User Manual of Tuffy 0.3

AnHai Doan Feng Niu Christopher Ré Jude Shavlik Ce Zhang
University of Wisconsin-Madison
{anhai, leonn, chrisre, shavlik, czhang}@cs.wisc.edu

May 1, 2011

Contents

1 Overview

TUFFY is an efficient Markov logic network inference engine.

Markov logic networks (MLNs) [7, 1] have emerged as a powerful framework that combines statistical and logical reasoning; they have been applied to many data intensive problems including information extraction, entity resolution, text mining, and natural language processing. Based on principled data management techniques, TUFFY is an MLN engine that achieves scalability and orders of magnitude speedup compared to prior art implementations. It is written in Java and relies on PostgreSQL. For a brief introduction to MLNs and the technical details of TUFFY, please see our VLDB 2011 paper [4].

When designing and developing TUFFY, we used ALCHEMY[1] as a reference system. Thus, users who have experiences with ALCHEMY should be able to pick up TUFFY easily.

The current version (0.3) of TUFFY is capable of the following MLN tasks:

- **MRF partitioning**, a technique that can result in dramatically improved result quality [4];
- **MAP inference**, where we want to find out the most likely possible world;
- **Marginal inference**, where we want to estimate marginal probabilities;
- **Weight learning**, where we want to learn the weights of MLN rules given training data.

Furthermore, TUFFY also provides the following functionalities beyond the realm of MLNs:

- **Datalog**: In addition to MLN rules, you can also execute Datalog rules in TUFFY.
- **Functions**: TUFFY comes with a library of common numeric/string/boolean functions, which can be used inside an MLN rule. In particular, you can perform arithmetic manipulation and comparison in MLN rules.
- **Predicate scoping**: Sometimes even grounding the atoms of one predicate would blow up your RAM. On the other hand, it's often the case that you only care about a particular subset of the exhaustive set of ground atoms. This feature allows you to explicitly specify the atoms you are interested in so that your program becomes runnable again..

This manual contains an installation guide, a usage guide, and a description of the internals of TUFFY. Appendix A lists built-in functions and operators; Appendix B lists the definition of some terminologies. For more information, please visit the project website at `http://research.cs.wisc.edu/hazy/tuffy/`.

2 Installation

TUFFY requires a Java virtual machine and PostgreSQL, both of which are free software. This section guides you on how to install and configure those prerequisites as well as TUFFY itself. Figure 1 lists the URLs where you can get them.

[1] `http://alchemy.cs.washington.edu/`

Java	http://www.java.com/en/download/manual.jsp
PostgreSQL	http://www.postgresql.org/download/
Tuffy	http://research.cs.wisc.edu/hazy/tuffy/

Figure 1: Links of required software

2.1 Installing Prerequisites

Install Java Runtime Environment If you don't have JRE 1.6 (or higher) on your machine, please go to http://www.java.com/en/download/manual.jsp to download and install it, following the steps below.

Install and Configure PostgreSQL If you don't have PostgreSQL 8.4 (or higher) on your machine, please download and install it.

1. Let `PG_DIST` be the location where you unpacked the source distribution. Let `PG_PATH` be the location where you want to install PostgreSQL. Run the following commands:

   ```
   cd PG_DIST
   ./configure --prefix=PG_PATH
   gmake; gmake install
   cd PG_PATH/bin
   initdb -D PG_PATH/data
   postmaster -D PG_PATH/data &
   createdb test
   psql test
   ```

   Among the commands above, `initdb` initializes a directory to store databases; `postmaster` launches the PostgreSQL server daemon (at default port 5432); `createdb` creates a new database (with name 'test' in our example); and `psql` takes you to the interactive console of PostgreSQL, where you can issue whatever SQL queries you like. Type '`\h`' for help, and '`\q`' to quit.

2. Install additional modules; we only need '`intarray`' and '`intagg`':

   ```
   cd PG_DIST/contrib/intarray
   gmake; gmake install
   cd PG_DIST/contrib/intagg
   gmake; gmake install
   ```

3. Create a PostgreSQL superuser with name, say, "tuffer". Look here, or simply run the following command:

   ```
   PG_PATH/bin/createuser -s -P tuffer
   ```

   You will be promoted for a password. Henceforth let's assume the password is "strongPasswoRd".

4. Create a database with name, say, "tuffydb". Look here, or simply run the following commands:

   ```
   PG_PATH/bin/createdb tuffydb
   PG_PATH/bin/createlang plpgsql tuffydb
   ```

2.2 Configuring Tuffy

Download Tuffy from the project website. After unpacking, you should see a configuration file named "tuffy.conf" (Figure 2; lines starting with '#' are comments). The four parameters in it are all you need to configure Tuffy. They control the PostgreSQL connection and the (temporary) working directory to be used by Tuffy. Tuffy will write temporary data to both places at runtime and clean them up when finishing. The default values are shown in Figure 2.

```
# JDBC connection string; must be PostgreSQL
db_url = jdbc:postgresql://localhost:5432/tuffydb

# Database username; must be a superuser
db_username = tuffer

# The password for db_username
db_password = strongPasswoRd

# The working directory; Tuffy may write sizable temporary data here
dir_working = /tmp/tuffy-workspace
```

Figure 2: Default Configuration File

3 Usage

In this section, we first build up an impression of Tuffy with a simple example, and then provide more details on how to run inference in Tuffy. In §3.4, we discuss how to use Tuffy to learn weights of MLN rules given a training dataset.

3.1 Hello World

We use a simple example to illustrate how to use Tuffy. The input/output formats as well as command options are mostly compatible with Alchemy.

