matching

Author

Apr 26, 2022

CONTENTS:

CHAPTER

ONE

MATCHING PACKAGE

1.1 Submodules

1.2 matching.balance module

matching.balance.**ecdf**(*x: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray[numpy.dtype bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]]*)

Fits an eCDF to *x* and returns a callable to evaluate it at any point(s)

Parameters **x** (*ArrayLike*) – to fit the eCDF to

Returns **inner** – evaluates the eCDF at point *v*

Return type Callable

matching.balance.**standardized_mean_difference**(*a: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_lik bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *b: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_lik bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*)

Computes the standardized mean difference between a and b.

rac{overline{a} - overline{b}}{sqrt{ ext{Var}(a) + ext{Var}(b)}}

a, b [npt.ArrayLike] to compute the standardized mean difference between

float standardized mean difference

matching.balance.**variance_ratio**(*a: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *b: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*) → float

Compute the variance ratio between *a* and *b*, defined as

rac{ ext{Var}(a)}{ ext{Var}(b)}

a, b [ArrayLike] To compute the variance ratio between

float variance ratio

1.3 matching.dataset module

class matching.dataset.**MatchingDataset**(*X: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._Suppo bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *z: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._Suppo bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *ids: Optional[Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._Suppo bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]] = None*, *treatment_encoded_as: Hashable = True*)

Bases: `object`

Container that validates inputs to be used in bipartite matching

X

of features to match on

Type DataFrame

z

of boolean treatment assignments

Type Series

ids

of observation IDs. Note that these are coerced to *str*

Type Series

X: pandas.core.frame.DataFrame

__init__(*X: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *z: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *ids: Optional[Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]] = None*, *treatment_encoded_as: Hashable = True*)

Parameters

- **X** (*ArrayLike*) – of features to match on. Will be converted to a *pd.DataFrame*
- **z** (*ArrayLike*) – of boolean treatment assignments. Will be converted to a *pd.Series*
- **ids** (*ArrayLike, default=None*) – of observation IDs. Will be converted to a *pd.Series*, and coerced to type *str*
- **treatment_encoded_as** (*Hashable, default=True*) – Value in z corresponding to treatment group membership.

property arrs: matching.dataset._MatchingDatasetNDArrays

Numpy array versions of *X*, *z*, and *ids*. Non-numeric columns of *X* are one-hot encoded

property control: matching.dataset._MatchingDataset

Dataset of just the control group

copy(*deep: bool = False*) → matching.dataset._MatchingDataset

Make a copy of this dataset.

Parameters **deep** (*bool*) – *False* by default. Whether or not to make a deep copy

Returns copy

Return type _MatchingDataset

property frame: pandas.core.frame.DataFrame

Dataframe representation of this dataset

groupby(*by: Union[Hashable, Iterable[Hashable], numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, **args*, ***kwargs*) → Iterator[matching.dataset._MatchingDataset]

Works like *pd.DataFrame.groupby*, except it yields _MatchingDataset instead

Parameters

- **by** (*Union[Hashable, Iterable[Hashable], npt.ArrayLike]*) – A label, sequence of labels, or indexing array to supply to *pd.DataFrame.groupby*
- ***args** – Passed to *pd.DataFrame.groupby*
- ****kwargs** – Passed to *pd.DataFrame.groupby*

Yields _MatchingDataset – Note that no copy is made

ids: pandas.core.series.Series

property nobs: int

Total number of observations associated with this dataset

summary() → pandas.core.frame.DataFrame

Print a summary of the matching _MatchingDataset.

Metrics include: - min, 25%, 50%, 75%, max; - mean; - standard deviation; - standardized mean difference; - variance ratio; - eCDF of mean.

