

# Towards Model-Driven Engineering for Big Data Analytics – An Exploratory Analysis of Domain-Specific Languages for Machine Learning

Dominic Breuker  
University of Muenster - ERCIS  
[breuker@ercis.de](mailto:breuker@ercis.de)

## Abstract

*Graphical models and general purpose inference algorithms are powerful tools for moving from imperative towards declarative specification of machine learning problems. Although graphical models define the principle information necessary to adapt inference algorithms to specific probabilistic models, entirely model-driven development is not yet possible. However, generating executable code from graphical models could have several advantages. It could reduce the skills necessary to implement probabilistic models and may speed up development processes. Both advantages address pressing industry needs. They come along with increased supply of data scientist labor, the demand of which cannot be fulfilled at the moment. To explore the opportunities of model-driven big data analytics, I review the main modeling languages used in machine learning as well as inference algorithms and corresponding software implementations. Gaps hampering direct code generation from graphical models are identified and closed by proposing an initial conceptualization of a domain-specific modeling language.*

## 1. Motivation

*Data Scientist* is the term for a new type of analysts possessing both the business knowledge to understand the problems companies are faced with as well as the technical skills to generate decision-relevant information from large-scale datasets. In times in which big data is on everyone's lips, the data scientist is a popular job description. Some even call it the sexiest job of the 21<sup>st</sup> century [1].

Business intelligence (BI) has traditionally been concerned with topics such as gathering, storing, and integrating data from various sources to make them available through data warehouses [2]. This data may be used purely for reporting or ad-hoc querying, but also for more advanced analytics [3]. The former is supported by technologies such as online analytical

processing (OLAP) while for the latter, probabilistic analysis techniques can be used with great success.

In the context of big data, much research focusses on technological aspects such as coping with petabyte-scale datasets. Various approaches to accomplish this are discussed in the literature [4], with the MapReduce paradigm being the most well-known [5]. Yet even more important than having the technology to process data is hiring talented people capable to make sense of it. Data alone is worthless without analysis. Demand for data scientists is way ahead of supply [6].

One reason why data scientists are that hard to find can be found by considering the skillset they have to possess. The machine learning and data mining communities have developed a long list of different algorithms for tackling all sorts of analysis problems. Among the most popular examples are k-means for clustering and support vector machines for classification [7]. Sophisticated software tools such as WEKA [8], SAS, or SPSS provide ready-to-use implementations of many algorithms, making them easy to apply. In any application though, it is the data scientist's task to find a mapping between these algorithms and the particular problem at hand. The challenge is that there is not necessarily an exact fit. Hence, a combination of different techniques might be most appropriate. In other cases, custom modifications might be necessary.

Even worse, an analyst needs deep understanding of both the domain and the various technical algorithms to successfully do an analysis — a rare combination [9]. Only domain experts can tell which questions are worth asking and which peculiarities of a problem can be exploited to fine-tune models [6]. Only a technical expert is capable of implementing them.

Hence, applied data analysis by means of standard tools is a much more creative design process than its name might suggest. Consequently, the machine learning community has put considerable efforts into developing a theory of designing customizable algorithms instead of proving a set of black-box procedures. Probabilistic graphical models are one piece of the puzzle. They constitute a versatile tool for representing probabilistic models, either standard ones (e.g., regression)

or entirely customized models. Algorithms adapting flexibly to varying model structures are the other piece. They allow deriving concrete implementations from a single theory [10].

That being said, implementing an algorithm specific to a given graphical model in software is still a demanding and effortful task. Fortunately, there is software shielding users from these complex issues. An example is Infer.NET [11], a library providing facilities for probabilistic model specification via an application programming interface (API). After choosing one of the available algorithms, the library takes over and executes it on the model. This way of implementing data analytics techniques is called a model-based approach to machine learning [12].

While such frameworks greatly reduce the complexity of implementing machine learning approaches, the full potential is not yet leveraged. Modeling is done programmatically via calls to an API. With probabilistic models being represented graphically anyways, it is easily conceivable to do the same thing entirely via visual notations. Not only may this further boost productivity. It may also facilitate collaboration in teams of technical and domain experts.

Software engineers in the field of model-driven engineering (MDE) have long acknowledged the usefulness of visual notations for designing software systems. Numerous frameworks have been developed [13]. Their common theme is that (1) a domain-specific modeling language (DSML) is used to express functionality in terms of the domain under analysis and that (2) a transformation engine generates executable code [14]. Most importantly, these techniques avoid errors made when programming manually and allow specifying functionality declaratively instead of imperatively. Although empirical research is costly and therefore scarce, DSMLs proved to increase productivity in several empirical studies [15], [16].

Consequently, the aim of this paper is to explore the opportunities of defining a DSML for probabilistic modeling. The design goal is to allow for entirely visual specification from which executable code can be generated that fits models to data. What I present in this paper is a first conceptualization of such a language. It synthesizes various modeling constructs presented in machine learning literature and extends them with the information necessary to derive executable code. This information is identified by studying the Infer.NET API. The ultimate goal of this research is to make the process of analyzing big data both easier and more efficient, thereby helping to close the gap between supply and demand of data scientists.