Input Conceptually, the input consists of three parts: the MLN program, the evidence, and the query (Figure 3). The program and the evidence are each specified in one or more plain text files. The query may be specified with text files and/or the command line.

The input format of Tuffy is compatible with that of Alchemy . For example, a program file consists of predicate definitions and weighted clauses (Figure 3a). Terms that begin with a lower-case letter are variables, while those beginning with an upper-case letter are constants. Existential quantifiers are supported. We do introduce a syntactic sugar, though: A predicate definition preceded by "`*`" is considered to have the closed world assumption (CWA); i.e., all ground atoms of this predicate not listed in the evidence are false. In Figure 3a, we make such an assumption with the predicate `Friends`. Alternatively, one may also specify CWA predicates with the "`-cw`" option in the command line.

```
// Predicate definitions
*Friends(person, person)
Smokes(person)
Cancer(person)
// Rule definitions
0.5  !Smokes(a1) v Cancer(a1)
0.4  !Friends(a1,a2) v !Smokes(a1) v Smokes(a2)
0.4  !Friends(a1,a2) v !Smokes(a2) v Smokes(a1)
```

(a) MLN program

```
Friends(Anna, Bob)
Friends(Anna, Edward)
Friends(Anna, Frank)
Friends(Edward, Frank)
Friends(Gary, Helen)
!Friends(Gary, Frank)
Smokes(Anna)
Smokes(Edward)
```

(b) Evidence

```
Cancer(x)
```

(c) Query

Figure 3: Sample Input to TUFFY

An evidence file is a list of ground atoms. Figure 3b shows a sample evidence file. Each atom is deemed to be true unless it's preceded by "`!`". In this particular example, the line "`!Friends(Gary, Frank)`" is superfluous since the predicate `Friends` is closed.

Suppose that our query is to determine the set of people who have cancer, which can be specified with the atom "`Cancer(x)`" (Figure 3c). We can specify this query in either of the following two ways. First, we may use the command line option "`-q Cancer(x)`" or "`-q Cancer`". Second, we may write a text file with the content "`Cancer(x)`" and specify "`-queryFile` *`fileName`*" in the command line, where *`fileName`* is the path to the text file. Sometimes the query is more complex than just a couple of atoms; in this case, the second approach can be convenient.

Inference There are two types of inference with MLNs: MAP inference and marginal inference. By default, TUFFY runs MAP inference to produce a most likely world; but you can use "`-marginal`" to instruct TUFFY to run marginal inference instead. More concretely, suppose you want to find the most likely set of people with cancer – which is MAP inference – then run the following command:

```
java -jar tuffy.jar -i samples/smoke/prog.mln -e samples/smoke/evidence.db
-queryFile samples/smoke/query.db -r out.txt
```

which would produce terminal output similar to Figure 6a. We have divided the screenshot into five segments to summarize the steps involved in this invocation of TUFFY: Setup, Parsing, Grounding, Inference, and Clean-up. To better understand the algorithms involved in Grounding and Inference, please read on and/or refer to the VLDB paper [4]. The inference result is written to "out.txt", whose content should resemble Figure 6a. This world actually has a cost 0; i.e., all rules are fully satisfied. For example, one can verify that every smoker has cancer, complying with the first rule in Figure 3a. Note that the output is a *projection* of the MAP world on the query. For example, in this particular case, even though the inference algorithm has determined that Bob and Frank are also smokers, these facts are not shown in the output file. Had the query contained both `Cancer(x)` and `Smokes(x)`, the output would also contain `Smokes(Bob)` and `Smokes(Frank)`.

Alternatively, you may want to estimate the probability that each person has cancer – which is marginal inference. In this case, run

```
java -jar tuffy.jar -marginal -i samples/smoke/prog.mln -e samples/smoke/evidence.db
-queryFile samples/smoke/query.db -r out.txt
```

