Technical Report

Probabilistic Graphical Models in RapidMiner

Nico Piatkowski

2/2011

technische universität dortmund

SFB 876 Verfügbarkeit von Information durch Analyse unter Ressourcenbeschränkung

Part of the work on this technical report has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A1.

Speaker: Prof. Dr. Katharina Morik
Address: TU Dortmund University
Joseph-von-Fraunhofer-Str. 23
D-44227 Dortmund
Web: http://sfb876.tu-dortmund.de

1 Introduction

Structured data has become common in a large set of machine learning tasks like text segmentation [9, 20], denoising and classification of images [25], 3-D depth reconstruction [19], protein side-chain prediction [28], or web page labeling [3]. A structured training instance $(\boldsymbol{y}, \boldsymbol{x})$ consists of a label vector $\boldsymbol{y} \in \mathcal{Y}_1 \times \mathcal{Y}_2 \times \cdots \times \mathcal{Y}_n$ and an observation vector $\boldsymbol{x} \in \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_m$. Both, $\boldsymbol{y}$ and $\boldsymbol{x}$, may contain dependencies within and between each other. Probabilistic Graphical Models (PGMs) [26] are a common approach to model the interactions of those variables. Although several variants of PGMs like Naive Bayes (NB), Bayesian Networks (BNs), Hidden Markov Models (HMMs) [18], Markov Random Fields (MRFs) [5], Gaussian Markov Random Fields [22] or Conditional Random Fields (CRFs) [9] exist, the Factor Graph representation as described by Kschischang et al. [8] covers them all, since it allows arbitrary funtional dependencies between random variables.

This Report describes the technical background and usage of the GraphMod plug-in for RapidMiner [12]. The plug-in enables RapidMiner to load factor graphs and interpret `Label` and `Attributes` which are contained in an `Example` as assignments to random variables. A set of examples which belong to the same `Batch` is treated as assignment to a whole factor graph. New operators allow the estimation of factor weights, the computation of the single-node marginal probability functions and the computation of the most probable assignment for each `Label`-node with several methods. All algorithms are optimized for parallel execution on common multi-core processors and NVIDIA CUDA [14] capable many-core processors (also known as Graphics Processing Unit).

The remainder of this report is organized as follows. Section 2 shortly reviews the general theory of PGMs. The RapidMiner operators which are contained in the GraphMod plug-in and technical details on their implementation are presented in Section 3. Section 4 shows how to create factor graphs which may be loaded by the plug-in. Two example processes are shown and explained in Sec. 5. Section 6 subsumes this report and gives a short outlook on future work.

2 Probabilistic Graphical Models

We will now introduce the necessary notation and the basic concept of PGMs. Following [8], we focus on graphical models which are represented as factor graphs. A factor graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ consists of a set of variable nodes $\mathcal{V}$, a set of factor nodes $\mathcal{F}$ and a set of undirected edges between factors and variable nodes $\mathcal{E} \subseteq \mathcal{F} \times \mathcal{V}$. In the following, we assume that the nodes in $\mathcal{V}$ are partitioned into two distinct multivariate random variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ with assignments $\boldsymbol{x}$ and $\boldsymbol{y}$ and discrete domains $\mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_m$ and $\mathcal{Y}_1 \times \mathcal{Y}_2 \times \cdots \times \mathcal{Y}_n$. The nodes in $\boldsymbol{Y}$ are called *hidden nodes* and those in $\boldsymbol{X}$ are the *observed* ones. For notational convenience, let $\boldsymbol{y} \in \mathcal{Y}^n$ and $\boldsymbol{x} \in \mathcal{X}^m$. Furthermore, let $\boldsymbol{y}_\mathcal{U}$ be the joint assignment $\boldsymbol{y}$ restricted to the nodes in an arbitrary set $\mathcal{U} \subseteq \boldsymbol{Y}$. The nodes $f \in \mathcal{F}$ are corresponding to positive functions $f : \mathcal{Y}^{|\Delta(f)|} \times \mathcal{X}^{|\tilde{\Delta}(f)|} \to \mathbb{R}^+$, where $\Delta : \mathcal{F} \to 2^{\boldsymbol{Y}}$ assigns[1] to a factor node a set of its adjacent nodes which correspond to

[1] Here, $2^\mathcal{U}$ denotes the power set of a set $\mathcal{U} \subseteq \mathcal{V}$.

random variables in $\boldsymbol{Y}$ and $\tilde{\Delta}$ those which corresponds to variables in $\boldsymbol{X}$, respectively.

