ananke-causal

The Ananke Team

Dec 05, 2023

GETTING STARTED

Ananke, named for the Greek primordial goddess of necessity and causality, is a Python package for causal inference using the language of graphical models.

Ananke provides a Python implementation of causal graphical models with and without unmeasured confounding, with a particular focus on causal identification, semiparametric estimation, and parametric likelihood methods.

Ananke is licensed under Apache 2.0 and source code is available at gitlab.

CHAPTER

ONE

CITATION

If you enjoyed this package, we would appreciate the following citation:

Additional relevant citations also include:

CHAPTER

TWO

CONTRIBUTORS

- Rohit Bhattacharya
- Jaron Lee
- Razieh Nabi
- Preethi Prakash
- Ranjani Srinivasan

CHAPTER

THREE

DOCUMENTATION

3.1 Installation

If graph visualization is not required then install via *pip*:

```
pip install ananke-causal
```

Alternatively, the package may be installed from gitlab by cloning and *cd* into the directory. Then, poetry can be used to install:

```
poetry install
```

3.1.1 Install with graph visualization

If graphing support is required, it is necessary to install graphviz.

Non M1 Mac instructions

Ubuntu:

```
sudo apt install graphviz libgraphviz-dev pkg-config
```

Mac Homebrew:

```
brew install graphviz
```

Fedora:

```
sudo yum install graphviz
```

Once graphviz has been installed, then:

```
pip install ananke-causal[viz] # if pip is preferred
poetry install --extras viz # if poetry is preferred
```

3.1.2 M1 Mac specific instructions

If on M1 see this issue. The fix is to run the following before installing:

```
brew install graphviz
python -m pip install \
    --global-option=build_ext \
    --global-option="-I$(brew --prefix graphviz)/include/" \
    --global-option="-L$(brew --prefix graphviz)/lib/" \
    pygraphviz
```

Install graphviz using the appropriate method for your OS

```
# Ubuntu

sudo apt install graphviz libgraphviz-dev pkg-config

# Mac

brew install graphviz

# Mac (M1)
## see https://github.com/pygraphviz/pygraphviz/issues/398

brew install graphviz
python -m pip install \
    --global-option=build_ext \
    --global-option="-I$(brew --prefix graphviz)/include/" \
    --global-option="-L$(brew --prefix graphviz)/lib/" \
    pygraphviz

# Fedora

sudo yum install graphviz
```

Install the latest release using pip.

```
pip3 install ananke-causal
```

For more details please see the gitlab, or the documentation for details on how to use Ananke.

3.2 Change Log

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

3.2.1 [Unreleased]

Added

- Automated most of the package release workflow
- Added support for parameterizing discrete data causal hidden variable DAGs using `pgmpy`
- Added Sympy symbolic conditional probability distributions (`ananke.factors.discrete_factors.SymCPD`), discrete factors, and variable elimination
- Added native BayesianNetwork class with support for both `pgmpy.factors.discrete.TabularCPD` and `ananke.factors.discrete_factors.SymCPD`.

Changed

- Improved documentation (install instructions, API reference, documentation for discrete models, suggested citations)
- Reworked `ananke.models.discrete` interface (split up into `ananke.models.bayesian_networks`, `ananke.estimation.empirical_plugin`, `ananke.identification.oracle`)
- Improvements to certain graphical operations (cleaning up `subgraph` implementation)

3.2.2 [0.3.3] - 2023-03-02

Fixed

- Fixed failing CI issues due to Poetry lock

3.2.3 [0.3.2] - 2023-03-02

Fixed

- Marked Python `graphviz` dependency as optional

3.2.4 [0.3.1] - 2023-03-02

Fixed

- Fixed merge conflicts

3.2.5 [0.3.0] - 2023-03-02

Added

- Added pre-commit hooks for automatic formatting and PEP8 compliance
- Add `pgmpy` integration for working with and identifying causal effects in discrete data models

Changed

- Changed package build system from `setup.py` to `pyproject.toml` with `poetry`
- Updated CI/CD to work with `poetry`
- Removed outdated dependencies on deprecated packages
- Removed `graphviz` as a required dependency to install `ananke`

Fixed

- Fix `models.binary_nested.BinaryNestedModel` failing to check if effect is identified

3.2.6 [0.2.1] - 2023-02-15

Fixed

- Fixed version numbers in `setup.py`

3.2.7 [0.2.0] - 2023-02-15

Added

- Add changelog for versions after 0.1.11.
- Add optimal adjustment set functionality, by Zixiao Wang (PR 53)

Fixed

- Updated contributing guidelines

3.2.8 [0.1.12] - 2023-01-23

Fixed

- Fixed typo in the Apache license
- Fixed incorrect year in copyright notice

3.2.9 [0.1.11] - 2023-01-23

Changed

- Switch from GPLv3 to Apache 2.0 license

3.3 ananke.graphs

3.3.1 ananke.graphs.admg

Class for acyclic directed mixed graphs (ADMGs) and conditional ADMGs (CADMGs).

class ananke.graphs.admg.**ADMG**(*vertices=[]*, *di_edges={}*, *bi_edges={}*, ***kwargs*)

Bases: *ananke.graphs.sg.SG*

Class for creating and manipulating (conditional) acyclic directed mixed graphs (ADMGs/CADMGs).

canonical_dag(*cardinality: Optional[Union[int, str]] = None*)

Computes the canonical DAG from the ADMG by inserting a variable in place of each bidirected edge.

For each bidirected edge 'X <-> Y', the inserted variable will take names of the form 'U_XY', the original bidirected edge is removed, and new directed edges 'U_X_Y -> Y' and 'U_X_Y -> X' are inserted. The variable names are in lexicographic order.

Params cardinality The cardinality of the inserted variables

Returns A directed acyclic graph

Return type *DAG*

fixable(*vertices*)

Check if there exists a valid fixing order and return such an order in the form of a list, else returns an empty list.

Parameters **vertices** – set of vertices to check fixability for.

Returns a boolean indicating whether the set was fixable and a valid fixing order as a stack.

property fixed

Returns all fixed nodes in the graph.

Returns

get_intrinsic_sets()

Computes intrinsic sets (and returns the fixing order for each intrinsic set).

Returns list of intrinsic sets and fixing orders used to reach each one

get_intrinsic_sets_and_heads()

Computes intrinsic sets mapped to a tuple of heads and tails of that intrinsic set, and fixing orders for each one.

Returns tuple of dict of intrinsic sets to heads and tails, and fixing orders for each intrinsic set

is_ancestral_subgraph(*other*)

Check that this graph is an ancestral subgraph of the other. An ancestral subgraph over variables S and intervention b G(S(b)) of a larger graph G(V(b)) is defined as a subgraph, such that ancestors of each node s in S with respect to the graph G(V(b_i)) are contained in S.

Parameters **other** – an object of the ADMG class.

Returns boolean indicating whether the statement is True or not.

is_subgraph(*other*)

Check that this graph is a subgraph of other, meaning it has a subset of edges and nodes of the other.

Parameters **other** – an object of the ADMG class.

Returns boolean indicating whether the statement is True or not.

`latent_projection`(*retained_vertices*)

Computes latent projection.

Parameters `retained_vertices` – list of vertices to retain after latent projection

Returns Latent projection containing retained vertices

`m_separated`(*X*, *Y*, *separating_set=[]*)

Computes m-separation for set *X* and set *Y* given *separating_set*

Parameters

- `X` – first vertex set
- `Y` – second vertex set
- `separating_set` – separating set

Returns boolean result of m-separation

`markov_blanket`(*vertices*)

Get the Markov blanket of a set of vertices.

Parameters `vertices` – iterable of vertex names.

Returns set corresponding to Markov blanket.

`markov_pillow`(*vertices*, *top_order*)

Get the Markov pillow of a set of vertices. That is, the Markov blanket of the vertices given a valid topological order on the graph.

Parameters

- `vertices` – iterable of vertex names.
- `top_order` – a valid topological order.

Returns set corresponding to Markov pillow.

`maximal_arid_projection`()

Get the maximal arid projection that encodes the same conditional independences and Vermas as the original ADMG. This operation is described in Acyclic Linear SEMs obey the Nested Markov property by Shpitser et al 2018.

Returns An ADMG corresponding to the maximal arid projection.

`mb_shielded`()

Check if the ADMG is a Markov blanket shielded ADMG. That is, check if two vertices are non-adjacent only when they are absent from each others' Markov blankets.

Returns boolean indicating if it is mb-shielded or not.

`nonparametric_saturated`()

Check if the nested Markov model implied by the ADMG is nonparametric saturated. The following is an implementation of Algorithm 1 in Semiparametric Inference for Causal Effects in Graphical Models with Hidden Variables (Bhattacharya, Nabi & Shpitser 2020) which was shown to be sound and complete for this task.

Returns boolean indicating if it is nonparametric saturated or not.

reachable_closure(*vertices*)

Obtain reachable closure for a set of vertices.

Parameters **vertices** – set of vertices to get reachable closure for.

Returns set corresponding to the reachable closure, the fixing order for vertices outside of the closure, and the CADMG corresponding to the closure.

subgraph(*vertices*)

Computes subgraph given a set of vertices.

Recomputes districts, since these may change when vertices are removed.

Parameters **vertices** – iterable of vertices

Returns subgraph

exception ananke.graphs.admg.**UndefinedADMGOperation**(**args*, ***kwargs*)

Bases: Exception

ananke.graphs.admg.**latent_project_single_vertex**(*vertex*, *graph*)

Latent project one vertex from graph

Parameters

- **vertex** – Name of vertex to be projected
- **graph** – ADMG

Returns

3.3.2 ananke.graphs.bg

Class for Bidirected Graphs (BGs).

class ananke.graphs.bg.**BG**(*vertices=[]*, *bi_edges={}*, ***kwargs*)

Bases: *ananke.graphs.admg.ADMG*

3.3.3 ananke.graphs.cg

Class for Lauritzen-Wermuth-Frydenberg chain graphs (LWF-CGs/CGs).

class ananke.graphs.cg.**CG**(*vertices=[]*, *di_edges={}*, *ud_edges={}*, ***kwargs*)

Bases: *ananke.graphs.sg.SG*

boundary(*vertices*)

Get the boundary of a set of vertices defined as the block and parents of the block.

Parameters **vertices** – iterable of vertex names.

Returns set corresponding to the boundary.

3.3.4 ananke.graphs.dag

Class for Directed Acyclic Graphs (DAGs).

class ananke.graphs.dag.**DAG**(*vertices=[]*, *di_edges={}*, ***kwargs*)

Bases: *ananke.graphs.admg.ADMG*, *ananke.graphs.cg.CG*

d_separated(*X*, *Y*, *separating_set=[]*)

Computes d-separation for set *X* and set *Y* given *separating_set*

Parameters

- **X** – first vertex set
- **Y** – second vertex set
- **separating_set** – separating set, default list()

Returns boolean result of d-separation

exception ananke.graphs.dag.**UndefinedDAGOperation**

Bases: Exception

3.3.5 ananke.graphs.graph

Base class for all graphs.

TODO: Add error checking

class ananke.graphs.graph.**Graph**(*vertices=[]*, *di_edges={}*, *bi_edges={}*, *ud_edges={}*, ***kwargs*)

Bases: object

add_biedge(*sib1*, *sib2*)

Add a bidirected edge to the graph.

Parameters

- **sib1** – endpoint 1 of edge.
- **sib2** – endpoint 2 of edge.

Returns None.

add_diedge(*parent*, *child*)

Add a directed edge to the graph.

Parameters

- **parent** – tail of edge.
- **child** – head of edge.

Returns None.

add_udedge(*neb1*, *neb2*)

Add an undirected edge to the graph.

Parameters

- **neb1** – endpoint 1 of edge.
- **neb2** – endpoint 2 of edge.

Returns None.

`add_vertex`(*name*, *cardinality=None*, ***kwargs*)

Add a vertex to the graph.

Parameters `name` – name of vertex.

Returns None.

`ancestors`(*vertices*)

Get the ancestors of a vertex or set of vertices.

Parameters `vertices` – single vertex name or iterable of vertex names to find ancestors for.

Returns set of ancestors.

`children`(*vertices*)

Get children of a vertex or set of vertices.

Parameters `vertices` – iterable of vertex names.

Returns set of children.

`copy`()

Returns copy of a graph

`delete_biedge`(*sib1*, *sib2*)

Delete given bidirected edge from the graph.

Parameters

- `sib1` – endpoint 1 of edge.
- `sib2` – endpoint 2 of edge.

Returns None.

`delete_diedge`(*parent*, *child*)

Deleted given directed edge from the graph.

Parameters

- `parent` – tail of edge.
- `child` – head of edge.

Returns None.

`delete_udedge`(*neb1*, *neb2*)

Delete given undirected edge from the graph.

Parameters

- `neb1` – endpoint 1 of edge.
- `neb2` – endpoint 2 of edge.

Returns None.

`delete_vertex`(*name*)

Delete a vertex from the graph. Additionally removes any edges from other vertices into the given vertex

Parameters `name` – name of vertex.

Returns None.

`descendants`(*vertices*)

Get the descendants of a vertex or set of vertices.

Parameters `vertices` – single vertex name or iterable of vertex names to find descendants for.

Returns set of descendants.

`directed_paths`(*source*, *sink*)

Get all directed paths between sets of source vertices and sink vertices.

Parameters

- `source` – a set of vertices that serve as the source.
- `sink` – a set of vertices that serve as the sink.

Returns list of directed paths.

`draw`(*direction=None*)

Visualize the graph.

:return : dot language representation of the graph.

`has_biedge`(*sib1*, *sib2*)

Check existence of a bidirected edge.

Parameters

- `sib1` – endpoint 1 of edge.
- `sib2` – endpoint 2 of edge.

Returns boolean result of existence.

`has_udedge`(*neb1*, *neb2*)

Check existence of an undirected edge.

Parameters

- `neb1` – endpoint 1 of edge.
- `neb2` – endpoint 2 of edge.

Returns boolean result of existence.

`neighbors`(*vertices*)

Get neighbors of a vertex or set of vertices.

Parameters `vertices` – iterable of vertex names.

Returns set of neighbors.

`parents`(*vertices*)

Get parents of a vertex or set of vertices.

Parameters `vertices` – iterable of vertex names.

Returns set of parents.

