Prototype software development

Last update: July 31, 2014 at 1:43pm

Contents

1 Introduction **1**

2 Model representation **2**
2.1 Static Bayesian networks 2
2.2 Dynamic Bayesian networks 3

3 Estimators and Potentials **4**

4 Database **5**
4.1 Static Database 5
4.2 Dynamic Database 5

5 Open issues **6**

1 Introduction

This document aims at explaining the design of the skeleton of the AMIDST software prototype that will include only the basic structures required to easily plug in, in future tasks, new inference and learning algorithms within an streaming data framework. It is a very first draft that must be refined and expanded with all the functionality. Classes in the diagrams include only those attributes and methods that are considered important for this first version of the design.

The prototype should be as generic as possible in the sense that specific features of new algorithms directly fit into the toolbox. For that purpose, a interface-based implementation will be addressed. At the same time, inheritance will be used only when absolutely necessary in order not to overload classes in excess. Apart from this generality it is important to keep an eye on our three use cases when designing the classes as they give us hints about the different scenarios to be considered for the prototype.

The data structures of the prototype will covered the following issues:

- Model representations for static and dynamic hybrid Bayesian networks.
- Probability estimators for the different types of distributions contained in the models.

- Potentials for the model variables.
- Support for processing data streams for static and dynamic models.
- Support f or scalability. Not sure if scalability influences the design at this stage? Is it specific of the algorithms for learning and inference that will be plugged in later?

In what follows, we explain the key aspects of the different classes that have been grouped by conceptual blocks for the ease of presentation, although they have some classes in common where all the diagrams are linked. The diagrams have been built using the UML design tool included in the IntelliJ IDE.

2 Model representation

For the sake of efficiency, we propose to represent separately static and dynamic models.

2.1 Static Bayesian networks

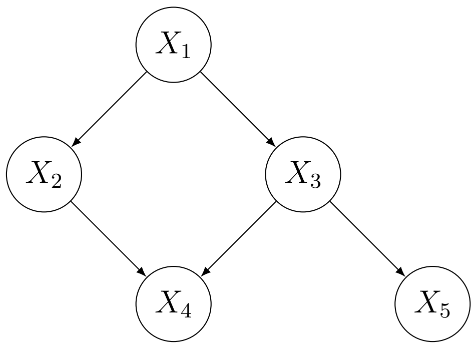


Figure 1: Static Bayesian network

The idea is to have an interface called `BayesianNetwork` for supporting different implementations of a static Bayesian network (Fig. 1). Our implementation of the interface is made through class `BayesNet`. This class contains several attributes:

- **parents**: a vector of *ParentSet* objects in which each element represent a variable in the model. The *ParentSet* object represents the parents as a vector of variable IDs.
- **distributions**: a vector of *Estimator* objects (see Section 3), one for each variable.
- **dataHeader**: a *StaticDataHeader* object (see Section 4.1) with the information about the observed variables.
- **modelHeader**: A *StaticModelHeader* object (see Section 4.2) with information about the variables.

As classifiers are a BN model with their own behavior, we decided to create their own interface called *Classifier* extending from interface *BayesianNetwork*. Thus, different classifier models could implement their own behavior, e.g., NB, TAN, ...