A general factor graph defines a function of all nodes in $\mathcal{V}$ that factorizes over smaller functions, namely those in $\mathcal{F}$. It is known from [5], that the joint probability density function (pdf) over a set of random variables $\mathcal{V}$ factors over the cliques of the corresponding dependency graph $\mathcal{G}$ if its edge set encodes a conditional independence structure between the variables in $\mathcal{V}$. Thus, a factor graph that models the same distribution contains a factor for each clique of $\mathcal{G}$. The pdf of a general MRF over a multivariate random variable $\boldsymbol{y}$ is shown in Eq. 1. The quantity $\mathcal{Z}$ is a normalization factor to ensure that $\sum_{\boldsymbol{y}} p(\boldsymbol{y}) = 1$.

$$p_{\boldsymbol{\theta}}(\boldsymbol{Y} = \boldsymbol{y}) = \mathcal{Z}^{-1} \prod_{f \in \mathcal{F}} f_{\boldsymbol{\theta}}(\boldsymbol{y}_{\Delta(f)}) \tag{1}$$

If an observed vector $\boldsymbol{x}$ is given, one may formulate a CRF (Eq. 2). In this case, the normalization is a function of $\boldsymbol{x}$. As PGMs are usually parametrized, the factors and thus the whole pdf depends on a parameter set $\boldsymbol{\theta}$. In case of MRFs, $\boldsymbol{\theta}$ contains one weight vector $\boldsymbol{\theta}_f$ per factor node, each of size $\mathcal{O}(|\mathcal{Y}|^{\Delta_{\max}})$. For CRFs, each $\boldsymbol{\theta}_f$ contains again a set of conditional weight vectors $\boldsymbol{\theta}_{f|v=x}$ for each possible combination of observed neighbor $v \in \tilde{\Delta}(f)$ and corresponding assignment $x \in \mathcal{X}$. Depending on the actual type of PGM, factors $f, g \in \mathcal{F}$ with a common domain may share the same set of parameters, that is $\boldsymbol{\theta}_f = \boldsymbol{\theta}_g$. A set of factors which share the same parameter set is called *factor template*.

$$p_{\boldsymbol{\theta}}(\boldsymbol{Y} = \boldsymbol{y} | \boldsymbol{X} = \boldsymbol{x}) = \mathcal{Z}(\boldsymbol{x})^{-1} \prod_{f \in \mathcal{F}} f_{\boldsymbol{\theta}}(\boldsymbol{y}_{\Delta(f)}, \boldsymbol{x}_{\tilde{\Delta}(f)}) \tag{2}$$

A factor node measures how likely a given assignment to its neighbors is. In discrete CRFs, they usually take the form of Eq. 3. Let $\mathbf{1}_{\boldsymbol{Y}_{\Delta(f)} = \boldsymbol{y}_{\Delta(f)}}$ be a binary vector of size $|\mathcal{Y}|^{\Delta_{\max}}$ which contains 0 on all positions except for position $\boldsymbol{y}_{\Delta(f)}$ where it contains a 1.

$$f_{\boldsymbol{\theta}}(\boldsymbol{y}_{\Delta(f)}, \boldsymbol{x}_{\tilde{\Delta}(f)}) = \prod_{v \in \tilde{\Delta}(f)} \exp\left[\left\langle \boldsymbol{\theta}_{f|v=\boldsymbol{x}_v}, \mathbf{1}_{\boldsymbol{Y}_{\Delta(f)} = \boldsymbol{y}_{\Delta(f)}} \right\rangle\right] \tag{3}$$

By definition of $\mathbf{1}_{\boldsymbol{Y}_{\Delta(f)} = \boldsymbol{y}_{\Delta(f)}}$, the dot product in Eq. 3 simply selects the weight which corresponds to the assignment $\boldsymbol{y}_{\Delta(f)}$.

To complete the framework for PGMs, Maximum Likelihood Estimation (MLE) is used to approximate the most probable set of parameters for a given set of N totally observed assignments (or training instances) $\mathcal{T} = \{(\boldsymbol{y}, \boldsymbol{x})_i\}_{1 \leq i \leq N}$. The parameters are updated by moving the current vector $\boldsymbol{\theta}^t$ into the direction of the steepest ascent of the likelihood. The resulting update rule for CRFs is given in Eq. 4. Here, η_t is a dynamic stepsize and $1_{\boldsymbol{y}'_{\Delta(f)} = \boldsymbol{y}_{\Delta(f)}}$ is a binary indicator function which evaluates to 1 iff the expression in the subscript is true and 0 otherwise. Furthermore, let $\mathcal{T}(v, x) := \{(\boldsymbol{y}', \boldsymbol{x}') \in \mathcal{T} | \boldsymbol{x}'_v = x\}$ be the set of all training examples where the random variable v was observed with assignment

x. The term $p_{\boldsymbol{\theta}}\left(\boldsymbol{y}_{\Delta(f)}|\boldsymbol{x}\right)$ corresponds to the marginal probability of nodes in $\Delta(f)$ having the assignment $\boldsymbol{y}_{\Delta(f)}$. The two most common methods for the computation of this quantity are investigated in Sec. 2.1. In case of MRFs, the update rule simplifies to Eq. 5. See [23] for a detailed derivation of these update rules.

$$\boldsymbol{\theta}^{t+1}_{f|v=x}(\boldsymbol{y}_{\Delta(f)}) = \boldsymbol{\theta}^{t}_{f|v=x}(\boldsymbol{y}_{\Delta(f)}) + \eta_t \sum_{(\boldsymbol{y}',\boldsymbol{x}')\in\mathcal{T}(v,x)} \left[1_{\boldsymbol{y}'_{\Delta(f)}=\boldsymbol{y}_{\Delta(f)}} - p_{\boldsymbol{\theta}}\left(\boldsymbol{y}_{\Delta(f)}|\boldsymbol{x}'\right)\right] \tag{4}$$

$$\boldsymbol{\theta}^{t+1}_{f}(\boldsymbol{y}_{\Delta(f)}) = \boldsymbol{\theta}^{t}_{f}(\boldsymbol{y}_{\Delta(f)}) + \eta_t \sum_{(\boldsymbol{y}',\boldsymbol{x}')\in\mathcal{T}} \left[1_{\boldsymbol{y}'_{\Delta(f)}=\boldsymbol{y}_{\Delta(f)}} - p_{\boldsymbol{\theta}}\left(\boldsymbol{y}_{\Delta(f)}\right)\right] \tag{5}$$

