Mocapy
A Toolkit for Inference and Learning in Dynamic Bayesian Networks

Version 0.726 (beta)
Januari 2006

Thomas Hamelryck
Bioinformatics Center
Institute of Molecular Biology
University of Copenhagen
Universitetsparken 15, bygning 10
2100 Copenhagen
Denmark

E-mail: thamelry@binf.ku.dk

Contents

Chapter 1

What is Mocapy?

1.1 Introduction

Mocapy is a freely available toolkit that performs maximum likelihood (ML) or maximum *a posteriori* (MAP) parameter learning and inference in Dynamic Bayesian Networks (DBNs), using Markov Chain Monte Carlo (MCMC) methods. One of the special features of Mocapy is that parameter learning can be done on a cluster computer, which makes the inherently slow MCMC approach applicable for real-life DBN architectures and data set sizes. I routinely use Mocapy for sequence datasets that contain several hundred thousands of observations. The toolkit can of course very well be used on a simple desktop computer with a single processor for smaller datasets. Mocapy is a general purpose toolkit, but it contains quite some functionality that is specific to the field of Bioinformatics, and in particular Structural Bioinformatics. Mocapy was for example used for constructing a probabilistic model of protein structure [13].

Mocapy is fully implemented in Python, which is an interpreted language and thus inherently slower than a compiled language like C++. However, by careful design and making use of Python's `numpy` module (which extends python with an efficient array datatype and matching functions implemented in C and FORTRAN) Mocapy reaches a quite acceptable speed. Mocapy is very portable and runs on any machine were Python and `numpy` can be installed. For the parallel features of Mocapy, you'll the Python MPI module `pyMPI` [19] as well.

The architecture of the DBN can be fully specified by the user, and various forms of discrete and real valued data are supported. Incidentally, Mocapy loosely stands for *(Markov Chain) Monte Carlo and Python*, two key ingredients of the toolkit.

1.2 Dynamic Bayesian Networks

So what is a Bayesian Network? A Bayesian Network (BN) is a Directed Acyclic Graph (DAG) in which the nodes represent random variables

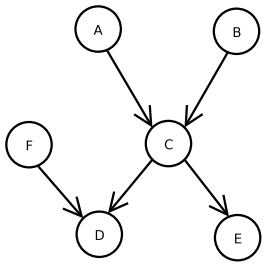

Figure 1.1: A very simple BN with 6 nodes and 5 edges.

and the edges represent conditional dependencies [9, 14]. The graph structure is a helpful tool for designing the probabilistic model and also makes it possible to develop efficient algorithms for learning and inference.

For example, in the simple BN shown in Fig. 1, the value of node C depends solely on the values of nodes A, B. Nodes A, B are *parent nodes* of node C, and node C is a *child node* of nodes A, B. The probability of observing a certain set of values a, b, c, d, e, f for nodes A, B, C, D, E, F is given by:

$$P(a, b, c, d, e, f) = P(c \mid a, b)P(d \mid c, f)P(e \mid c)P(a)P(b)P(f)$$

A Dynamic Bayesian Network (DBN) is a BN that can be used to build a probabilistic model of sequences (for a good overview, see [22]). The term 'dynamic' refers to the fact that a DBN is often used to model time sequences. In that respect, Sequential Bayesian Network would actually be a better name, since DBNs are also used to model sequences in which time does not play a role (for example amino acid or DNA sequences).

The well-known Hidden Markov Model (HMM, [26, 8]) can be considered as the simplest possible DBN. Variants of the classic HMM like Factorial HMMs [10] can be treated as DBNs as well. In general, the DBN provides a rich framework in which many HMM-like and other probabilistic models can be treated in a uniform way. Another advantage of using the DBN paradigm is that it can lead to a drastic reduction in the number of parameters as compared to an HMM representation. A DBN toolkit like like Mocapy allows the user to concentrate on the model itself, without having to implement customized inference and learning algorithms.

Consider a sequence of observations. Each position in the sequence (called a sequence *slice*) is characterized by n random variables. Each slice in the sequence can be represented by an ordinary BN, which is said to be duplicated along the sequence positions. The sequential dependencies are then represented by edges between the consecutive DBNs. Hence, a DBN is defined by two components:

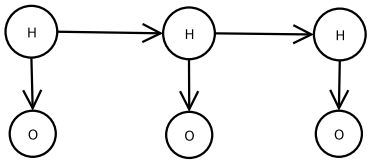

Figure 1.2: Three slices of a simple HMM in the DBN representation. Hidden nodes are labelled H, observed nodes O.

- A set of nodes that represent all random variables for a given position (also called a *slice*) in the sequence, and the edges between those nodes. This set of nodes and edges forms the BN that will be duplicated to model the sequence.
- A set of edges that connect nodes in two consecutive slices in the sequence.

A DBN's structure can be specified conveniently by specifying all the nodes and edges that belong to two consecutive slices. This set of nodes and edges is then duplicated as necessary to model a given sequence. Some examples will probably make this more clear.

Fig. 1.2 shows the DBN representation of a HMM. In the figure, three slices are shown, corresponding to a sequence of length 3. The BN describing each slice consists simply of the hidden node H, the observed node O, and an edge that connects these two nodes. Two consecutive slices are connected by an edge between the two consecutive hidden nodes H.

Note that this diagram should not be confused with the typical diagrams that are used to represent HMMs. In a BN diagram, the nodes are random variables and the edges are conditional dependencies. In a HMM diagram, the nodes are states and the edges represent possible transitions between states. In a DBN diagram of an ordinary HMM, an HMM state corresponds to a value of the hidden random variable, and a transition between states corresponds to a non-zero entry in the conditional probability distribution matrix of the DBN.

Fig. 1.3 represents a Factorial HMM [10]. In this case, there are three hidden nodes (H_1 to H_3), which are the parents of the observed node O. The value of the observed node O depends on the values of the three hidden H nodes. The BN corresponding to each slice consists of nodes H_1, H_2, H_3 and O, and the connections between the H nodes and the O node. The edges that connect the consecutive BNs are the (H_1, H_1), (H_2, H_2) and (H_3, H_3) edges.

A DBN typically models a sequence by considering a set of observed nodes whose values depend on a set of hidden nodes (that is, nodes whose values cannot be observed). The process of assigning probabilities to the possible values of the hidden nodes given the values of the

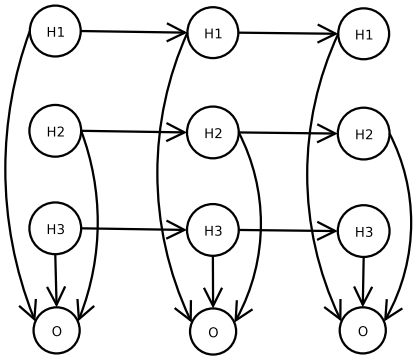

Figure 1.3: Three slices of a Factorial HMM. Hidden nodes are labelled $H1, H2, H3$, observed nodes are labelled O.

observed nodes is called *inference*. For HMMs, inference is typically done using the Forward-Backward algorithm.

Once the architecture of a DBN is defined, its parameters are typically optimized in a *learning step*. The aim is find a set of parameters for the DBN that assigns a high probability to a certain relevant set of training sequences. This is typically done using the *Expectation Maximization* (EM) algorithm, which leads to a ML or MAP point estimate of the parameters. For example, the EM algorithm applied to an HMM leads to the well known Baum-Welch algorithm.

This is of course a very rudimentary, informal and incomplete introduction to HMMs, BNs and DBNs. Luckily, there are many excellent introductory books and articles available (for example [26, 9, 14, 2, 8, 21, 22]).

1.3 Inference in DBNs

Inference in DBNs can be done in many different ways. Deterministic algorithms like *junction tree inference* and *belief propagation* (leading to the forward-backward algorithm for HMMs) are based on deterministic message-passing algorithms, while non-deterministic algorithms use various sampling methods to perform inference.

Mocapy uses a Markov Chain Monte Carlo (MCMC) technique called Gibbs sampling to perform inference [23, 11, 4], i.e. to approximate the probability distribution over the values of the hidden nodes. As far as I know, Mocapy is the only publicly available toolkit that applies Gibbs sampling to ML/MAP learning and inference in DBNs.

Sampling methods like Gibbs sampling are attractive for the following reasons:

- They are less prone to get stuck in a local minimum during parameter learning.

- They are comparatively easy to parallelize efficiently on a cluster computer with a minimum of performance loss due to sending messages between the cluster nodes.
- They allow complicated network architectures and large datasets, especially when implemented on cluster computers.
- They are flexible: it is quite easy to try out various sampling variants (for example Metropolis-Hastings), and to use non-standard node types (i.e. apart from the standard discrete and Gaussian nodes).

Tests with Mocapy show that the combination of Gibbs sampling and DBNs is quite powerful, although perhaps on the slow side if you do not have a cluster computer available and need to work with a large dataset.

How does Gibbs sampling work? G cycles (or sweeps) of Gibbs sampling are performed like this:

1. The values of the hidden nodes are initialised, for example by sampling according to $P(h_{l,n} \mid \mathrm{Pa}(h_{l,n}))$, where $h_{l,n}$ is the value of hidden node n at sequence position l, and $\mathrm{Pa}(h_{l,n})$ are the values of the parents of node n at position l. For this way of initialization, sampling should start at the root nodes, and continue towards the leaf nodes.
2. Pick a random hidden node $H_{l,n}$ (i.e. hidden node n at sequence position l).
3. Sample a value $h_{l,n}$ for $H_{l,n}$ based on the values of the nodes in its so-called *Markov Blanket*, i.e. the values of the child and parent nodes of node $H_{l,n}$:

$$h_{l,n} \sim P(h_{l,n} \mid \mathrm{Pa}(h_{l,n})) \prod_{c \in \mathrm{Ch}_{(h_{l,n})}} P(c \mid \mathrm{Pa}(c))$$

   where $\mathrm{Ch}(h_{l,n})$ is the set of children of node n at position l.
4. Go to step 2, until the sequence has been sampled G times.