`pre`(*vertices*, *top_order*)

Find all nodes prior to the given set of vertices under a topological order.

Parameters

- `vertices` – iterable of vertex names.
- `top_order` – a valid topological order.

Returns list corresponding to the order up until the given vertices.

`siblings`(*vertices*)

Get siblings of a vertex or set of vertices.

Parameters `vertices` – vertex name or iterable of vertex names.

Returns set of neighbors.

`subgraph`(*vertices*)

Return a subgraph on the given vertices (i.e. a graph containing only the specified vertices and edges between them).

Parameters `vertices` – set containing names of vertices in the subgraph.

Returns a new Graph object corresponding to the subgraph.

`topological_sort`()

Perform a topological sort from roots (parentless nodes) to leaves, as ordered by directed edges on the graph.

Returns list corresponding to a valid topological order.

3.3.6 ananke.graphs.ig

Class for Intrinsic Graphs (IGs) – a mixed graph used to compute intrinsic sets of an ADMG in polynomial time.

`class` ananke.graphs.ig.`IG`(*admg*)

Bases: *ananke.graphs.graph.Graph*

`add_new_biedges`(*s*)

Naive O(|I(G)| choose 2) implementation. Must ensure that biedges not added to ancestors.

Parameters `s` – Frozen set corresponding to the new vertex.

Returns None.

`bidirected_connected`(*s1*, *s2*)

Check if two sets are bidirected connected in the original ADMG

Parameters

- `s1` – First set corresponding to a vertex in the IG.
- `s2` – Second set corresponding to a vertex in the IG.

Returns boolean corresponding to connectedness.

`get_heads_tails`()

Iterate through all intrinsic sets and get the recursive heads and tails.

Returns List of tuples corresponding to (head, tail).

`get_intrinsic_sets`()

Get all intrinsic sets for given ADMG.

Returns

`maintain_subset_relation`(*s*)

Add di edges to a newly inserted vertex s so as to maintain the subset relation. :param s: Frozenset corresponding to the new vertex. :return: None.

merge(*s1*, *s2*)

Merge operation on two sets s1 and s2 to give a (possibly) new intrinsic set s3.

Parameters

- **s1** – First set corresponding to a vertex in the IG.
- **s2** – Second set corresponding to a vertex in the IG.

Returns None.

3.3.7 ananke.graphs.missing_admg

Class for missing data acyclic directed mixed graphs

class ananke.graphs.missing_admg.**MissingADMG**(*vertices=[]*, *di_edges={}*, *bi_edges={}*, ***kwargs*)

Bases: *ananke.graphs.admg.ADMG*

draw(*direction=None*)

Visualize the graph.

:return : dot language representation of the graph.

3.3.8 ananke.graphs.sg

Class for segregated graphs (SGs).

class ananke.graphs.sg.**SG**(*vertices=[]*, *di_edges={}*, *bi_edges={}*, *ud_edges={}*, ***kwargs*)

Bases: *ananke.graphs.graph.Graph*

add_biedge(*sib1*, *sib2*, *recompute=True*)

Add a bidirected edge to the graph. Overridden to recompute districts.

Parameters

- **sib1** – endpoint 1 of edge.
- **sib2** – endpoint 2 of edge.
- **recompute** – boolean indicating whether districts should be recomputed.

Returns None.

add_udedge(*neb1*, *neb2*, *recompute=True*)

Add an undirected edge to the graph. Overridden to recompute blocks

Parameters

- **neb1** – endpoint 1 of edge.
- **neb2** – endpoint 2 of edge.
- **recompute** – boolean indicating whether blocks should be recomputed.

Returns None.

block(*vertex*)

Returns the block of a vertex.

Parameters **vertex** – name of the vertex.

Returns set corresponding to block.

property blocks

Returns list of all blocks in the graph.

Returns list of sets corresponding to blocks in the graph.

delete_biedge(*sib1*, *sib2*, *recompute=True*)

Delete given bidirected edge from the graph. Overridden to recompute districts.

Parameters

- **sib1** – endpoint 1 of edge.
- **sib2** – endpoint 2 of edge.
- **recompute** – boolean indicating whether districts should be recomputed.

Returns None.

delete_udedge(*neb1*, *neb2*, *recompute=True*)

Delete given undirected edge from the graph. Overridden to recompute blocks.

Parameters

- **neb1** – endpoint 1 of edge.
- **neb2** – endpoint 2 of edge.
- **recompute** – boolean indicating whether blocks should be recomputed.

Returns None.

district(*vertex*)

Returns the district of a vertex.

Parameters **vertex** – name of the vertex.

Returns set corresponding to district.

property districts

Returns list of all districts in the graph.

Returns list of sets corresponding to districts in the graph.

fix(*vertices*)

Perform the graphical operation of fixing on a set of vertices.

Does not check that vertices are fixable.

Parameters **vertices** – iterable of vertices to be fixed.

Returns None.

fixable(*vertices*)

Check if there exists a valid fixing order and return such an order in the form of a list, else returns an empty list.

Parameters **vertices** – set of vertices to check fixability for.

Returns a boolean indicating whether the set was fixable and a valid fixing order as a stack.

3.3.9 ananke.graphs.ug

Class for Undirected Graphs (UGs).

class ananke.graphs.ug.**UG**(*vertices=[]*, *ud_edges={}*, ***kwargs*)

Bases: *ananke.graphs.cg.CG*

3.3.10 ananke.graphs.vertex

Class to represent vertex of a graphical model.

The vertex class should be general enough to fit into any kind of graphical model – UGs, DAGs, CGs, ADMGs, CPDAGs, CADMGs etc.

class ananke.graphs.vertex.**Vertex**(*name*, *fixed=False*, *cardinality=2*, *attributes=None*)

Bases: object

3.4 ananke.models

3.4.1 ananke.models.bayesian_network

class ananke.models.bayesian_network.**BayesianNetwork**(*graph:* ananke.graphs.dag.DAG, *cpds: Dict[str, Union[pgmpy.factors.discrete.CPD.TabularCPD, ananke.factors.discrete_factor.SymCPD]]*)

Bases: *ananke.graphs.dag.DAG*

copy()

Returns copy of a graph

fix(*variables: List[str]*)

Performs an intervention by setting the conditional distribution of each intervened variable to be a point mass at the intervened value. This is a faithful operation (the graph is changed and the associated CPD structure reflects the lack of parents).

get_cpds(*vertex*)

to_pgmpy()

Converts into pgmpy.models.BayesianNetwork object.

ananke.models.bayesian_network.**create_symbolic_cpds**(*ls_dag*, *use_uniform_unobs_var=True*)

ananke.models.bayesian_network.**generate_random_cpds**(*graph*, *dir_conc=10*, *context_variable='S'*, *seed=42*)

Given a graph and a set of cardinalities for variables in a DAG, constructs random conditional probability distributions. Supports optional contexts and context variable to generate CPDs consistent with a context specific DAG for data fusion.

Parameters

- **graph** – A graph whose variables have cardinalities, and optionally
- **dir_conc** – The Dirichlet concetration parameter
- **context_variable** – Name of the context variable

ananke.models.bayesian_network.**intervene**(*net*, *treatment_dict*)

Performs an intervention on a pgmpy.models.BayesianNetwork, by setting the conditional distribution of each intervened variable to be a point mass at the intervened value. Does not alter the structure of the parents of the network (i.e. is a non-faithful operation).

If you have an ananke.models.BayesianNetwork, consider using the .fix(treatment_dict) method instead, which has the further advantage of performing the operation faithfully (the underlying DAG is modified accordingly, and the parents of the intervened variables in that conditional probability distributions are removed).

Parameters

- **net** (*pgmpy.models.BayesianNetwork*) – pgmpy.models.Bayesian Network
- **treatment_dict** (*dict*) – dictionary of variables to values:

3.4.2 ananke.models.binary_nested

Implements the Nested Markov parameterization for acyclic directed mixed graphs over binary variables.

The code relies on the following papers:

[ER12] Evans, R. J., & Richardson, T. S. (2012). Maximum likelihood fitting of acyclic directed mixed graphs to binary data. arXiv preprint arXiv:1203.3479. [ER19] Evans, R. J., & Richardson, T. S. (2019). Smooth, identifiable supermodels of discrete DAG models with latent variables. Bernoulli, 25(2), 848-876.

class ananke.models.binary_nested.**BinaryNestedModel**(*graph*)

Bases: object

Estimates parameters of nested binary model using the iterative maximum likelihood algorithm (Algorithm 1) of [ER12]. Performs scipy.minimize constrained minimization of negative log-likelihood.

estimate(*treatment_dict=None*, *outcome_dict=None*, *check_identified=True*)

Estimates p(Y(a)=y) for treatment A=a, outcome Y=y using the binary nested Markov parameters.

Parameters

- **treatment_dict** – dict of treatment variables to values
- **outcome_dict** – dict of outcome variables to values
- **check_identified** – boolean to check that effect is identified, default True

Returns interventional probability p(Y(a)) if identified, else return None

fit(*X*, *q_vector=None*, *tol=1e-08*, **args*, ***kwargs*)

Fits the binary nested model. Let N the number of observations, M the number of variables.

Parameters

- **X** – Either a N x M pandas DataFrame where each row represents an observation, or an 2 ** M x (M+1) pandas DataFrame, where each row represents the observation through a count variable
- **args** –
- **kwargs** –

Returns

ananke.models.binary_nested.**compute_M**(*G*, *partition_head_dict*, *district*, *heads_tails_map*, *terms*)

Compute M matrix as given in Section 3.1 of [ER12]

Parameters

- **G** – ADMG graph
- **partition_head_dict** – dictionary mapping every subset of given district to constituent heads
- **district** – a district of the graph
- **heads_tails_map** – dictionary mapping heads to tails
- **terms** – list of terms of given district

Returns

ananke.models.binary_nested.**compute_P**(*partition_head_dict*, *q_vector_keys*, *heads_tails_map*, *terms*)

Computes P matrix as given in Section 3.1 of [ER12]

Parameters

- **partition_head_dict** – dictionary mapping every subset of given district to constituent heads
- **q_vector_keys** – list of parameter names
- **heads_tails_map** – dictionary mapping heads to tails
- **terms** – list of all terms for a given district

Returns

ananke.models.binary_nested.**compute_all_M_and_P**(*G*, *intrinsic_dict*)

Computes for each district of a graph, the corresponding M and P matrices required to compute the expression

$$p(q) = \prod_j M_j \exp(P_j \log q_j)$$

See Section 3.1 of [ER12] for further details.

Parameters

- **G** – ADMG
- **intrinsic_dict** – mapping of intrinsic set to heads and tails

Returns

ananke.models.binary_nested.**compute_counterfactual_binary_parameters**(*G*, *q_vector*, *x_dict*, *y_dict*)

Computes a counterfactual (interventional quantity) p(Y=y | do(X=x)) using nested Markov parameters and graph

Parameters

- **G** – ADMG
- **q_vector** – A dictionary of nested Markov parameters
- **x_dict** – A dictionary of treatment variables to treatment values
- **y_dict** – A dictionary of outcome variables to outcome values

Returns Computed probability p(Y=y | do(X=x))

ananke.models.binary_nested.**compute_district_bool_map**(*q_vector_keys*, *districts*)

Parameters

- **q_vector_keys** –

- **districts** –

Returns

ananke.models.binary_nested.**compute_likelihood_district**(*nu_dict*, *q_vector*, *district*, *heads_tails_map*, *intrinsic_dict*)

Compute likelihood directly using Equation (1) of [ER12].

This likelihood is not recommended for use in maximization as it is more efficiently expressed using M and P computations.

Parameters

- **nu_dict** – a dictionary representing a single observation of variables to value
- **q_vector** – dictionary mapping the head, tail, tail value to parameter value
- **district** – district of graph
- **heads_tails_map** – mapping of heads to tails
- **intrinsic_dict** – mapping of intrinsic set to heads and tails

Returns

ananke.models.binary_nested.**compute_partition_head_dict**(*intrinsic_dict*, *district*)

Compute partition head dictionary. Maps every subset of a district to its constituent maximal recursive heads

Parameters

- **intrinsic_dict** – dictionary mapping intrinsic sets to (heads, tails) of that set
- **district** – district of graph

Returns

ananke.models.binary_nested.**compute_q_indices_by_district**(*q_vector_keys*, *districts*)

Computes a boolean indexing array that indicates which q parameters are involved in which districts.

Parameters

- **q_vector_keys** –
- **districts** –

Returns

ananke.models.binary_nested.**compute_terms**(*district*, *intrinsic_dict*, *heads_tails_map*)

Computes list of terms (product of q parameters) and the partition head dictionary. Each term is a tuple of (frozenset(all heads), tuple(all tails), tuple(values of all tails)).

Parameters

- **district** – district of graph
- **intrinsic_dict** – dictionary mapping intrinsic sets to (heads, tails) of that set
- **heads_tails_map** – dictionary mapping heads to tails

Returns

ananke.models.binary_nested.**compute_theta_bool_map**(*q_vector_keys*, *variables*)

Compute map from variable to boolean indexing vector of q_parameters which have heads containing that variable. In this map, the indices are not reindexed by district. The boolean vector selects parameters which have heads containing that variable.

Parameters

- **q_vector_keys** – list of q_vector keys
- **variables** – list of variables

Returns

ananke.models.binary_nested.**compute_theta_reindexed_bool_map**(*q_vector_keys*, *districts*)

Creates a mapping from a variable to a boolean indexing vector.

This boolean vector indexes only parameters whose heads are involved in the district of that variable. It selects parameters which have heads containing that variable.

Used to construct A and b matrices for partial likelihood.

Parameters

- **q_vector_keys** – list of q_vector keys
- **districts** – list of districts

Returns

ananke.models.binary_nested.**construct_A_b**(*variable*, *q*, *theta_reindexed_bool_map*, *M*, *P*)

Constructs A and b matrices (eqn 4, Evans and Richardson 2013) for constraining parameters of given variable, in district which admits matrices M, P

Parameters

- **variable** – name of variable
- **q_vector** – q_vector in OrderedDict format
- **M** – M matrix
- **P** – P matrix

Returns

ananke.models.binary_nested.**get_heads_tails_map**(*intrinsic_dict*)

Get mapping of heads to tails from a mapping of intrinsic sets

Parameters **intrinsic_dict** – mapping of intrinsic sets of some graph to heads and tails

Returns

ananke.models.binary_nested.**get_recursive_heads**(*intrinsic_dict*)

Compute all heads of intrinsic sets

Parameters **intrinsic_dict** – mapping of intrinsic sets of some graph to heads and tails

Returns list of all heads

ananke.models.binary_nested.**initialize_q_vector**(*intrinsic_dict*)

Generates the q_vector, a dictionary mapping (heads: frozenset, tails: tuple, value of tail: tuple) to parameter. Default q parameter values are 1/2^#(head), for each head.

