The Alchemy System for Statistical Relational AI: User Manual

Stanley Kok Parag Singla Matthew Richardson Pedro Domingos
Department of Computer Science and Engineering,
University of Washington

May 2, 2006

1 Introduction

Welcome to the Alchemy system! This manual consists of the following sections:

1. Introduction
2. Installation
3. Quick Start
4. Syntax
5. Predicates and Functions
6. Notes on Code Design

The Alchemy package provides a series of algorithms for statistical relational learning and probabilistic logic inference, based on the Markov logic representation. If you are not already familiar with Markov logic, we recommend that you read the papers *Markov Logic Networks* [5], *Discriminative Training of Markov Logic Networks* [7], and *Learning the Structure of Markov Logic Networks* [3] (mln.pdf, dtmln.pdf, and lsmln.pdf in the `papers/` directory) before reading this manual.

We welcome your feedback on any aspect of the Alchemy package. Please email us at `alchemy@cs.washington.edu` to let us know what you find easy or hard to use, what results you have obtained with Alchemy, the features you wish to have but are not currently provided, and any bugs that you encounter.

Please cite Kok et al. (2005) [4] if you use the Alchemy system.

Please be aware that this is a beta release. Some aspects of the documentation may not be as clear, and some aspects of its usage may not be as user-friendly, as you would like. We have tested the code but some bugs may inadvertently still remain.

This beta release includes:

- Discriminative weight learning
- Generative weight learning
- Structure learning
- MAP/MPE inference (including memory and time efficient)
- MCMC (Gibbs sampling) inference
- Support for native and linked-in functions

In the next release we plan to include:

- EM (to handle ground atoms with unknown truth values during learning)
- More efficient MCMC inference (in memory and time)
- Block inference over variables with mutually exclusive and exhaustive values
- Specification of probabilities instead of weights for formulas in an MLN, and of probabilities for ground atoms in a database
- Specification of indivisible formulas (i.e. formulas that should not be broken up into separate clauses)
- More extensive documentation

Alchemy uses:

- The MaxWalkSat package of Kautz et al. (1997) [2].
- A port from Fortran to C++ of the L-BFGS-B package of Zhu et al. (1997) [8].
- A port from Lisp to C++ of the CNF conversion code of Russell and Norvig (2002) [6].
- The C++ code to compute the inverse cumulative standard normal distribution of Acklam (2003) [1].
- The C++ command line parsing code due to Jeff Bilmes (1992).

The development of Alchemy was partly funded by DARPA grant FA8750-05-2-0283 (managed by AFRL), DARPA contract NBCH-D030010 (subcontracts 02-000225 and 55-000793), NSF grant IIS-0534881, ONR grants N00014-02-1-0408 and N00014-05-1-0313, a Sloan Research Fellowship, and an NSF CAREER Award (both of these to Pedro Domingos). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NSF, ONR, or the United States Government.

2 Installation

This release is meant for the Linux platform. If you wish to use it on Windows, you will have to make some code changes (e.g., use Windows-specific C++ system calls).

We compiled and tested the code using:

- Fedora Core 4
- Bison 2.0
- Flex 2.5.4
- g++ 4.0.2
- Perl 5.2.1

We assume you have placed the `alchemy.tgz` file in the directory `/home`. Unzip and untar the file with the command `tar xvfz alchemy.tgz`. The `/home/alchemy` directory should be created. Henceforth we refer to `/home/alchemy` as `ALCHDIR`.

In `ALCHDIR/src/makefile`, you should set the `BASEDIR` variable to `ALCHDIR`, and ensure that the `GCC`, `FLEX`, and `BISON` variables are correctly set to your g++ compiler, Flex lexical analyzer generator, and Bison parser generator, respectively. (If you do not have Flex and Bison installed on your system, you can obtain them from: `http://www.gnu.org/software/flex` and `www.gnu.org/software/bison`.)

To compile the code, simply type `make depend; make` in the `ALCHDIR/src` directory. The executables will be compiled into `ALCHDIR/bin`. You may wish to change the `BIN` variable in `makefile` to place the compiled executables in a different directory.

