User Manual for Libra 1.1.2c

Daniel Lowd <lowd@cs.uoregon.edu>
Amirmohammad Rooshenas <pedram@cs.uoregon.edu>

April 22, 2019

1 Overview

The Libra Toolkit provides a collections of algorithms for learning and inference with probabilistic models in discrete domains. Each algorithm can be run from the command-line with a variety of options using the `libra` script:

```
libra <command> [options]
```

Different algorithms share many common options and file formats, so they can be used together for experimental or application-driven workflows. This user manual gives a brief overview of Libra's functionality, describes the file formats used, and explains the operation of each command in the toolkit. See the developer's guide for information on modifying and extending Libra.

1.1 Representations

In Libra, each probabilistic model represents a probability distribution, $P(\mathcal{X})$, over set of discrete random variables, $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$. Libra supports Bayesian networks (BNs), Markov networks (MNs), dependency networks (DNs) [8], sum-product networks (SPNs) [19], arithmetic circuits (ACs) [6], and mixtures of trees (MT) [17].

BNs and DNs represent a probability distribution as a collection of conditional probability distributions (CPDs), each denoting the conditional probability of one random variable given its parents. The most common CPD representation is a table with one entry for each joint configuration of the variable and its parents. In addition to tables, Libra also supports tree CPDs and more complex feature-based CPDs. In a tree CPD, nodes and branches specify conditions on the parent variables, and each leaf contains the probability of the target variable given the conditions on the path to that leaf. When many configurations have identical probabilities, trees can be exponentially more compact than table CPDs. Feature-based CPDs use a weighted set of conjunctive features. Each conjunctive feature represents a set of conditions on the parent variables and target variables, similar to the paths in

the decision tree, except that they need not be mutually exclusive or exhaustive. Feature-based CPDs can easily represent logistic regression CPDs, boosted trees, and other complex CPDs. Libra represents MNs as collections of factors or potential functions, each of which can be represented as a table, tree, or set of conjunctive features. For general background BNs and MNs, see Koller and Friedman [9]. For an introduction to DNs, see Heckerman et al. [8].

SPNs and ACs represent a probability distribution as a directed acyclic graph of sum and product nodes, with parameters or indicator variables at the leaves. When the indicator variables are set to encode a particular configuration of the random variables, the root of the circuit evaluates to that configuration's probability. The indicator variables can also be set so that one or more variables are marginalized, which permits efficient computation of marginal and conditional probabilities. The ability to perform efficient, exact inference is a key strength of SPNs and ACs. SPNs and ACs are also very flexible, since they can encode complex structures that do not correspond to any compact BN or MN. Libra has separate SPN and AC representations. The SPN representation shows high-level structure more clearly, but the AC representation is better supported for inference. The SPN representation can be converted to the AC representation.

The mixture of trees (MT) representation represents a probability distribution as a weighted sum of tree-structured BNs. Since SPNs are good at modeling mixtures, Libra represents mixtures of trees as SPNs.

For more detail about the model file formats, see Section 4.

1.2 Algorithms

The learning algorithms in Libra fit a probabilistic model to the empirical distribution represented by the training data. The learning methods currently available in Libra require that the data be fully-observed so that no variable values are missing. Most methods learn the model structure as well as its parameters.

Libra includes the following commands for learning probabilistic models:

- `cl`: The Chow-Liu algorithm for tree-structured BNs [5]
- `bnlearn`: Learning BNs with tree CPDs [4]
- `dnlearn`: Learning DNs with tree or logistic regression CPDs [8]
- `dnboost`: Learning DNs with boosted tree CPDs
- `acbn`: Using ACs to learn tractable BNs with tree CPDs [11]
- `acmn`: Using ACs to learn tractable MNs with conjunctive features [13]
- `idspn`: The ID-SPN algorithm for learning SPN structure [20]
- `mtlearn`: Learning mixtures of trees [17]
- `dn2mn`: Learning MNs from DNs [15]
- `mnlearnw`: MN weight learning, to maximize L1/L2 penalized pseudo-likelihood
- `acopt`: Parameter learning for ACs, to match an empirical distribution or another BN or MN [12]

Inference algorithms compute marginal and joint probabilities, given evidence. For models that are represented as ACs or SPNs, Libra supports:

- `acquery`: Exact inference in ACs
- `spquery`: Exact inference in SPNs

The following inference algorithms are supported for BNs, MNs, and DNs:

- `mf`: Mean field inference [14]
- `gibbs`: Gibbs sampling
- `icm:` Iterated conditional modes [2]

For BNs and MNs, three more inference algorithms are supported:

- `bp`: Loopy belief propagation [18]
- `maxprod`: Max-product
- `acve`: AC variable elimination [3]

The last method compiles a BN or MN into an AC. Thus, `acve` and `acquery` can be used together to perform exact inference in a BN or MN.

A few utility commands round out the toolkit:

- `bnsample`: BN forward sampling
- `mscore`: Likelihood and pseudo-likelihood model scoring
- `mconvert`: Model conversion and conditioning on evidence
- `spn2ac`: Convert SPNs to ACs
- `fstats`: File information for any supported file type

2 Installation

Libra was designed to run from the command line under Linux and Mac OS X. It also may work with Windows (Cygwin environment), but it is not completely supported in the current version. Libra can be installed using OPAM or manually.

2.1 Installation Using OPAM

The recommended way to install Libra is through the OPAM package manager. Detailed instructions for installing OPAM on various platforms can be found at `http://opam.ocaml.org/doc/Install.html`.

On macOS this might be done using homebrew (`https://brew.sh/`):

```
brew install opam
```

Or with the built-in package manager for your Linux distribution (`apt`, `dnf`):

```
dnf install opam
```

Once installed, opam will likely have installed the latest version of OCaml by default. We will need to use 4.02 for Libra:

```
opam switch create 4.02.1
opam init
```

Finally, to install libra-tk, run:

```
opam install libra-tk
```

Binaries will be installed in the <*version*>`/bin/` subdirectory of your OPAM installation, where <*version*> is the OCaml version being used (e.g., `system`, `4.02.1`, `4.01.0`). Documentation and example files will be installed in the OPAM subdirectory <*version*>`/share/doc/libra-tk`.

On OS X, the default Clang compiler may fail to build `ocaml-expat` with the following error:

```
expat_stubs.c:23:10: fatal error: 'caml/mlvalues.h' file not found
```

To fix this, run the following command before installing:

```
export C_INCLUDE_PATH=`ocamlc -where`:$C_INCLUDE_PATH
```

Note that this command uses backtics (`), not single quote (').

2.2 Manual Installation

If you prefer, you can build and install Libra manually. Libra depends on OCaml (version 4.00.0 or later), the `ocamlfind` build tool, the `ocaml-expat` library, and `oasis` To install these using OPAM, first install OPAM (see above) and then run:

```
opam install ocamlfind ocaml-expat oasis
```

Libra also depends on GNU Make and the Expat XML parsing library. The automated tests depend on Perl, awk, and the diff utility. In some OS environments, these are already installed by default.

After installing the dependencies, download the Libra source code and unpack the source distribution:

```
tar -xzvf libra-tk-1.1.2c.tar.gz
cd libra-tk-1.1.2c
```

This creates the `libra-tk-1.1.2c/` directory and makes it your working directory. For the remainder of this document, all paths will be relative to this directory.

Before building, you can optionally use the `configure` script to change installation directories and other settings. Run `./configure --help` to see a list of configuration options. For example, `./configure --prefix ~` changes the installation directory from `/usr/local` to your home directory.

Next, build the executables:

```
make
```

All programs should now be present in the directory `bin/`, with one executable file for each command in Libra. For convenience, you may use the `libra` script, which acts as an interface to all of the commands in the toolkit.