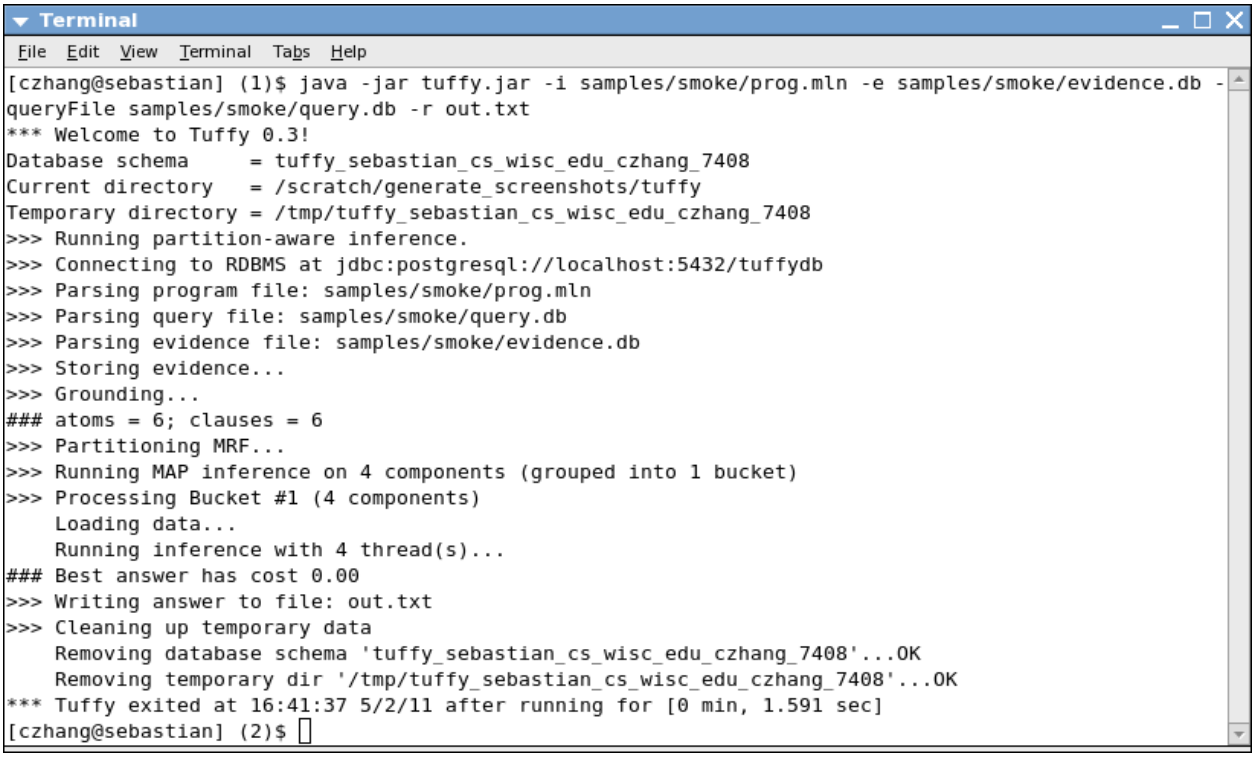

Figure 4: Terminal Output of MAP Inference using Tuffy

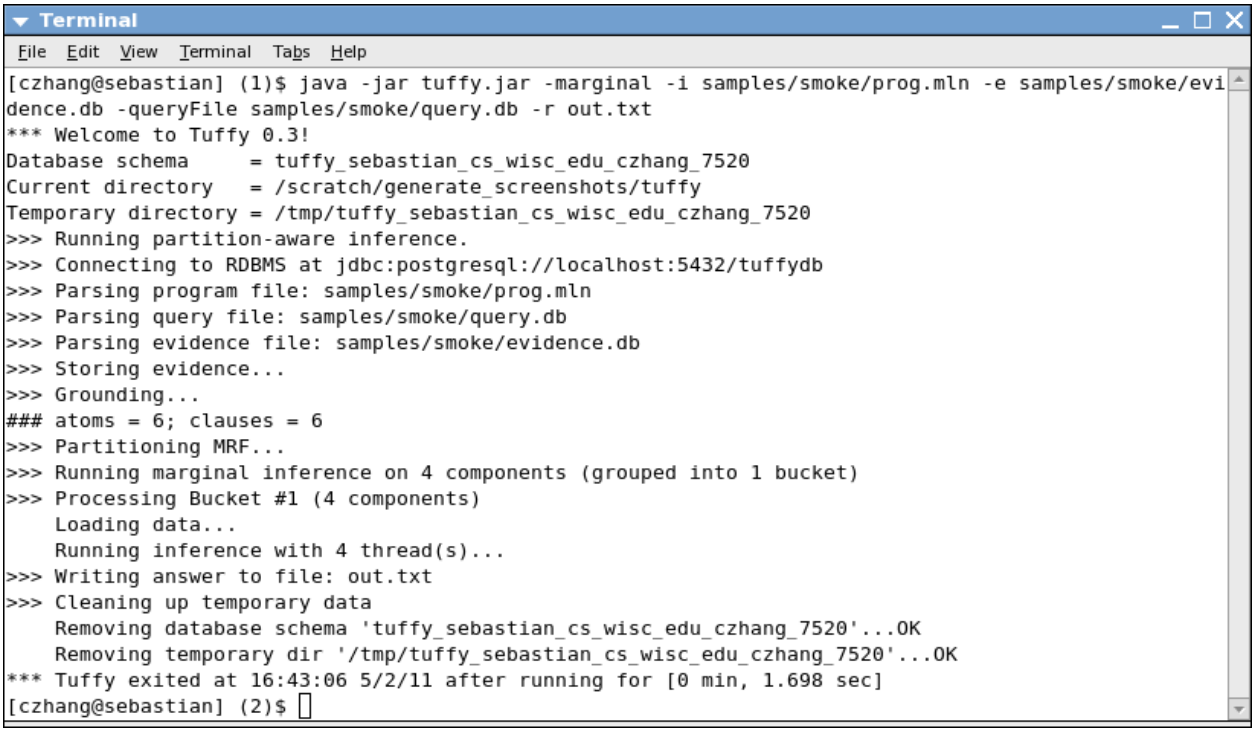

Figure 5: Terminal Output of Marginal Inference using Tuffy

```
Cancer(Anna)
Cancer(Bob)
Cancer(Edward)
Cancer(Frank)
```

(a) Output of MAP inference

```
0.75 Cancer(Edward)
0.65 Cancer(Anna)
0.50 Cancer(Bob)
0.45 Cancer(Frank)
```

(b) Output of marginal inference

Figure 6: Sample Output of Tuffy

which would produce a screenshot like Figure 5 and an output file "out.text" with content similar to Figure 6b. The numbers before each atom are the estimated marginal probabilities. Atoms absent from the output have probability 0.

3.2 Tuffy Programs

A Tuffy program is a text file consisting of two sections: predicates and rules. Besides MLN rules – which are first-order clauses with weights – Tuffy also supports two other types of rules: Datalog rules and *scoping rules*.

