()` method.
- **n_sampling** (*int*) – Number of bootstrapping samples.
- **labels** (*array-like, optional (default=None)*) – List of feature lables. If set labels, the output feature name will be the specified label.

lingam.utils.**print_dagc**(*dagc*, *n_sampling*, *labels=None*)

Print DAGs of bootstrap result to stdout.

Parameters

- **dagc** (*dict*) – List of directed acyclic graphs sorted by count in descending order. This can be set the value returned by `BootstrapResult.get_directed_acyclic_graph_counts()` method.
- **n_sampling** (*int*) – Number of bootstrapping samples.
- **labels** (*array-like, optional (default=None)*) – List of feature lables. If set labels, the output feature name will be the specified label.

lingam.utils.**make_prior_knowledge**(*n_variables*, *exogenous_variables=None*, *sink_variables=None*, *paths=None*, *no_paths=None*)

Make matrix of prior knowledge.

Parameters

- **n_variables** (*int*) – Number of variables.
- **exogenous_variables** (*array-like, shape (index, ..), optional (default=None)*) – List of exogenous variables(index). Prior knowledge is created with the specified variables as exogenous variables.
- **sink_variables** (*array-like, shape (index, ..), optional (default=None)*) – List of sink variables(index). Prior knowledge is created with the specified variables as sink variables.

- **paths** (*array-like, shape ((index, index), ..), optional (default=None)*) – List of variables(index) pairs with directed path. If `(i, j)`, prior knowledge is created that xi has a directed path to xj.
- **no_paths** (*array-like, shape ((index, index), ..), optional (default=None)*) – List of variables(index) pairs without directed path. If `(i, j)`, prior knowledge is created that xi does not have a directed path to xj.

Returns **prior_knowledge** – Return matrix of prior knowledge used for causal discovery.

Return type array-like, shape (n_variables, n_variables)

`lingam.utils.`**`remove_effect`**(*X*, *remove_features*)

Create a dataset that removes the effects of features by linear regression.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Data, where `n_samples` is the number of samples and `n_features` is the number of features.
- **remove_features** (*array-like*) – List of features(index) to remove effects.

Returns **X** – Data after removing effects of `remove_features`.

Return type array-like, shape (n_samples, n_features)

`lingam.utils.`**`make_dot`**(*adjacency_matrix*, *labels=None*, *lower_limit=0.01*, *prediction_feature_indices=None*, *prediction_target_label='Y(pred)'*, *prediction_line_color='red'*, *prediction_coefs=None*, *prediction_feature_importance=None*, *ignore_shape=False*)

Directed graph source code in the DOT language with specified adjacency matrix.

Parameters

- **adjacency_matrix** (*array-like with shape (n_features, n_features)*) – Adjacency matrix to make graph, where `n_features` is the number of features.
- **labels** (*array-like, optional (default=None)*) – Label to use for graph features.
- **lower_limit** (*float, optional (default=0.01)*) – Threshold for drawing direction. If float, then directions with absolute values of coefficients less than `lower_limit` are excluded.
- **prediction_feature_indices** (*array-like, optional (default=None)*) – Indices to use as prediction features.
- **prediction_target_label** (*string, optional (default='Y(pred)')*) – Label to use for target variable of prediction.
- **prediction_line_color** (*string, optional (default='red')*) – Line color to use for prediction's graph.
- **prediction_coefs** (*array-like, optional (default=None)*) – Coefficients to use for prediction's graph.
- **prediction_feature_importance** (*array-like, optional (default=None)*) – Feature importance to use for prediction's graph.
- **ignore_shape** (*boolean, optional (default=False)*) – Ignore checking the shape of adjaceny_matrix or not.

Returns **graph** – Directed graph source code in the DOT language. If order is unknown, draw a double-headed arrow.

Return type graphviz.Digraph

lingam.utils.**get_sink_variables**(*adjacency_matrix*)
The sink variables(index) in the adjacency matrix.

Parameters **adjacency_matrix** (*array-like, shape (n_variables, n_variables)*) – Adjacency matrix, where n_variables is the number of variables.

Returns **sink_variables** – List of sink variables(index).