To install the executables and PDF documentation, run:

```
make install
```

Depending on your system and the installation path, this command may require superuser permissions. To run the installation step as a superuser, use the command `sudo make install`. Alternately, you can change the installation directories using `configure`.

To create HTML documentation for the libraries, run:

```
make doc
```

This generates ocamldoc-based HTML files for the libraries and puts them in `doc/html`. Open `doc/html/index.html` in a web browser to view the documentation.

3 Quick Start

This section will demonstrate basic usage of the Libra toolkit through a short tutorial. In this tutorial, we will train several models from data, evaluate model accuracy on held-out data, and answer queries exactly and approximately. All necessary files are included in the `doc/examples/` directory of the standard distribution. These files are also copied during installation. If you installed Libra using OPAM, they can be found in the <*version*>`/share/doc/libra-tk/examples/` subdirectory of your OPAM installation, where <*version*> is the OCaml version being used (e.g., `system`, `4.02.1`, `4.01.0`). If you installed Libra manually, the default installation directory is `/usr/local/share/doc/libra-tk/examples/`.

As our dataset, we will use the Microsoft Anonymous Web Data[1], which records the areas (Vroots) of microsoft.com that each user visited during one week in February 1998. A small subset of this data is present in the examples directory, already converted into Libra's data format: each line is one sample, represented as a list of comma-separated values (one discrete value per variable).

To continue with the quick start, change to the examples directory:

```
cd doc/examples
```

[1] Available from the UCI repository at: http://kdd.ics.uci.edu/databases/msweb/msweb.html

3.1 Training Models

To train a tree-structured BN with the Chow-Liu algorithm, use the `cl` command:

```
libra cl -i msweb.data -s msweb.schema -o msweb-cl.bn -prior 1
```

The most important options are `-i`, for indicating the training data, and `-o`, for indicating the output file. Libra has a custom format for Bayesian networks (`.bn`) and also supports two external formats, BIF (`.bif`) and XMOD (`.xmod`). (See Section 4 for more information about file formats.) Libra will automatically guess the model type from the filename suffix.

The `-s` option allows you to specify a variable schema, which defines the number of values for each variable. If unspecified, Libra will guess from the training data. `-prior` specifies the prior counts to use when estimating the parameters of the network, for smoothing[2].

The `-log` flag redirects log output to another file instead of standard output. You can use the flag `-v` to enable more verbose output, or `-debug` to turn on detailed debugging output. These three flags are valid with any command, although for some commands they have minimal effect. Try them with `cl` to see what happens.

To see a list of all options, run `libra cl` without any other arguments. This trick works for every command in Libra. You can run `libra` with no arguments to see a list of all commands.

Other learning algorithms offer similar options. For example, to learn a Bayesian network with tree CPDs, use `bnlearn`:

```
libra bnlearn -i msweb.data -s msweb.schema
              -o msweb-bn.bn -prior 1 -ps 10
```

The option `-ps 10` specifies a "per-split" penalty, so that a tree CPD is only extended when the new split increases the log-likelihood by more than 10. This acts as a structure prior, to control overfitting.

The other programs for learning models from data are: `dnlearn` and `dnboost`, for learning DNs, and `acbn`, `acmn`, `idspn`, and `mtlearn`, for learning various kinds of tractable models in which exact inference is efficient.

3.2 Model Scoring

We can compute the average log-likelihood per example using the `mscore` program:

```
libra mscore -m msweb-cl.bn -i msweb.test
libra mscore -m msweb-bn.bn -i msweb.test
```

[2]Specifically, the option `-prior` α specifies a symmetric Dirichlet prior with parameter α/d, where d is the number of possible values for the variable.

`-m` specifies the MN, BN, DN, or AC to score, and `-i` specifies the test data. The filetype is inferred from the extension. To see the likelihood of every test case individually, enable verbose output with `-v`. For DNs and MNs, `mscore` reports pseudo-log-likelihood instead of log-likelihood.

You can obtain additional information about models or data files using `fstats`:

```
libra fstats -i msweb-cl.bn
libra fstats -i msweb.test
```

3.3 Inference

We can either use exact inference or approximate inference. To do exact inference, we must first compile the learned model into an AC:

```
libra acve -m msweb-cl.bn -o msweb-cl.ac
```

The AC is an inference representation in which many queries can be answered efficiently. However, depending on the structure of the BN, the resulting AC could be very large. Section 7.1 gives more information about `acve`. Now that we have an AC, we can use it to answer queries using `acquery`. The simplest query is computing all single-variable marginals:

```
libra acquery -m msweb-cl.ac -marg
```

`-marg` specifies that we want to compute the marginals. The output is a list of 294 marginal distributions, which is a bit overwhelming. We can also use `acquery` to compute conditional log probabilities $\log P(q|e)$, where q is a query and e is evidence:

```
libra acquery -m msweb-cl.ac -q msweb.q -ev msweb.ev
```

`acquery` also supports MPE (most probable explanation) queries and a number of other options; see Section 7.2 for more information.

For approximate inference, we can use mean field (`mf`), loopy belief propagation (`bp`), or Gibbs sampling (`gibbs`). Their basic options are the same as the tools for exact inference, but each algorithm has some specific options:

```
libra mf -m msweb-bn.bn -q msweb.q -ev msweb.ev -v
libra bp -m msweb-bn.bn -q msweb.q -ev msweb.ev -v
libra gibbs -m msweb-bn.bn -q msweb.q -ev msweb.ev -v
```

See Section 7 for detailed descriptions of the inference algorithms.

4 File Formats

This section describes the file formats supported by Libra.

4.1 Data and Evidence

For data points, Libra uses comma-separated lists of variable values, with asterisks representing values that are unknown. This allows the same format to be used for training examples and evidence configurations. Each data point is terminated by a newline. In some programs (`mscore`, `acbn`, and most inference methods), each data point may be preceded by an optional weight, in order to learn or score a weighted set of examples. The default weight is 1.0. The following are all valid data points:

```
0,0,1,0,4,3,0
0.2|0,0,1,1,2,0,1
1000|0,0,0,0,0,0,0
```

Evidence files have the same comma separated format, except when the value of a variable has not been observed as evidence, we put a '*' in the related column:

```
0,0,1,*,4,3,*
```

The corresponding query data point should be consistent with the evidence. For example:

```
0,0,1,1,4,3,0
```

The above query and evidence can be used to find the conditional probability of $P(X_4, X_7 | X_1 = 0, X_2 = 0, X_3 = 1, X_5 = 4, X_6 = 3)$.

4.2 Graphical Models

Libra has custom file formats for BNs (`.bn`), DNs (`.dn`), and MNs (`.mn`) which allows for very flexible factors, including trees, tables, and conjunctive features. Model parameters are represented as log probabilities or unnormalized log potential functions. See Figure 1 for an example of a complete `.bn` file.

The first line in the file is a comma-separated variable schema listing the range of each variable. The next line specifies the model type (BN, DN, or MN). The rest of the file defines the conditional probability distributions (CPDs) (for BNs and DNs) or factors (for MNs), enclosed within braces ({}). Each CPD definition begins with the name of the child variable (`v1`, `v2`, `v3`, etc.) followed by a set of factor definitions enclosed in braces. MNs do not contain CPDs, and simply use the raw factors.