The single-node marginals $p_{\boldsymbol{\theta}}\left(\boldsymbol{y}_v|\boldsymbol{x}\right)$ may also be used to derive the most probable assignment for a node v by choosing an assignment according to Eq. 6.

$$\boldsymbol{y}^*_v = \arg\max_{\boldsymbol{y}_v\in\mathcal{Y}} p_{\boldsymbol{\theta}}\left(\boldsymbol{y}_v|\boldsymbol{x}\right) \tag{6}$$

Although it is possible to convert a factor graph with factors of arbitrary size into a pairwise model (See [26], Appendix E.3) we keep the general notation, since the corresponding pairwise model simplyfies the notation but hides the exponential runtime complexity of marginalization.

2.1 Marginal distribution

One of the main targets when dealing with PGMs is to compute the margnial distribution (Eq. 7). They may be used to predict the most probable assignment for a variable node or to adjust the PGM parameters $\boldsymbol{\theta}$. For notational convenience, let $\mathcal{V}$ be the set $\{1, 2, \ldots, n\}$.

$$p_{\boldsymbol{\theta}}(\boldsymbol{y}_v|\boldsymbol{x}) = \sum_{\boldsymbol{y}_1}\sum_{\boldsymbol{y}_2}\cdots\sum_{\boldsymbol{y}_{v-1}}\sum_{\boldsymbol{y}_{v+1}}\cdots\sum_{\boldsymbol{y}_n} p_{\boldsymbol{\theta}}(\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_n|\boldsymbol{x}) \tag{7}$$

Obviosly, the complexity of a complete enumeration of all possible assignments to the whole graph is $\mathcal{O}(|\mathcal{Y}|^{n-1})$, which is intractable for most choices of n. The GraphMod plug-in currently contains two methods for marginalization which are much more efficient than the naive enumeration. The first one exploits the factorization of $p_{\boldsymbol{\theta}}$ to reduce the time complexity. The second one is based on a sampling approach.

2.1.1 Belief Propagation

The computation of conditional marginal probabilities $p(\boldsymbol{y}_{\Delta(f)}|\boldsymbol{x})$ with Belief Propagation (BP) consists in repeatedly computing Eq. 8 and 9. Both equations have to be computed for all possible combinations of factor node $f \in \mathcal{F}$, neighboring variable node $v \in \Delta(f)$, corresponding assignment $\boldsymbol{y}_v \in \mathcal{Y}$ and training instance $\boldsymbol{x} \in \mathcal{T}$. This process may be

interpreted as the sending of messages and is therefore called Belief Propagation [16] or Sum-Product algorithm [8].

$$m_{f\to v}(\boldsymbol{y}_v|\boldsymbol{x}) = \sum_{\boldsymbol{y'}_{\Delta(f)-v}} f_{\boldsymbol{\theta}}(\boldsymbol{y}_v, \boldsymbol{y'}_{\Delta(f)-v}, \boldsymbol{x}) \prod_{u\in\Delta(f)-v} m_{u\to f}(\boldsymbol{y'}_u|\boldsymbol{x}) \tag{8}$$

$$m_{v\to f}(\boldsymbol{y}_v|\boldsymbol{x}) = \prod_{g\in\Delta^{-1}(v)-f} m_{g\to v}(\boldsymbol{y}_v|\boldsymbol{x}) \tag{9}$$

One iteration of BP is finished, if all nodes have computed their new outbound messages. The number of iterations I and therefore the overall complexity of BP heavily depends on the actual graphical structure. In general, when the graph is undirected and contains loops, both equations must be computed multiple times until convergence or if a maximum number of iterations $I_{\max}$ is reached. In tree structured undirected models, the number of iterations is known, since the process can safely be stopped if each node has received the messages from all other nodes in the graph through its neighbors. In case of tree structured directed models, messages only need to be passed in edge direction. Hence, message must be computed once for each depth-level of the tree. Independent of the graphical structure, the complexity of computing an outgoing message from a factor is $\mathcal{O}(\Delta_{\max}|\mathcal{Y}|^{\Delta_{\max}})$ and therefore exponential in $\Delta_{\max}$.

