Smile.jl Documentation

Release 1.0

Tim Wheeler

February 11, 2015

Contents

The *Smile.jl* package is a wrapper for the C++ SMILE librarydeveloped by the Decision Systems Laboratory at the University of Pittsburgh. Smile allows users to do structure modelling, inference, and learning for Bayesian Networks.

Contents:

CHAPTER 1

Available Types

Smile.jl represents SMILE objects as Julia types with void pointers to their C++ counterparts. Garbage collection is handled automatically through the use of a finalizer set in the type constructor.

The available types are:

All objects are constructed without parameters, with the exception of SysCoordinates, which requires a NodeDefinition.

CHAPTER 2

Installation

The wrapper library for Smile comes pre-built and is installed automatically when Smile is added through the Julia package manager:

```
Pkg.add("Smile")
```

2.1 Advanced

SMILE is a C++ project for which headers and static libraries are freely distributed. It is recommended that users download a copy of the latest version of **SMILE_** for their operating system.

– _SMILE: https://dslpitt.org/genie/

Software for the C++ wrapper is located in /src/wrapper/. If you installed this package through the Julia package manager it will be in ~/.julia/Smile/src/wrapper/.

Extract the downloaded files in the same directory as the wrapper code. Rename the extracted folder to "lib". Copy libsmile.a and libsmilearn.a into the wrapper folder. The filestructure should now be:

libsmile.a	Smile core static library
libsmilearn.a	Smilearn static library
smile_wrapper.cpp	Cpp wrapper source file
smile_wrapper.hpp	Cpp wrapper header file
smile_compile.sh	Compilation bash script
lib/	Smile Cpp compiled source

Run the bash script:

```
$ bash -x ./smile_compile.sh
```

This will produce smile_wrapper.o and libsmilejl.so.1.0. All that is left to do is place it on your library search path and perform the linking.

Move the file and perform linking:

```
$ sudo mv libsmilejl.so.1.0 /usr/lib
$ sudo ln -sf /usr/lib/libsmilejl.so.1.0 /usr/lib/libsmilejl.so.1
$ sudo ln -sf /usr/lib/libsmilejl.so.1.0 /usr/lib/libsmilejl.so
```

Restart the terminal to ensure libsmilejl is found before using Smile.jl

CHAPTER 3

QuickStart Guide

This tutorial will cover some basic steps in creating a Bayesian network. It closely mimics the first smile tutorial.

We start by declaring an instance of a network.

```
net = Network()
```

Now we are going to create a node called Success. The node will represent a random discrete event (CPT = Conditional Probability Table).

```
success = add_node(net, DSL_CPT, "Success")
somenames = IdArray()
add(somenames, "Sucess")
add(somenames, "Failure")
set_number_of_outcomes(definition(get_node(net, success)), somenames)
```

Here we create the node, obtained a node handle, and then proceed to give it two states. We can similary create a second node with three states.

```
forecast = add_node(net, DSL_CPT, "Forecast")
flush(somenames)
add(somenames, "Good")
add(somenames, "Moderate")
add(somenames, "Poor")
set_number_of_outcomes(definition(get_node(net, forecast)), somenames)
```

Notice that the syntax used closely follows what is defined in SMILE.

Next we add an arc from "Success" to "Forecast" to represent the conditional dependencies of the latter on the former:

```
add_arc(net, success, forecast)
```

Note that handles of the nodes are required to do this.

Next we fill in the probability distribution using the Smile.jl equivalent of the DSL_doubleArray.

```
theprobs = DoubleArray()
set_size(theprobs, 2)
set_at(theprobs, 0, 0.2)
set_at(theprobs, 1, 0.8)
set_definition(definition(get_node(net, success)), theprobs)
```

Specifying the 2x3 probability table for the second node is done as follows:

```
thecoordinates = SysCoordinates(definition(get_node(net, forecast)))
set_unchecked_value(thecoordinates, 0.4); next(thecoordinates)
```

```
set_unchecked_value(thecoordinates, 0.4); next(thecoordinates)
set_unchecked_value(thecoordinates, 0.2); next(thecoordinates)
set_unchecked_value(thecoordinates, 0.1); next(thecoordinates)
set_unchecked_value(thecoordinates, 0.3); next(thecoordinates)
set_unchecked_value(thecoordinates, 0.6);
```

Note that there was no checking. This is for speed reasons, and how the SMILE tutorial was written.