Let me illustrate the concept of the Markov Blanket with an example. Suppose node C has two parents A, B and two children D, E (see Fig. 1.1). In addition, node D has F as its single parent. All nodes except C are observed (with node values a, b, d, e, f). Then, a node value c for node C will be sampled from the following distribution:

$$c \sim P(C = \cdot \mid a, b) P(e \mid C = \cdot) P(d \mid C = \cdot, f)$$

The above algorithm describes a *Random Sweep Gibbs sampler*, i.e. the nodes and sequence positions are traversed in a random way during one sampling cycle. Other traversals are also possible in principle.

In a DBN, parents and children might be present in three different slices (for example, A might be in slice $l-1$, B, D, F in slice l and E in slice $l+1$). This makes the bookkeeping in a DBN slightly more complicated than in a BN.

Typically, the first cycles of Gibbs sampling (called the *burn in period*) are discarded, because the values of the sampled nodes need some time to reach a relevant distribution. There are unfortunately no simple rules that tell you how long the burn in period needs to be. Testing convergence on an artificial dataset for which the expected parameter values are known is an accepted way to evaluate convergence. This artificial dataset can for example be generated by generating a set of sequences through sampling the DBN or by using a standard data set.

Each time a hidden node is sampled, the sampled node value is stored. When the sampling is finished, the sampled node values can be used to approximate the probability distribution of the node values of the hidden nodes.

1.4 Parameter learning in DBNs

1.4.1 Expectation Maximization

Parameter learning of a DBN with hidden nodes is done using the *Expectation Maximization* (EM) method (for a good introduction to learning in BNs, see [9, 14, 22], a very clear explanation of the EM algorithm can be found in [8]). In the E-step, the values of the hidden nodes are inferred using the current DBN parameter setting. In the subsequent M-step, the inferred values of the hidden nodes are used to update the DBN's parameters. The E- and M-step cycles are repeated until convergence (or until the user's patience is exhausted). Parameter learning by EM leads to a ML or MAP (if one uses priors on the parameters of the DBN) point estimate of the parameters.

Let's take a look at this in a bit more detail (the following discussion is largely based on [9]). In learning, we want to maximize the log likelihood (LogLik) $\mathcal{L}$ in function of the DBN's parameters θ:

$$\mathcal{L}(\theta) = \log \sum_{\mathbf{h}} P(\mathbf{h}, \mathbf{o}|\theta)$$

where $\mathbf{h} = h_1, \ldots, h_m$ is the set of hidden node values, and $\mathbf{o} = o_1, \ldots, o_n$ is the set of observed node values. The sum runs over all possible hidden node value combinations $\mathbf{h}_1, \ldots, \mathbf{h}_k$. In practice, this computation is intractable using a brute force approach. However, the problem can be simplified by making use of a lower bound $\mathcal{F}(\theta)$ of $\mathcal{L}(\theta)$. First, we introduce an arbitrary probability distribution $q(\mathbf{h})$ over $\mathbf{h}$:

$$\mathcal{L}(\theta) = \log \sum_{\mathbf{h}} q(\mathbf{h}) \frac{P(\mathbf{h}, \mathbf{o}|\theta)}{q(\mathbf{h})}$$

Because the log function is a concave function, we can apply Jensen's inequality[1] to obtain a lower bound $\mathcal{F}$ of $\mathcal{L}$. Hence:

$$\mathcal{L}(\theta) \geq \mathcal{F}(q(\mathbf{h}), \theta) = \sum_{\mathbf{h}} q(\mathbf{h}) \log \frac{P(\mathbf{h}, \mathbf{o}|\theta)}{q(\mathbf{h})}$$

$$\mathcal{F}(q(\mathbf{h}), \theta) = \sum_{\mathbf{h}} q(\mathbf{h}) \log P(\mathbf{h}, \mathbf{o}|\theta) - \sum_{\mathbf{h}} q(\mathbf{h}) \log q(\mathbf{h})$$

This expression is similar to the negative of the *free energy* in statistical mechanics, where the first term is an (average) energy term [2] and the second term is an entropy term.

In the EM procedure, $\mathcal{F}(q(\mathbf{h}), \theta)$ is first optimized with respect to $q(\mathbf{h})$ keeping θ fixed at its current value θ^{curr}, and then with respect to θ using $q(\mathbf{h})^{\text{next}}$ obtained from the E-step.

E-step:

$$q(\mathbf{h})^{\text{next}} = \text{argmax}_{q(\mathbf{h})} \left[\mathcal{F}(q(\mathbf{h}), \theta^{\text{curr}})\right]$$

M-step:

$$\theta^{\text{next}} = \text{argmax}_{\theta} \left[\mathcal{F}(q(\mathbf{h})^{\text{next}}, \theta)\right]$$

The maximum in the E-step arises for the following value of q:

$$q(\mathbf{h}) = P(\mathbf{h} \mid \mathbf{o}, \theta^{\text{curr}})$$

which is simply the posterior or smoothing distribution. In Mocapy, this distribution is approximated using Gibbs sampling.

The EM cycle is repeated until convergence or until the maximum number of cycles is reached. In the next section, I will explain how Mocapy does this in practice using Monte Carlo EM or Stochastic EM.

1.4.2 Monte Carlo EM for DBNs

From the above discussion, it is clear that parameter learning using the EM algorithm requires a method to perform inference over the possible hidden node values. If one uses a stochastic procedure to perform the E-step, a stochastic version of the EM algorithm is obtained. There are two reasons to use a stochastic E-step. First, deterministic inference might be intractable. Second, certain stochastic versions of the EM algorithm are more robust than the classic version of EM.

---

[1] For any concave function f the following is true: $E\left[\text{f}(x)\right] \leq \text{f}(E\left[x\right])$, where $E\left[\cdot\right]$ is the expectation according to $P(x)$. To apply the inequality in the above case, note that the summation in the $\mathcal{L}(\theta)$ expression is simply the expectation of $f(\mathbf{h}) = \frac{P(\mathbf{h},\mathbf{o}|\theta)}{q(\mathbf{h})}$ with respect to $q(\mathbf{h})$.

[2] The energy of a configuration $(\mathbf{h}, \mathbf{o})$ is in this case defined as $E(\mathbf{h}, \mathbf{o}) = -\log P(\mathbf{h}, \mathbf{o}|\theta)$. The first term is thus the average energy according to $q(\mathbf{h})$, and the second term is the entropy $\text{S}(\cdot)$ of the $q(\mathbf{h})$ distribution. Thus, the free energy $\text{A}(q(\mathbf{h}))$ associated with $q(\mathbf{h})$ is $\text{A}(q(\mathbf{h})) = -\mathcal{F}(q(\mathbf{h}), \theta) = \langle E(\mathbf{h}, \mathbf{o})\rangle_{q(\mathbf{h})} - \text{S}\left[q(\mathbf{h})\right]$.

In *Monte Carlo EM (MC-EM)* the hidden node values are reset to random values after each step, a set of samples is generated that is discarded during a burn-in period, and a typically large number of samples is used by the EM step [1].

In the *Stochastic EM (S-EM)* variant of MC-EM, the hidden node values are kept after each EM step, and in each E-step only one sample is generated for each hidden node [6, 24, 1]. In contrast to MC-EM, S-EM has some clear advantages over deterministic EM algorithms:

- S-EM is less dependent on starting conditions.
- The deterministic EM algorithm has a tendency to get stuck at saddle points, or insignificant local maxima. This is much less so for the S-EM algorithm.
- Because only one value needs to be sampled for each hidden node in the E-step of S-EM, it is much faster than MC-EM.

S-EM is especially suited for large datasets. For small datasets, MC-EM is a better choice.

Mocapy supports both forms of MC-EM. Tests have pointed out that S-EM is both fast and efficient if you have a lot of data (for example, I routinely use several of 100.000s slices in combination with S-EM).

I here briefly sketch the background of the MC-EM and S-EM methods. Recall the optimal distribution $q(\mathbf{h})$ in subsection 1.4.1:

$$q(\mathbf{h}) = P(\mathbf{h} \mid \mathbf{o}, \theta^{\text{curr}})$$

In Mocapy, $q(\mathbf{h})$ is approximated by sampling a set of hidden node values from $P(\mathbf{h}|\mathbf{o}, \theta^{\text{curr}})$ by Gibbs sampling. Each hidden node value is sampled conditional upon all the other node values, which due to the graph structure of a DBN, corresponds to sampling conditional upon the Markov blanket of the node. At the end of this E-step, the sampled hidden node values and the values of the observed nodes can be treated as a completed dataset in which all node values are observed. In the M-step, the completed dataset is then used to update the parameters of the model.

How is this all performed in practice? One typically starts with a DBN initiated with random values for the parameters and the hidden nodes, or starting values that somehow 'make sense'. In the E-step (which is the inference step), the values of the hidden nodes are inferred using the current DBN parameter values, i.e. a set of samples is generated by Gibbs sampling. Each time a node is sampled, the *Expected Sufficient Statistics* (ESS) for a hidden node are updated. The ESS are set of numerical values that describe the samples in such a way that the the parameters of a node's probability distribution can

be calculated[3]. The ESS of all samples are then used in the M-step to update the parameters.

For example, in the case of discrete nodes, the ESS are simply the counts of the parent/child value combinations observed in the generated samples [9]. Each time a discrete node is sampled, the values of the node and of its parents are evaluated and the relevant count is updated. In the M-step, parent/child counts are then used to update the parameters of the discrete node.

[3]During parallel execution of Mocapy, only the ESS need to be passed around, not the entire sampled sequences. This saves quite some time on communication between the CPU's in the cluster.

Chapter 2

Installation, license and other practicalities

2.1 Prerequisites

Mocapy is implemented in 100% Python, but uses a lot of compiled code behind the screens in the form of Python extension modules. In addition, Mocapy's design aims to be clear, but also efficient. A C/C++ re-implementation of Mocapy with the same features and flexibility would probably not be spectacularly faster, I think. In order to run Mocapy, you need to have these packages installed:

- Python: the Python interpreter. Most systems come with Python readily installed. If not, download Python from `www.python.org`.
- `numpy`: the Python numerical extension module. `Numpy` provides Python with an array/matrix data type and various useful operations like matrix multiplication, generating random numbers and linear algebra functionality. `Numpy` is largely compiled code, and hence can lead to spectacular speedups. Mocapy relies heavily on `numpy` functionality. Download `numpy` at `http://numpy.scipy.org/`, if it's not already installed.
- `pyMPI` and MPI: MPI (Message Passing Interface) is the de-facto standard library to implement a program on a cluster computer. It should be readily available on your cluster computer. `pyMPI` is the Python binding to MPI. `pyMPI` can be downloaded from `http://sourceforge.net/projects/pympi/` (you need at least version 2.4beta - earlier versions have memory leaks). You do not need MPI/`pyMPI` if you are planning to use Mocapy on a single CPU machine of course.
- `SciPy`: `SciPy` is a library of Python modules with a focus on scientific applications. `SciPy` contains the `Cephes` library of special functions, which are used in some modules. Get it at `http://www.scipy.org/`.