Parameters **intrinsic_dict** – mapping of intrinsic sets of some graph to heads and tails

Return q_vector

ananke.models.binary_nested.**maximal_heads**(*list_H*, *intrinsic_dict*)

Returns maximal heads for a set of heads. Only defined if all heads in list_H (list) are in IHT, ie, they are all heads of intrinsic sets

ananke.models.binary_nested.**permutations**(*n*, *k=2*)

Computes tuples of all permutations of n variables with cardinality k.

Parameters

- **n** – number of variables
- **k** – cardinality of each variable

Returns

ananke.models.binary_nested.**process_data**(*df*, *count_variable=None*)

Parameters

- **data** – pandas DataFrame columns of variables and rows of observations
- **count_variable** – optional name of counting variable, if data is provided as a summary table

Returns a vector of counts, ordered in ascending order of v for p(V=v)

ananke.models.binary_nested.**recursive_partition_heads**(*B*, *intrinsic_dict*)

Partition an arbitrary (sub)set of vertices B (in V) into recursive heads

Parameters

- **B** – arbitrary possibly empty subset of vertices
- **intrinsic_dict** – map of intrinsic sets to heads and tails of that set

3.4.3 ananke.models.fixtures

3.4.4 ananke.models.linear_gaussian_sem

Class for Linear Gaussian SEMs parametrized by a matrix B representing regression coefficients and a matrix Omega representing correlated errors

class ananke.models.linear_gaussian_sem.**LinearGaussianSEM**(*graph*)

Bases: object

bic(*X*)

Calculate Bayesian information criterion of the data given the model.

Parameters

- **X** – a N x M dimensional data matrix.
- **weights** – optional 1d numpy array with weights for each data point (rows with higher weights are given greater importance).

Returns a float corresponding to the log-likelihood.

draw(*direction=None*)

Visualize the graph.

:return : dot language representation of the graph.

fit(*X*, *tol=1e-06*, *disp=None*, *standardize=False*, *max_iters=100*)

Fit the model to data via (weighted) maximum likelihood estimation

Parameters

- **X** – data – a N x M dimensional pandas data frame.
- **weights** – optional 1d numpy array with weights for each data point (rows with higher weights are given greater importance).

Returns self.

neg_loglikelihood(*X*)

Calculate log-likelihood of the data given the model.

Parameters

- **X** – a N x M dimensional data matrix.
- **weights** – optional 1d numpy array with weights for each data point (rows with higher weights are given greater importance).

Returns a float corresponding to the log-likelihood.

total_effect(*A*, *Y*)

Calculate the total causal effect of a set of treatments A on a set of outcomes Y.

Parameters

- **A** – iterable corresponding to variable names that act as treatments.
- **Y** – iterable corresponding to variable names that act as outcomes.

Returns a float corresponding to the total causal effect.

ananke.models.linear_gaussian_sem.**is_positive_definite**(*X*)

3.5 ananke.identification

3.5.1 ananke.identification.missing_id

Class for missing ID

class ananke.identification.missing_id.**MissingFullID**(*graph*)

Bases: object

id()

Function to ID the full law

Returns boolean is ID or not

3.5.2 ananke.identification.one_line

Class for one line ID algorithms.

exception ananke.identification.one_line.**NotIdentifiedError**

Bases: Exception

Custom error for when desired functional is not identified.

class ananke.identification.one_line.**OneLineAID**(*graph*, *treatments*, *outcomes*)

Bases: object

functional(*experiments*)

Creates a string representing the identifying functional

Parameters **experiments** – A list of sets denoting the interventions of the available experimental distributions

Returns

id(*experiments*)

Checks if identification query is identified given the set of experimental distributions.

Parameters **experiments** – a list of ADMG objects in which intervened variables are fixed.

class ananke.identification.one_line.**OneLineGID**(*graph*, *treatments*, *outcomes*)

Bases: *ananke.identification.one_line.OneLineAID*

id(*experiments=[]*)

Checks if identification query is identified given the set of experimental distributions.

Parameters **experiments** – A list of ADMG objects denoting the interventions of the available experimental distributions.

Returns boolean indicating if query is ID or not.

class ananke.identification.one_line.**OneLineID**(*graph*, *treatments*, *outcomes*)

Bases: `object`

draw_swig(*direction=None*)

Draw the proper SWIG corresponding to the causal query.

Returns dot language representation of the SWIG.

export_intermediates(*folder='intermediates'*)

Export intermediate CADMGs obtained during fixing.

Parameters **folder** – string specifying path to folder where the files will be written.

Returns None.

functional()

Creates and returns a string for identifying functional.

Returns string representing the identifying functional.

id()

Run one line ID for the query.

Returns boolean that is True if p(Y(a)) is ID, else False.

ananke.identification.one_line.**assert_valid_witness**(*net_1: Union[pgmpy.models.BayesianNetwork.BayesianNetwork, ananke.models.bayesian_network.BayesianNetwork]*, *net_2: Union[pgmpy.models.BayesianNetwork.BayesianNetwork, ananke.models.bayesian_network.BayesianNetwork]*, *observed_variables: list*, *treatment_dict: dict*, *outcome_variables=None*)

Asserts that two BayesianNetwork objects represent a valid witness for identification, meaning that the two Bayesian networks agree on the marginal distribution over *observed_variables* but disagree in at least one part of the counterfactual distribution for *outcome_variables* under the intervention specified by *treatment_dict*.

Parameters

- **net_1** – The first BayesianNetwork object
- **net_2** – The second BayesianNetwork object
- **observed_variables** – A list of variables for the observed margin
- **treatment_dict** – A dictionary of treatment variables and values
- **outcome_variables** – An optional list of outcome variables. If left unspecified, then it is

all variables in *observed_variables* except those in *treatment_dict*.

ananke.identification.one_line.**check_experiments_ancestral**(*admg*, *experiments*)

Check that each experiment G(S(b_i)) is ancestral in ADMG G(V(b_i)) https://simpleflying.com/

Parameters

- **admg** – An ADMG
- **experiments** – A list of ADMGs representing experiments

Returns

ananke.identification.one_line.**check_experiments_conform_to_gid**(*admg*, *experiments*)

ananke.identification.one_line.**get_allowed_intrinsic_sets**(*experiments*)

ananke.identification.one_line.**get_required_intrinsic_sets**(*admg*)

ananke.identification.one_line.**powerset**(*[1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3*)

3.5.3 ananke.identification.opt_adjust

Optimal adjustment sets finding

class ananke.identification.opt_adjust.**OptAdjustment**(*graph*)

Bases: object

generate_opt_adjustment_set(*vertex1*, *vertex2*)

Fits the binary nested model. Let N the number of observations, M the number of variables.

Parameters

- **self** – DAG
- **vertex1** – inference from vertex1, say treatment
- **vertex2** – inference to vertex2, say outcome

ananke.identification.opt_adjust.**get_min_set**(*G*, *input*, *vertex1*, *vertex2*)

function for minimal set

Parameters

- **G** – DAG
- **input** – optimal set found using get_opt_set function
- **vertex1** – inference from vertex1, say treatment
- **vertex2** – inference to vertex2, say outcome

ananke.identification.opt_adjust.**get_opt_set**(*G*, *vertex1*, *vertex2*)

function to get the optimal set from two vertices

Parameters

- **G** – the graph
- **vertex1** – inference from vertex1, say treatment
- **vertex2** – inference to vertex2, say outcome

ananke.identification.opt_adjust.**get_opt_set_from_set**(*G*, *node_set1*, *node_set2*)

function to get the optimal set from two vertices sets

Parameters

- **G** – the graph
- **node_set1** – set of inference from
- **node_set2** – set of inference to

3.5.4 ananke.identification.oracle

ananke.identification.oracle.**compute_effect_from_discrete_model**(*net*, *treatment_dict*, *outcome_dict*, *conditioning_dict=None*)

Compute the causal effect by directly performing an intervention in a Bayesian Network corresponding to the true structural equation model to obtain the counterfactual distribution, and then computing the marginal distribution of the outcome. Note that this function does not consider issues of identification as interventions are performed in the true model (regardless if those interventions were identified).

Parameters

- **net** – A Bayesian Network representing the causal problem. Note that this object is used only as a representation of the observed data distribution.
- **treatment_dict** – Dictionary of treatment variables to treatment values.
- **outcome_dict** – Dictionary of outcome variables to outcome values.

3.6 ananke.estimation

3.6.1 ananke.estimation.automated_if

Class for automated derivation of influence functions.

class ananke.estimation.automated_if.**AutomatedIF**(*graph*, *treatment*, *outcome*)

Bases: object

IF for a single treatment and single outcome.

3.6.2 ananke.estimation.counterfactual_mean

Class that provides an interface to estimation strategies for the counterfactual mean E[Y(t)]

class ananke.estimation.counterfactual_mean.**CausalEffect**(*graph*, *treatment*, *outcome*)

Bases: object

Provides an interface to various estimation strategies for the ACE: E[Y(1) - Y(0)].

compute_effect(*data*, *estimator*, *model_binary=None*, *model_continuous=None*, *n_bootstraps=0*, *alpha=0.05*, *report_log_odds=True*)

Bootstrap functionality to compute the Average Causal Effect if the outcome is continuous or the Causal Odds Ratio if the outcome is binary. Returns the point estimate as well as lower and upper quantiles for a user specified confidence level.

Parameters

- **data** – pandas data frame containing the data.
- **estimator** – string indicating what estimator to use: e.g. eff-apipw.
- **model_binary** – string specifying modeling strategy to use for binary variables: e.g. glm-binary.
- **model_continuous** – string specifying modeling strategy to use for continuous variables: e.g. glm-continuous.
- **n_bootstraps** – number of bootstraps.
- **alpha** – the significance level with the default value of 0.05.
- **report_log_odds** – if set to True, and the outcome variable is binary, then reports the log odds ratio.

Returns one float corresponding to ACE/OR if n_bootstraps=0, else three floats corresponding to ACE/OR, lower quantile, upper quantile.

3.6.3 ananke.estimation.empirical_plugin

ananke.estimation.empirical_plugin.**compute_district_factor**(*graph:* ananke.graphs.admg.ADMG, *obs_dist*, *fixing_order*)

Compute the interventional distribution associated with a district (or equivalently, its fixing order)

Parameters

- **graph** (*ananke.ADMG*) – Graph representing the problem
- **obs_dist** (*pgmpy.discrete.DiscreteFactor*) – Probability distribution corresponding to the graph
- **fixing_order** – A fixing sequence for the implied district D

ananke.estimation.empirical_plugin.**estimate_effect_from_discrete_dist**(*oid*, *obs_dist*, *treatment_dict*, *outcome_dict*)

Performs the ID algorithm to identify a causal effect given a discrete probability distribution representing the observed data distribution.

Parameters

- **oid** (OneLineID) – Ananke OneLineID object

- **obs_dist** – observed data distribution
- **treatment_dict** – dictionary of treatment variables and values
- **outcome_dict** – dictionary of outcome variables and values

ananke.estimation.empirical_plugin.**get_obs_dist_from_net**(*net*, *graph*)

3.7 Quickstart

This is a short tutorial for users that want to get into the thick of things right away. However, we highly recommend that users read the other docs provided in order to get a feel for all of the functionality provided in Ananke. The estimators here are based on theory in our paper Semiparametric Inference For Causal Effects In Graphical Models With Hidden Variables (Bhattacharya, Nabi, & Shpitser, 2020).

Following are the necessary packages that need to be imported for this tutorial.

[1]:
```
from ananke.graphs import ADMG
from ananke.identification import OneLineID
from ananke.estimation import CausalEffect
from ananke.datasets import load_afixable_data
from ananke.estimation import AutomatedIF
import numpy as np
```

3.7.1 Creating a Causal Graph

In Ananke, the most frequently used causal graph is an **acyclic directed mixed graph (ADMG)**. Roughly, directed edges $X \to Y$ indicate X is a direct cause of Y and bidirected edges $X \leftrightarrow Y$ indicate the presence of one or more unmeasured confounders between X and Y. Let's say we are studying the efficacy of a new antiretroviral therapy vs. an old one. Call this the treatment T where $T = 1$ represents receiving the new drug and $T = 0$ represents receiving the old. Our outcome of interest is the patients CD4 counts post treatment. Thus, our target of interest (in potential outcomes notation) is the counterfactual contrast $\psi \equiv E[Y(1)] - E[Y(0)]$. This is how one may create a causal graph, with additional variables relevant to the given problem.

[2]:
```
vertices = ['Income', 'Insurance', 'ViralLoad', 'Education', 'T', 'Toxicity', 'CD4']
di_edges = [('ViralLoad', 'Income'), ('ViralLoad', 'T'), ('ViralLoad', 'Toxicity'),
            ('Education', 'Income'), ('Education', 'T'), ('Education', 'Toxicity'),
            ('Income', 'Insurance'), ('Insurance', 'T'), ('T', 'Toxicity'), ('Toxicity',
↪'CD4'), ('T', 'CD4')]
bi_edges = [('Income', 'T'), ('Insurance', 'ViralLoad'), ('Education', 'CD4')]
G = ADMG(vertices, di_edges, bi_edges)
G.draw(direction="LR")
```

[2]:

3.7.2 Identification of the Causal Effect

Using Ananke, we can ask whether the effect of a given treatment T on a given outcome Y is identified or not. We have implemented the one line ID algorithm provided in Nested Markov Properties for Acyclic Directed Mixed Graphs which is sound and complete in identifying $p(Y(t))$ or equivalently $p(Y|do(t))$. We show how to use this in Ananke through the following example.

[3]:
```
one_id = OneLineID(graph=G, treatments=['T'], outcomes=['CD4'])
one_id.id()
```

[3]:
```
True
```

3.7.3 Estimation of the Causal Effect

Ananke provides an easy interface in order to compute causal effects. First, we instantiate a `CausalEffect` object.