The DSML I propose is not domain specific with respect to an industry or a particular analytical class.

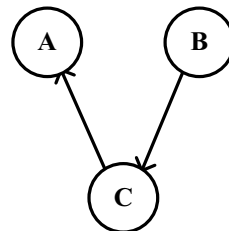
Rather, it is specific to the domain of doing inference in probabilistic models with no or partial observations.

To establish a background on graphical models, I review probabilistic modeling languages, the algorithms they can be used with, and corresponding software packages in Section 2. Section 3 reflects graphical models against the backdrop of MDE. The DSML for probabilistic models together with an outline of how code generation can be accomplished are discussed in Section 4. Finally, Section 5 concludes and outlines future research.

## 2. Graphical Models

### 2.1 The Purpose of Graphical Models

Graphical models are a tool to describe probabilistic models visually. The main ingredients of these models are random variables. A graphical model describes the joint distribution over all the variables it contains [17]. Consider the simple example illustrated in Figure 1. The graph represents a probability distribution  $P(A, B, C)$  over a set of three random variables  $\{A, B, C\}$ . The purpose of graphical models is to support a user in asking questions about random variables, very much in the way a database answers questions about the data it stores [18]. For instance, a user might want to know the distribution over variable A only, i.e., with B and C marginalized out.



$$P(A, B, C) = P(A|C) \cdot P(C|B) \cdot P(B)$$

**Figure 1. Simple directed graphical model with three variables**

To illustrate how graphical models can help answering these questions, consider an example in which all variables A, B, and C are discrete and can assume six different values (1 to 6). For each variable, the probabilities of this happening may depend on all other variables. The question we are seeking an answer to shall be: what is the distribution  $P(A)$ ?

Given the joint distribution  $P(A, B, C)$  this question is easy to answer. Simply marginalize out B and C, i.e., sum over all the possible combinations of values B and C can assume:  $P(A) = \sum_{B, C} P(A, B, C)$ . As B and C

can assume values from 1 to 6, there are  $6^2 = 36$  such combinations. While this solution works for toy problems it soon becomes intractable when models have not three but thousands of variables.

The remedy comes in form of conditional independencies that can be exploited to speed up the computations. Using the product rule of probability<sup>1</sup>, we can rewrite  $\sum_{B,C} P(A, B, C) = \sum_{B,C} P(A|B, C) \cdot P(C|B) \cdot P(B)$  which brings no immediate gain. Inspecting the formula in Figure 1 though we see that  $P(A, B, C)$  factorizes to  $P(A|C) \cdot P(C|B) \cdot P(B)$ , which means that A is conditionally independent of B given C. This is also reflected in the graph visualizing the distribution. There is no arc from B to A. As a consequence, we can change the order of summing and multiplying when computing  $P(A)$ :

$$\begin{aligned} P(A) &= \sum_{B,C} P(A|B, C) \cdot P(C|B) \cdot P(B) \\ &= \sum_{B,C} P(A|C) \cdot P(C|B) \cdot P(B) \\ &= \sum_C P(A|C) \cdot \sum_B P(C|B) \cdot P(B) \end{aligned}$$

The effect of this change is that we no longer compute a single sum with  $6^2$  terms but two individual sums with only  $6 \cdot 2$  terms. Thus, this computation is three times as fast as the original one.

While only discrete variables have been used in the example above, the theory is more general than that [19]. Random variables may equally well have different domains and different distributions (e.g., real values and Gaussians). A wide range of stochastic models can be expressed in this way. Conditional independencies encoded in a graph's structure allow answering questions regarding marginal distributions by doing a series of local computations.

While traditional machine learning techniques estimate parameters of models, the Bayesian paradigm is gaining widespread popularity in recent times. In Bayesian models a parameter is equipped with a distribution (the *prior*), i.e., is treated as a random variable. The goal is not to estimate a particular value for the parameter but to infer a distribution over values after observing training data (the *posterior*). Key selling arguments include making uncertainty in parameter estimates explicit (in form of variance in the posterior) and enabling an analyst to specify domain knowledge not reflected in the data (by modifying the prior) [20].

As a consequence, the only two things needed in the toolbox of a Bayesian are a modeling language to express distributions as graphs and a so called inference algorithm, i.e., an algorithm processing these





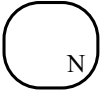
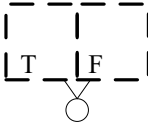
graphs to answer questions regarding conditional marginal distributions (for this reason, graphical models are most popular among Bayesians [21]). The upcoming two subsections are devoted to introduce these two types of tools.

## 2.2 Modeling Languages

Common to all graphical models is that they are used to represent conditional independencies between sets of random variables. Also, they are all graphs, i.e., they consist of vertices and edges. Apart from that, different classes of graphical models exist. Most relevant in this context are directed graphical models, undirected graphical models, and factor graphs [19].

It is worth noting that graphical models are not as standardized as the Unified Modeling Language (UML) [22] for software engineering or the Business Process Model and Notation (BPMN) [23] for business process modeling. Different authors may thus use different modeling constructs. My treatment follows Bishop's introduction to this topic [19], but some extensions from other literature are integrated as well.