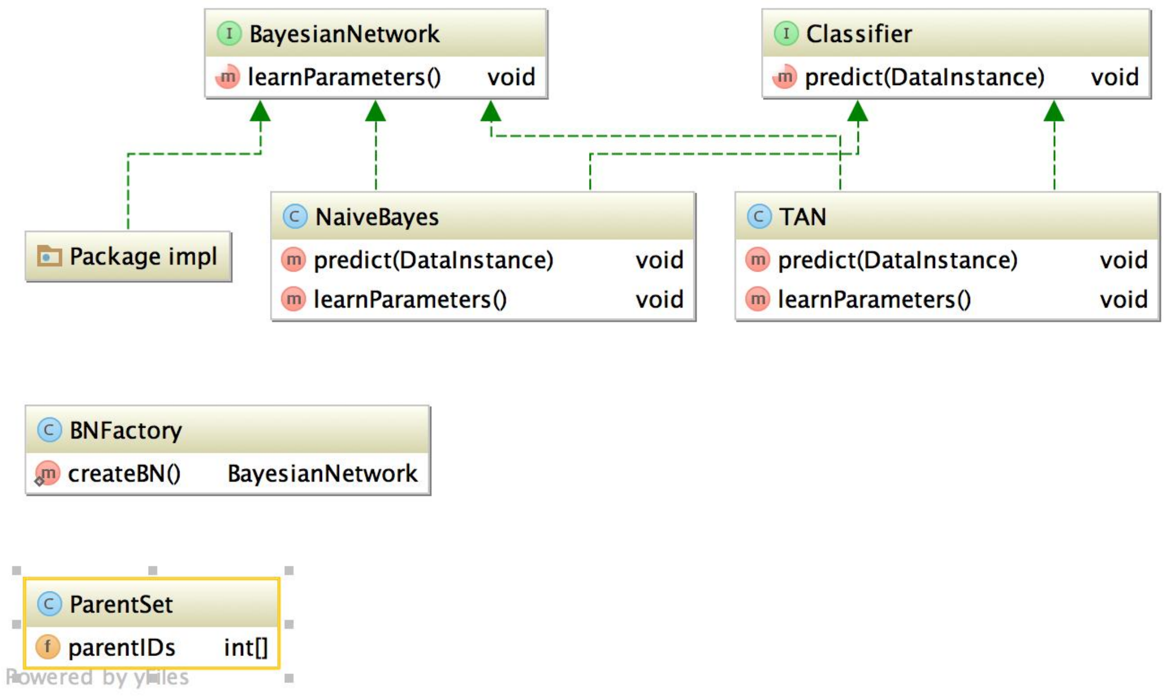
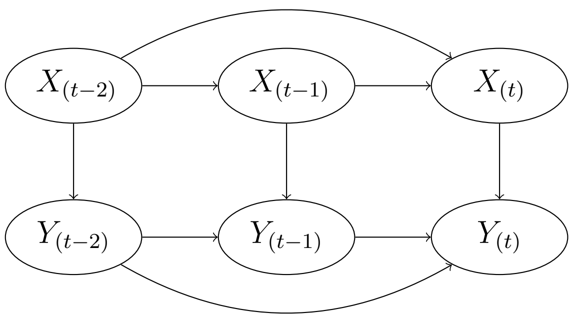


Figure 2: Class diagram for the static BNs.

2.2 Dynamic Bayesian networks

The idea is to represent a dynamic model with any Markov order t. Consider a simple static Bayesian network $X \to Y$. The dynamic version of order $t = 2$ is shown in Fig. 3



$$\begin{tikzcd}
X_{(t-2)} \arrow[r] \arrow[d] \arrow[rr, bend left] & X_{(t-1)} \arrow[r] \arrow[d] & X_{(t)} \arrow[d] \\
Y_{(t-2)} \arrow[r] \arrow[rr, bend right] & Y_{(t-1)} \arrow[r] & Y_{(t)}
\end{tikzcd}$$

Figure 3: Example of a dynamic Bayesian network with Markov order $t = 2$.

Let $X_1, \ldots, X_N$ the set of variables. In order to identity a variable in a time t, we use the following indexes:

- Time t: $[0, 1, \ldots, N-1]$
- Time $t-1$: $[N, 1, \ldots, 2N-1]$
- Time $t-2$: $[2N, 1, \ldots, 3N-1]$

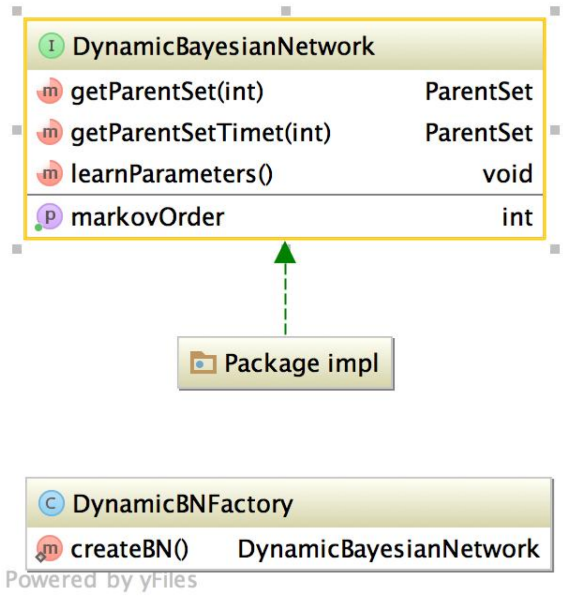


Figure 4: Class diagram for the dynamic BNs.

