lingam Documentation

Release 1.8.3

Author

Mar 10, 2024

CONTENTS:

1 Installation Guide **1**

2 Tutorial **3**

2.1 DirectLiNGAM . . . 3
2.1.1 Model . . . 3
2.1.2 Import and settings . . . 4
2.1.3 Test data . . . 4
2.1.4 Causal Discovery . . . 5
2.1.5 Independence between error variables . . . 5
2.2 Bootstrap . . . 6
2.2.1 Import and settings . . . 6
2.2.2 Test data . . . 6
2.2.3 Bootstrapping . . . 7
2.2.4 Causal Directions . . . 7
2.2.5 Directed Acyclic Graphs . . . 7
2.2.6 Probability . . . 8
2.2.7 Total Causal Effects . . . 8
2.2.8 Bootstrap Probability of Path . . . 9
2.3 How to use prior knowledge in DirectLiNGAM . . . 10
2.3.1 Import and settings . . . 10
2.3.2 Utility function . . . 10
2.3.3 Test data . . . 11
2.3.4 Make Prior Knowledge Matrix . . . 11
2.3.5 Causal Discovery . . . 12
2.4 MultiGroupDirectLiNGAM . . . 12
2.4.1 Model . . . 12
2.4.2 Import and settings . . . 13
2.4.3 Test data . . . 13
2.4.4 Causal Discovery . . . 14
2.4.5 Independence between error variables . . . 15
2.4.6 Bootstrapping . . . 16
2.4.7 Causal Directions . . . 16
2.4.8 Directed Acyclic Graphs . . . 17
2.4.9 Probability . . . 18
2.4.10 Total Causal Effects . . . 18
2.4.11 Bootstrap Probability of Path . . . 19
2.5 Total Effect . . . 19
2.6 Causal Effect on predicted variables . . . 20
2.6.1 Identification of Feature with Greatest Causal Influence on Prediction . . . 20
2.6.2 Estimation of Optimal Intervention . . . 21

2.6.3 Use a known causal model . . . 21
2.7 VARLiNGAM . . . 21
2.7.1 Model . . . 21
2.7.2 Import and settings . . . 22
2.7.3 Test data . . . 22
2.7.4 Causal Discovery . . . 23
2.7.5 Independence between error variables . . . 24
2.7.6 Bootstrap . . . 24
2.7.6.1 Bootstrapping . . . 24
2.7.7 Causal Directions . . . 25
2.7.8 Directed Acyclic Graphs . . . 25
2.7.9 Probability . . . 26
2.7.10 Total Causal Effects . . . 27
2.8 VARMALiNGAM . . . 28
2.8.1 Model . . . 28
2.8.2 Import and settings . . . 28
2.8.3 Test data . . . 29
2.8.4 Causal Discovery . . . 29
2.8.5 Independence between error variables . . . 31
2.8.6 Bootstrap . . . 31
2.8.6.1 Bootstrapping . . . 31
2.8.7 Causal Directions . . . 31
2.8.8 Directed Acyclic Graphs . . . 32
2.8.9 Probability . . . 34
2.8.10 Total Causal Effects . . . 35
2.9 Longitudinal LiNGAM . . . 36
2.9.1 Model . . . 36
2.9.2 Import and settings . . . 36
2.9.3 Test data . . . 37
2.9.4 Causal Discovery . . . 38
2.9.5 Independence between error variables . . . 41
2.9.6 Bootstrapping . . . 42
2.9.7 Causal Directions . . . 42
2.9.8 Directed Acyclic Graphs . . . 43
2.9.9 Probability . . . 47
2.9.10 Total Causal Effects . . . 48
2.10 BottomUpParceLiNGAM . . . 49
2.10.1 Model . . . 49
2.10.2 Import and settings . . . 50
2.10.3 Test data . . . 50
2.10.4 Causal Discovery . . . 51
2.10.5 Independence between error variables . . . 52
2.10.6 Bootstrapping . . . 52
2.10.7 Causal Directions . . . 52
2.10.8 Directed Acyclic Graphs . . . 53
2.10.9 Probability . . . 53
2.10.10 Total Causal Effects . . . 53
2.10.11 Bootstrap Probability of Path . . . 55
2.11 How to use prior knowledge in BottomUpParceLiNGAM . . . 55
2.11.1 Import and settings . . . 55
2.11.2 Utility function . . . 55
2.11.3 Test data . . . 56
2.11.4 Make Prior Knowledge Matrix . . . 56
2.11.5 Causal Discovery . . . 57

2.12 RCD . . . 57
2.12.1 Model . . . 57
2.12.2 Import and settings . . . 58
2.12.3 Test data . . . 58
2.12.4 Causal Discovery . . . 59
2.12.5 Independence between error variables . . . 60
2.12.6 Bootstrapping . . . 60
2.12.7 Causal Directions . . . 60
2.12.8 Directed Acyclic Graphs . . . 61
2.12.9 Probability . . . 61
2.12.10 Total Causal Effects . . . 62
2.12.11 Bootstrap Probability of Path . . . 63
2.13 Draw Causal Graph . . . 63
2.13.1 Import and settings . . . 63
2.13.2 Draw the result of LiNGAM . . . 63
2.13.3 Save graph . . . 64
2.13.4 Draw the result of LiNGAM with prediction model . . . 64
2.13.5 Highlight paths between specified nodes . . . 65
2.13.6 Draw the result of LiNGAM with emphasis on descendants and ancestors . . . 65
2.13.7 Draw the result of Bootstrap with emphasis on descendants and ancestors . . . 66
2.14 LiNA . . . 66
2.14.1 Model . . . 66
2.14.2 Import and settings . . . 67
2.14.3 Single-domain test data . . . 67
2.14.4 Causal Discovery for single-domain data . . . 68
2.14.5 Multi-domain test data . . . 69
2.14.6 Causal Discovery for multi-domain data . . . 70
2.15 RESIT . . . 71
2.15.1 Model . . . 71
2.15.2 Import and settings . . . 71
2.15.3 Test data . . . 72
2.15.4 Causal Discovery . . . 72
2.15.5 Bootstrapping . . . 73
2.15.6 Causal Directions . . . 73
2.15.7 Directed Acyclic Graphs . . . 74
2.15.8 Probability . . . 74
2.15.9 Bootstrap Probability of Path . . . 75
2.16 LiM . . . 75
2.16.1 Model . . . 75
2.16.2 Import and settings . . . 76
2.16.3 Test data . . . 76
2.16.4 Causal Discovery for linear mixed data . . . 77
2.17 CAM-UV . . . 77
2.17.1 Model . . . 77
2.17.2 Import and settings . . . 78
2.17.3 Test data . . . 78
2.17.4 Causal Discovery . . . 80
2.18 MultiGroupRCD . . . 80
2.18.1 Import and settings . . . 80
2.18.2 Test data . . . 80
2.18.3 Causal Discovery . . . 82
2.19 Finding ancestors of each variable . . . 83
2.19.1 Import and settings . . . 84
2.19.2 Test data . . . 84

2.19.3 Extract the ancestors of each observed variable . . . 85
2.20 F-correlation . . . 85
2.20.1 Import and settings . . . 85
2.20.2 Test data . . . 85
2.20.3 Linear correlated data (Uncorrelated) . . . 86
2.20.4 Linear correlated data (Strongly correlated) . . . 87
2.20.5 Non-linear correlated data (Quadratic function) . . . 87
2.20.6 Non-linear correlated data (Sin function) . . . 88
2.21 Visualization of nonlinear causal effect . . . 89
2.21.1 Import and settings . . . 89
2.21.2 Data generation . . . 90
2.21.3 Causal discovery . . . 90
2.21.4 Visualization . . . 90
2.22 EvaluateModelFit . . . 91
2.22.1 Import and settings . . . 91
2.22.2 When all variables are continuous data . . . 91
2.22.3 Test data . . . 91
2.22.4 Causal Discovery . . . 92
2.22.5 Evaluation . . . 92
2.22.6 When the data has hidden common causes . . . 92
2.22.7 Test data . . . 92
2.22.8 Causal Discovery . . . 93
2.22.9 When the data has ordinal variables . . . 93
2.22.10 Test data . . . 93

3 API Reference **95**
3.1 ICA-LiNGAM . . . 95
3.2 DirectLiNGAM . . . 97
3.3 MultiGroupDirectLiNGAM . . . 99
3.4 VAR-LiNGAM . . . 101
3.5 VARMA-LiNGAM . . . 103
3.6 LongitudinalLiNGAM . . . 106
3.7 BootstrapResult . . . 108
3.8 VARBootstrapResult . . . 111
3.9 VARMABootstrapResult . . . 113
3.10 LongitudinalBootstrapResult . . . 116
3.11 BottomUpParceLiNGAM . . . 119
3.12 RCD . . . 121
3.13 CausalEffect . . . 123
3.14 utils . . . 124
3.15 LiNA . . . 130
3.16 RESIT . . . 133
3.17 LiM . . . 135
3.18 CAM-UV . . . 136
3.19 MultiGroupRCD . . . 136

4 Indices and tables **141**

Python Module Index **143**

Index **145**

CHAPTER

ONE

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

TWO

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
4. No hidden common causes

Denote observed variables by x_i and error variables by e_i and coefficients or connection strengths b_{ij}. Collect them in vectors x and e and a matrix B, respectivelly. Due to the acyclicity assumption, the adjacency matrix B can be permuted to be strictly lower-triangular by a simultaneous row and column permutation. The error variables e_i are independent due to the assumption of no hidden common causes.

Then, mathematically, the model for observed variable vector x is written as

$$ x = Bx + e. $$

[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.

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

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)
```

```
['1.24.4', '2.0.3', '0.20.1', '1.8.3']
```

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)
```

```
x5 <--- x0 (b>0) (100.0%)
x1 <--- x0 (b>0) (100.0%)
x1 <--- x2 (b>0) (100.0%)
x4 <--- x2 (b<0) (100.0%)
x0 <--- x3 (b>0) (98.0%)
x4 <--- x0 (b>0) (98.0%)
x2 <--- x3 (b>0) (96.0%)
x3 <--- x2 (b>0) (4.0%)
```

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]: 84.0%
    x0 <--- x3 (b>0)
    x1 <--- x0 (b>0)
    x1 <--- x2 (b>0)
    x2 <--- x3 (b>0)
    x4 <--- x0 (b>0)
    x4 <--- x2 (b<0)
```

(continues on next page)

(continued from previous page)

```
    x5 <--- x0 (b>0)
DAG[1]: 3.0%
    x0 <--- x3 (b>0)
    x1 <--- x0 (b>0)
    x1 <--- x2 (b>0)
    x3 <--- x2 (b>0)
    x4 <--- x0 (b>0)
    x4 <--- x2 (b<0)
    x5 <--- x0 (b>0)
DAG[2]: 2.0%
    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.   0.03 0.98 0.02 0.  ]
 [1.   0.   1.   0.02 0.   0.01]
 [0.01 0.   0.   0.96 0.   0.01]
 [0.   0.   0.04 0.   0.   0.  ]
 [0.98 0.01 1.   0.02 0.   0.02]
 [1.   0.   0.02 0.02 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()
```

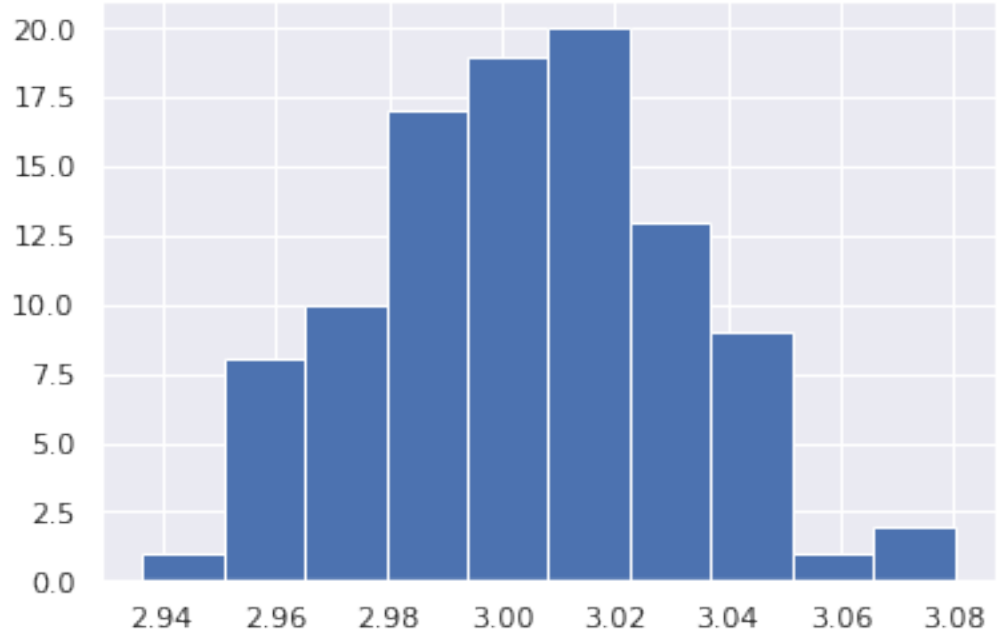
```
df.sort_values('probability', ascending=True).head()
```

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
```

(continues on next page)

(continued from previous page)

```
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`: 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.

```
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

[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.

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.

Denote observed variables of Group g by $x_i^{(g)}$ and error variables by $e_i^{(g)}$ and coefficients or connection strengths $b_{ij}^{(g)}$. Collect them in vectors $x`and : math : `e^{(g)}$ and a matrix $B^{(g)}$, respectivelly. Due to the assumptions of acyclicity and common topological causal orders, the adjacency matrix $B^{(g)}$ can be permuted to be strictly lower-triangular by a simultaneous row and column permutation common to the groups. The error variables $e_i^{(g)}$ are independent due to the assumption of no hidden common causes.

