pygobnilp Documentation

James Cussens

Sep 27, 2020

Contents:

CHAPTER 1

The Gobnilp class

class pygobnilp.gobnilp.**Gobnilp**

Subclass of the Gurobi Model class specific to learning Bayesian networks. See documentation for the Gurobi Model class for all methods not documented here.

exception StageError

Raised when a method is called at the wrong stage of learning.

exception UserConstraintError

Raised when there is a problem with a user-defined constraint.

__init__()

Initialise a Gobnilp object

absolute_generation_difference

Dictionary of absolute generation difference variables (if these variables have been created, by default they are not)

Assuming the appropriate constraints have been added, if it exists `absolute_generation_difference[v1,v2]` is the absolute value of `generation_difference[v1,v2]`.

Raises *Gobnilp.StageError* – If no absolute generation difference variables have been created

Type dict

add_basic_constraints()

Adds the most useful constraints

Adds the constraints added by the following methods:

- *add_constraints_oneparentset*
- *add_constraints_setpacking*
- *add_constraints_arrow_family*
- *add_constraints_arrow_adjacency*

- *add_constraints_clusters*

add_basic_variables()
Adds the most useful Gurobi MIP variables

Adds the variables added by the following methods:

- *add_variables_family*
- *add_variables_arrow*
- *add_variables_adjacency*

Arrow and adjacency variables are given a higher branching priority than family variables to encourage a balanced search tree.

add_constraints_4b()
Adds "4B" constraints.

All possibly useful 4B constraints are created but these are stored as lazy constraints with a lazy setting of 3 which means that a 4B constraint is pulled into the MIP model only when they cut off the current linear relaxation solution (or integer solution).

See Bayesian Network Structure Learning with Integer Programming: Polytopes, Facets and Complexity (Cussens et al, JAIR) for details

add_constraints_absgendiff()
Adds constraints linking generation difference variables to absolute generation difference variables

add_constraints_arrow_adjacency()
Add constraints that there is an adjacency between v1 and v2 in the undirected skeleton if there is either an arrow from v1 to v2, or an arrow from v2 to v1 in the DAG

add_constraints_arrow_family()
Adds constraints linking arrow variables to family variables

If `pa` is not a parent of `ch` in any family then the corresponding arrow variable `ch<-pa` is simply removed.

add_constraints_arrow_total_order()
Adds constraints linking arrow variables to total order variables

add_constraints_bests()
Add the constraint that at least one BN variable has its best scoring parent set selected

add_constraints_choose_best_for_order()
Adds the constraint that the highest scoring parent set should be chosen whenever whenever the total order variables indicate that doing so would not cause a cycle

add_constraints_chordal()
Adds simple constraints to rule out non-chordal DAGs i.e. those without v-structures (aka immoralities)

Constraints are roughly of this sort: a<-{b,c} + b<-{a,c} + c<-{a,b} <= a-b

add_constraints_clusters(*cluster_cuts=True*, *matroid_cuts=False*, *matroid_constraints=False*)
Adds cluster constraints

For any cluster of BN variable, the cluster constraint states that at least one element in the cluster has no parents in that cluster.

These constraints are always added lazily, since there are exponentially many of them.

Parameters

- **cluster_cuts** (*bool*) – If True then cluster constraints are added as cuts (i.e. as soon as the linear relaxation is solved). If False, we wait until a candidate integer solution is found.
- **matroid_cuts** (*bool*) – If True then cuts corresponding to rank 2 matroids are also added.
- **matroid_constraints** (*bool*) – If True then constraints corresponding to rank 2 matroids are added when an integer solution corresponding to a cyclic digraph is generated.

add_constraints_cycles()
Adds cycle constraints (on arrow variables)

Since there are exponentially many possible cycles, these constraints are added lazily (via a callback).

add_constraints_gen_arrow_indicator()
Adds constraints stating that an arrow from parent to child means that the child's generation number is strictly greater than the parent's.

add_constraints_gen_index_link()
Adds the constraint linking generation to generation index variables

add_constraints_gendiff()
Adds constraints linking generation difference variables to generation variables.

add_constraints_genindex()
Adds the constraint that each variable has exactly one generation index and that each of these indices is distinct

add_constraints_kbranching(*k=0*)
Adds a constraint so that the learned BN is a k-branching.

A DAG is a branching if each child has at most one parent. A DAG is a k-branching if there is a set of at most k edges the removal of which results in a branching

Parameters **k** (*int*) – The value of k

add_constraints_one_dag_per_MEC(*dynamic=True*, *careful=False*)
Adds a constraint that only one DAG per Markov equivalence class is feasible.

The constraint is effected by adding appropriate lazy constraints when Gurobi generates a DAG solution.

Parameters

- **dynamic** (*bool*) – If True then which DAG is feasible for each Markov equivalence class is arbitrary (it will be the first one Gurobi comes across). If false then the representative DAG is fixed (and is determined by the method *dec2mag*).
- **careful** (*bool*) – If True then all the lazy constraints are stored (not just posted) to ensure that new solutions satisfy them. (The value for *careful* is ignored if *dynamic* is False.)

add_constraints_oneparentset()
Adds the constraint that each child has exactly one parent set.

add_constraints_polytree()
Adds the constraint that the DAG should be a polytree

Constraints (and cuts) ruling out cycles in the undirected skeletong are always added lazily, since there are exponentially many of them.

Cluster constraints are removed if this constraint added since ruling out cycles in the undirected skeleton prevents any in the DAG.

add_constraints_setpacking()
Adds constraints like a<-b,c + b<-a,c + c<-a,b <= 1. That is an example for a "triple". Also adds similar constraints for 4-tuples.

add_constraints_sumgen()
Adds the constraint that sum of generation numbers is n*(n-1)/2

add_constraints_total_order(*lazy=0*)
Adds constraints so that total order variables represent a total order

Parameters **lazy** (*int*) – Controls the 'laziness' of these constraints by settng the Lazy attribute of the constraints. See the Gurobi documentation

add_forbidden_adjacency(*uv*)
Add a constraint that a pair of vertices must not be adjacent.

Parameters **uv** (*iter*) – Pair of nodes

Raises *Gobnilp.UserConstraintError* – If the adjacency is also obligatory.

add_forbidden_ancestor(*u*, *v*)
Add constraint that there can be no directed path from *u* to *v*

Parameters

- **u** (*str*) – Start of path
- **v** (*str*) – End of path

Raises *Gobnilp.UserConstraintError* – If the vertices are the same or the directed path is also obligatory.

add_forbidden_arrow(*u*, *v*)
Add a constraint that there can be no arrow between two vertices

Parameters

- **u** (*str*) – Head of forbidden arrow
- **v** (*str*) – Tail of forbidden arrow

Raises *Gobnilp.UserConstraintError* – If the vertices are the same or the arrow is also obligatory.

add_obligatory_adjacency(*uv*)
Add a constraint that a pair of vertices must be adjacent.