Return type array-like

lingam.utils.**get_exo_variables**(*adjacency_matrix*)
The exogenous variables(index) in the adjacency matrix.

Parameters **adjacency_matrix** (*array-like, shape (n_variables, n_variables)*) – Adjacency matrix, where n_variables is the number of variables.

Returns **exogenous_variables** – List of exogenous variables(index).

Return type array-like

lingam.utils.**find_all_paths**(*dag*, *from_index*, *to_index*, *min_causal_effect=0.0*)
Find all paths from point to point in DAG.

Parameters

- **dag** (*array-like, shape (n_features, n_features)*) – The adjacency matrix to fine all paths, where n_features is the number of features.
- **from_index** (*int*) – Index of the variable at the start of the path.
- **to_index** (*int*) – Index of the variable at the end of the path.
- **min_causal_effect** (*float, optional (default=0.0)*) – Threshold for detecting causal direction. Causal directions with absolute values of causal effects less than `min_causal_effect` are excluded.

Returns

- **paths** (*array-like, shape (n_paths)*) – List of found path, where n_paths is the number of paths.
- **effects** (*array-like, shape (n_paths)*) – List of causal effect, where n_paths is the number of paths.

lingam.utils.**simulate_linear_sem**(*adjacency_matrix*, *n_samples*, *sem_type*, *noise_scale=1.0*)
Simulate samples from linear SEM with specified type of noise.

Parameters

- **adjacency_matrix** (*array-like, shape (n_features, n_features)*) – Weighted adjacency matrix of DAG, where `n_features` is the number of variables.
- **n_samples** (*int*) – Number of samples. n_samples=inf mimics population risk.
- **sem_type** (*str*) – SEM type. gauss, exp, gumbel, logistic, poisson.
- **noise_scale** (*float*) – scale parameter of additive noise.

Returns **X** – Data generated from linear SEM with specified type of noise, where `n_features` is the number of variables.

Return type array-like, shape (n_samples, n_features)

lingam.utils.**simulate_linear_mixed_sem**(*adjacency_matrix*, *n_samples*, *sem_type*, *dis_con*, *noise_scale=None*)

Simulate mixed samples from linear SEM with specified type of noise.

Parameters

- **adjacency_matrix** (*array-like, shape (n_features, n_features)*) – Weighted adjacency matrix of DAG, where n_features is the number of variables.
- **n_samples** (*int*) – Number of samples. n_samples=inf mimics population risk.
- **sem_type** (*str*) – SEM type. gauss, mixed_random_i_dis.
- **dis_con** (*array-like, shape (1, n_features)*) – Indicator of discrete/continuous variables, where "1" indicates a continuous variable, while "0" a discrete variable.
- **noise_scale** (*float*) – scale parameter of additive noise.

Returns **X** – Data generated from linear SEM with specified type of noise, where n_features is the number of variables.

Return type array-like, shape (n_samples, n_features)

lingam.utils.**is_dag**(*W*)

Check if W is a dag or not.

Parameters **W** (*array-like, shape (n_features, n_features)*) – Binary adjacency matrix of DAG, where n_features is the number of features.

Returns **G** – Returns true or false.

Return type boolean

lingam.utils.**count_accuracy**(*W_true*, *W*, *W_und=None*)

Compute recalls and precisions for W, or optionally for CPDAG = W + W_und.

Parameters

- **W_true** (*array-like, shape (n_features, n_features)*) – Ground truth graph, where n_features is the number of features.
- **W** (*array-like, shape (n_features, n_features)*) – Predicted graph.
- **W_und** (*array-like, shape (n_features, n_features)*) – Predicted undirected edges in CPDAG, asymmetric.

Returns

- **recall** (*float*) – (true positive) / (true positive + false negative).
- **precision** (*float*) – (true positive) / (true positive + false positive).

lingam.utils.**simulate_parameter**(*B*, *w_ranges=((-2.0, -0.5), (0.5, 2.0))*)

Simulate SEM parameters for a DAG.

Parameters

- **B** (*array-like, shape (n_features, n_features)*) – Binary adjacency matrix of DAG, where n_features is the number of features.
- **w_ranges** (*tuple*) – Disjoint weight ranges.