Then, mathematically, the model for observed variable vector $x^{(g)}$ is written as

$$ x^{(g)} = B^{(g)}x^{(g)} + e^{(g)}. $$

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
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)
```

```
['1.24.4', '2.0.3', '0.20.1', '1.8.3']
```

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],
```

(continues on next page)

(continued from previous page)

```
              [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 0x7f87aca0fcd0>
```

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.   ]
 [ 2.962  0.     2.015  0.     0.     0.   ]
 [ 0.     0.     0.     5.97   0.     0.   ]
 [ 0.     0.     0.     0.     0.     0.   ]
 [ 8.01   0.    -1.006  0.     0.     0.   ]
 [ 3.998  0.     0.     0.     0.     0.   ]]
```

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

```
[[ 0.     0.     0.     3.483  0.     0.   ]
 [ 3.526  0.     2.486  0.     0.     0.   ]
 [ 0.     0.     0.     6.503  0.     0.   ]
 [ 0.     0.     0.     0.     0.     0.   ]
 [ 8.478  0.    -1.484  0.     0.     0.   ]
 [ 4.494  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.    0.939 0.07  0.838 0.467 0.818]
 [0.939 0.    0.195 0.755 0.053 0.254]
 [0.07  0.195 0.    0.851 0.374 0.507]
 [0.838 0.755 0.851 0.    0.422 0.755]
```

(continues on next page)

(continued from previous page)

```
 [0.467 0.053 0.374 0.422 0.    0.581]
 [0.818 0.254 0.507 0.755 0.581 0.   ]]
```

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

```
[[0.    0.521 0.973 0.285 0.459 0.606]
 [0.521 0.    0.805 0.785 0.078 0.395]
 [0.973 0.805 0.    0.807 0.378 0.129]
 [0.285 0.785 0.807 0.    0.913 0.99 ]
 [0.459 0.078 0.378 0.913 0.    0.269]
 [0.606 0.395 0.129 0.99  0.269 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%)
x4 <--- x2  (100.0%)
x5 <--- x0  (100.0%)
x1 <--- x3  (4.0%)
```

```
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%)
x5 <--- x0  (100.0%)
x2 <--- x3  (100.0%)
x4 <--- x0  (100.0%)
x4 <--- x2  (100.0%)
x4 <--- x3  (12.0%)
```

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]: 87.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x5 <--- x0
DAG[1]: 4.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x5 <--- x0
    x5 <--- x2
    x5 <--- x3
DAG[2]: 4.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x1 <--- x3
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x5 <--- x0
```

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

```
DAG[0]: 60.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x5 <--- x0
DAG[1]: 8.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x2 <--- x3
```

(continues on next page)

(continued from previous page)

```
    x4 <--- x0
    x4 <--- x2
    x5 <--- x0
    x5 <--- x2
DAG[2]: 7.0%
    x0 <--- x3
    x1 <--- x0
    x1 <--- x2
    x2 <--- x3
    x4 <--- x0
    x4 <--- x2
    x4 <--- x3
    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.   1.   0.   0.  ]
 [1.   0.   1.   0.04 0.   0.02]
 [0.01 0.   0.   1.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.  ]
 [1.   0.   1.   0.02 0.   0.  ]
 [1.   0.   0.04 0.04 0.   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 explanatory 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

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.

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

```
cylinders
```

[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.

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]:

1. Linearity
2. Non-Gaussian continuous error variables (except at most one)
3. Acyclicity of contemporaneous causal relations
4. 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.

Denote observed variables at time point t by $x_i(t)$ and error variables by $e_i(t)$. Collect them in vectors $x(t)$ and $e(t)$, respectivelly. Further, denote by matrices B_τ adjacency matrices with time lag τ.

Due to the acyclicity assumption of contemporaneous causal relations, the coefficient matrix B_0 can be permuted to be strictly lower-triangular by a simultaneous row and column permutation. The error variables $e_i(t)$ are independent due to the assumption of no hidden common causes.

Then, mathematically, the model for observed variable vector $x(t)$ is written as

$$ x(t) = \sum_{ \tau = 0}^k B_{ \tau } x(t - \tau) + e(t).$$

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)
```

```
['1.24.4', '2.0.3', '0.20.1', '1.8.3']
```

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],
```

(continues on next page)

[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.

(continued from previous page)

```
    [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.

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

```
<lingam.var_lingam.VARLiNGAM at 0x7fc1a642d970>
```

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.   , -0.136,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ],
       [-0.484,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 0.075, -0.21 ,  0.   ,  0.   ,  0.   ],
       [ 0.168,  0.   ,  0.   ,  0.   ,  0.   ]])
```

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

```
array([[-0.358,  0.   ,  0.073,  0.302,  0.   ],
       [ 0.   , -0.338, -0.154, -0.335,  0.423],
       [ 0.   ,  0.   ,  0.424,  0.112,  0.493],
       [-0.386, -0.1  , -0.266,  0.   , -0.159],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ]])
```

```
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],
```

(continues on next page)

(continued from previous page)

```
       [-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.   ],
       [-0.456,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 0.   , -0.22 ,  0.   ,  0.   ,  0.   ],
       [ 0.157,  0.   ,  0.   ,  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.    0.127 0.104 0.042 0.746]
 [0.127 0.    0.086 0.874 0.739]
 [0.104 0.086 0.    0.404 0.136]
 [0.042 0.874 0.404 0.    0.763]
 [0.746 0.739 0.136 0.763 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)
```

```
x2(t) <--- x4(t-1) (b>0) (100.0%)
x2(t) <--- x2(t-1) (b>0) (100.0%)
x0(t) <--- x0(t-1) (b<0) (95.0%)
x1(t) <--- x1(t-1) (b<0) (86.0%)
x1(t) <--- x4(t-1) (b>0) (85.0%)
x3(t) <--- x0(t-1) (b<0) (78.0%)
x2(t) <--- x4(t) (b<0) (60.0%)
x0(t) <--- x3(t-1) (b>0) (48.0%)
```

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]: 5.0%
    x0(t) <--- x0(t-1) (b<0)
    x0(t) <--- x3(t-1) (b>0)
    x1(t) <--- x0(t) (b<0)
    x1(t) <--- x0(t-1) (b<0)
    x1(t) <--- x1(t-1) (b<0)
    x1(t) <--- x4(t-1) (b>0)
    x2(t) <--- x0(t) (b<0)
    x2(t) <--- x4(t) (b<0)
    x2(t) <--- x2(t-1) (b>0)
    x2(t) <--- x4(t-1) (b>0)
    x3(t) <--- x0(t) (b>0)
    x3(t) <--- x0(t-1) (b<0)
    x3(t) <--- x2(t-1) (b<0)
    x3(t) <--- x4(t-1) (b<0)
DAG[1]: 5.0%
```

(continues on next page)

(continued from previous page)

```
    x0(t) <--- x0(t-1) (b<0)
    x0(t) <--- x3(t-1) (b>0)
    x1(t) <--- x0(t) (b<0)
    x1(t) <--- x2(t) (b>0)
    x1(t) <--- x0(t-1) (b<0)
    x1(t) <--- x1(t-1) (b<0)
    x1(t) <--- x4(t-1) (b>0)
    x2(t) <--- x0(t) (b<0)
    x2(t) <--- x4(t) (b<0)
    x2(t) <--- x2(t-1) (b>0)
    x2(t) <--- x4(t-1) (b>0)
    x3(t) <--- x0(t) (b>0)
    x3(t) <--- x0(t-1) (b<0)
    x3(t) <--- x2(t-1) (b<0)
DAG[2]: 5.0%
    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) <--- x1(t) (b>0)
    x2(t) <--- x3(t) (b>0)
    x2(t) <--- x0(t-1) (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)
```

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.   0.6  0.04 0.06 0.14]
 [0.39 0.   0.25 0.18 0.16]
 [0.65 0.68 0.   0.67 0.84]
 [0.51 0.6  0.07 0.   0.66]
 [0.35 0.28 0.01 0.09 0.  ]]
Probability of B1:
 [[1.   0.   0.3  1.   0.02]
 [0.56 1.   0.94 0.67 1.  ]
 [0.8  0.02 1.   0.25 1.  ]
 [1.   0.24 1.   0.08 1.  ]
 [0.02 0.   0.03 0.07 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.

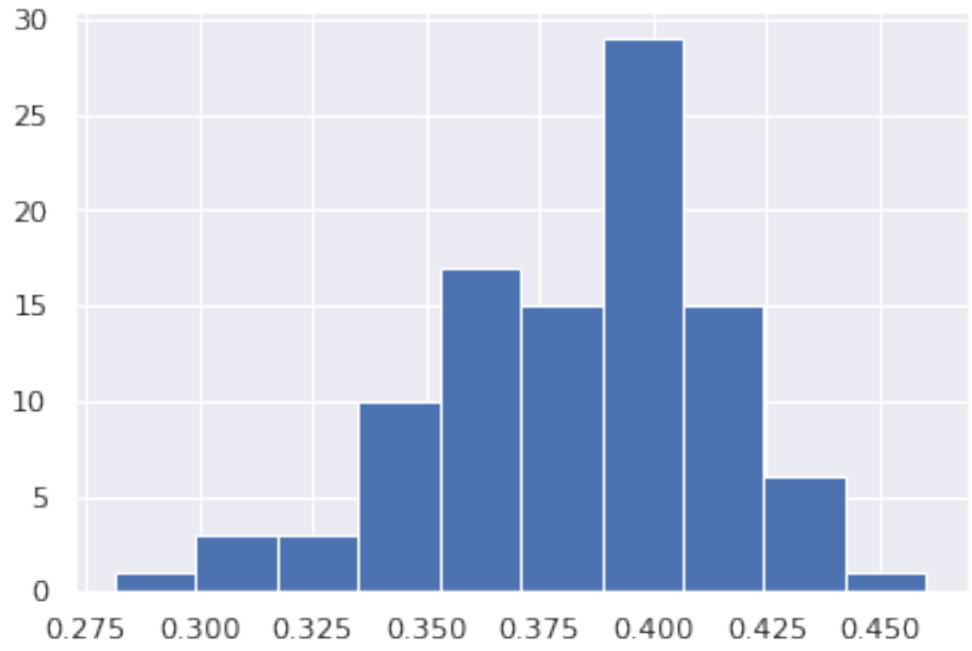
```
df.sort_values('effect', ascending=False).head()
```

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[Page 28, 1]:

1. Linearity
2. Non-Gaussian continuous error variables (except at most one)
3. Acyclicity of contemporaneous causal relations
4. No hidden common causes between contempraneous error variables

Denote observed variables at time point t by $x_i(t)$ and error variables by $e_i(t)$. Collect them in vectors $x(t)$ and $e(t)$, respectivelly. Further, denote by matrices B_τ and Ω_ω coefficient matrices with time lags τ and ω, respectivelly.

Due to the acyclicity assumption of contemporaneous causal relations, the adjacency matrix B_0 can be permuted to be strictly lower-triangular by a simultaneous row and column permutation. The error variables $e_i(t)$ are independent due to the assumption of no hidden common causes.

Then, mathematically, the model for observed variable vector $x(t)$ is written as

$$ x(t) = \sum_{ \tau = 0}^k B_{ \tau } x(t - \tau) + e(t) - \sum_{ \omega = 1}^{\ell} \Omega_{ \omega } e(t- \omega).$$

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

2.8.2 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
import lingam
from lingam.utils import make_dot, print_causal_directions, print_dagc

import warnings
warnings.filterwarnings('ignore')
```

(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)

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

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

```
['1.24.4', '2.0.3', '0.20.1', '1.8.3']
```

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.   , -0.194,  0.   ,  0.   ],
       [-0.354,  0.   ,  0.191,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 0.558, -0.228,  0.   ,  0.   ,  0.   ],
       [ 0.115,  0.   ,  0.   ,  0.   ,  0.   ]])
```

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

```
array([[ 0.   , -0.394, -0.509,  0.   ,  0.659],
       [-0.3  ,  0.   ,  0.   ,  0.211,  0.404],
       [-0.281,  0.21 ,  0.   ,  0.118,  0.25 ],
       [ 0.082,  0.762,  0.178,  0.137, -0.819],
       [-0.507,  0.   , -0.278,  0.   ,  0.336]])
```

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

```
array([[ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 0.209,  0.365,  0.   ,  0.   ,  0.531],
       [ 0.   , -0.579, -0.105, -0.298, -0.235],
       [ 0.   ,  0.171,  0.414,  0.302,  0.   ],
       [ 0.297,  0.435,  0.482,  0.376, -0.438]])
```

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.   ],
       [ 0.587, -0.209,  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.    0.622 0.388 0.    0.539]
 [0.622 0.    0.087 0.469 0.069]
 [0.388 0.087 0.    0.248 0.229]
 [0.    0.469 0.248 0.    0.021]
 [0.539 0.069 0.229 0.021 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)
```

```
y3(t) <--- y4(t-1) (b<0) (98.0%)
y3(t) <--- y1(t-1) (b>0) (98.0%)
y0(t) <--- y4(t-1) (b>0) (96.0%)
y1(t) <--- e4(t-1) (b>0) (91.0%)
y2(t) <--- e1(t-1) (b<0) (80.0%)
y4(t) <--- e2(t-1) (b>0) (71.0%)
y1(t) <--- e0(t-1) (b>0) (64.0%)
y2(t) <--- e4(t-1) (b<0) (62.0%)
```

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]: 1.0%
    y0(t) <--- y1(t) (b<0)
    y0(t) <--- y1(t-1) (b<0)
    y0(t) <--- y2(t-1) (b<0)
    y0(t) <--- y4(t-1) (b>0)
    y0(t) <--- e0(t-1) (b>0)
    y0(t) <--- e1(t-1) (b<0)
    y0(t) <--- e3(t-1) (b>0)
    y0(t) <--- e4(t-1) (b>0)
    y1(t) <--- y2(t) (b>0)
    y1(t) <--- y2(t-1) (b>0)
    y1(t) <--- y3(t-1) (b>0)
    y1(t) <--- y4(t-1) (b<0)
    y1(t) <--- e0(t-1) (b>0)
    y1(t) <--- e2(t-1) (b>0)
    y1(t) <--- e3(t-1) (b>0)
    y1(t) <--- e4(t-1) (b>0)
    y2(t) <--- y4(t-1) (b>0)
    y2(t) <--- e0(t-1) (b<0)
    y2(t) <--- e2(t-1) (b<0)
    y2(t) <--- e3(t-1) (b<0)
    y2(t) <--- e4(t-1) (b<0)
    y3(t) <--- y1(t) (b<0)
    y3(t) <--- y2(t) (b>0)
    y3(t) <--- y1(t-1) (b>0)
    y3(t) <--- y2(t-1) (b>0)
    y3(t) <--- y3(t-1) (b>0)
    y3(t) <--- y4(t-1) (b<0)
    y3(t) <--- e0(t-1) (b>0)
    y3(t) <--- e2(t-1) (b>0)
    y3(t) <--- e3(t-1) (b>0)
    y3(t) <--- e4(t-1) (b>0)
    y4(t) <--- y0(t) (b>0)
    y4(t) <--- y1(t) (b>0)
    y4(t) <--- y3(t) (b>0)
    y4(t) <--- y1(t-1) (b>0)
    y4(t) <--- y2(t-1) (b>0)
    y4(t) <--- y3(t-1) (b>0)
    y4(t) <--- y4(t-1) (b<0)
    y4(t) <--- e0(t-1) (b<0)
```