Parameters **uv** (*iter*) – Pair of nodes

Raises *Gobnilp.UserConstraintError* – If the adjacency is also forbidden.

add_obligatory_ancestor(*u*, *v*)
Add a constraint that there must be a directed path between specified vertices.

Parameters

- **u** (*str*) – Start of path
- **v** (*str*) – End of path

Raises *Gobnilp.UserConstraintError* – If the vertices are the same or the directed path is also forbidden or would create a cycle.

add_obligatory_arrow(*u*, *v*)
Add constraint that there must be an arrow from *u* to *v*

Parameters

- **u** (*str*) – Head of arrow
- **v** (*str*) – Tail of arrow

Raises *Gobnilp.UserConstraintError* – If the vertices are the same or the arrow is also forbidden or would create a cycle.

add_obligatory_conditional_independence(*a*, *b*, *s*)

Add a constraint that each BN variable in *a* must be independent of each BN variable in *b* conditional on *s*.

Parameters

- **a** (*iter*) – A set of BN variables
- **b** (*iter*) – A set of BN variables
- **s** (*iter*) – A set, possibly empty, of BN variables

Raises *Gobnilp.UserConstraintError* – If *a* or *b* is empty or the 3 sets are not disjoint or the desired conditional independence is not possible (given other constraints).

add_obligatory_independence(*a*, *b*)

Add a constraint that each BN variable in *a* must be independent of each BN variable in *b*.

Parameters

- **a** (*iter*) – A set of BN variables
- **b** (*iter*) – A set of BN variables

Raises *Gobnilp.UserConstraintError* – If the 3 sets are not disjoint or the desired conditional independence is not possible (given other constraints).

add_variables_absgendiff(*branch_priority=0*)

Adds variables representing the absolute difference in generation number between each pair of distinct BN variables.

These variables are constrained to have a **lower bound of 1**. So as long as constraints are posted connecting these variables to the *generation_difference* variables and thus ultimately to the *generation* variables, then each BN variable will have a different generation number.

Calling *add_constraints_absgendiff* ensure that these variables indeed are equal to the absolute values of generation difference variables.

Generation variables are added with *add_variables_gen*. Generation difference variables are added with *add_variables_gendiff*. See the documentation for these two methods for details of how to add appropriate constraints.

All these variables are given objective value 0. (This can be overridden using the `Obj` attribute of the variable.)

Parameters **branch_priority** (*int*) – The Gurobi branching priority for the absolute generation difference variables.

add_variables_adjacency(*branch_priority=0*)

Adds binary Gurobi MIP adjacency variables to the model

The adjacency variable corresponding to `{v1,v2}` is set to 1 iff there is an arrow from `v1` to `v2` or an arrow from `v2` to `v1`.

To connect these variables appropriately to arrow variables it is necessary to call *add_constraints_arrow_adjacency*.

All these variables are given objective value 0. (This can be overridden using the `Obj` attribute of the variable.)

Parameters **branch_priority** (*int*) – The Gurobi branching priority for the adjacency variables.

add_variables_arrow(*branch_priority=0*)
Adds binary Gurobi MIP arrow variables to the model

The arrow variable corresponding to `(pa,ch)` is set to 1 iff there is an arrow from `pa` to `ch` in a learned BN.

To connect these variables appropriately to family variables it is necessary to call *add_constraints_arrow_family*.

All these variables are given objective value 0. (This can be overridden using the `Obj` attribute of the variable.)

Parameters **branch_priority** (*int*) – The Gurobi branching priority for the arrow variables.

add_variables_family(*branch_priority=0*, *best_first_branch_priority=False*)
Adds binary Gurobi MIP family variables to the model

This method should only be called after data (or local scores) have been read in using, for example, a method such as `input_discrete_data`

Parameters

- **branch_priority** (*int*) – The Gurobi branching priority for the family variables. This value is ignored if *best_first_branch_priority* is True.
- **best_first_branch_priority** (*bool*) – If True then the branching priority for the family variables for any given child are (totally) ordered according to local score, with the higher scoring families given higher branching priority than lower ones.

add_variables_gen(*branch_priority=0*)
Adds generation variables to the model

A generation number for a variable in a DAG is an integer such that any variable has a generation number stricty greater than any of its parents.

To connect these variables appropriately to arrow variables it is necessary to call *add_constraints_gen_arrow_indicator*.

To set the sum of all generation numbers to n*(n-1)/2 use *add_constraints_sumgen*.

All these variables are given objective value 0. (This can be overridden using the `Obj` attribute of the variable.)

Parameters **branch_priority** (*int*) – The Gurobi branching priority for the generation variables.

add_variables_gendiff(*branch_priority=0*)
Adds variables representing the difference in generation number between distinct BN variables

Generation and generation difference variables are connected appropriately with *add_constraints_gendiff*.

All these variables are given objective value 0. (This can be overridden using the `Obj` attribute of the variable.)

Parameters **branch_priority** (*int*) – The Gurobi branching priority for the generation difference variables.

add_variables_genindex(*branch_priority=0*, *earlyfirst=True*)
Adds binary variables indicating whether a BN variable has a particular generation number

Parameters

- **branch_priority** (*int*) – The Gurobi branching priority for the generation index variables. (Ignored if *earlyfirst==True*.)
- **earlyfirst** (*bool*) – Generation index variable for low generation numbers have higher branching priority than those for high generation numbers.

add_variables_kbranching(*branch_priority=0*, *ub=None*)
Adds a variable which is the number of arcs that must be deleted for the learned DAG to be a *branching*. In a branching each node has at most one parent

Parameters

- **branch_priority** (*int*) – The Gurobi branching priority for the generation difference variables.
- **ub** (*int*) – An upper bound for this variable

add_variables_kbranching_ch(*branch_priority=0*)
Adds variables for recording max(0,|parents|-1) for each child

Parameters **branch_priority** (*int*) – The Gurobi branching priority for the generation difference variables.

add_variables_total_order(*branch_priority=0*)
Adds binary Gurobi MIP total order variables to the model

The total order variable corresponding to `(v1,v2)` is set to 1 iff `v1 > v2` in the total order associated with a learned BN. Parents always come before children in the total order.

To connect these variables appropriately to arrow variables it is necessary to call *add_constraints_arrow_total_order*.

All these variables are given objective value 0. (This can be overridden using the `Obj` attribute of the variable.)

Parameters **branch_priority** (*int*) – The Gurobi branching priority for the total order variables.

adjacency
Dictionary of adjacency variables (if these variables have been created, by default they are)

Assuming the appropriate constraints have been added, `adjacency[{v1,v2}]` is the adjacency variable indicating that `v1` and `v2` are adjacent.

Raises *Gobnilp.StageError* – If no adjacency variables have been created

Type dict

allowed_user_constypes = ('forbidden_arrows', 'forbidden_adjacencies', 'obligatory_arr
Constraint types available to users. Used as key values when providing constraints via a dictionary. For each constraint type there is also a method for adding constraints whose name has add_ as a prefix and is in the singular.