2.2 Obtaining Mocapy

Mocapy and its documentation can be downloaded from Sourceforge (`https://sourceforge.net/projects/mocapy/`) as a gzipped tar file containing the Python source code. Simply unzip and untar it in a suitable directory.

2.3 Installing Mocapy

Installing Mocapy is simple, using Python's `Distutils` mechanism:

```
python setup.py install
```

Depending on where you have Python installed, you might have to be root to do this. If you want to install Mocapy in a non-standard location, take a look at the standard `setup.py` options (see `http://www.python.org/doc/2.3.4/inst/inst.html`). Don't forget to make sure that your `PYTHONPATH` is correctly set in that case.

Mocapy will run on every platform where Python and `numpy` can be installed (which means just about any platform, including Windows). For parallel execution, you will need MPI and `pyMPI`. So far Mocapy was successfully run on a Linux laptop with a single processor, a 250 CPU Linux cluster of PC's, and a SUN Sunfire machine (which is an SMP machine) running Solaris.

2.4 Running Mocapy

A Mocapy script is a simple Python program. Hence, on a simple desktop computer one uses something like this:

```
python hmm.py > hmm.log
```

In the case of parallel execution on a cluster, the Mocapy script should be executed by the `pyMPI` executable that comes with the pyMPI module. The `pyMPI` executable is in turn executed by the `mpirun` program (or similar, depending on your MPI variant). The number of processors that need to be used is typically specified with the `-np` switch. For example:

```
mpirun -np 10 pyMPI hmm.py > hmm.log
```

Note that the number of processors should be equal to or smaller than the number of training sequences, and that it only makes sense to run Mocapy on a cluster if you are performing parameter learning using a set of sequences (i.e. at least two)!

Running a program on a cluster is highly dependent on which cluster and queuing system you are using, so the details might vary for your system.

2.5 License

Mocapy comes with the *GNU Lesser General Public License* (see section 10 for more information). Informally speaking (please refer to the license file itself for the legally binding license), you can use Mocapy in any commercial or open source program, as long as you make the full Mocapy source code (including the License file) publicly available if you alter the source code in any way.

2.6 Other DBN packages

Is Mocapy the best available tool for you? That depends entirely on what you want to do (ML, MAP or Bayesian parameter learning, structure learning,...), how much data you have, the available computer power, etc. As far as I know, Mocapy is the only publically available toolkit that can handle directional statistics, using Von Mises-Fisher and Kent distributions. Before you decide to use Mocapy, you might want to take a look at the following packages for example:

- Kevin Murphy's Bayes Net Toolbox for Matlab (BNT, `http://www.cs.ubc.ca/~murphyk/Software/BNT/bnt.html`): this widely used toolkit supports HMMs and general DBNs. Unlike Mocapy, it mainly focuses on deterministic methods like junction tree inference. Much of the overall design of Mocapy was inspired by this toolkit.
- The Graphical Models Toolkit (GMTK, `http://ssli.ee.washington.edu/~bilmes/gmtk/`): Specially focused on DBNs, but only binaries are available.
- The Open Source Probabilistic Networks Library (OpenPNL, `https://sourceforge.net/projects/openpnl/`): Implemented in C++. Only limited support for DBNs last time I checked.

Chapter 3

Background and Features

3.1 Parallelization

Learning a DBN's parameters using MC-EM can be done in parallel on a cluster computer. Learning is typically done from a set of sequences. In Mocapy, the set of sequences is divided over the available processors. Each processor then does the E-step for the sequences assigned to it, ie. infers the hidden node values by sampling. The ESS are calculated from the sampled sequences on each processor, and sent to the master processor where the M-step is done. The master processor calculates the new parameters from the received ESS, and sends them back to the slave processors.

If the sequence are of widely different lengths and they are distributed ad random, the parallelisation might become inefficient due to the fact that all processors need to wait until inference has been done for all sequences. This is called a *load balancing problem*, which is a classic problem for parallelized programs. However, if the sequence data can be divided equally over the available processors, the speedup is roughly linear with the number of processors!

3.2 Performance

Mocapy is designed to deal with large datasets. I routinely use datasets with 250.000 slices (corresponding to 1600 protein sequences), in combination with a HMM with an observed Kent node (ie. each slice consists of a discrete value and a 3D real vector). It takes less than 10h to do 250 S-EM steps on 20 processors (including 50 burn-in steps). Convergence was reached after about 50 EM steps (including 50 burn-in steps, i.e. after about 3h 30min). Speedup by parallelization is close to linear in the number of processors (i.e. two processors are twice as fast as one processor), if the sequences can be distributed uniformly over the processors. If you have smaller datasets (or a lot of patience), it is of course perfectly possible to run Mocapy on a desktop computer.

3.3 DBN architectures

The architecture of the DBN, i.e. the number and type of the nodes and the connections between the nodes, can be specified by the user. In principle, there are no limits with respect to the number of nodes, edges between nodes or data size, provided that you have enough memory and patience. Gaussian, Kent, Von Mises-Fisher and Dirichlet nodes need to have exactly one discrete parent node, and they need to be in the same slice as their parent. Partly observed nodes (i.e. some values are missing in the sequence for a particular node) are supported.

3.4 Node types

Currently, Mocapy has five different node types, each based on a specific probability distribution: discrete, Gaussian, Dirichlet, Von Mises-Fisher and Kent. In the future, more node types might be added. Adding a new node simply means implementing a new `Node` subclass with a set of suitable methods.

Discrete nodes deal with integer/labelled data, Gaussian nodes with real value data (of arbitrary dimension), Dirichlet nodes with probabilities (a set of strictly positive real numbers that sum to one) and VMF and Kent nodes with angular/directional data.

3.4.1 Gaussian and Discrete nodes

Discrete and (multidimensional) Gaussian distributions are standard, so they'll not be discussed at length here. The technicalities of parameter estimation for Gaussian nodes can be found in [22]. Mocapy supports Gaussian distributions with unconstrained, spherical and diagonal covariance matrices.

3.4.2 VMF nodes

The Von Mises-Fisher (VMF) distribution is much less standard [5]. It can be used to model a set of points on the circle (S^1 in $\mathbb{R}^2$), the sphere (S^2 in $\mathbb{R}^3$) or a higher dimensional sphere. The S^1 case corresponds to the better known Von Mises distribution on the circle. To my knowledge, Mocapy is the only publicly available toolkit that allows building a probabilistic model of sequences of directional data. I believe directional distributions like the VMF distribution will come to play a major role in Bioinformatics in the near future, especially in the modelling of angles and dihedral angles in molecules. For an excellent overview of directional statistics, see the book of Mardia & Jupp [18].

The p-dimensional VMF distribution with mean direction μ and concentration parameter κ has the following probability density function:

$$\text{VMF}_{p,\mu,\kappa}(x) = C(p,\kappa)e^{\left(\kappa\mu^t x\right)}$$

The normalizing term $C(p,\kappa)$ is defined as:

$$C(p,\kappa) = \frac{\kappa^{2p-1}}{(2\pi)^{p/2} I_{p/2-1}(\kappa)}$$

where $I_{p/2-1}$ is the modified Bessel function of the second type and order $p/2-1$. This function is implemented by the `Cephes` function iv, as present in the `SciPy` Python toolkit. The variable p is the dimension of the unit vector that describes the direction (i.e. a vector on the unit sphere). For example, a single angle is described by a 2D unit vector that lies on the unit circle, and hence $p = 2$ in this case. In the latter case, the distribution is the classic Von Mises distribution on the circle S^1.

As mentioned before, the parameter κ is called the concentration parameter: the larger κ is, the more the density will be concentrated around the mean direction μ. Basically, the VMF distribution is the spherical equivalent of a (multidimensional) Gaussian distribution with spherical covariance. Hence, the equal probability contours of the VMF density function on the sphere are circular.

3.4.3 Kent nodes

The Kent distribution [15, 17, 25, 16, 3, 13], also known as the 5-parameter Fisher-Bingham distribution, is another directional distribution, in this case on the sphere S^2 in $\mathbb{R}^3$. It is a member of a larger class of *N*-dimensional distributions called the Fisher-Bingham class of distributions. For this class of distributions, parameter estimation becomes difficult if the dimension becomes larger than 2 (which is the dimension of the Kent distribution[1]). The density function of the Kent distribution is:

$$K_{\kappa,\beta,\gamma_1,\gamma_2,\gamma_3}(x) = C(\kappa,\beta)\exp\left\{\kappa x\cdot\gamma_1 + \beta\left[(x\cdot\gamma_2)^2 - (x\cdot\gamma_3)^2\right]\right\}$$

The various parameters can be interpreted as follows:

- κ: a concentration parameter, like κ in the case of the VMF distribution. The higher κ, the more concentrated the density becomes.
- β: determines how elliptical the equal probability contours of the distribution will be. The larger β (with condition $\kappa > 2\beta$), the more elliptical. If $\beta = 0$, the Kent distribution becomes the Von Mises distribution (i.e. the VMF distribution on the 3D sphere).

[1] Note that - in simple words - S^2 is the surface of the three-dimensional ball. This surface is of course two-dimensional, but a point on the sphere is often specified by a three-dimenional unit vector. Equivalently, a point on S^2 can be specified by a pair of polar angles.

- γ_1: the mean vector. The three γ vectors are 3D unit vectors with end points on S^2.
- γ_2: the main axis of the elliptical equal probability contours.
- γ_3: the secondary axis of the elliptical equal probability contours.