[4]:
```
ace_obj = CausalEffect(graph=G, treatment='T', outcome='CD4')  # setting up the␣
↪CausalEffect object
```

```
 Treatment is a-fixable and graph is mb-shielded.

 Available estimators are:

1. IPW (ipw)
2. Outcome regression (gformula)
3. Generalized AIPW (aipw)
4. Efficient Generalized AIPW (eff-aipw)

Suggested estimator is Efficient Generalized AIPW
```

In this case, it recommends *Efficient Generalized AIPW* which is to say, that the estimator used to compute the effect in this case looks a lot like Augmented IPW (which is doubly robust). Further, Ananke uses semiparametric theory in order to provide an estimator that achieves the lowest asymptotic variance.

Given the list of estimators, it is up to the user to specify what estimators they want to work with. For instance, if the user decides to use the efficient generalized AIPW, they only need to use the keyword given in front of it, i.e., `eff-aipw`, when computing the effect. All the nuisance models are fit using generalized linear models provided in the `statsmodels`. Users interested in using different modeling approaches can refer to the documentation on accessing the functional form of the influence functions at the end of this notebook.

Let's load up some toy data and use all of the available estimators in order to compute the causal effect.

[5]:
```
data = load_afixable_data() # load some pre-simulated data
ace_ipw = ace_obj.compute_effect(data, "ipw")
ace_gformula = ace_obj.compute_effect(data, "gformula")
ace_aipw = ace_obj.compute_effect(data, "aipw")
ace_eff = ace_obj.compute_effect(data, "eff-aipw")
print("ace using IPW = ", ace_ipw)
print("ace using g-formula = ", ace_gformula)
```

(continues on next page)

(continued from previous page)

```
print("ace using AIPW = ", ace_aipw)
print("ace using efficient AIPW = ", ace_eff)
```

```
ace using IPW =  0.4917151858666866
ace using g-formula =  0.4968077996596518
ace using AIPW =  0.5005117929752623
ace using efficient AIPW =  0.491715185866723
```

In addition, the user can run bootstraps and obtain $(1-\alpha)*100\%$ confidence level for the causal effect point estimate. The user needs to specify two arguments: number of bootstraps `n_bootstraps` and the significance level α `alpha`. The confidence interval can then be obtained via bootstrap percentile intervals, as described in All of Nonparametric Statistics. In this case, the call to the `compute_effect` returns three values: the first corresponds to the effect computed on the original data, the second and third are the pre-specified lower and upper quantiles, i.e., $(\frac{\alpha}{2}, 1-\frac{\alpha}{2})$. The default value for α is set to be 0.05. We illustrate this through the following toy example.

[6]:
```
np.random.seed(0)  # setting the seed to get consistent numbers in the documentation
ace, Ql, Qu = ace_obj.compute_effect(data, "eff-aipw", n_bootstraps = 5, alpha=0.05)
print(" ace = ", ace, "\n",
      "lower quantile = ", Ql, "\n",
      "upper quantile = ", Qu)
```

```
 ace =  0.491715185866723
 lower quantile =  0.45738153193196984
 upper quantile =  0.5898403691123407
```

While Ananke has its own built-in esimation strategy for every identifiable causal effect concerning a single treatment and outcome, users may be interested in building their own estimators based on the identifying functionals/nonparametric influence functions. This allows users to use their own preferred estimation strategies such as sample splitting, using their preferred machine learning model etc. Below, we provide an example for a new causal graph reflecting some background knowledge and we are interested in the causal effect of the treatment T on the outcome Y.

[7]:
```
vertices = ['C2', 'C1', 'T', 'M1', 'M2', 'Y']
di_edges = [('C2', 'T'), ('C2', 'M1'), ('C2', 'M2'), ('C2', 'Y'),
            ('C1', 'T'), ('C1', 'M2'),
            ('T', 'M1'), ('M1', 'M2'), ('M2', 'Y'), ('T', 'Y')]
bi_edges = [('T', 'M2'), ('M1', 'Y')]
G = ADMG(vertices, di_edges, bi_edges)
influence = AutomatedIF(G, 'T', 'Y')
print("beta primal = ", influence.beta_primal_, "\n")
print("beta dual = ", influence.beta_dual_, "\n")
print("np-IF = ", influence.nonparametric_if_, "\n")
print("efficient IF = \n", influence.eff_if_, "\n")
G.draw(direction="LR")
```

```
 Treatment is p-fixable and graph is mb-shielded.

 Available estimators are:

1. Primal IPW (p-ipw)
2. Dual IPW (d-ipw)
3. APIPW (apipw)
4. Efficient APIPW (eff-apipw)
```

(continues on next page)

(continued from previous page)

```
Suggested estimator is Efficient APIPW

beta primal =  I(T=t) x 1/[p(T|C1,C2)p(M2|M1,C1,T,C2)] x _T p(T|C1,C2)p(M2|M1,C1,T,C2) x␣
↪Y

beta dual =  [p(M1|T=t,C2)/p(M1|T,C2)] x E[Y|T=t,M1,M2,C2]

np-IF =  E[primal or dual|C2] - E[primal or dual] + E[primal or dual|C1,C2] - E[primal␣
↪or dual|C2] + E[dual|C1,T,C2] - E[dual|C2,C1] + E[primal|M1,C1,T,C2] - E[primal|C2,C1,
↪T] + E[dual|M2,C1,C2,M1,T] - E[dual|C2,C1,T,M1] + E[primal|M2,C1,C2,M1,Y,T] -␣
↪E[primal|C2,C1,T,M1,M2]

efficient IF =
 E[primal or dual|C2] - E[primal or dual] + E[primal or dual|C1] - E[primal or dual] +␣
↪E[dual|C1,T,C2] - E[dual|C1,C2] + E[primal|M1,T,C2] - E[primal|T,C2] + E[dual|M1,M2,C1,
↪T,C2] - E[dual|M1,C1,T,C2] + E[primal|M1,Y,M2,T,C2] - E[primal|T,M1,M2,C2]
```

[7]:

3.8 Causal Inference with Graphical Models

Broadly speaking, in causal inference we are interested in using data from observational studies (as opposed to randomized controlled trials), in order to answer questions of the following form – What is the causal effect of setting via an **intervention** (possibly contrary to fact) some variable A to value a on some outcome Y. That is, causal inference concerns itself with the expression of counterfactual distributions obtained from observational distributions through the process of interventions. Causal effects may sometimes be determined using randomized controlled trials (RCTs), but these are often expensive or unethical – think forcing a random portion of the population to smoke to determine the effect of smoking on lung cancer. Also note that RCTs themselves may be subject to all kinds of messiness such as missing data and dropout from clinical trials so that even if data from an RCT were available, causal inference helps eliminate bias in the estimate of the causal effect arising from such messiness.

The goal of causal inference then is to determine whether such causal effects can be teased out from purely observational data or messy data. When such counterfactual quantities can be written as functions of the observed data, they are said to be **identified**. Not all causal effects can be identified, but many can – if we impose enough assumptions on the observed data distribution. Assumptions then, is the price we pay for identifiability, and so being transparent about our assumptions when making causal claims is paramount. This is where **graphical models** allow us to concisely, and intuitively state our set of assumptions. In the words of Miguel Hernán, "Graphical models allow us to draw our assumptions before our conclusions."

A graph $\mathcal{G}$ is defined by a set of vertices V and edges that could be directed (e.g., $X \rightarrow$ Y) interpreted as X being a direct cause of Y, bidirected (e.g., $X \leftrightarrow Y$) interpreted as the existence of unmeasured common causes of both X and Y ($X \leftarrow U \rightarrow Y$), or undirected (e.g., $X - Y$) interpreted as a symmetric relationship between X and Y. In Ananke, we currently support acyclic graphical models i.e., we do not allow for cyclic causality. So how exactly do graphs help us encode assumptions? Consider the **Directed Acyclic Graph (DAG)** below.

[1]:
```
# imports for this notebook

from ananke import graphs
from ananke import identification
```

(continues on next page)

(continued from previous page)

```
vertices = ['A', 'M', 'Y', 'C']
edges = [('A', 'M'), ('M', 'Y'), ('C', 'A'), ('C', 'Y')]
dag = graphs.DAG(vertices, edges)
# the direction LR is just to lay the vertices of the graph out from left to right
# instead of top to bottom which is the default
dag.draw(direction='LR')
```

[1]:

All interventional distributions in causal models of a DAG are identified by application of the g-formula. For example, $p(Y|\mathrm{do}(a))$ written as $p(Y(a))$ in potential outcomes notation which we will use more commonly here, is identified as

$$p(Y(a)) = \sum_{C,M} p(Y|M,C)p(M|A)p(C)|_{A=a}.$$

But as alluded to earlier, this comes at a price. The assumption made by a DAG (in addition to conditional independence constraints that can be read off by the graphical criterion of d-separation) is that of causal sufficiency, i.e., the absence of bidirected edges assumes the absence of unmeasured common causes. If we are unwilling to assume causal sufficiency, we can impose generalized independence constraints given by Nested Markov models of an **Acyclic Directed Mixed Graph (ADMG)** representing marginals of DAG models. The ADMG corresponding to the scenario where C is unobserved is shown below.

[2]:
```
vertices = ['A', 'M', 'Y']
di_edges = [('A', 'M'), ('M', 'Y')]
bi_edges = [('A', 'Y')]
admg = graphs.ADMG(vertices, di_edges=di_edges, bi_edges=bi_edges)
admg.draw(direction='LR')
```

[2]:

In this case $p(Y(a))$ is still identified but not all interventional distributions of an ADMG are identified. A sound and complete algorithm for identification in ADMGs is known due to Tian and Pearl, and Shpitser and Pearl. In Ananke, we use a purely graphical formulation of the aforementioned works that uses the fixing operation (ϕ) on nested Markov models, in order to answer identification queries. The following is an example of identification using Ananke for the front-door graph shown above.

[3]:
```
outcomes = ['Y']
treatments = ['A']
id_pya = identification.OneLineID(graph=admg, treatments=treatments, outcomes=outcomes)
print('Identifed =', id_pya.id(), '; Functional =', id_pya.functional())
```

```
{('M',): ['Y', 'A'], ('Y',): ['M', 'A']}
Identifed = True ; Functional = M AY(p(V);G) AM(p(V);G)
```

Another implicit assumption made in the above examples was that our data consisted of independent and identically distributed (iid) samples. **Chain Graphs (CGs)**, consisting of directed and undirected edges such that there are no partially directed cycles, have emerged as a popular graphical model of **interference** (a violation of the independence assumption). Consider a scenario in which our population consists of dyads (say couples) capable of influencing each others outcomes. This may be depicted as shown below.

[4]:
```
vertices = ['C1', 'A1', 'M1', 'Y1', 'C2', 'A2', 'M2', 'Y2']
di_edges = [('A1', 'M1'), ('M1', 'Y1'), ('C1', 'A1'), ('C1', 'Y1'),
            ('A2', 'M2'), ('M2', 'Y2'), ('C2', 'A2'), ('C2', 'Y2'),
            ('M1', 'Y2'), ('M2', 'Y1')]
ud_edges = [('M1', 'M2')]
cg = graphs.CG(vertices, di_edges=di_edges, ud_edges=ud_edges)
cg.draw(direction='LR')
```

[4]:

All interventional distributions of a CG are identified by a CG version of the g-formula. Note however, that chain graphs assume causal sufficiency (lack of bidirected edges). If we'd like to further relax this assumption, we use **Segregated Graphs (SGs)**. Once again, if we do not observe Cs in the CG shown above, we obtain the SG below. A sound and complete algorithm for identification in SGs was provided by Sherman and Shpitser.

[5]:
```
vertices = ['A1', 'M1', 'Y1', 'A2', 'M2', 'Y2']
di_edges = [('A1', 'M1'), ('M1', 'Y1'),
            ('A2', 'M2'), ('M2', 'Y2'),
            ('M1', 'Y2'), ('M2', 'Y1')]
ud_edges = [('M1', 'M2')]
bi_edges = [('A1', 'Y1'), ('A2', 'Y2')]
sg = graphs.SG(vertices, di_edges=di_edges, ud_edges=ud_edges, bi_edges=bi_edges)
sg.draw(direction='LR')
```

[5]:

We end this section by providing a hierarchy of graphical models (shown below). Two of the models in this hierarchy – bidirected graphs (BGs), and undirected graphs (UGs) are often not considered by themselves in causal analysis but are building blocks of more complicated graphical models that are, as reflected in the graph hierarchy. At the top of the hierarchy, are SGs comprised of directed, bidirected, and undirected edges. An SG with no undirected edges is an ADMG, and an SG with no bidirected edges is a CG. A BG is an ADMG with no directed edges, and a UG is a CG with no directed edges. Finally, a DAG is an ADMG with no bidirected edges *or* alternatively a CG with no undirected edges. As we go further up in the hierarchy we relax more assumptions, but identification theory becomes trickier, and so does estimation. For example, a generalized likelihood for BGs, ADMGs, and SGs is not known.

[6]:
```
# a graph of the hierarchy of graphs
vertices = ['SG', 'ADMG', 'CG', 'DAG', 'BG', 'UG']
edges = [('SG', 'ADMG'), ('SG', 'CG'), ('ADMG', 'DAG'), ('CG', 'DAG'), ('ADMG', 'BG'), (
↪'CG', 'UG')]
graphs.UG(vertices, edges).draw()
```

[6]:

3.9 Semiparametric Inference For Causal Effects

In this section, we discuss how to estimate the effect of treatment T on outcome Y commonly expressed through the use of counterfactuals of the form $Y(t)$ or $Y|\text{do}(t)$ – which reads as the potential outcome Y had treatment been assigned to t. We follow the estimation procedure outlined in our work Semiparametric Inference For Causal Effects In Graphical Models With Hidden Variables (Bhattacharya, Nabi, & Shpitser, 2020). If the outcome is continuous, the effect is typically captured by the average causal effect (ACE) defined as,

$\psi = E[Y(1)] - E[Y(0)]$, (when Y is continuous.)

When Y is binary, the effect is typically reported as the log of odds ratios (LOR) as follows.

$\psi = \log(\frac{p(Y(1)=1)/p(Y(1)=0)}{p(Y(0)=1)/p(Y(0)=0)})$, (when Y is binary.)