See also:

input_user_conss_from_dict *add_forbidden_arrow* *add_forbidden_adjacency* *add_obligatory_arrow* *add_obligatory_adjacency* *add_obligatory_ancestor* *add_forbidden_ancestor* *add_obligatory_conditional_independence*

Type tuple

arrow
Dictionary of arrow variables (if these variables have been created, by default they are)

Assuming the appropriate constraints have been added, `arrow[pa,ch]` is the arrow variable indicating that there is an arrow from `pa` to `ch`.

Raises *Gobnilp.StageError* – If no arrow variables have been created

Type dict

before(*stage1*, *stage2*)
Is stage1 strictly after stage2?

Parameters

- **stage1** (*str*) – A Gobnilp learning stage
- **stage2** (*str*) – A Gobnilp learning stage

Raises `KeyError` – If any of the arguments are not the names of Gobnilp learning stages

Returns Whether *stage1* is strictly before *stage2*

Return type bool

between(*stage1*, *stage2*, *stage3*)
Is stage2 strictly after stage1 but not strictly after stage 3?

Parameters

- **stage1** (*str*) – A Gobnilp learning stage
- **stage2** (*str*) – A Gobnilp learning stage
- **stage3** (*str*) – A Gobnilp learning stage

Raises `KeyError` – If any of the arguments are not the names of Gobnilp learning stages

Returns Whether *stage2* is strictly after *stage1* but not strictly after *stage3*

Return type bool

bn_variables
The BN variables (in order)

Type list

bn_variables_index
Maps each BN variable to its index in the sorted list of BN variables

Type dict

child
`child[i]` is the child in the family with index `i`.

See also:

get_family_index, *parents* and *family_list*.

Type list

clear_basic_model()
Removes variables and constraints added by *make_basic_model*.

dag2mec(*dag*)
Finds the Markov equivalence class for a DAG

Parameters **dag** (*iterable*) – A DAG represented by a collection of (indices for) 'families', where each family is a BN variable together with its parents.

Returns A set of sets which is a sparse representation of the characteristic imset which represents the Markov equivalence class. If (and only if) a set has value 1 in the characteristic imset then it is included as an element in the returned set.

Return type frozenset

See also:

Note that the BN variable, its parents and the binary MIP variable for a family can be recovered from its index using the methods *child*, *parents* and `family_variable`, respectively.

A rough 'inverse' of this method is *mec2dag*.

data
Data associated with the instance

Type pandas.DataFrame

data_arities
Arities of the variables in the data

The order of the arities matches the order of the variables in the data, see: *data_variables* not the order in *bn_variables* (which is always in sorted order).

Raises AttributeError if continuous data being used

Type numpy.array

data_variables
the variables in the data

Variables are in the order supplied by the original data source not the order in *bn_variables* (which is always in sorted order).

Type list

family
Dictionary of family variables (if these variables have been created, by default they are)

Assuming the appropriate constraints have been added, `family[child][parent_set]` is the family variable indicating that `parent_set` is the parent set for `child`.

Raises *Gobnilp.StageError* – If no family variables have been created

Type dict

family_list
`family_list[i]` is the family variable for the family with index `i`. (if these variables have been created, by default they are)

See also:

get_family_index, *child*, *parents*

Raises *Gobnilp.StageError* – If no family variables have been created

Type list

family_scores
Dictionary of family scores (aka local scores)

`family_scores[child][parentset]` is the local score for `child` having `parentset` (a frozenset) as its parents.

Raises *Gobnilp.StageError* – If no local scores have been created

Type dict

forbidden_adjacencies
If *{u,v}* is included then an adjacency between *u* to *v* is forbidden.

See also:

add_forbidden_adjacency

Type set

forbidden_ancestors
If *(u,v)* is included then an directed path from *u* to *v* is forbidden.

See also:

`add_forbidden_ancestors`

Type set

forbidden_arrows
If *(u,v)* is included then an arrow from *u* to *v* is forbidden.

See also:

add_forbidden_arrow

Type set

generation
Dictionary of generation variables (if these variables have been created by calling *add_variables_gen*)

Assuming appropriate constraints have been added, by, for example, calling the method *add_constraints_gen_arrow_indicator*, then `generation[v1]` is the generation number for `v1`.

See Section 3.1.2 of Maximum likelihood pedigree reconstruction using integer programming (Cussens, 2010) for details of how generation variables can be used to rule out cycles in directed graphs.

Raises *Gobnilp.StageError* – If no generation variables have been created

See also:

- *add_variables_gen*
- *add_constraints_gen_arrow_indicator*

Type dict

generation_difference
Dictionary of generation difference variables (if these variables have been created by calling *add_variables_gendiff*)

Assuming the appropriate constraints have been added, by, for example, calling the method *add_constraints_gendiff*, then `generation_difference[v1,v2] = generation[v1] - generation[v2]`

Raises *Gobnilp.StageError* – If no generation difference variables have been created

See also:

- *add_variables_gendiff*
- *add_constraints_gendiff*

Type dict

generation_index
Dictionary of generation index variables (if these variables have been created, by default they are not)

Assuming the appropriate constraints have been added, if it exists `generation_index[v1,pos]` indicates whether `v1` has generation number `pos`

Raises *Gobnilp.StageError* – If no generation index variables have been created

Type dict

get_family_index
Maps a family to its index

`get_family_index[child][parents]` is the index for the given family.

See also:

child and *parents* and *family_list*.

Type dict

gobnilp_optimize()
Solve the MIP model constructed by Gobnilp

This overrides Gurobi's optimize method by hard coding in calls to a callback function for adding Gobnilp specific lazy constraints (and cuts).

input_local_scores(*local_scores*)
Read local scores from a dictionary.

Once this method has been run, methods for adding MIP variables, such as *add_variables_family* and *add_basic_variables*, can be used.

Parameters **local_scores** (*dict*) – Dictionary containing local scores. `local_scores[child][parentset]` is the score for `child` having parents `parentset` where `parentset` is a frozenset.

input_user_conss(*consfile*)
Read user constraints from a Python module and store them

If *consfile* is None then this method returns silently.