(continues on next page)

(continued from previous page)

```
    y4(t) <--- e1(t-1) (b>0)
    y4(t) <--- e2(t-1) (b>0)
    y4(t) <--- e3(t-1) (b<0)
    y4(t) <--- e4(t-1) (b<0)
DAG[1]: 1.0%
    y0(t) <--- y1(t-1) (b<0)
    y0(t) <--- y2(t-1) (b<0)
    y0(t) <--- y4(t-1) (b>0)
    y0(t) <--- e0(t-1) (b>0)
    y1(t) <--- y3(t) (b<0)
    y1(t) <--- y0(t-1) (b<0)
    y1(t) <--- y1(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) <--- y1(t) (b>0)
    y2(t) <--- y3(t) (b>0)
    y2(t) <--- y4(t-1) (b>0)
    y2(t) <--- e0(t-1) (b<0)
    y2(t) <--- e1(t-1) (b<0)
    y2(t) <--- e3(t-1) (b>0)
    y2(t) <--- e4(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) <--- e0(t-1) (b<0)
    y3(t) <--- e1(t-1) (b>0)
    y3(t) <--- e2(t-1) (b>0)
    y4(t) <--- y0(t) (b>0)
    y4(t) <--- y0(t-1) (b<0)
    y4(t) <--- y1(t-1) (b>0)
    y4(t) <--- y4(t-1) (b<0)
    y4(t) <--- e0(t-1) (b<0)
    y4(t) <--- e1(t-1) (b>0)
    y4(t) <--- e2(t-1) (b>0)
DAG[2]: 1.0%
    y0(t) <--- y1(t-1) (b<0)
    y0(t) <--- y2(t-1) (b<0)
    y0(t) <--- y4(t-1) (b>0)
    y0(t) <--- e0(t-1) (b>0)
    y1(t) <--- y0(t) (b<0)
    y1(t) <--- y2(t) (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) <--- e2(t-1) (b>0)
    y1(t) <--- e3(t-1) (b>0)
    y1(t) <--- e4(t-1) (b>0)
    y2(t) <--- y0(t) (b<0)
    y2(t) <--- y0(t-1) (b<0)
    y2(t) <--- y4(t-1) (b>0)
    y2(t) <--- e1(t-1) (b<0)
```

(continues on next page)

(continued from previous page)

```
    y2(t) <--- e2(t-1) (b<0)
    y2(t) <--- e3(t-1) (b<0)
    y2(t) <--- e4(t-1) (b<0)
    y3(t) <--- y1(t) (b<0)
    y3(t) <--- y2(t) (b>0)
    y3(t) <--- y1(t-1) (b>0)
    y3(t) <--- y2(t-1) (b>0)
    y3(t) <--- y3(t-1) (b>0)
    y3(t) <--- y4(t-1) (b<0)
    y3(t) <--- e1(t-1) (b>0)
    y3(t) <--- e2(t-1) (b>0)
    y3(t) <--- e3(t-1) (b>0)
    y3(t) <--- e4(t-1) (b>0)
    y4(t) <--- y0(t) (b>0)
    y4(t) <--- y1(t) (b>0)
    y4(t) <--- y3(t) (b>0)
    y4(t) <--- y1(t-1) (b>0)
    y4(t) <--- y2(t-1) (b>0)
    y4(t) <--- y4(t-1) (b<0)
    y4(t) <--- e0(t-1) (b<0)
    y4(t) <--- e2(t-1) (b>0)
    y4(t) <--- e4(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.   0.26 0.46 0.26 0.3 ]
 [0.53 0.   0.54 0.37 0.33]
 [0.22 0.41 0.   0.38 0.12]
 [0.6  0.62 0.35 0.   0.38]
 [0.69 0.28 0.27 0.43 0.  ]]
Probability of psi1:
 [[0.41 1.   1.   0.2  1.  ]
 [0.64 0.63 0.6  0.83 0.85]
 [0.76 0.69 0.64 0.47 0.95]
 [0.55 1.   0.71 0.73 1.  ]
 [0.94 0.79 0.71 0.53 0.74]]
Probability of omega1:
 [[0.76 0.35 0.54 0.43 0.47]
 [0.87 0.77 0.63 0.5  1.  ]
 [0.63 0.95 0.66 0.84 0.93]
 [0.41 0.85 0.88 0.68 0.49]
 [0.66 0.81 0.92 0.58 0.69]]
```

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.

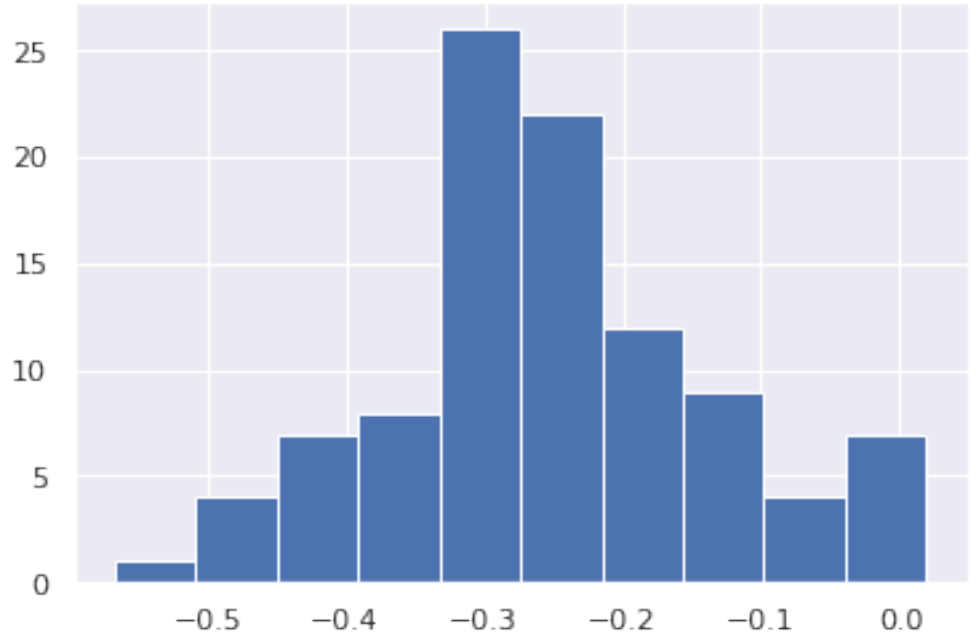
```
df.sort_values('effect', ascending=False).head()
```

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:

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

Denote observed variables and error variables of m-the sample at time point m by $x_i^m(t)$ and $e_i^m(t)$. Collect them from variables in vectors $x^m(t)$ and $e^m(t)$. Further, collect them from samples in matrices $X(t)$ and $E(t)$. Further, denote by $B(t, t-\tau)$ adjacency matrices with time lag τ.

Due to the assumptions of acyclicity, the adjacency matrix $B(t, t)$ can be permuted to be strictly lower-triangular by a simultaneous row and column permutation. The error variables $e_i^m(t)$ are independent due to the assumption of no hidden common causes.

Then, mathematically, the model for observed variable matrix $X(t)$ is written as

$$ X(t) = \sum_{ \tau = 0}^k B (t, t-\tau) X(t - \tau) + E(t).$$

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)
```

```
['1.24.4', '2.0.3', '0.20.1', '1.8.3']
```

[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.

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-) 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, :]);
```

(continues on next page)

(continued from previous page)

```
        ext[i, :] = ext[i, :] / np.std(ext[i, :]);

    # create B(t,t-)
    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-) 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-) 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])
```

(continues on next page)

(continued from previous page)

```
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])
```

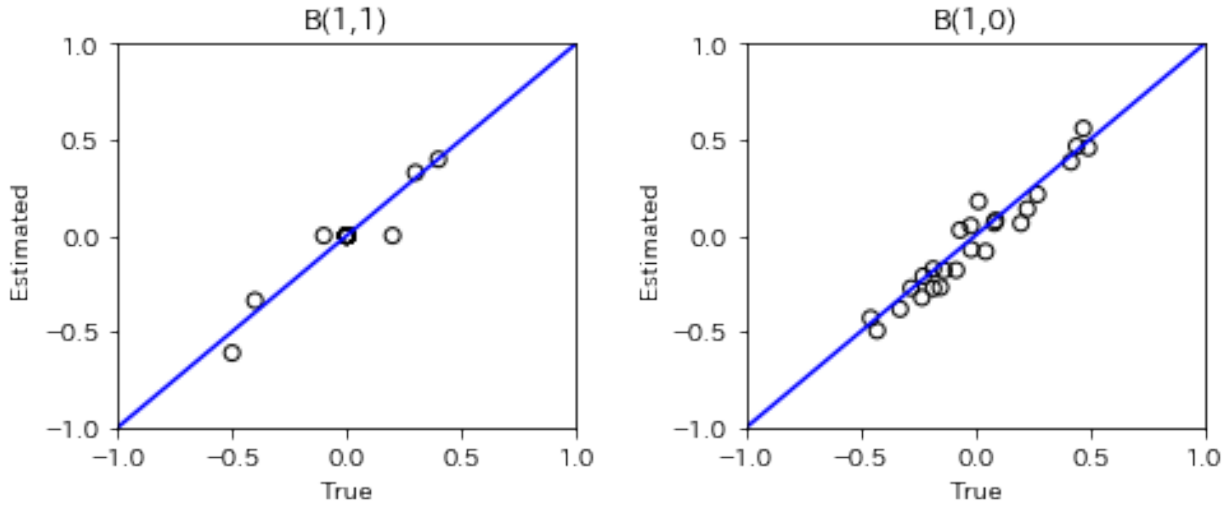
```
B(0,0):
[[nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]]
B(0,-1):
[[nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]
 [nan nan nan nan nan]]
B(1,1):
[[ 0.     0.     0.     0.    -0.611]
 [ 0.     0.     0.     0.398  0.   ]
 [ 0.     0.328  0.     0.     0.   ]
 [ 0.     0.     0.     0.     0.   ]
 [ 0.    -0.338  0.     0.     0.   ]]
B(1,0):
[[ 0.029  0.064 -0.27   0.065 -0.18 ]
 [ 0.139 -0.211 -0.43   0.558  0.051]
 [-0.181  0.178  0.466  0.214  0.079]
 [ 0.384 -0.083 -0.495 -0.072 -0.323]
 [-0.174 -0.383 -0.274 -0.275  0.457]]
B(2,2):
[[ 0.     0.     0.     0.     0.   ]
 [ 0.     0.    -0.696  0.     0.487]
 [ 0.231  0.     0.     0.     0.   ]
 [ 0.     0.    -0.409  0.     0.   ]
 [ 0.25   0.     0.     0.     0.   ]]
B(2,1):
[[ 0.194  0.2    0.031 -0.473 -0.002]
 [-0.376 -0.038  0.16   0.261  0.102]
 [ 0.117  0.266 -0.05   0.523 -0.019]
 [ 0.249 -0.448  0.473 -0.001 -0.177]
 [-0.177  0.309 -0.112  0.295 -0.273]]
```

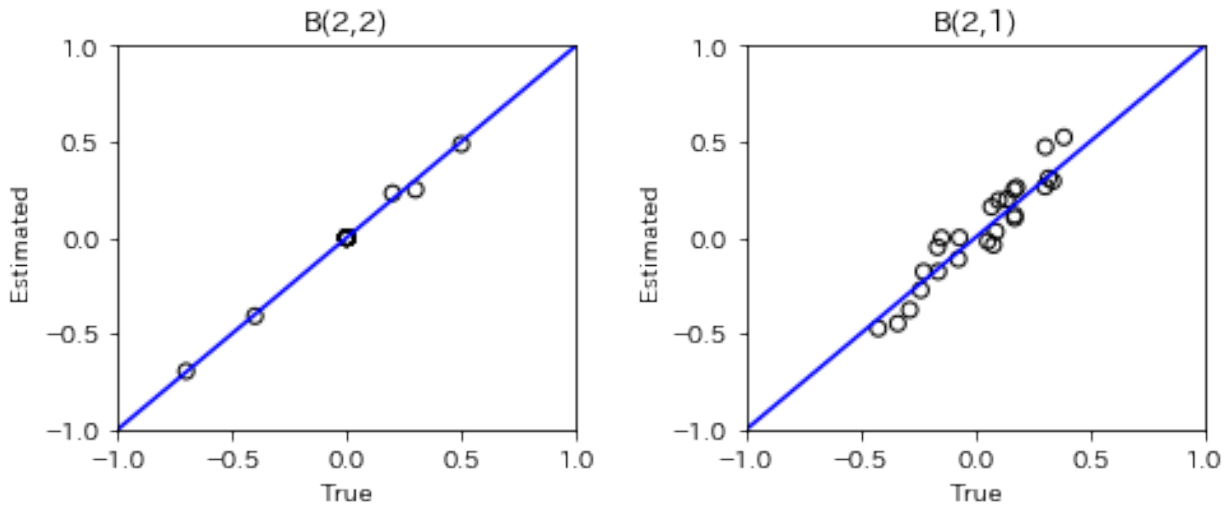
```
for t in range(1, n_timepoints):
    B_t, B_tau = model.adjacency_matrices_[t]
    plt.figure(figsize=(7, 3))

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

(continues on next page)

(continued from previous page)

