Developer's Guide for Libra 0.3.0

Daniel Lowd <lowd@cs.uoregon.edu>

August 1, 2010

1 Introduction

The goals in developing Libra were for the code to be efficient, concise, and have few external dependencies. This document describes the design of the Libra toolkit, and how to use or extend its functionality.

2 Implementation

Libra is implemented in OCaml, a programming language in the ML family that supports procedural, functional, and object-oriented programming paradigms. Libra was developed using OCaml 3.11, but should run under (some) earlier versions as well.

2.1 Why OCaml?

Programs written in OCaml can be compiled to either byte code or native code. Native code performance is much faster than Python and often competitive with C++. This speed is important when implementing algorithms that could run for hours or days. OCaml is statically typed with automatic type inference, which leads to programs that are much more concise and easier to refactor than similar programs written in Java and C++, and safer than programs written in Python. OCaml code has a foreign function interface that could be used to develop Python or C++ interfaces to Libra in the future.

As an experiment, I compared the speed of the Libra Gibbs sampler to a similar one that I wrote in C++ some years ago. To my surprise, I found that the OCaml implementation was actually 23% faster! Theoretically, a carefully optimized C++ program should almost never be slower than one in OCaml. In practice, however, working in OCaml can make it easier to optimize code, leading to a faster implementation given limited development time.

2.2 Programming Style

Libra uses both procedural and functional paradigms. Functional paradigms can lead to faster development and more concise code, but procedural methods are often necessary for obtaining faster code that uses less memory, and make some algorithms easier to implement. The overriding goal in Libra is practicality, not purity. Hence, mutation is used fairly heavily.

We have chosen not to use the object-oriented features of OCaml. (OCaml's variant types are typically the preferred approach when inheritance isn't required.) Named and optional parameters

are an OCaml feature that we are not currently using, but may use at some point in the future. We are not yet using `ocamldoc`, but plan to do so in the future.

Libra does not depend on any external libraries, except those included in the distribution.

3 Organization

All source code is in subdirectories of the `src/` directory. Each subdirectory is devoted to either a library (e.g., **data**, **bn**, etc.) or one or more programs (e.g., **aclearn**, **bnutil**, **bninference**, etc.).

A hierarchy of Makefiles follows the directory hierarchy. The master Makefile is `src/Makefile`, which calls the Makefiles of the subdirectories. Shared definitions are placed in `src/Makefile.shared`, which is included by the sub-Makefiles. Much of the work is done by OCamlMakefile (included), written by Markus Mottl, which automatically detects dependencies and performs compilation and linking. After compilation, all libraries and their interfaces are placed in `src/lib/` and all binaries are placed in `bin/`.

To rebuild all libraries and programs, run:

```
cd src
make clean; make
```

4 Supporting Libraries

The toolkit employs seven libraries: **ext**, extensions to the standard library; **data**, reading and writing of examples; **mn**, Markov networks and log-linear models; **bn**, Bayesian networks; **circuit**, arithmetic circuits; **lbfgs**, the limited-memory BFGS optimization algorithm; and **expat**, the Expat parser. The last two libraries consist of OCaml bindings to the original libraries written in C. The L-BFGS code was originally written in FORTRAN by Jorge Nocedal and ported to C by Naoaki Okazaki. Expat was written by James Clarke, and the OCaml bindings were developed by Maas-Maarten Zeeman. Below we describe in more detail the libraries that were developed internally to Libra.

4.1 Ext Library

The **ext** library is built from a single file, `ext.ml`. It includes miscellaneous utility functions and extensions to the OCaml standard library (including List and Array). If you wish to add a general-purpose functions, such as an `Array.map4` function that performs a `map` over four lists at once, **ext** is the place to implement it. **ext** also includes common arguments, shared mechanisms for working with log files, timer utilities, and more.

4.2 Data Library

The **data** library includes facilities for reading discrete-valued data points from a file, writing them to output streams, reading in variable schemas (represented as an array of variable dimensions), checking that a data point is valid according to a given schema (i.e., having the correct length and a legal value in each dimension), and reading and writing lists of marginal distributions. Future functionality for this library could include support for other file formats, such as ARRF, or more general-purpose manipulation methods.