See also:

allowed_use_constypes

Such constraints can be read in prior to computation of local scores, and can make that computation more efficient

Parameters **consfile** (*str/None*) – If not None then a file containing user constraints

input_user_conss_from_dict(*consdict*)

Read user constraints from a dictionary and store them

The keys of the dictionary must be strings each of which should name an allowed constraint type. Each value should be either (1) a sequence of items where each item is a sequence (perhaps of length one) providing argument(s) to give to the corresponding "add_.." method. For example, dkt1 (below) would be an acceptable dictionary (assuming A, B, C, E and F) are BN variables:

```
dkt1={'forbidden_adjacencies':[['AB'],['BC']],
'obligatory_arrows':[['E','F']]}
```

or (2) a function which, when given the Gobnilp object, returns a sequence of the same form as in case 1). For example, dkt2 would be an acceptable dictionary:

```
def no_arrows(gobnilp):
    return ([(v,w) for v in gobnilp.bn_variables for w in gobnilp.bn_
↪variables if w!=v])
dkt2={'forbidden_arrows':no_arrows}
```

See also:

allowed_use_constypes *add_forbidden_arrow* *add_forbidden_adjacency* *add_obligatory_arrow* *add_obligatory_adjacency* *add_obligatory_ancestor* *add_forbidden_ancestor* *add_obligatory_conditional_independence*

Such constraints can be read in prior to computation of local scores, and can make that computation more efficient

Parameters **consdict** (*dict/None*) – Dictionary mapping the names of allowed constraint types to constraints

Raises ValueError – If dictionary contains a key that is not (the name of) an allowed constraint type

learn(*data_source=None*, *varnames=None*, *header=True*, *comments='#'*, *delimiter=None*, *start='no data'*, *end='output written'*, *data_type='discrete'*, *score='BDeu'*, *local_score_fun=None*, *k=1*, *sdresidparam=True*, *standardise=False*, *arities=None*, *palim=3*, *alpha=1.0*, *nu=None*, *alpha_mu=1.0*, *alpha_omega=None*, *starts=()*, *local_scores_source=None*, *nsols=1*, *kbest=False*, *mec=False*, *polytree=False*, *chordal=False*, *consfile=None*, *consdict=None*, *settingsfile=None*, *pruning=True*, *edge_penalty=0.0*, *plot=True*, *abbrev=True*, *output_scores=None*, *output_stem=None*, *output_dag=True*, *output_cpdag=True*, *output_ext=('pdf',)*, *verbose=0*, *gurobi_output=False*, ***params*)

Parameters

- **data_source** (*str/array_like*) – If not None, name of the file containing the discrete data or an array_like object. If None, then it is assumed that data has previously been read in.
- **varnames** (*iterable/None*) – Names for the variables in the data. If *data_source* is a filename then this value is ignored and the variable names are those given in the file. Otherwise if None then the variable names will X1, X2, ...
- **header** (*bool*) – Ignored if *data* is not a filename with continuous data. Whether a header containing variable names is the first non-comment line in the file.

- **comments** (*str*) – Ignored if *data* is not a filename with continuous data. Lines starting with this string are treated as comments.
- **delimiter** (*None/str*) – Ignored if *data* is not a filename with continuous data. String used to separate values. If None then whitespace is used.
- **start** (*str*) – Starting stage for learning. Possible stages are: 'no data', 'data', 'local scores', 'MIP model', 'MIP solution', 'BN(s)' and 'CPDAG(s)'.
- **end** (*str*) – End stage for learning. Possible values are the same as for *start*.
- **data_type** (*str*) – Indicates the type of data. Must be either 'discrete' or 'continuous'
- **score** (*str*) – Name of scoring function used for computing local scores. Must be one of the following: BDeu, BGe, DiscreteLL, DiscreteBIC, DiscreteAIC, GaussianLL, GaussianBIC, GaussianAIC, GaussianL0. This value is ignored if *local_score_fun* is not None.
- **local_score_fun** (*fun/None*) – If not None a local score function such that *local_score_fun(child,parents)* computes *(score,ub)* where *score* is the desired local score for *child* having parentset *parents* and *ub* is either *None* or an upper bound on the local score for *child* with any proper superset of *parents*
- **k** (*float*) – Penalty multiplier for penalised log-likelihood scores (eg BIC, AIC) or tuning parameter ('lambda^2) for l_0 penalised Gaussian scoring (as per van de Geer and Buehlmann)
- **sdresidparam** (*bool*) – For Gaussian BIC and AIC, whether (like bnlearn) to count the standard deviation of the residuals as a parameter when computing the penalty
- **standardise** (*bool*) – Whether to standardise continuous data.
- **arities** (*array_like/None*) – Arities for the discrete variables. If *data_source* is a filename then this value is ignored and the arities are those given in the file. Otherwise if None then the arity for a variable is set to the number of distinct values observed for that variable in the data. Ignored for continuous data.
- **palim** (*int/None*) – If an integer, this should be the maximum size of parent sets.
- **alpha** (*float*) – The equivalent sample size for BDeu local score generation.
- **nu** (*iter/None*) – The mean vector for the Normal part of the normal-Wishart prior for BGe scoring. If None then the sample mean is used.
- **alpha_mu** (*float*) – Imaginary sample size value for the Normal part of the normal-Wishart prior for BGe scoring.
- **alpha_omega** (*float/None*) – Degrees of freedom for the Wishart part of the normal-Wishart prior for BGe scoring. Must be at least the number of variables. If None then set to 2 more than the number of variables.
- **starts** (*iter*) – A sequence of feasible DAGs the highest scoring one of which will be the initial incumbent solution. Each element in the sequence can be either a bnlearn model string or an nx.DiGraph instance. If this value is not empty, a local scoring function must be provided.
- **local_scores_source** (*str/file/dict/None*) – Ignored if None. If not None then local scores are not computed from data. but come from *local_scores_source*. If a string then the name of a file containing local scores. If a file then the file containing local scores. If a dictionary, then `local_scores[child][parentset]` is the score for `child` having parents `parentset` where `parentset` is a frozenset.
- **nsols** (*int*) – Number of BNs to learn

- **kbest** (*bool*) – Whether the *nsols* learned BNs should be a highest scoring set of *nsols* BNs.
- **mec** (*bool*) – Whether only one BN per Markov equivalence class should be feasible.
- **polytree** (*bool*) – Whether the BN must be a polytree.
- **chordal** (*bool*) – Whether the BN represent a chordal undirected graph (i.e. have no immoralities).
- **consfile** (*str/None*) – If not None then a file (Python module) containing user constraints. Each such constraint is stored indefinitely and it is not possible to remove them.
- **consdict** (*dict/None*) – If not None then a dictionary containing user constraints. The dictionary is used as input to *input_user_conss_from_dict*
- **settingsfile** (*str/None*) – If not None then a file (Python module) containing values for the arguments for this method. Any such values override both default values and any values set by the method caller.
- **pruning** (*bool*) – Whether not to include parent sets which cannot be optimal when acyclicity is the only constraint.
- **edge_penalty** (*float*) – The local score for a parent set with *p* parents will be reduced by *p*edge_penalty*.
- **plot** (*bool*) – Whether to plot learned BNs/CPDAGs once they have been learned.
- **abbrev** (*bool*) – When plotting whether to abbreviate variable names to the first 3 characters.
- **output_scores** (*str/file/None*) – If not None, then a file or name of a file to write local scores
- **output_stem** (*str/None*) – If not None, then learned BNs will be written to "output_stem.ext" for each extension defined in *output_ext*. If multiple DAGs have been learned then output files are called "output_stem_0.ext", "output_stem_1.ext" ...
- **output_dag** (*bool*) – Whether to write DAGs to any output files
- **output_cpdag** (*bool*) – Whether to write CPDAGs to any output files
- **output_ext** (*tuple*) – File extensions.
- **verbose** (*int*) – How much information to show when adding variables and constraints (and computing scores)
- **gurobi_output** (*bool*) – Whether to show output generated by Gurobi.
- ****params** – Arbitrary Gurobi model parameter settings. For example if this method is called with TimeLimit=3, then the Gurobi model parameter TimeLimit will be set to 3