Different factor types (table, tree, feature set) have different formats. The simplest is a factor for a single feature, which is written out as a real-valued weight and a list of variable conditions. For example the following line defines a feature with a weight of 1.2 for the conjunction $(X_0 = 1) \wedge (X_3 = 0) \wedge (X_4 \neq 2)$:

```
1.2 +v0_1 +v3_0 -v4_2
```

example.bn:

```
2,2,2
BN {

v0 {
  table {
  -7.60e+00 +v0_1
  -5.00e-04 +v0_0
  }
}

v1 {
  tree {
   (v0_0
        (v1_0 -1.91e-02 -3.97e+00)
        (v1_0 -5.32e-02 -2.96e+00))
  }
}

v2 {
  features {
  1.5 +v0_0 +v1_0 +v2_0
  2.5 +v0_1 +v1_1 +v2_1
  }
}

}
```

Figure 1: Example .bn file showing three types of CPDs.

A feature set factor consists of a list of mutually exclusive features, each in the format described above. The list is surrounded is preceded by the word "`features`" and an opening brace ('{'), and followed by a closing brace ('}'). For example:

```
features {
-1.005034e-02 +v5_1 +v0_1
-2.302585e+00 +v5_0 +v0_1
-4.605170e+00 +v5_1 +v0_0
-1.053605e-01 +v5_0 +v0_0
}
```

A table factor has the same format as a feature set, except with the word "`table`" in place of the word "`features`", and the features in the list need not be mutually exclusive. After reading a table factor, Libra creates an internal tabular representation. The size of this table is exponential in the number of variables referenced by

the listed features.

The format of a tree factor is similar to a LISP s-expression, as illustrated in the following example:

```
tree {
(v1_0
    (v3_0
        (v0_0
            (v2_0 -1.905948e-02 -3.969694e+00)
            (v2_0 -5.320354e-02 -2.960105e+00))
        (v2_0 -2.341261e-01 -1.566675e+00))
    (v2_0 -2.121121e-01 -1.654822e+00))
}
```

When $x_1 = 0$, $x_3 = 1$, and $x_2 = 1$, then the log value of this factor is `-1.566675`.

To indicate an infinite weight, write "`inf`" or `-inf`".

4.3 Arithmetic Circuits

For arithmetic circuits, Libra uses a custom file format (`.ac`). The first line of a `.ac` file specifies the range of each variable. For example, `(2 2 2)` indicates a domain with three binary-valued variables. The following lines list the nodes in the network, one per line. Each line specifies the node type and any of its parameters, such as the value of a constant node, the variable and value index for an indicator variable node, and the indices of child nodes for sum and product nodes. For example, `n 0.9995` represents the constant value 0.9995, `v 2 0` represents the indicator variable for the first value of the third variable, and `* 0 13` represents the product of the first and fourteenth nodes. Each node must appear after all of its children. The root of the circuit is therefore the last node, followed by the string `EOF`.

After defining all nodes, an arithmetic circuit file optionally describes how its parameters relate to conjunctive features. For example, `1 0.981081 +v0_0 +v1_0` indicates that the second node in the circuit contains the parameter associated with the conjunctive feature $(X_0 = 0) \wedge (X_1 = 0)$. Libra uses this supplementary information when converting circuits into Markov networks, optimizing their parameters, or computing pseudo-likelihood.

See Figure 2 for a complete example. This circuit was generated from the BN in Figure 1, using the command: `libra acve -m example.bn -o example.ac`. Note that, unlike `.bn` and `.mn` files, all parameters in the circuit files are stored as raw values, not log values.

4.4 Sum-Product Networks and Mixtures of Trees

Libra represents sum-product networks with a custom file format (`.spn`). Each `.spn` file is a directory containing a model file `spac.m` and many `.ac` files. In the model

example.ac:

```
(2 2 2)
v 1 0
n 0.981081
n 0.999500
v 0 0
* 2 3
n 4.481689
* 4 5
* 1 6
n 0.948190
n 0.000500
v 0 1
* 9 10
* 8 11
+ 7 12
* 0 13
v 1 1
n 0.018873
* 16 4
n 0.051819
* 18 11
+ 17 19
* 15 20
+ 14 21
v 2 0
* 22 23
* 1 4
+ 25 12
* 0 26
n 12.182494
* 11 28
* 18 29
+ 17 30
* 15 31
+ 27 32
v 2 1
* 33 34
+ 24 35
EOF

2 0.999500 +v0_0
9 0.000500 +v0_1
1 0.981081 +v0_0 +v1_0
8 0.948190 +v0_1 +v1_0
16 0.018873 +v0_0 +v1_1
18 0.051819 +v0_1 +v1_1
5 4.481689 +v0_0 +v1_0 +v2_0
28 12.182494 +v0_1 +v1_1 +v2_1
EOF
```

Figure 2: Example .ac file based on the BN from Figure 1. For compactness and readability, the file listing has been split into two columns.

file, a node is represented as: `n` <*nid*> <*type*> <*pid*> where *nid* and *pid* are the id of the node and its parent, respectively, and *type* can be `+`, `*`, or `ac`. For each node with type `ac`, there is a corresponding file `spac-nid.ac` in the `.spn` directory. Libra uses the same file format for mixtures of trees.

See Figure 3 for an example model file. This model contains seven nodes, representing a mixture of products of arithmetic circuits. The first two lines specify the network name (`spac`) and the number of nodes (7). The first node (`n 0 + -1`) is a sum node with no parent. The next line specifies its children, nodes 1 and 2. For sum nodes, an additional line specifies the log mixture weight of each child. The following line specifies that this first node is a probability distribution over all three variables (0 1 2). The next two nodes are product nodes, each with two children, and each representing a distribution over all three variables. Last are the `ac` nodes, referencing the external files `spac-1.ac` through `spac-4.ac`. However, each of these circuits is a probability distribution over a different subset of the domain variables:

spac.m:

```
spac
7
n 0 + -1
1 2
-1.204 -0.357
0 1 2
n 1 * 0
3 4
0 1 2
n 2 * 0
5 6
0 1 2
n 3 ac 1
0 1
n 4 ac 1
2
n 5 ac 2
1 2
n 6 ac 2
0
```

Figure 3: Example model file from a `.spn` directory. This represents a mixture of products of arithmetic circuits.

the first is a distribution over variables 0 and 1, the second is over variable 2, the third is over variables 1 and 2, and the last is over variable 0.

4.5 External File Formats

For interoperability with other toolkits, Libra supports several other file formats as well. For Bayesian networks and dependency networks, Libra (mostly) supports two previously defined file formats. The first is the Bayesian interchange format (BIF) for BNs and DNs with table CPDs.[3] Note that this is different from the newer XML-based XBIF format, which may be supported in the future.[4] The second is the WinMine Toolkit XMOD format, which supports both table and tree CPDs.[5]

Libra also supports the Markov network model file format used the by the UAI inference competition[6]. However, Libra does not currently support the UAI evidence

[3]BIF is described here: http://www.cs.cmu.edu/~fgcozman/Research/InterchangeFormat/Old/xmlbif02.html.

[4]Scripts to translate between BIF and XBIF are available here: http://ssli.ee.washington.edu/~bilmes/uai06InferenceEvaluation/uai06-repository/scripts/.

[5]The WinMine Toolkit also provides a visualization tool for XMOD files, `DNetBrowser.exe`.

[6]The UAI inference file format is described here: http://www.cs.huji.ac.il/project/UAI10/fileFormat.php.

or result file formats.

5 Common Options

Libra is designed to be run on the command line in a UNIX-like environment or called by scripts in research or application workflows. No GUI environment is provided. A list of options for any command can be produced by running it with no arguments, or by running it with a `-help` or `--help` argument.

The following output options are valid with every command:

`-log` <*file*>: Output logging information to the specified file

`-v`: Enable verbose logging output. Verbose output always lists the full command line arguments, and often includes additional timing information.

`-debug`: Enable both verbose and debugging logging output. Debugging output varies from program to program and is subject to change.