Predicate Schema The first section of a Tuffy program – the (predicate) schema – consists of a list of predicate declarations. For example, lines 2-4 in Figure 3a specify three predicates: `Friends`, `Smokes`, and `Cancer`. Each predicate declaration specifies a predicate name with a list of argument *types*. Each type is

```
    p(x,y), q(y,z) => s(x,z).
 2  !p(x,y) v !q(y,z) v s(x,z)
-1   p(x,y) => !q(y,z) v s(x,z)
 3   EXIST y student(x) => advisedBy(x,y)
```

Figure 7: MLN rules

supported by a set of constants that is populated by the corresponding predicate arguments in the rules and the evidence. For example, in the example above, we have only one type (i.e., `person`) containing a set of person names ({`Anna`, `Edward`, `Helen`, ...}). Optionally, you may give each argument a unique[2] name that is more meaningful and that can be used to differentiate arguments of the same type; e.g., `Friends(person a, person b)`. A predicate declaration may be preceded by an asterisk to indicate that the predicate is *closed* – i.e., tuples of this predicate absent from the evidence file are considered false. In contrast, for an *open* predicate, such tuples are considered to have *unknown* truth values. Full-evidence predicates should always be closed.

Key Constraints Many applications requires key dependencies among the argument of a predicate. For example, we may use the predicate `POS(sentence, position, tag)` for part-of-speech tags. Naturally, we require that each position in a sentence can have only one tag. To enforce this constraint, we can declare `POS(sentence, position, tag!)`. In general, the subset of arguments not annotated with `!` form a *possible-world key*; i.e., true atoms in a possible world cannot have identical values on those arguments.

The second section of a TUFFY program is a list of rules, where each rule may be an MLN rule, a Datalog rule, or a *scoping rule*. Furthermore, those rules may be augmented with *conditions*, which are boolean expressions over the variables and constants in the rule. To construct conditions, TUFFY comes with a library of common numeric/string/boolean functions that are listed in Appendix A.

MLN Rules An MLN rule can be either *soft* or *hard*. Both soft rules and hard rules contain a first-order logic formula. In soft rules, the formula is preceded by a real number (positive or negative), which is called the *weight*; in hard rules, the formula is followed by a period mark, indicating a weight of $+\infty$. The formula may be in clausal form or an implication. Note that we require that all formulas be able to be transformed into the clausal form $\mathtt{l}_1$ `v` $\mathtt{l}_2$ `v` $\cdots$ `v` $\mathtt{l}_n$ where $\mathtt{l}_i$ are literals. As such, an acceptable implication formula must be in the form $\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_m$ `=>` $\mathtt{q}_1$ `v` $\mathtt{q}_2$ `v` $\cdots$ `v` $\mathtt{q}_n$; i.e., the antecedent must be a conjunction and the consequent must be a disjunction. Variables in a formula may be existentially quantified using the keyword `EXIST`. See Figure 7 for some example MLN rules. Note that the formulas in the first three rules are equivalent.

The arguments in a literal may be either a variable or a constant. Variables start with a lower-case letter. A constant either 1) starts with a capital letter; 2) is a number; or 3) is a double-quoted string. A double-quoted string may contain escaped characters, e.g. `\t`, `\"`, and `\n`. Internally, all constants are stored as strings, and dynamic type cast is used to evaluate the built-in numeric/string functions and operators.

Parameterized Weights As a syntactic sugar, TUFFY allows you to embed the rule weight in the rule itself. For example, suppose we want to use multi-class logistic regression (LR) to predicate POS tags. As shown in Figure 8, we can use the relation `Confidence` to store the LR parameters in the field `wgt` (which must be of the built-in type `float_`). Then we can use one MLN rule to express the LR model.

[2] among all arguments of this predicate

```
*Tokens(sentence, position, word)
*Confidence(word, tag, float_ wgt)
POS(sentence, position, tag!)
wgt:  Tokens(s, p, w), Confidence(w, t, wgt) => POS(s, p, t)
```

Figure 8: Parameterized wights for MLN rules

```
3  phrases(pid, str), personName(n), [contains(str,n)] => isaPerson(pid)
2  teamScore(t1, s1), teamScore(t2, s2), s1 > s2 => winner(t1)
8  teamScore(t, s), [s % 2 = 1 AND floor(sqrt(s)) = 0] => goodTeam(t)
users(uid, name, hashedPswd), easyPasswords(p), [md5(hashedPswd) = p] => cracked(name, p).
```

Figure 9: MLN rules with conditions

Conditions Besides literals, an MLN rule may also have *conditions*, which are arbitrary boolean expressions using `NOT`, `AND`, and `OR` to connect atomic boolean expressions. An atomic boolean expression can be an arithmetic comparison (e.g., `s1 > s2 + 4`) or a boolean function (e.g., `contains(name, "Jake")`). Furthermore, the functions can be nested, e.g., `endsWith(lower(trim(name)), "jr.")`. Note that all the variables in a boolean expression must appear in some literal in the same formula. A condition can appear as the last component in either the antecedent (if any) or the consequent, and must be enclosed with a pair of brackets. If a condition is an arithmetic comparison, however, it can appear naked (i.e., without the brackets) after all literals and before the bracketed condition (if any) in either the antecedent (if any) or the consequent. Figure 9 contain some sample MLN rules with conditions.

Datalog Rules Tuffy also supports Datalog rules. For example, in the last rule in Figure 9, if both `users` and `easyPasswords` are evidence predicates, then it's more appropriate to replace it with the Datalog rule as shown in the first line in Figure 10. On the other hand, if `teamScore` is a predicate with uncertainties, then the two `teamScore` rules in Figure 9 cannot be converted to Datalog rules. When executing a Datalog rule, Tuffy issues a SQL query to generate tuples for the head predicate that are absent from the evidence, marking them as true evidence.