The normalizing factor $C(\kappa, \beta)$ is given by:

$$C(\kappa, \beta) = \frac{\sqrt{(\kappa - 2\beta)(\kappa + 2\beta)}}{2\pi e^{\kappa}}$$

The advantage of the Kent distribution over the VMF distribution on S^2 is that the equal probability contours of the density are not restricted to be circular: they can be elliptical as well. The Kent distribution is equivalent to a Gaussian distribution with unrestricted covariance. Hence, for 3D directional data (points on S^2) the Kent distribution is richer than the corresponding VMF distribution, i.e. it might be more suited if the data contains non-circular clusters.

3.4.4 Dirichlet nodes

The Dirichlet distribution is a probability distribution over probability parameters. It is mostly used as a prior for a set of probability parameters. However, it can of course also be used to model probability parameters themselves, like a set of observed amino acid distributions (i.e. a set of 20 positive real numbers whose sum is one). For example, recently, a HMM model with observed Dirichlet nodes (which was called an Hidden Markov-Dirichlet Multinomial or HMDM) was used to model structurally similar DNA motifs that are encoded by widely different sequences [28]. Again, Mocapy is as far as I know the only publicly available toolkit that allows this kind of modelling.

The Dirichlet node was implemented using two sources. Sampling from a Dirichlet distribution is described in Luc Devroye's wonderful book '*Non-Uniform Random Variate Generation*', which is entirely available online[2]. The estimation of the Dirichlet distribution's parameters is described in [20].

3.5 Inference

Inference means calculating the probabilities associated with the values of the hidden nodes, given a set of values for the observed nodes. Mocapy provides two approaches to inference: approximative methods based on Gibbs sampling and deterministic methods based on treating the DBN as a HMM.

[2] http://www-cgrl.cs.mcgill.ca/~luc/rnbookindex.html

Calculating the probability distribution over the values of the hidden nodes (i.e. the smoothing or posterior distribution) and calculating the most probably combination of the hidden node values (i.e. the Viterbi path) can be done in a deterministic way by treating the DBN as a HMM, and applying classic HMM algorithms.

Turning the DBN into a HMM can lead to intractable calculation when you have a lot of hidden nodes that can adopt a large number of values. If you have 10 discrete hidden nodes that each can adopt 4 different values, this would lead to a HMM with one hidden node that can adopt $4^{10} = 1048576$ states. In this case, one can approximate the posterior distribution and the Viterbi path using approximative methods based on Gibbs sampling.

3.5.1 Deterministic inference

3.5.1.1 Posterior distribution

The posterior or smoothing distribution, $P(h_{l,n} \mid \mathbf{o})$ for every sequence position l and hidden node n, can be calculated by treating the DBN as a HMM and applying the Forward-Backward algorithm [2, 8]. Scaling of the probabilities is done as described in [2], appendix D.

3.5.1.2 Viterbi path

The Viterbi path is the sequence of hidden node values that best explains the values of the observed nodes (i.e. that leads to the highest likelihood value). In other words it means calculating the following:

$$\mathbf{h}^v = \arg\max_{\mathbf{h}} [P(\mathbf{h} \mid \mathbf{o})]$$

where $\mathbf{o} = o_1, \ldots, o_n$ are the values of the observed nodes and $\mathbf{h}^v = h_1^v, \ldots, h_m^v$ is the Viterbi path. The Viterbi path plays a crucial role in many Bioinformatics applications [2, 8].

The Viterbi path can be calculated by treating the DBN as a HMM and applying a dynamic programming algorithm [2, 8]. Mocapy's Viterbi path calculation uses Log probabilities to avoid underflows.

3.5.2 MCMC inference

3.5.2.1 Posterior distribution

For large DBNs, the Forward-Backward algorithm can become intractable. In that case one can approximate the posterior distribution by generating a large set of samples (i.e. by Gibbs sampling) and examining the values of the hidden nodes. The quality of the approximation will of course depend on the complexity of your model and the number of samples used.

3.5.2.2 Viterbi path

As is the case for the Forward-Backward algorithm, the deterministic calculation of the Viterbi path can become intractable for large DBNs. Again, one can use an approximative algorithm based on Gibbs sampling.

In principle, one could generate a large set of samples and simply pick the one with the highest LogLik. In practice, the number of possible hidden node combinations is vast and this approach becomes infeasible. A good solution to this problem is to generate a set of samples (say 100) and to construct a sequence with a high LogLik by tying together fragments from the samples.

The algorithm to approximate a Viterbi path using a set of sampled sequences uses a Dynamic Programming approach to combine the best hidden node values from a set of sampled sequences (see Algorithm 1). The algorithm is conceptually similar to a Viterbi path approximation algorithm [12] which uses a set of samples generated by *Particle Filtering*, an online stochastic inference method. The latter algorithm was applied to an HMM with continuous hidden and observed states.

The algorithm can be used in an iterative manner. In the first run, an initial Viterbi path approximation is calculated from a set of samples. In the next steps, new samples are generated but this time the Viterbi path approximation that was calculated in the previous step is added to the set of samples. In this way, one gradually improves the approximation. The procedure can be stopped when the LogLik seems to have converged. This procedure can of course get stuck in a local maximum.

3.6 Parameter learning using Monte Carlo-EM

Parameter learning is done using MC-EM, making use of Gibbs sampling for the E-step (i.e. the inference step). When new sampling methods are added, these will also become available for the MC-EM procedure.

3.7 Priors

In Mocapy, discrete nodes support the use of priors. Currently, three different priors are supported (for a discussion of these priors in the context of biological sequence analysis, see [8]).

First, let's consider how an entry in a CPD of a discrete node is calculated. A CPD is a matrix filled with values of the type $P(N = n \mid \pi_N = \mathbf{p})$, which corresponds to the chance of observing value n for node N when N's parent nodes π_N adopt values $\mathbf{p} = p_1, \ldots, p_p$. If all sequence

Algorithm 1 Mocapy's Viterbi path approximation algorithm. $samples[m, l]$ corresponds to slice l in sample m. $P(samples[m, l]|samples[n, l-1])$ is the likelihood of all the node values in slice l, where the values for the nodes are taken from position l of sample m, and the nodes' parent values in the previous slice are taken from position $l-1$ of sample n. Sequence indices run from 0 to $L-1$. If a previous approximation of the Viterbi path is available, it can be added to the initial list of samples. The procedure will then come up with an improved variant of it or produce the same path.

```
S=number of samples
L=sequence length
V=number of scalars in a sequence slice
samples=sequence (S × L × V matrix)
score=matrix(S × L)
path=matrix(S × L)
# Start of dynamic programming
# 1. Initialization
for m = 0 to S − 1:
    score[m, 0]=log P(samples[m, 0])
# 2. Recursion
for l = 0 to L − 1:
    for m = 0 to S − 1:
        p=matrix(S)
        for n = 0 to S − 1:
            p[n]=log P(samples[m, l] | samples[n, l − 1])
        index=argmax(p)
        score[m, l]=score[m, l − 1]+p[index]
        path[m, l]=index
# 3. Termination
index=argmax(score[L − 1])
viterbi=matrix(L × V)
viterbi[L − 1]=samples[index, L − 1]
# 4. Backtracking
for l = L − 2 to 0:
    index=path[index, l]
    viterbi[l]=samples[index, l]
report viterbi
```

node values are known, this is simply calculated as follows:

$$P(N = n \mid \pi_N = \mathbf{p}) = \frac{\text{counts}(N = n, \pi_N = \mathbf{p})}{\sum_{\mathbf{m}} \text{counts}(N = m, \pi_N = \mathbf{p})}$$

where $\text{counts}(N = n, \pi_n = \mathbf{p})$ is the number of times the node value n is observed for node N together with parent values $\mathbf{p}$ in the sequences. The sum runs over all possible node values $\mathbf{m}$ for N. Let's now consider some classic prior types:

- Pseudo counts: the CPD entries are calculated as follows:

  $$P(N = n \mid \pi_N = \mathbf{p}) = \frac{\text{counts}(N = n, \pi_N = \mathbf{p}) + C}{\sum_{\mathbf{m}} [\text{counts}(N = m, \pi_N = \mathbf{p}) + C]}$$

  where C is the pseudo count.

- Proportional to a background distribution $Q(N = n)$. In this case the CPD is calculated as follows:

  $$P(N = n \mid \pi_N = \mathbf{p}) = \frac{\text{counts}(N = n, \pi_N = \mathbf{p}) + AQ(N = n)}{\sum_{\mathbf{m}} \text{counts}(N = m, \pi_N = \mathbf{p}) + A}$$

  where $Q(N)$ is a given background distribution of N and A is the weight put on the pseudo counts as compared to the observed counts. If A is large compared to the real counts, $P(N = n \mid \pi_N = \mathbf{p})$ will be close to $Q(N = n)$, while if A is comparatively small, the effect of the prior becomes negligible. For protein alignments for example $A = 20$ is usual.

- Mixture of Dirichlet distributions: the CPD is calculated from a Dirichlet mixture model according to Sjölander *et al.* [27].

3.8 Structure learning is absent!

Structure learning is absent in Mocapy. I also have no plans to implement it, since I do not need it, at least not at the moment. Anybody who wants to add this functionality to Mocapy is welcome to contact me.

Chapter 4

Using Mocapy

4.1 Preliminaries

The first step is to import the Mocapy module:

```
from Mocapy import *
```

Obviously, sampling depends to a great extend on the generation of random numbers. By using a specific seed for the random number generating that Mocapy is using, one can make the process reproducible, if that is desirable. Mocapy uses the random number generators in the `numpy` module `random` and the standard Python module `random`. These random number generators can be seeded (with two numbers) to reproduce a Mocapy run exactly:

```
mocapy_seed(a,b)
```

Note that importing Mocapy automatically import the `pyMPI` module as well, which can be very useful. For example, the `pyMPI` module can be used for controlling the output, or to load the data for each separate processor. The following attributes and methods are particularly useful:

```
# Obtain the rank of the processor
rank=mpi.rank
# Scatter a sequence of objects over all processors
local_sequence_list=mpi.scatter(sequence_list)
# Find out how many processor are being used
nr=mpi.size
# Print out the LogLik
if mpi.rank==0:
    print LL
```

The examples in the `example` directory illustrate the use of the `mpi` module. More information can of course be found in the pyMPI documentation.

4.2 Representation of the sequence data

Mocapy stores data in `numpy` arrays. Each sequence corresponds to one array. Another array, the `mismask` array, indicates whether the value of a certain node at a certain position in the sequence is observed or not. A zero in the mismask array indicates the node value is observed, while a 1 indicates the node value is not observed.