```
    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.    0.026 0.064 0.289 0.051]
 [0.026 0.    0.363 0.821 0.581]
 [0.064 0.363 0.    0.067 0.098]
 [0.289 0.821 0.067 0.    0.059]
 [0.051 0.581 0.098 0.059 0.   ]]
```

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

```
[[0.    0.715 0.719 0.593 0.564]
 [0.715 0.    0.78  0.7   0.504]
 [0.719 0.78  0.    0.532 0.591]
 [0.593 0.7   0.532 0.    0.401]
 [0.564 0.504 0.591 0.401 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)
```

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

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) <--- x1(1) (b>0)
    x0(1) <--- x3(1) (b>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) <--- x3(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]: 2.0%
    x0(1) <--- x1(1) (b>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)
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
    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]: 5.0%
    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)
```

(continues on next page)

(continued from previous page)

```
    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) <--- 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.   0.37 0.1  0.12 1.  ]
  [0.   0.   0.   1.   0.  ]
  [0.   0.98 0.   0.5  0.24]
  [0.   0.   0.   0.   0.  ]
  [0.   1.   0.11 0.28 0.  ]]

 [[0.91 0.93 1.   0.94 0.97]
  [0.99 0.99 1.   1.   0.94]
  [1.   1.   1.   0.99 0.84]
  [1.   0.98 1.   0.92 1.  ]
  [0.98 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.   0.37 0.1  0.12 1.  ]
 [0.   0.   0.   1.   0.  ]
 [0.   0.98 0.   0.5  0.24]
 [0.   0.   0.   0.   0.  ]
 [0.   1.   0.11 0.28 0.  ]]
B(1,0):
[[0.91 0.93 1.   0.94 0.97]
 [0.99 0.99 1.   1.   0.94]
 [1.   1.   1.   0.99 0.84]
 [1.   0.98 1.   0.92 1.  ]
 [0.98 1.   1.   1.   1.  ]]
B(2,2):
[[0.   0.   0.   0.   0.  ]
 [0.06 0.   1.   0.04 1.  ]
 [0.8  0.   0.   0.   0.07]
 [0.03 0.02 1.   0.   0.1 ]
 [0.91 0.   0.   0.01 0.  ]]
B(2,1):
[[1.   1.   0.91 1.   0.92]
 [1.   0.86 1.   1.   0.96]
 [0.95 1.   0.95 1.   0.82]
```

(continues on next page)

(continued from previous page)

```
 [1.   1.   1.   0.92 0.99]
 [1.   1.   0.97 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.

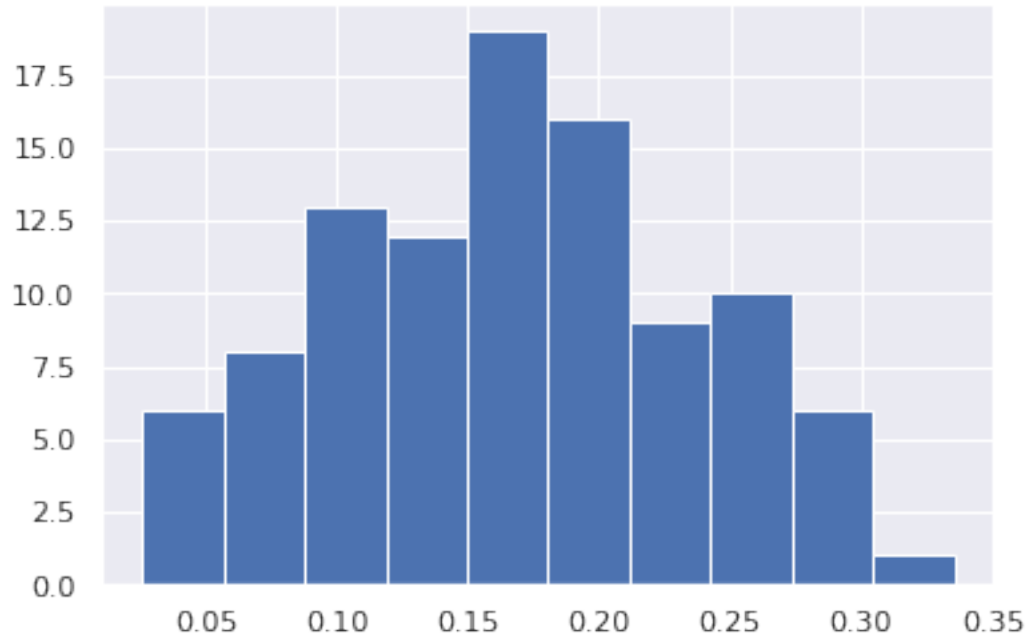
```
df.sort_values('effect', ascending=False).head()
```

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[Page 49, 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].

Denote observed variables by x_i and error variables by e_i and coefficients or connection strengths b_{ij}. Collect them in vectors x and e and a matrix B, respectivelly. Due to the acyclicity assumption, the adjacency matrix B can be permuted to be strictly lower-triangular by a simultaneous row and column permutation. The error variables e_i except those

[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.

corresponding to exogenous observed variables are independent due to the assumption that only exogenous observed variables may share hidden common causes.

Then, mathematically, the model for observed variable vector x is written as

$$ x = Bx + e. $$

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)
```

```
['1.24.4', '2.0.3', '0.20.1', '1.8.3']
```

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],
```

(continues on next page)

(continued from previous page)

```
              [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 0x7fb69052ca60>
```

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.   ,  0.511,  0.   ,  0.   ],
       [ 0.504,  0.   ,  0.499,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,    nan,  0.   ,  0.   ],
       [ 0.   ,  0.   ,    nan,  0.   ,  0.   ,  0.   ],
       [ 0.481,  0.   , -0.473,  0.   ,  0.   ,  0.   ],
       [ 0.519,  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.    0.523   nan   nan 0.8   0.399]
 [0.523 0.      nan   nan 0.44  0.6  ]
 [  nan   nan 0.      nan   nan   nan]
 [  nan   nan   nan 0.      nan   nan]
 [0.8   0.44    nan   nan 0.    0.446]
 [0.399 0.6     nan   nan 0.446 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%)
x0 <--- x3 (b>0) (12.0%)
x1 <--- x4 (b>0) (6.0%)
x4 <--- x1 (b>0) (4.0%)
```

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]: 12.0%
    x4 <--- x0 (b>0)
    x4 <--- x2 (b<0)
DAG[2]: 10.0%
    x0 <--- x3 (b>0)
    x1 <--- x0 (b>0)
    x1 <--- x2 (b>0)
    x4 <--- x0 (b>0)
    x4 <--- x2 (b<0)
    x5 <--- x0 (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.   0.06 0.  ]
 [0.   0.   0.   0.02 0.   0.  ]
 [0.   0.   0.   0.   0.   0.  ]
 [0.45 0.04 0.45 0.02 0.   0.01]
 [0.26 0.   0.   0.   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])
```

(continues on next page)

(continued from previous page)

```
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()
```

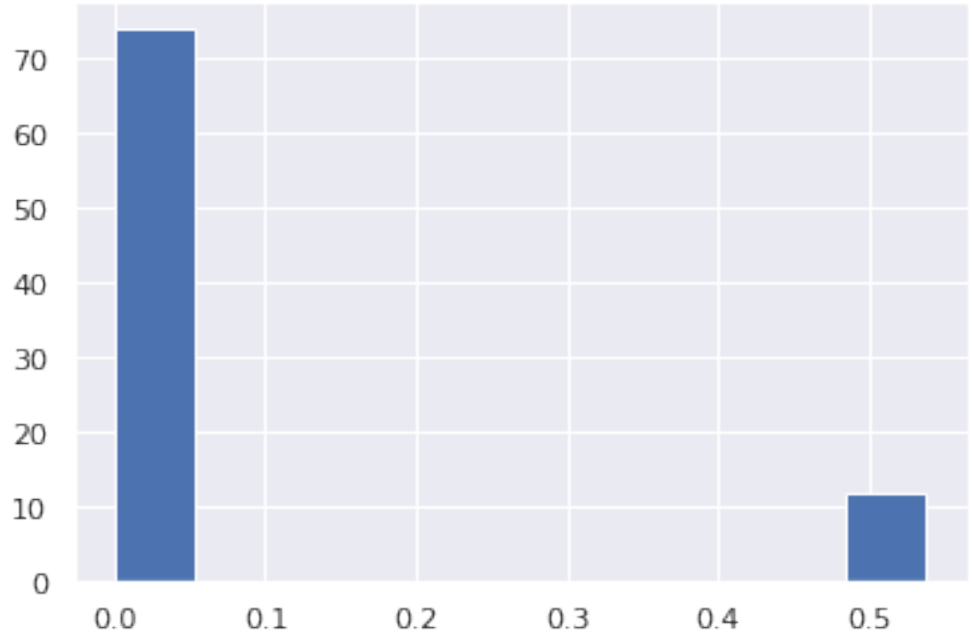
```
df.sort_values('probability', ascending=True).head()
```

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`: 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.

```
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],
)

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)[3] 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[Page 57, 1], this method makes the following assumptions:

[3] 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.

[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.

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)
```

```
['1.24.4', '2.0.3', '0.20.1', '1.8.3']
```

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.

```
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 0x7f76b338e8b0>
```

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],
```

(continues on next page)

(continued from previous page)

```
       [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)
```

```
x4 <--- x0 (b>0) (58.0%)
x0 <--- x5 (b>0) (51.0%)
x2 <--- x0 (b>0) (30.0%)
x3 <--- x5 (b>0) (30.0%)
x0 <--- x1 (b>0) (22.0%)
x0 <--- x3 (b>0) (18.0%)
x1 <--- x5 (b>0) (18.0%)
x2 <--- x5 (b>0) (15.0%)
```

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]: 6.0%
DAG[1]: 4.0%
    x4 <--- x0 (b>0)
DAG[2]: 4.0%
    x2 <--- x0 (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.   0.22 0.   0.18 0.   0.51]
 [0.   0.   0.   0.   0.   0.18]
 [0.3  0.13 0.   0.12 0.05 0.15]
 [0.   0.   0.   0.   0.   0.3 ]
 [0.58 0.11 0.02 0.11 0.   0.05]
 [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
```

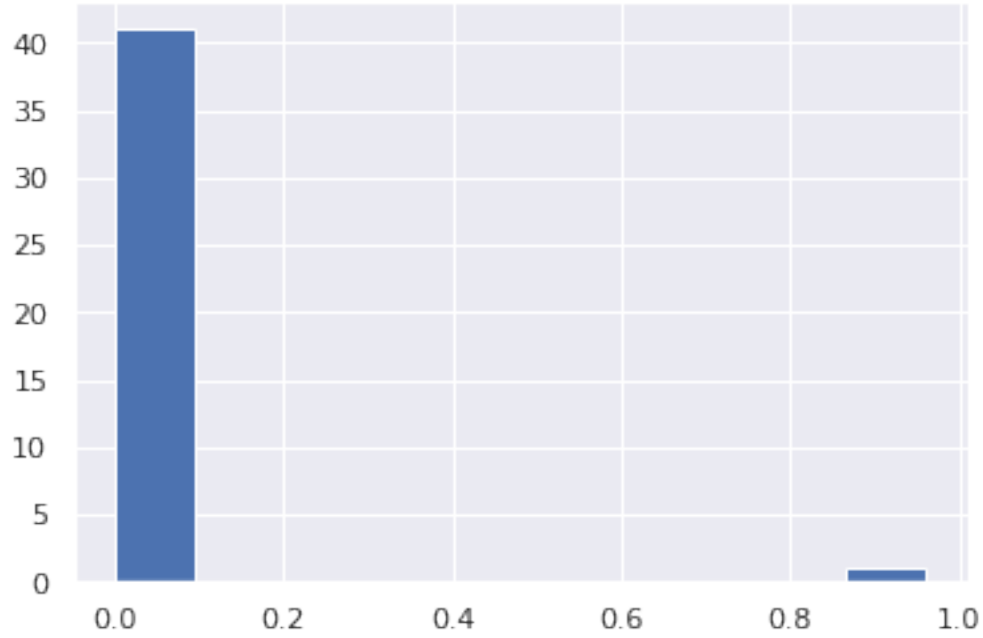
We can easily perform sorting operations with pandas.DataFrame.

```
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()

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_)
```

2.13.5 Highlight paths between specified nodes

make_dot highlights the path specified by the path argument.

```
make_dot(model.adjacency_matrix_, path=(3, 1))
```

If detect_cycles is True, simple cycles are displayed with a dashed edge.

```
result = model.adjacency_matrix_.copy()
result[0, 1] = 100
result[3, 1] = 100

make_dot(result, path=(3, 1), path_color="red", detect_cycle=True)
```

2.13.6 Draw the result of LiNGAM with emphasis on descendants and ancestors

`make_dot_highlight` highlights descendants or ancestors of the graph.

The first argument is the result and the second argument is the index of the target variable. There are four types of cluster names: target, ancestor, descendant, and others. target contains only the node specified in the second argument. Nodes that are ancestors or descendants of target belong to ancestor or descendant. The number appended to the cluster name is the distance from target. Other nodes belong to others.

```
make_dot_highlight(model.adjacency_matrix_, 0)
```

It is also possible to disable the display of clusters of ancestors and descendants.

```
make_dot_highlight(model.adjacency_matrix_, 0, max_dsc=0, max_anc=None)
```

It is also possible to suppress the display of the others cluster.