After BP has terminated, the single-node marginals may be approximated with Eq. 10 and those of a factor node with Eq. 11. The normalization factors $\mathcal{Z}_v$ and $\mathcal{Z}_{\Delta(f)}$ have to ensure that the corresponding functions sum to 1.

$$p_{\boldsymbol{\theta}}(\boldsymbol{y}_v|\boldsymbol{x}) = \mathcal{Z}_v(\boldsymbol{x})^{-1} m_{f\to v}(\boldsymbol{y}_v|\boldsymbol{x}) m_{v\to f}(\boldsymbol{y}_v|\boldsymbol{x}) \tag{10}$$

$$p_{\boldsymbol{\theta}}(\boldsymbol{y}_{\Delta(f)}|\boldsymbol{x}) = \mathcal{Z}_{\Delta(f)}(\boldsymbol{x})^{-1} f(\boldsymbol{y}_{\Delta(f)}, \boldsymbol{x}_{\tilde{\Delta}(f)}) \prod_{v\in\Delta(f)} m_{v\to f}(\boldsymbol{y}_v|\boldsymbol{x}) \tag{11}$$

In case of tree structured graphs, the computed margnials are exact. A formal proof is omitted here, but one easily derives Eq. 7 by recursively substituting Eq. 8 and Eq. 9 into Eq. 10 followed by a rearrengement of terms and factors. In loopy graphs, BP will converge if the dynamic range of each factor $f_{\boldsymbol{\theta}}$ satisfies a certain property [6].

2.1.2 Markov Chain Monte Carlo

Another common way to approximate the marginal probabilities in factor graphs is a Markov Chain Monte Carlo (MCMC) method called Gibbs sampling [4]. The idea is to repeatedly replace each single-node assignment $\boldsymbol{y}_v$ with a value picked from a distribution which is conditioned on the current values of all other variable nodes. This process can be seen as generating a realization of a Markov chain.

$$\boldsymbol{y}_v^{t+1} \sim \mathcal{Z}_v(\boldsymbol{x})^{-1} \prod_{f\in\Delta^{-1}(v)} f_{\boldsymbol{\theta}}(\boldsymbol{y}_v, \boldsymbol{y}^t_{\Delta(f)-v}, \boldsymbol{x}_{\tilde{\Delta}(f)}) \tag{12}$$

The algorithm consists in repeatedly sampling each nodes assignment according to Eq. 12 for a fixed number of I_{max} iterations while counting how often node v had a certain assignment $\boldsymbol{y}_v$. Let $\#(v = y|\boldsymbol{x})$ be this absolute frequency, then the relative frequency $\#(v = y|\boldsymbol{x})I^{-1}$ is an estimate for the wanted marginal probability $p_{\boldsymbol{\theta}}(\boldsymbol{y}_v|\boldsymbol{x})$. To estimate the factor marginals $p_{\boldsymbol{\theta}}(\boldsymbol{y}_{\Delta(f)}|\boldsymbol{x})$, the joint assignments to all neighbors of each factor node has to be counted after each MCMC iteration, i.e. after all variable nodes have been resampled. This procedure will indeed converge to the true marginal distribution if I approaches infinity. As an variety, one may only count assignments which pass a randomized acceptance procedure, which leads to the Metropolis-Hastings algorithm [11]. A more detailed explanation about the application of this algorithm to factor graphs can be found in [27].

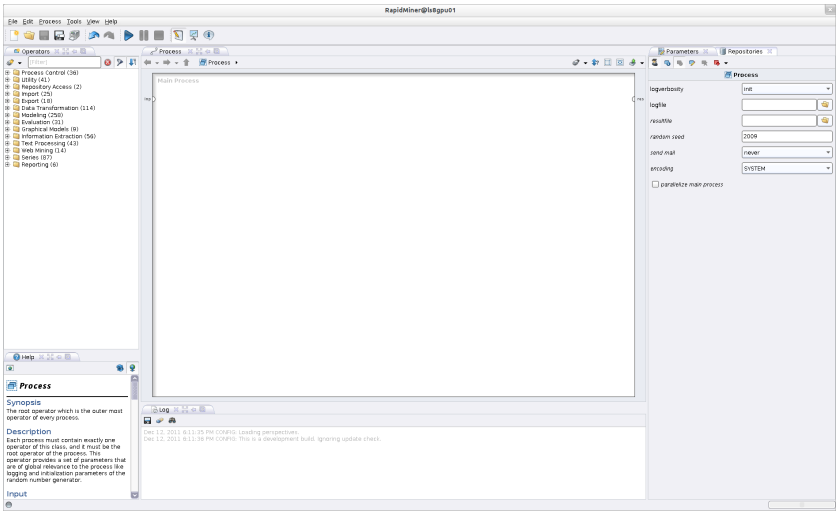

Figure 1: Screenshot of RapidMiner version 5.1. The folder `Graphical Models` contains 9 new operators.

3 The GraphMod plug-in

The GraphMod plug-in for RapidMiner (Fig. 1) implements PGMs like described in Sec. 2 of this report. A usual GraphMod process loads or creates a factor graph and either estimates the model parameters from a given example set or approximates the marginal distribution, which also yields the most probable assignment for each node. Additionally, the plug-in contains operators to modify a factor graph or to extract the estimated weight vectors.