Returns Note that this is multi-indexed by feature name, then metric (i.e. "min"). Columns represent treatment assignment

Return type pd.DataFrame

property treatment: matching.dataset._MatchingDataset

Dataset of just the treatment group

z: pandas.core.series.Series

1.4 matching.distance module

matching.distance.**Absolute**

alias of *matching.distance.L1Norm*

matching.distance.**Euclidean**

alias of *matching.distance.L2Norm*

class matching.distance.**Exact**(*tol: float = 1e-16*)

Bases: matching.distance._Distance

Exact distance metric. Allows exact matches on all features only to be added as edges to the graph. Results are identical to using a *Norm* match with *max_distance=0*, but contains speed optimizations to avoid useless computation

tol

A small number to replace 0-weight edges (ie edges of perfect matches) with. This is so that the sparse matrix representation does not drop matches. See DEFAULT_DISTANCE_TOL for details

Type float, default=`DEFAULT_DISTANCE_TOL`

class matching.distance.**L1Norm**(*min_distance: float = 0*, *max_distance: float = inf*, *tol: float = 1e-16*)

Bases: *matching.distance.Norm*

L1 *Norm* distance, aka Manhattan distance. Equal to the sum of the absolute differences

p

Which Lp Norm to use. Either an integer value, or infinity

Type Union[int, float], default=1

min_distance

Minimum allowable edge weight. Can be used as a tuning parameter for identifying similar groups

Type float, default=0

max_distance

Maximum allowable edge weight. Can be used as a tuning parameter for identifying similar groups

Type float, default=inf

tol

A small float. Edge weights of 0 (perfect matches) are replaced with this value, so that the sparse matrix representation does not drop perfect matches (normally, 0 = no edge). Does not affect *max_distance*. See *DEFAULT_DISTANCE_TOL* for details

Type float, default=`DEFAULT_DISTANCE_TOL`

p: Union[int, float] = 1

class matching.distance.**L2Norm**(*min_distance: float = 0*, *max_distance: float = inf*, *tol: float = 1e-16*)

Bases: *matching.distance.Norm*

L2 *Norm* distance, aka Euclidean distance. Equal to the square root of the sum of squared differences

p

Which LpNorm to use

Type Union[int, float], default=2

min_distance

Minimum allowable edge weight. Can be used as a tuning parameter for identifying similar groups

Type float, default=0

max_distance

Maximum allowable edge weight. Can be used as a tuning parameter for identifying similar groups

Type float, default=inf

tol

A small float. Edge weights of 0 (perfect matches) are replaced with this value, so that the sparse matrix representation does not drop perfect matches (normally, 0 = no edge). Does not affect *max_distance*. See *DEFAULT_DISTANCE_TOL* for details

Type float, default=`DEFAULT_DISTANCE_TOL`

p: Union[int, float] = 2

matching.distance.**Manhattan**

alias of *matching.distance.L1Norm*

class matching.distance.**Norm**(*p: Union[int, float]*, *min_distance: float = 0*, *max_distance: float = inf*, *tol: float = 1e-16*)

Bases: matching.distance._Distance, matching.distance._Tuneable, matching.distance._Norm

Distance metric for Lp Norms.

p

Which LpNorm to use. Either an integer value, or infinity

Type Union[int, float]

min_distance

Minimum allowable edge weight. Can be used as a tuning parameter for identifying similar groups

Type float, default=0

max_distance

Maximum allowable edge weight. Can be used as a tuning parameter for identifying similar groups

Type float, default=inf

tol

A small float. Edge weights of 0 (perfect matches) are replaced with this value, so that the sparse matrix representation does not drop perfect matches (normally, 0 = no edge). Does not affect *max_distance*. See *DEFAULT_DISTANCE_TOL* for details

Type float, default=`DEFAULT_DISTANCE_TOL`

1.5 matching.graph module

class matching.graph.**BipartiteGraphParameters**(*init_weight: float = 1*, *treatment_bipartite_attr: Hashable = 1*, *control_bipartite_attr: Hashable = 0*, *treatment_color: str = 'red'*, *control_color: str = 'blue'*, *match_group_col: str = 'match_group'*)