Returns **adjacency_matrix** – Weighted adj matrix of DAG, where n_features is the number of features.

Return type array-like, shape (n_features, n_features)

lingam.utils.**simulate_dag**(*n_features*, *n_edges*, *graph_type*)
Simulate random DAG with some expected number of edges.

Parameters

- **n_features** (*int*) – Number of features.
- **n_edges** (*int*) – Expected number of edges.
- **graph_type** (*str*) – ER, SF.

Returns **B** – binary adjacency matrix of DAG.

Return type array-like, shape (n_features, n_features)

lingam.utils.**predict_adaptive_lasso**(*X*, *predictors*, *target*, *gamma=1.0*)
Predict with Adaptive Lasso.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where n_samples is the number of samples and n_features is the number of features.
- **predictors** (*array-like, shape (n_predictors)*) – Indices of predictor variable.
- **target** (*int*) – Index of target variable.

Returns **coef** – Coefficients of predictor variable.

Return type array-like, shape (n_features)

lingam.utils.**likelihood_i**(*x*, *i*, *b_i*, *bi_0*)
Compute local log-likelihood of component i.

Parameters

- **x** (*array-like, shape (n_features, n_samples)*) – Data, where `n_samples` is the number of samples and `n_features` is the number of features.
- **i** (*array-like*) – Variable index.
- **b_i** (*array-like*) – The i^th column of adjacency matrix, B[i].
- **bi_0** (*float*) – Constant value for the i^th variable.

Returns **ll** – Local log-likelihood of component i.

Return type float

lingam.utils.**log_p_super_gaussian**(*s*)
Compute density function of the normalized independent components.

Parameters **s** (*array-like, shape (1, n_samples)*) – Data, where `n_samples` is the number of samples.

Returns **x** – Density function of the normalized independent components, whose disturbances are super-Gaussian.

Return type float

lingam.utils.**variance_i**(*X*, *i*, *b_i*)
Compute empirical variance of component i.

Parameters

- **x** (*array-like, shape (n_features, n_samples)*) – Data, where `n_samples` is the number of samples and `n_features` is the number of features.

- **i** (*array-like*) – Variable index.
- **b_i** (*array-like*) – The i^th column of adjacency matrix, B[i].

Returns **variance** – Empirical variance of component i.

Return type float

3.14 LiNA

class lingam.**LiNA**(*w_threshold=0.3*, *lambda1=0.1*, *lambda2=0.1*, *loss_type='laplace'*, *max_iter=100*, *h_tol=1e-08*, *rho_max=1e+16*)

Implementation of LiNA Algorithm [1]_

References

__init__(*w_threshold=0.3*, *lambda1=0.1*, *lambda2=0.1*, *loss_type='laplace'*, *max_iter=100*, *h_tol=1e-08*, *rho_max=1e+16*)

Construct a LiNA model.

Parameters

- **w_threshold** (*float (default=0.3)*) – Drop edge if the weight btw. latent factors is less than w_threshold.
- **lambda1** (*float, optional (default=0.1)*) – L1 penalty parameter.
- **lambda2** (*float, (default=0.1)*) – L2 penalty parameter.
- **loss_type** (*str, (default='laplace')*) – Type of distribution of the noise.
- **max_iter** (*int, (default=100)*) – Maximum number of dual ascent steps.
- **h_tol** (*float, (default=1e-8)*) – Tolerance parameter of the acyclicity constraint.
- **rho_max** (*float, (default=1e+16)*) – Maximum value of the regularization parameter rho.

adjacency_matrix_

Estimated adjacency matrix between latent factors.

Returns **adjacency_matrix_** – The adjacency matrix of latent factors, where `n_features_latent` is the number of latent factors.

Return type array-like, shape (n_features_latent, n_features_latent)

fit(*X*, *G_sign*, *scale*)

Fit the model to X with measurement structure and latent factors' scales.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of measurement features.
- **G_sign** (*array-like, shape (n_features, n_features_latent)*) – Measurement structure matrix, where `n_features_latent` is the number of latent factors and `n_features` is the number of measurement features.
- **scale** (*array-like, shape (1, n_features_latent)*) – Scales of the latent factors.