We end by writing the network to a file, either as a ".dsl" or ".xdsl".

```
write_file(net, "mynet.xdsl")
```

CHAPTER 4

Naive Bayes

This learning algorithm creates a Naive Bayes graph structure in which a single class variable points to all other variables. The probability tables are filled out using Expectation Maximization.

4.1 Parameters

classVariableId: the variable id corresponding to the class variable

```
LearnParams_NaiveBayes() = new("class")
LearnParams_NaiveBayes(var::String) = new(var)
```

4.2 Examples

```
net = Network()
learn!( net, dset, LearnParams_NaiveBayes())
```

CHAPTER 5

Tree Augmented Naive Bayes

This learning algorithm creates a Tree Augmented Naive Bayes (TAN) graph structure in which a single class variable have no parents and all other variables have the class as a parent and at most one other attribute as a parent.

The probability tables are filled out using Expectation Maximization.

5.1 Parameters

classvar: the variable id corresponding to the class variable, `String`

maxSearchTime: the maximum runtime for the algorithm, milliseconds, `Cint`

seed: the random seed to use, 0 for time-based random seed, `Uint32`

maxcache: the maximum cache size, `Uint64`

```
LearnParams_TreeAugmentedNaiveBayes() = new("class", 0, 0, 2048)
LearnParams_TreeAugmentedNaiveBayes(class) = new(class, 0, 0, 2048)
```

5.2 Examples

```
net = Network()
learn!( net, dset, LearnParams_TreeAugmentedNaiveBayes())
```

5.3 Algorithm

The TAN algorithm is $O(n^2 \log n)$, where *n* is the number of graph vertices:

1. Compute the mutual information between each pair of attributes
2. Build a complete undirected graph in which the vertices are the attributes *n* variables. The edges are weighted according to the pairwise mutual information
3. Build a maximum weighted spanning tree
4. Transform the resulting undirected graph to a directed graph by selecting the class variable as the root node and seeting the direction of all edges outward from it
5. Construct a TAN model by adding an arc from the class variable to all other variables

5.4 Reference

The Decision Systems Laboratory recommends the following reference:

Friedman, N., Geiger, D., & Goldszmidt, M. (1997). Bayesian network classifiers. *Machine learning*, 29(2), 131-163.

CHAPTER 6

Bayesian Search

This learning algorithm uses the Bayesian Search procedure. It is a general-purpose graph structure learning algorithm, meaning it will attempt to search the full space of graphs for the best graph.

The probability tables are filled out using Expectation Maximization.

The algorithm runs a partially directed graph search over Markov equivalence classes instead of directly searching the space of DAGs (which is superexponential in n).

6.1 Parameters

```
type LearnParams_BayesianSearch
    maxparents            :: Int            # limits the maximum number of parents a node can have
    maxsearchtime         :: Int            # maximum runtime of the algorithm [seconds?] (0 = infini
    niterations           :: Int            # number of searches (and indirectly number of random res
    linkprobability       :: Float64        # defines the probabilty of having an arc between two nod
    priorlinkprobability :: Float64         # defines a prior existence of an arc between two nodes
    priorsamplesize       :: Int            # influences the "strength" of prior link probability.
    seed                  :: Int            # random seed (0=none)
    forced_arcs           :: Vector{Tuple}  # a list of (i->j) arcs which are forced to be in the net
    forbidden_arcs        :: Vector{Tuple}  # a list of (i->j) arcs which are forbidden in the networ
    tiers                 :: Vector{Tuple}  # a list of (i->tier) associating nodes with a particular

    LearnParams_BayesianSearch() = new(5, 0, 20, 0.1, 0.001, 50, 0, Array(Tuple,0), Array(Tuple,0), A
end
```

6.2 Examples

```
net = Network()
learn!( net, dset, LearnParams_BayesianSearch())
```

6.3 Algorithm

Bayesian Search is not a specific algorithm, but a class of algorithms. Thus, what exactly is going on under the hood in SMILE is not known. However, the following is true about partially directed graph search and surmised to be true about Bayesian Search.

A Markov Equivalence class is a graph candidate containing both directed and undirected edges. A directed acyclic graph *G* is a member of the Markov equivalence class encoded by a particually directed graph *G'* iff *G* has the same edges as *G'* without regard to direction and has the same v-structures as *G'*. It follows that the space of marov equivalence classes is smaller.

Two graphs are *Markov equivalent* if they encode the same set of conditional independent assumptions. A Markov Equivalence class is thus a set containing all directed acyclic graphs that are Markov equivalent to each other.