```
make_dot_highlight(model.adjacency_matrix_, 0, max_dsc=0, max_anc=None, draw_
↪others=False)
```

2.13.7 Draw the result of Bootstrap with emphasis on descendants and ancestors

It is possible to visualize results that include the cyclic portion, such as the result of a bootstrap.

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

```
median = np.median(result.adjacency_matrices_, axis=0)
make_dot(median, lower_limit=0)
```

Applying `make_dot_highlight` to this graph draws the following graph. Dashed edges indicate simple cycles.

```
make_dot_highlight(median, 0, detect_cycle=True)
```

You can reduce the edges by setting lower_limit.

```
make_dot_highlight(median, 0, detect_cycle=True, lower_limit=0.1)
```

You can also set the color map and the spacing of the nodes.

```
make_dot_highlight(median, 0, lower_limit=0.001, cmap="cool", vmargin=3, hmargin=3)
```

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.

This method is based on the LiNA model as shown below,

$$f^{(m)} = B^{(m)} f^{(m)} + \varepsilon^{(m)},$$
$$x^{(m)} = G^{(m)} f^{(m)} + e^{(m)},$$

where $\varepsilon^{(m)}$ and $e^{(m)}$ are random vectors that collect external influences, and errors, respectively, and they are independent with each other. $f^{(m)}$ and $x^{(m)}$ are random vectors that collect latent factors, and observed data, respectively. $B^{(m)}$ is a matrix that collects causal effects $b_{ij}^{(m)}$ between $f_i^{(m)}$ and $f_j^{(m)}$, while $G^{(m)}$ collects factor loadings $g_{ij}^{(m)}$ between $f_j^{(m)}$ and $x_i^{(m)}$. m stands for the m^{th} domain.

This method makes the following assumptions.

1. Linearity
2. Acyclicity
3. No causal relations between observed variables

[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.

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, :])
    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))
```

(continues on next page)

(continued from previous page)

```
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])
```

(continues on next page)

(continued from previous page)

```
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])
    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

Denote observed variables by x_i and error variables by e_i. The error variables e_i are independent due to the assumption of no hidden common causes. Then, mathematically, the model for observed variables x_i is written as

$$ x_i = f_i (pa(x_i))+e_i, $$

where f_i are some nonlinear (differentiable) functions and $pa(x_i)$ are the parents of x_i.

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__])
```

(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)

```
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],
    [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)
    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 is based on the LiM model as shown below,

i) As for the continuous variable, its value assigned to each of x_i is a linear function of its parent variables denoted by $x_{\mathrm{pa}(i)}$ plus a non-Gaussian error term e_i, that is,

$$x_i = e_i + c_i + \sum_{j \in \mathrm{pa}(i)} b_{ij} x_j, \quad e_i \sim Non - Gaussian(\cdot),$$

where the error terms e_i are continuous random variables with non-Gaussian densities, and the error variables e_i are independent of each other. The coefficients b_{ij} and intercepts c_i are constants.

ii) As for the discrete variable, its value equals 1 if the linear function of its parent variables $x_{\mathrm{pa}(i)}$ plus a Logistic error term e_i is larger than 0, otherwise, its value equals 0. That is,

$$x_i = \begin{cases} 1, & e_i + c_i + \sum_{j \in \mathrm{pa}(i)} b_{ij} x_j > 0 \\ 0, & \text{otherwise} \end{cases}, \quad e_i \sim Logistic(0, 1), \tag{2.1}$$

where the error terms e_i follow the Logistic distribution, while the other notations are identical to those in continuous variables.

This method makes the following assumptions.

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

[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.

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, only_global=True)
```

```
<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)[2] assumes an extension of the basic CAM model[1] to include unobserved variables. This method makes the following assumptions:

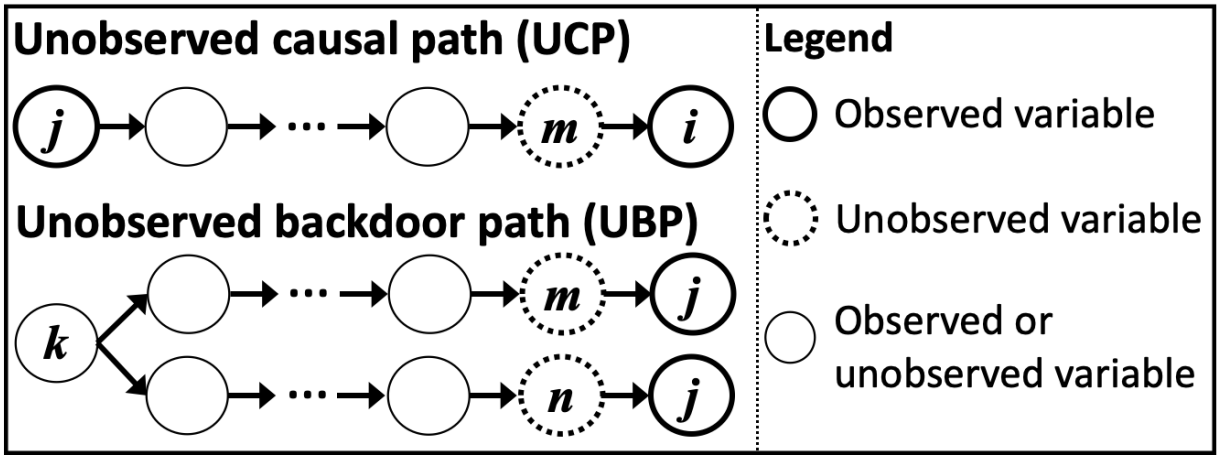
1. The effects of direct causes on a variable form generalized additive models (GAMs).
2. The causal structures form directed acyclic graphs (DAGs).

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.

[2] T. N. Maeda and S. Shimizu. Causal additive models with unobserved variables. In Proc. 27th Conference on Uncertainty in Artificial Intelligence (UAI2021), PMLR 161:97-106, 2021.

[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

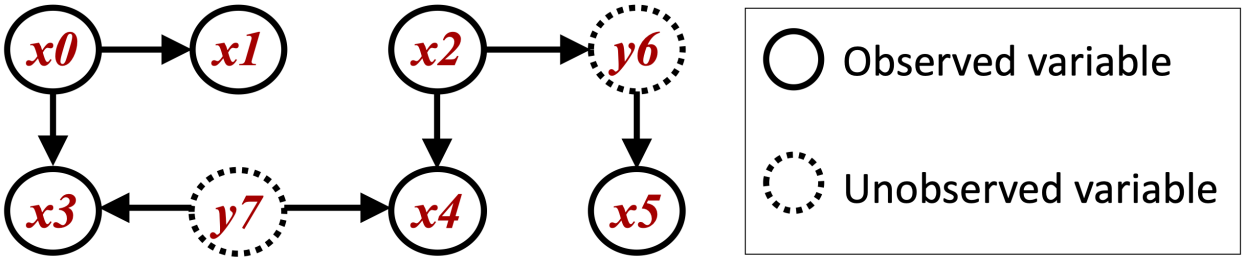
2.17.2 Import and settings

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)
```

(continues on next page)

(continued from previous page)

```
    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]]

    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
```

(continues on next page)

(continued from previous page)

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

2.18 MultiGroupRCD

2.18.1 Import and settings

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

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

2.18.2 Test data

We generate two datasets consisting of 6 variables and 1 latent variable.

```
def get_coef():
    coef = random.random()
    return coef if coef >= 0.5 else coef - 1.0
get_external_effect = lambda n: np.random.normal(0.0, 0.5, n) ** 3
```

```
B1 = np.array([[       0.0,        0.0,        0.0,        0.0,        0.0, get_coef(), ␣
↪      0.0],
               [       0.0,        0.0,        0.0,        0.0,        0.0, get_coef(), ␣
↪      0.0],
               [get_coef(), get_coef(),        0.0,        0.0,        0.0,        0.0, ␣
↪      0.0],
               [       0.0,        0.0, get_coef(),        0.0,        0.0,        0.0,␣
↪get_coef()],
               [       0.0,        0.0, get_coef(),        0.0,        0.0,        0.0,␣
↪get_coef()],
               [       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]])

samples = 1000
x5 = get_external_effect(samples)
x6 = get_external_effect(samples)
x0 = x5 * B1[0, 5] + get_external_effect(samples)
x1 = x5 * B1[1, 5] + get_external_effect(samples)
x2 = x0 * B1[2, 0] + x1 * B1[2, 1] + get_external_effect(samples)
x3 = x2 * B1[3, 2] + x6 * B1[3, 6] + get_external_effect(samples)
x4 = x2 * B1[4, 2] + x6 * B1[4, 6] + get_external_effect(samples)

# x5, x6 is a latent variable.
X1 = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T ,columns=['x0', 'x1', 'x2', 'x3',
↪'x4', 'x5'])
```

```
make_dot(B1, labels=['x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'f1(x6)'])
```

```
B2 = np.array([[       0.0,        0.0,        0.0,        0.0,        0.0, get_coef(), ␣
↪      0.0],
               [       0.0,        0.0,        0.0,        0.0,        0.0, get_coef(), ␣
↪      0.0],
               [get_coef(), get_coef(),        0.0,        0.0,        0.0,        0.0, ␣
↪      0.0],
               [       0.0,        0.0, get_coef(),        0.0,        0.0,        0.0,␣
↪get_coef()],
               [       0.0,        0.0, get_coef(),        0.0,        0.0,        0.0,␣
↪get_coef()],
               [       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]])

samples = 1000
x5 = get_external_effect(samples)
x6 = get_external_effect(samples)
x0 = x5 * B2[0, 5] + get_external_effect(samples)
x1 = x5 * B2[1, 5] + get_external_effect(samples)
x2 = x0 * B2[2, 0] + x1 * B2[2, 1] + get_external_effect(samples)
x3 = x2 * B2[3, 2] + x6 * B2[3, 6] + get_external_effect(samples)
```

(continues on next page)

(continued from previous page)

```
x4 = x2 * B2[4, 2] + x6 * B2[4, 6] + get_external_effect(samples)

# x5, x6 is a latent variable.
X2 = pd.DataFrame(np.array([x0, x1, x2, x3, x4, x5]).T ,columns=['x0', 'x1', 'x2', 'x3',
↪'x4', 'x5'])
```

```
make_dot(B2, labels=['x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'f1(x6)'])
```

We create a list variable that contains two datasets.

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

2.18.3 Causal Discovery

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

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

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={5}
M1={5}
M2={0, 1, 5}
M3={0, 1, 2, 5}
M4={0, 1, 2, 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.

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

```
[[ 0.     0.     0.     0.     0.    -0.92 ]
 [ 0.     0.     0.     0.     0.    -0.528]
 [-0.701 -0.821  0.     0.     0.     0.   ]
 [ 0.     0.    -0.891  0.       nan  0.   ]
 [ 0.     0.    -0.664    nan  0.     0.   ]
 [ 0.     0.     0.     0.     0.     0.   ]]
```

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

```
[[ 0.     0.     0.     0.     0.     0.778]
 [ 0.     0.     0.     0.     0.     0.528]
 [ 0.755  0.621  0.     0.     0.     0.   ]
 [ 0.     0.     1.023  0.       nan  0.   ]
 [ 0.     0.    -0.996    nan  0.     0.   ]
 [ 0.     0.     0.     0.     0.     0.   ]]
```

To compare, we run *MultiGroupRCD* with single dataset concatenating two datasets. You can see that the causal structure cannot be estimated correctly for a single dataset.

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

model_all = lingam.RCD()
model_all.fit(X_all)

ancestors_list = model.ancestors_list_

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

make_dot(model_all.adjacency_matrix_)
```

```
(2000, 6)
M0={5}
M1={5}
M2={0, 1, 5}
M3={0, 1, 2, 5}
M4={0, 1, 2, 5}
M5=set()
```

2.19 Finding ancestors of each variable

By using `utils.extract_ancestors`, which implements Algorithm 1 of RCD method[1], we can extract the ancestors of variables. Since RCD allows for the existence of unobserved common causes, we can search for ancestors even when there are unobserved common causes, as in the following example.

References

[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.

2.19.1 Import and settings

```
import random

import numpy as np
import pandas as pd

from sklearn.utils import check_array
from lingam.utils import make_dot, extract_ancestors
```

2.19.2 Test data

```
def get_coef():
    coef = random.random()
    return coef if coef >= 0.5 else coef - 1.0
get_external_effect = lambda n: np.random.normal(0.0, 0.5, n) ** 3

B = np.array([[       0.0,        0.0,        0.0,        0.0,        0.0, get_coef(),
↪     0.0],
              [       0.0,        0.0,        0.0,        0.0,        0.0, get_coef(),
↪     0.0],
              [get_coef(), get_coef(),        0.0,        0.0,        0.0,        0.0,
↪     0.0],
              [       0.0,        0.0, get_coef(),        0.0,        0.0,        0.0,
↪get_coef()],
              [       0.0,        0.0, get_coef(),        0.0,        0.0,        0.0,
↪get_coef()],
              [       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]])

samples = 500
f0 = get_external_effect(samples)
f1 = get_external_effect(samples)
x0 = f0 * B[0, 5] + get_external_effect(samples)
x1 = f0 * B[1, 5] + get_external_effect(samples)
x2 = x0 * B[2, 0] + x1 * B[2, 1] + get_external_effect(samples)
x3 = x2 * B[3, 2] + f1 * B[3, 6] + get_external_effect(samples)
x4 = x2 * B[4, 2] + f1 * B[4, 6] + get_external_effect(samples)

# f0 and f1 are latent confounders
X = pd.DataFrame(np.array([x0, x1, x2, x3, x4]).T ,columns=['x0', 'x1', 'x2', 'x3', 'x4
↪'])

make_dot(B, labels=['x0', 'x1', 'x2', 'x3', 'x4', 'f0', 'f1'])
```

2.19.3 Extract the ancestors of each observed variable

```
M = extract_ancestors(X)

for i in range(X.shape[1]):
    if len(M[i]) == 0:
        print(f'x{i} has no ancestors.')
    else:
        print(f'The ancestors of x{i} are ' + ', '.join([f'x{n}' for n in M[i]]))
```