Raises `ValueError` – If *start= 'no data'* but no data source or local scores source has been provided

learned_bn
Learned BN (a maximally scoring one if several learned)

Type *BN*

learned_bns
Learned BNs

Type tuple

learned_scores
Learned BNs

Type tuple

local_scores
alternative name for family_scores

make_basic_model(*nsols=1*, *kbest=False*, *mec=False*, *polytree=False*, *chordal=False*)
Adds standard variables and constraints to the model, together with any user constraints

Variables added by *add_basic_variables*. Constraints added by *add_basic_constraints*.

Parameters

- **nsols** (*int*) – Number of BNs to learn
- **kbest** (*bool*) – Whether the *nsols* learned BNs should be a highest scoring set of *nsols* BNs.
- **mec** (*bool*) – Whether only one BN per Markov equivalence class should be feasible.
- **polytree** (*bool*) – Whether the BN must be a polytree
- **chordal** (*bool*) – Whether the BN must contain no immoralities

Raises *Gobnilp.StageError* – If local scores are not yet available.

mec2dag(*c*, *vs=None*)
Returns a DAG representative from a Markov equivalence class.

Parameters

- **c** (*frozenset*) – A Markov equivalence class represented by a characteristic imset which is itself (sparsely) represented by the set of all sets with value 1 in the characteristic imset. This method assumes that *c* is a valid characteristic imset without any checking.
- **vs** (*iterable*) – An ordering for the BN variables which determines the particular DAG representative returned. If `None` the ordering determined by Python `sort` is used.

Returns The DAG represented by a list of family indices, or None if the required families are not represented.

Return type list/None

See also:

A rough 'inverse' of this method is *dag2mec*.

n
The number of BN variables

Type int

obligatory_adjacencies
If *{u,v}* is included then an adjacency between *u* to *v* is obligatory.

See also:

add_obligatory_adjacency

Type set

obligatory_ancestors
If *(u,v)* is included then an directed path from *u* to *v* is obligatory.

See also:

`add_obligatory_ancestors`

Type set

obligatory_arrows
If *(u,v)* is included then an arrow from *u* to *v* is obligatory.

See also:

add_obligatory_arrow

Type set

obligatory_conditional_independences
If *(a,b,s)* is included, where each is a frozenset, then each BN variable in *a* must be independent of each BN variable in *b* conditional on *s*.

See also:

add_obligatory_conditional_independence, *add_obligatory_independence*

Type set

ordered_parentsets
For each child a list of parent sets sorted by local score

Higher scoring parent sets come before lower scoring ones. Each parent set is a frozenset.

Raises *Gobnilp.StageError* – If no local scores have been created

Type dict

parents
`parents[i]` is the parent set in the family with index `i`.

The parent set is a frozenset.

See also:

get_family_index, *child*, *family_list*

Type list

rawdata
Raw data associated with the instance

Returns a two-dimensional array with one row for each datapoint (and thus one colun for each variable).

If the data is discrete the array entries are of dtype uint32 and if the data is continuous the entries are of dtype float64

Type numpy.array

return_local_scores(*local_score_fun*, *palim=3*, *pruning=True*)
Return a dictionary for each child variable where the keys are the child variables and the values map parent sets to local scores.

Not all parent sets are included. If *palim* is not None, then only those parent sets of cardinality at most *palim* can be included.

Also, when *pruning=True*, a parent set is only included if its local score exceeds that of all its proper subsets.

local_score_fun should be a function that computes a local score.

Parameters

- **local_score_fun** (*fun/None*) – If not None a local score function such that *local_score_fun(child,parents)* computes *(score,ub)* where *score* is the desired local score for *child* having parentset *parents* and *ub* is either *None* or an upper bound on the local score for *child* with any proper superset of *parents*
- **palim** (*int/None*) – If not None then the maximal size of a parent set
- **pruning** (*bool*) – Whether not to include parent sets which cannot be optimal.

Returns A dictionary *dkt* such that *dkt[child][parentset]* is the local score for *child* having parent set *parentset* (where *parentset* is a frozenset).

Return type dict

set_bn_variables(*bnvars*)
Set the BN variables to be a subset of existing BN variables

Parameters **bnvars** (*iter*) – A subset of the existing BN variables

Raises ValueError – If *bnvars* is not a subset of the variables in the data

set_stage(*stage*)
Manually set the stage of learning

Parameters **stage** (*str*) – The desired stage of learning

Raises ValueError – If *stage* is not among the list of possible stages

set_starts(*dags*)
Provide a set of 'starting' DAGs

The highest scoring of these DAGs will become the initial incumbent in the search for the best DAG. So the learned DAG will have a score at least as good as the best of these.

This method should be called prior to computing local scores to ensure that the local scores required for each starting DAG are computed even if parent sets in starting DAGs are bigger than the current limit on parent sets. (So limits on parent set size do not affect starting DAGs).

Parameters **dags** (*iter*) – Collection of DAGS. Each individual DAG must be either a bnlearn modelstring or a nx.DiGraph object.

sol2fvs()
Extracts the family variables set to true by some Gurobi solution

The solution corresponding to the current value of Gurobi parameter SolutionNumber is used.

Returns A pair of lists. The first list contains the families as (child,parents) tuples, where parents is a frozenset. The second list contains Gurobi binary MIP variables for the families.

Return type tuple

stage
Stage of solving

Type str

stages = ('no data', 'data', 'local scores', 'MIP model', 'MIP solution', 'BN(s)', 'CP
A tuple of strings giving Gobnilp's stages of learning (in order).

Type tuple

stages_set = frozenset({'local scores', 'CPDAG(s)', 'MIP model', 'no data', 'MIP solut
The set of Gobnilp's stages of learning.

Type frozenset

total_order
Dictionary of total order variables (if these variables have been created, by default they are not)

Assuming the appropriate constraints have been added, if it exists `total_order[v1,v2]` is the total order variable indicating that `v1 > v2`.

Raises *Gobnilp.StageError* – If no total order variables have been created

Type dict

write_local_scores(*f*)
Write local scores to a file

Parameters **f** (*str/file*) – If a string the name of the file to write to (where "-" leads to writing to standard output). Otherwise a file object.

CHAPTER 2

The BN class

class pygobnilp.gobnilp.**BN**(**args*, ***kwargs*)

Subclass of networkx.DiGraph. See documentation for networkx.DiGraph for all methods not documented here.