In Ananke, we consider identification and estimation of these quantities using the language of causal graphs. We illustrate these through a series of examples with increasing levels of complications. The good news is, all the user needs to specify is the **data (as a Pandas data frame)** and a story for what they believe the world looks like in the form of an **acyclic directed mixed graph (ADMG)** where each bidirected edge encodes unmeasured common confounders. Ananke will then provide suggestions for the best estimator given the graphical story provided by the user. If it is not possible to compute the ACE/LOR given this story (i.e., the quantity is not *identified*) then Ananke will also inform the user if this is the case.

Following are the necessary packages that need to be imported.

[1]:
```
from ananke.graphs import ADMG
from ananke.estimation import CausalEffect
from ananke.datasets import load_afixable_data, load_conditionally_ignorable_data, load_
↪frontdoor_data
from ananke.estimation import AutomatedIF
import numpy as np
```

Let's say we are studying the efficacy of a new antiretroviral therapy vs. an old one. Call this the treatment T where $T = 1$ represents receiving the new drug and $T = 0$ represents receiving the old. Our outcome of interest is the patients CD4 counts post treatment.

3.9.1 Estimation via Adjustment

As a first pass, an analyst might check the causal effect under the causal graphical model where all common confounders of the treatment and outcome are measured. A highly simplistic story would be as follows. The initial viral load of the patient affects both which treatment the patient is assigned to as well as their final outcome. This is represented graphically as follows. Just be careful not to have spaces in your variable names, the machine learning packages underlying Ananke are not compatible with that.

[2]:
```
vertices = ['ViralLoad', 'T', 'CD4']
di_edges = [('ViralLoad', 'T'), ('ViralLoad', 'CD4'), ('T', 'CD4')]
G = ADMG(vertices, di_edges)
G.draw(direction="LR")
```

[2]:

We now load some toy data on these three variables, and ask Ananke to estimate the effect. We see that as soon as we instantiate the `CausalEffect` object, Ananke offers a list of estimators that are available under the specified causal graph. Further, it recommends one that achieves the semiparametric efficiency bound, i.e., the estimator with the lowest variance.

[3]:
```
ace_obj = CausalEffect(graph=G, treatment='T', outcome='CD4')  # setting up the␣
↪CausalEffect object
```

```
 Treatment is a-fixable and graph is mb-shielded.

 Available estimators are:

1. IPW (ipw)
2. Outcome regression (gformula)
3. Generalized AIPW (aipw)
4. Efficient Generalized AIPW (eff-aipw)

Suggested estimator is Efficient Generalized AIPW
```

In this case, it recommends *Efficient Generalized AIPW* which is to say, that the estimator used to compute the effect in this case looks a lot like Augmented IPW. Further, Ananke uses semiparametric theory in order to provide an estimator that achieves the lowest asymptotic variance. Turns out, the efficient estimator in this case is trivial, but the next case demonstrates a harder scenario.

Given the list of estimators, it is up to the user to specify what estimators they want to work with. For instance, if the user decides to use the efficient generalized AIPW, they only need to use the keyword given in front of it, i.e., `eff-aipw`, when computing the effect. All the nuisance models are fit using generalized linear models provided in the

`statsmodels`. Users interested in using different modeling approaches can refer to the documentation on accessing the functional form of the influence functions at the end of this notebook.

[4]:
```
data = load_conditionally_ignorable_data() # loading the data
ace = ace_obj.compute_effect(data, "eff-aipw") # computing the effect
print("ace = ", ace, "\n")
```

```
ace =  0.9848858526281847
```

A realistic model will rarely be so simple. So what if we want to build a more complicated model such as the one shown below. Where, bidirected edges ($\leftrightarrow$) are used to encode unmeasured common confounders. For example, we hypothesize that we do not have information on some unmeasured common confounders between an individual's income and treatment assignment.

[5]:
```
vertices = ['Income', 'Insurance', 'ViralLoad', 'Education', 'T', 'Toxicity', 'CD4']
di_edges = [('ViralLoad', 'Income'), ('ViralLoad', 'T'), ('ViralLoad', 'Toxicity'),
            ('Education', 'Income'), ('Education', 'T'), ('Education', 'Toxicity'),
            ('Income', 'Insurance'), ('Insurance', 'T'), ('T', 'Toxicity'), ('Toxicity',
↪'CD4'), ('T', 'CD4')]
bi_edges = [('Income', 'T'), ('Insurance', 'ViralLoad'), ('Education', 'CD4')]
G = ADMG(vertices, di_edges, bi_edges)
G.draw(direction="LR")
```

[5]:

[6]:
```
ace_obj = CausalEffect(graph=G, treatment='T', outcome='CD4')  # setting up the ACE␣
↪object
```

```
 Treatment is a-fixable and graph is mb-shielded.

 Available estimators are:

1. IPW (ipw)
2. Outcome regression (gformula)
3. Generalized AIPW (aipw)
4. Efficient Generalized AIPW (eff-aipw)

Suggested estimator is Efficient Generalized AIPW
```

In the following, we print the outputs of all four estimators.

[7]:
```
data = load_afixable_data() # load some pre-simulated data
ace_ipw = ace_obj.compute_effect(data, "ipw")
ace_gformula = ace_obj.compute_effect(data, "gformula")
ace_aipw = ace_obj.compute_effect(data, "aipw")
ace_eff = ace_obj.compute_effect(data, "eff-aipw")
print("ace using IPW = ", ace_ipw)
print("ace using g-formula = ", ace_gformula)
print("ace using AIPW = ", ace_aipw)
print("ace using efficient AIPW = ", ace_eff)
```

```
ace using IPW =  0.4917151858666866
ace using g-formula =  0.49680779965965227
```

(continues on next page)

(continued from previous page)

```
ace using AIPW =  0.50051179297526 27
ace using efficient AIPW =  0.4917151858667226
```

In addition, the user can run bootstraps and obtain $(1-\alpha)*100\%$ confidence level for the causal effect point estimate. The user needs to specify two arguments: number of bootstraps `n_bootstraps` and the significance level α `alpha`. The confidence interval can then be obtained via bootstrap percentile intervals, as described in All of Nonparametric Statistics. In this case, the call to the `compute_effect` returns three values: the first corresponds to the effect computed on the original data, the second and third are the pre-specified lower and upper quantiles, i.e., $(\frac{\alpha}{2}, 1-\frac{\alpha}{2})$. The default value for α is set to be 0.05. We illustrate this through the following toy example.

[8]:
```
np.random.seed(0)  # setting the seed to get consistent numbers in the documentation
ace, Ql, Qu = ace_obj.compute_effect(data, "eff-aipw", n_bootstraps=5, alpha=0.05)
print(" ace = ", ace, "\n",
      "lower quantile = ", Ql, "\n",
      "upper quantile = ", Qu)
```

```
 ace =  0.4917151858667208
 lower quantile =  0.45738153193197817
 upper quantile =  0.5898403691123565
```

3.9.2 Estimation beyond Adjustment

The previous examples showed that there exists a large class of causal graphs for which we can obtain the causal effect via covariate adjustment. But what happens when covariate adjustment fails? It turns out, that there is an even wider class of models for which we can obtain the causal effect, *even* when we cannot block all backdoor paths from the treatment to the outcome. Revisiting our running example on antiretroviral therapies: Let's first see an example where the causal effect is not (non-parametrically) identified from the observed data.

[9]:
```
vertices = ['ViralLoad', 'T', 'CD4']
di_edges = [('ViralLoad', 'T'), ('ViralLoad', 'CD4'), ('T', 'CD4')]
bi_edges = [('T', 'CD4')]
G = ADMG(vertices, di_edges, bi_edges)
G.draw(direction="LR")
```

[9]:

You will notice that this differs ever so slightly yet crucially from our first example in that we as analysts now believe that there are unmeasured confounders between the treatment (T) and the outcome (CD4). If we ask Ananke to estimate the causal effect under this model of the world, it will declare (correctly) that the effect is not identified non-parametrically under the given model.

[10]:
```
ace_obj = CausalEffect(graph=G, treatment='T', outcome='CD4')
```

```
Effect is not identified!
```

So what can we do then? Well, if we believe another model wherein the effect of the treatment is mediated completely through some biologically known mechanism whose by-product we can measure. Say this occurs through a reduction in viral load, and that it addition to a baseline measurement we have a second follow-up measurement.

[11]:
```
vertices = ['ViralLoad1', 'T', 'ViralLoad2', 'CD4']
di_edges = [('ViralLoad1', 'T'), ('ViralLoad1', 'CD4'), ('ViralLoad1', 'ViralLoad2'),
            ('T', 'ViralLoad2'), ('ViralLoad2', 'CD4')]
bi_edges = [('T', 'CD4')]
```

(continues on next page)

(continued from previous page)

```
G = ADMG(vertices, di_edges, bi_edges)
G.draw(direction="LR")
```

[11]:

It so happens that this is quite similar to the front-door graph that may be familiar to some users, and the effect is now identified! Here is how we can estimate it using Ananke.

[12]:
```
ace_obj = CausalEffect(graph=G, treatment='T', outcome='CD4')  # setting up the ACE␣
↪object
data = load_frontdoor_data()
ace = ace_obj.compute_effect(data, "eff-apipw")
print("ace = ", ace)
```

```
 Treatment is p-fixable and graph is mb-shielded.

 Available estimators are:

1. Primal IPW (p-ipw)
2. Dual IPW (d-ipw)
3. APIPW (apipw)
4. Efficient APIPW (eff-apipw)

Suggested estimator is Efficient APIPW

ace =  -0.7813625198154692
```

3.9.3 Useful Graphical Criteria

In the previous section, we have seen that causal effects are not always identified, and if they are, they could be identified and estimated in different ways. Here we provide some simple graphical criteria that a user should keep in mind while creating their graphical model that may help guide them to obtain the best estimator possible. These graphical criteria and their corresponding estimators can be found in our work Semiparametric Inference For Causal Effects In Graphical Models With Hidden Variables.

Adjustment Fixability

If the treatment of interest (say T) has no bidirected path to any of its descendants (i.e., there is no $T \leftrightarrow \cdots \leftrightarrow X$ for any X that is a descendant of T), then we say the treatment is adjustment or a-fixable. In this case, Ananke will find estimators that are based on IPW and covariate adjustment as well as a semiparametric estimator that has the form of **Augmented IPW**. Ananke will always recommend **Generalized AIPW** as the preferred estimator due to its double robustness property. Further, if the graph satisfies a property known as the mb-shielded (Markov blanket shielded) property, Ananke will be able to provide the most **efficient influence function** based estimator – one that achieves the semiparametric efficiency bound, and recommend using this instead when possible. Below is an example of a causal graph where the treatment is a-fixable and the graph is mb-shielded.

[13]:
```
vertices = ['Income', 'Insurance', 'ViralLoad', 'Education', 'T', 'Toxicity', 'CD4']
di_edges = [('ViralLoad', 'Income'), ('ViralLoad', 'T'), ('ViralLoad', 'Toxicity'),
            ('Education', 'Income'), ('Education', 'T'), ('Education', 'Toxicity'),
            ('Income', 'Insurance'), ('Insurance', 'T'), ('T', 'Toxicity'), ('Toxicity',
↪'CD4'), ('T', 'CD4')]
```

(continues on next page)

(continued from previous page)

```
bi_edges = [('Income', 'T'), ('Insurance', 'ViralLoad'), ('Education', 'CD4')]
G = ADMG(vertices, di_edges, bi_edges)
ace_obj = CausalEffect(graph=G, treatment='T', outcome='CD4')  # setting up the ACE␣
↪object
G.draw(direction="LR")
```

```
 Treatment is a-fixable and graph is mb-shielded.

 Available estimators are:

1. IPW (ipw)
2. Outcome regression (gformula)
3. Generalized AIPW (aipw)
4. Efficient Generalized AIPW (eff-aipw)

Suggested estimator is Efficient Generalized AIPW
```

[13]:

Primal Fixability

If the treatment of interest (say T) has no bidirected path to any of its **children** (i.e., there is no $T \leftrightarrow \cdots \leftrightarrow X$ for any X that is a child of T), then we say the treatment is primal or p-fixable. Notice that due to the graphical criterion being related to children as opposed to all descendants, p-fixability is strictly more general than a-fixability. Further, p-fixability covers many important and popular classes of causal graphs such as the front-door graph.

When T is p-fixable, Ananke finds estimators based on Primal IPW, Dual IPW, and **Augmented Primal IPW**. Ananke will always recommend Augmented Primal IPW as the preferred estimator. As before, when the graph is mb-shielded, Ananke is also able to provide the efficient influence function based estimator and will instead recommend **Efficient Augmented Primal IPW**. Below is an example of an mb-shielded graph where T is p-fixable.

[14]:
```
vertices = ['ViralLoad', 'Income', 'T', 'Toxicity', 'Adherence', 'CD4']
di_edges = [('ViralLoad', 'T'), ('ViralLoad', 'Toxicity'), ('ViralLoad', 'Adherence'), (
↪'ViralLoad', 'CD4'),
            ('Income', 'T'), ('Income', 'Adherence'),
            ('T', 'Toxicity'), ('Toxicity', 'Adherence'), ('Adherence', 'CD4'), ('T',
↪'CD4')]
bi_edges = [('T', 'Adherence'), ('Toxicity', 'CD4')]
G = ADMG(vertices, di_edges, bi_edges)
ace_obj = CausalEffect(graph=G, treatment='T', outcome='CD4')  # setting up the ACE␣
↪object
G.draw(direction="LR")
```

```
 Treatment is p-fixable and graph is mb-shielded.

 Available estimators are:

1. Primal IPW (p-ipw)
2. Dual IPW (d-ipw)
3. APIPW (apipw)
4. Efficient APIPW (eff-apipw)
```

(continues on next page)

(continued from previous page)

```
Suggested estimator is Efficient APIPW
```

[14]:

Nested Fixability

Finally, this if the treatment is neither a-fixable nor p-fixable, Ananke checks if it is what we call nested fixable. That is, the treatment is nested fixable if running Algorithm 2 in our paper succeeds in finding an estimator. Ananke will then allow the user to use either Nested IPW or **Augmented Nested IPW** but its recommendation will be the latter. Below is an example.