In structure search we would like to maximize the posterior probability of the structure given the data, $\arg\max_G P(G|D)$. An equivalent formulation seeks to maximize the *Bayesian Score*.

In general, two structures belonging to the same Markov equivalence class may be given different scores. However, specific priors, such as the *BDeu* prior, can be used to ensure score equivalence.

6.3.1 Greedy Hill Climbing

Searching over the space of graphs typically runs as follows:

1. Start with a random intial graph
2. Search for the next-best graph reachable by applying one of each of the following operations:
    - Adding a new undirected or directed edge
    - Removing an existing edge
    - Reversing an existing directed edge
    - Converting $A - B - C$ to $A \rightarrow B \leftarrow C$
3. Each candidate graph is scored. The Bayesian Score is defined only for directed acyclic graphs, so a member of the Markov equivalence graph must be generated from which the score is computed
4. The graph with the highest score is selected, and the process is continued until a local maxima is reached

6.4 Reference

Kochenderfer, M. (2014). *Decision Making under Uncertainty*, 47-55.

CHAPTER 7

Greedy Thick Thinning

This learning algorithm uses the Greedy Thick Thinning procedure. It is a general-purpose graph structure learning algorithm, meaning it will attempt to search the full space of graphs for the best graph.

The probability tables are filled out using Expectation Maximization.

7.1 Parameters

```
type LearnParams_GreedyThickThinning
    maxparents           :: Int # limits the maximum number of parents a node can have
    priors               :: Int # either K2 or BDeu
    netWeight            :: Float64 # for BDeu prior it defines the weight for the uniform prior
    forced_arcs          :: Vector{Tuple}  # a list of (i->j) arcs which are forced to be in the netw
    forbidden_arcs       :: Vector{Tuple}  # a list of (i->j) arcs which are forbidden in the networ
    tiers                :: Vector{Tuple}  # a list of (i->tier) associating nodes with a particular
    LearnParams_GreedyThickThinning() = new(5, DSL_K2, 1.0, Tuple[], Tuple[], Tuple[])
end
```

7.2 Examples

```
net = Network()
learn!( net, dset, LearnParams_GreedyThickThinning())
```

7.3 Algorithm

The Greedy Thick Thinning algorithm starts with an empty graph and repeatedly adds the next arc which maximizes the Bayesian Score metric until a local maxima is reached. It then removes arcs untils a local maxima is reached.

The algorithm is thus fairly efficient at the expense of being prone to being trapped in local maxima.

Two priors can be used. The *BDeu* prior ensures equal scoring across Markov equivalance classes. The *K2* prior is constant across all variables and is typically used for maximizing $P(G \mid D)$ when searching the space of graphs directly.

7.4 Reference

Hesar, A. and Tabatabaee, H. and Jalali, M. (2012). *Structure Learning of Bayesian Networks Using Heuristic Methods*.

CHAPTER 8

PC

This learning algorithm uses the PC procedure. It is a general-purpose graph structure learning algorithm, meaning it will attempt to search the full space of graphs for the best graph.

The probability tables are filled out using Expectation Maximization.

8.1 Parameters

```
type LearnParams_PC
    maxcache      :: Uint64 # the maximum algorithm cache size
    maxAdjacency  :: Cint   # thought to be the max number of parents
    maxSearchTime :: Cint   # the maximum runtime of the algorithm, ms
    significance  :: Float64 # the significance level used in independence tests
    forced_arcs    :: Vector{Tuple}  # a list of (i->j) arcs which are forced to be in the network
    forbidden_arcs :: Vector{Tuple}  # a list of (i->j) arcs which are forbidden in the network
    tiers          :: Vector{Tuple}  # a list of (i->tier) associating nodes with a particular tier

    LearnParams_PC() = new(2048, 8, 0, 0.05, Tuple[], Tuple[], Tuple[])
end
```

8.2 Examples

```
net = Network()
learn!( net, dset, LearnParams_PC())
```

8.3 Algorithm

- Start with a complete undirected graph on all n variables, with edges between all nodes
- For each pair of variables X and Y, and each set of other variables S, see if X and Y are conditionally indepdendent given S. If so, remove the edge between X and Y
- Find colliders by checking for conditional dependence; orient the edges of colliders
- Try to orient undirected edges by consistency with already-oriented edges; do this recursively until no more edges can be oriented

8.4 Reference

Spirtes, P. and Glymour, C. and Scheines, R. (2001). *Causation, Prediction, and Search*.