At present this class only implements the structure of a BN - the DAG.

__str__()

Returns a textual representation of the BN

Returns A textual representation of the BN

Return type str

adjacency_matrix()

The adjacency matrix

Returns The adjacency matrix

Return type numpy.matrix

bnlearn_modelstring()

Return a string representation suitable for bnlearn's "modelstring" function

Returns A string representation of the BN structure (DAG) suitable for bnlearn's "modelstring" function

Return type str

compute_compelled(*compelled=()*)

Determines which directed edges are present in all DAGs Markov equivalent to the given DAG (i.e. which are compelled to have this direction).

Whether an edge has its direction compelled is given by the "compelled" attribute of the edge.

Starting from a initial set of edges whose direction is *compelled* to be that given in the DAG, the following 3 rules from Koller and Friedman (here shown in Prolog) are used:

```
%R1
compelled(Y,Z) :- edge(Y,Z), compelled(X,Y), not edge(X,Z), not edge(Z,X).
```

(continues on next page)

(continued from previous page)

```
%R2
compelled(X,Z) :- edge(X,Z), compelled(X,Y), compelled(Y,Z).
%R3
compelled(X,Z) :- edge(X,Z), compelled(Y1,Z), compelled(Y2,Z),
                  (edge(X,Y1);edge(Y1,X)),  (edge(X,Y2);edge(Y2,X)).
```

This method uses the "semi-naive evaluation" algorithm for computing the relevant least Herbrand model, which is the set of all the compelled edges.

Parameters **compelled** (*iter*) – Edges to set as compelled in addition to those involved in *immoralities*.

connected(*u*, *v*)
Are *u* and *v* connected (in either direction)

Parameters

- **u** (*str*) – A node
- **v** (*str*) – A node

Returns Whether *u* and *v* are connected

Return type bool

cpdag_str()
Returns a textual representation of the CPDAG

Returns A textual representation of the CPDAG

Return type str

directed_arrow_colour = 'red'
Colour to indicate a directed arrow

directed_arrow_text = '->'
Text to indicate a directed arrow

minimal_ancestral_graph(*nodes*)
Find the minimal ancestral graph of self containing *nodes*

Parameters **nodes** (*iter*) – The nodes for the minimal ancestral graph

Returns The minimal ancestral graph

Return type network.DiGraph

plot(*abbrev=True*)
Generate and show a plot of the CPDAG/DAG

A DAG from the Markov equivalence class defined by the CPDAG is shown. Reversible and irreversible arrows are distinguished by colour. By default the colours are black and red, respectively.

Parameters **abbrev** (*int*) – Whether to abbreviate variable names to first 3 characters.

satisfy_ci(*a*, *b*, *s*)
Does the DAG satisfy this conditional independence relation: *a* is independent of *b* conditional on *s*?

Parameters

- **a** (*iter*) – A set of BN variables
- **b** (*iter*) – A set of BN variables
- **s** (*iter*) – A set, possibly empty, of BN variables

Returns A pair where the first element is a bool stating whether the given conditional independence relation is satisfied and the second is the minimal ancestral graph containing a, b and s.

Return type tuple

undirected_arrow_colour = 'black'
Colour to indicate a undirected edge

undirected_arrow_text = '-'
Text to indicate a undirected edge

CHAPTER 3

The MN class

class pygobnilp.gobnilp.**MN**(**args*, ***kwargs*)

Subclass of networkx.Graph. See documentation for networkx.Graph for all methods not documented here.

At present this class only implements the structure of a Markov network - an undirected graph

satisfy_ci(*a*, *b*, *s*)

Does the Markov network satisfy *a* _|_ *b* | *s*? i.e. is there a path from a node in *a* to a node in *b* which avoids nodes in *s*?

This method does not check that *a*, *b* and *s* are disjoint or that *a* and *b* are non-empty.

Parameters

- **a** (*iter*) – A set of nodes in a Markov network
- **b** (*iter*) – A set of nodes in a Markov network
- **s** (*iter*) – A set of nodes in a Markov network

Returns Whether the Markov network satisfies *a* _|_ *b* | *s*

Return type bool

CHAPTER 4

The Data class

class pygobnilp.scoring.**Data**

Complete data (either discrete or continuous)

This is an abstract class

rawdata()

The data without any information about variable names.

Returns The data

Return type numpy.ndarray

variables()

Returns The variable names

Return type list

varidx()

Returns Maps a variable name to its position in the list of variable names.

Return type dict

CHAPTER 5

The DiscreteData class

class pygobnilp.scoring.**DiscreteData**(*data_source*, *varnames=None*, *arities=None*)

Bases: *pygobnilp.scoring.Data*

Complete discrete data

__init__(*data_source*, *varnames=None*, *arities=None*)

Initialises a *DiscreteData* object.

If *data_source* is a filename then it is assumed that:

1. All values are separated by whitespace
2. Empty lines are ignored
3. Comment lines start with a '#'
4. The first line is a header line stating the names of the variables
5. The second line states the arities of the variables
6. All other lines contain the actual data

Parameters

- **data_source** (*str/array_like/Pandas.DataFrame*) – Either a filename containing the data or an array_like object or Pandas data frame containing it.
- **varnames** (*iter*) – Variable names corresponding to columns in the data. Ignored if *data_source* is a filename or Pandas DataFrame (since they will supply the variable names). Otherwise if not supplied (=*None*) then variables names will be: X1, X2, ...
- **arities** (*iter*) – Arities for the variables corresponding to columns in the data. Ignored if *data_source* is a filename or Pandas DataFrame (since they will supply the arities). Otherwise if not supplied (=*None*) the arity for each variable will be set to the number of distinct values observed for that variable in the data.

arities()

Returns The arities of the variables.

Return type numpy.ndarray

arity(*v*)

Parameters **v** (*str*) – A variable name

Returns The arity of *v*

Return type int

data()

The data with all values converted to unsigned integers.

Returns The data

Return type pandas.DataFrame

data_length()

Returns The number of datapoints in the data

Return type int

make_contab_adtree(*variables*)

Compute a marginal contingency table from data or report that the desired contingency table would be too big.

Parameters **variables** (*iter*) – The variables in the marginal contingency table.

Returns

1st element is of type ndarray: If the contingency table would have too many then the array is empty (and the 2nd element of the tuple should be ignored) else an array of counts of length equal to the product of the *arities*. Counts are in lexicographic order of the joint instantiations of the columns (=variables) 2nd element: the 'strides' for each column (=variable)

Return type tuple

CHAPTER 6

The AbsDiscreteLLScore class

class pygobnilp.scoring.**AbsDiscreteLLScore**(*data_source*, *varnames=None*, *arities=None*)

Bases: *pygobnilp.scoring.DiscreteData*

Abstract class for discrete log likelihood scores

entropy(*variables*)

Compute the entropy for the empirical distribution of some variables

Parameters **variables** (*iter*) – Variables

Returns The entropy for the empirical distribution of *variables* and the number of joint instantiations of *variables* if not too big else None

ll_score(*child*, *parents*)

The fitted log-likelihood score for *child* having *parents*

In addition to the score the number of joint instantations of the parents is returned. If this number would cause an overflow *None* is returned instead of the number.