Option names for the following common options are mostly standardized among the Libra commands that use them:

`-i` <*file*>: Train or test data

`-m` <*file*>: Model file

`-o` <*file*>: Output model or data

`-seed` <*int*>: Seed for the random number generator

`-q` <*file*>: Query file

`-ev` <*file*>: Query evidence file

`-mo` <*file*>: File for writing marginals or MPE states

`-sameev`: If specified, use the first line in the evidence file as the evidence for all queries.

The last four options are exclusive to inference algorithms. Inference methods that compute probabilities print out the conditional log probability of each query given the evidence (optional). If no query is specified, the marginal distribution of each non-evidence variable is printed out instead. Methods that compute the most probable explanation (MPE) state print out the Hamming loss between the true and inferred states divided by the number of non-evidence variables. If no query file is specified, then MPE methods print out the MPE state for each evidence.

6 Learning Methods

6.1 `cl`: Chow-Liu Algorithm

Options:

`-i`	Training data file
`-s`	Data schema (optional)
`-o`	Output Bayesian network
`-prior`	Prior counts of uniform distribution

The Chow-Liu algorithm (**cl**) [5] learns the maximum likelihood tree-structured BN from data. The algorithm works by first computing the mutual information between each pair of variables and then greedily adding the edge with highest mutual information (excluding edges that would form cycles) until a spanning tree is formed. (For sparse data, faster implementations are possible [16].) The option `-prior` determines the prior counts used for smoothing.

```
libra cl -i msweb.data -s msweb.schema -o msweb-cl.bn -prior 1.0
```

6.2 `bnlearn`: Learning Bayesian Networks

Options:

`-i`	Training data file
`-s`	Data schema (optional)
`-o`	Output Bayesian network (XMOD or BN format)
`-prior`	Prior counts of uniform distribution
`-parents`	Restrictions on variable parents
`-ps`	Per-split penalty [-1.0]
`-kappa`	Alternate per-split penalty specification [0.0]
`-psthresh`	Output models for various per-split penalty thresholds [false]
`-maxs`	Maximum number of splits [1000000000]

The `bnlearn` command learns a BN with decision-tree CPDs (see [4]). In order to avoid overfitting, `bnlearn` uses a per-split penalty for early stopping (`-ps`), similar to the penalty on the number of parameters used by Chickering et al. [4]. The `-kappa` option is equivalent to a setting a per-split penalty of log kappa. With the `-psthresh` option, `bnlearn` will output models for different values of `-ps` as learning progresses, without needing to rerun training.

```
libra bnlearn -i msweb.data -o msweb.xmod -prior 1.0 -psthresh
```

An allowed parents file may be specified (`-parents` <*file*>), which restricts the sets of parents structure learning is allowed to choose for each variable. Restrictions can limit the parents to a specific set of parents ("`none except 1 2 8 10`") or to any parent not in a list ("`all except 3 5`"). An example parent file is below:

```
# This is a comment
0: all except 1 3   # var 0 may not have var 1 or 3 as a parent
1: none except 5 2  # only vars 5 and 2 may be parents of var 1
2: none             # var 2 may have no parents
5: all              # var 5 may have any parents
```

6.3 `acbn`: Learning Tractable BNs with ACs

Options:

`-i`	Training data file
`-s`	Data schema (optional)
`-o`	Output circuit
`-mo`	Output Bayesian network (XMOD or BN format)
`-prior`	Prior counts of uniform distribution [1]
`-parents`	Restrictions on variable parents
`-pe`	Per-edge penalty [0.1]
`-shrink`	Shrink edge cost before halting [false]
`-ps`	Per-split penalty [-1.0]
`-kappa`	Alternate per-split penalty specification [0.0]
`-psthresh`	Output models for various per-split penalty thresholds [false]
`-maxe`	Maximum number of edges [1000000000]
`-maxs`	Maximum number of splits [1000000000]
`-quick`	'Quick and dirty' heuristic [false]
`-qgreedy`	Find greedy local optimum after quick heuristic [false]
`-freq`	Number of iterations between saving current circuit [-1]

LearnAC (`acbn`) [11] learns a BN with decision-tree CPDs using the size of the corresponding AC as a learning bias. This effectively trades off accuracy and inference complexity, guaranteeing that the final model will support efficient, exact inference. `acbn` outputs both an AC (specified with `-o`) and a BN (specified with `-mo`). It uses the same BN learning as `bnlearn` as well as some additional options that are specific to learning an AC.

The LearnAC algorithm penalizes the log-likelihood with the number of edges in AC, which relates to the complexity of inference. The option `-pe` specifies the per-edge penalty for `acbn`. When the per-edge penalty is higher, `acbn` will more strongly prefer structures with few edges. The per-split penalty is for early stopping, in order to avoid overfitting.

```
libra acbn -i msweb.data -o msweb.ac -mo msweb-bn.xmod -ps 1 -pe 0.1
```

We can also bound the size of the AC to a certain number of edges using the `-maxe` option. The combination of `-maxe 100000 -shrink` specifies the maximum number of edges (100,000, in this case) and tells `acbn` to keep running until this edge limit is met, reducing the per-edge cost as necessary. This way, we can start

with a conservative edge cost (such as 100), select only the most promising simple structures, and make compromises later as our edge budget allows. A final option is `-quick`, which relaxes the default, greedy heuristic to be only approximately greedy. In practice, it is often an order of magnitude faster with only slightly worse accuracy.

Algorithms that learn ACs, such as `acbn`, can be much slower than those that only learn graphical models, such as `bnlearn`. The reason is that `acbn` is also learning a complex inference representation, and manipulating that representation can be relatively expensive.

For good results in a reasonable amount of time, we recommend using `-quick`, `-maxe`, and `-shrink`:

```
libra acbn -i msweb.data -o msweb.ac -mo msweb-bn2.bn -quick
                                     -maxe 100000 -shrink
```

6.4 `acmn`: Learning Tractable MNs with ACs

Options:

`-i`	Training data file
`-s`	Data schema (optional)
`-o`	Output circuit
`-mo`	Output Markov network
`-sd`	Standard deviation of Gaussian weight prior [1.0]
`-l1`	Weight of L1 norm [0.0]
`-pe`	Per-edge penalty [0.1]
`-sloppy`	Score tolerance heuristic [0.01]
`-shrink`	Shrink edge cost before halting [false]
`-ps`	Per-split penalty [-1.0]
`-psthresh`	Output models for various per-split penalty thresholds [false]
`-maxe`	Maximum number of edges [1000000000]
`-maxs`	Maximum number of splits [1000000000]
`-quick`	'Quick and dirty' heuristic [false]
`-halfsplit`	Produce only one child feature in each split [false]
`-freq`	Number of iterations between saving current circuit [-1]

The ACMN algorithm [13], `acmn`, learns a tractable Markov network. ACMN outputs an AC augmented with Markov network features and weights. The `-mo` option specifies a second output file for the learned MN structure. ACMN supports both L1 (`-l1`) and L2 (`-sd`) regularization to avoid overfitting parameters. Like `acbn`, ACMN uses per-split and per-edge penalties. See Section 6.3 and Section 6.2 for common options: `-ps, -pe, -psthresh, -maxe, -maxs, -shrink, -quick`.

```
libra acmn -i msweb.data -s msweb.schema -o msweb-mn.ac -l1 5
     -sd 0.1 -maxe 100000 -shrink -ps 5 -mo msweb.mn
```

Currently, Libra's implementation of `acmn` only supports binary-valued variables.