This is most easily explained with a simple example. Suppose the DBN has two nodes, a 2-dim Gaussian child node and a discrete parent node. Suppose the sequence is 100 units long. The following code creates suitable `data` and `mismask` arrays:

```
data=zeros((100,3), 'd')
mismask=zeros((100,2), 'd')
mismask[:,0]=1
```

Dimensions 1 and 2 in the data array will hold the real values that are associated with the 2-dim Gaussian node, while dimension zero holds the integer value of the discrete node. Of course, it is your responsibility to fill the data array with meaningful values somehow. Note that the order of the nodes in the DBN definition determines the order of the values in the data array. Parents need to come first, followed by their children.

Note that the mismask does not need to be an array. It needs to be a list-like object that simply returns 1 or 0 for each node/sequence position tuple (for example, index `[0, 50]` means the value of the first node at sequence position 50). Here's an example of such a list-like `mismask` object:

```
class IsMissing:
    def __getitem__(self, index):
        # Get sequence position an node rank
        l,n=index
        if n==0:
            return 1
        else:
            return 0
```

This class flags node 0 as missing and node 1 as observed for all sequence positions, which is typical for a classical HMM with one discrete hidden node.

4.3 Creating the DBN

4.3.1 Creating the nodes

The first step in the definition of a DBN is the creation of the appropriate nodes. For example, the following code creates a discrete node with node size 4 (i.e. it can adopt values 0,1,2 or 3):

```
dnode=DiscreteNode(node_size=4)
```

The node size is the number of different values the node can adopt, starting from 0. Since no initial values are specified for the Conditional Probability Density (CPD), they are initiated to random values (while avoiding transition probabilities that are 0). The CPD is a matrix that holds the probabilities of each combination of node and parent values. For example, a discrete node of size 2 with two parents of sizes 3 and 4 respectively will have a $3 \times 4 \times 2$ matrix as its CPD. The sum of the matrix values (i.e. of the probabilities of all the parent and node value combinations) needs to be 1. For more explanation on the expected order of the parents, see next paragraph.

Of course, one can initiate a discrete parent with specific CPD values. For the above described node a possible initialisation is:

```
cpd=zeros((3,4,2), 'd')+1.0
cpd=normalize_cpd(cpd)
dnode=DiscreteNode(node_size=2, user_cpd=cpd)
```

Similarly, Gaussian, VMF, Kent and Dirichlet nodes are created as follows:

```
gnode=GaussianNode(dim=2, node_size=4)
vnode=VMFNode(dim=2, node_size=4)
knode=KentNode(node_size=4)
dnode=DirichletNode(dim=2, node_size=4)
```

The `dim` parameter specifies the dimension of the real valued output vector. For the Kent distribution, dim $= 3$. Again, the parameters of the Gaussian, VMF, Kent and Dirichlet distributions are by default initiated to random values. The node size specifies the number of distributions that are generating the observations. It is expected that these nodes have exactly one discrete parent with the same node size.

Initialization with specific values (in this example random again) for the Gaussian node is done like this:

```
from RndCov import RndCov
m=random((3,2))
c=array((rnd_cov(2), rnd_cov(2), rnd_cov(2))
gnode=GaussianNode(dim=2, node_size=3, user_means=m,
    user_covs=c)
```

The `rnd_cov` function generates random covariance matrices for a given dimension. For the Gaussian node, there are three possible choices for the covariance matrix: full, spherical or diagonal. This is specified by the `cov_type` parameter, which should be `'FULL'`, `'SPHE'` or `'DIAG'`.

In addition, one can also use one tied covariance matrix, i.e. all the Gaussian distributions share the same value for this matrix. This is specified by setting the argument `cov_tied=1`.

Initialization of the VMF node with user defined parameters is similar:

```
m=random((3,2))
k=random(3)+1
vnode=VMFNode(dim=2, node_size=3, user_mus=m,
    user_kappas=k)
```

When the Kent node is created with random parameters, the maximum values for the κ and β parameters can be specified. The default values are 20 and 2, respectively. Keep in mind that $\kappa > 2\beta$. Note that the dimension does not need to be specified for the Kent node because the Kent distribution is a distribution on the 3D sphere (i.e. dim $= 3$). For example:

```
knode=KentNode(node_size=3, max_kappa=40, max_beta=4)
```

Specific values for κ, β and the $\gamma_1, \gamma_2, \gamma_3$ vectors can also be given of course:

```
# Kappas between 10 and 20
k=(1+random(3))*10
# Betas between 1 and 2
b=1+random(3)
# Average direction
e1=array((1,0,0), 'd')
# Primary axis
e2=array((0,1,0), 'd')
# Secondary axis
e3=array((0,0,1), 'd')
# Put e1,e2,e3 into one array
a=array((e1,e2,e3))
knode=KentNode(node_size=3, user_kappas=k, user_betas=b,
    user_axes=a)
```

Finally, the Dirichlet node can be initialized with user defined alpha values:

```
a=random((4,2))
DirichletNode(dim=2, node_size=4, user_alphas=a)
```

One can also generate random alpha values between a given maximum/minimum pair:

```
DirichletNode(dim=2, node_size=4, alpha_min=1.0,
    alpha_max=5.0)
```

Sometimes it is useful to fix a node's parameters during learning. This can be done by setting the `node.fixed` attribute to 1. Setting the attribute to 0 will allow the parameters to be updated again in the following EM steps.

4.3.2 Defining the DBN architecture

After creating the necessary node objects, the next step is to specify the DBN architecture. One needs to the following things:

- Define the nodes in two consecutive slices. These nodes are arguments to the initialisation of the `DBN` class.
- Specify the connections between the nodes, both inside the slices and between the two slices. This is done using the `add_intra` and `add_inter` methods of the `DBN` object.
- Finally, initialize the DBN by calling the `construct` method.

The following example is typical for a HMM:

```
start_nodes=[h0, o0]
end_nodes=[h1, o1]
mdbn=DBN(start_nodes, end_nodes)
mdbn.add_intra(0, 1)
mdbn.add_inter(0, 0)
mdbn.construct()
```

The arguments to the `add_intra` and `add_inter` methods are indices that refer to the position of the nodes in the lists of start and end nodes. Note that the nodes in these two lists need to be in topological order! Parents should come first, followed by their children later in the list.

In the above HMM example, the observed nodes are *untied*, i.e. the observed node that models the observations at $l = 0$ is different from the observed node that models those at $l > 0$. It is possible to use *tied* nodes as well, by using the same observed node `ot` for all positions l:

```
start_nodes=[h0, ot]
end_nodes=[h1, ot]
mdbn=DBN(start_nodes, end_nodes)
mdbn.add_intra(0, 1)
mdbn.add_inter(0, 0)
mdbn.construct()
```

Note that node `ot` is connected to both `h0` and `h1`. It is not possible to tie `h0` and `h1`, because they have different conditional probability distributions (`h0` has no parents, while `h1` has one discrete hidden parent).

4.4 Large DBNs

If you have a lot of nodes, referring to them by their indices gets a bit complicated. Therefore, you can give your nodes an optional name, and use the node names instead of the indices to define the architecture of the DBN.

For example:

```
node0=DiscreteNode(node_size=5, name='Hidden0')
node1=DiscreteNode(node_size=5, name="Hidden1')
node2=DiscreteNode(node_size=20, name='Observed')
start_nodes=[hidden0, observed]
end_nodes=[hidden1, observed]
dbn=DBN(start_nodes, end_nodes)
dbn.add_inter('hidden0', 'hidden1')
dbn.add_intra('hidden1', 'observed')
dbn.construct()
```

In addition, creating the training data might be difficult if you have a lot of nodes. In that case, you can use the `SequenceConstructor` class.

4.5 Sampling a sequence from a DBN

Once a DBN is created, it is easy to sample a sequence using the `sample_sequence` method:

```
seq, ll=dbn.sample_sequence(100)
```

The method returns a sequence (here of length 100) and the LogLik. This method is of course ideal to create a test test for the learning procedure.

Like all data (see next section), the returned sequence is a `numpy` array. It's dimensions are `(output_size, sequence_length)`. The output size is the number of numbers that fill in the values of all nodes. For example, the output size for a 2D Gaussian node and a discrete node is $2 + 1 = 3$.

4.6 LogLik of a fully observed sequence

When all nodes are observed, calculation of the LogLik is of course trivial. In this case, the calculation can be done using a DBN object:

```
ll=dbn.calc_ll(seq)
```

The returned LogLik is the LogLik divided by the number of sequence slices (i.e. $\mathcal{L}^{\mathcal{M}}$).

4.7 Calculation of the posterior distribution

4.7.1 Approximative method

Approximative calculation of the posterior distribution is done using an `InfEngineMCMC` object. This object is created using four arguments:

- A `DBN` object.
- A sampler object. This object takes care of sampling the hidden nodes in the DBN. Currently, this needs to be a `GibbsRandom` object.
- A sequence object, i.e. a `numpy` array containing the data.
- A mismask object, which indicates which nodes are hidden where in the sequence.

The `InfEngine` objects is created as follows:

```
sampler=GibbsRandom(dbn)
inf_engine=InfEngineMCMC(dbn, sampler, seq, mismask)
```

Sampling in Mocapy is simply done by generating a set of sequences where the hidden nodes have been sampled using Gibbs sampling. The probability of each hidden node value given the observed node values can then be calculated easily from the set of generated sequences. The likelihood of a sequence can be calculated in a similar way from the likelihoods of the generated sequences.

How is this all done? First, one generates a sample generator object form an `InfEngine` object:

```
sample_gen=inf_engine.get_sample_generator(burn_in_steps=50,
    init_random=1)
```

The hidden node values are initialised to random values, and 50 burn-in steps are done. After that, the sample generator object can be used to generate sample sequences. This is done using the object's `next` method[1], which performs a single Gibbs sampling sweep over the sequence and returns the result (note that it re-uses the hidden node values of the previous step to start with, and no burn-in is done). The

[1]The sample generator object is a Python generator object.

generator also return the LogLik of the generated sequence, and the number of slices in the sequence.