A Datalog rule is essentially a conjunctive query. If a body literal is positive (resp. negative), Tuffy takes all true (resp. false) evidence for the corresponding predicate. The SQL query is generated by joining those evidence tuples subject to variable binding and conditions in the body. Sometimes one may wish to join all the materialized tuples of a literal regardless of their truth values; to do that, precede the body literal with `+`.

If there are multiple Datalog rules for the same head predicate, Tuffy takes the union of the query results from each rule. Rules are executed sequentially in the same order as they appear in the program file.

Scoping Rules By default, the complete set of tuples (i.e., ground atoms) for a predicate is the cross product of its argument types. (The truth value of each tuple can be either true, false, or unknown.)

```
cracked(name, p) :- users(uid, name, hashedPswd), easyPasswords(p), [md5(hashedPswd) = p].
phraseSent(phrase, sentence) :- location(article, para, sentence, phrase).
```

Figure 10: Datalog rules

```
isaPerson(art, para, sent, ph) := nounPhrases(ph), location(art, para, sent, ph).
gameWinner(team, game) := teamMentions(article, team), gameMentions(article, game).
```

Figure 11: Scoping rules

However, oftentimes we only intend to model a far smaller set of tuples than the cross product – which is the case, e.g., when there are functional dependencies among the arguments. For example, consider the predicate declaration `isaPerson(article, paragraph, sentence, phrase)` which represents the set of phrases deemed to refer to a person. Suppose the last argument (of type `phrase`) is the key of this predicate; i.e., given an assignment to the last argument, the other three arguments are fixed. Then we'd like there to be a total of $|\{\text{phrases}\}|$ tuples for `isaPerson`. However, unaware of this dependency, by default Tuffy would consider a total of $|\{\text{articles}\}| \times |\{\text{paragraphs}\}| \times |\{\text{sentences}\}| \times |\{\text{phrases}\}|$ tuples, which would explode in no time!

We call this issue the *scoping* problem of MLN predicates. To prevent Tuffy from generating unsensible tuples, the user can provide *scoping rules* to limit the scope of tuples for predicates like `isaPerson`. Scoping rules have the same syntax as Datalog rules, except that `:-` is replaced with `:=`. Tuffy generates the same SQL query as with Datalog rules, but mark the new tuples as unknown (false if the head predicate is closed) instead of true. To mark the new tuples as unknown even if the head predicate is closed, precede the head with `+`.

As with Datalog rules, scoping rules are also executed sequentially. Each time a scoping rule is executed, new tuples non-existent in the (head) predicate table are appended to the predicate table.

3.3 Data Files

In this section, we describe the input format to Tuffy in more details.

3.3.1 Evidence

To perform MLN inference with Tuffy, the user need to provide one or more evidence files with the `-e` command option. Each evidence file consists of a list of atoms. If an atom is preceded by `!`, then this atom is false; otherwise it is true. Those atoms are *hard evidence*.

Soft Evidence In addition to hard evidence, the user may also provide *soft evidence* by preceding an atom with a float number between 0.0 and 1.0. Intuitively, this number can be seen as the prior probability that the atom is true. To incorporate such prior probabilities into the MLN semantics, a soft evidence with probability p is internally treated as an MLN rule with the formula being the same atom, and the log odds weight $\ln \frac{p}{1-p}$.

3.3.2 Queries

The user shall also provide a set of query atoms via one or more query files ("`-queryFile`") and/or via the command line ("`-q`").

3.3.3 Output

The user should specify an output file with "`-r`" or "`-o`", to which the inference results will be written. As we have seen in the `Cancer` example, the output of MAP inference is a list of true ground atoms, while the output of marginal inference is the estimated probabilities of ground atoms (leaving out those with probability 0).

3.4 Weight Learning

Weight learning takes as input a training dataset and an MLN program without weight; it tries to compute the optimal weights of the MLN rules by maximizing the likelihood of the training data. Tuffy implements the Diagonal Newton discriminative learner as described in Lowd and Domingos [3].

Similar to inference, weight learning also takes as input an MLN program, evidence, and query. The difference is that (1) the supplied weights of MLN rules will be ignored by Tuffy; and (2) the evidence should specify a fully instantiated world – i.e., it should provide the "answer" to the query.

The clauses in the MLN program should be assigned a non-zero weight if you do not want Tuffy to ignore them while learning. However, the exact value of these weights does not matter, because Tuffy will learn a better one for you. So just assigning 1s to all of them will be fine.

The query atoms in the query file will not be inferred, instead they mark a subset of evidences to which you want the resulting MLN fits.

All atoms in the evidence file matching your query atoms will be considered as training data. Atoms not matching your query will be considered as regular evidences. You only need to provide positive atoms, and other atoms not appearing will be regarded as negative atoms.

An example can be found in `bench/rc1000/`. In these files, all the `Category(.,.)` atoms in the evidence file will be treated as training data. To learn the weights for this program, run

```
java -jar tuffy.jar -learnwt -i samples/rc1000/prog.mln -e samples/rc1000/evidence.db
-queryFile samples/rc1000/query.db -r out.txt -mcsatSamples 50 -dMaxIter 100
```

The option `-learnwt` indicates that Tuffy will run in the learning mode. The option `-dMaxIter 100` means that Tuffy will run at most 100 iterations. A screenshot of running weight learning can be found in Figure 12.