The class diagram of a dynamic model is shown in Fig. 3. To represent the model structure we are going to use 2 attributes:

- `parentsAtTimet`: a vector of ParentSet of size N. At each position a vector of indexes (see above) are stored.
- `parentsAtTimeBeforeT`: a matrix of ParentSet. The first component is devoted to identify the index of the variable (from 1 to N). The second component identities the time. For the example of order two, the dimension of this second component is 2 ($t-1$ and $t-2$).

The same strategy is used to store the attributes `EstimatorTimet` and `EstimatorTimeBeforet`. Similar to the static approach, we will use an interface called `DynamicBayesianNetwork` and our implementation of a dynamic BN is made through class `DynamicBayesNet`.

3 Estimators and Potentials

We have considered two possible representations of the quantitative component of the variables in the model. The idea is to have, on one hand, *Estimators*, specially designed to be updated from sufficient statistics during learning. On the other hand, it should be convenient to have a lighter data structure, *Potential*, with the learning results and specially oriented only to inference. In this way, we try to optimize both learning and inference.

Regarding *Estimators* we only consider non-exponential distribution families as we are using a sufficient statistic-based learning approach. Fig. 5 shows the data structure considered. The idea is to have an interface `Estimator` with a set of classes implementing it:

`DiscreteEstimator`, `GaussianEstimator` and `MultivariateGaussianEstimator`. For conditional estimator, we define another interface called `ConditionalEstimators` extending from `Estimator`. The implemented conditional estimators cover all the possible structures involving continuous and discrete variables, i.e., `D_D_ConditionalEstimator`, `C_D_ConditionalEstimator`, `C_DC_ConditionalEstimator` and `C_C_ConditionalEstimator`.

Data structure for *Potentials* is shown in Fig. 6. Class `Potential` will have methods for combining, marginalizing potentials, ... In contrast to the *Estimators*, this class is designed so that their probability distributions are not modified.

4 Database

For modeling the data structures related to the dataset, there are several issues and limitations to be considered. Firstly, data (in general) will not be loaded in memory so streaming techniques are considered instead. Secondly, the data processing for learning dynamic models requires a special treatment (at least in Cajamar). For the sake of efficiency, we propose to design separately static and dynamic representations of the database structure for static and dynamic models, respectively. Also, because a same problem rarely implies the use of both representations of the database simultaneously.

Let consider the following notation.

Variables: $X_1, \ldots, X_n$

A dataset $\mathcal{D}$ contains a set of elements in the form $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}\}$, where each $\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_n^{(i)})$ represents a data instance.

4.1 Static Database

The static representation of the database shown in Fig. 7 will be used for learning a static model under data streaming. Class `DataInstance` represents a unique instance $\mathbf{x}^{(i)}$ of the dataset. The idea is to have an *iterator* class responsible for processing the different data instances in the database. This is made by class `DataStream`.

In the same level, class `DataStreamWindow` covers the situation in which going back in the set of data instances is somehow required for updating the estimators. It represents a memory window with the latest n data instances processed so far.

Class `StaticDataHeader` will store general information about the variables. First, a vector indicating which variables are observed and second, the list of variables with their information (`class StaticModelHeader`). There is a class `Variable` with the attributes *name*, *varID* (useful to identify and process a variable efficiently), among others.

4.2 Dynamic Database

The dynamic processing of the dataset leads to a more complex design. We have identified two possible ways of arriving streaming data:

1. Each time t a set of values for the variables arrives. For $t = 2$ this will be a `SequenceData` object:

$\mathbf{x}_{(t-2)}$	$\mathbf{x}_{(t-1)}$	$\mathbf{x}_{(t)}$

2. Each time t a set of values for the variables for a set of individuals arrives. This is the case of Cajamar in which every day we have a data matrix with values for all the variables and all the clients:

Client 1	$\mathbf{x}_{(t-2)}^{(1)}$	$\mathbf{x}_{(t-1)}^{(1)}$	$\mathbf{x}_{(t)}^{(1)}$
...	...	...	...
Client m	$\mathbf{x}_{(t-2)}^{(m)}$	$\mathbf{x}_{(t-1)}^{(m)}$	$\mathbf{x}_{(t)}^{(m)}$

The idea is that every time we have to process a data matrix (`BucketSequenceData`) with m rows (each row corresponds to a `SequenceData` object) and t columns.

