User Manual for Libra 0.4.0

Daniel Lowd <lowd@cs.uoregon.edu>

July 5, 2011

1 Introduction

A number of graphical model toolkits have been developed over the years, quite a few of which are open source. Libra is the first open source machine learning toolkit to directly support Bayesian networks (BNs) with decision tree conditional probability distributions (CPDs), Markov networks (MNs) with sparse factors, and arithmetic circuits (ACs) as an inference representation.[1] Libra stands for *L*earning and *I*nference in *B*ayesian networks, *R*andom fields and *A*rithmetic circuits. In version 0.4.0, Libra also supports dependency networks. Libra's strength is exploiting context-specific independence to allow exact inference in models with high treewidth. The latest version of Libra can be obtained from `http://libra.cs.uoregon.edu/`. Libra is released under a modified BSD license.

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 tutorial for a step-by-step introduction to using Libra, and the developer's guide for information on modifying and extending Libra.

2 Algorithms

Libra includes implementations of various algorithms for learning and inference with different representations. Representations include Bayesian networks (BNs) with tree or table conditional probability distributions (CPDs); Markov networks (MNs) with factors represented as trees, tables, or sets of (mutually exclusive) conjunctive features; and dependency networks (DNs) with tree or table CPDs.

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

- Mean field inference (`mf`) [11]
- Gibbs sampling (`gibbs`)

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

- Belief propagation (`bp`) [13]
- Max-product (`maxprod`)

[1] The WinMine Toolkit [3] and Ace (http://reasoning.cs.ucla.edu/ace/) were partial inspirations for Libra and offer some overlapping functionality, but neither is open source.

- AC variable elimination (`acve`) [1]

The last method compiles a BN or MN into an arithmetic circuit in which exact inference is efficient.

We also support the following learning methods:

- Chow-Liu algorithm (`cl`) [4]
- Dependency network learning (`dnlearn`) [6]
- Learning BNs with local structure (`aclearnstruct`) [2]
- LearnAC (`aclearnstruct`) [9]

LearnAC 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. It outputs both a BN and an AC. This is the most complex algorithm in the toolkit by far.

Finally, for models that are represented as arithmetic circuits, we support:

- Exact inference (`acquery`)
- Maximum likelihood parameter learning (`acopt`)

A few utility programs round out the toolkit:

- BN forward sampling (`bnsample`)
- Likelihood or psuedo-likelihood model scoring (`mscore`)
- Model conversion and conditioning on evidence (`mconvert`)
- File info, for any supported file type (`fstats`)

Existing functionality is organized around the central theme of using ACs for learning and inference, and comparing them to baseline BN and MN methods.

3 File Formats

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` and `aclearn`), 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
```

For arithmetic circuits, Libra uses a custom file format (.ac) that lists the dimension of each variable in the first line of the file, followed by 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. Each node must appear before all of its parents. The root of the circuit is therefore the last node. After defining all nodes, an arithmetic circuit file optionally describes how its parameters relate to conjunctive features.

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.[2] Note that this is different from the newer XML-based XBIF format, which may be supported in the future.[3]

The second is the WinMine Toolkit XMOD format, which supports both table and tree CPDs.[4] The advantage of tree CPDs over tables is that they can easily express *context-specific* independencies, and allow more compact CPDs for nodes with many parents. Support for tree CPDs is one key advantage of Libra over other BN toolkits.

For Markov networks, Libra defines a custom format. The first line is a comma-separated variable schema. Following are the factor definitions. Different factor types (table, tree, feature set) have different formats. The simplest is a factor for a single features, 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
```

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))
```

[2]Described here: http://www.cs.cmu.edu/~fgcozman/Research/InterchangeFormat/Old/xmlbif02.html.

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

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

```
    (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`".

Libra also supports the Markov network model file format used the by the UAI inference competition, described here: `http://www.cs.huji.ac.il/project/UAI10/fileFormat.php`. However, Libra does not currently support the UAI evidence or result file formats.

4 Using Libra

The best introduction to installing Libra and using it to solve problems is the Libra tutorial, which is available from the Libra home page and included in the standard distribution. In this section, we provide more complete descriptions of the algorithms and their command line options.

4.1 Common Options

Programs in Libra are 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. We now describe common options shared by many or all programs in the toolkit.

The output of the Libra programs is controlled by the following options, available in every program:

`-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 programs that use them:

`-c` <*file*>: Arithmetic circuit

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

`-m` <*file*>: Model file, in MN, XMOD, BIF, or AC format

`-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. Each inference algorithm prints out the conditional log probability of each query given the evidence. To output timing information as well, use the verbose flag, `-v`.

4.2 Mean Field

Mean field (`mf`) is an approximate inference algorithm that attempts to minimize the reverse 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'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 [7]). With the `-roundrobin` flag, Libra will instead update all marginals in parallel. 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.

Our implementation is the first to support mean field inference dependency networks, using the `-depnet` option. Since a dependency network may not represent a consistent probability distribution, the reverse KL divergence is undefined. However, the algorithm can be applied to dependency networks and tends to converge in practice [11].