3 Quick Start

We provide a simple example in `ALCHDIR/exdata` to help you get started. Throughout the example we assume you are in that directory.

3.1 Input Files

Predicates and functions are declared and first-order formulas are specified in `.mln` files. For example, at the top of `univ.mln`, we declare the predicates `professor`, `student`, etc., and the function `motherOf` as well as the types of their parameters. The first appearance of a predicate or function in a `.mln` file is taken to be its declaration. You can express arbitrary first-order formulas in a `.mln` file (more on this in Section 4). Note that a variable must begin with a lowercase character, and a constant with an uppercase one in a formula. A formula can be preceded by a weight or terminated by a period, but not both. A period signifies that a formula is "hard" (i.e., worlds that violate it should have zero or negligible probability).

Types and constants are also declared in `.mln` files. For example, in `univ-train.mln` we declare the types `person`, `title`, etc., and their associated constants.

Ground atoms are defined in `.db` (database) files. Ground atoms preceded by `!` (e.g., `!professor(Bart)`) are false, by `?` are unknown, and by neither are true. If the closed-world assumption is made for a predicate, its ground atoms that are not defined in a `.db` file are false, while if the open world assumption is made, its undefined ground atoms are unknown. In `univ-train.db`, we specified all the true ground atoms of the predicates `professor`, `student`, etc. Function mappings are defined in the `.db` file, as well.

Linked-in functions and predicates are defined in a separate C++ file. An example file, `functions.cpp` is supplied. There are certain guidelines which must be followed when defining linked-in functions. This is discussed in Section 5.

3.2 Weight Learning

To learn the weights of formulas, run the `learnwts` executable, e.g., `ALCHDIR/bin/learnwts -g -i univ.mln -o univ-out.mln -t univ-train.db -functions univ-train.func`. `-g` specifies that generative learning is to be used. Alternatively, you can use `-d` for discriminative learning, e.g., `ALCHDIR/bin/learnwts -d -i univ.mln -o univ-out.mln -t univ-train.db -functions univ-train.func -ne advisedBy,student,professor`. `-i` and `-o` specify the input and output `.mln` files as `univ.mln` and `univ-out.mln` respectively.

`-t` specifies the `.db` file that is to be used by weight learning. You can specify more than one `.db` file after `-t` in a comma separated list (e.g., `-t univ1.db,univ2.db`). The universe of constants are those that appear in the `.db` files. By default, all the constants are assumed to belong to one database. If this is not the case, you can use the option `-multipleDatabases` to specify that the constants in each `.db` file belong to a separate database, and should not be mixed with those in other `.db` files (e.g, `-t ai.db,graphics.db,systems.db -multipleDatabases`). `-functions` specifies the `.func` file containing the function mappings used by weight learning.

In the current version of Alchemy `.db` files that are used for learning can only contain true or false atoms (no unknowns). If there are constants that do not appear in the `.db` files, you can specify one or more `.mln` files containing the missing constants, and append them after the input `.mln` file, e.g., `-i univ.mln,univ-train.mln`. (You may wish to specify the extra `.mln` files when there are constants that only appear in false ground atoms of a closed-world predicate, or only in unknown ground atoms of an open world predicate. Such ground atoms need not be defined in `.db` files.) By default, unit clauses for all predicates are added to the MLN during weight learning. (You can change this with the `-noAddUnitClauses` option.)

The `-ne` option is used to specify non-evidence predicates. For discriminative learning, at least one non-evidence predicate must be specified. For generative learning, the specified predicates are included in the (weighted) pseudo-log-likelihood computation; if none are specified, all are included.

During weight learning, each formula is converted to conjunctive normal form (CNF), and a weight is learned for each of its clauses. If a formula is preceded by a weight in the input `.mln` file, the weight is divided equally among the formula's clauses. The weight of a clause is used as the mean of a Gaussian prior for the learned weight. If a formula is

terminated by a period (i.e., the formula is a hard one), each of the clauses in its CNF is given a prior weight that is twice the maximum of the soft clause weights. If neither a weight nor a period is specified, a default prior weight is used for each of the formula's clauses; you can specify a default with the `-priorMean` option. If a unit formula contains variables that are followed by the `!` operator, the code automatically creates formulas stating that the variables have mutually exclusive and exhaustive values (see Section 4). The default prior weight for each clause in the CNF of those formulas is 1.5 times the maximum of the soft clause weights. (See Section 6.2 on how to change the default prior weights.)