6.5 `idspn`: Learning Sum-Product Networks

Options:

`-i`	Training data file
`-s`	Data schema (optional)
`-o`	Output SPN directory
`-l1`	Weight of L1 norm [5.0]
`-l`	EM clustering penalty [0.2]
`-ps`	Per-split penalty [10.0]
`-k`	Max sum nodes' cardinalities [5]
`-sd`	Standard deviation of Gaussian weight prior [1.0]
`-cp`	Number of concurrent processes [4]
`-vth`	Vertical cut thresh [0.001]
`-ext`	Maximum number of node extensions [5]
`-minl1`	Minimum value for components' L1 priors [1.0]
`-minedge`	Minimum edge budget for components [200000]
`-minps`	Minimum split penalty for components [2.0]
`-seed`	Random seed
`-f`	Force to override output SPN directory

ID-SPN (`idspn`) [20] learns sum-product networks (SPNs) using direct and indirect variable interactions. The ouput is in the `.spn` format, which is a directory containing `.ac` files for AC nodes and a model file `spac.m` that holds the structure of the SPN. We can convert the `.spn` directory to an `.ac` file using `spn2ac`.

ID-SPN calls `acmn` to learn the AC nodes, so `acmn` must be in the path. The behavior of `acmn` can be adjusted using the `-l1`, `-ps`, and `-sd` options. We can control the number of times that ID-SPN tries to expand an AC node by using the `-ext` option. Setting this parameter to a large number will lead to longer learning times and possibly overfitting. ID-SPN adjusts the parameters of `acmn` based on the number of variables and samples it uses for learning an AC node. We can set a minimum for these options using `-minl1`, `-minedge`, and `-minps`, which helps control overfitting. ID-SPN uses clustering for learning sum nodes. We can control the number of clusters by adjusting the prior over the number of clusters (`-l`) or by setting the maximum number of clusters (`-k`). To learn product nodes, ID-SPN uses a cut threshold (`-vth`), and it supposes that two variables are independent if their mutual information value is less than that threshold. When running ID-SPN on a multicore machine, we can increase the number of concurrent processes using the `-cp` option to reduce the learning time.

```
libra idspn -i msweb.data -o msweb.spn -k 10 -ext 5 -l 0.2
        -vth 0.001 -ps 20 -l1 20
```

(This example will take a long time to run.)

Libra's current implementation of `idspn` only supports binary-valued variables.

6.6 `mtlearn`: Learning Mixtures of Trees

Options:

`-i`	Training data file
`-s`	Data schema (optional)
`-o`	Output SPN directory
`-k`	Number of component [4]
`-seed`	Random seed
`-f`	Force to override output SPN directory

Libra supports learning mixtures of trees (`mtlearn`) [17]. A mixture of trees can be viewed as a sum node and many AC nodes, in which every AC represents a Chow-Liu tree [5]. Therefore, `mtlearn`'s output has the same format as `idspn`, `.spn`. We can convert the `.spn` directory to an `.ac` file using the `spn2ac` command.

For `mtlearn`, the option `-k` determines the number of trees. The performance of `mtlearn` can be sensitive to the random initialization, so it may take several runs to get good results. The seed for the random generator can be specified using `-seed` in order to enable repeatable experiments.

```
libra mtlearn -i msweb.data -s msweb.schema -o msweb-mt.spn -k 10 -v
```

(This example will take several minutes to run.)

6.7 `dnlearn`: DN Structure Learning

Options:

`-i`	Training data file
`-s`	Data schema (optional)
`-o`	Output dependency network
`-ps`	Per-split penalty [0.0]
`-kappa`	Alternate per-split penalty specification [0.0]
`-prior`	Prior counts of uniform distribution [1.0]
`-tree`	Use decision tree CPDs [default]
`-mincount`	Minimum number of examples at each decision tree leaf [10]
`-logistic`	Use logistic regression CPDs [false]
`-l1`	Weight of L1 norm for logistic regression [0.1]

A dependency network (DN) specifies a conditional probability distribution for each variable given its parents. However, unlike a BN, the graph of parent-child relationships may contain cycles. With Libra, we can learn a dependency network with tree-structured conditional probability distributions using `dnlearn`:

```
libra dnlearn -i msweb.data -s msweb.schema -o msweb-dn.dn -prior 1
```

The algorithm is similar to that of Heckerman et al. [8]. As with `bnlearn`, the user can set the prior counts on the multinomial leaf distributions (`-prior`) as well as a per-split penalty (`-ps`) to prevent overfitting.

If all variables are binary-valued, logistic regression CPDs can be used instead (`-logistic`). To obtain sparsity, tune the L1 regularization parameter with the `-l1` option.

```
libra dnlearn -i msweb.data -s msweb.schema -o msweb-dn-l1.dn
        -logistic -l1 2
```

6.8 `dnboost`: DN Structure Learning with Logitboost

Options:

`-i`	Training data file
`-valid`	Validation data
`-s`	Data schema (optional)
`-o`	Output dependency network
`-numtrees`	Number of trees per CPD
`-numleaves`	Number of leaves per tree
`-nu`	Shrinkage [0-1] (0 => line search)
`-mincount`	Minimum number of examples at each decision tree leaf [10]

Dependency networks with boosted decision trees as the conditional model can be learned with `dnboost`. The boosting method is based on the logitboost algorithm [7]. Important parameters include the number of trees to learn for each conditional distribution (`-numtrees`), the number of leaves for each tree (`-numleaves`), and shrinkage (`-shrinkage`). To use validation data for early stopping, specify the file with `-valid`.

In our limited experience, boosted decision trees usually work worse than single decision trees or logistic regression CPDs.

6.9 `dn2mn`: Learning MNs from DNs

Options:

`-m`	Model file (DN)
`-o`	Output Markov network
`-i`	Input data
`-base`	Average over base instances from data
`-bcounts`	Average over base instances from data, using cached counts
`-marg`	Average over all instances, weighted by data marginals [default]
`-order`	Variable order file
`-rev`	Add reverse orders, for symmetry [default]
`-norev`	Do not add reverse orders
`-single`	Average over given orders
`-linear`	Average over all rotations of all orders [default]
`-all`	Average over all possible orderings
`-maxlen`	Max feature length for -linear or -all [default: unlimited]
`-uniq`	Combine weights for duplicate features (may be slower)

The DN2MN algorithm (`dn2mn`) [15] converts a DN into an MN representing a similar distribution. The advantage of DNs is that they are easier to learn; the advantage of MNs is clearer semantics and more inference algorithms. DN2MN allows a learned DN to be converted to an MN for inference, giving the best of both worlds. For consistent DNs, this conversion is exact. If the CPDs are inconsistent with each other, as is typical for learned DNs, then the resulting MN will not represent the exact same distribution. The approximation is often improved by averaging over multiple orderings or base instances (see [15] for more details).

For averaging over multiple base instances, use `-i` to specify a set of input examples, which are either used to learn the marginal distribution over the base instances (`-marg`) or used as the base instances directly (`-base` or `-bcounts`). The default variable order is $(1, 2, \ldots, n)$. To specify a different order or set of orders from a file, use `-order`. To add in reverse orders, use `-rev`. To add in rotations over all orderings (e.g., $(2, 3, \ldots, n, 1)$), use `-linear`. To sum over all possible orderings, use `-all`. To obtain faster performance and smaller models, use `-maxlen` to restrict the use of `-all` or `-linear` to short features. Longer features are then handled with more efficient methods (`-linear` instead of `-all` or `-single` instead of `-linear`).

Default options are `-marg`, `-rev`, and `-linear`, which typically provides a compact and robust approximation, especially when given representative data with `-i`:

```
libra dn2mn -m msweb-dn.dn -o msweb-dn.mn -i msweb.data
```

To sum over a single order instead, use `-base`, `-norev`, and `-single`. The conversion process often leads to many duplicate features. To merge these automatically, use `-uniq`. This may lead to slower performance.