4.3 Mn Library

The **mn** library reads, writes, and represents Markov networks (BNs) with factors represented as tables, trees, sets of features, or individual features. A dummy "constant" factor is also allowed. Most of the meat is in `factor.ml`, which defines the different types of factors, how to get the factor value for different configurations, how to convert among different types of factors, how to simplify them given evidence, and how to write them to a file or stream.

Some preliminary framework is also present for tied weights, but it is not currently being used.

4.4 Bn Library

The **bn** library reads, writes, and represents Bayesian networks (BNs) with either tree or table conditional probability distributions (CPDs). The basic type is defined in `bnType.ml`, which is then included by `bn.ml`. Files `bifFile.ml` and `xmodFile.ml` support reading and writing Bayesian networks in the Bayesian interchange format (BIF) and WinMine .xmod format, respectively. The reason for the split between `bnType.ml` and `bn.ml` is to avoid circular dependencies, since the file formats depend on the Bayesian network data type, and we wish to make these file formats available through the single interface of `bn.ml`.

Methods for accessing a Bayesian network include computing the conditional probability of a node given a state vector that specifies the values of its parents, computing the probability of a node given its Markov blanket, converting to a Markov network, generating a sample from an empty network (i.e., one with no evidence), and more. Methods for setting CPDs are provided, but are somewhat limited.

Future functionality could include better interfaces for manipulating the structure and parameters of a BN, support for other file formats (such as the more recent XML-based BIF format), and better error handling.

4.5 Circuit Library

The **circuit** library reads, writes, and represents arithmetic circuits. An arithmetic circuit is a directed, acyclic graph in which each interior node is a sum or product of its children, and each leaf is an indicator variable or numeric constant. Arithmetic circuits can be used to represent any log linear model. Arbitrary marginal and conditional probabilities can be computed in linear time in the size of the circuit, but the circuit may have exponential size in the original model. See (Darwiche, 2003) for more information on arithmetic circuits.

The file `node.ml` defines a data type representing a node in an arithmetic circuit, which is one of the following types:

- `TimesNode`: the product of its children
- `PlusNode`: the sum of its children
- `VarNode`(var, value): the indicator variable which is 1 if the specified variable can take on a particular value and 0 otherwise
- `ConstNode`(w): a constant with value e^w
- `DummyNode`: a placeholder for representing a "null" node

It includes basic facilities for constructing, evaluating, and even iterating over a graph of nodes. It also defines sets and hash maps of nodes.

All node definitions are included by `circuit.ml`, which defines the circuit data type and methods for constructing circuits and using them for inference.

5 Modifying Libraries

At this point (version 0.3.0), the library interfaces are not fixed and could change in the future. If you wish to add an algorithm to Libra and you need additional functions in one of the libraries, you are encouraged to add this functionality. If you change a library's implementation, you will need to recompile all programs that depend on the library yourself using commands such as the following.

```
cd src
make clean; make
```

If you modify a library's interface as well, then you must rebuild the interface. For now, the recommended procedure is to just use the compiler's output. For example, to rebuild the **circuit** library interface, execute the following:

```
cd src/circuit
ocamlc -I ../lib -i circuit.ml > circuit.mli
```

The `-I ../lib` directive is important so that the compiler can find the interfaces for other libraries that this library may depend on.

6 Automated Testing

A few automated tests are available in the `test/` directory. To run all of them, use the `runall.sh` script:

```
cd test
./runall.sh
```

These scripts help identify when a modification has broken something.

Each subdirectory of `test/` contains tests pertaining to the programs from the corresponding subdirectory of `src/`. For instance, `test/util` contains tests for `mscore` and `bnsample`. Tests within a single directory are run using the `run.sh` script. For example, to run just the `util` tests, execute the following:

```
cd test/util
./run.sh
```

Note that these are not unit tests. They test the complete functionality of programs, not the individual functions within them. Their main use at this point is to make it easier to identify when a modification has resulted in broken functionality. Currently (version 0.3.0), coverage is low. We expect to add more tests in the future, or possibly adopt a better testing framework (e.g., OUnit).