When multiple databases are used, the CNF of a formula with existentially quantified variables or variables with mutually exclusive and exhaustive values may be different across the databases. This occurs because we have to ground the variables to constants that are different across the databases. When this happens, we learn a weight for the formula rather than for each clause in its CNF.

You can view all the options by typing `ALCHDIR/bin/learnwts` without any parameters. After weight learning, the output `.mln` file contains the weights of the original formulas (commented out), as well as those of its derived clauses.

3.3 Structure Learning

To learn the structure (clauses and weights) of an MLN generatively, you use the `learnstruct` executable, e.g., `ALCHDIR/bin/learnstruct -i univ-empty.mln -o univ-empty-out.mln -t univ-train.db -penalty 0.5`. `learnstruct` uses beam search to find new clauses to add to an MLN. It can start from both empty and non-empty MLNs. When it starts from a non-empty MLN, it does not modify clauses that are derived from existentially quantified formulas or those containing variables with mutually exclusive and exhaustive values. Its options are similar to those of `learnwts`. In addition, it has options for controlling techniques that speed up the search. You can also restrict the types of clauses created during structure learning (see Section 6.3). Type `ALCHDIR/bin/learnstruct` without any parameters to view all options.

3.4 Inference

To perform inference, run the `infer` executable, e.g., `ALCHDIR/bin/infer -i univ-out.mln -e univ-test.db -r univ.results -q advisedBy,student,professor -c -p -mcmcMaxSteps 20000 -functions univ-test.func`.

`-i` specifies the input `.mln` file. In that file all formulas must be preceded by a weight or terminated by a period (but not both). An exception is a unit formula with variables followed by the `!` operator. Such a unit formula can be preceded by a weight, or terminated by a period, or neither. (For such a unit formula, the code automatically creates formulas stating that the variables have mutually exclusive and exhaustive values. See Section 4.) Each formula in the input `.mln` file is converted to CNF. If a weight precedes the formula, it is divided equally among its CNF clauses. If the formula is terminated by a period (i.e.,

the formula is hard), each of its CNF clauses is given a default weight that is twice the maximum soft clause weight. If neither weight nor period is specified for a unit formula with variables followed by `!`, each of its CNF clauses is given a default weight that is 1.5 times the maximum soft clause weight. (See Section 6.2 on how to change the default weights.)

`-e` specifies the evidence `.db` file; a comma-separated list can be used to specify more than one `.db` file. `-functions` specifies the function mappings used for inference. `-r` specifies the output file which contains the inference results.

`-q` specifies the query predicates. You can specify more than one query predicate, and restrict the query to particular groundings, e.g., `-q advisedBy(x,Ida),advisedBy(Ida,Geri)`. (Depending on the shell you are using, you may have to enclose the query predicates in quotes because of the presence of parentheses.) You can also use the `-f` option to specify a file (same format as a `.db` file without false and unknown atoms) containing the query ground atoms you are interested in. (You may use both `-q` and `-f` together.) `-p` indicates that inference using Gibbs sampling is to be used, and the probabilities that the query atoms are true are to be returned. `-mcmcMaxSteps` is used to specify the maximum number of Gibbs sampling steps.

To use MAP inference instead, specify either the `-m` or `-a` option. The former only returns the true ground atoms, while the latter returns both true and false ones. For MAP inference, the output file also contains the weight assigned to a hard ground clause, fraction of hard ground clauses that are satisfied, the sum of their weights, and the sum of the weights of satisfied soft ground clauses. During MAP inference, each hard clause (derived from a hard formula with a terminating period) is given a weight that is the sum of the soft clause weights plus 10. The current version of Alchemy flips clauses with negative weights and distributes the weight among the resulting unit clauses. These are treated as separate clauses, although from a satisfiability standpoint, this is not quite correct; if one unit clause is satisfied, then the entire original clause is satisfied and the complete weight should be used, not just the new weight of the unit clause. Future versions of Alchemy will handle negative weights in this manner.