**Table 1. Constructs of directed graphical models**

<i>Construct</i>	<i>Symbol</i>	<i>Meaning</i>
Latent variable		Defines a random variable.
Observed variable		Defines a random variable clamped to an observation.
Parameter		Defines a parameter of the model.
Dependency relationship		Indicates that the distribution of one random variable depends on another variable or parameter.
Plate		Defines an area that is repeated N times.
Gate		Depending on a random variable, it selects different parts of the model with respect to the random variable's value.

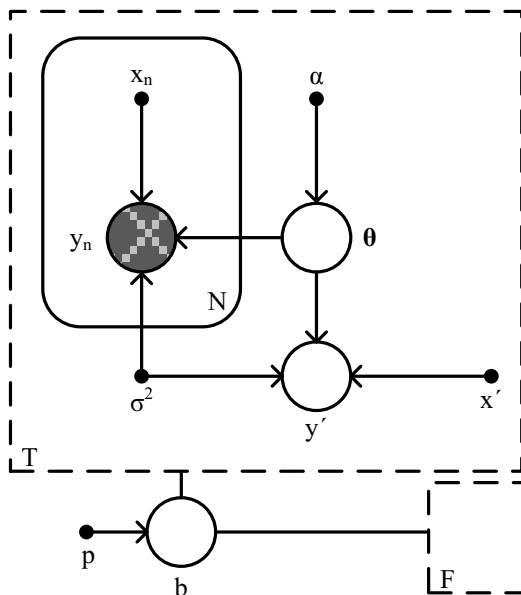
<sup>1</sup> The product rule of probability says that  $P(X, Y) = P(X|Y) \cdot P(Y)$ .

Directed graphical models represent a joint distribution by breaking it down into local pieces. All constructs introduced in the following are presented in Table 1. The main building blocks are random variables, denoted by circles. They are connected by directed arcs which specify the dependency relationships among them. A variable's conditional distribution depends on all its parent variables but no others. The joint distribution emerges when all conditional distributions are multiplied. In the example of Figure 1 variable B has no parent. Hence, the corresponding distribution  $P(B)$  is independent of A and C. Variable A though has a parent and so the corresponding distribution  $P(A|C)$  depends on it.

If a model is to be fitted to data, training data must be part of it. This is done by making a distinction between latent and observed variables. The latter are shaded grey to visualize the difference. It is said that such a variable is clamped to a value.

Another necessity is to express parameters of the model which is done using small black circles. Similar to observed random variables they have constant values. Like variables, they may be sources of arcs as a random variable's distribution may depend on them. However, they do not have a distribution themselves.

As models may consist of huge numbers of random variables, syntactic sugar is useful to express structural analogies. This is achieved with the plate notation. A plate is an area within the model that is repeated a number of times. This way, one can for instance repeat a part of the model for each point in the training data.



**Figure 2. Graphical model for polynomial regression (adapted from [19], Figure 8.7)**

A similar construct developed by Minka and Winn [24] can be used to describe mixture models in a compact way. It is called a gate and can be used to switch on and off different parts of a model depending on a (discrete) random variable. Each value of this random variable is associated with an area in the model. The entire model is a mixture of all areas weighted with the respective mixing probabilities.

Consider the example of Bayesian polynomial regression adapted from Bishop [19]. It illustrates the use of all these constructs (c.f. Figure 2).

A set of training data consisting of  $N$  pairs of real variables  $(x_n, y_n)$  is given. The task is to predict  $y'$  using a new data point  $x'$ . To explain the data, the goal is to fit a polynomial  $f(x, \theta)$  of order  $k$  with coefficient vector  $\theta = (\theta_1, \dots, \theta_k)^T$ . To account for the uncertainty associated with this explanation, a Gaussian noise model defines the likelihood function  $P(\mathbf{y}|\mathbf{x}, \theta, \sigma^2) = \prod_{n=1}^N \mathcal{N}(y_n|f(x_n, \theta), \sigma^2)$ . To keep things simple,  $\sigma^2$  is assumed to be known. Following the Bayesian approach, we also want to account for uncertainty regarding the polynomial  $f$  itself. Hence, variance in its parameters is formalized using a zero-centered, multivariate Gaussian prior  $P(\theta|\alpha) = \mathcal{N}(\theta|0, \alpha^{-1}\mathbf{I})$  with known precision  $\alpha$ . A typical question could now be to infer the predictive distribution  $P(y'|x', \mathbf{x}, \mathbf{y}, \alpha, \sigma^2)$  to reason about likely values of  $y'$  given  $x'$ , the training data, and the model specification.

Another question could be assessing the quality of this model. A Bayesian way of doing so is to compute model evidence, i.e., the likelihood of the model with all random variables marginalized out [25]. The higher the evidence, the better is the model. Hence, this quantity can be used to compare models. To represent this, a gate has been added in Figure 2 to switch between the regression model and an empty model using a Bernoulli random variable  $b$ . Model evidence is computed by inferring the posterior over  $b$  [24].