5 Open issues

- Consider the situation in which data stream exceeds the data processing of the system. A kind of buffer could be a solution.
- Scalability inside the algorithms. Does it influence the definition of the prototype or only when including the learning/inference algorithms?

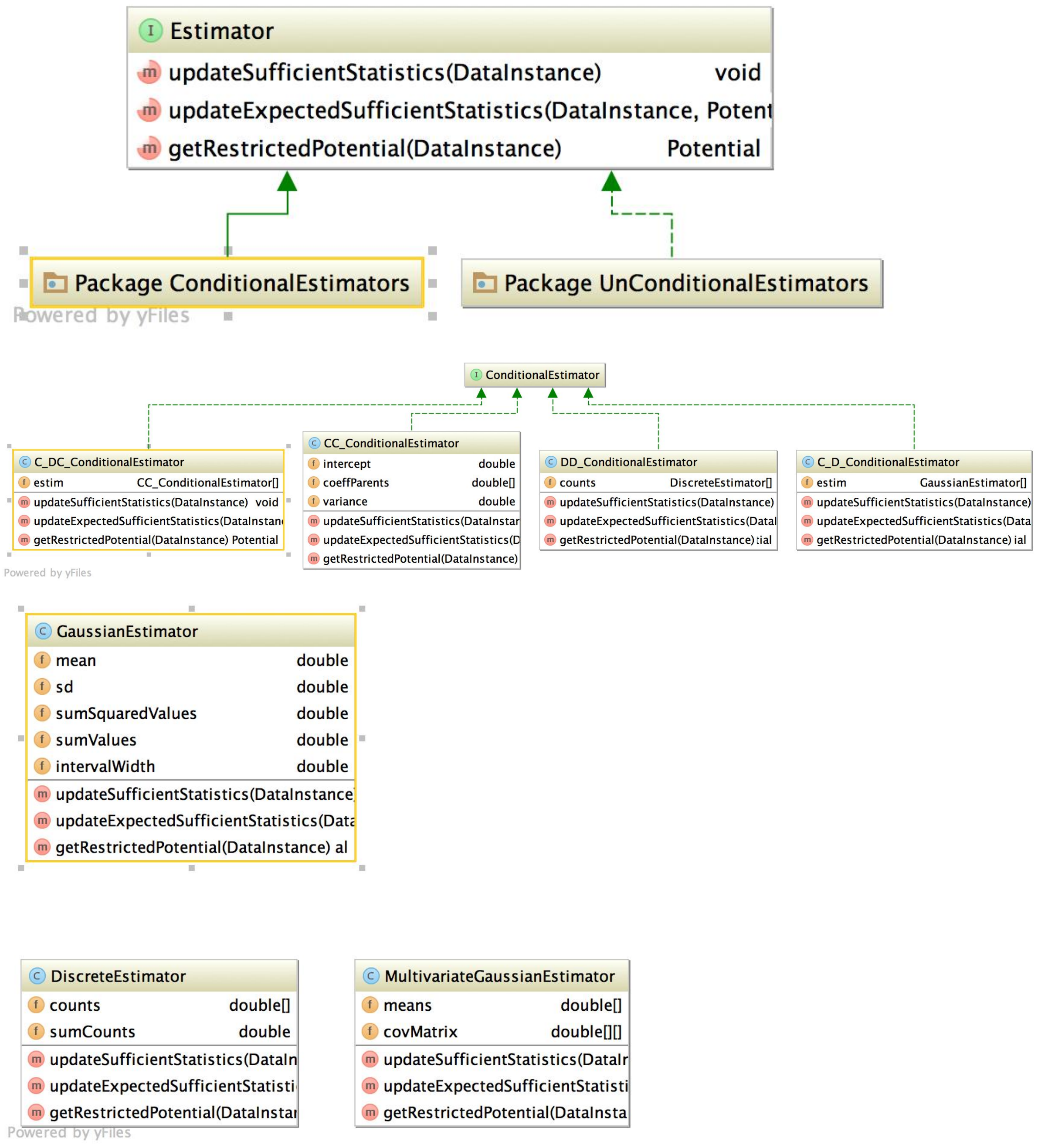

Estimator
updateSufficientStatistics(DataInstance) void
updateExpectedSufficientStatistics(DataInstance, Potent
getRestrictedPotential(DataInstance) Potential
Package ConditionalEstimators
Package UnConditionalEstimators
Powered by yFiles
ConditionalEstimator
C_DC_ConditionalEstimator
estim CC_ConditionalEstimator[]
updateSufficientStatistics(DataInstance) void
updateExpectedSufficientStatistics(DataInstan
getRestrictedPotential(DataInstance) Potential
CC_ConditionalEstimator
intercept double
coeffParents double[]
variance double
updateSufficientStatistics(DataInstar
updateExpectedSufficientStatistics(D
getRestrictedPotential(DataInstance)
DD_ConditionalEstimator
counts DiscreteEstimator[]
updateSufficientStatistics(DataInstance)
updateExpectedSufficientStatistics(Datal
getRestrictedPotential(DataInstance) tial
C_D_ConditionalEstimator
estim GaussianEstimator[]
updateSufficientStatistics(DataInstance)
updateExpectedSufficientStatistics(Data
getRestrictedPotential(DataInstance) ial
Powered by yFiles
GaussianEstimator
mean double
sd double
sumSquaredValues double
sumValues double
intervalWidth double
updateSufficientStatistics(DataInstance
updateExpectedSufficientStatistics(Data
getRestrictedPotential(DataInstance) al
DiscreteEstimator
counts double[]
sumCounts double
updateSufficientStatistics(DataIn
updateExpectedSufficientStatisti
getRestrictedPotential(DataInstar
MultivariateGaussianEstimator
means double[]
covMatrix double[][]
updateSufficientStatistics(DataIr
updateExpectedSufficientStatisti
getRestrictedPotential(DataInsta
Powered by yFiles

Figure 5: Class diagram for the Estimators.