By default, all first-order predicates are open world. You can control the open/closed-world assumptions with the `-c` and `-o` options. Type `ALCHDIR/bin/infer` without any parameters to see all available options, including those pertaining to Gibbs sampling and MAP inference.

3.4.1 Memory-efficient MAP inference

MAP inference involves the propositionalization of the knowledge base and the running of a satisfiability solver MaxWalkSat [2] on all of the resulting clauses. This can be done with less memory (due to the typical sparseness of relational domains) with the LazySat algorithm. Most clauses are trivially satisfied and do not need to be held in memory. By using the `-lazy` option, the memory-efficient variant is run.

If the MaxWalkSat version is chosen (i.e. `-lazy` is omitted), then Alchemy determines if it can be fully instantiated based on the amount of main memory. Alternatively, the user

can define a maximum limit of memory to be used, in kilobytes, with the option `-mwsLimit`. Alchemy then uses this limit to determine which version should be used.

4 Syntax

Markov Logic Networks are first-order logic formulas with weights, and the core of the syntax of input files used with Alchemy are based on just that, first-order logic. Alchemy also provides various extensions to this syntax and provides a mechanism for computing linked-in and internally implemented predicates and functions. These topics are discussed in the following sections.

4.1 First-Order Logic

You can express an arbitrary first-order formula in an `.mln` file. The syntax for logical connectives is as follows: `!` (not), `^` (and), `v` (or), `=>` (implies), `<=>` (if and only if), `FORALL/forall/Forall` (universal quantification), and `EXIST/exist/Exist` (existential quantification). Operator precedence is as follows: not > and > or > implies > if and only if > forall = exists. Operators with the same precedence are evaluated left to right. You can use parentheses to enforce a different precedence, or to make precedence explicit (e.g., `(A=>B)=>C` as opposed to `A=>(B=>C)`). Universal quantifiers at the outermost level can be omitted, i.e., free variables are interpreted as universally quantified at the outermost level. Quantifiers can be applied to more than one variable at once (e.g., `forall x,y`). The infix equality sign (e.g., `x = y`) can be used as a shorthand for the equality predicate (e.g., `equals(x,y)`).

4.2 MLN Syntax

For convenience, Alchemy provides three additional operators: `*`, `+` and `!`. When predicates in a formula are preceded by `*`, Alchemy considers all possible ways in which `*` can be replaced by `!`, e.g., `*student(x) ^ *professor(x)` is expanded into four formulas:

- `student(x) ^ professor(x)`
- `!student(x) ^ professor(x)`
- `student(x) ^ !professor(x)`
- `!student(x) ^ !professor(x)`

This syntax allows you to compactly express a relational Markov network in Markov logic.

The `+` operator makes it possible to learn "per constant" weights. When a variable in a formula is preceded by `+`, a separate weight is learned for each formula obtained by grounding that variable to one of its values. For example, if the input formula is `hasPosition(x,+y)`, a separate weight is learned for the three formulas:

- `hasPosition(x,Faculty)`
- `hasPosition(y,Faculty_adjunct)`
- `hasPosition(y,Faculty_emeritus)`

If multiple variables are preceded by `+`, a weight is learned for each combination of their values. When there are multiple databases, the type of the variable to which `+` is applied must have the same constants in all the databases. This ensures that the same formulas are generated for each database.

The `!` operator allows you to specify variables that have mutually exclusive and exhaustive values. For example, if the input formula is `position(x, y!)`, we generate three formulas:

- `position(x, Faculty) v position(x, Faculty_emeritus) v position(x, Faculty_emeritus)`
- `position(x, Faculty) => !position(x, Faculty_adjunct) ^ !position(x, Faculty_emeritus)`
- `position(x, Faculty_adjunct) => !position(x, Faculty) ^ !position(x, Faculty_emeritus)`
- `position(x, Faculty_emeritus) => !position(x, Faculty) ^ !position(x, Faculty_adjunct)`

(The first formula states that person x has at least one position, while the last three ensure that person x has at most one position.) Note that the `!` appears after a variable. `!` can only be used in a formula with exactly one non-negated predicate, and can be applied to any number of the predicate's variables.