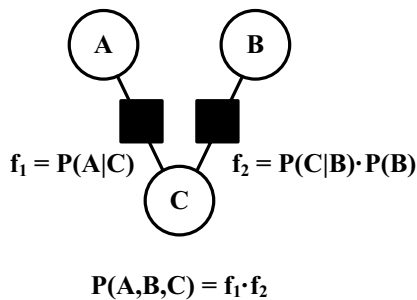
Undirected graphs are another class of graphical models. The most obvious difference to directed graphs is that edges are undirected. A more important difference is that the set of conditional independencies that can be encoded is not the same. As a consequence, some models expressed in one language cannot be converted to the other notation [19]. Apart from undirected arrows, the modeling language is the same.

Factor graphs are a third notation for distributions. As discussed above, graphical models describe how a joint distribution can be decomposed into factors. Factor graphs are an explicit representation of this factorization [19]. Their basic constructs are listed in Table 2. However, they are not restricted to only these three. Apart from directed dependencies, any constructs used in the directed graphical models may be used too.

**Table 2. Basic constructs of factor graphs**

Construct	Symbol	Meaning
Variable	○	Defines a random variable.
Factor	■	Correspond to factors of the joint distribution.
Relationship	—	Indicates which random variables are part of which factor.

Random variables in factor graphs are again represented in form of circles, possibly clamped to observed values. The factors of a joint distribution are associated with *factor* constructs connected via undirected arcs to all variables used in the factors. An example is illustrated in Figure 3. The joint distribution is split up into two factors  $f_1$  and  $f_2$ . As there can be different ways of splitting up a joint distribution, there can be different factor graphs describing the same joint distribution.



**Figure 3. A factor graph corresponding to the directed graphical model of Figure 1.**

### 2.3 Inference Algorithms

The goal of inference algorithms is to compute unknown quantities from those that are known. Given a set of latent random variables  $\mathbf{x}_l$  and a set of observed random variables  $\mathbf{x}_o$ , probabilistic inference can be described as the problem of computing a distribution  $P(\mathbf{x}_l|\mathbf{x}_o)$  over the latent variables given the observed ones [25]. As mentioned above, in a Bayesian approach in which traditional parameters are treated as latent variables, inference can be used to solve a wide range of problems such as prediction for new data.

A number of general purpose algorithms have been developed in the past. *General purpose* refers to the fact that these algorithms are not tailored to a specific probabilistic model. Instead, they can be adapted to

different models, and graphical models provide a formalism facilitating these adaptations.

On a broad level, inference algorithms can be classified into exact and approximate algorithms. Exact are all algorithms that always deliver the correct solution. Notable examples are the junction tree algorithm [26], which works on any kind of graphical model, and the sum-product algorithm [27], which can be applied to tree-structured factor graph. Problematic about these algorithms is that they are often inapplicable to real world problems. On the one hand, the sum-product algorithm severely restricts a graph's structure and can only be used with a limited number of models. On the other hand, the more general junction tree algorithm is computationally intractable for all but the simplest models [19]. Exact algorithms are also restricted to discrete variables or linear Gaussian models [25].

Unfortunately, one cannot expect to find an efficient and exact solution for the general case [28]. Hence, there is a huge field of research dealing with approximate inference. There are two main types of methods: deterministic and stochastic approximations.

A famous representative of the first type is mean field approximation. The idea is to replace the original, intractable distribution with an approximate form with additional independencies added to ensure tractability. Minimizing the difference between the original distribution and the approximate form delivers an approximation that can be used for inference. Variational Message Passing (VMP) [29] is a general purpose algorithm doing this on graphical models. Alternatively, the Expectation propagation (EP) algorithm [30] can be used. The difference compared to VMP is that the minimization is done in another way. As approximations, these algorithms do not necessarily return a correct solution. They may be far off and do not provide an estimate of the approximation's accuracy.

Stochastic approximations do not approximate the functional form of a distribution but perform sampling instead. The idea is that any property of a distribution can be approximated by drawing and averaging over a sufficiently large number of samples. For virtually all models of interest, Markov Chain Monte Carlo (MCMC) methods can be used. They construct a markov chain whose stationary distribution is the one from which samples shall be drawn [25]. A popular special case of MCMC is Gibbs sampling which can be applied to a wide range of graphical models [31].

A comparison of these inference algorithms is found in Table 3. Stochastic inference is better with respect to accuracy. It will converge to solutions of any accuracy eventually while deterministic algorithms may deliver bad results no matter how long they run. Sampling can also be applied to a wider range of problems. It can be cumbersome to apply deterministic

algorithms to more elaborate distributions [32]. They score though when it comes to performance. Sampling may require long running times until convergence is reached. Only for few, very large models, sampling may be more efficient [25]. It is also not trivial to configure a sampler and to diagnose convergence. Deterministic methods can be used more easily.

**Table 3. Comparison of classes of approximate inference algorithms**

<i>Criterion</i>	<i>Inference algorithms</i>	
	<b>Deterministic</b>	<b>Stochastic</b>
Accuracy	-	+
Expressiveness	-	+
Performance	+	-
Usability	+	-

## 2.4 Modeling environments

There are several software packages that implement general purpose inference algorithms for graphical models. Murphy [33] reviewed a number of them. While the published review is relatively old, the accompanying website<sup>2</sup> is updated regularly. At the time of writing it lists 68 packages, each of which is characterized along several dimensions. To identify suitable candidates for a machine learning MDE approach, I filtered the list in the following way.