Parameters

- **child** (*str*) – The child variable
- **parents** (*iter*) – The parent variables

Returns

(1) The fitted log-likelihood local score for the given family and

(2) the number of joint instantations of the parents (or None if too big)

Return type tuple

score(*child*, *parents*)

Return LL score minus complexity penalty for *child* having *parents*, and also upper bound on the score for proper supersets.

To compute the penalty the number of joint instantations is multiplied by the arity of the child minus one. This value is then multiplied by log(N)/2 for BIC and 1 for AIC.

Parameters

- **child** (*str*) – The child variable
- **parents** (*iter*) – The parent variables

Raises ValueError – If the number of joint instantations of the parents would cause an overflow when computing the penalty

Returns The local score for the given family and an upper bound on the local score for proper supersets of *parents*

Return type tuple

CHAPTER 7

The DiscreteLL class

class pygobnilp.scoring.**DiscreteLL**(*data*)

Bases: *pygobnilp.scoring.AbsDiscreteLLScore*

__init__(*data*)

Initialises a *DiscreteLL* object.

Parameters **data** (DiscreteData) – data

score(*child*, *parents*)

Return the fitted log-likelihood score for *child* having *parents*, and also upper bound on the score for proper supersets of *parents*.

Parameters

- **child** (*str*) – The child variable
- **parents** (*iter*) – The parent variables

Returns The fitted log-likelihood local score for the given family and an upper bound on the local score for proper supersets of *parents*

Return type tuple

CHAPTER 8

The DiscreteBIC class

class pygobnilp.scoring.**DiscreteBIC**(*data*, *k=1*)

Bases: *pygobnilp.scoring.AbsDiscreteLLScore*

__init__(*data*, *k=1*)

Initialises a *DiscreteBIC* object.

Parameters

- **data** (DiscreteData) – data
- **k** (*float*) – Multiply standard BIC penalty by this amount, so increase for sparser networks

CHAPTER 9

The DiscreteAIC class

class pygobnilp.scoring.**DiscreteAIC**(*data*, *k=1*)

Bases: *pygobnilp.scoring.AbsDiscreteLLScore*

__init__(*data*, *k=1*)

Initialises an *DiscreteAIC* object.

Parameters

- **data** (DiscreteData) – data
- **k** (*float*) – Multiply standard AIC penalty by this amount, so increase for sparser networks

CHAPTER 10

The BDeu class

class pygobnilp.scoring.**BDeu**(*data*, *alpha=1.0*)
Bases: *pygobnilp.scoring.DiscreteData*

Discrete data with attributes and methods for BDeu scoring

__init__(*data*, *alpha=1.0*)
Initialises a *BDeu* object.

Parameters

- **data** (DiscreteData) – data
- **alpha** (*float*) – The *equivalent sample size*

alpha
The *equivalent sample size* used for BDeu scoring

Type float

bdeu_score_component(*variables*, *alpha=None*)
Compute the BDeu score component for a set of variables (from the current dataset).

The BDeu score for a child v having parents Pa is the BDeu score component for Pa subtracted from that for v+Pa

Parameters

- **variables** (*iter*) – The names of the variables
- **alpha** (*float*) – The effective sample size parameter for the BDeu score. If not supplied (=None) then the value of *self.alpha* is used.

Returns The BDeu score component.

Return type float

bdeu_scores(*palim=None*, *pruning=True*, *alpha=None*)
Exhaustively compute all BDeu scores for all child variables and all parent sets up to size *palim*. If *pruning* delete those parent sets which have a subset with a better score. Return a dictionary dkt where dkt[child][parents] = bdeu_score

Parameters

- **palim** (*int/None*) – Limit on parent set size
- **pruning** (*bool*) – Whether to prune
- **alpha** (*float*) – ESS for BDeu score. If not supplied (=None) then the value of *self.alpha* is used.

Returns dkt where dkt[child][parents] = bdeu_score

Return type dict

clear_cache()
Empty the cache of stored BDeu component scores

This should be called, for example, if new scores are being computed with a different alpha value

upper_bound_james(*child*, *parents*, *alpha=None*)
Compute an upper bound on proper supersets of parents

Parameters

- **child** (*str*) – Child variable.
- **parents** (*iter*) – Parent variables
- **alpha** (*float*) – ESS value for BDeu score. If not supplied (=None) then the value of *self.alpha* is used.

Returns An upper bound on the local score for parent sets for *child* which are proper supersets of *parents*

Return type float

CHAPTER 11

The ContinuousData class

class pygobnilp.scoring.**ContinuousData**(*data*, *varnames=None*, *header=True*, *comments='#'*, *delimiter=None*, *standardise=False*)

Bases: *pygobnilp.scoring.Data*

Complete continuous data

__init__(*data*, *varnames=None*, *header=True*, *comments='#'*, *delimiter=None*, *standardise=False*)

Continuous data

Parameters

- **data** (*numpy.ndarray/str*) – The data (either as an array or a filename containing the data)
- **varnames** (*iterable/None*) – The names of the variables. If not given (=None) then if *data* is a file having the variable names as a header then these are used else the variables are named X1, X2, X3, etc
- **header** (*bool*) – Ignored if *data* is not a filename. Whether a header containing variable names is the first non-comment line in the file.
- **comments** (*str*) – Ignored if *data* is not a filename. Lines starting with this string are treated as comments.
- **delimiter** (*None/str*) – Ignored if *data* is not a filename. String used to separate values. If None then whitespace is used.
- **standardise** (*bool*) – Whether to standardise the date to have mean 0 and sd = 1.

data()

The data as a Pandas dataframe.

Returns The data

Return type pandas.DataFrame

CHAPTER 12

The AbsGaussianLLScore class

class pygobnilp.scoring.**AbsGaussianLLScore**(*data*, *varnames=None*, *header=True*, *comments='#'*, *delimiter=None*, *standardise=False*)

Bases: *pygobnilp.scoring.ContinuousData*

Abstract class for Gaussian log-likelihood scoring

gaussianll(*variables*)

Compute the Gaussian log-likelihood of some variables

Parameters **variables** (*iter*) – Variables

Returns The Gaussian log-likelihood of some variables

ll_score(*child*, *parents*)

The Gaussian log-likelihood score for a given family, plus the number of free parameters

Parameters

- **child** (*str*) – The child variable
- **parents** (*iter*) – The parents

Returns

First element of tuple is the Gaussian log-likelihood score for the family for current data Second element is number of free parameters which is number of parents plus 1 (for intercept)

Return type tuple

CHAPTER 13

The GaussianLL class

class pygobnilp.scoring.**GaussianLL**(*data*)

Bases: *pygobnilp.scoring.AbsGaussianLLScore*

__init__(*data*)

Initialises an *GaussianLL* object.