You can include comments in the `.mln` file with `//` and `/* */` like in C++.

The character `@` and `$` are reserved and should not be used. Due to the internal processing of functions, variable names should not start with `funcVar` and predicate names should not start with `isReturnValueOf`.

A formula in an `.mln` file can be preceded by a number representing the weight of the formula. A formula can also be terminated by a period (`.`), indicating that it is a hard formula. However, a formula cannot have both a weight and a period. In a formula, you can have a line break after `=>`, `<=>`, `^` and `v`.

A legal identifier is a sequence of alphanumeric characters plus the characters `-` (hyphen), `_` (underscore), and `'` (prime); `'` cannot be the first character. Variables in formulas must begin with a lowercase letter, and constants must begin with an uppercase one. Constants may also be expressed as strings (e.g., `Alice` and ``` ``A Course in Logic'' ``` are both acceptable as constants).

Note that Alchemy does not use Skolemization to remove existential quantifiers when converting a formula to CNF. Instead, it replaces existentially quantified subformulas by disjunctions of all their groundings. (Skolemization is sound for resolution, but not sound in general.) For example, when there are only two constants `Alice` and `Bob`, the formula `EXIST`

`x,y advisedBy(x,y)` becomes: `advisedBy(Alice,Alice) v advisedBy(Alice,Bob) v advisedBy(Bob,Alice) v advisedBy(Bob,Bob)`. This may result in very large CNF formulas, and existential quantifiers (or negated universal quantifiers) should be used with care.

Types and constants can be declared in an `.mln` file with the following syntax: `<typename> = { <constant1>, <constant2>, ... }`, e.g., `person = { Alice, Bob }`. You can also declare integer types, e.g., `ageOfStudent = { 18, ..., 22 }`. You may have a line break between constants. Each declared type must have at least one constant. A constant is considered to be declared the first time it is encountered in a type declaration, a formula, or a ground atom (in a `.db` file).

You can include other `.mln` files in a `.mln` file with the `"#include"` keyword. For example, you can include formulas about a university domain in your `.mln` file about a company domain with `#include "university.mln"`.

The executables will print out error messages when they encounter syntax or semantic errors. Each message will indicate the line and column number of the error (lines start from 1 and columns from 0). An error may not be exactly at the indicated column but near it, and an error may also be a result of previous ones (much like compiler error messages).

Predicates and functions play a large role in Alchemy and this topic (including syntax) is covered extensively in the next section.

5 Predicates and Functions

Predicates and functions can be used in three distinct ways in Alchemy: user-defined, linked-in or internally implemented. For most applications, the user provides a finite set of predicates and functions along with some true/false groundings (user-defined); however, Alchemy also allows for the user to define his/her own functions and predicates as C++ code (linked-in) and supplies the user with the most commonly used functions and predicates (internal), as discussed in the previous section. The internal handling of functions and predicates is essentially the same, as predicates can be treated as boolean functions.

In Alchemy, atoms (predicates applied to a tuple of terms) can be certain or uncertain. An atom is certain if it appears in a `.db` (database) file. If a closed-world assumption is made and the atom does not appear, then the atom is assumed to be false. The equality predicate is always treated as an uncertain predicate. In the current version of Alchemy, functions must be certain. In future versions it will be possible to implement uncertain functions and perform inference on them.

5.1 User-defined Predicates and Functions

User-defined predicates and functions is the standard way of using predicates and functions in MLNs. The process consists of declaration, definition and usage. Declaration must occur first. This is done by listing all predicates and functions in the `.mln` file with the following syntax:

- **Predicates**: `<predicatename>(<type1>, ... , <typen>)`
- **Functions**: `<returntype> <functionname>(<type1>, ... , <typen>)`

Predicate definitions, or ground atoms, are defined in a `.db` (database) file. Each line in the file can have one or more ground atoms. Empty lines are permitted. False ground atoms are specified with a preceding `!` (e.g., `!advisedBy(Alice,Bob)`), and unknown ones are specified with a preceding `?` (e.g., `?advisedBy(Alice,Bob)`). True ground atoms are specified without any preceding symbol (e.g., `advisedBy(Alice,Bob)`). Note that if a closed-world assumption is made for a predicate, you only need to specify the true ground atoms in a `.db` file. All other unspecified ground atoms are false. (You can specify the false ground atoms too, if you wish.) Likewise, if an open world assumption is made, you only need to specify the true and false ground atoms. All other unspecified ground atoms are unknown. (Again, you can specify the unknown ground atoms, if you wish.)