3.1 Layout of `ExampleSets`

Like operators for time series, the GraphMod plug-in requires that the data which is contained in an `ExampleSet` obeys a certain layout. Each row has to contain an example identifier i and node identifier j. The remaining fields of a data row correspond to a vector $(\boldsymbol{y}_j^{(i)}, \boldsymbol{x}_{j,1}^{(i)}, \ldots, \boldsymbol{x}_{j,m}^{(i)})$ which at least contains the assignment $\boldsymbol{y}_j^{(i)}$ of hidden node

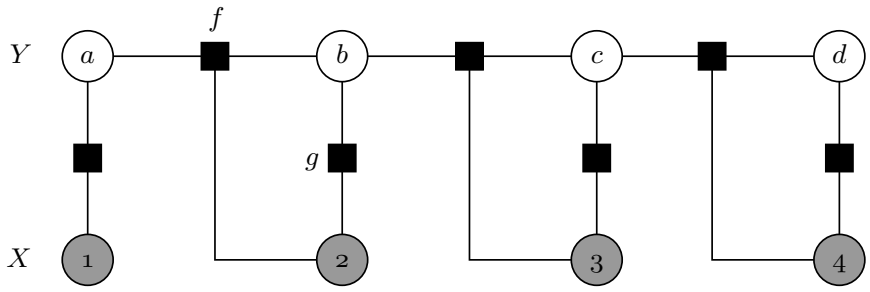

Figure 2: Factor graph of a linear-chain conditional random field.

j in example i and in case of conditioned models also the assignments $\boldsymbol{x}_{j,1}^{(i)}, \ldots, \boldsymbol{x}_{j,m}^{(i)}$ of some observed nodes. The column of $\boldsymbol{y}_j^{(i)}$ needs to have the role `Label` and the $\boldsymbol{x}_{j,m}^{(i)}$ should be marked as `Regular Attributes`. This data layout fits to regular structures like linear-chains or grids but also applies to irregular structures like proteins.

3.2 Operator: `Create Factor Graph`

The `Create Factor Graph` operator can be used to create new factor graphs. It has no input ports and its output is a new factor graph, which is either empty, i.e. it contains no nodes, or obeys a linear-chain structure (Fig. 2). The corresponding property is called `Structure` which allows to choose between `Empty` and `Linear-chain`. In case of predefined structures, optional parameters may appear, e.g. for linear-chain structures the names of attributes which should be used as observations at each time step can be supplied as comma seperated list. Additionally, it is possible to indicate that all hidden nodes $v \in \boldsymbol{Y}$ have the same domain by setting the binary property `Shared domain` to `true`.

3.3 Operator: `Add Factor Node`

This operator reads a factor graph as input, adds a new factor to it and returns the resulting factor graph. The main property of a factor node is its `Neighborhood`. The neighbors of a new factor node are defined by a comma seperated list of integers which are used to address hidden nodes with their node identifiers. To connect a factor with observed nodes, attribute names have to be supplied, each followed by an underscore and a node identifier. Thus, the connectivity of the whole graph is fully specified by its set of factors. Each factor may also get a `Name` which can be set as a property of this operator. If no `Name` is supplied, the new factor will get the name `unnamed_`k, whereas $k \geq 1$ equals the total number of unnamed nodes. Furthermore, a factor may get a `Tempalte Name`, which identifies the template to which a factor belongs. If two factors $f, g \in \mathcal{F}$ have the same `Tempalte Name`, they share the same set of parameters, that is $\boldsymbol{\theta}_f = \boldsymbol{\theta}_g$. Because of the template names, it is possible to apply a model which was learned on graph G to a different graoh G'. If a factor node is unconditioned, i.e. it is not connected to any observed node, then the weights of this factor may be set manually by activating the

`Set weights` button. This is not possible for conditional factors, since the domains of the observed nodes are extracted from an `ExampleSet` which is unknown at construction time of the graph.

3.4 Operator: `Set Neighborhood`

The neighborhood of a factor node may be changed with this operator. It reads a factor graph as input, changes the neighborhood of a given factor node and returns the resulting factor graph. The factor is addressed by its `Name`. The `Neighborhood` should be specified as described above.

3.5 Operator: `Set Template`

If a factor node should be remapped to another template, this operator can be used. It reads a factor graph as input, changes the `Template Name` of a given factor and returns the resulting factor graph. The operator expects the `Name` of an existing factor and replaces its corresponding template with another one which has to be supplied as property `Template Name`.

3.6 Operator: `Generate Sample`

This operator expects a model as input and generates an example set which contains samples that are drawn from the estimated distribution. To use this operator, each factor in the corresponding graph has to be unconditioned, i.e. has no `Regular Attributes` as neighbors. The only property of this operator is the `Number of samples` which should be generated. One factor is randomly selected and a joint assignment to his neighboring variable nodes is sampled according to its marginal distribution $p_{\boldsymbol{\theta}}(\boldsymbol{y}_{\Delta(f)}|\boldsymbol{x})$. The process is continued at all factors which still have unassigned neighbors.

3.7 Operator: `Load GraphML`

Instead of creating a factor graph directly in RapidMiner, is is possible to load a factor graph from a GraphML [2] file. The graph may be either supplied by the `File` input port of the operator or by providing the `File Name`. In both cases, the operator delivers a factor graph object at its output port. The GraphML format is based on XML and supports the entire range of possible graph structure constellations. The semantics is derived from the definition of factor graphs [8]. A black rectangle is interpreted as a factor node. A black ellipse is considered as observed node and all other ellipses represent hidden nodes. GraphML allows to equip nodes with names. In case of a factor node, this name is interpreted as its `Template Name`. The name of observed nodes has to correspond to an attribute name, followed by an underscore seperated node identifier like described above. Hidden nodes must have an integer number as name which corresponds to its node identifier. For now the only editor which is known to generate compatible files is

the free graphical editor yEd. The creation of factor graphs with yEd is described in Sec. 4.1. Currently it is not possible to store an entire model in a GraphML file, only the graphical structure can be loaded. In analogy to the `Create Factor Graph` operator, it is possible to indicate that all hidden nodes $v \in \boldsymbol{Y}$ have the same domain by setting the binary property `Shared domain` to `true`.