The following example generates 1000 sequences sampled from the smoothing distribution (with the hidden node values filled in) and calculates $\mathcal{L}^{\mathcal{M}}(\theta)$, the average LogLik per slice:

```
for i in range(0, 1000):
    seq, ll, nr_slices=sample_gen.next()
    total_ll+=ll
    total_nr_slices+=nr_slices
print 'LL=', total_ll/total_nr_slices
```

4.7.2 Deterministic method

Calculating the posterior distribution using the Forward-Backward algorithm can be done using the `InfEngineHMM` class. Arguments to `InfEngineHMM` are a DBN object, a sequence and the indices of the hidden nodes[2]:

```
hidden_node_indices=[0,1]
ie=InfEngineHMM(dbn, seq, hidden_node_indices)
```

The `get_smoothed` method returns the LogLik of the sequence, an array gamma containing the probabilities all hidden node value combinations for each sequence, and the corresponding slices.
For example:

```
gamma, loglik, slices=inf_engine.get_smoothed()
```

The slice at sequence position 0, with the first hidden node value combination:

```
slice=slices[0,0]
```

The slice at sequence position 0, with the second hidden node value combination:

```
slice=slices[0,1]
```

The probability $P(h_1 \mid \mathbf{o})$ of the above slice:

```
prob=gamma[0,1]
```

[2]This class does not make use of a mismask, so you can't have nodes that are sometimes observed and sometimes not in this case.

4.8 Calculating the Viterbi path

4.8.1 Approximative

Mocapy approximates the Viterbi path using a set of sampled sequences. This is done by applying a dynamic programming algorithm that strings together pieces of the sampled sequences so as to find the sequence with the highest likelihood.

The path is calculated by the `InfEngineMCMC` class. First, one creates a Viterbi path generator object:

```
# Infer the hidden nodes
mcmc=GibbsRandom(dbn)
inf=InfEngineMCMC(dbn, mcmc, seq, mismask)
viterbi_gen=inf.get_viterbi_generator(mcmc_steps=50,
    burn_in_steps=50, init_random=1)
```

Burn-in and random initialisation are done once initially as explained in the previous section. The first time the `next` method of the Viterbi path generator is called, it generates 50 sequence samples, and uses these to approximate the Viterbi path. Subsequent calls of `next` again sample 50 sequences. But this time, the previously generated Viterbi path is added to this set before a new Viterbi path approximation is calculated. In other words, after the initial `next` call, subsequent calls try to improve the initially generated Viterbi path approximation:

```
for i in range(0, 1000):
    seq, ll, nr_slices=viterbi_gen.next()
    print 'LL=', ll/nr_slices
```

The sequence of `next` calls is typically stopped when the LogLik has converged. The more sequences are sampled and the higher the number of `next` calls, the more precise the Viterbi path will be. Keep in mind that it is in principle possible to get stuck in a local LogLik maximum, so you might want to retry the procedure with different seed values (using `mocapy_seed`) and select the best Viterbi path. The complexity of the procedure to calculate the Viterbi path is O^2, so the method can become slow if you use many sequences and `next` calls.

4.8.2 Deterministic

Deterministic calculation of the Viterbi path can be done using the `InfEngineHMM` class:

```
hidden_node_indices=[0,1]
ie=InfEngineHMM(dbn, seq, hidden_node_indices)
viterbi_path, ll=ie.get_viterbi()
```

4.9 Parameter learning

4.9.1 S-EM and MC-EM

Parameter learning is done using the `EMEngine` class. The E- and M-step are done by the `do_E_step` and `do_M_step`, respectively. The average LogLik per slice is returned by the `get_loglik` method.

The number of steps of the sampling and burn in procedures are specified by the `nr_steps` and `burn_in` arguments of the `do_E_step` method. Other arguments to `EMEngine` are the DBN object whose parameters will be updated and the object that will actually perform the sampling steps. Currently, Mocapy only implements Gibbs sampling in the `GibbsRandom` class, but more sampling method will be added. In addition, it is also possible to attach weights to the sequences using the `weight_list` argument: this should be a list of weights (from 0.0 to 1.0) for each sequence.

The following code shows the MC-EM procedure (100 EM steps) on a single processor. Each E-step initializes the hidden node values to random values, performs 10 burn-in steps and generates 10 sequence samples for each sequence. The M-step uses the latter 10 samples to update the parameters.

```
mcmc=GibbsRandom(model_dbn)
em=EMEngine(dbn, mcmc, seq_list, mismask_list)
for i in range(0, 100):
    em.do_E_step(mcmc_steps=50, burn_in_steps=50,
        init_random=1)
    ll=em.get_loglik()
    em.do_M_step()
    print 'LL=', ll
```

The next code snippet shows an S-EM procedure. At the first iteration, the hidden node values are initialised to random values, and 50 burn-in steps are done. After that, the hidden node values are reused between E-steps. Each E-step generates a single sample for each sequence, which is then used in the M-step.

```
mcmc=GibbsRandom(model_dbn)
em=EMEngine(dbn, mcmc, seq_list, mismask_list)
for i in range(0, 100):
    if i==0:
      em.do_E_step(mcmc_steps=1, burn_in_steps=50,
          init_random=1)
    else:
        em.do_E_step(mcmc_steps=1, burn_in_steps=0,
            init_random=0)
    ll=em.get_loglik()
    em.do_M_step()
```

```
        print 'LL=', ll
```

Other approaches are possible. After a certain amount of iterations of MC-EM, when the parameters are getting closer to their optimal values, one can skip the burn-in steps and the random initialisation of the hidden nodes. It is of course also possible to turn off the random initialisation while still having a burn in period, which can give the values of the hidden nodes the chance to adjust to the new parameters.

In the case of multiple processors, the sequences and their mismasks should be scattered over the available processors, for example using the `scatter` method of the `mpi` module. Here's an MC-EM example:

```
local_seq_list=mpi.scatter(seq_list)
local_mismask_list=mpi.scatter(mismask_list)
mcmc=GibbsRandom(model_dbn)
em=EMEngine(dbn, mcmc, local_seq_list, local_mismask_list)
for i in range(0, 100):
    em.do_E_step(mcmc_steps=10, burn_in_steps=10,
        init_random=1)
    ll=em.get_loglik()
    em.do_M_step()
    # Print ll at main node
    if mpi.rank==0:
        print 'LL=', ll
```

Of course, you can also implement your own way to distribute the sequences over the processors, i.e. taking the lengths of the sequences into account in order to get a good load balance over the processors. The sequences can of course easily be uploaded on each cluster node separately to minimize memory use:

```
filename='sequence_set_nr_%i.dat' % mpi.rank
local_seq_list=load_sequences(filename)
```

It's of course up to you to implement your own custom `load_sequences` function.

4.9.2 Using Priors

Mocapy supports three different prior types for discrete nodes: pseudo counts (`PseudoCountPrior` class), proportional to a given background distribution (`BackgroundProportionalPrior` class) and using a Dirichlet mixture (`DirichletMixturePrior` class).

Creating `PseudoCountPrior` and `BackgroundProportionalPrior` objects is done as follows (for a `DiscreteNode` with `node_size=2` that has one parent with `node_size=3`):

```
# Pseudo counts
counts=array((10, 10, 10), 'd')
p1=PseudoCountPrior(counts)
# Background proportional
backgrounds=array(((0.1, 0.9),(0.5, 0.5),(0.1, 0.9)), 'd')
weights=array((20, 20, 20), 'd')
p2=BackgroundProportionalPrior(backgrounds, weights)
```

The following code creates a mixture of Dirichlet distributions (with random parameters) that consists of 9 components (again for a `DiscreteNode` with `node_size=2` - the number of parents does not matter here, since the same mixture is applied regardless of the parents[3]):

```
# Alpha parameters (avoid ai==0)
a=random((9,2))+0.05
# Mixture weights
q=random(9)+0.05
# Normalize q
q=q/sum()
# Create object
p=DirichletMixturePrior(a, q)
```

Sjölander *et al.* [27] constructed a set of Dirichlet mixtures that are appropriate for amino acid distributions (i.e. with 20 probability parameters). These mixtures are publicly available at `http://www.cse.ucsc.edu/research/compbio/dirichlets/`. The mixtures files can be read in with the `read_mixture_params` function from the `DiscretePriors` module:

```
a, q=read_mixture_params('uprior.9comp')
p=DirichletMixturePrior(a, q)
```

For all priors the use is the same. The prior object is simply passed to the `DiscreteNode` object on initialization using the `prior` argument:

```
dnode=DiscreteNode(node_size=2, prior=p)
```

4.10 DBN Persistence

After an expensive learning step, one probably wants to save the trained DBN for future use. It's quite easy to save a trained DBN by making use of Python's built-in persistency features.

Saving a trained DBN object can be done as follows:

```
dbn.save('dbn.pickle')
```

[3] In principle, one could use a different mixture for every parent combination.

Loading the DBN again is done with the `load_dbn` function:

```
dbn=load_dbn('dbn.pickle')
```

You can for example save a copy of the `DBN` object after every EM-step, and re-use a saved `DBN` object when something goes wrong during learning (i.e. a power failure).

Saving and loading DBNs is done using the functionality of the `pickle` standard Python module. You can use this mechanism to save and retrieve any Mocapy object if needed.