Bases: object

Container of various parameters relevant to initializing, plotting, and otherwise working with bipartite graphs using networkx

init_weight

The default weight between nodes in the naive graph, before *set_edges* is called

Type float

treatment_bipartite_attr

What the "bipartite" attribute for will be set to for each treatment node in the graph

Type Hashable

control_bipartite_attr

What the "bipartite" attribute for will be set to for each control node in the graph

Type Hashable

treatment_color

Color to plot treatment nodes as

Type str

control_color

Color to plot control nodes as

Type str

match_group_col

Name of column to add to the graph data, indicating which match the observation belongs to

Type str

control_bipartite_attr: Hashable = 0

control_color: str = 'blue'

init_weight: float = 1

match_group_col: `str = 'match_group'`

treatment_bipartite_attr: `Hashable = 1`

treatment_color: `str = 'red'`

class matching.graph.**MatchingGraph**(*X: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsA bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]], z: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsA bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]], ids: Optional[Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsA bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]] = None*, **, init_naive_graph: bool = False*)

Bases: `object`

MatchingGraph offers a simple framework to - calculate a distance metric between two sides of a bipartite graph, - iteratively filter this graph by enforcing other constraints on nodes and edges, - calculate a "matched" subset of this graph via greedy, grouping, or optimal matching strategies, and - compare the balance between input and matched datasets.

At construction, the user supplies three *ArrayLike`s:* - *`X*, an array of features to match on; - *z*, an array of binary treatment assignments; and - *ids*, an optional array of observation IDs. If not supplied, default IDs are enumerated from 0 upwards.

These inputs are parsed into the attribute *input_data*. The user can then proceed to *set_edges* by supplying a `_Distance` measure, optionally specifying specific columns of *X* to *include* or *exclude* from the distance calculation.

Once the edges are calculated, the user can then use one of the matching strategies provided to "match" their dataset. Alternatively, the user can proceed to iteratively filter the graph further, via the *filter_edges* and *filter_nodes*. - *filter_edges* allows the user to enforce additional distance metrics on the data, while preserving the edge weights from *set_edges*. For example, a user may calculate edge weights on a propensity score, but then require an exact match on certain categorical features, as well as a cutoff for the Mahalanobis distance between the feature vectors. - *filter_nodes* allows the user to filter nodes via degree, or label. For example, the user may wish to only keep nodes of degree at least 3, after an initial pruning on *Mahalanobis* distance.

input_data

The data which is supplied to the constructor (*X*, *z*, *ids*) is parsed into an *input_data* object, with attributes *X*, *z*, and *ids*. You can convert this to a *pd.DataFrame* via *<MatchingGraphInst>.input_data.frame*

Type `MatchingDataset`

graph_data

Just the data for the nodes currently present on the graph. Has all the same properties as *input_data*. Note that this is a derived property that is re-calculated each time it is accessed.

Type `MatchingDataset`

match_data

This is the same as *graph_data*, except it adds an additional column to *X*: "match_group", an integer indicating which matched subgraph the match belongs to

Type MatchingDataset

graph

Bipartite graph, where nodes represent observation IDs and edge weights correspond to the cost of matching

Type nx.Graph

distance

The distance measure used to calculate the edge weights on the graph

Type _Distance

_params

Private attribute that provides a single container for default arguments to pass to graph functions

Type *BipartiteGraphParameters*

__init__(*X: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *z: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *ids: Optional[Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsArray[numpy.dtype]], bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]] = None*, ***, *init_naive_graph: bool = False*) → None