First, all software packages not having an API were discarded. There are several standalone software tools. While some of them are well-developed, their major disadvantage is that their models cannot be integrated as components into other software environments.

Second, all software packages not fully supporting continuous nodes were discarded. Many of the tools work only with discrete random variables or may support continuous variables by discretization or sampling only. Expressiveness is severely reduced when using only discrete variables. Hence, focusing on the more flexible packages seems reasonable.

Five software packages were left for closer inspection after filtering. They are listed in Table 4 along with the corresponding programming languages as well as the inference algorithms they support. Almost all packages offer support for inference via stochastic approximations. Infer.NET [11] and JAGS [34] support ordinary Gibbs sampling. Stan uses a method called Hamiltonian Monte Carlo Sampling [35].

<sup>2</sup> <http://www.cs.ubc.ca/~murphyk/Software/bnsoft.html>  
(Last updated 12 February 2013)

Blaise<sup>3</sup> uses some other form of MCMC. Regarding Blaise, I was not able to find source code for download. Deterministic approximations are supported only by BayesBlock<sup>4</sup> through a variational algorithm and by Infer.NET, which allows users to choose between VMP and EP.

As discussed in Section 2.3, both deterministic and stochastic inference approaches have their pros and cons. Hence, I chose the Infer.NET library as a starting point to explore the possibilities of model-driven probabilistic data analysis as it supports both approaches.

Infer.NET is a C# library that allows specifying probabilistic models via its API. Internally, it compiles the user's code to more efficient code on which inference algorithms can be run automatically. Models are created by defining variable objects and connecting them with factor objects. Hence, it closely resembles the factor graph thinking. Variables may have prior distributions if used in conjunction with distribution factors. Their distribution may also be defined as a function of other variables. For instance, a variable could be the sum of two others. Infer.Net offers a wide array of distributions and other types of factors.

**Table 4. Software packages for graphical models**

<i>Name</i>	<i>Language</i>	<i>Inference algorithms</i>	
		<b>Deterministic</b>	<b>Stochastic</b>
Bayes-Blocks	Python C++	x	
Blaise	Java		x
Infer.NET	C#	x	x
JAGS	Java		x
Stan	C++ R		x

## 3. A Model-driven Critique

From the perspective of model-driven engineering, graphical models constitute an interesting development in the field of machine learning. The most important aspect is the move away from standard procedures for specific problems that must be adapted and glued together. Instead, these new approaches strictly separate the problem definition from the solution strategy. Declarative modeling languages like A Mathematical Programming Language (AMPL) [36] demonstrate the usefulness of this idea for the case of optimization

<sup>3</sup> <http://publications.csail.mit.edu/abstracts/abstracts07/bonawitz/bonawitz.html>

<sup>4</sup> <http://research.ics.aalto.fi/bayes/software/>

problems. Users can focus on the important things — modeling the domain mathematically — and they can use any method they want to have the problem solved. Reliable standard software with a declarative interface has been a key driver in the success story of linear programming as it enabled widespread industry applications [37]. Graphical models and general purpose inference algorithms may play the same role in today’s big data analytics challenges.

Moreover, graphical models could be interpreted as a DSML for probabilistic modeling. They provide dedicated constructs needed to specify a probabilistic model on a conceptual level. They also come with general purpose algorithms working on these models. Algorithms adapting themselves to the graph structure could be interpreted as transformation engines generating model-specific inference code. Hence, they offer the two constituent elements of MDE technology: a DSML and a transformation engine [14].

Although some literature on graphical models may suggest a graphical model is all one needs to do inference, this is not entirely true. Graphical models are meant as a device conveying the structure of a model to ease the inference algorithm’s derivation. However, they leave many things unspecified. Among them are obvious things such as the types of distributions being used, but also more complicated aspects such as which variables are connected with each other when their relationships cross the borders of plates. Consider the simple example of Figure 1. Whether the variables A, B, and C are discrete or real and which distributions are used to define the model cannot be seen. As a consequence, a direct correspondence of visual models and program code cannot be established.

Consistent with this observation, none of the libraries in table 4 provides facilities for graphically specifying graphical models. Instead, APIs are called to create them. While the process of modeling resembles that of drawing a model (and specifying, on the way, the additional information that is needed), no actual visualization connected to the code is created.

I argue that this way of modeling in code leaves untapped a considerable part of the benefits graphical models may provide. Most importantly, visual languages can further boost productivity. Direct improvements might be realized since specifying models graphically avoids mistakes in the source code. Much more significant though can be secondary effects. Machine learning components integrated into information infrastructures of enterprises will not be developed by a single person but by teams that are subject to employee turnover. New employees could more easily familiarize themselves with visual models than with code. A direct, formal connection between visual mod-

els and code also ensures alignment that can otherwise be lost easily over time as the software is modified.

It is worth noting that modeling Bayesian networks is different than using graphical user interfaces (GUIs) of statistical software packages such as SPSS. The former define distributions (cf. Section 2.1) while the latter are used to chain together the statistical techniques implemented in the software package.