[15]:
```
vertices = ['Income', 'Exercise', 'T', 'Toxicity', 'Adherence', 'CD4', 'ViralLoad']
di_edges = [('ViralLoad', 'T'), ('ViralLoad', 'CD4'), ('Income', 'T'), ('Exercise', 'CD4
↪'),
            ('T', 'Toxicity'), ('T', 'CD4'), ('Toxicity', 'Adherence'), ('Adherence',
↪'CD4')]
bi_edges = [('Income', 'Toxicity'), ('Exercise', 'Income'), ('Exercise', 'T'),
            ('ViralLoad', 'Adherence'), ('ViralLoad', 'CD4')]
G = ADMG(vertices, di_edges, bi_edges)
ace_obj = CausalEffect(graph=G, treatment='T', outcome='CD4')  # setting up the ACE␣
↪object
G.draw(direction="LR")
```

```
 Effect is identified.

 Available estimators:

1. Nested IPW (n-ipw)
2. Augmented NIPW (anipw)

Suggested estimator is Augmented NIPW
```

[15]:

Note that naturally the class of models that satisfy a-fixability, p-fixability and nested fixability are subsets of each other. That is,

$$\text{adjustment fixability} \subset \text{primal fixability} \subset \text{nested fixability}.$$

Thus, if the user wanted to use Nested IPW for a-fixable graphs, Ananke would not stop you. However, the reverse will result in an error from Ananke. Further, if the effect is not identified, Ananke will not allow the user to use any of the estimators.

3.9.4 Accessing the Functional Form of the Influence Function

For advanced users, we also provide tools to access the functional form of the influence funciton directly. If the treatment is **adjustment fixable**, the code spits out the functional form of the non-parametric influence function given in Theorem 2 in our paper. If the treatment is **primal fixable**, the code provides the functional form of the non-parametric influence function given in Theorem 11 in our paper. Further, if the graph is **mb-shielded**, we provide the respective efficient influence function. In the following examples we are interested in the effect of treatment T on outcome Y.

[16]:
```
vertices = ['Z1', 'Z2', 'C2', 'C1', 'T', 'M', 'Y']
di_edges = [('C2', 'Z1'), ('C2', 'T'), ('C2', 'M'),
            ('C1', 'Z1'), ('C1', 'T'), ('C1', 'M'),
            ('Z1', 'Z2'), ('Z2', 'T'), ('T', 'M'), ('M', 'Y'), ('T', 'Y')]
bi_edges = [('Z1', 'T'), ('Z2', 'C2'), ('C1', 'Y')]
G = ADMG(vertices, di_edges, bi_edges)
influence = AutomatedIF(G, 'T', 'Y')
print("np-IF = ", influence.nonparametric_if_, "\n")
print("beta primal = ", influence.beta_primal_, "\n")
print("efficient IF = \n", influence.eff_if_, "\n")
G.draw(direction="LR")
```

```
 Treatment is a-fixable and graph is mb-shielded.

 Available estimators are:

1. IPW (ipw)
2. Outcome regression (gformula)
3. Generalized AIPW (aipw)
4. Efficient Generalized AIPW (eff-aipw)

Suggested estimator is Efficient Generalized AIPW

np-IF =  I(T=t) x 1/p(T|C2,Z1,C1,Z2) x (Y - E[Y|T=t,C2,Z1,C1,Z2]) + E[Y|T=t,C2,Z1,C1,Z2]␣
↪-

beta primal =  I(T=t) x 1/p(T|C2,Z1,C1,Z2) x Y

efficient IF =
 E[primal|C1] - E[primal] + E[primal|T,M,Y,C1] - E[primal|T,M,C1] + E[primal|C2] -␣
↪E[primal] + E[primal|C2,Z1,C1] - E[primal|C2,C1] + E[primal|T,M,C2,C1] - E[primal|T,C2,
↪C1] + E[primal|Z1,C2,Z2] - E[primal|C2,Z1]
```

[16]:

[17]:
```
vertices = ['C2', 'C1', 'T', 'M1', 'M2', 'Y']
di_edges = [('C2', 'T'), ('C2', 'M1'), ('C2', 'M2'), ('C2', 'Y'),
            ('C1', 'T'), ('C1', 'M2'),
            ('T', 'M1'), ('M1', 'M2'), ('M2', 'Y'), ('T', 'Y')]
bi_edges = [('T', 'M2'), ('M1', 'Y')]
G = ADMG(vertices, di_edges, bi_edges)
influence = AutomatedIF(G, 'T', 'Y')
print("beta primal = ", influence.beta_primal_, "\n")
print("beta dual = ", influence.beta_dual_, "\n")
print("np-IF = ", influence.nonparametric_if_, "\n")
```

(continues on next page)

(continued from previous page)

```
print("efficient IF = \n", influence.eff_if_, "\n")
G.draw(direction="LR")
```

```
 Treatment is p-fixable and graph is mb-shielded.

 Available estimators are:

1. Primal IPW (p-ipw)
2. Dual IPW (d-ipw)
3. APIPW (apipw)
4. Efficient APIPW (eff-apipw)

Suggested estimator is Efficient APIPW

beta primal =  I(T=t) x 1/[p(T|C2,C1)p(M2|T,C2,C1,M1)] x _T p(T|C2,C1)p(M2|T,C2,C1,M1) x␣
↪Y

beta dual =  [p(M1|T=t,C2)/p(M1|T,C2)] x E[Y|T=t,M2,C2,M1]

np-IF =  E[primal or dual|C2,C1] - E[primal or dual|C1] + E[primal or dual|C1] -␣
↪E[primal or dual] + E[dual|T,C2,C1] - E[dual|C1,C2] + E[primal|T,C2,C1,M1] -␣
↪E[primal|C1,C2,T] + E[dual|T,C1,M1,M2,C2] - E[dual|C1,C2,T,M1] + E[primal|T,C1,M1,M2,Y,
↪C2] - E[primal|C1,C2,T,M1,M2]

efficient IF =
 E[primal or dual|C2] - E[primal or dual] + E[primal or dual|C1] - E[primal or dual] +␣
↪E[dual|T,C2,C1] - E[dual|C2,C1] + E[primal|T,C2,M1] - E[primal|T,C2] + E[dual|T,C2,C1,
↪M1,M2] - E[dual|T,C2,C1,M1] + E[primal|T,C2,M1,M2,Y] - E[primal|T,M2,C2,M1]
```

[17]:

3.10 Identification With Surrogates

Causal identification is usually framed as a problem where one wishes to recover some causal query $p(Y|do(a))$ from some observed data distribution $p(V)$ (Shpitser and Pearl, 2006; Huang and Valtorta, 2006). Recall $p(Y(a))$ and $p(Y|do(a))$ are equivalent but the do-notation used here is notationally cleaner.

However, recently there has been interest in the scenario where identification is performed with respect to other distributions. An analyst might have access to a set of experiments (where certain variables have been intervened to fixed values), and while the causal query might not be identified in any one experiment, jointly they might suffice. Lee et al., 2019 provide a sound and complete algorithm when the experiments are of the form $p(V \setminus X|do(x))$ for some intervened variables $X = x$.

Lee and Shpitser, 2020 (a different Lee!) extend the applicability of this algorithm by showing that it remains sound and complete for experiments which are ancestral subgraphs, with respect to the original graph under the intervention of the experiment, while recasting the results of Lee et al., 2019 using the one-line ID formulation provided in Richardson et al. 2017.

Following are the necessary packages that need to be imported.

[21]:
```
from ananke import graphs
from ananke import identification
```

Let's say that we are interested in a system represented by the following graph. We can think of $X1$ as a treatment for cardiac disease, $X2$ as a treatment for obesity (say a particular diet), W as blood pressure (which perhaps for the purposes of this toy example is influenced only by the cardiac treatment), and Y as the final health outcome. As a first pass, we may posit the following causal graph, indicating that we only believe unmeasured confounding to exist between a person's treatment assignment and diet. We are interested in the causal query represented by $p(Y|do(x_1, x_2))$.

[22]:
```
vertices = ["X1", "X2", "W", "Y"]
di_edges = [("X1", "W"), ("W", "Y"), ("X2", "Y")]
bi_edges = [("X1", "X2")]
G = graphs.ADMG(vertices, di_edges, bi_edges)
G.draw(direction="TD")
```

[22]:

If we query the `OneLineID` algorithm in Ananke, we will see that this query is indeed identified.

[23]:
```
one_id = identification.OneLineID(graph=G, treatments=['X1', 'X2'], outcomes=['Y'])
one_id.id()
```

[23]:
```
True
```

However, since we are in a clinical healthcare setting, we should expect the presence of more confounding variables. So we update our causal graph to include other hidden confounders between the treatments and outcomes.

[24]:
```
vertices = ["X1", "X2", "W", "Y"]
di_edges = [("X1", "W"), ("W", "Y"), ("X2", "Y")]
bi_edges = [("X1", "X2"), ("X1", "W"), ("X2", "Y")]
G = graphs.ADMG(vertices, di_edges, bi_edges)
G.draw(direction="TD")
```

[24]:

We can now verify that under this model, the query is not identified from the observed data distribution $p(V) := p(Y, W, X_2, X_1)$ using `OnelineID`:

[25]:
```
one_id = identification.OneLineID(graph=G, treatments=['X1', 'X2'], outcomes=['Y'])
one_id.id()
```

[25]:
```
False
```

It seems that we are stuck! However, in the next section we discuss how access to smaller subsets of experimental data on the variables involved may help us identify our causal query.

3.10.1 GID

What if we had access to experiments? We construct two experiments - one where X_1 is fixed to value x_1, and another where X_2 is fixed to value x_2. Perhaps, it is not possible to run an experiment where both X_1 and X_2 are intervened upon (financial reasons, ethical reasons, etc.) but these smaller experiments are indeed possible.

[26]:
```
vertices = ["X1", "X2", "W", "Y"]
di_edges = [("X1", "W"), ("W", "Y"), ("X2", "Y")]
bi_edges = [("X1", "X2"), ("X1", "W"), ("X2", "Y")]
G1 = graphs.ADMG(vertices, di_edges, bi_edges)
G1.fix(["X1"])
G1.draw(direction="TD")
```

[26]:

[27]:
```
vertices = ["X1", "X2", "W", "Y"]
di_edges = [("X1", "W"), ("W", "Y"), ("X2", "Y")]
bi_edges = [("X1", "X2"), ("X1", "W"), ("X2", "Y")]
G2 = graphs.ADMG(vertices, di_edges, bi_edges)
G2.fix(["X2"])
G2.draw(direction="TD")
```

[27]:

It happens that the causal query is indeed identified:

[28]:
```
g_id = identification.OneLineGID(graph=G, treatments=["X1", "X2"], outcomes=["Y"])
```

[29]:
```
g_id.id(experiments=[G1, G2])
```

[29]:
```
True
```

with corresponding identifying functional

[30]:
```
g_id.functional(experiments=[G1, G2])
```

[30]:
```
'W X2,Y p(W,X2,Y | do(X1))X1,W p(W,X1,Y | do(X2))'
```

3.10.2 AID

The astute reader will notice that perhaps we didn't need all of the experimental distributions. Rather, a margin would have sufficed - on the first experiment, we could have marginalized Y and X_2, and still achieved identification. The reason is that the intrinsic set (and its parents) would have been identified anyways. Based on the results provided in Lee and Shpitser, 2020, we consider identification from the following experiment. The first graph is an ancestral subgraph with respect to $G(V(x_1))$. The second graph remains unchanged, as does the graph defining the system we are interested in.

[31]:
```
vertices = ["X1","W"]
di_edges = [("X1", "W")]
bi_edges = [("X1", "W")]
G1 = graphs.ADMG(vertices, di_edges, bi_edges)
G1.fix(["X1"])
G1.draw(direction="TD")
```

[31]:

[32]:
```
a_id = identification.OneLineAID(graph=G, treatments=["X1", "X2"], outcomes=["Y"])
```

[33]:
```
a_id.id(experiments=[G1, G2])
```

[33]:
```
True
```

[34]:
```
a_id.functional(experiments=[G1, G2])
```

[34]:
```
'W  p(W | do(X1))X1,W p(W,X1,Y | do(X2))'
```

The causal query remains identified, but the identification formula has changed. Notably, no fixing operations are needed under the first experiment $p(W|do(X1))$ since it is exactly the required kernel.

3.11 Linear Structural Equation Models

The use of structural equation models and path analysis in causal models associated with a graph can be traced back to the works of Sewall Wright and Trygve Haavelmo. In this notebook we demonstrate how graphs can be used to postulate causal mechanisms in the presence of confounding, and provide examples of model selection, and computation of causal effects, when the causal mechanism arises as a result of a linear system of structural equations.

Given an ADMG $\mathcal{G}$, the linear structural equation model (SEM) with correlated errors (corresponding to unmeasured confounding) associated with $\mathcal{G}$ is defined as the set of multivariate normal distributions with covariance matrices of the form

$\Sigma = (IB)^{-T}\Omega(IB)^{-1}$,

where $\omega_{ij} = \omega_{ji} = 0$ unless $i \leftrightarrow j$ exists in $\mathcal{G}$, and $b_{ij} = 0$ unless $j \rightarrow i$ exists in $\mathcal{G}$. The matrix Ω and therefore Σ is assumed to be positive semi-definite. Since $\mathcal{G}$ is acyclic, B is assumed to be lower triangular.

Following are the packages that need to be imported for the examples below.

[1]:
```
from ananke.graphs import ADMG
from ananke.models import LinearGaussianSEM
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore')
```

3.11.1 Arid Graphs

Before going further, it is important to define the class of graphs that we will be working with because statistical models associated with arbitrary Acyclic Directed Mixed Graphs (ADMGs) are not always identified. However, it has been shown by Drton, Foygel, and Sullivant that SEMs associated with a certain class of ADMGs that lack convergent aborescences or C-trees, hence the coinage arid graphs, are everywhere identified. Here, we will focus on linear SEMs associated with **Maximal Arid Graphs (MArGs)** introduced by Shpitser, Evans, and Richardson and use the maximal arid projection defined therein, in order to convert a non-arid ADMG to the corresponding MArG that implies the same conditional and nested Markov constraints.

Consider the non-arid graph $\mathcal{G}$ below.