Similarly, the user supplies function definitions in a `.db` file. Each line contains exactly one definition of the form `<returnvalue> = <functionname>(<constant1>, ... , <constantn>)`, e.g., `Alice = motherOf(Bob)`. The mappings given are assumed to be the only ones present in the domain.

5.2 Internal Functions and Predicates

Alchemy implements several internal predicates and functions. These are widely used operators such as those from arithmetic, string concatenation, etc. The common arithmetic and comparison operators found in most programming languages can also be used in infix notation. Note, the equality predicate, introduced in the previous section, is not internally computed, but rather it is something you can do inference over and can be used with any type. Here is a list of the internally implemented predicates and functions with the infix notation where available:

Internal Predicates		
Symbol	**Declaration**	**Explanation**
>	greaterThan(int, int)	Tests if first argument is greater than the second
<	lessThan(int, int)	Tests if first argument is less than the second
>=	greaterThanEq(int, int)	Tests if first argument is greater than or equal to the second
<=	lessThanEq(int, int)	Tests if first argument is less than or equal to the second
(none)	substr(string, string)	Tests if first argument is a substring of the second

Internal Functions		
Symbol	**Declaration**	**Explanation**
(none)	int succ(int)	Returns the successor of the argument (+1)
+	int plus(int, int)	Returns the addition of the two arguments
-	int minus(int, int)	Returns the first argument minus the second argument
*	int times(int, int)	Returns the multiplication of the two arguments
/	int dividedBy(int, int)	Returns the first argument divided by the second argument
%	int mod(int, int)	Returns the remainder of the first argument divided by the second argument
(none)	string concat(string, string)	Returns the concatenation of the two arguments

The types of the variables used in internal functions and predicates (and the return type of functions) are determined by the parser as it encounters each formula. This information is used, for example, to count the number of groundings during inference or learning. If the type can not be determined, an error is thrown and Alchemy exits. For example, if the formula `name(x) ^ substr(y,x) => name(y)` is given, the type of variables `x` and `y` can be determined from the declaration of the predicate `name`.

5.3 Linked-In Functions and Predicates

If a predicate or function is not internally implemented in Alchemy, but you wish to use it, then you can define this as a piece of C++ code. The file `ALCHDIR/exdata/functions.cpp` serves as a template for building linked-in functions and predicates. Here are some basic guidelines:

- C++ functions implementing predicates must take only arguments of type `string` and return type `bool`.
- C++ functions implementing functions must take only arguments of and return type `string`.
- All functions and predicates must be between `extern "C"` { and } (see `functions.cpp`. Otherwise, the function names will be mangled by the C++ compiler and a dynamic look-up is not possible.
- g++ 4.0.2 and the glibc library are necessary (other versions have not been tested).

In order to use linked-in functions in an MLN, the functions have to be declared and the location of the C++ file has to be disclosed. The location of the file is arbitrary; however, it must be made available to Alchemy. Here are the steps:

- Include the C++ file in your .mln file, i.e. `#include "functions.cpp"` (use the absolute path).

- Declare the predicate or function just as you usually would, i.e. `number max(number, number)`.
- Declare an interval or enumeration of constants for the types used, i.e. `number={1,...5}`.

The name of the types is arbitrary; however, the user is responsible for ensuring they can be converted to integers, etc. in the function itself, if needed. When the `include` statement is encountered, the C++ file is compiled and a shared object file `functions.so` is put in the current directory. This file is used to dynamically call the linked-in functions and predicates.