Initialize a *MatchingGraph* object with necessary data

Parameters

- **X** (*ArrayLike*) – of features to match on. Will be converted to a *pd.DataFrame*
- **z** (*ArrayLike*) – of boolean treatment assignments. Will be converted to a *pd.Series*
- **ids** (*ArrayLike, default=None*) – of observation IDs. Will be converted to a *pd.Series*, and coerced to type *str*
- **init_naive_graph** (*bool, default=False*) – Keyword argument only. *False* by default; whether or not to initialize a "naive", fully-connected graph on init. This can be time-consuming. NOTE: if this is *False*, the user MUST call *set_edges* before doing any filtering! Otherwise the graph will remain empty

distance: matching.distance._Distance

draw(*with_labels: bool = False*, **args*, ***kwargs*) → None

Draw the graph. You can adjust the colors using *self._params.treatment_color* and *self._params.control_color*

Parameters

- **with_labels** (*bool, default=False*) – Whether to draw the *ids* on the graph
- ***args** – Passed to *networkx.draw_networkx*
- ****kwargs** – Passed to *networkx.draw_networkx*

draw_bipartite(*with_labels: bool = False*, **args*, ***kwargs*) → None

Draw the graph using a bipartite layout, with the two groups clearly separated. You can adjust the colors using *self._params.treatment_color* and *self._params.control_color*

Parameters

- **with_labels** (*bool, default=False*) – Whether to draw the *ids* on the graph
- ***args** – Passed to *networkx.draw_networkx*
- ****kwargs** – Passed to *networkx.draw_networkx*

filter_edges(*distance: matching.distance._Distance*, ***, *include: Optional[Iterable[Hashable]] = None*, *exclude: Optional[Iterable[Hashable]] = None*, *drop_isolated: bool = True*) → matching.graph._MatchingGraph

Filter edges of the bipartite graph determined by *distance*. This only has any effect if supplied *distance* measure has non-zero *min_distance* or finite *max_distance*.

By default, this __keeps__ edges which also exist in the filter graph.

Parameters

- **distance** (*_Distance*) – Distance measure to use
- **include** (*Optional[Iterable[Hashable]]*) – Features to __include__ in the distance calculation, *None* by default. When *None*, all features from *self.input_data.X* are included. When *Hashable* or *Iterable[Hashable]*, distance only uses corresponding columns of *self.input_data.X*
- **exclude** (*Optional[Iterable[Hashable]]*) – Features to __exclude__ in the distance calculation, *None* by default. When *None*, no features from *self.input_data.X* are excluded. When *Hashable* or *Iterable[Hashable]*, distance drops corresponding columns of *self.input_data.X*
- **drop_isolated** (*bool*) – *True* by default. Whether or not to remove any nodes with 0 connected edges, following filtering

Returns same instance. This allows method chaining. The filtered graph is available in *self.graph*

Return type _MatchingGraph

filter_nodes(***, *min_degree: int = 0*, *max_degree: Union[int, float] = inf*, *keep_nodes: Optional[Iterable[str]] = None*, *drop_nodes: Optional[Iterable[str]] = None*) → matching.graph._MatchingGraph

Filter nodes of the graph

Parameters

- **min_degree** (*int*) – 0 by default. Only keep nodes of degree (# of connected edges) at least this quantity
- **max_degree** (*Union[int, float]*) – *np.inf* by default. Only keep nodes of degree (# of connected edges) at most this quantity
- **keep_nodes** (*Optional[Iterable[str]]*) – If supplied, a set of node labels to keep; all other nodes are dropped. *None* by default
- **drop_nodes** (*Optional[Iterable[str]]*) – If supplied, a set of node labels to drop; all other nodes are kept. *None* by default

Returns same instance. This allows method chaining. The filtered graph is available in *self.graph*

Return type _MatchingGraph

filter_subgraphs(*, *min_order: int = 0*, *max_order: Union[int, float] = inf*, *min_size: int = 0*, *max_size: Union[int, float] = inf*, *min_treatment: int = 0*, *max_treatment: Union[int, float] = inf*, *min_control: int = 0*, *max_control: Union[int, float] = inf*, *max_control_to_treatment_ratio: float = inf*, *max_treatment_to_control_ratio: float = inf*)

Filter the subgraphs of *self.graph* via a variety of metrics.