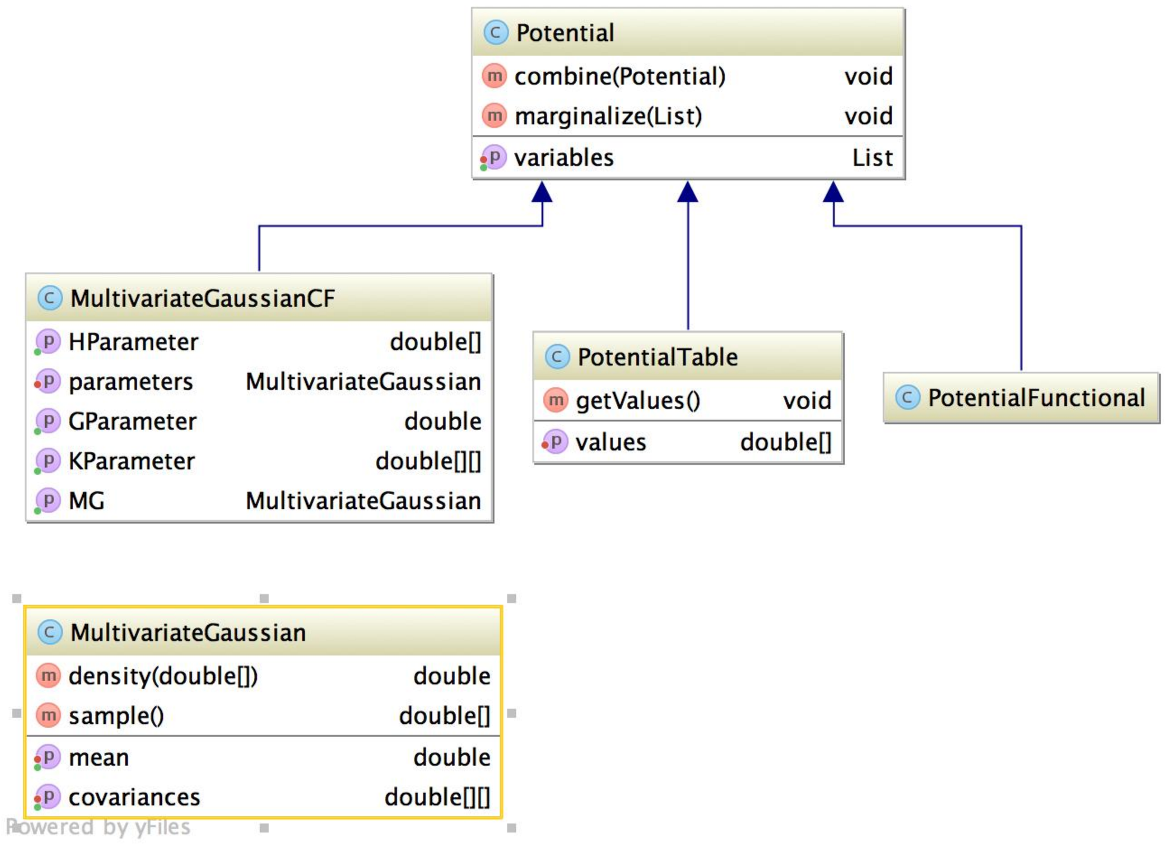


Figure 6: Class diagram for the Potentials.