```
x0 has no ancestors.
x1 has no ancestors.
The ancestors of x2 are x0, x1
The ancestors of x3 are x0, x1, x2
The ancestors of x4 are x0, x1, x2
```

2.20 F-correlation

By using `utils.f_correlation`, which implements the F-correlation[1], we can compute the nonlinear correlation between two variables.

References

2.20.1 Import and settings

```
import numpy as np
import matplotlib.pyplot as plt
from lingam.utils import f_correlation
```

2.20.2 Test data

```
def linear_data(n, r):
    a = np.random.randn(n)
    e1 = np.random.randn(n)
    e2 = np.random.randn(n)
    if r < 0:
        r = -r
        x = -np.sqrt(r)*a - np.sqrt(1-r)*e1
    else:
        x = np.sqrt(r)*a + np.sqrt(1-r)*e1
    y = np.sqrt(r)*a + np.sqrt(1-r)*e2
    return x, y

def x2_data(n):
    x = np.random.uniform(-5, 5, n)
```

(continues on next page)

[1] Bach, Francis R., and Michael I. Jordan. "Kernel independent component analysis." Journal of machine learning research 3.Jul (2002): 1-48.

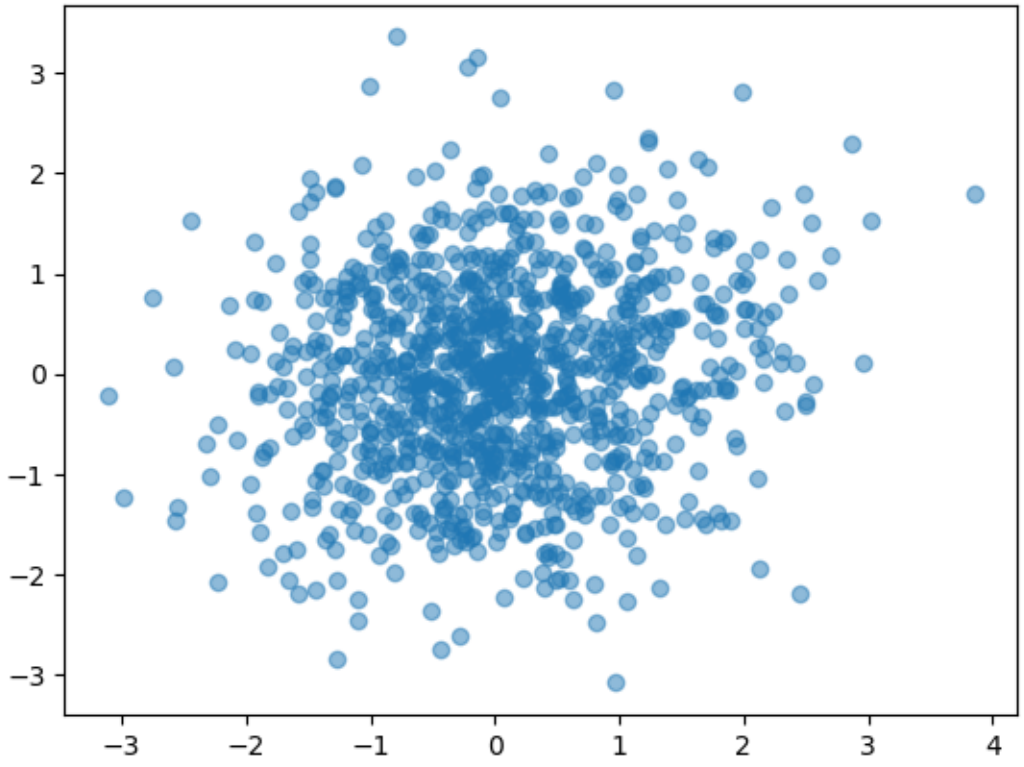
(continued from previous page)

```
    e = np.random.randn(n)
    y = 0.5 * (x ** 2) + e
    return x, y

def sin_data(n):
    e = np.random.randn(n)
    x = np.random.uniform(-5, 5, n)
    y = 5 * np.sin(x) + e
    return x, y
```

2.20.3 Linear correlated data (Uncorrelated)

```
x, y = linear_data(1000, 0.1)
corr = np.corrcoef(x, y)[0, 1]
print(f"Pearson's correlation coefficient= {corr:.3f}")

corr = f_correlation(x, y)
print(f'F-correlation= {corr:.3f}')

plt.scatter(x, y, alpha=0.5)
plt.show()
```

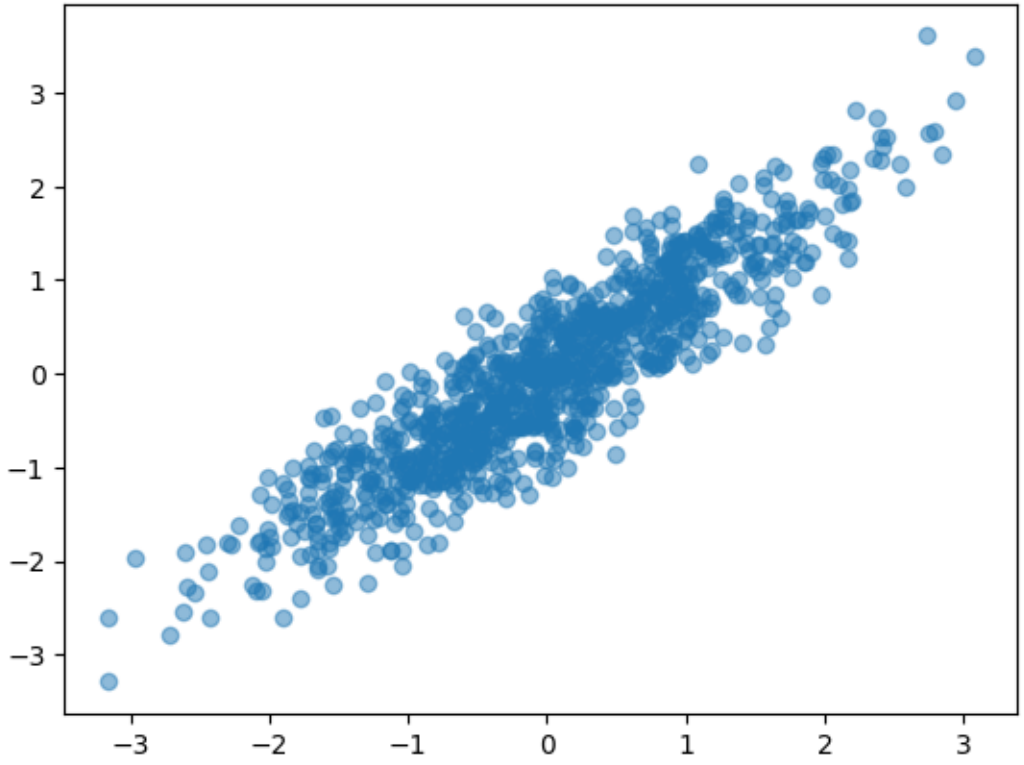
```
Pearson's correlation coefficient= 0.126
F-correlation= 0.120
```



2.20.4 Linear correlated data (Strongly correlated)

```
x, y = linear_data(1000, 0.9)
corr = np.corrcoef(x, y)[0, 1]
print(f"Pearson's correlation coefficient= {corr:.3f}")

corr = f_correlation(x, y)
print(f'F-correlation= {corr:.3f}')

plt.scatter(x, y, alpha=0.5)
plt.show()
```

```
Pearson's correlation coefficient= 0.907
F-correlation= 0.814
```



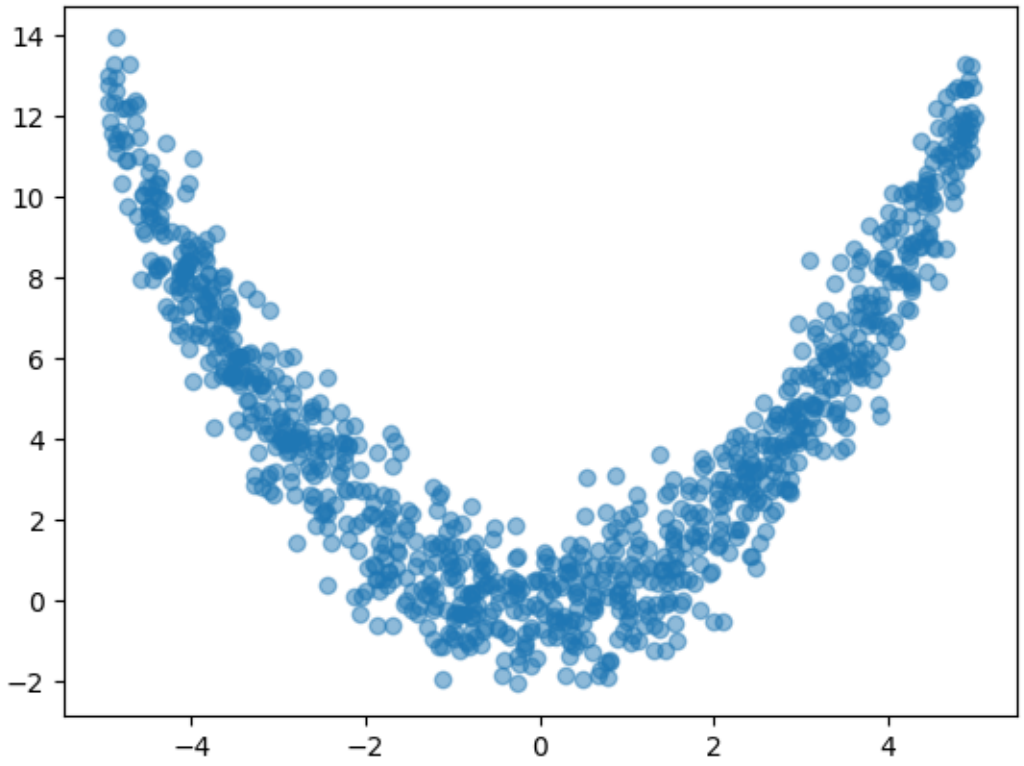
2.20.5 Non-linear correlated data (Quadratic function)

```
x, y = x2_data(1000)
corr = np.corrcoef(x, y)[0, 1]
print(f"Pearson's correlation coefficient= {corr:.3f}")

corr = f_correlation(x, y)
print(f'F-correlation= {corr:.3f}')

plt.scatter(x, y, alpha=0.5)
```

(continues on next page)

(continued from previous page)

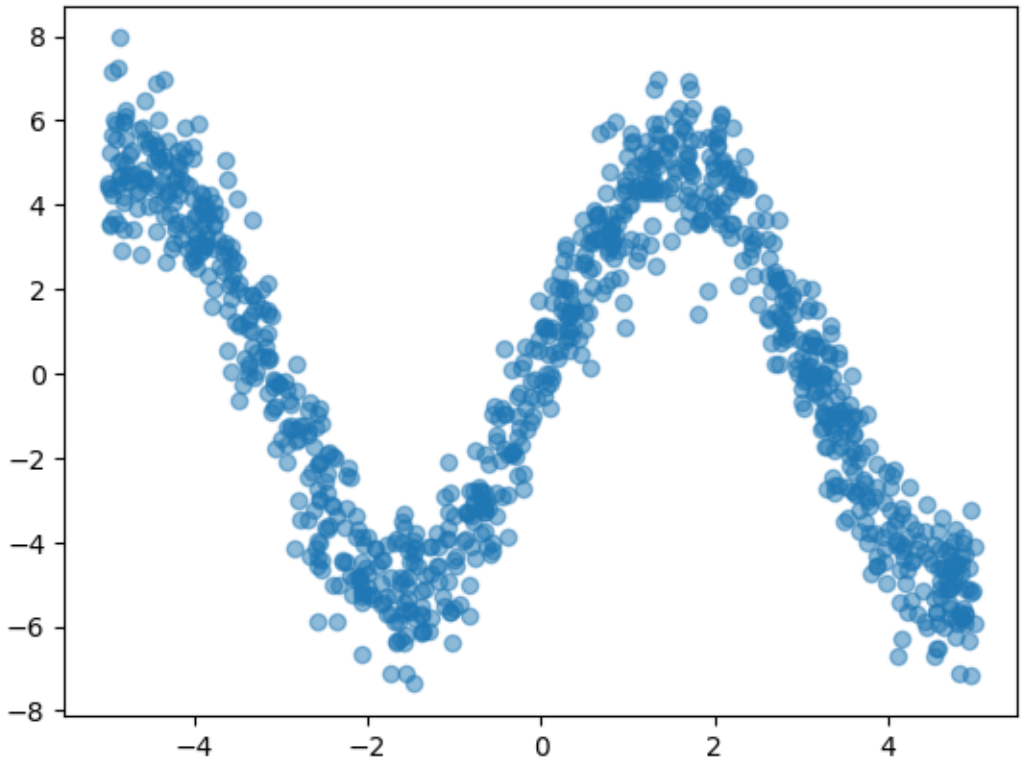
```
plt.show()


Pearson's correlation coefficient= 0.037
F-correlation= 0.848
```



2.20.6 Non-linear correlated data (Sin function)

```
x, y = sin_data(1000)
corr = np.corrcoef(x, y)[0, 1]
print(f"Pearson's correlation coefficient= {corr:.3f}")

corr = f_correlation(x, y)
print(f'F-correlation= {corr:.3f}')

plt.scatter(x, y, alpha=0.5)
plt.show()
```

```
Pearson's correlation coefficient= -0.275
F-correlation= 0.853
```



2.21 Visualization of nonlinear causal effect

2.21.1 Import and settings

```
import numpy as np
import pandas as pd

from lingam import RESIT
from sklearn.ensemble import RandomForestRegressor

from lingam.utils import make_dot, visualize_nonlinear_causal_effect
import matplotlib.pyplot as plt

np.random.seed(0)
```

2.21.2 Data generation

```
n_samples = 1000

def N(size):
    return np.random.uniform(size=size) - 0.5

X = np.zeros([n_samples, 5])
X[:, 0] = N(n_samples)
X[:, 1] = 3 * (X[:, 0] + 0.25) * (X[:, 0] - 0.25) + N(n_samples)
X[:, 2] = -0.75 * (X[:, 0] - 1) * (X[:, 0] - 2) + 1.5 * X[:, 1] + N(n_samples)
X[:, 3] = 5 * (X[:, 1] + 0.4) * (X[:, 1] - 0.1) * (X[:, 1] - 0.5) + 1 * np.log(5 * X[:,␣
↪2] + 20) + N(n_samples)
X[:, 4] = -0.8 * (X[:, 3] - 1.5) * (X[:, 3] - 3.5) + N(n_samples)
X = pd.DataFrame(X, columns=[f'X{i}' for i in range(5)])

pd.plotting.scatter_matrix(X, figsize=(8, 8), alpha=0.5)
plt.show()
```

2.21.3 Causal discovery

```
regressor = RandomForestRegressor()
model = RESIT(regressor=regressor, alpha=1)
bs_result = model.bootstrap(X, n_sampling=100)
```

2.21.4 Visualization

```
fig = plt.figure(figsize=(9, 6))

fig = visualize_nonlinear_causal_effect(X, bs_result, RandomForestRegressor(), "X2", "X3
↪", fig=fig)

for ax in fig.get_axes():
    ax.legend()
    ax.grid()

fig.tight_layout()
fig.show()
```

2.22 EvaluateModelFit

This example explains how to use *lingam.utils.evaluate_model_fit*. This function returns the mode fit of the given adjacency matrix to the data.

2.22.1 Import and settings

```
import numpy as np
import pandas as pd
from scipy.special import expit
import lingam
from lingam.utils import make_dot

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

import warnings
warnings.filterwarnings("ignore")

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

```
['1.24.4', '2.0.3', '1.8.2']
```

2.22.2 When all variables are continuous data

2.22.3 Test data

```
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()
```

2.22.4 Causal Discovery

Perform causal discovery to obtain the adjacency matrix.