Returns **self** – Returns the instance of self.

Return type object

measurement_matrix_
Estimated measurement matrix between measurement variables and latent factors.

Returns **measurement_matrix_** – The measurement matrix between measurement variables and latent factors, where `n_features_latent` is the number of latent factors and `n_features` is the number of measurement variables.

Return type array-like, shape (n_features, n_features_latent)

class lingam.**MDLiNA**(*w_threshold=0.3*, *lambda1=0.1*, *lambda2=0.1*, *loss_type='laplace'*, *max_iter=100*, *h_tol=1e-08*, *rho_max=1e+16*, *no_of_domain=2*, *no_of_latent_1domain=3*)
Implementation of MD-LiNA Algorithm [1]_

References

__init__(*w_threshold=0.3*, *lambda1=0.1*, *lambda2=0.1*, *loss_type='laplace'*, *max_iter=100*, *h_tol=1e-08*, *rho_max=1e+16*, *no_of_domain=2*, *no_of_latent_1domain=3*)
Construct an MD-LiNA model.

Parameters

- **w_threshold** (*float (default=0.3)*) – Drop edge if the weight btw. latent factors is less than w_threshold.
- **lambda1** (*float, optional (default=0.1)*) – L1 penalty parameter.
- **lambda2** (*float, (default=0.1)*) – L2 penalty parameter.
- **loss_type** (*str, (default='laplace')*) – Type of distribution of the noise.
- **max_iter** (*int, (default=100)*) – Maximum number of dual ascent steps.
- **h_tol** (*float, (default=1e-8)*) – Tolerance parameter of the acyclicity constraint.
- **rho_max** (*float, (default=1e+16)*) – Maximum value of the regularization parameter rho.
- **no_of_domain** (*int, (default=2)*) – Number of domains.
- **no_of_latent_1domain** (*float, (default=3)*) – Number of latent factors in a domain.

adjacency_matrix_
Estimated adjacency matrix between latent factors of interest, which is shared by all domains.

Returns **adjacency_matrix_** – The adjacency matrix of latent factors of interest, where `n_features_latent_1domain` is the number of latent factors of interest.

Return type array-like, shape (n_features_latent_1domain, n_features_latent_1domain)

fit(*X*, *G_sign*, *scale*)
Fit the model to X with measurement structure and latent factors' scales.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples of all domains and `n_features` is the number of features of all domains.

- **G_sign** (*array-like, shape (n_features, n_features_latent)*) – Measurement structure matrix, where `n_features_latent` is the number of latent factors of all domains and `n_features` is the number of measurement variables of all domains.
- **scale** (*array-like, shape (1, n_features_latent)*) – Scales of the latent factors.

Returns **self** – Returns the instance of self.

Return type object

measurement_matrix_
Estimated measurement matrix between measurement variables and latent factors from all domains.

Returns **measurement_matrix_** – The measurement matrix between measurement variables and latent factors, where `n_features_latent` is the number of latent factors and `n_features` is the number of measurement variables from all domains.

Return type array-like, shape (n_features, n_features_latent)

3.15 RESIT

class lingam.**RESIT**(*regressor*, *random_state=None*, *alpha=0.01*)
Implementation of RESIT(regression with subsequent independence test) Algorithm[1]

References

Notes

RESIT algorithm returns an **adjacency matrix consisting of zeros or ones**, rather than an adjacency matrix consisting of causal coefficients, in order to estimate nonlinear causality.

__init__(*regressor*, *random_state=None*, *alpha=0.01*)
Construct a RESIT model.

Parameters

- **regressor** (*regressor object implementing 'fit' and 'predict' function (default=None)*) – Regressor to compute residuals. This regressor object must have `fit` method and `predict` function like scikit-learn's model.
- **random_state** (*int, optional (default=None)*) – `random_state` is the seed used by the random number generator.
- **alpha** (*float, optional (default=0.01)*) – Alpha level for HSIC independence test when removing superfluous edges.

adjacency_matrix_
Estimated adjacency matrix.