This approach is most common when using Coarsened Exact Matching (CEM) or other Exact Matching constraints. In this case, the maximum allowable edge weight is 0 – a perfect match – and so "matching" in a 1:k fashion is arbitrary: you are just giving up data, when match qualities are all identical.

This method is also useful in non-exact matching scenarios, where the max_distance has been tuned to be sufficiently small such that the subgraphs are indeed identifying highly similar observations on their own, and there is no desire for a 1:k match

Parameters

- **min_order** (*int, default=0*) – Minimum number of nodes in the subgraph for the matches to be included. Default is 2
- **max_order** (*int, default=inf*) – Maximum number of nodes in the subgraph for the matches to be included. Default is *np.inf*
- **min_size** (*int, default=0*) – Minimum number of edges in the subgraph for the matches to be included. Default is 1
- **max_size** (*int, default=inf*) – Maximum number of edges in the subgraph for the matches to be included. Default is *np.inf*
- **min_treatment** (*int, default=0*) – Minimum number of treatment nodes in the subgraph for the matches to be included. Default is 1
- **max_treatment** (*Union[int, float], default=inf*) – Maximum number of treatment nodes in the subgraph for the matches to be included. Default is *np.inf*
- **min_control** (*int, default=0*) – Minimum number of control nodes in the subgraph for the matches to be included. Default is 1
- **max_control** (*Union[int, float], default=inf*) – Maximum number of control nodes in the subgraph for the matches to be included. Default is *np.inf*
- **max_control_to_treatment_ratio** (*float, default=inf*) – Maximum ratio of control: treatment within a subgraph for the matches to be included. Default is *np.inf*
- **max_treatment_to_control_ratio** (*float, default=inf*) – Maximum ratio of treatment: control within a subgraph for the matches to be included. Default is *np.inf*

Returns same instance. This allows method chaining. The fitted filtered is available in *self.graph*

Return type _MatchingGraph

graph: networkx.classes.graph.Graph

property graph_data: *matching.dataset.MatchingDataset*

The MatchingDataset for just the data currently present on the graph

input_data: *matching.dataset.MatchingDataset*

match(*, *n_match: int = 1*, *min_match: int = 1*, *replace: bool = False*, *method: Union[str, matching.graph.MatchingMethod] = 'greedy'*)

Conduct matching based on the current nodes and edges in *self.graph*.

This method is a wrapper around other methods of this class, and dispatches to them according to *method*. All parameter descriptions below come with the caveat that they are simply ignored where not applicable. See documentation for other *match_** methods for details on what is used where, and how

Parameters

- **n_match** (*int, default=1*) – Maximum number of matches per treatment group observation. 1 by default
- **min_match** (*int, default=1*) – Minimum number of matches per treatment group observation. Patients with less than this amount of matches are dropped from the result
- **replace** (*bool, default=False*) – Whether or not to conduct matching with replacement. If *True*, then two treatment group observations can match with the same control group observation. *False* by default
- **method** (*Union[str, MatchingMethod], default="greedy"*) – Matching method to use. Valid options are (case-insensitive): - "greedy" == "fast" - "optimal" == "hungarian" == "kuhn" == "munkres"

Returns same instance. This allows method chaining. The matched graph is available in *self.graph*

Return type _MatchingGraph

property match_data: *matching.dataset.MatchingDataset*

The MatchingDataset for just the data currently present on the graph. An additional column to "X" is added: "match_group", indicating which subgraph of *self.subgraphs* the match belongs to This is the data compared against the input data when computing balance metrics.

match_greedy(*, *n_match: int = 1*, *min_match: int = 1*, *replace: bool = False*) → matching.graph._MatchingGraph

Conduct matching based on the current nodes and edges in *self.graph* via a greedy approach. The greedy algorithm works as follows: 1. Sort graph edges by weight, from smallest (closest match) to largest 2. For each edge,

a. if the treatment observation already has *n_match* matches, skip; otherwise,

b. if *replace* = *True* __OR__ the control observation has not yet been matched, add this control as a match to the treatment observation