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

```
array([[ 0.   ,  0.   ,  0.   ,  2.994,  0.   ,  0.   ],
       [ 2.995,  0.   ,  1.993,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  5.957,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 7.998,  0.   , -1.005,  0.   ,  0.   ,  0.   ],
       [ 3.98 ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ]])
```

2.22.5 Evaluation

Calculate the model fit of the given adjacency matrix to given data.

```
lingam.utils.evaluate_model_fit(model.adjacency_matrix_, X)
```

2.22.6 When the data has hidden common causes

2.22.7 Test data

```
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()
```

2.22.8 Causal Discovery

nan represents having a hidden common cause.

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

```
array([[ 0.   ,    nan,  0.   ,    nan,  0.   ,  0.   ],
       [   nan,  0.   ,  0.   ,    nan,  0.   ,  0.   ],
       [-0.22 ,  0.593,  0.   ,  0.564,  0.   ,  0.   ],
       [   nan,    nan,  0.   ,  0.   ,  0.   ,  0.   ],
       [ 0.542,  0.   , -0.529,  0.   ,  0.   ,  0.   ],
       [ 0.506,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ]])
```

```
lingam.utils.evaluate_model_fit(model.adjacency_matrix_, X)
```

2.22.9 When the data has ordinal variables

2.22.10 Test data

```
x3 = np.random.uniform(size=1000)
x0 = 0.6*x3 + np.random.uniform(size=1000)

# discrete
x2 = 1.2*x3 + np.random.uniform(size=1000)
x2 = expit(x2 - np.mean(x2))
vec_func = np.vectorize(lambda p: np.random.choice([0, 1], p=[p, 1 - p]))
x2 = vec_func(x2)

x1 = 0.6*x0 + 0.4*x2 + np.random.uniform(size=1000)
x5 = 0.8*x0 + np.random.uniform(size=1000)
x4 = 1.6*x0 - 0.2*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()
```

```
adjacency_matrix = np.array([
    [0.0, 0.0, 0.0, 0.6, 0.0, 0.0],
    [0.6, 0.0, 0.4, 0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0, 1.2, 0.0, 0.0],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
    [1.6, 0.0,-0.2, 0.0, 0.0, 0.0],
```

(continues on next page)

(continued from previous page)

```
    [0.8, 0.0, 0.0, 0.0, 0.0, 0.0]]
)
```

Specify whether each variable is an ordinal variable in *is_ordinal*.

```
lingam.utils.evaluate_model_fit(adjacency_matrix, X, is_ordinal=[0, 0, 1, 0, 0, 0])
```

CHAPTER

THREE

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.

property 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.

[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.

Return type
BootstrapResult

`property 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]

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[Page 97, 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].

property 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.

[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.

Returns
result – Returns the result of bootstrapping.

Return type
BootstrapResult

`property 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[Page 99, 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.

property 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, . . .)

property 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

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

- **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*, ...)

property 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

estimate_total_effect2(*from_index*, *to_index*)

Estimate total effect using causal model.

Parameters

- **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=True*, *ar_coefs=None*, *lingam_model=None*, *random_state=None*)

Implementation of VAR-LiNGAM Algorithm[1]

References

`__init__`(*lags=1*, *criterion='bic'*, *prune=True*, *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=True)*) – Whether to prune the adjacency matrix of lags.
- **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.

[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.

property adjacency_matrices_

Estimated adjacency matrix.

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

property 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

estimate_total_effect2(*n_features*, *from_index*, *to_index*, *from_lag=0*)

Estimate total effect using causal model.

Parameters

- **n_features** – 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)

`property 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=True, 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=True, 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.

[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

- **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=True)*) – Whether to prune the adjacency matrix of lags.
- **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.

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

Returns

result – Returns the result of bootstrapping.

Return type

TimeseriesBootstrapResult

property 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

estimate_total_effect2(*n_features*, *from_index*, *to_index*, *from_lag=0*)

Estimate total effect using causal model.

Parameters

- **n_features** – 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)

property 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]

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.

property adjacency_matrices_

Estimated adjacency matrices.

Returns

adjacency_matrices_ – The list of adjacency matrix B(t,t) and B(t,t-) for longitudinal datasets. The shape of B(t,t) and B(t,t-) 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-) 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-)), …)

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*, …)

property 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.

[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.

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 (timepoint, 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

`estimate_total_effect2`(*from_t*, *from_index*, *to_t*, *to_index*)

Estimate total effect using causal model.

Parameters

- **_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)

property 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*, *resampled_indices=None*)

The result of bootstrapping.

__init__(*adjacency_matrices*, *total_effects*, *resampled_indices=None*)

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.
- **resampled_indices** (*array-like, shape (n_sampling, resample_size), optional (default=None)*) – The list of original index of resampled samples.

property 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

property resampled_indices_

The list of original index of resampled samples.

Returns
resampled_indices_ – The list of original index of resampled samples, where `n_sampling` is the number of bootstrap sampling and `resample_size` is the size of each subsample set.

Return type
array-like, shape (n_sampling, resample_size)

property 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 VARBootstrapResult

class lingam.**VARBootstrapResult**(*adjacency_matrices*, *total_effects*)

The result of bootstrapping for Time series algorithm.

__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.

property 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_lag=0*, *to_lag=0*, *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_lag** (*int*) – Number of lag at the start of the path. `from_lag` should be greater than or equal to `to_lag`.
- **to_lag** (*int*) – Number of lag at the end of the path. `from_lag` should be greater than or equal to `to_lag`.
- **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

property resampled_indices_

The list of original index of resampled samples.

Returns
resampled_indices_ – The list of original index of resampled samples, where `n_sampling` is the number of bootstrap sampling and `resample_size` is the size of each subsample set.

Return type
array-like, shape (n_sampling, resample_size)

property 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)

3.9 VARMABootstrapResult

class lingam.**VARMABootstrapResult**(*adjacency_matrices*, *total_effects*, *order*)

The result of bootstrapping for Time series algorithm.

__init__(*adjacency_matrices*, *total_effects*, *order*)

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.

property 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_lag=0*, *to_lag=0*, *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_lag** (*int*) – Number of lag at the start of the path. `from_lag` should be greater than or equal to `to_lag`.
- **to_lag** (*int*) – Number of lag at the end of the path. `from_lag` should be greater than or equal to `to_lag`.
- **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

property resampled_indices_

The list of original index of resampled samples.

Returns
resampled_indices_ – The list of original index of resampled samples, where `n_sampling` is the number of bootstrap sampling and `resample_size` is the size of each subsample set.

Return type
array-like, shape (n_sampling, resample_size)

property 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 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.

property 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

property 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)

3.11 BottomUpParceLiNGAM

class lingam.**BottomUpParceLiNGAM**(*random_state=None*, *alpha=0.1*, *regressor=None*, *prior_knowledge=None*, *independence='hsic'*, *ind_corr=0.5*)

Implementation of ParceLiNGAM Algorithm[1]

References

__init__(*random_state=None*, *alpha=0.1*, *regressor=None*, *prior_knowledge=None*, *independence='hsic'*, *ind_corr=0.5*)

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[Page 119, 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.
- **independence** (*{'hsic', 'fcorr'}, optional (default='hsic')*) – Methods to determine independence. If 'hsic' is set, test for independence by HSIC. If 'fcorr' is set, independence is determined by F-correlation.
- **ind_corr** (*float, optional (default=0.5)*) – The threshold value for determining independence by F-correlation; independence is determined when the value of F-correlation is below this threshold value.

property **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)

[1] T. Tashiro, S. Shimizu, and A. Hyvärinen. ParceLiNGAM: a causal ordering method robust against latent confounders. Neural computation, 26.1: 57-83, 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

property 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

estimate_total_effect2(*from_index*, *to_index*)

Estimate total effect using causal model.

Parameters

- **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)

3.12 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'*, *independence='hsic'*, *ind_corr=0.5*)

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'*, *independence='hsic'*, *ind_corr=0.5*)

Construct a RCD model.

Parameters

- **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.

[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.

- scott : Scott's Rule of Thumb.
- silverman : Silverman's Rule of Thumb.

- **independence** (*{'hsic', 'fcorr'}, optional (default='hsic')*) – Methods to determine independence. If 'hsic' is set, test for independence by HSIC. If 'fcorr' is set, independence is determined by F-correlation.
- **ind_corr** (*float, optional (default=0.5)*) – The threshold value for determining independence by F-correlation; independence is determined when the value of F-correlation is below this threshold value.

property 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)

property 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)

3.13 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)

[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.

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.
- **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

3.14 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*, *path=None*, *path_color=None*, *detect_cycle=False*, *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.
- **path** (*tuple, optional (default=None)*) – Path to highlight. Tuple of start index and end index.
- **path_color** (*string, optional (default=None)*) – Colors to highlight a path.
- **detect_cycle** (*boolean, optional (default=False)*) – Highlight simple cycles.
- **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.**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

lingam.utils.**extract_ancestors**(*X*, *max_explanatory_num=2*, *cor_alpha=0.01*, *ind_alpha=0.01*, *shapiro_alpha=0.01*, *MLHSICR=True*, *bw_method='mdbs'*)

Extract a set of ancestors of each variable Implementation of RCD Algorithm1[1]

References

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.
- **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.

lingam.utils.**f_correlation**(*x*, *y*)

Implementation of F-correlation[2]

[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.

[2] Bach, Francis R., and Michael I. Jordan. "Kernel independent component analysis." Journal of machine learning research 3.Jul (2002): 1-48.

References

Parameters

- **x** (*array-like, shape (n_samples)*) – Data, where `n_samples` is the number of samples.
- **y** (*array-like, shape (n_samples)*) – Data, where `n_samples` is the number of samples.

Returns
The valus of F-correlation.

Return type
float

lingam.utils.**visualize_nonlinear_causal_effect**(*X*, *cd_result*, *estimator*, *cause_name*, *effect_name*, *cause_positions=None*, *percentile=None*, *fig=None*, *boxplot=False*)

Visualize non-linear causal effect.

Parameters

- **X** (*pandas.DataFrame, shape (n_samples, n_features)*) – Training data used to obtain cd_result.
- **cd_result** (*array-like with shape (n_features, n_features) or* `BootstrapResult`) – Adjacency matrix or BootstrapResult. These are the results of a causal discovery.
- **estimator** (*estimator object*) – `estimator` used for non-linear regression. Regression with `estimator` using `cause_name` and covariates as explanatory variables and `effect_name` as objective variable. Those covariates are searched for in `cd_result`.
- **cause_name** (*str*) – The name of the cause variable.
- **effect_name** (*str*) – The name of the effect variable.
- **cause_positions** (*array-like, optional (default=None)*) – List of positions from which causal effects are calculated. By default, `cause_positions` stores the position at which the value range of X is divided into 10 equal parts.
- **percentile** (*array-like, optional (default=None)*) – A tuple consisting of three percentile values. Each value must be greater than 0 and less than 100. By default, (95, 50, 5) is set.
- **fig** (*plt.Figure, optional (default=None)*) – If `fig` is given, draw a figure in `fig`. If not given, plt.fig is prepared internally.
- **boxplot** (*boolean, optional (default=False)*) – If True, draw a box plot instead of a scatter plot for each `cause_positions`.

Returns
fig – Plotted figure.

Return type
plt.Figure

lingam.utils.**evaluate_model_fit**(*adjacency_matrix*, *X*, *is_ordinal=None*)

evaluate the given adjacency matrix and return fit indices

Parameters

- **adjacency_matrix** (*array-like, shape (n_features, n_features)*) – Adjacency matrix representing a causal graph. The i-th column and row correspond to the i-th column of X.
- **X** (*array-like, shape (n_samples, n_features)*) – Training data.
- **is_ordinal** (*array-like, shape (n_features,)*) – Binary list. The i-th element represents that the i-th column of X is ordinal or not. 0 means not ordinal, otherwise ordinal.

Returns
fit_indices – Fit indices. This API uses semopy's calc_stats(). See semopy's reference for details.

Return type
pandas.DataFrame

3.15 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.

property 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)

[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. of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21), 2021: 2097–2103.

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

property 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[2]

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.

[2] Y. Zeng, S. Shimizu, R. Cai, F. Xie, M. Yamamoto and Z. Hao. Causal Discovery with Multi-Domain LiNGAM for Latent Factors. In Proc. of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21), 2021: 2097–2103.