Another method for learning an MN from a DN is to run `mnlearnw`, which, when given a DN model with `-m`, uses the DN features but relearns the parameters from data.

6.10 `mnlearnw`: MN Weight Learning

Options:

`-m`	Input Markov network structure
`-i`	Training data
`-o`	Output model
`-maxiter`	Maximum number of optimization iterations [100]
`-sd`	Standard deviation of Gaussian weight prior [1.0] (¡ 0 = inf)
`-l1`	Weight of L1 norm [0.0]
`-clib`	Use C implementation for speed [true]
`-noclib`	Use OCaml implementation
`-cache`	Enable feature cache (much faster, but more memory) [true]
`-nocache`	Disable feature cache

MN weight learning (`mnlearnw`) selects weights to minimize the penalized pseudo-likelihood of the training data. Both L1 and L2 penalties are supported. The weight of the L1 norm can be set using `-l1` and the standard deviation of the L2 prior can be set using `-sd`. Optimization is performed using the L-BFGS [10] or OWL-QN [1] algorithm. To achieve the best performance for most applications, the internal optimization is implemented in C and caches all features relevant to each example. These optimizations can be disabled with `-noclib` and `-nocache`, respectively. Learning terminates when the optimization converges or when the maximum iteration number is reached (`-maxiter`).

6.11 `acopt`: AC Parameter Learning

Options:

`-m`	Arithmetic circuit to optimize
`-o`	Output circuit
`-ma`	Model to approximate (BN or MN)
`-i`	Empirical distribution to approximate
`-gibbs`	Approximate sampled empirical distribution [false]
`-gspeed`	Number of samples (fast/medium/slow/slower/slowest)
`-gc`	Number of MCMC chains
`-gb`	Number of burn iterations
`-gs`	Number sampling iterations
`-gdn`	Sample from CPD instead of Markov blanket
`-norb`	Disable Rao-Blackwellization
`-seed`	Random seed
`-ev`	Evidence file
`-init`	Initial variable marginals
`-maxiter`	Maximum number of optimization iterations [100]

AC parameters can be learned or otherwise optimized using `acopt`. Given training data (`-i` <*file*>), `acopt` finds parameters that maximize the log-likelihood of the training data. This works for any AC that represents a log linear model, including BNs and MNs. Our implementation uses L-BFGS [10], a standard convex optimization method. The gradient of the log-likelihood is computed in each iteration by differentiating the circuit, which is linear in circuit size. Since this is a convex problem, the parameters will eventually converge to their optimal values. The maximum number of iterations can be specified using `-maxiter`.

Given a BN or MN to approximate (`-ma`) and, optionally, evidence (`-ev`), `acopt` supports two other kinds of parameter optimization.

The first is to minimize the KL divergence between the given BN or MN (optionally conditioned on evidence) and the AC (*i.e.*, $D_{\mathrm{KL}}(\mathrm{AC}||\mathrm{BN})$). This is similar to mean field inference, except that an AC is used in place of a fully factored distribution, and the optimization is performed using L-BFGS instead of message passing.

The second type of optimization (`-gibbs`) approximately minimizes the KL divergence in the other direction, $D_{\mathrm{KL}}(\mathrm{BN}||\mathrm{AC})$ or $D_{\mathrm{KL}}(\mathrm{MN}||\mathrm{AC})$, by generating samples from the BN or MN (optionally conditioned on evidence) and selecting AC parameters to maximize their likelihood. Samples are generated using Gibbs sampling, with parameters analogous to those for the `gibbs` command (Section 7.7). The most important options are `-gspeed` or `-gc`/`-gb`/`-gs` to set the number of samples. Increasing the number of samples yields a better approximation but takes longer to run. This is similar to running `gibbs`, saving the samples using `-so`, and then running `acopt` with `-i`, as described above. However, `acopt -gibbs` is faster since it only needs to compute the sufficient statistics instead of storing and reloading the entire set of samples.

The main application of `acopt` is for for performing approximate inference using ACs, as described by [12]. This can be done by generating samples from a BN (`bnsample`), learning an AC from the samples (`acbn`), and then optimizing the AC's parameters for specific evidence (`acopt`) (see [12]).

7 Inference Methods

7.1 `acve`: AC Variable Elimination

Options:

`-m` Model file (BN or MN)

`-o` Output circuit

AC variable elimination (`acve`) [3] compiles a BN or MN by simulating variable elimination and encoding the addition and multiplication operations into an AC:

```
libra acve -m msweb-cl.xmod -o msweb-cl.ac
```

ACVE represents the original and intermediate factors as algebraic decision diagrams (ADDs) with AC nodes at the leaves. As each variable is summed out, the leaves of the ADDs are replaced with new sum and product nodes. By producing an AC, ACVE builds a representation that can answer many different queries. By using ADDs, ACVE can exploit context-specific independence much better than previous methods based on variable elimination. See Chavira and Darwiche [3] for details.

The one difference between Libra's implementation and the standard algorithm is that its ADDs allow k-way splits for variables with k values. In the standard algorithm, k-valued variables are converted into k Boolean variables, along with constraints to ensure that exactly one of these variables is true at a time. We also omit the circuit node cache, which we find has little effect on circuit size at the cost of significantly slowing compilation.

If no output file is specified, then `acve` does not create a circuit but simply sums out all variables and prints out the value of the log partition function ($\log Z$). To compute conditional probabilities without creating an AC, use `mconvert -ev` to

condition the model on different evidence and `acve` to compute the log partition functions of the conditioned models. The log probability of a query ($\log P(q|e)$) is the difference between the log partition function of the model conditioned on query and evidence ($\log Z_{q,e}$) and the model conditioned only on evidence ($\log Z_e$).

7.2 `acquery`: Exact AC Inference

Options:

`-m`	Input arithmetic circuit
`-ev`	Evidence file
`-q`	Query file
`-sameev`	Use the same evidence for all queries
`-preprune`	Prune circuit for the evidence
`-marg`	Use conditional marginals instead of joint distribution
`-mpe`	Compute most probable explanation (MPE) state
`-mo`	Output file for marginals or MPE

Exact inference in ACs is done through `acquery`, which accepts similar arguments to the approximate inference algorithms. See Darwiche [6] for a thorough description of ACs and how to use them for inference. We provide a brief description of the methods below.

To compute the probability of a conjunctive query, we set all indicator variables in the AC to zero if they are inconsistent with the query and to one if they are consistent. For instance, to compute $P(X_1 = \text{true} \wedge X_3 = \text{false})$, we would set the indicator variables for $X_1 = \text{false}$ and $X_3 = \text{true}$ to zero and all others to one. Evaluating the root of the circuit gives the probability of the input query. Conditional probabilities are answered by taking the ratio of two unconditioned probabilities:

$$P(Q|E) = \frac{P(Q \wedge E)}{P(E)}$$

where Q and E are conjunctions of query and evidence variables, respectively. Both $P(Q \wedge E)$ and $P(E)$ can be computed using previously discussed methods. Evaluating the circuit is linear in the size of the circuit.

We can also differentiate the circuit to compute all marginals in parallel (`-marg`), optionally conditioned on evidence. Differentiating the circuit consists of an upward pass and a downward pass, each of which is linear in the size of the circuit. See Darwiche [6] for more details.

In addition, we can compute the most probable explanation (MPE) state (`-mpe`), which is the most likely configuration of the non-evidence variables given the evidence. When the MPE state is not unique, `acquery` selects one of the MPE states arbitrarily.

We can also use `acquery` to compute the probabilities of configurations of multiple variables, such as the probability that variables two through five all equal 0.