3.8 Operator: `Load MRF`

As an alternative to direct graph creation or GraphML files, unconditioned graphical models, i.e. MRFs, may be loaded by a simple proprietary file format. The graph may be either supplied by the `File` input port of the operator or by providing the `File Name`. In contrast to GraphML files, MRF files usually contain factor weights. Hence, the operator delivers a factor graph object and a `Model` at its output ports. Such models may be used as input for the `Generate Sample` operator or, if applied by a `Apply Model` operator to compute the single-node marginal distributions. In order to use the `Apply Model` operator, an example set has to be supplied which is not available in case of Markov Random Fields. Hence, the `Load MRF` operator additionally outputs an example set which contains one GraphMod example. The content of an MRF file is basicly a list of variables followed by list of factors. Each factor may may be followed by a list of weights. Since this file format does not allow the use of names or other identifiers for factors, the explicit use of factor templates in MRF files is not possible. A detailed example on how to create an MRF file is given in Sec. 4.2.

3.9 Operator: `PGM`

The PGM operator expects a factor graph and an example set as input and outputs a model, which contains a set of parameters $\boldsymbol{\theta}$ and a copy of the graphical structure on which the model was trained on. The structural information has to be stored in the model, since it is required to apply the model to unlabeled testing data. The engine which performs the actual computation of the marginal distribution may be changed by the property `Engine`. Possible values are `LBP` which is an implementation of the method described in Sec. 2.1.1 and `MCMC` which corresponds to the sampling procedure described in Sec. 2.1.2. The other properties of this operator are the same for both engines. The property `Marginalization Iterations` sets the maximum number of iterations $I_{\max}$. To perform a Stochastic Gradient Descent training, the property `Batched Training` can be set to `true`. In this case, the batchsize should be set with parameter `Batchsize`. The number of updates (Eqs. 4 and 5) which are performed can be set up with the property `Training Iterations`. The stepsize η which is used in the update rules may be `fixed` or `annealed`. If η is fixed its value can be set as property `Stepsize`. Otherwise, the stepsize will be annealed like $\eta_t = \frac{1}{\sqrt{t}}$, whereas t is the number of the current update. Finally, the type of `Hardware parallelization` may be set to either `Multi-Core (CPU)` or `Many-Core (GPU)`. Sec. 3.10.1 gives a short overview on the parallelization.

3.10 Operator: `Exchange Factor Graph`

A model which was learned on a graph, may be coupled with another graph if both graphs are using the same factor templates. This operator expects a factor graph and a model which was generated by the `PGM` operator as input, checks both for compatibility by comparing the corresponding sets of factor templates and outputs a model which contains the new factor graph. The model may then be applied as usual.

3.10.1 Parallelization

The Multi-Core parallelization is based on the GraphLab framework [10]. GraphLab is written in C++ and allows an easy distribution of computations which are associated with nodes over the cores of multi-core processors. Although a exemplary JAVA binding of GraphLab is available, most of the GraphLab classes had to be converted to JAVA to ensure the compatibility with RapidMiner.

In general, all computations which are local to one node (e.g. message computations or resampling and counting) are called *task*. For one iteration of BP or MCMC, the tasks are distributed over all available cores. In order to perform training or testing a PGM, the tasks have to be computed for a set of instances. The multi-core implementation iterates over this set while the many-core implementation performs the local computations for each instance in parallel. Furthermore, the many-core version applies Thread-Cooperative LBP to reduce the complexity of marginalization. The Thread-Cooperative approach is described in a former technical report [17].

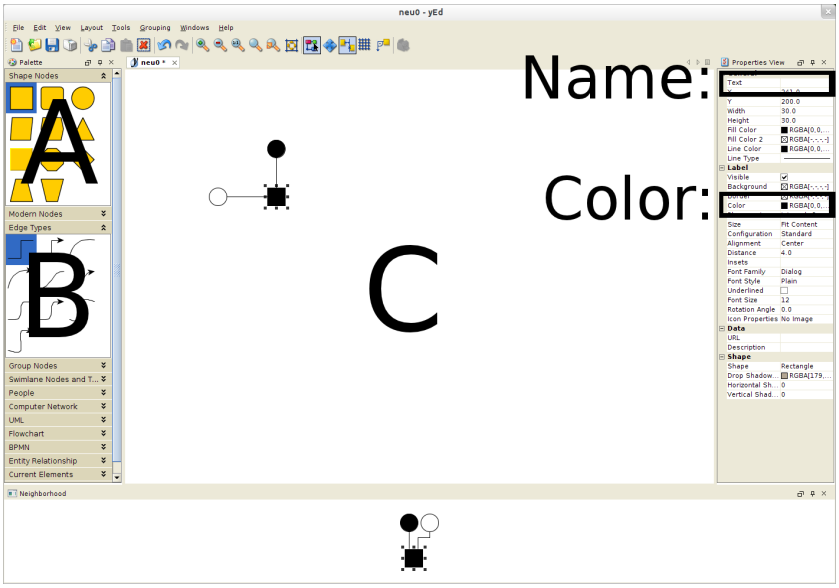

Figure 3: Screenshot of yEd version 3.8. The shape of a new node may be selected in area A, the edge type can be choosen in area B and the actual graph is drawn in area C. The name and the color of a node may be set in right panel.