3. Remove matches with fewer than *min_match* matched control observations

Parameters

- **n_match** (*int*) – Maximum number of matches per treatment group observation. 1 by default
- **min_match** (*int*) – Minimum number of matches per treatment group observation. Patients with less than this amount of matches are dropped from the result
- **replace** (*bool, default=False*) – Whether or not to conduct matching with replacement. If *True*, then two treatment group observations can match with the same control group observation. *False* by default

Returns same instance. This allows method chaining. The matched graph is available in *self.graph*

Return type _MatchingGraph

match_optimal(*, *n_match: int = 1*, *min_match: int = 1*, *replace: bool = False*) → matching.graph._MatchingGraph

Conduct matching based on the current nodes and edges in *self.graph* via the Hungarian algorithm. The Hungarian algorithm solves the _assignment **problem_**, performing a simulatenous bipartite matching which minimizes the total sum of the edge weights.

Parameters

- **n_match** (*int*) – Maximum number of matches per treatment group observation. 1 by default
- **min_match** (*int*) – Minimum number of matches per treatment group observation. Patients with less than this amount of matches are dropped from the result
- **replace** (*bool = False*) – Whether or not to conduct matching with replacement. If *True*, then two treatment group observations can match with the same control group observation. *False* by default

Returns same instance. This allows method chaining. The matched graph is available in *self.graph*

Return type _MatchingGraph

Notes

This algorithm runs in cubic time, with respect to the number of edges on the graph. As such, users should choose a suitable *max_distance* for the distance measure, or make use of *filter_edges* and/or *filter_nodes* to limit the size (# of edges) of the graph, before running.

networkx and *scipy* are a bottleneck to performance – their implementations of this minimum weight bipartite matching algorithm are each written in pure python.

property node_colors: List[str]

Passed as *node_color* to *networkx* drawing functions

set_edges(*distance: matching.distance._Distance*, *, *include: Optional[Iterable[Hashable]] = None*, *exclude: Optional[Iterable[Hashable]] = None*, *allow_new_node: bool = True*, *allow_new_edge: bool = True*) → matching.graph._MatchingGraph

Set the edges of the bipartite graph to have weights determined by *distance*

Parameters

- **distance** (_Distance) – Distance measure to use
- **include** (*Optional[Iterable[Hashable]]*) – Features to __include__ in the distance calculation, *None* by default. When *None*, all features from *self.input_data.X* are included. When *Hashable* or *Iterable[Hashable]*, distance only uses corresponding columns of *self.input_data.X*
- **exclude** (*Optional[Iterable[Hashable]]*) – Features to __exclude__ in the distance calculation, *None* by default. When *None*, no features from *self.input_data.X* are excluded. When *Hashable* or *Iterable[Hashable]*, distance drops corresponding columns of *self.input_data.X*
- **allow_new_node** (*bool*) – Whether to allow new nodes to be added to the graph, *True* by default.
- **allow_new_edge** (*bool*) – Whether to allow new edges to be added to the graph, *True* by default.

Returns same instance. This allows method chaining. The fitted graph is available in *self.graph*

Return type _MatchingGraph

property subgraphs: List[networkx.classes.graph.Graph]

Get the connected component subgraphs of *self.graph* in a list. NOTE: No copy is made here, so you will need to make a copy of each subgraph if you don't want changes to be reflected in the graph

class matching.graph.**MatchingMethod**(*value*)

Bases: enum.Enum

Enum for valid matching methods, with some aliases

FAST = 1

GREEDY = 1

HUNGARIAN = 2

KUHN = 2

MUNKRES = 2

OPTIMAL = 2

exception matching.graph.**NotBipartiteError**

Bases: Exception

Raised if the graph is not bipartite

1.6 matching.graph_utils module

matching.graph_utils.**get_connected_subgraphs**(*G: networkx.classes.graph.Graph*, *, *copy: bool = False*, *deep: bool = False*, ***kwargs*) → List[networkx.classes.graph.Graph]

Split the contents of *G* into a list of connected subgraphs.