Figure 13 lists the resulting output file of weight learning on the `rc1000` dataset. This file consists of two weight assignments. The first group is the average weights over all iterations. This is used to combat overfitting. The second group, which is commented by default, is the weights found at the maximum (i.e., at the last iteration). Either of these alternatives can be used as learning result.

Comments: The user is expected to note that, learning is often slower a process than inference. This is because there can be more than hundreds inference invocations while learning. Although decreasing the value of `-mcsatSamples` may reduce the required time of each iteration, setting too small value may need more iterations for the learner to converge. This, therefore, may increase the total learning time or potentially influence the learning quality. We recommend the user to provide smaller MLN world because we find empirically that a well sampled smaller world can achieve similar weights as a larger world using much less time.

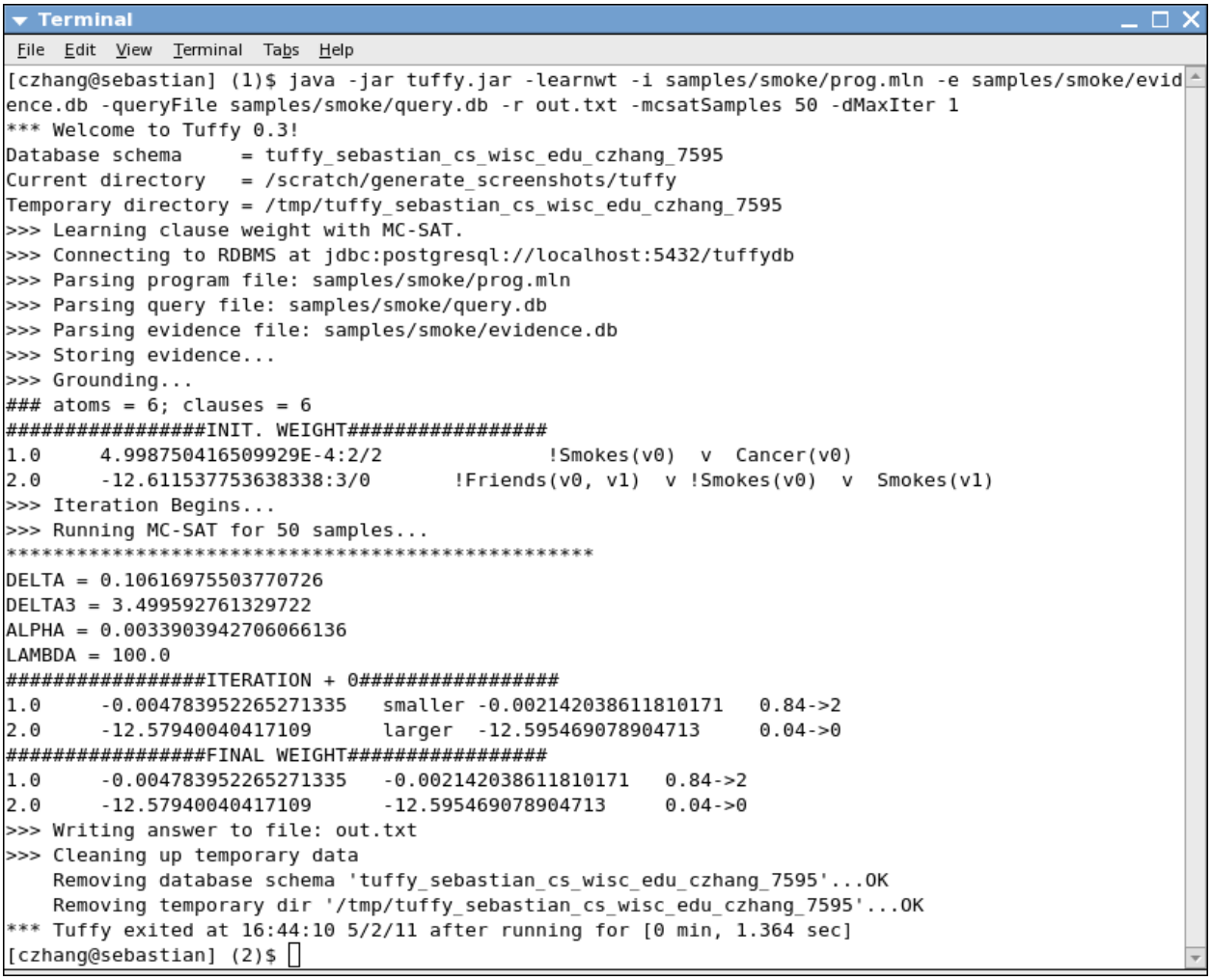

Figure 12: Terminal Output of Weight Learning using Tuffy

```
//////////////AVERAGE WEIGHT OF ALL THE ITERATIONS//////////////
0.158       !wrote(v0, v1)  v !wrote(v0, v2)  v  category(v1, v3)  v !category(v2, v3)  //1.0
0.5338       !refers(v0, v1)  v  category(v1, v2)  v !category(v0, v2)  //2.0
0.6636       !refers(v0, v1)  v  category(v0, v2)  v !category(v1, v2)  //3.0
7.91        sameCat(v0, v1)  v !category(v2, v1)  v !category(v2, v0)  //4.0
-0.0534        category(v0, Networking)  //5.1
-0.8758        category(v0, HumanComputerInteraction)  //5.10
-0.0534        category(v0, Networking)  //5.11
0.8018        category(v0, Programming)  //5.2
0.5143        category(v0, OperatingSystems)  //5.3
-1.5975        category(v0, HardwareandArchitecture)  //5.4
0.466        category(v0, DataStructuresAlgorithmsandTheory)  //5.5
-1.1965        category(v0, EncryptionandCompression)  //5.6
-1.4175        category(v0, InformationRetrieval)  //5.7
-0.2061        category(v0, Databases)  //5.8
1.1559        category(v0, ArtificialIntelligence)  //5.9


//////////////WEIGHT OF LAST ITERATION//////////////
// 0.1518       !wrote(v0, v1)  v !wrote(v0, v2)  v  category(v1, v3)  v !category(v2, v3)  //1.0
// 0.309       !refers(v0, v1)  v  category(v1, v2)  v !category(v0, v2)  //2.0
// 0.4983       !refers(v0, v1)  v  category(v0, v2)  v !category(v1, v2)  //3.0
// 3.3875        sameCat(v0, v1)  v !category(v2, v1)  v !category(v2, v0)  //4.0
// 0.6662        category(v0, Networking)  //5.1
// 0.4361        category(v0, HumanComputerInteraction)  //5.10
// 0.6662        category(v0, Networking)  //5.11
// 1.937        category(v0, Programming)  //5.2
// 1.7944        category(v0, OperatingSystems)  //5.3
// -0.3188        category(v0, HardwareandArchitecture)  //5.4
// 1.6255        category(v0, DataStructuresAlgorithmsandTheory)  //5.5
// 0.1284        category(v0, EncryptionandCompression)  //5.6
// -0.0385        category(v0, InformationRetrieval)  //5.7
// 1.1474        category(v0, Databases)  //5.8
// 2.0739        category(v0, ArtificialIntelligence)  //5.9
```

Figure 13: Sample Result of Tuffy Learning

```
USAGE:
 -activateAll               : Mark all unknown atoms as active during grounding.
 -conf VAL                  : Path of the configuration file. Default='./tuffy.c
                              onf'
 -cw (-closedWorld) VAL     : Specify closed-world predicates. Separate with
                              comma.
 -dMaxIter N                : Max number of iterations for learning. DEFAULT=500
 -db VAL                    : The database schema from which the evidence is