Returns **adjacency_matrix_** – The adjacency matrix B of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features, n_features)

[1] Jonas Peters, Joris M Mooij, Dominik Janzing, and Bernhard Sch olkopf. Causal discovery with continuous additive noise models. Journal of Machine Learning Research, 15:2009-2053, 2014.

bootstrap(*X*, *n_sampling*)
Evaluate the statistical reliability of DAG based on the bootstrapping.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.
- **n_sampling** (*int*) – Number of bootstrapping samples.

Returns **result** – Returns the result of bootstrapping.

Return type *BootstrapResult*

causal_order_
Estimated causal ordering.

Returns **causal_order_** – The causal order of fitted model, where n_features is the number of features.

Return type array-like, shape (n_features)

estimate_total_effect(*X*, *from_index*, *to_index*)
Estimate total effect using causal model.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.
- **from_index** – Index of source variable to estimate total effect.
- **to_index** – Index of destination variable to estimate total effect.

Returns **total_effect** – **Because RESIT is a nonlinear algorithm, it cannot estimate the total effect and always returns a value of zero**

Return type float

fit(*X*)
Fit the model to X.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns **self** – Returns the instance itself.

Return type object

get_error_independence_p_values(*X*)
Calculate the p-value matrix of independence between error variables.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Original data, where n_samples is the number of samples and n_features is the number of features.

Returns **independence_p_values** – **RESIT always returns zero**

Return type array-like, shape (n_features, n_features)

3.16 LiM

class lingam.**LiM**(*lambda1=0.1*, *loss_type='mixed'*, *max_iter=150*, *h_tol=1e-08*, *rho_max=1e+16*, *w_threshold=0.1*)

Implementation of LiM Algorithm[1]

References

__init__(*lambda1=0.1*, *loss_type='mixed'*, *max_iter=150*, *h_tol=1e-08*, *rho_max=1e+16*, *w_threshold=0.1*)

Construct a LiM model.

Parameters

- **lambda1** (*float, optional (default=0.1)*) – L1 penalty parameter.
- **loss_type** (*str, (default='mixed')*) – Type of distribution of the noise.
- **max_iter** (*int, (default=150)*) – Maximum number of dual ascent steps.
- **h_tol** (*float, (default=1e-8)*) – Tolerance parameter of the acyclicity constraint.
- **rho_max** (*float, (default=1e+16)*) – Maximum value of the regularization parameter rho.
- **w_threshold** (*float (default=0.1)*) – Drop edge if the weight btw. variables is less than w_threshold.

adjacency_matrix_

Estimated adjacency matrix between mixed variables.

Returns **adjacency_matrix_** – The adjacency matrix of variables, where n_features is the number of observed variables.

Return type array-like, shape (n_features, n_features)

fit(*X*, *dis_con*)

Fit the model to X with mixed data.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where n_samples is the number of samples and n_features is the number of observed variables.
- **dis_con** (*array-like, shape (1, n_features)*) – Indicators of discrete or continuous variables, where "1" indicates a continuous variable, while "0" a discrete variable.

Returns **self** – Returns the instance of self.

Return type object

[1] Zeng Y, Shimizu S, Matsui H, et al. Causal discovery for linear mixed data[C]//Conference on Causal Learning and Rea- soning. PMLR, 2022: 994-1009.

3.17 CAM-UV

class lingam.**CAMUV**(*alpha=0.01*, *num_explanatory_vals=2*)
Implementation of CAM-UV Algorithm[1]

References

__init__(*alpha=0.01*, *num_explanatory_vals=2*)
Construct a CAM-UV model.

Parameters

- **alpha** (*float, optional (default=0.01)*) – Alpha level.
- **num_explanatory_vals** (*int, optional (default=2)*) – Maximum number of explanatory variables.

[1] T.N.Maeda and S.Shimizu. Causal additive models with unobserved variables. In Proc. Thirty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI). PMLR 161:97-106, 2021.

CHAPTER 4

Indices and tables

- genindex

Python Module Index

l

Index

Symbols

A

B

C

D

E

F

G

I

L

M

P

R

S

T

V