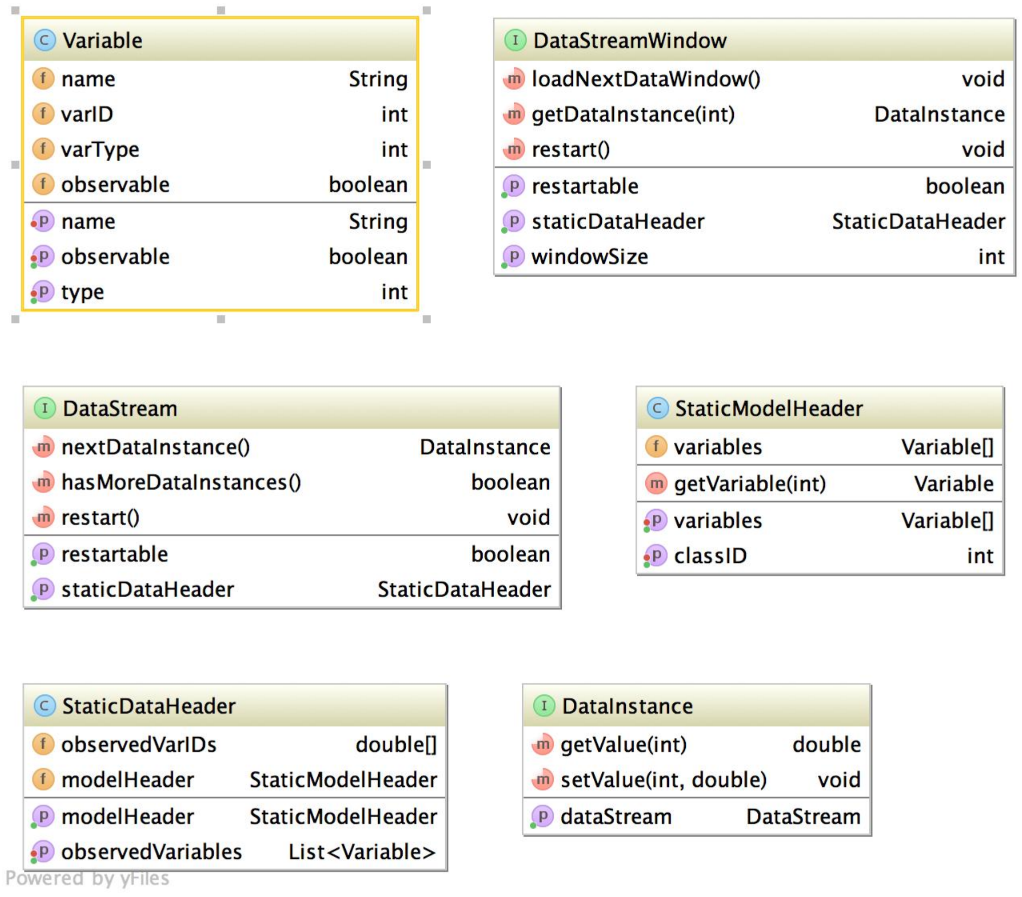


Figure 7: Class diagram for the static database.