                              loaded.
 -dbNeedTranslate           : Whether we can directly use tables in this DB
                              schema. See -db option.
 -dontBreak                 : Forbid WalkSAT steps that would break hard
                              clauses. DEFAULT=FALSE
 -dribble VAL               : File where terminal output will be written to.
 -dual                      : Run both MAP and marginal inference. Results will
                              be written to $out_file.map and $out_file.marginal
                              respectively.
 -e (-evidence) VAL         : REQUIRED. Input evidence file(s). Separate with
                              comma.
 -gz                        : Compress output files in gzip.
 -help                      : Display command options.
 -i (-mln) VAL              : REQUIRED. Input MLN program(s). Separate with
                              comma.
 -keepData                  : Keep the data in the database upon exiting.
 -learnwt                   : Run Tuffy in discriminative weight learning mode.
 -lineHeader VAL            : Precede each line printed on the console with
                              this string.
 -marginal                  : Run marginal inference with MC-SAT.
 -maxFlips N                : Max number of flips per try. Default=[#atoms] X 10
 -maxTries N                : Max number of tries in WalkSAT. Default=1
 -mcsatParam N              : Set x; each step of MC-SAT retains each
                              non-violated clause with probability 1-exp(-|weigh
                              t|*x). DEFAULT=1.
 -mcsatSamples N            : Number of samples used by MC-SAT. Default=20.
 -minProb N                 : Mininum probability for output of marginal
                              inference.
 -nopart                    : Disable MRF partitioning. (Partitioning is
                              enabled by default.)
 -o (-r, -result) VAL       : REQUIRED. Output file.
 -printResultsAsPrologFacts : print Results As Prolog Facts.
 -psec N                    : Dump MCSAT resluts every [psec] seconds.
 -q (-query) VAL            : Query atom(s). Separate with comma.
 -queryFile VAL             : Input query file(s). Separate with comma.
 -threads N                 : The max num of threads to run in parallel.
                              Default = #available processors
 -timeout N                 : Timeout in seconds.
 -verbose N                 : Verbose level (0-3). Default=0
```

Figure 14: Command line options

```
2  !hasWord(d, NICE) v review(d, POS)
4  !hasWord(d, AWFUL) v review(d, NEG)
5  !hasWord(d, AWESOME) v review(d, POS)
6  !hasWord(d, TERRIBLE) v review(d, NEG)
3  !hasWord(d, EXCELLENT) v review(d, POS)
```

(a) Clauses of the same pattern

```
w  !hasWord(d, c1) v review(d, c2)
```

w	c1	c2
2	NICE	POS
4	AWFUL	NEG
5	AWESOME	POS
6	TERRIBLE	NEG
3	EXCELLENT	POS

(b) Clauses consolidated into one

Figure 15: Example of clause normalization

3.5 Command Options

Figure 14 lists the command options of Tuffy. Note that three of them are required: the program, the evidence, and the output file.

4 Tuffy Internals

Tuffy is implemented in Java and relies on PostgreSQL. The source depends on the following libraries: ANTLR 3, PostgreSQL JDBC4, Args4J, and JUnit 4. The Java doc can be found at `http://research.cs.wisc.edu/hazy/tuffy/`.

4.1 Representations

4.1.1 Symbols

As a standard practice, Tuffy uses a symbol table to convert all logic constants into integer IDs. Since the constants in MLNs are typically typed, we also keep track of constant type information; note that a constant may be shared by multiple types.