To do this, we need to create a query file that defines every configuration we're interested in. The format is identical to Libra data files, except that we use "*" in place of an actual value for variables whose values are unspecified. `msweb.q` contains three example queries.

```
libra acquery -m msweb-cl.ac -q msweb.q
```

`acquery` outputs the log probability of each query as well as the average over all queries and its standard deviation. With the `-v` flag, `acquery` will print out query times in a second column.

An evidence file can be specified as well using `-ev`, using the same format as the query file:

```
libra acquery -m msweb-cl.ac -q msweb.q -ev msweb.ev -v
```

Evidence can also be used when computing marginals. If you specify both `-q` and `-marg`, then the probability of each query will be computed as the product of the marginals computed. This is typically less accurate, since it ignores correlations among the query variables.

To obtain the most likely variable state (possibly conditioned on evidence), use the `-mpe` flag:

```
libra acquery -m msweb-cl.ac -ev msweb.ev -mpe
```

This computes the most probable explanation (MPE) state for each evidence configuration. If the MPE state is not unique, `acquery` will select one arbitrarily. If you specify a query with `-q`, then `acquery` will print out the fraction of query variables that differ from the predicted MPE state:

```
libra acquery -m msweb-cl.ac -q msweb.q -ev msweb.ev -mpe
```

7.3 `spquery`: Exact Inference in Sum-Product Networks

Options:

`-m`	Input SPN (SPAC) model directory
`-ev`	Evidence file
`-q`	Query file

For models represented in the `.spn` format, we can use `spquery` to find the log probability of queries or the conditional log probability of queries given evidence:

```
libra spquery -m msweb-mt.spn -q msqweb.q -ev msweb.ev
```

For more complex queries, such as computing the most probable explanation (MPE) state or marginals, you must first use `spn2ac` to convert the `.spn` file to the `.ac` format and then use `acquery`, Section 7.2.

7.4 `mf`: Mean Field

Options:

`-m`	Model file (BN, MN, or DN)
`-mo`	Output file for marginals
`-ev`	Evidence file
`-sameev`	Use the same evidence for all queries
`-q`	Query file
`-maxiter`	Maximum number of iterations [50]
`-thresh`	Convergence threshold [0.0001]
`-depnet`	Interpret factors as CPDs in a cyclical dependency network
`-roundrobin`	Update marginals in order instead of skipping unchanged vars

Mean field (`mf`) is an approximate inference algorithm that attempts to minimize the KL divergence between the specified MN or BN (possibly conditioned on evidence) and a fully factored distribution (*i.e.*, a product of single-variable marginals):

```
libra mf -m msweb-cl.bn
libra mf -m msweb-cl.bn -q msweb.q
libra mf -m msweb-cl.bn -q msweb.q -ev msweb.ev -v
```

Note that there is no `-marg` option, since mean field always approximates the distribution as a product of marginals.

Libra's implementation updates one marginal at a time until all marginals have converged, using a queue to keep track of which marginals may need to be updated (see Algorithm 11.7 from [9]). With the `-roundrobin` flag, Libra will instead update all marginals in order. The stopping criteria can be adjusted using the parameters `-thresh` (convergence threshold) or `-maxiter` (maximum number of iterations). Rather than working directly with table or tree CPDs, `mf` converts both to a set of features and works directly with the log-linear representation, ensuring that the compactness of tree CPDs is fully exploited.

Libra's implementation is the first to support mean field inference in DNs, using the `-depnet` option. Since a DN may not represent a consistent probability distribution, the mean field objective is undefined. However, the message-passing updates can still be applied to DNs and they tend to converge in practice [14].

7.5 `bp`: Loopy Belief Propagation

Options:

`-m`	Model file (BN or MN)
`-mo`	Output file for marginals
`-ev`	Evidence file
`-sameev`	Use the same evidence for all queries
`-q`	Query file
`-maxiter`	Maximum number of iterations [50]
`-thresh`	Convergence threshold [0.0001]

Loopy belief propagation (`bp`) is the application of an exact inference algorithm for trees to general graphs that may have loops:

```
libra bp -m msweb-cl.bn -q msweb.q -ev msweb.ev -v
libra bp -m msweb-cl.bn -q msweb.q -ev msweb.ev -sameev -v
```

The `-sameev` option in the second command runs BP only once with the first evidence configuration in `msweb.ev`, and then reuses the marginals for answering all queries in `msweb.q`.

`bp` is implemented on a factor graph, in which variables pass messages to factors and factors pass messages back to variables in each iteration. All factor-to-variable or variable-to-factor messages are passed in parallel, a message passing schedule known as "flooding." For BNs, each factor is a CPD for one of the variables. For factors represented as trees or sets of features, the running time of a single message update is linear in the number of leaves or features, respectively. This allows `bp` to run on networks with factors that involve 100 or more variables, as long as the representation is compact.

7.6 `maxprod`: Max-Product

Options:

`-m`	Model file (BN or MN)
`-mo`	Output file for MPE states
`-ev`	Evidence file
`-sameev`	Use the same evidence for all queries
`-q`	Query file
`-maxiter`	Maximum number of iterations [50]
`-thresh`	Convergence threshold [0.0001]

The max-product algorithm (`maxprod`) is an approximate inference algorithm to find the most probable explanation (MPE) state, the most likely configuration of the non-evidence variables given the evidence. Like `bp`, max-product is an exact inference algorithm in tree-structured networks, but may be incorrect in graphs with loops. Max-product is implemented identically to `bp`, but replacing sum operations with max.

Examples:

```
libra maxprod -m msweb-cl.bn -ev msweb.ev -mo msweb-cl.mpe
libra maxprod -m msweb-bn.bn -q msweb.q -ev msweb.ev -v
```

7.7 `gibbs`: Gibbs Sampling

Options:

`-m`	Model file (BN, MN, or DN)
`-depnet`	Sample from CPD only, not Markov blanket
`-mo`	Output file for marginals
`-so`	Output file for samples
`-marg`	Compute marginals
`-ev`	Evidence file
`-sameev`	Use the same evidence for all queries
`-q`	Query file
`-speed`	Number of samples (fast/medium/slow/slower/slowest)
`-chains`	Number of MCMC chains
`-burnin`	Number of burn-in iterations
`-sampling`	Number sampling iterations
`-norb`	Disable Rao-Blackwellization
`-prior`	Total prior counts
`-seed`	Random seed

Gibbs sampling (`gibbs`) is an instance of Markov-chain Monte Carlo (MCMC) that generates samples by resampling a single variable at a time conditioned on its Markov blanket. The probability of any query can be computed by counting the fraction of samples that satisfy the query. When evidence is specified, the values of the evidence variables are fixed and never resampled. By default, Libra's implementation computes the probabilities of conjunctive queries (*e.g.*, $P(X_1 \wedge X_2 \wedge \neg X_4)$) or marginal queries (*e.g.*, $P(X_1)$), optionally conditioned on evidence. This is potentially more powerful than MF and BP, which only compute marginal probabilities. To compute only marginal probabilities with Gibbs sampling, use the `-marg` option. This is helpful when the specific queries are very rare (such as long conjunctions) but can be approximated well as the product of the individual marginal probabilities.

The running time of Gibbs sampling depends on the number of samples taken. Use `-burnin` to set the number of burn-in iterations (sampling steps thrown away before counting the samples); use `-sampling` to set the number of sampling iterations; and use `-chains` to set the number of repeated sampling runs. For convenience, these parameters can also be set using the `-speed` option which allows arguments of `fast`, `medium`, `slow`, `slower`, and `slowest`, which range from 1000 to 10 million total sampling iterations. All speeds except for `fast` use 10 chains and a number of burn-in iterations equal to 10% of the sampling iterations. Samples can be output to a file using the `-so` option.