[2]:
```
# create a non-arid graph
# the bow arc A->C, A<->C is a C-tree
vertices = ["A", "B", "C", "D"]
di_edges = [("A", "B"), ("A", "C"), ("B", "C"), ("C", "D")]
bi_edges = [("A", "C"), ("B", "D")]
G = ADMG(vertices, di_edges=di_edges, bi_edges=bi_edges)
G.draw(direction="LR")
```

[2]:

We can obtain the corresponding maximal arid projection $\mathcal{G}^{\dagger}$ like so.

[3]:
```
G_arid = G.maximal_arid_projection()
G_arid.draw(direction="LR")
```

[3]:

3.11.2 Fitting and Causal Effect Estimation

We now demonstrate how to fit data originating from a system of linear structural equation models associated with an ADMG, as defined in the first section of the notebook. We first generate data according to a linear SEM associated with the MArG $\mathcal{G}^{\dagger}$ above.

[4]:
```
# define number of samples and number of vertices
N = 5000
dim = 4

# define the Omega matrix
omega = np.array([[1, 0, 0, 0],
                  [0, 1, 0, 0.8],
                  [0, 0, 1, 0],
                  [0, 0.8, 0, 1]])

# define the B matrix
beta = np.array([[0, 0, 0, 0],
                 [3, 0, 0, 0],
                 [1.2, -1, 0, 0],
                 [0, 0, 2.5, 0]])


# generate data according to the graph
true_sigma = np.linalg.inv(np.eye(dim) - beta) @ omega @ np.linalg.inv((np.eye(dim) -␣
↪beta).T)
X = np.random.multivariate_normal([0] * dim, true_sigma, size=N)
data = pd.DataFrame({"A": X[:, 0], "B": X[:, 1], "C": X[:, 2], "D": X[:, 3]})
```

We then check that we are able to recover the true parameters of the data generating distribution. The **LinearGaussianSEM** class has a special draw functionality to draw the associated MArG and label the parameters corresponding to each edge.

[5]:
```
model = LinearGaussianSEM(G_arid)
model.fit(data)
model.draw(direction="LR")
```

[5]:

We now compute some causal effects through simple path analysis. Consider the total causal effect of D on A and A on D. The former is zero as expected, and the latter is obtained as the sum of contributions through all directed paths from A to D and should be close to the true value of -4.5.

[6]:
```
print("E[A(d)] =", str(model.total_effect(["D"], ["A"])))
print("E[D(a)] =", str(model.total_effect(["A"], ["D"])))
```

```
E[A(d)] = 0
E[D(a)] = -4.495622612107653
```

To demonstrate what could go wrong when we use a non-arid graph. Consider fitting the same data utilizing a model associated with the non-arid graph $\mathcal{G}$ from before. We now see that our coefficient and causal estimates are now completely awry.

[7]:
```
model = LinearGaussianSEM(G)
model.fit(data)
```

(continues on next page)

(continued from previous page)

```
print("E[D(a)] =", str(model.total_effect(["A"], ["D"])))
model.draw(direction="LR")
```

```
E[D(a)] = -5.986677143661288
```

[7]:

3.11.3 Model Selection

How do we pick the correct graphical model? Linear Gaussian SEMs associated with MArGs are **curved exponential families** and thus we can use the BIC score to test for goodness of fit. The graph $\mathcal{G}^\dagger$ implies a Verma constraint (a dormant conditional independence) where $A \amalg D$ in the kernel and associated CADMG obtained by fixing C. More formally,

$$\phi_C\left(p(V); \mathcal{G}^\dagger\right) \equiv \frac{p(A,B,C,D)}{p(C|\text{mb}(C))} = p(A,B)p(D|C),$$

where $\text{mb}(C)$ is the Markov blanket of C. Consider a graph $\mathcal{G}_s$ which is the same as $\mathcal{G}^\dagger$ but with an additional directed edge $A \to D$. There are no conditional independences or Vermas implied by $\mathcal{G}_s$. That is, the model associated with $\mathcal{G}_s$ is a super model of the true model and contains one extra parameter associated with the extra directed edge. When choosing between these two hypotheses, the BIC should prefer $\mathcal{G}^\dagger$ as it is more parsimonious. Let's look at the comparison.

[8]:
```
di_edges = [("A", "B"), ("A", "C"), ("B", "C"), ("C", "D"), ("A", "D")]
bi_edges = [("B", "D")]
G_saturated = ADMG(vertices, di_edges=di_edges, bi_edges=bi_edges)

# calculate BIC for the true model
model_truth = LinearGaussianSEM(G_arid)
model_truth.fit(data)
bic_truth = 2*model_truth.neg_loglikelihood(data) + model_truth.n_params*np.log(N)

# calculate BIC for the saturated model
model_saturated = LinearGaussianSEM(G_saturated)
model_saturated.fit(data)
bic_saturated = 2*model_saturated.neg_loglikelihood(data) + model_saturated.n_params*np.
↪log(N)

print("BIC of the true model", bic_truth)
print("BIC of the saturated model", bic_saturated)
print("The BIC of the true model should be lower (better) than the saturated model.",␣
↪bic_truth < bic_saturated)
```

```
BIC of the true model 15050.380761997665
BIC of the saturated model 15058.78346569897
The BIC of the true model should be lower (better) than the saturated model. True
```

What if we pick a (slightly) incorrect model where we posit that the observed correlation between B and D is due to a directed edge $B \to D$ instead of confounding? The BIC should prefer any super model of the truth to an incorrect model.

[9]:
```
di_edges = [("A", "B"), ("B", "C"), ("C", "D"), ("B", "D"), ("A", "C")]
bi_edges = []
G_incorrect = ADMG(vertices, di_edges=di_edges, bi_edges=bi_edges)
```

(continues on next page)

(continued from previous page)

```
# calculate BIC for incorrect model
model_incorrect = LinearGaussianSEM(G_incorrect)
model_incorrect.fit(data)
bic_incorrect = 2*model_incorrect.neg_loglikelihood(data) + model_incorrect.n_params*np.
↪log(N)

print("BIC of the incorrect model", bic_incorrect)
print("The BIC of the true model should be lower (better) than the incorrect model.",␣
↪bic_truth < bic_incorrect)
model_incorrect.draw(direction="LR")
```

```
BIC of the incorrect model 19516.512118364873
The BIC of the true model should be lower (better) than the incorrect model. True
```

[9]:

Finally, we consider the most interesting scenario – a model that encodes the exact same nested Markov constraints. It turns out that two nested Markov equivalent models will be statistically indistinguishable and their BIC scores will be exactly the same.

[10]:
```
di_edges = [("A", "C"), ("B", "C"), ("C", "D")]
bi_edges = [("A", "B"), ("B", "D")]
G_equivalent = ADMG(vertices, di_edges=di_edges, bi_edges=bi_edges)

# calculate BIC for the nested Markov equivalent model
model_equivalent = LinearGaussianSEM(G_equivalent)
model_equivalent.fit(data)
bic_equivalent = 2*model_equivalent.neg_loglikelihood(data) + model_equivalent.n_
↪params*np.log(N)

print("BIC of the equivalent model", bic_equivalent)
print("The difference in BIC =",
      round(abs(bic_equivalent - bic_truth), 2),
      "between equivalent models should be indistinguishable.")
```

```
BIC of the equivalent model 15050.380764500856
The difference in BIC = 0.0 between equivalent models should be indistinguishable.
```

What we've shown here is that the BIC score can be used to select between different hypotheses. In DAG models, a greedy search procedure was posited by Chickering that surprisingly yields a globally optimal DAG model while greedily searching through the space of equivalence classes of DAGs. Such a greedy search procedure for nested Markov models is still an open problem – indeed the question of nested Markov equivalence in itself is still an open problem.

3.12 Maximum Likelihood for Binary Data ADMGs

This tutorial introduces maximum likelihood fitting for discrete data ADMGs, using the Mobius inverse parameterization. Compared to latent variable DAGs, these models are identified everywhere in the parameter space, and do not contain singularities.

In this implementation, we consider binary data ADMGs. We hope to implement general discrete data modelling capabilities at a future time.

- [1] T. S. Richardson, R. J. Evans, J. M. Robins, and I. Shpitser, "Nested Markov Properties for Acyclic Directed Mixed Graphs," arXiv:1701.06686 [stat], Jan. 2017, Accessed: Nov. 24, 2019. [Online]. Available:

http://arxiv.org/abs/1701.06686

- [2] R. J. Evans and T. S. Richardson, "Maximum likelihood fitting of acyclic directed mixed graphs to binary data," arXiv:1203.3479 [cs, stat], Mar. 2012, Accessed: Sep. 04, 2018. [Online]. Available: http://arxiv.org/abs/1203.3479
- [3] Hauser, R.M., Sewell, W.H. and Herd, P. Wisconsin Longitudinal Study (WLS), 1957–2012. Available at http://www.ssc.wisc.edu/wlsresearch/documentation/. Version 13.03, Univ. Wisconsin–Madison, WLS.
- [4] R. J. Evans and T. S. Richardson, "Smooth, identifiable supermodels of discrete DAG models with latent variables," Bernoulli, vol. 25, no. 2, pp. 848–876, May 2019, doi: 10.3150/17-BEJ1005.

[1]:
```
import pandas as pd
import numpy as np
from ananke.graphs import ADMG
from ananke.models import binary_nested
from ananke.datasets import load_wisconsin_health_study
```

[2]:
```
%load_ext autoreload

%autoreload 2
```

We transcribe data from [4] below. This represents data from the Wisconsin Longitudinal Study [3]. Variables measured are: * X: an indicator of whether family income in 1957 was above $5k; * Y: an indicator of whether the respondents income in 1992 was above $37k; * M: an indicator of whether the respondent was drafted into the military; * E: an indicator of whether the respondent had education beyond high school.

[3]:
```
graph = ADMG(vertices=['X', 'E', 'M', 'Y'], di_edges=[('X', 'E'), ('E', 'M'), ('X', 'Y
↪')], bi_edges=[('E', 'Y')])
```

[4]:
```
graph.draw()
```

[4]:

[5]:
```
df = load_wisconsin_health_study()
```

[6]:
```
df.head()
```

[6]:

	X	E	M	Y	count
0	0	0	0	0	241
1	0	0	0	1	162
2	0	0	1	0	53
3	0	0	1	1	39
4	1	0	0	0	161

3.12.1 Investigating the causal effect of X on Y

As detailed in [4], we investigate the causal effect of family income in 1957 on respondent income in 1992.

[7]:
```
X = binary_nested.process_data(df=df, count_variable='count')
```

[8]:
```
X
```

[8]:
```
array([241, 162, 161, 148,  53,  39,  33,  29,  82, 176, 113, 364,  13,
        16,  16,  30])
```

[9]:
```
bnm = binary_nested.BinaryNestedModel(graph)
```

[10]:
```
bnm = bnm.fit(X=X)
```

```
/Users/jaron/Projects/ananke/ananke/models/binary_nested.py:511: RuntimeWarning: invalid␣
↪value encountered in log
  logger.debug(np.log(A @ x - b))
/Users/jaron/Projects/ananke/ananke/models/binary_nested.py:512: RuntimeWarning: invalid␣
↪value encountered in log
  lld = np.sum(counts * np.log(A @ x - b))
```

[11]:
```
bnm.fitted_params
```

[11]:
```
OrderedDict([((frozenset({'E'}), ('X',), (0,)), 0.6329923225577604),
             ((frozenset({'E'}), ('X',), (1,)), 0.41498894592949687),
             ((frozenset({'E', 'Y'}), ('X',), (0,)), 0.37595907634663206),
             ((frozenset({'E', 'Y'}), ('X',), (1,)), 0.21700230650140587),
             ((frozenset({'M'}), ('E',), (0,)), 0.8221708988986646),
             ((frozenset({'M'}), ('E',), (1,)), 0.9074074044189431),
             ((frozenset({'X'}), (), ()), 0.4665871121546948),
             ((frozenset({'Y'}), ('X',), (0,)), 0.49744245337806176),
             ((frozenset({'Y'}), ('X',), (1,)), 0.36129757488088415)])
```

We attempt to estimate the average causal effect of X on Y. This is given by

$$p(Y = 1|do(X = 1)) - p(Y = 1|do(X = 0)) = (1 - \theta_Y(X = 1)) - (1 - \theta_Y(X = 0))$$

[12]:
```
pY1_X1 = (1 - bnm.fitted_params[(frozenset({'Y'}), ('X', ), (1,))])
pY1_X1
```

[12]:
```
0.6387024251191158
```

[13]:
```
pY1_X0 = (1 - bnm.fitted_params[(frozenset({'Y'}), ('X', ), (0,))])
pY1_X0
```

[13]:
```
0.5025575466219383
```

[14]:
```
pY1_X1 - pY1_X0
```

[14]:
```
0.1361448784971775
```

Interestingly, we do not recover the result in [4], in which $p(Y = 1|do(X = 1)) = 0.50$ and $p(Y = 1|do(X = 0)) = 0.36$. However, we obtain a similar looking result if $X \equiv 1 - X$ and $Y \equiv 1 - Y$:

[15]:
```
df_flip = df.copy()
```

[16]:
```
df_flip["X"] = 1 - df_flip["X"]
df_flip["Y"] = 1 - df_flip["Y"]
```

[17]:
```
bnm_flip = binary_nested.BinaryNestedModel(graph)
X_flip = binary_nested.process_data(df=df_flip, count_variable='count')
bnm_flip = bnm.fit(X=X_flip)
```

[18]:
```
pY1_X1_flip = (1 - bnm_flip.fitted_params[(frozenset({'Y'}), ('X', ), (1,))])
pY1_X0_flip = (1 - bnm_flip.fitted_params[(frozenset({'Y'}), ('X', ), (0,))])

```

[19]:
```
pY1_X1_flip
```

[19]:
```
0.4974424530066138
```

[20]:
```
pY1_X0_flip
```

[20]:
```
0.36129757712073496
```

Given the unit testing coverage of ananke we have reason to believe that some error has occurred in the application section of [3].

3.12.2 Analysing the effect of E on Y

While investigating the causal effect of X on Y is interesting, it does not lead to policy changes since intervening on income in the past is not feasible. However, intervening on whether a student has education after high school is something that can feasibly be affected through policy changes. Thus, we consider the effect of X on Y.