Parameters

- **copy** (*bool, default=False*) – Whether or not to make a copy of *G* before computing the subgraphs
- **deep** (*Optional[bool], default=False*) – Whether or not to make a _deep_ copy of *G*. Only used if *copy == True*
- ****kwargs** – Passed to *nx.Graph.copy()* if *copy=True* and *deep=False*

Returns Subgraphs of *G*, in a list. Each subgraph is fully connected

Return type List[nx.Graph]

1.7 matching.preprocessing module

exception matching.preprocessing.**UnsupportedModel**

Bases: Exception

Raised when an unsupported model is provided

matching.preprocessing.**autocoarsen**(*X: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsA bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *n_bins: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._SupportsA bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]] = 5*) → numpy.ndarray

Automatically coarsen all columns of *X* to a certain number of bins

Parameters

- **X** (*ArrayLike*) – input array to coarsen
- **n_bins** (*ArrayLike*) – either a single *int*, or an array of `int`s, specifying number of bins for each column

Returns of coarsened *X*

Return type ndarray

matching.preprocessing.**autocoarsen_cv**(*X: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._Suppor bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *min_k: int = 1*, *max_k: int = 10*) → numpy.ndarray

Automatically coarsen all columns of *X* to a certain number of bins

matching.preprocessing.**prognostic_score**(*X: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._Supp bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *y: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._Supp bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *z: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._Supp bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *model: Optional[matching._typing.sklearn_types._Fittable] = None*, *use_logit: bool = False*) → numpy.ndarray

Compute prognostic scores for the data. Prognostic scores are fit by predicting outcomes (y) from X, using only the CONTROL dataset. This fitted model is then used to get predicted scores over the WHOLE dataset.

Parameters

- **X** (*ArrayLike*) – of features to predict y with
- **y** (*ArrayLike*) – of outcomes
- **z** (*ArrayLike*) – of binary treatment assignments
- **model** (*Optional[_Fittable]*) – Model to use in prediction. Can be a regressor (for continuous response) or a classifier (for discrete response) By default, Logistic Regression with no penalty is used
- **use_logit** (*bool = False*) – Whether or not to take the logit of the predicted values. Only pertinent when *model* is a classifier (and predicts probabilities)

Returns of prognostic scores, aka predicted outcomes, aka y-hat

Return type ndarray

matching.preprocessing.**propensity_score**(*X: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._Supp bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *z: Union[numpy.typing._array_like._SupportsArray[numpy.dtype], numpy.typing._nested_sequence._NestedSequence[numpy.typing._array_like._Supp bool, int, float, complex, str, bytes, numpy.typing._nested_sequence._NestedSequence[Union[bool, int, float, complex, str, bytes]]]*, *model: Optional[matching._typing.sklearn_types._Fittable] = None*, *use_logit: bool = False*) → numpy.ndarray

Compute propensity scores for the data. Propensity scores are predicted treatment assignments (z) from X. This fitted model is then used to get predicted scores over the WHOLE dataset.

Parameters

- **X** (*ArrayLike*) – of features to predict y with
- **z** (*ArrayLike*) – of binary treatment assignments
- **model** (Optional[_Fittable]) – Model to use in prediction. Can be a regressor (for continuous z) or a classifier (for discrete z) By default, Logistic Regression with no penalty is used
- **use_logit** (*bool, default=False*) – Whether or not to take the logit of the predicted values. Only pertinent when *model* is a classifier (and predicts probabilities)

Returns of propensity scores, aka predicted treatments, aka z-hat

Return type ndarray

1.8 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

N

O

P

S

T

U

V

X

Z