## 4. Model-driven Engineering for Machine Learning

### 4.1 A Graphical Modeling Language

To develop a conceptual model of an engine transforming a graphical model to executable code, the first step is to define a DSML containing all necessary information. Graphical modeling languages are a good starting point but do not contain sufficient information. To approach the problem of designing the DSML, I have analyzed the Infer.NET modeling API against the backdrop of the modeling languages discussed in Section 2.2. The result is an initial proposal for a DSML. A model of its abstract syntax can be found in Figure 4. The rationale behind it is as follows.

The main constructs of Infer.NET models are *variables* which are either *random* or *observed*. Both have associated data types stored in the attribute *type*, and the latter have an attribute called *value* to store the observation. Although it is possible to define constants as well their use is discouraged. The code generated by the library must be recompiled each time a constant value changes. Observed variables can be changed without recompilation.

Variables are connected to each other via factors. They define the distributions over the variables. Hence, Infer.NET models are very similar to factor graphs. A difference is that they can be thought of as directed. Factors can be of different types. They broadly fall into the classes of distribution factors (Gaussian, discrete, ...) and derived factors (sum of two or more variables, inner product of random vectors, ...), which is documented in the attribute *type*. A factor is parameterized by at least one variable, which is captured in the *is parameter for* relationship. If more than one variable is required and they cannot be used interchangeably, it is necessary to specify the variables’ roles (e.g., mean and variance parameters of a Gaussian). All factors define a distribution over exactly one random variable, which is indicated by the *produces* relationship.

Both variables and factors are subsumed in the abstract entity type *Node* which can be given a textual description (the *name* attribute). Nodes can lie in two types of areas. They can be in a *Plate*, in which case

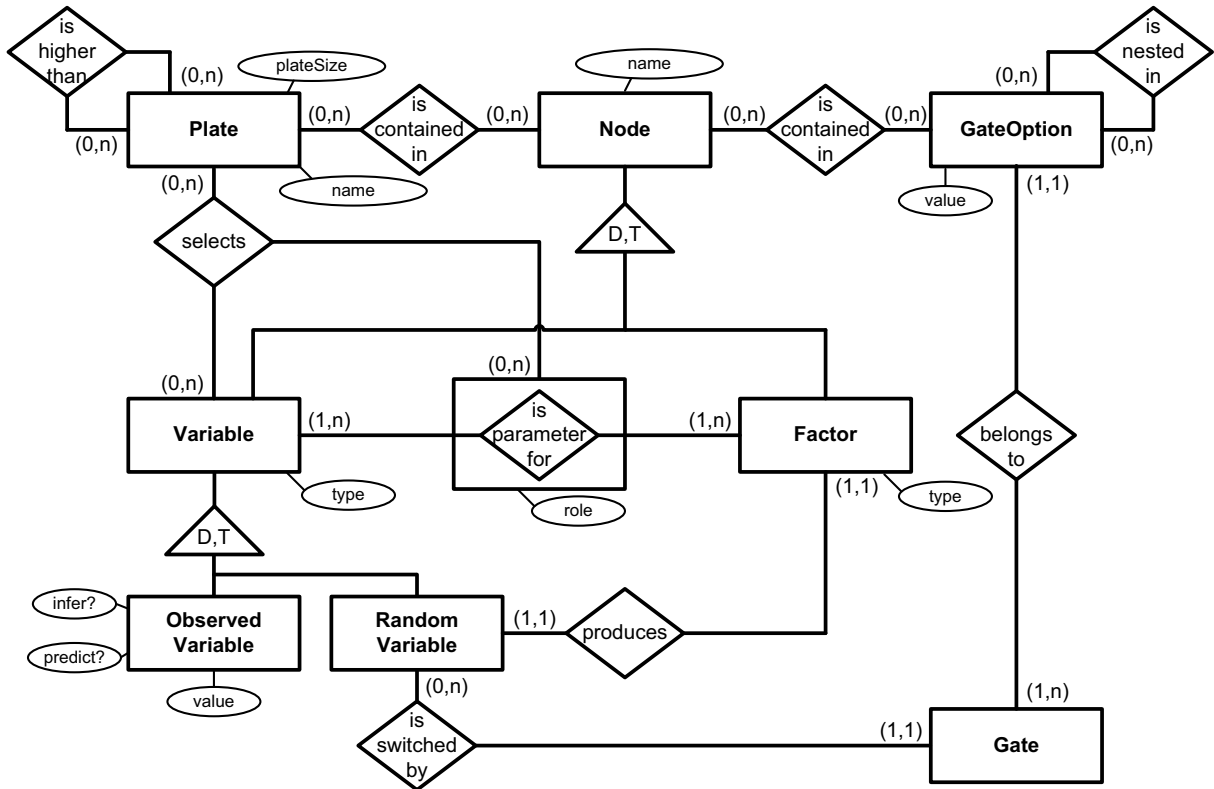


Figure 4. Abstract syntax of the DSML in entity relationship notation

they are not single nodes but arrays of nodes indexed by the plate. The index runs from one to *plateSize*. If a node belongs to multiple plates, it becomes a high-dimensional jagged array. To consistently index these arrays, an arbitrary total order must be defined on the plates. It is codified in the *is higher than* relationship.