Parameters **data** (ContinuousData) – data

score(*child*, *parents*)

Return the fitted log-likelihood score for *child* having *parents*, and also upper bound on the score for proper supersets of *parents*.

Parameters

- **child** (*str*) – The child variable
- **parents** (*iter*) – The parent variables

Returns The fitted log-likelihood local score for the given family and an upper bound on the local score for proper supersets of *parents*

Return type tuple

CHAPTER 14

The GaussianBIC class

class pygobnilp.scoring.**GaussianBIC**(*data*, *k=1*, *sdresidparam=True*)

Bases: *pygobnilp.scoring.AbsGaussianLLScore*

__init__(*data*, *k=1*, *sdresidparam=True*)

Initialises an *GaussianBIC* object.

Parameters

- **data** (ContinuousData) – data
- **k** (*float*) – Multiply standard BIC penalty by this amount, so increase for sparser networks
- **sdresidparam** (*bool*) – Whether to count the standard deviation of the residuals as a parameter when computing the penalty

score(*child*, *parents*)

Return the fitted Gaussian BIC score for *child* having *parents*, and also upper bound on the score for proper supersets of *parents*.

Parameters

- **child** (*str*) – The child variable
- **parents** (*iter*) – The parent variables

Returns The Gaussian BIC local score for the given family and an upper bound on the local score for proper supersets of *parents*

Return type tuple

CHAPTER 15

The GaussianAIC class

class pygobnilp.scoring.**GaussianAIC**(*data*, *k=1*, *sdresidparam=True*)

Bases: *pygobnilp.scoring.AbsGaussianLLScore*

__init__(*data*, *k=1*, *sdresidparam=True*)

Initialises an *GaussianAIC* object.

Parameters

- **data** (ContinuousData) – data
- **k** (*float*) – Multiply standard AIC penalty by this amount, so increase for sparser networks
- **sdresidparam** (*bool*) – Whether to count the standard deviation of the residuals as a parameter when computing the penalty

score(*child*, *parents*)

Return the fitted Gaussian AIC score for *child* having *parents*, and also upper bound on the score for proper supersets of *parents*.

Parameters

- **child** (*str*) – The child variable
- **parents** (*iter*) – The parent variables

Returns The Gaussian AIC local score for the given family and an upper bound on the local score for proper supersets of *parents*

Return type tuple

CHAPTER 16

The GaussianL0 class

class pygobnilp.scoring.**GaussianL0**(*data*, *k=1*)

Bases: *pygobnilp.scoring.AbsGaussianLLScore*

Implements score discussed in "l_0-Penalized Maximum Likelihood for Sparse Directed Acyclic Graphs" by Sara van de Geer and Peter Buehlmann. Annals of Statistics 41(2):536-567, 2013.

__init__(*data*, *k=1*)

Initialises an *GaussianL0* object.

Parameters

- **data** (ContinuousData) – data
- **k** (*float*) – Tuning parameter for L0 penalty. Called "lambda^2" in van de Geer and Buehlmann

score(*child*, *parents*)

Return the fitted Gaussian AIC score for *child* having *parents*, and also upper bound on the score for proper supersets of *parents*.

Parameters

- **child** (*str*) – The child variable
- **parents** (*iter*) – The parent variables

Returns The Gaussian AIC local score for the given family and an upper bound on the local score for proper supersets of *parents*

Return type tuple

CHAPTER 17

The BGe class

class pygobnilp.scoring.**BGe**(*data*, *nu=None*, *alpha_mu=1.0*, *alpha_omega=None*, *prior_matrix=None*)

Bases: *pygobnilp.scoring.ContinuousData*

Continuous data with attributes and methods for BGe scoring

__init__(*data*, *nu=None*, *alpha_mu=1.0*, *alpha_omega=None*, *prior_matrix=None*)

Create a BGe scoring object

Parameters

- **data** (*ContinuousData*) – The data
- **nu** (*numpy.ndarray/None*) – the mean vector for the normal part of the normal-Wishart prior. If not given (=None), then the sample mean is used.
- **alpha_mu** (*float*) – imaginary sample size for the normal part of the normal-Wishart prior.
- **alpha_omega** (*int/None*) – The degrees of freedom for the Wishart part of the normal-Wishart prior. Must exceed p-1 where p is the number of variables. If not given (=None) then *alpha_omega* is set to p+2.
- **prior_matrix** (*numpy.ndarray/None*) – The prior matrix 'T' for the Wishart part of the normal-Wishart prior. If not given (=None), then this is set to t*I_n where t = alpha_mu*(alpha_omega-n-1)/(alpha_mu+1)

bge_component(*vs*)

Compute the BGe component for given variables

The BGe score for a family child<-parents is the component for child+parents minus the component for parents (+ a constant term which just depends on the number of parents).

Parameters **vs** (*iter*) – Variable names

Returns The BGe component for the given variables

Return type float

bge_score(*child*, *parents*)
The BGe score for a given family, plus upper bound

Parameters

- **child** (*str*) – The child variable
- **parents** (*iter*) – The parents

Returns

First element of tuple isf the BGe score for the family for current data (using current hyperparameters)
Second element is an upper bound.

Return type tuple

CHAPTER 18

Functions in the gobnilp module

Python version of GOBNILP

pygobnilp.gobnilp.**from_bnlearn_modelstring**(*modelstring*)

Return a DAG from a bnlearn modelstring

Parameters **modelstring** (*str*) – A bnlearn modelstring defining a DAG

Returns The DAG as a networkx Digraph

Return type networkx.DiGraph

pygobnilp.gobnilp.**read_local_scores**(*f*, *verbose=False*)

Read local scores from a named file, standard input or a file object, and return a dictionary dkt where dkt[child][parentset] is the local score for the family child<-parentset

The file is assumed to be in "Jaakkola" format.

Parameters **f** (*str/file object*) – The file containing the local scores.

Returns Dictionary containing local scores

Return type dict

pygobnilp.gobnilp.**mhs**(*subsets*, *ground_set=None*)

Return a minimal hitting set for a set of subsets

A hitting set is a set of elements from the ground set that has non empty intersection with each of the given subsets. A minimal hitting set is a hitting set with minimal cardinality. This function uses Gurobi to solve this NP-hard problem.

Parameters

- **subsets** (*iter*) – The collection of subsets for which the minimal hitting set is sought. These could be, for example, a list of lists of strings where the strings are elements of the *ground_set*.
- **ground_set** (*iter*) – The ground set: each subset must be a subset of this ground set. If missing (=None) then the ground set is the union of all the given subsets.

Raises ValueError – If Gurobi cannot solve the minimal hitting set problem.

Returns A minimal hitting set which will be a subset of the ground set, or None if there is no hitting set.

Return type list/None

CHAPTER 19

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

T

U

V

W