Chapter 5

Examples

5.1 Learning

5.1.1 HMM with discrete output node

This example shows how a simple HMM with discrete output is implemented in Mocapy. The program first creates a DBN object with random parameters and samples a set of sequences from it for use as artificial training data. Then, a second DBN object with random parameters is trained using these generated sequences. During learning, the LogLik values are printed.

```
from numpy import zeros
from Mocapy import *

# Seed random number generators for reproducibility
mocapy_seed(5, 6, 7)

# Number of training sequences
N=20
# Sequence lengths
T=100

# Gibbs sampling parameters
MCMC_STEPS=10
MCMC_BURN_IN=10

# HMM hidden and observed node sizes
H_SIZE=4
O_SIZE=10

# The target DBN (this DBN generates the data)
th0=DiscreteNode(node_size=H_SIZE)
th1=DiscreteNode(node_size=H_SIZE)
to0=DiscreteNode(node_size=O_SIZE)
```

```
start_nodes=[th0, to0]
end_nodes=[th1, to0]
node_list=[th0, th1, to0]
tdbn=DBN(start_nodes, end_nodes)
tdbn.add_intra(0, 1)
tdbn.add_inter(0, 0)
tdbn.construct()

# The model DBN (this DBN will be trained)
# Fix hidden node at t=0, and initialize with target CPD
# since we are only using 20 sequences
mh0=DiscreteNode(node_size=H_SIZE, user_cpd=th0.cpd)
mh0.fixed=1
mh1=DiscreteNode(node_size=H_SIZE)
mo0=DiscreteNode(node_size=O_SIZE)
start_nodes=[mh0, mo0]
end_nodes=[mh1, mo0]
node_list=[mh0, mh1, mo0]
mdbn=DBN(start_nodes, end_nodes)
mdbn.add_intra(0, 1)
mdbn.add_inter(0, 0)
mdbn.construct()

# Generate the data
seq_list=[]
mismask_list=[]
if mpi.rank==0:
    for i in range(0, N):
      seq, ll=tdbn.sample_sequence(T)
      seq_list.append(seq)
      mismask=zeros((T,2))
      mismask[:,0]=1
      mismask_list.append(mismask)
# Scatter data over processors
local_seq_list=mpi.scatter(seq_list)
local_mismask_list=mpi.scatter(mismask_list)

mcmc=GibbsRandom(mdbn)
em=EMEngine(mdbn, mcmc, local_seq_list, local_mismask_list)

# Start EM loop
for i in range(0, 100):
    em.do_E_step(MCMC_STEPS, MCMC_BURN_IN)
    ll=em.get_loglik()
    if mpi.rank==0:
        print 'LL=', ll
    em.do_M_step()
```

5.1.2 HMM with Gaussian output node

Changing the above HMM into an HMM with real valued Gaussian observations is trivial. Below only the relevant difference with the previous HMM code is shown.

```
# Specify covariance type (SPHE, FULL or DIAG)
COV_TYPE="FULL"
# Dimension of Gaussian node
DIM=2

# The target DBN (this DBN generates the data)
th0=DiscreteNode(node_size=H_SIZE)
th1=DiscreteNode(node_size=H_SIZE)
to0=GaussianNode(dim=DIM, node_size=H_SIZE, cov_type=COV_TYPE)

# The model DBN (this DBN will be trained)
mh0=DiscreteNode(node_size=H_SIZE, user_cpd=th0.cpd)
mh0.fixed=1
mh1=DiscreteNode(node_size=H_SIZE)
mo0=GaussianNode(dim=DIM, node_size=H_SIZE,
    cov_type=COV_TYPE)
```

5.1.3 HMM with VMF output node

Here's an HMM with a VMF output node. Again, only a the relevant part of the code is shown.

```
# An angle is encoded as a 2D vector
DIM=2

# The target DBN
th0=DiscreteNode(node_size=H_SIZE)
th1=DiscreteNode(node_size=H_SIZE)
to0=VMFNode(dim=DIM, node_size=H_SIZE)
# The model DBN
mh0=DiscreteNode(node_size=H_SIZE, user_cpd=th0.cpd)
mh0.fixed=1
mh1=DiscreteNode(node_size=H_SIZE)
mo0=VMFNode(dim=DIM, node_size=H_SIZE)
```

5.1.4 HMM with Kent output node

Here's the relevant part of an HMM with a Kent output node.

```
# The target DBN
th0=DiscreteNode(node_size=H_SIZE)
th1=DiscreteNode(node_size=H_SIZE)
```

```
to0=KentNode(node_size=H_SIZE, max_kappa=50, max_beta=10)

# The model DBN
mh0=DiscreteNode(node_size=H_SIZE, user_cpd=th0.cpd)
mh0.fixed=1
mh1=DiscreteNode(node_size=H_SIZE)
mo0=KentNode(node_size=H_SIZE, max_kappa=20, max_beta=2)
```

5.1.5 HMM with Dirichlet output node

Finally, here's a HMM that outputs a set of probabilities (in this case 4) using an observed Dirichlet node. Note that the maximum value for the alpha parameters is lower for the trained DBN, to avoid very low initial probabilities during training (a low value of the alpha parameter leads to a broad distribution).

```
# Maximum alpha of the target DBN
ATARGET=50
# Maximum alpha of the model DBN
AMODEL=10
# Dim of the Dirichlet output
DIM=4

# The target DBN
th0=DiscreteNode(node_size=H_SIZE)
th1=DiscreteNode(node_size=H_SIZE)
to0=DirichletNode(dim=DIM, node_size=H_SIZE, alpha_max=ATARGET)

# The model DBN
mh0=DiscreteNode(node_size=H_SIZE, user_cpd=th0.cpd)
mh0.fixed=1
mh1=DiscreteNode(node_size=H_SIZE)
mo0=DirichletNode(dim=DIM, node_size=H_SIZE,
    alpha_max=AMODEL)
```

5.1.6 Factorial HMM with discrete output node

The following code fragment illustrates how to implement the Factorial HMM that is shown in figure 1.3:

```
# Hidden nodes
th1=DiscreteNode(node_size=H_SIZE)
th2=DiscreteNode(node_size=H_SIZE)
th3=DiscreteNode(node_size=H_SIZE)

# Observed node
to=DiscreteNode(node_size=O_SIZE)
```

```
start_nodes=[th01, th02, th03, to]
end_nodes=[th1, th2, th3, to]

tdbn=DBN(start_nodes, end_nodes)

# Intra slice connections
tdbn.add_intra(0, 3)
tdbn.add_intra(1, 3)
tdbn.add_intra(2, 3)

# Inter slice connections
tdbn.add_inter(0, 0)
tdbn.add_inter(1, 1)
tdbn.add_inter(2, 2)

tdbn.construct()
```

In this case, the first three nodes in a slice are hidden. The corresponding mismask for a sequence can be for instance:

```
# L=length of the sequence
mismask=zeros((L,4))
# Hidden nodes
mismask[:,0]=1
mismask[:,1]=1
mismask[:,2]=1
# Node 4 is observed
```

5.2 Inference

5.2.1 Posterior distribution

5.2.2 Deterministic Viterbi

5.2.3 Stochastic Viterbi

5.2.4 Sampling

Chapter 6

Mocapy's design

Mocapy's design is fairly transparent. Figure 6.1 shows a UML-like diagram of its architecture, featuring inheritance and interactions among the classes. Some points to note are:

- All node classes are subclasses of the `Node` class. A `Node` subclass inherits some standard methods and needs to have a number of standard methods. A new `Node` subclass can immediately be used, without changing any of the other classes.
- The interface between the data/samples and the nodes is implemented in two classes: the `DiscreteFamily` and `ContinuousFamily` class, for nodes that use discrete and continuous (multidimensional) data, respectively. These should be used in new `Node` subclasses. Keeping track of the data and samples is a major task, so these classes save a lot of work! The name 'Family' refers to the fact that the classes keep track of the values of a node, its children and its parents at a certain sequence position.
- EM learning is done by the `EMEngine` class.
- Deterministic inference (smoothing and Viterbi) is done by the `InfEngineHMM` class. Sampling and stochastic Viterbi is done by the `InfEngineMCMC` class.
- Currently there is one class that does the actual sampling, i.e. the `GibbsSampler` class. It is easy to implement a new sampler class by using the `MCMC` class as a base class.
- The `EMEngine`, `InfEngineHMM`, `InfEngineMCMC` and `GibbsSampler` classes all interact directly with the `DBN` and `Node` classes to perform their operations. What goes on in the `Node` classes itself is however completely shielded by the uniform `Node` class interface.
- The `Kent`, `VonMisesFisher`, `MultiGauss` and `Dirichlet` classes implement sampling and calculate the probability density for their respective distributions.

- The `KentEstimator, GaussianEstimator` and `DirichletEstimator` calculate the ESS and the new parameters from the data.
- The `DiscretePriors` module contains the priors for discrete nodes.
- There are some useful general classes that are not shown in the diagram as well. The `Sphere` module contains methods that deal with directional data on the N-dimensional sphere (like generating random vectors on the sphere). The `Vector` module implements a vector class and various related functions like rotation around a vector. These modules are used mainly for the Kent and VMF nodes. The `ArrayMisc` module implements various methods that act on `numpy` arrays. The `Protein` module contains some functions related to biological sequences.

The code is well documented (using Epydoc mark-up language and comments in the code) and should be easy to understand, modify and extend. Contributions, suggestions, comments and bug fixes are very welcome!