[21]:
```
graph = ADMG(vertices=['E', 'M', 'Y'],
             di_edges=[('E', 'M'), ('M', 'Y')],
             bi_edges=[('E', 'Y')])
bnm = binary_nested.BinaryNestedModel(graph)
```

[22]:
```
df = load_wisconsin_health_study().groupby(["E", "M", "Y"])["count"].sum().reset_index()
```

[23]:
```
df
```

[23]:

	E	M	Y	count
0	0	0	0	402
1	0	0	1	310
2	0	1	0	86
3	0	1	1	68
4	1	0	0	195
5	1	0	1	540
6	1	1	0	29
7	1	1	1	46

[24]:
```
df["E"].mean()
```

[24]:
```
0.5
```

[25]:
```
df.groupby(["M", "Y"])["count"].sum()
```

[25]:
```
M  Y
0  0    597
   1    850
1  0    115
   1    114
Name: count, dtype: int64
```

[26]:
```
graph.draw()
```

[26]:

[27]:
```
bnm = bnm.fit(X=df, tol=1e-12)
```

```
/Users/jaron/Projects/ananke/ananke/models/binary_nested.py:511: RuntimeWarning: invalid␣
↪value encountered in log
  logger.debug(np.log(A @ x - b))
/Users/jaron/Projects/ananke/ananke/models/binary_nested.py:512: RuntimeWarning: invalid␣
↪value encountered in log
  lld = np.sum(counts * np.log(A @ x - b))
```

We attempt to estimate the average causal effect of E on Y. Recall that the front-door identification formula in general is given as

$$p(Y(e)) = \sum_m q(Y|M = m, E = e)q(M = m|E = e)$$

where q here denotes Markov kernels. It turns out that $q(Y|M, E)$ is not a function of E after M has been fixed, so this is equal to $q(Y|M)$. Then,

$$p(Y(e)) = \sum_m q(Y|M = m)q(M = m|E = e)$$

Furthermore, note that the Mobius parameters are kernels which are evaluated at the reference value (zero).

$$p(Y = 1|do(X = E)) = (1 - \theta_Y(M = 0))(\theta_M(E = e)) + (1 - \theta_Y(M = 1)) * (1 - \theta_M(E = e))$$

[28]:
```
bnm.fitted_params
```

[28]:
```
OrderedDict([((frozenset({'E'}), (), ()), 0.5167064529214306),
             ((frozenset({'E', 'Y'}), ('M',), (0,)), 0.2917359440408035),
             ((frozenset({'E', 'Y'}), ('M',), (1,)), 0.2885503638292143),
             ((frozenset({'M'}), ('E',), (0,)), 0.8221709018146659),
             ((frozenset({'M'}), ('E',), (1,)), 0.907407405514132),
             ((frozenset({'Y'}), ('M',), (0,)), 0.4199566848971338),
             ((frozenset({'Y'}), ('M',), (1,)), 0.47542389281964714)])
```

[29]:
```
theta_M0_E0 = bnm.fitted_params[(frozenset({'M'}), ('E',), (0,))]
theta_M0_E1 = bnm.fitted_params[(frozenset({'M'}), ('E',), (1,))]
theta_Y0_M0 = bnm.fitted_params[(frozenset({'Y'}), ('M',), (0,))]
theta_Y0_M1 = bnm.fitted_params[(frozenset({'Y'}), ('M',), (1,))]
```

[30]:
```
pY1_E0 = (1 - theta_Y0_M0) * theta_M0_E0 + (1 - theta_Y0_M1) * (1 - theta_M0_E0)
```

[31]:
```
pY1_E1 = (1 - theta_Y0_M0) * theta_M0_E1 + (1 - theta_Y0_M1) * (1 - theta_M0_E1)
```

[32]:
```
pY1_E1 -  pY1_E0
```

[32]:
```
0.004727830873286432
```

[ ]:
```

```

3.13 Discrete Models for Identification Algorithm Development

In causal inference we are often concerned with performing identification, which is determining whether a causal query is a function of the observed data. Algorithms have been developed to address these questions in a variety of settings. When developing new algorithms for identification, the first aspect is soundness - whether an algorithm correctly returns the causal effect.

In this notebook we outline some helpful tools intended to assist in the development of such algorithms.

We'll first go through setting up causal models using `ananke.models.BayesianNetwork`, which supports both numerical and symbolic representations of conditional probability distributions through `pgmpy.factors.discrete.TabularCPD` and `ananke.factors.SymCPD` respectively.

Given this causal model, we can directly introduce an intervention, and then compute the causal query in the intervened distribution. Then, we can pass the observed data distribution from the causal model into a proposed identification algorithm, and verify that the result agrees with the computed truth.

This development pattern should be useful for both checking correctness of proposed identification algorithms, and also serve as a concrete implementation for understanding how causal identification works.

[27]:
```
import sympy as sp

from pgmpy.inference import VariableElimination
from pgmpy.models import BayesianNetwork

from ananke.graphs import ADMG, DAG
from ananke.identification import OneLineID
from ananke.models import bayesian_network
from ananke.estimation import empirical_plugin
from ananke.identification import oracle
from ananke.inference.variable_elimination import variable_elimination
```

3.13.1 Front-door graph

We first consider the simple case of the front-door graph. In this example we demonstrate the API which implements the ID algorithm (Shpitser and Pearl, 2006).

We initialize an ADMG representing this graph, and provide variable cardinalities.

[28]:
```
di_edges = [("A", "M"), ("M", "Y")]
bi_edges = [("A", "Y")]
```

(continues on next page)

(continued from previous page)

```
graph = ADMG(vertices={"A": 2, "M": 2, "Y": 2}, di_edges=di_edges, bi_edges=bi_edges)
graph.draw()
```

[28]:

This distribution is an ADMG, which has bidirected edges representing unmeasured confounding. To continue, we convert this ADMG into its canonical DAG form, which replaces each bidirected edge with a single hidden variable.

[29]:
```
dag = graph.canonical_dag(cardinality=2)
```

[30]:
```
dag.draw()
```

[30]:

Next, we generate conditional probability distributions consistent with this graph. This provides a causal model with factorization

$$p(A, M, Y, U_{A,Y}) = p(Y|M, U_{A,Y})p(A|U_{A,Y})p(M|A)p(U_{A,Y})$$

We represent this as `net`.

[31]:
```
cpds = bayesian_network.generate_random_cpds(graph=dag, dir_conc=10)
net = bayesian_network.BayesianNetwork(graph=dag, cpds=cpds)
```

We are next interested in a causal model where $do(A = 1)$ has been implemented. Then,

$$p(Y(a = 1), M(a = 1), U_{A,Y}(a = 1)) = p(Y|M, U_{A,Y})p(M|A = 1)p(U_{A,Y})$$

We represent this as `int_net`.

[32]:
```
treatment_dict = {"A": 1}
outcome_dict = {"Y": 1}
```

In this model, the causal effect $p(Y(a = 1) = 1)$ is simply the marginal distribution evaluated at $Y = 1$:

[33]:
```
truth = oracle.compute_effect_from_discrete_model(net, treatment_dict, outcome_dict)
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

We can also compute the causal effect directly from the observed data distribution.

[34]:
```
oid = OneLineID(graph, list(treatment_dict), list(outcome_dict))
```

[35]:
```
obs_dist = variable_elimination(net, ["A", "M", "Y"])
```

```
  0%|          | 0/1 [00:00<?, ?it/s]
```

[36]:
```
result = empirical_plugin.estimate_effect_from_discrete_dist(
    oid, obs_dist, treatment_dict, outcome_dict
)
```

```
INFO:ananke.estimation.empirical_plugin:implied district is {'M'}
INFO:ananke.estimation.empirical_plugin:fixing by q(Y|M, A)
INFO:ananke.estimation.empirical_plugin:fixing by q(A|Y)
INFO:ananke.estimation.empirical_plugin:implied district is {'Y'}
INFO:ananke.estimation.empirical_plugin:fixing by q(M|A)
INFO:ananke.estimation.empirical_plugin:fixing by q(A|M, Y)
```

[37]:
```
print("Effect computed through intervention of model:  ", truth)
print("Effect computed through identification algorithm: ", result)
```

```
Effect computed through intervention of model:   0.5971619133337924
Effect computed through identification algorithm:  0.5971619133337924
```

The result agrees with the truth up to floating point precision.

Using symbolic computation

We can also perform these computations using the `ananke.factors.SymCPD`, which allows each conditional probability distribution to be specified using symbols (rather than numerical values).

[38]:
```
cpds, all_vars = bayesian_network.create_symbolic_cpds(dag)
net = bayesian_network.BayesianNetwork(graph=dag, cpds=cpds)
```

[39]:
```
truth = oracle.compute_effect_from_discrete_model(net, treatment_dict, outcome_dict)
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

[40]:
```
truth
```

[40]: $0.5q_{M01} \cdot (1 - q_{Y000}) + 0.5q_{M01} \cdot (1 - q_{Y001}) + 0.5 \cdot (1 - q_{M01})\,(1 - q_{Y010}) + 0.5 \cdot (1 - q_{M01})\,(1 - q_{Y011})$

[41]:
```
obs_dist = variable_elimination(net, ["A", "M", "Y"])
```

```
  0%|          | 0/1 [00:00<?, ?it/s]
```

[42]:
```
result = empirical_plugin.estimate_effect_from_discrete_dist(
    oid, obs_dist, treatment_dict, outcome_dict
)
```

```
INFO:ananke.estimation.empirical_plugin:implied district is {'M'}
INFO:ananke.estimation.empirical_plugin:fixing by q(Y|M, A)
INFO:ananke.estimation.empirical_plugin:fixing by q(A|Y)
INFO:ananke.estimation.empirical_plugin:implied district is {'Y'}
INFO:ananke.estimation.empirical_plugin:fixing by q(M|A)
INFO:ananke.estimation.empirical_plugin:fixing by q(A|M, Y)
```

[43]:
```
result
```

[43]: $$\frac{\left(q_{M01}\left(\frac{q_{Y000}\cdot(1-q_{A00})}{2}+\frac{q_{Y001}\cdot(1-q_{A01})}{2}\right)+q_{M01}\left(\frac{(1-q_{A00})(1-q_{Y000})}{2}+\frac{(1-q_{A01})(1-q_{Y001})}{2}\right)\right)\left(\frac{q_{M00}\left(\frac{q_{A00}\cdot(1-q_{Y000})}{2}+\frac{q_{A01}\cdot(1-q_{Y001})}{2}\right)}{}\right.}{\left((1-q_{M01})\left(\frac{q_{Y010}\cdot(1-q_{A00})}{2}+\frac{q_{Y011}\cdot(1-q_{A01})}{2}\right)+(1-q_{M01})\left(\frac{(1-q_{A00})(1-q_{Y010})}{2}+\frac{(1-q_{A01})(1-q_{Y011})}{2}\right)\right)\left(\frac{(1-q_{M00})\left(\frac{q_{A00}\cdot(1-q_{Y010})}{2}+\right.}{}\right.}$$

To check that the truth and result agree, we can take the difference and simplify it, and if it is zero then they agree:

[44]:
```
sp.simplify(result - truth)
```

[44]:
```
0
```

3.13.2 Conditional Ignorability

We demonstrate this example to show off how a user might compute identification queries using a lower level interface. This is useful if the developed identification algorithm does not align with the format of the ID algorithm.

[45]:
```
di_edges = [("A", "Y"), ("C", "Y"), ("C", "A")]
bi_edges = []
c_graph  = ADMG(vertices={"A": 2, "C": 2, "Y": 2}, di_edges=di_edges, bi_edges=bi_edges)
c_graph.draw()
```

[45]:

[46]:
```
c_dag = c_graph.canonical_dag(cardinality=2)
```

[47]:
```
cpds = bayesian_network.generate_random_cpds(graph=c_dag, dir_conc=10)
net = bayesian_network.BayesianNetwork(graph=c_dag, cpds=cpds)
```

We first set up an intervened version of the model, with the desired intervention $do(A = 1)$.

[48]:
```
treatment_dict = {"A": 1}
outcome_dict = {"Y": 1}

int_net = net.copy()
int_net.fix(treatment_dict)
```

[48]:
```
<ananke.models.bayesian_network.BayesianNetwork at 0x287e44400>
```

Then we compute the marginal of Y in this distribution which is the causal parameter of interest $p(Y(a = 1))$. This gives us the truth.

[49]:
```
truth = variable_elimination(int_net, ['Y']).get_value(**outcome_dict)
```

```
  0%|          | 0/2 [00:00<?, ?it/s]
```

To compute the causal effect another way, we use only the observed data distribution and implement the g-formula result:

$$p(Y(a = 1)) = \sum_C p(Y|A = 1, C)p(C)$$

[51]:
```
p_YAC = variable_elimination(net, ['Y', 'A', 'C'])
p_Y_AC = p_YAC.divide(p_YAC.marginalize(["Y"], inplace=False), inplace=False)
p_Y_A1C = p_Y_AC.reduce([("A", 1)], inplace=False)
p_C = p_YAC.marginalize(["A", "Y"], inplace=False)
p_Y_do_A1 = p_Y_A1C.product(p_C, inplace=False).marginalize(["C"], inplace=False)
```

```
0it [00:00, ?it/s]
```

[52]:
```
result = p_Y_do_A1.get_value(**outcome_dict)
```

[53]:
```
print("Effect computed through intervention of model:  ", truth)
print("Effect computed through identification algorithm: ", result)
```

```
Effect computed through intervention of model:   0.5890045340192529
Effect computed through identification algorithm:  0.589004534019253
```

Again, the result agrees with the truth up to floating point precision.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[LBNS23] Jaron J. R. Lee, Rohit Bhattacharya, Razieh Nabi, and Ilya Shpitser. Ananke: a python package for causal inference using graphical models. 2023. arXiv:2301.11477.

[BNS20] Rohit Bhattacharya, Razieh Nabi, and Ilya Shpitser. Semiparametric inference for causal effects in graphical models with hidden variables. *arXiv preprint arXiv:2003.12659*, 2020.

[LS20] Jaron J. R. Lee and Ilya Shpitser. Identification Methods With Arbitrary Interventional Distributions as Inputs. *arXiv preprint arXiv:2004.01157 [cs, stat]*, 2020.

[NBS20] Razieh Nabi, Rohit Bhattacharya, and Ilya Shpitser. Full law identification in graphical models of missing data: completeness results. *arXiv preprint arXiv:2004.04872*, 2020.

PYTHON MODULE INDEX

a

INDEX

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V