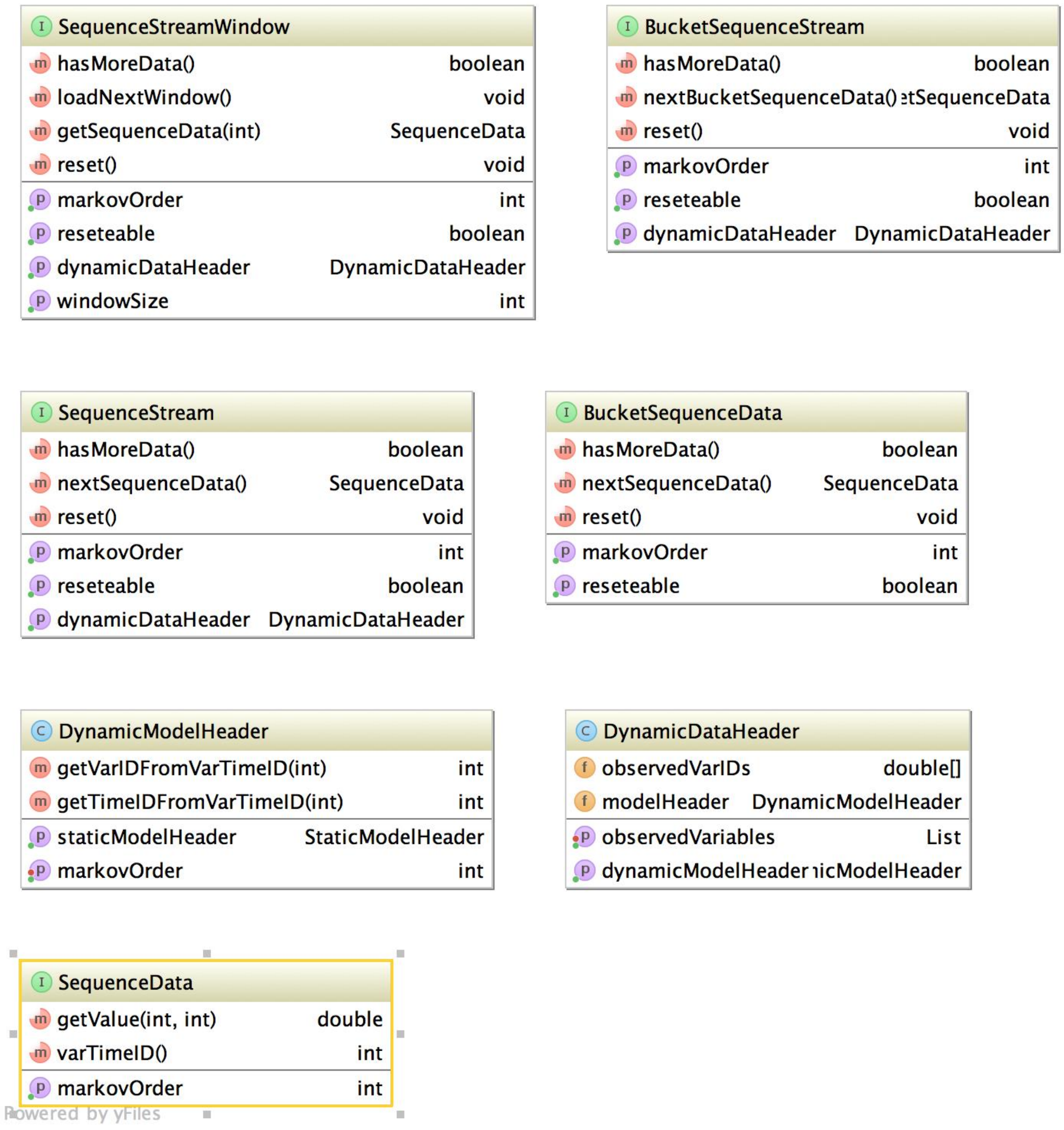


Figure 8: Class diagram for the dynamic database.