4 File formats for factor graphs

Information about an actual graphical structure is needed in order to apply graphical models. Those structures may directly generated within RapidMiner or loaded from file by the operators described in Sec. 3. Here, the generation of graphs with external tools is described. Those graphs may be subsequently loaded into RapidMiner.

4.1 Generating GraphML files with yEd

GraphML is an XML-based file format for graphs and may therefore be written by hand. As mentioned above, factor graphs may be easily generated with a free software called yEd[2]. A screenshot of yEd is shown in Fig. 3. The program allows graph creation by simply dragging the nodes from area A and dropping them on area C. Nodes may be connected by selecting an edge type in area B, clicking on a node and dropping the edge on another node. Only nodes with `Rectangle` or `Ellipse` shape are recognized by the `Load GraphML` operator. Nodes must be connected by a `Polyline` which corresponds to an undirected edge. Each node should have assigned a name and a color. A white ellipse is interpreted as hidden node and must have an integer number for name which corresponds to its node identifier. Black ellipses are interpreted as observed nodes. Their name should match the name of an attribute in RapidMiner followed by an integer (node identifier) to select the actual data row which contains the assignment to this node. Fianlly, a black rectangle corresponds to a factor node. The name of a factor identifies its template. A explained in Sec. 3.3, factors with the same name will share the same set of parameters. Since the native file format for yEd is GraphML, the designed model needs simply be saved on disk and may be loaded afterwards by the `Load GraphML` operator.

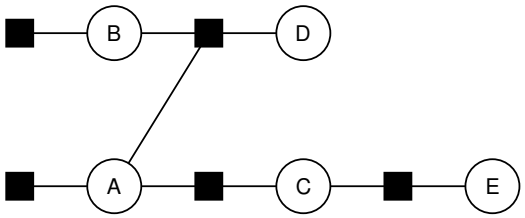

Figure 4: A simple Markov Random Field.

4.2 Writing MRF files

The second file type supported by GraphMod is a simple list format. It mainly consists of two lists. The elements of the first list are starting after a line which solely contains the string `variables:`. Each element is a pair of name and domain size of the corresponding random variable. The second list is introduced by the keyword `factors:`. Each line of this list corresponds to a factor f and begins with slash separated list of variables. Those variables are forming the neighborhood $\Delta(f)$ of the current factor.

[2]The program is available for download at: `http://www.yworks.com/en/products_yed_about.html`

$$||\Delta(f)|| = \prod_{v \in \Delta(f)} |\mathrm{Dom}(v)| \tag{13}$$

The neighborhood is terminated by a double-slash `//`, followed by a space separated list of $||\Delta(f)||$ weights. The total number of weights may be computed with Eq. 13. As an example, Listing 1 generates the factor graph which is shown in Fig. 4. Since names or other identifiers for factors are not supported, the explicit use of factor templates is not possible in this file format.

```
variables:
A 2
B 2
C 2
D 2
E 2
factors:
A //-0.356 -1.203///
B //-0.916 -0.510///
A /C //-0.510 -1.609 -0.916 -0.223///
A /B /D //-0.223 -1.609 -0.105 -1.203 -1.609 -0.223 -2.302 -0.356///
C /E //-0.223 -1.203 -1.609 -0.356///
```

Listing 1: Sample MRF file

5 Sample processes

We will now present two exemplary sample processes to clarify the usage of the GraphMod plug-in within RapidMiner. The samples contain a description of how to connect the involved operators and the input which should be supllied to achieve a desired output.

5.1 Simple Markov Random Field

This example shows how to apply Markov Random Fields by using the MRF file format. The first part of this process-chain is the `Load MRF` operator. The file which is shown in Lst. 1 is used as input, whereas the given factor weights are logs of the entries of the corresponding conditional probability tables. The graphical structure and the weights were taken from an example for baysian networks[3]. The second operator is `Apply Model`. It is connected with the model output port as well as the example set output port of the `Load MRF` operator. Finally, all output of the `Apply Model` are connected to the output ports of this process. When the process is started, it applies Belief Propagation to estimate the single-node marginal probabilities. The result is shown in a simple table which contains the marginal probabilities for each node an each assignment.

[3]URL of example for baysian networks: `http://cs.nyu.edu/faculty/davise/ai/bayesnet.html`

6 Summary and Future Work

This report presented the GraphMod plug-in, which allows the easy application of probabilistic graphical models within RapidMiner. The first Section cantained a short introduction. The theoretical foundations of PGM together with the two most common algorithms for estimation of the marginal distribution were introduced in the second Section. The third Section described the operators which are contained in the GraphMod plug-in. Section 4 showed how to create file representations of factor graphs and two examples of how to use the operators in RapidMiner processes were given in the last Section.

To support a broader range of software plattforms, Windows and MacOS support will be added in the upcoming release. The GPU acceleration will be enhanced to support OpenCL [15] in order to be compatible with upcoming many-core architectures.

The parameter update, given by Eq. 4 or Eq. 5, is done by Stochastic Gradient Descent (SGD) optimization [25]. Additional optimization techniques may be added in future releases, since the implementations of BP and MCMC are completely decoupled from the optimization procedure.