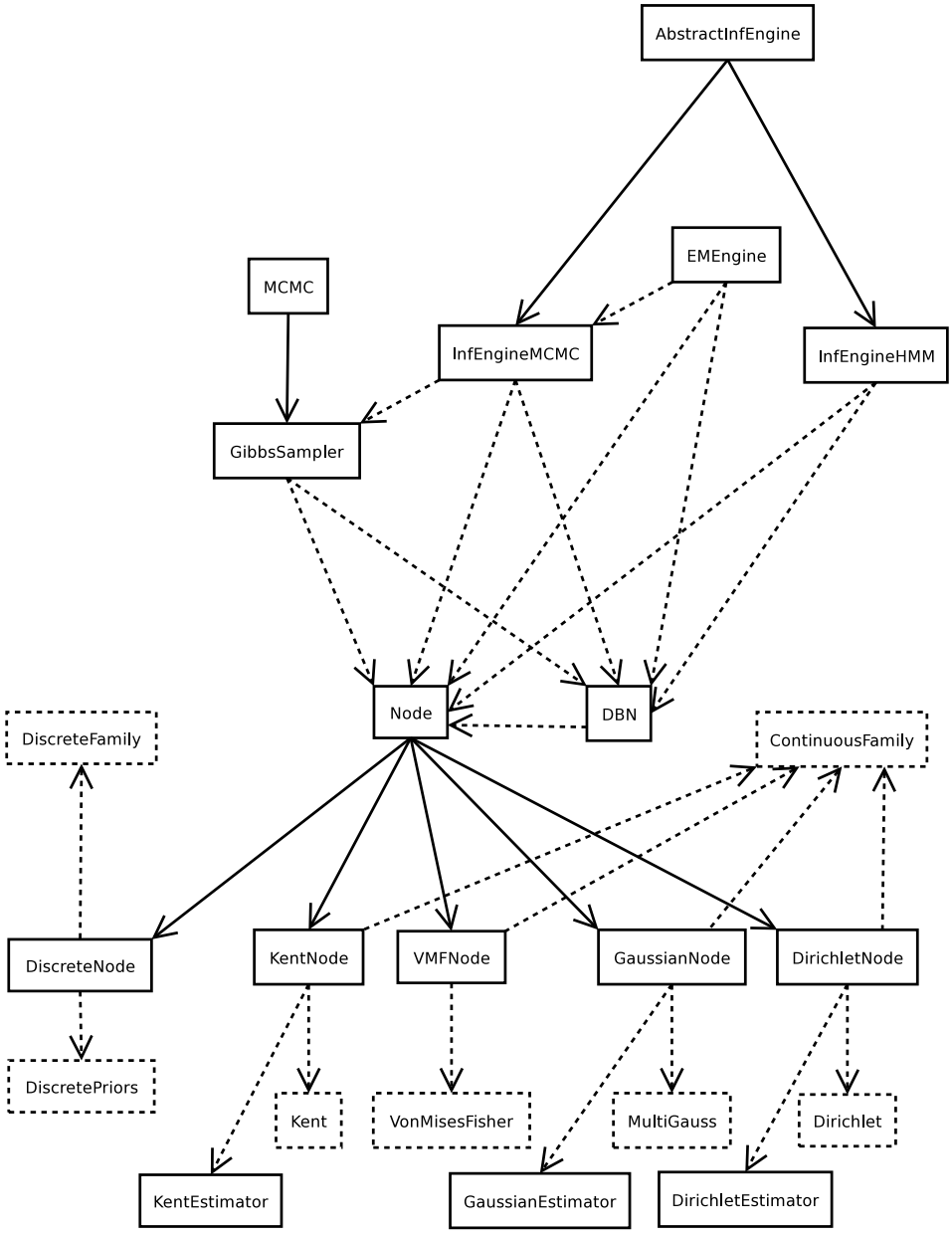

Figure 6.1: Mocapy's overall design. Full arrows indicate inheritance, dotted arrows indicate a dependency (i.e. 'a uses b'). Dotted boxes indicate modules, ordinary boxes indicate classes.

Chapter 7

Extending Mocapy

I've taken great care to document Mocapy's code, and have tried to keep it's architecture (see previous chapter) as clear as possible without compromising performance too much. Hence, it should be possible to understand how Mocapy works, and subsequently to extend the toolkit.

7.1 More Node types

Adding new node types is easy in Mocapy. For example, it took only two days to implement the VMF node after figuring out how to calculate the ESS and do the sampling (and that included testing and finding an error in an article's pseudocode).

New node classes should be subclasses of the `Node` class in the `Node` module. The `Node` class implements various methods that are used by all nodes. Take a look at the `Node` module to find out what methods need to be implemented. Once the node is implemented, it can be directly used by Mocapy, without altering additional code.

In particular, you should be able to:

- Calculate the likelihood, given the values of the node and the values of the parent nodes.
- Update the parameters of the node, given a sample of node/parent values. This most often means the calculation of the MAP or ML estimate of the parameters of a certain probability distribution.
- Sample a node's value, given the values of its parents. This is necessary if you want to generate sequences from the DBN, or if the node's value is not always observed. For a continuous node, this means generating random numbers following a particular distribution.

Often, these three things are not entirely trivial. For an example of these three operations for an interesting node type (the VMF node), see the technical report by Dhillon and Sra [5].

7.2 More sampling methods

It's easy to add new sampling algorithms. Sampling is done by a subclass of the `MCMC` base class in the `MCMC` module. Currently, only random-sweep Gibbs sampling is implemented in the `GibbsRandom` class, which can also be found in the `MCMC` module. This can serve as a good example of how to proceed.

Chapter 8

Future developments

8.1 More node types

Mocapy can currently handle discrete, real valued, probability parameter and directional data using the discrete, Gaussian, Dirichlet and VMF/Kent nodes, respectively. It's easy to add new node types (for example a Bingham node, which can be used to model axial data), and these might be added when I need them and find the time to implement them, or if Mocapy users are so kind as to donate new node types. For the near future, I hope to add a distribution on the torus, which would allow to model a pair of dependent torsion angles. Also in the pipeline are the multivariate Polya distribution and the Poisson distribution.

8.2 More sampling methods

Sampling methods like Metropolis-Hastings could be added fairly easily to Mocapy. I'm also interested in adding support for some type of Blocked Gibbs sampling, which is necessary for the implementation of Hidden Semi-Markov Models.

8.3 Priors

Currently, priors are only available for discrete nodes. I'm planning to add priors for Dirichlet[1] and Gaussian nodes, which are necessary to avoid the well known problem of trapping states during parameter learning of mixture models [7, 6].

[1] Note that using Dirichlet priors for discrete nodes is of course already implemented.

Chapter 9

Acknowledgements

Mocapy is part of a protein structure prediction project at the *Bioinformatics Centre, University of Copenhagen, Denmark*. Funding came from the *Lundbeckfond* (February 2004-January 2005, `www.lundbeckfonden.dk`) and a Marie Curie Intra-European fellowship within the 6th European Community Framework Programme (after February 2005). I thank Anders Krogh, *Bioinformatics Center, University of Copenhagen*, for his support of this project. John T. Kent, *Department of Statistics, University of Leeds, UK*, kindly contributed a method to sample efficiently from the Kent distribution.

In addition, I thank Ole Winther, *Digital Signal Processing, Informatics and Mathematical Modelling, Technical University of Denmark*, Niels Hansen, *Bioinformatics Center, University of Copenhagen*, Søren Feodor Nielsen, *Department of Statistics and Operations research, Institute for Mathematical Sciences, University of Copenhagen*, John Kent, Kanti Mardia and Charles Taylor, *University of Leeds, UK*, Jesper Ferkinghoff-Borg, *Niels Bohr Institute, Denmark* for interesting discussions and suggestions.

All shortcomings in the project are of course due to the author of the toolkit.

Chapter 10

GNU Lesser General Public License

Bibliography

[1] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50:5–43, 2003.

[2] Pierre Baldi and Søren Brunak. *Bioinformatics : the machine learning approach*. MIT Press, 1998.

[3] W. Boomsma, JT. Kent, KV. Mardia, CC. Taylor, and T. Hamelryck. Graphical models and directional statistics capture protein structure. In S. Barber, PD. Baxter, KV. Mardia, and RE. Walls, editors, *Interdisciplinary Statistics and Bioinformatics*, volume 25, pages 91–94. Leeds University Press, 2006.

[4] SP. Brooks. Markov chain Monte Carlo method and its applications. *The Statistician*, 47:69–100, 1998.

[5] IS. Dhillon and S. Sra. Modeling Data using Directional Distributions. Technical report, University of Texas, Austin, 2003.

[6] J. Diebolt and EHS. Ip. *Markov Chain Monte Carlo in practice*, chapter 15: Stochastic EM: method and application, pages 259–273. Chapman & Hall/CRC, 1996.

[7] J. Diebolt and CP. Robert. Estimation of finite mixture distributions through bayesian sampling. *J. Royal Stat. Soc. B*, 56:363–375, 1994.

[8] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge Univ. Press, 2000.

[9] Z. Ghahramani. Learning dynamic Bayesian networks. *Lecture Notes in Computer Science*, 1387:168–197, 1998.

[10] Zoubin Ghahramani and Michael I. Jordan. Factorial hidden markov models. *Machine Learning*, 29:245–273, 1997.

[11] W.R. Gilks, S.T. Richardson, and D.J. Spiegelhalter, editors. *Markov Chain Monte-Carlo in practice*. Chapman & Hall/CRC, London, 1996.

[12] SJ. Godsill, A. Doucet, and M. West. Maximum a posteriori sequence estimation using Monte Carlo particle filters. *Ann. Inst. Stat. Math.*, 53:82–96, 2001.

[13] T. Hamelryck, JT. Kent, and A. Krogh. Sampling realistic protein conformations using local structural bias. *PLoS Comput. Biol.*, 2(9):e131, 2006.

[14] MI. Jordan, editor. *Learning in graphical models*. MIT Press, 1998.

[15] JT. Kent. The Fisher-Bingham distribution on the sphere. *J. Royal Stat. Soc.*, 44:71–80, 1982.

[16] JT Kent and T. Hamelryck. Using the Fisher-Bingham distribution in stochastic models for protein structure. In S. Barber, PD. Baxter, KV. Mardia, and RE. Walls, editors, *Quantitative Biology, Shape Analysis, and Wavelets*, volume 24, pages 57–60. Leeds University Press, 2005.

[17] P. Leong and S. Carlile. Methods for spherical data analysis and visualization. *J. Neurosci. Met.*, 80:191–200, 1998.

[18] K. V. Mardia and P. Jupp. *Directional Statistics*. John Wiley and Sons Ltd., 2nd edition, 2000.

[19] P. Miller. pyMPI - An introduction to parallel Python using MPI, 2002.

[20] TP. Minka. Estimating a Dirichlet distribution. Technical report, Microsoft Research, Cambridge, UK, 2003.

[21] K. Murphy. An introduction to graphical models. Technical report, UC Berkeley, Computer Science Division, 2001.

[22] K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, 2002.

[23] R. Neal. Probablistic inference using markov chain monte carlo methods. Technical report, Dept. of Computer Science, University of Toronto, September 1993.

[24] SF. Nielsen. The stochastic em algorithm: Estimation and asymptotic results. *Bernoulli*, 6:457–489, 2000.

[25] D. Peel, WJ. Whiten, and GJ. McLachlan. Fitting Mixtures of Kent Distributions to Aid in Joint Set Identification. *J. Am. Stat. Ass.*, 96:56–63, 2001.

[26] LR. Rabiner. A Tutorial in Hidden Markov Models and Selected Applications in Speech Recognition. *Proc. of the IEEE*, 77:257–286, 1989.

[27] K. Sjölander, K. Karplus, M. Brown, R. Hughey, A. Krogh, I.S. Mian, and D. Haussler. Dirichlet mixtures: a method for improved detection of weak but significant protein sequence homology. *Comput. Appl. Biosci.*, 12:327–45, 1996.

[28] E.P. Xing and R.M. Karp. MotifPrototyper: a Bayesian profile model for motif families. *Proc. Natl. Acad. Sci. USA*, 101:10523–8, 2004.