4.3 Loopy Belief Propagation

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

4.4 Max-Product

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 a tree, but may be incorrect in graphs with loops. Max-product is implemented identically to `bp`, but replacing sum operations with max.

4.5 Gibbs Sampling

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, our 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 well approximated 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 we wish to compute $P(X_3)$. At some point, while resampling X_3, we find that the probability of $X_3 = \text{true}$ given its current Markov blanket is 0.001. After flipping a biased coin, we set X_3 to false. In traditional Gibbs sampling, we would add a count of 1 to the case where X_3 is false and 0 to the case where X_3 is true. In our Rao-Blackwellized version, we add counts of 0.999 and 0.001, respectively. This applies both to computing conjunctive queries and marginals. It can be disabled with the flag `-norb`.

Gibbs sampling can be run on a BN, MN, or dependency network. Dependency networks must be in an accepted BN file format, and the `-depnet` flag must be used.

4.6 AC Variable Elimination

AC variable elimination (`acve`) [1] compiles a BN or MN by simulating variable elimination and encoding the addition and multiplication operations into an 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 can answer many queries simultaneously. By using ADDs, ACVE can exploit context-specific independence much better than previous methods based on variable elimination. See Chavira and Darwiche [1] for details.

The one difference between our implementation and the standard algorithm is that we extend our ADDs to 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.

4.7 Chow-Liu Algorithm

The Chow-Liu algorithm (**cl**) [4] 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 [12].)

4.8 AC Structure Learning

LearnAC (`aclearnstruct`) [9] 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. It outputs both a BN and an AC. This is the most complex algorithm in the toolkit by far.

To learn a BN without learning an AC, use the `-noac` flag. This performs the exact same structure search as before, but without constructing an AC, and with no penalty based on inference complexity. This is useful for obtaining a baseline BN.

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   # only vars 1 and 3 may be parents of 0
1: none except 5 2  # var 1 may only have var 5 or 2 as a parent
2: none             # var 2 may have no parents
5: all              # var 5 may have any parents
```

4.9 DN Structure Learning

A dependency network (DN) specifies a conditional probability distribution (CPD) for each variable given its parents [6]. However, unlike a Bayesian network, the graph of parent-child relationships may contain cycles. As a result, DNs do not always represent consistent probability distributions, making their semantics somewhat problematic. However, they can still be learned from data and queried using Gibbs sampling or the mean field inference algorithm [11].

Dependency networks with tree-structured conditional probabiity distributions can be learned with `dnlearn`. The algorithm used is similar to that of Heckerman et al. [6]. As with `aclearnstruct`, the user can set the prior counts on the multinomial leaf distributions (`-prior`) as well as a per-split penalty (`-ps`) to prevent overfitting. The `-kappa` option is equivalent to a setting a per-split penalty of log kappa.

4.10 AC Parameter Learning

AC parameters can be learned or otherwise optimized using `acopt`. The input circuit is specified with `-c`.

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 Markov networks. Our implementation uses L-BFGS [8], 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 (`-m`) and, optionally, evidence (`-ev`), `acopt` supports two other kinds of parameter optimization. Neither is yet described in the literature.

The first is to minimize the reverse KL divergence between the source network (optionally conditioned on evidence) and the AC (*i.e.*, $D_{\mathrm{KL}}(\mathrm{AC}||\mathrm{BN})$). This is similar to mean field, 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 regular KL divergence, $D_{\mathrm{KL}}(\mathrm{BN}||\mathrm{AC})$ 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 in the `gibbs` program. 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 [10]. The key idea of [10] is to generate samples from a Bayesian network (`bnsample`), learn an AC from the samples (`aclearn`), and then optimize the AC's parameters for specific evidence (`acopt`).

4.11 Exact AC Inference

Exact inference in ACs is done through `acquery`, which accepts similar arguments to the approximate inference algorithms. See Darwiche [5] 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 [5] for more details.

Finally, 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 `-mpe` is used without a query file, it prints out the MPE state for each evidence. When a query file is specified, it prints out the fraction of non-evidence variables that are different between the query state and the inferred MPE state. When the MPE state is not unique, `acquery` selects one of the MPE states arbitrarily.

4.12 Utilities

The program `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 of a set of examples for BNs, MNs, or DNs. Pseudo-log-likelihood is not currently supported for ACs.

The program `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.

The program `mconvert` performs conversions among the AC, BN, and MN file formats, and conditions models on evidence (`-ev`). ACs can be converted to ACs or MNs; BNs can be converted to BNs or MNs; and MNs can be converted to MNs. If an evidence file is specified (using `-ev`), then the output model must be an AC or MN. If the `-feat` option is specified when outputting an MN, then each factor in the MN will be a set of features.

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

References

[1] 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.

[2] 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.

[3] D. M. Chickering. The WinMine toolkit. Technical Report MSR-TR-2002-103, Microsoft, Redmond, WA, 2002.

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

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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 249257. Morgan Kaufmann, 1999.

[13] 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.