By default, Libra uses Rao-Blackwellization to make the probabilities slightly more accurate. This adds *fractional* counts to multiple states by examining the distribution of the variable to be resampled. For instance, suppose the sampler is computing the marginal distribution $P(X_3)$, and that the probability of $X_3 = \text{true}$ is 0.001, given the current state of its Markov blanket. A standard Gibbs sampler adds 1 to the count of the next sampled state and 0 to the other. In contrast, Libra's Rao-Blackwellized sampler adds 0.001 to the counts for X_3 being true and 0.999 to the counts for X_3 being false. This applies both to computing conjunctive queries and marginals. It can be disabled with the flag `-norb`.

`gibbs` can answer queries using either the full joint distribution (the default) or just the marginals (with `-marg`). The latter option is helpful for estimating the probabilities of rare events that may never show up in the samples.

To print the raw samples to a file, use `-so`. To obtain repeatable experiments, use `-seed` to specify a random seed.

Examples:

```
libra gibbs -m msweb-bn.bn -mo msweb-gibbs.marg
libra gibbs -m msweb-bn.bn -q msweb.q -ev msweb.ev -v
libra gibbs -m msweb-bn.bn -q msweb.q -ev msweb.ev -speed medium -v
libra gibbs -m msweb-bn.bn -q msweb.q -ev msweb.ev -marg -sameev
```

Gibbs sampling can be run on a BN, MN, or DN. To force a `.xmod` or `.bif` file to be treated as a DN when sampling, use the `-depnet` flag.

7.8 `icm`: Iterated Conditional Modes

Options:

`-m`	Model file (BN, MN, or DN)
`-mo`	Output file for MPE states
`-depnet`	Treat BN as a DN when computing conditional probabilities
`-ev`	Evidence file
`-sameev`	Use the same evidence for all queries
`-q`	Query file
`-restarts`	Number of random restarts [1]
`-maxiter`	Maximum number of iterations [-1]
`-seed`	Random seed

Iterated conditional modes (`icm`) [2] is a simple hill-climbing algorithm for MPE inference. Starting from a random initial state, it sets each variable in turn to its most likely state given its Markov blanket, until it reaches a local optimum. Multiple random restarts can lead to better optima. In an inconsistent DN, the algorithm is not guaranteed to terminate.

```
libra icm -m msweb-bn.bn -ev msweb.ev
```

8 Utilities

8.1 `mscore`: Model Scoring

Options:

`-m`	Input model (BN, MN, DN, or AC)
`-depnet`	Score model as a DN, using pseudo-log-likelihood
`-i`	Test data file
`-pll`	Compute pseudo-log-likelihood
`-pervar`	Give MN PLL or BN LL for each variable separately.
`-clib`	Use C library for computing PLL
`-noclib`	Use only OCaml code for computing PLL

The command `mscore` computes the log-likelihood of a set of examples for BNs or ACs. For MNs, `mscore` can be used to compute the unnormalized log-likelihood, which will differ from the true log-likelihood by `log Z`, the log partition function of the MN. Using the `-pll` flag, `mscore` will compute the pseudo-log-likelihood (PLL) of a set of examples instead. For DNs, `mscore` always computes the PLL. With `-pervar`, `mscore` provides the log-likelihood or PLL for each variable separately.

8.2 `bnsample`: BN Forward Sampling

Options:

`-m`	Model file (BN)
`-o`	Output file
`-n`	Number of samples to generate [1000]
`-seed`	Random seed

The command `bnsample` can be used to generate a set of independent samples from a BN using forward sampling. Each variable is sampled given its parents, in topological order. Use `-n` to indicate the number of samples and `-seed` to choose a random seed.

8.3 `mconvert`: Model Format Conversion

Options:

`-m`	Input model file (BN, MN, or AC)
`-o`	Output model file
`-ev`	Evidence file
`-feat`	Convert MN factors to features
`-dn`	Convert factors to conditional distributions, producing a DN

The command `mconvert` performs conversions among the AC, BN, DN, and MN file formats, and conditions models on evidence (`-ev`). ACs can be converted to ACs or MNs; BNs can be converted to BNs, MNs, or DNs; MNs can be converted to MNs or DNs; and DNs can be converted to DNs or MNs. Converting from a

DN to an MN will keep the same set of features but not the same distribution; to maintain (approximately) the same distribution, use the `dn2mn` algorithm instead. If an evidence file is specified (using `-ev`), then the output model must be an AC or MN. Conditioning on evidence "reduces" the model by removing portions of the model that are inconsistent with the evidence. If the `-feat` option is specified, then each factor will be a set of features. The `-dn` option converts the factors of a BN or MN to a set of conditional probability distributions, one for each variable given its Markov blanket.

8.4 `spn2ac`: SPN to AC Conversion

Options:

`-m` Input SPN (SPAC) model directory

`-o` Output arithmetic circuit

The command `spn2ac` takes a sum-product network `.spn` and converts it to an equivalent arithmetic file.

8.5 `fstats`: File Information

Options:

`-i` Input file (BN, MN, DN, AC, or data)

`fstats` gives basic information for files of most types supported by Libra.

References

[1] G. Andrew and J. Gao. Scalable training of l1-regularized log-linear models. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, pages 33–40. ACM Press, 2007.

[2] J. Besag. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society, Series B*, 48:259–302, 1986.

[3] M. Chavira and A. Darwiche. Compiling Bayesian networks using variable elimination. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 2443–2449, 2007.

[4] D. Chickering, D. Heckerman, and C. Meek. A Bayesian approach to learning bayesian networks with local structure. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 80–89, Providence, RI, 1997. Morgan Kaufmann.

[5] C. K. Chow and C. N Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14:462–467, 1968.

[6] A. Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM*, 50(3):280–305, 2003.

[7] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28, 1998.

[8] D. Heckerman, D. M. Chickering, C. Meek, R. Rounthwaite, and C. Kadie. Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research*, 1:49–75, 2000.

[9] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Cambridge, MA, 2009.

[10] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(3):503–528, 1989.

[11] D. Lowd and P. Domingos. Learning arithmetic circuits. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, Helsinki, Finland, 2008. AUAI Press.

[12] D. Lowd and P. Domingos. Approximate inference by compilation to arithmetic circuits. In *Advances in Neural Information Processing Systems 23*, Vancouver, BC, 2010. Curran Associates, Inc.

[13] D. Lowd and A. Rooshenas. Learning Markov networks with arithmetic circuits. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2013)*, Scottsdale, AZ, 2013.

[14] D. Lowd and A. Shamaei. Mean field inference in dependency networks: An empirical study. In *Proceedings of the Twenty-Fifth National Conference on Artificial Intelligence*, San Francisco, CA, 2011. AAAI Press.

[15] Daniel Lowd. Closed-form learning of Markov networks from dependency networks. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, Catalina Island, CA, 2012. AUAI Press.

[16] M. Meila. An accelerated Chow and Liu algorithm: Fitting tree distributions to high-dimensional sparse data. In *Proceedings of the Sixteenth International Conference on Machine Learning*, page 249–257. Morgan Kaufmann, 1999.

[17] M. Meila and M. Jordan. Learning with mixtures of trees. *Journal of Machine Learning Research*, 1:1–48, 2000.

[18] K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, Stockholm, Sweden, 1999. Morgan Kaufmann.

[19] H. Poon and P. Domingos. Sum-product networks: A new deep architecture. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI-11)*, Barcelona, Spain, 2011. AUAI Press.

[20] A. Rooshenas and D. Lowd. Learning sum-product networks with direct and indirect interactions. In *Proceedings of the Thirty-First International Conference on Machine Learning*, Beijing, China, 2014. JMLR: W&CP 32.