- **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.

property 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

property 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)

3.16 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.

property 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

property causal_order_

Estimated causal ordering.

[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.

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)

3.17 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.

property 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*, *only_global=False*, *is_poisson=False*)

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.
- **only_global** (*boolean, optional (default=False)*) – If True, then the method will only perform the global optimization to estimate the causal structure, without the local search phase.
- **is_poisson** (*boolean, optional (default=False)*) – If True, then the method will use poisson regression model to compute the log-likelihood in the local search phase.

[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.

Returns
self – Returns the instance of self.

Return type
object

3.18 CAM-UV

class lingam.**CAMUV**(*alpha=0.01*, *num_explanatory_vals=2*, *independence='hsic'*, *ind_corr=0.5*)

Implementation of CAM-UV Algorithm[1]

References

__init__(*alpha=0.01*, *num_explanatory_vals=2*, *independence='hsic'*, *ind_corr=0.5*)

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.
- **independence** (*{'hsic', 'fcorr'}, optional (default='hsic')*) – Methods to determine independence. If 'hsic' is set, test for independence by HSIC. If 'fcorr' is set, independence is determined by F-correlation.
- **ind_corr** (*float, optional (default=0.5)*) – The threshold value for determining independence by F-correlation; independence is determined when the value of F-correlation is below this threshold value.

property 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)

3.19 MultiGroupRCD

class lingam.**MultiGroupRCD**(*max_explanatory_num=2*, *cor_alpha=0.01*, *ind_alpha=0.01*, *shapiro_alpha=0.01*, *MLHSICR=True*, *bw_method='mdbs'*, *independence='hsic'*, *ind_corr=0.5*)

Implementation of RCD Algorithm with multiple groups

[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.

__init__(*max_explanatory_num=2*, *cor_alpha=0.01*, *ind_alpha=0.01*, *shapiro_alpha=0.01*, *MLHSICR=True*, *bw_method='mdbs'*, *independence='hsic'*, *ind_corr=0.5*)

Construct a model.

Parameters

- **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=True)*) – 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.
- **independence** (*{'hsic', 'fcorr'}, optional (default='hsic')*) – Methods to determine independence. If 'hsic' is set, test for independence by HSIC. If 'fcorr' is set, independence is determined by F-correlation.
- **ind_corr** (*float, optional (default=0.5)*) – The threshold value for determining independence by F-correlation; independence is determined when the value of F-correlation is below this threshold value.

property 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, . . .)

property 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_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*, ...)

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

estimate_total_effect2(*from_index*, *to_index*)

Estimate total effect using causal model.

Parameters

- **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)

CHAPTER

FOUR

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

l

lingam, 103
lingam.utils, 124

INDEX

Symbols

__init__() (*lingam.BootstrapResult method*), 108
__init__() (*lingam.BottomUpParceLiNGAM method*), 119
__init__() (*lingam.CAMUV method*), 136
__init__() (*lingam.CausalEffect method*), 123
__init__() (*lingam.DirectLiNGAM method*), 97
__init__() (*lingam.ICALiNGAM method*), 95
__init__() (*lingam.LiM method*), 135
__init__() (*lingam.LiNA method*), 130
__init__() (*lingam.LongitudinalBootstrapResult method*), 116
__init__() (*lingam.LongitudinalLiNGAM method*), 106
__init__() (*lingam.MDLiNA method*), 131
__init__() (*lingam.MultiGroupDirectLiNGAM method*), 99
__init__() (*lingam.MultiGroupRCD method*), 136
__init__() (*lingam.RCD method*), 121
__init__() (*lingam.RESIT method*), 133
__init__() (*lingam.VARBootstrapResult method*), 111
__init__() (*lingam.VARLiNGAM method*), 101
__init__() (*lingam.VARMABootstrapResult method*), 113
__init__() (*lingam.VARMALiNGAM method*), 103

A

adjacency_matrices_ (*lingam.BootstrapResult property*), 108
adjacency_matrices_ (*lingam.LongitudinalBootstrapResult property*), 116
adjacency_matrices_ (*lingam.LongitudinalLiNGAM property*), 106
adjacency_matrices_ (*lingam.MultiGroupDirectLiNGAM property*), 99
adjacency_matrices_ (*lingam.MultiGroupRCD property*), 137
adjacency_matrices_ (*lingam.VARBootstrapResult property*), 111
adjacency_matrices_ (*lingam.VARLiNGAM property*), 101
adjacency_matrices_ (*lingam.VARMABootstrapResult property*), 113
adjacency_matrices_ (*lingam.VARMALiNGAM property*), 104
adjacency_matrix_ (*lingam.BottomUpParceLiNGAM property*), 119
adjacency_matrix_ (*lingam.CAMUV property*), 136
adjacency_matrix_ (*lingam.DirectLiNGAM property*), 97
adjacency_matrix_ (*lingam.ICALiNGAM property*), 95
adjacency_matrix_ (*lingam.LiM property*), 135
adjacency_matrix_ (*lingam.LiNA property*), 130
adjacency_matrix_ (*lingam.MDLiNA property*), 132
adjacency_matrix_ (*lingam.MultiGroupDirectLiNGAM property*), 99
adjacency_matrix_ (*lingam.RCD property*), 122
adjacency_matrix_ (*lingam.RESIT property*), 133
ancestors_list_ (*lingam.MultiGroupRCD property*), 137
ancestors_list_ (*lingam.RCD property*), 122

B

bootstrap() (*lingam.BottomUpParceLiNGAM method*), 119
bootstrap() (*lingam.DirectLiNGAM method*), 97
bootstrap() (*lingam.ICALiNGAM method*), 95
bootstrap() (*lingam.LongitudinalLiNGAM method*), 106
bootstrap() (*lingam.MultiGroupDirectLiNGAM method*), 99
bootstrap() (*lingam.MultiGroupRCD method*), 137
bootstrap() (*lingam.RCD method*), 122
bootstrap() (*lingam.RESIT method*), 133
bootstrap() (*lingam.VARLiNGAM method*), 102
bootstrap() (*lingam.VARMALiNGAM method*), 104
BootstrapResult (*class in lingam*), 108
BottomUpParceLiNGAM (*class in lingam*), 119

C

CAMUV (*class in lingam*), 136

causal_order_ (*lingam.BottomUpParceLiNGAM property*), 120
causal_order_ (*lingam.DirectLiNGAM property*), 98
causal_order_ (*lingam.ICALiNGAM property*), 96
causal_order_ (*lingam.MultiGroupDirectLiNGAM property*), 100
causal_order_ (*lingam.RESIT property*), 133
causal_order_ (*lingam.VARLiNGAM property*), 102
causal_order_ (*lingam.VARMALiNGAM property*), 104
causal_orders_ (*lingam.LongitudinalLiNGAM property*), 106
CausalEffect (*class in lingam*), 123

D

DirectLiNGAM (*class in lingam*), 97

E

estimate_effects_on_prediction() (*lingam.CausalEffect method*), 123
estimate_optimal_intervention() (*lingam.CausalEffect method*), 123
estimate_total_effect() (*lingam.BottomUpParceLiNGAM method*), 120
estimate_total_effect() (*lingam.DirectLiNGAM method*), 98
estimate_total_effect() (*lingam.ICALiNGAM method*), 96
estimate_total_effect() (*lingam.LongitudinalLiNGAM method*), 107
estimate_total_effect() (*lingam.MultiGroupDirectLiNGAM method*), 100
estimate_total_effect() (*lingam.MultiGroupRCD method*), 138
estimate_total_effect() (*lingam.RESIT method*), 134
estimate_total_effect() (*lingam.VARLiNGAM method*), 102
estimate_total_effect() (*lingam.VARMALiNGAM method*), 104
estimate_total_effect2() (*lingam.BottomUpParceLiNGAM method*), 120
estimate_total_effect2() (*lingam.LongitudinalLiNGAM method*), 107
estimate_total_effect2() (*lingam.MultiGroupDirectLiNGAM method*), 100
estimate_total_effect2() (*lingam.MultiGroupRCD method*), 138
estimate_total_effect2() (*lingam.VARLiNGAM method*), 102
estimate_total_effect2() (*lingam.VARMALiNGAM method*), 105
evaluate_model_fit() (*in module lingam.utils*), 129
extract_ancestors() (*in module lingam.utils*), 128

F

f_correlation() (*in module lingam.utils*), 128
find_all_paths() (*in module lingam.utils*), 126
fit() (*lingam.BottomUpParceLiNGAM method*), 120
fit() (*lingam.DirectLiNGAM method*), 98
fit() (*lingam.ICALiNGAM method*), 96
fit() (*lingam.LiM method*), 135
fit() (*lingam.LiNA method*), 130
fit() (*lingam.LongitudinalLiNGAM method*), 107
fit() (*lingam.MDLiNA method*), 132
fit() (*lingam.MultiGroupDirectLiNGAM method*), 100
fit() (*lingam.MultiGroupRCD method*), 138
fit() (*lingam.RCD method*), 122
fit() (*lingam.RESIT method*), 134
fit() (*lingam.VARLiNGAM method*), 103
fit() (*lingam.VARMALiNGAM method*), 105

G

get_causal_direction_counts() (*lingam.BootstrapResult method*), 108
get_causal_direction_counts() (*lingam.LongitudinalBootstrapResult method*), 116
get_causal_direction_counts() (*lingam.VARBootstrapResult method*), 111
get_causal_direction_counts() (*lingam.VARMABootstrapResult method*), 114
get_directed_acyclic_graph_counts() (*lingam.BootstrapResult method*), 109
get_directed_acyclic_graph_counts() (*lingam.LongitudinalBootstrapResult method*), 117
get_directed_acyclic_graph_counts() (*lingam.VARBootstrapResult method*), 111
get_directed_acyclic_graph_counts() (*lingam.VARMABootstrapResult method*), 114
get_error_independence_p_values() (*lingam.BottomUpParceLiNGAM method*), 121
get_error_independence_p_values() (*lingam.DirectLiNGAM method*), 98
get_error_independence_p_values() (*lingam.ICALiNGAM method*), 96
get_error_independence_p_values() (*lingam.LongitudinalLiNGAM method*), 107
get_error_independence_p_values() (*lingam.MultiGroupDirectLiNGAM method*),

101
get_error_independence_p_values() (*lingam.MultiGroupRCD method*), 138
get_error_independence_p_values() (*lingam.RCD method*), 122
get_error_independence_p_values() (*lingam.RESIT method*), 134
get_error_independence_p_values() (*lingam.VARLiNGAM method*), 103
get_error_independence_p_values() (*lingam.VARMALiNGAM method*), 105
get_exo_variables() (*in module lingam.utils*), 126
get_paths() (*lingam.BootstrapResult method*), 109
get_paths() (*lingam.LongitudinalBootstrapResult method*), 117
get_paths() (*lingam.VARBootstrapResult method*), 112
get_paths() (*lingam.VARMABootstrapResult method*), 115
get_probabilities() (*lingam.BootstrapResult method*), 110
get_probabilities() (*lingam.LongitudinalBootstrapResult method*), 118
get_probabilities() (*lingam.VARBootstrapResult method*), 112
get_probabilities() (*lingam.VARMABootstrapResult method*), 115
get_sink_variables() (*in module lingam.utils*), 126
get_total_causal_effects() (*lingam.BootstrapResult method*), 110
get_total_causal_effects() (*lingam.LongitudinalBootstrapResult method*), 118
get_total_causal_effects() (*lingam.VARBootstrapResult method*), 113
get_total_causal_effects() (*lingam.VARMABootstrapResult method*), 115

I

ICALiNGAM (*class in lingam*), 95

L

likelihood_i() (*in module lingam.utils*), 127
LiM (*class in lingam*), 135
LiNA (*class in lingam*), 130
lingam
    module, 94–96, 98, 101, 103, 105, 108, 110, 113, 116, 118, 121, 123, 130, 132, 134, 136
lingam.utils
    module, 124
log_p_super_gaussian() (*in module lingam.utils*), 127
LongitudinalBootstrapResult (*class in lingam*), 116
LongitudinalLiNGAM (*class in lingam*), 106

M

make_dot() (*in module lingam.utils*), 125
make_prior_knowledge() (*in module lingam.utils*), 124
MDLiNA (*class in lingam*), 131
measurement_matrix_ (*lingam.LiNA property*), 131
measurement_matrix_ (*lingam.MDLiNA property*), 132
module
    lingam, 94–96, 98, 101, 103, 105, 108, 110, 113, 116, 118, 121, 123, 130, 132, 134, 136
    lingam.utils, 124
MultiGroupDirectLiNGAM (*class in lingam*), 99
MultiGroupRCD (*class in lingam*), 136

P

predict_adaptive_lasso() (*in module lingam.utils*), 127
print_causal_directions() (*in module lingam.utils*), 124
print_dagc() (*in module lingam.utils*), 124

R

RCD (*class in lingam*), 121
remove_effect() (*in module lingam.utils*), 125
resampled_indices_ (*lingam.BootstrapResult property*), 110
resampled_indices_ (*lingam.VARBootstrapResult property*), 113
resampled_indices_ (*lingam.VARMABootstrapResult property*), 116
residuals_ (*lingam.LongitudinalLiNGAM property*), 108
residuals_ (*lingam.VARLiNGAM property*), 103
residuals_ (*lingam.VARMALiNGAM property*), 105
RESIT (*class in lingam*), 133

T

total_effects_ (*lingam.BootstrapResult property*), 110
total_effects_ (*lingam.LongitudinalBootstrapResult property*), 118
total_effects_ (*lingam.VARBootstrapResult property*), 113
total_effects_ (*lingam.VARMABootstrapResult property*), 116

V

VARBootstrapResult (*class in lingam*), 111
variance_i() (*in module lingam.utils*), 127
VARLiNGAM (*class in lingam*), 101

VARMABootstrapResult (*class in lingam*), 113
VARMALiNGAM (*class in lingam*), 103
visualize_nonlinear_causal_effect() (*in module lingam.utils*), 129