It can occur that a variable being a parameter for a factor is associated to a different set of plates than the factor is. In this case, it must be specified how the mismatch should be resolved. There can be two cases. First, the factor can lie in a plate in which the variable is not (more factors than variables). This can be resolved by feeding the variable into all the factors. The other possibility is that the variable lies in a plate in which the factor is not (more variables than factors). Similarly to the first case, all variables can be fed into the factor (if the specific type of factor can handle them). In any case, if only a specific variable or factor should be used and not all, a selector variable must be defined. The ternary relationship *selects* codifies this.

Nodes may also lie within a *GateOption* to allow for describing mixture models. Gate options belong to exactly one *Gate*, which in turn belongs to exactly one random variable. The value associated with a gate option must be a possible value of that random variable. It is specified using the *value* attribute. Gate options may be nested in each other as documented in the *is nested in* relationship.

When using the model, the purpose is to infer the posterior distributions conditioned on all observations to make predictions for new data. Hence, it must be indicated which observed variables (*priors*) are to be replaced with inferred posteriors. This is done using the Boolean attribute *infer?* of the entity type *Observed Variable*. After setting the posteriors, training data will be replaced with new data. Any variable for which no new data will be available can be marked as one for which a posterior predictive distribution shall be evaluated. This is done using the *predict?* attribute.

## 4.2 Code Generation Scheme

Based on the DSML from Section 4.1 a simple code generation scheme can be defined. The basic skeleton consists of a single C# class with three methods *GenerateModel*, *InferPosteriors*, and *MakePredictions*. Most relevant is *GenerateModel*, which is why I discuss code generation for this method only.

Ignoring gates for the moment, *GenerateModel* can be structured into different areas as illustrated in Figure 5. First, ranges must be defined that are used as indices for plates. This is done by processing all plates of a model and inserting a line of code for each of them:

```
Range name = new Range(plateSize).Named("name");
```



The second step is to define all variables. Again, this can be done by processing all entities of type *Variable* and appending lines of code:

```
varName =
Variable.New<dataType>().Named("varName");
```

*varName* and *dataType* are taken from the corresponding attributes of the entities. Additionally, the variable must be declared as a class attribute:

```
Variable<dataType> varName;
```

The third step is to define the factors. Hence, all entities of type factor must be processed to append new lines of code in the factor definition area. The structure of the code depends on a factor's type. Examples are:

```
varName = Variable<dataType>.Random(varDistName);
var3Name = var1Name > var2Name;
```

The first line assigns a distribution object *varDistName* to a variable *varName*. *varDistName* is an object representing a distribution. The second line defines the distribution over a Boolean variable *var3Name* as the probability of *var1Name* being larger than *var2Name*.

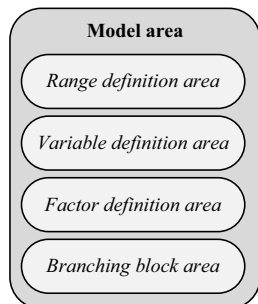


Figure 5. Structure of the model area.

If any variable belongs to a plate, the variable definition code is changed to an array version with corresponding dimensions. The factor definition also iterates over these dimensions. Special care is necessary if entries in the *selects* relationship are encountered. Fortunately, it can be handled easily by using the selecting variable as an index.

Finally, it must be accounted for the gates used in the model. This is done by nesting model areas into each other in the same way the *GateOption* entities are nested. A designated root model area is created first. All other model areas are nested by putting them into the *Branching block area* of their parent model area. Gates are defined in Infer.NET using different selection methods. If the selecting variable *branch* is Bernoulli, the code will look like this:

```
using (Variable.If(branch)){ // one model area }
using (Variable.IfNot(branch)){ // other area}
```

## 5. Conclusion, Limitations, and Outlook

Motivated by the lack of data scientists in industry I have proposed an initial conceptualization of a DSML supporting code generation from visual representations of probabilistic models for big data analytics. Starting at existing notations, extensions have been defined based on analysis of the Infer.NET modeling API. How modeling constructs correspond to code has been illustrated informally by a rudimentary code generation scheme. However, no actual code generator for the Infer.NET library has been implemented yet. Doing so would substantiate the claim that the DSML does in fact cover all constructs necessary. I plan to address this limitation in the future by implementing a DSML using Microsoft's Visualization and Modeling SDK.

Another limitation is the focus on only a single library, namely Infer.NET. Referring back to the analogy of linear programming, the ideal DSML would be library-independent and could generate code for a wide range of them. Unfortunately, these libraries are in an earlier stage of development. In the near future, I do not expect them to be standardized and stable enough to ensure easy interoperability. Integrating other libraries can therefore be a long term goal only.

It might be fruitful though to analyze standalone software packages that have been discarded in this paper's review. Some of them have GUIs, yet they are often meant to be used for educational examples instead of sophisticated models (e.g., DoodleBUGS, the GUI of the BUGS project [31]). Nevertheless, such GUIs may inform the design of concrete syntax for the DSML and could also reveal deficits that have not shown up so far. A review is left to future research.