6 Notes on Code Design

The C++ source code found in `ALCHDIR/src` is divided into six directories: `util/`, `parser/`, `logic/`, `learnwts/`, `learnstruct/` and `infer/`. Most of the code is found in `.h` files for convenient inlining. We avoided the use of polymorphism, since virtual functions are not inlined and we would like to have as much inlining as possible for the code to run quickly. Most of the `.h` files have names that are the same as those of the classes they contain.

6.1 Utilities

The `util` directory contains "utility" classes. `Argument` is a class used to parse command line arguments. `Array` is a template class representing an array, and is used widely in the code. `HashArray` is similar to an `Array` except that it is backed up by a map so that its elements are unique. `HashList` is similar to a `HashArray` except that it is a list implementation. In `hashint.h` and `hashstring.h` are the definitions of `HashArrays` containing `int`s and `string`s. `ArraysAccessor` allows you to iterate through all combinations of items in several arrays. Both `DualMap` and `ConstDualMap` map `int`s to `string`s and vice versa. They are mainly used by `Domain` in `logic/` to hold predicates, types etc. `StrInt` is a data structure used by `DualMap` and `ConstDualMap`. `MeanVariance` is used to compute the mean and variance of a set of numbers. `MultDArray` represents a multi-dimensional array. `PowerSet` generates the powerset of `{0...n}` except the null set. `Timer` measures user time in seconds, and contains a function to print time. `util.h` is used to contain commonly used functions that can be shared across modules. `Random` is a random number generator.

6.2 Parser

In the `parser/` directory, `follex.y` and `fol.y` are the input files for Flex (lexical analyzer) and Bison (parser generator) respectively. `fol.y` contains the grammar rules that are used to parse first-order logic formulas, and the code that fires when each rule is encountered. `folhelper.h` contains our variables and functions that are used in `fol.y` and `follex.y`. All Flex and Bison variables and functions begin with the characters `yy`. Using a similar convention, all of our variables and functions that are used in `follex.y`, `fol.y` and `folhelper.h` begin with `zz`. The main function is `runYYParser()` that parses a `.mln` file

and creates an MLN and Domain (see Section 6.3 below). If you want to add variables to be used in fol.y or folhelper.h, please see the note at the top of folhelper.h. You can also change the default weights given to hard clauses and clauses derived from formulas containing mutually exclusive and exhaustive values by setting HARD_WEIGHT_MULTIPLIER/HARD_WEIGHT and EXIST_UNIQUE_WEIGHT_MULTIPLIER/EXIST_UNIQUE_WEIGHT at the top of folhelper.h. StrFifoList is a list used in fol.y to hold tokens in the order that they are extracted by Flex. ListObj contains the algorithm to convert a first-order formula to CNF. It approximates lisp in its use of lists to represent a prefix form of first-order logic. replacefolcpp.pl is a perl script that replaces certain code in fol.cpp (generated by Bison from fol.y) so that it is C++ compliant. If you are using a version of Bison that is less than 2.0, you may have to uncomment the lines at the bottom of the file. For debugging purposes, you can set the variables follexDbg (in follex.y) and folDbg (in fol.y) to see the order in which tokens are extracted, as well as the order in which the grammar rules are executed.

6.3 Logic

The logic/ directory contains classes related to first-order logic. PredicateTemplate represents a predicate declaration, while a Predicate is its definition. Likewise for FunctionTemplate and Function. Observe that the code for Predicate and Function is similar, and we could have made one the superclass of the other. However, we avoided polymorphism for the sake of inlining their functions. A Term represents a constant, a variable or a function. A Predicate contains one or more Terms. Clause is an array of Predicates, and contains the important functions for counting the number of true groundings of a clause, and for finding unknown ground clauses. ClauseFactory creates clauses for structure learning. It includes a function validClause() in which you can specify rules to restrict the kinds of clauses created. ClauseSampler contains an algorithm that estimates the number of true groundings of a clause by sampling the clause's groundings. It uses TrueFalseGroundingsStore to store groundings of predicates. MLN represents a Markov Logic Network with a set of Clauses. clausehelper.h and mlnhelper.h contains the auxiliary data structures used by Clause and MLN respectively. Database provides the truth values of ground Predicates (ground atoms), and keeps the truth values of all ground atoms in memory. GroundPreds is a data structure for holding ground Predicates, and is mainly used for testing purposes. A Domain contains the declared types, constants, predicates, and functions, and provides information about them (e.g., the number of constants of a type). It also holds a pointer to a Database.