The current selection of marginal distribution estimation algorithmes will be extended with Stochastic Belief Propagation [13] to reduce the complexity of ordinary BP, Gaussian Belief Propagation [1] to support numerical random variables which follow a gaussian distribution and Kernel Belief Propagation [21] to support numerical variables with an arbitrary distribution.

References

[1] Bickson, D.: Gaussian Belief Propagation: Theory and Application. Ph.D. thesis, University of Jerusalem (2009)

[2] Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.: GraphML Progress Report: Structural Layer Proposal. Proceedings of th 9th International Symposium on Graph Drawing (GD '01), LNCS 2265 pp. 501–512 (2002)

[3] Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T., Nigam, K., Slattery, S.: Learning to extract symbolic knowledge from the world wide web. In: Proceedings of the 15th national conference on Artificial Intelligence. pp. 509–516. AAAI '98/IAAI '98, American Association for Artificial Intelligence, Menlo Park, CA, USA (1998)

[4] Geman, S., Geman, D.: Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-6(6), 721–741 (1984)

[5] Hammersley, J.M., Clifford, P.: Markov fields on finite graphs and lattices. Unpublished Manuscript (1971)

[6] Ihler, A.T., Willsky, A.S.: Loopy Belief Propagation: Convergence and Effects of Message Errors. Journal of Machine Learning Research 6, 905–936 (2005)

[7] Jungermann, F.: Documentation of the Information Extraction Plugin for RapidMiner (2011)

[8] Kschischang, F.R., Frey, B.J., Loeliger, H.A.: Factor Graphs and the Sum-Product Algorithm. IEEE Trans. on Infor. Theory 47(2), 498–519 (2001)

[9] Lafferty, J., McCallum, A., Pereira, F.: Conditional Random Fields: Probabilistic models for segmenting and labeling sequence data. Proceedings 18th International Conf. on Machine Learning pp. 282–289 (2001)

[10] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new parallel framework for machine learning. In: Conference on Uncertainty in Arti. Intell. (UAI). California (July 2010)

[11] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. Journal of Medical Physics 21(6), 1087–1092 (1953)

[12] Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., Euler, T.: YALE: Rapid Prototyping for Complex Data Mining Tasks. In: Ungar, L., Craven, M., Gunopulos, D., Eliassi-Rad, T. (eds.) KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 935–940. ACM, New York, NY, USA (August 2006)

[13] Noorshams, N., Wainwright, M.J.: Stochastic Belief Propagation: Low-Complexity Message-Passing with Guarantees. Graphical Models pp. 1–33 (2011)

[14] NVIDIA Corporation: CUDA Programming Guide 4.0 (June 2011)

[15] OpenCLWorkingGroup: The OpenCL Specification 1.1. Khronos Group (September 2010)

[16] Pearl, J.: Probabilistic reasoning in intelligent systems: networks of plausible inference. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1988)

[17] Piatkows, N.: Parallel Algorithms for GPU accelerated Probabilistic Inference. In: Morik, K., Rhode, W. (eds.) Technical Report for Collaborative Research Center SFB876, pp. 6–10. TU Dortmund, Graduate School of the SFB876 (October 2011)

[18] Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE 77(2), 257–286 (Feb 1989)

[19] Saxena, A., Chung, S.H., Ng, A.Y.: 3-D depth reconstruction from a single still image. International Journal of Computer Vision (IJCV) 76, 53–69 (2007)

[20] Settles, B.: Biomedical Named Entity Recognition using Conditional Random Fields and rich feature sets. In: Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and its Applications. pp. 104–107. JNLPBA '04, Ass. for Computational Linguistics, Stroudsburg, PA, USA (2004)

[21] Song, L., Gretton, A., Bickson, D., Low, Y., Guestrin, C., Computational, G., Unit, N.: Kernel Belief Propagation. Artificial Intelligence and Statistics 15 (2011)

[22] Speed, Kiiveri: Gaussian Markov Distributions over Finite Graphs. Annals of Statistics 14(1), 138–150 (1986)

[23] Sutton, C., McCallum, A.: An Introduction to Conditional Random Fields for Relational Learning. In: Getoor, L., Taskar, B. (eds.) Introduction to Statistical Relational Learning. MIT Press (2007)

[24] Tjong Kim Sang, E.F., Buchholz, S.: Introduction to the CoNLL-2000 shared task: chunking. Association for Computational Linguistics, NJ, USA (2000)

[25] Vishwanathan, S.V.N., Schraudolph, N.N., Schmidt, M.W., Murphy, K.P.: Accelerated training of conditional random fields with stochastic gradient methods. In: ICML '06: Proceedings of the 23rd international conference on Machine learning. pp. 969–976. ACM, New York, NY, USA (2006)

[26] Wainwright, M.J., Jordan, M.I.: Graphical models, exponential families, and variational inference. Foundations and Trends in Machine Learning 1, 1–305 (January 2008)

[27] Wick, M., McCallum, A., Miklau, G.: Scalable probabilistic databases with factor graphs and MCMC. Proceedings VLDB Endow. 3, 794–804 (September 2010)

[28] Yanover, C., Schueler-Furman, O., Weiss, Y.: Minimizing and learning energy functions for Side-Chain prediction. Springer, Berlin (2007)