## 6. References

- [1] T. H. Davenport and D. J. Patil, "Data Scientist : The Sexiest Job Of the 21st Century Data Scientist," *Harvard Business Review*, no. October, 2012.
- [2] S. Chaudhuri, U. Dayal, and V. Narasayya, "An overview of business intelligence technology," *Communications of the ACM*, vol. 54, no. 8, pp. 88–98, 2011.
- [3] H. J. Watson, "Tutorial: Business Intelligence – Past, Present, and Future," *Communications of AIS*, vol. 25, no. 39, pp. 487–510, 2009.
- [4] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A Comparison of Approaches to Large-Scale Data Analysis," in *SIGMOD '09*, 2009, pp. 165–178.
- [5] J. Dean and S. Ghemawat, "MapReduce : Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 1–13, 2008.

- [6] A. McAfee and E. Brynjolfsson, "Big Data: The Management Revolution," *Harvard Business Review*, October, 2012.
- [7] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, M. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, 2007.
- [8] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.
- [9] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From data mining to knowledge discovery: an overview," in *Advances in Knowledge Discovery and Data Mining*, 1996, pp. 1–34.
- [10] C. M. Bishop, "A New Framework for Machine Learning," in *IEEE World Congress on Computational Intelligence (WCCI 2008)*, 2008, pp. 1–24.
- [11] T. Minka, J. Winn, J. Guiver, and D. Knowles, "Infer.NET 2.5," 2012.
- [12] C. M. Bishop, "Model-based Machine Learning," *Philosophical transactions Series A Mathematical physical and engineering sciences*, vol. 371, no. 1984, 2013.
- [13] H. Giese and S. Henkler, "A survey of approaches for the visual model-driven development of next generation software-intensive systems," *Journal of Visual Languages Computing*, vol. 17, no. 6, pp. 528–550, 2006.
- [14] D. C. Schmidt, "Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.
- [15] J. Kärnä, J. P. Tolvanen, and S. Kelly, "Evaluating the Use of Domain-Specific Modeling in Practice," in *9th Workshop on Domain-Specific Modeling at OOPSLA*, 2009.
- [16] J. P. Tolvanen and S. Kelly, "Defining Domain-Specific Modeling Languages to Automate Product Derivation : Collected Experiences," in *9th International Software Product Line Conference*, 2005, vol. 3714, pp. 198–209.
- [17] D. Heckerman, "A Tutorial on Learning With Bayesian Networks," Redmond, 1996.
- [18] M. I. Jordan, "An introduction to probabilistic graphical models," 2003.
- [19] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York: Springer, 2006.
- [20] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning - Data Mining, Inference, and Prediction*, 2nd ed. 2009.
- [21] M. I. Jordan, "Graphical models," *Statistical Science*, vol. 19, no. 1, pp. 140–155, 2004.
- [22] OMG, "Unified Modeling Language (UML) - Version 2.4.1 August 2011." 2011.
- [23] OMG, "Business Process Model and Notation (BPMN) - Version 2.0 January 2011." 2011.
- [24] T. Minka and J. Winn, "Gates: A graphical notation for mixture models," in *Neural Information Processing Systems (NIPS)*, 2008, pp. 1073–1080.
- [25] K. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.
- [26] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society Series B (Methodological)*, vol. 50, no. 2, pp. 157–224, 1988.
- [27] F. R. Kschischang and B. J. Frey, "Factor Graphs and the Sum-Product Algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.
- [28] P. Dagum and M. Luby, "Approximating probabilistic inference in Bayesian belief networks is NP-hard," *Artificial Intelligence*, vol. 60, pp. 141–153, 1993.
- [29] J. Winn and C. M. Bishop, "Variational Message Passing," *Journal of Machine Learning Research*, vol. 6, no. 1, pp. 661–694, 2005.
- [30] T. P. Minka, "Expectation propagation for approximate Bayesian inference," in *UAI'01*, 2001, pp. 362–369.
- [31] D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best, "The BUGS project: Evolution, critique and future directions," *Statistics in Medicine*, vol. 28, no. 25, pp. 3049–3067, 2009.
- [32] M. P. Wand, J. T. Ormerod, S. A. Padoan, and R. Frühwirth, "Mean field variational Bayes for elaborate distributions," *Bayesian Analysis*, vol. 6, no. 4, pp. 1–48, 2011.
- [33] K. P. Murphy, "Software for Graphical models: a review," *International Society for Bayesian Analysis Bulletin*, vol. 14, pp. 13–15, 2007.
- [34] M. Plummer, "JAGS : A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling JAGS : Just Another Gibbs Sampler," in *3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, 2003.
- [35] Stan Development Team, "Stan Modeling Language User's Guide and Reference Manual - Version 1.3." 2013.
- [36] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modelling Language for Mathematical Programming*. Monterey: Duxbury Press, Brooks/Cole Pub. Co., 2002.
- [37] G. B. Dantzig, "Linear Programming," *Operations Research*, vol. 50, no. 1, pp. 42–47, 2002.