6.4 Weight Learning

The learnwts directory contains code for learning the weights of formulas. The mainline is in learnwts.cpp. If you do not want to print the clauses as their number of true groundings are being counted during generative learning, you can set the variable PRINT_CLAUSE_DURING_COUNT to false at the top of learnwts.cpp. learnwts.h contains functions used in learnwts.cpp that can be shared with other modules. PseudoLogLikelihood

computes the (weighted) pseudo-log-likelihood given the constants in one or more `Domain`s, and clauses in an `MLN`. `LBFGSB` is an optimization routine that finds the optimal weights, i.e., the weights that give the highest (weighted) pseudo-log-likelihood. `VotedPerceptron` contains the algorithm for discriminative learning. `IndexTranslator` is used to translate between clause weights and the weights that are optimized. It is required when the CNF of a formula is different across multiple databases, e.g., when the formula has existentially quantified variables, or variables with mutually exclusive and exhaustive values.

6.5 Structure Learning

The `learnstruct/` directory contains code for the generative learning of MLN structure. The mainline is in `learnstruct.cpp`. `structlearn.h` contains most of the structure learning code. `structlearn.cpp` contains the code that handles formulas with variables that are existentially quantified, or have mutually exclusive and exhaustive values.

6.6 Inference

The `infer/` directory contains code for performing inference. The mainline is in `infer.cpp`. `infer.h` contains functions used in `infer.cpp` that can be shared with other modules. `GroundPredicate` and `GroundClause` are the counterparts of `Predicate` and `Clause` in `logic/`. We created separate classes for inference in order to save space since most of the instance variables in `Predicate` and `Clause` are not needed during inference, and inference requires us to ground the MLN to create a Markov random field that may take up a lot of memory. `MRF` represents the Markov random field and contains the code for Gibbs sampling. `GelmanConvergenceTest` is used to determine convergence during burn-in, and `ConvergenceTest` is used to determine convergence during Gibbs sampling. `MRF` also performs MAP inference by calling an external executable `maxwalksat`. `maxwalksat.h` contains a wrapper class for that executable. `mwsloc.pl` is a perl script that writes in `maxwalksat.cpp` the location of the `maxwalksat` executable. Memory efficient MAP inference is performed in the class LazyWalkSat (`lazywalksat.h/lazywalksat.cpp`), based on the MaxWalkSat package of Kautz et al. (1997) [2]. This class uses `lwinfo.h` as its interface to the database and `lwutil.h` for various utilities.

References

[1] P. J. Acklam. An algorithm for computing the inverse normal cumulative distribution function. 2003. http://home.online.no/~pjacklam/notes/invnorm/impl/misra/normsinv.html.

[2] H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. In D. Gu, J. Du, and P. Pardalos, editors, *The Satisfiabil-*

ity Problem: Theory and Applications, pages 573–586. American Mathematical Society, New York, NY, 1997. http://www.cs.washington.edu/homes/kautz/walksat/.

[3] S. Kok and P. Domingos. Learning the structure of Markov logic networks. In *Proceedings of the Twenty-Second International Conference on Machine Learning*, pages 441–448, Bonn, Germany, 2005. ACM Press.

[4] S. Kok, P. Singla, M. Richardson, and P. Domingos. The Alchemy system for statistical relational AI. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 2005. http://www.cs.washington.edu/ai/alchemy/.

[5] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 2005. To appear.

[6] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapter 8. Prentice Hall, Upper Saddle River, NJ, 2002. http://aima.cs.berkeley.edu/lisp/doc/overview-LOGIC.html.

[7] P. Singla and P. Domingos. Discriminative training of Markov logic networks. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 868–873, Pittsburgh, PA, 2005. AAAI Press.

[8] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization. *ACM Transactions on Mathematical Software*, 23(4):550–560, 1997. http://www.ece.northwestern.edu/~nocedal/lbfgsb.html.