4.1.2 Program

Tuffy *consolidates* MLN clauses of the same pattern. This is motivated by the fact that many real-world MLNs could contain thousands or more clauses with only a handful of patterns. If we process (e.g., grounding) each clause separately, the SQL queries issued by Tuffy would incur substantial amount of overhead. Consequently, we want to represent them in compact forms – by "consolidating" clauses of the same pattern. This is done by "*lifting*" the constants together with the weight as "*meta-variables*". As such, we can represent clauses of the same pattern with a table that stores instantiations of those meta-variables. Figure 15 illustrates this process.

4.1.3 Data

Tuffy uses PostgreSQL to store input data and intermediate data (e.g., the ground Markov network). Specifically, we create a *symbol table*, one table for each type, and one table for each predicate. The symbol table has the schema `Constants(INT cid, STRING symbol)` and stores the map between constants and their assigned IDs. A type table has the schema `Type_typeName(INT cid)`, and stores the constant IDs that belong to this type (see . A predicate table has the schema

cid	symbol
1	Anna
2	Bob
3	Edward
4	Frank
5	Gary
6	Ellen

(a) Symbol table

id	person1	person2
1	1	2
4	1	3
7	1	4
10	3	4
13	5	6

(b) Predicate table of `Friends` (partial)

id	person1
2	1
5	3

(c) Predicate table of `Smoke` (partial)

Figure 16: Example loading results (partial)

Algorithm 1: KBMC

Input: C: a set of MLN clauses
Input: Q: a set of query atoms
Output: R: relevant atoms to Q

make all literals in C positive
$R \leftarrow Q$
while true do
  $\Delta \leftarrow \emptyset$
  for all atom $\mathtt{r} \in R$ **do**
    for all clause $\mathtt{c} \in C$ **do**
      for all literal $\mathtt{l} \in \mathtt{c}$ **do**
        $\theta \leftarrow mgu(\mathtt{r}, \mathtt{l})$
        $\Delta \leftarrow \Delta \cup \theta(\mathtt{c})$
  $R \leftarrow R \cup \Delta$
  remove subsumed atoms from R
  if R didn't change in this round **then**
    return R

Algorithm 2: LazyClosure

Input: an MLN program with evidence
Output: A_c: active ground atoms
Output: G_c: active ground clauses

$A_c \leftarrow \emptyset; G_c \leftarrow \emptyset$
while true do
  $\mathtt{G} \leftarrow$ ground clauses that are not satisfied by the evidence, and not satisfied by inactive atoms (those not in A_c)
  $\mathtt{A} \leftarrow$ atoms contained in $\mathtt{G}$
  if $\mathtt{A} \subseteq A_c$ **then**
    return A_c, G_c
  else
    $A_c \leftarrow A_c \cup \mathtt{A}$
    $G_c \leftarrow G_c \cup \mathtt{G}$

Figure 17: Grounding Algorithms of Tuffy

```
Pred_predicateName(INT id, BOOL truth, INT club,INT args),
```

where *args* is a list of all the arguments of this predicate. The `club` attribute indicates the type of a ground atom (query, evidence, etc.), and will be used during the grounding phase. The attribute `truth` takes value from {true, false, unknown}.

Figure 16 shows a partial snapshot of the database after loading the program and data in Figure 3.

4.2 Grounding

The grounding process is implemented with SQL queries in Tuffy. To improve the efficiency of grounding, Tuffy borrows ideas from knowledge-base model construction (KBMC) [9] and lazy reference [6].

Datalog and Scoping Rules After loading the evidence, Tuffy first executes the Datalog rules and scoping rules in the same order as they appear in the program.

KBMC Next, Tuffy runs KBMC (Algorithm 1; *mgu* = most general unifier) to analyze the MLN rules to identify (first-order) atoms that are relevant to the query; those atoms are then materialized in the predicate tables. In addition, KBMC also identifies the set of MLN rules that are relevant to the query.

Lazy Closure Grounding Conceptually, we might ground an MLN formula by enumerating all possible assignments to its free variables. However, this is both impractical and unnecessary. Fortunately, in practice a vast majority of ground clauses are satisfied by evidence regardless of the assignments to unknown truth values; we can safely discard such clauses [8].

Pushing this idea further, [6] proposes a method called "lazy inference" which is implemented by Alchemy. Specifically, Alchemy works under the more aggressive hypothesis that most atoms will be false in the final solution, and in fact throughout the entire execution. To make this idea precise, call a ground clause *active* if it can be violated by flipping zero or more active atoms, where an atom is active if its value *flips* at any point during execution. Alchemy keeps only *active ground clauses* in memory, which can be much smaller than the full set of ground clauses. Furthermore, as on-the-fly incremental grounding is more expensive than batch grounding, Alchemy uses the following one-step look-ahead strategy at the beginning: assume all atoms are inactive and compute active clauses; *activate* the atoms that appear in any of the active clauses,; and recompute active clauses. This "look-ahead" procedure could be repeatedly applied until convergence, resulting in an *active closure*. Tuffy implements this closure algorithm (Algorithm 2).

MRF The result of MLN grounding is a Markov random field (MRF), which is a hypergraph over active atoms with active clauses being the hyperedges. The MRF is stored in a *clause table* in PostgreSQL, which has the schema `Clauses(cid INT, lits INT[], weight FLOAT)`, where each tuple represents a hyperedge; `cid` is a unique ID, `lits` is an integer array representing ground literals, and `weight` is the weight of this hyperedge. Note that one hyperedge could be the result of merging the ground clauses from multiple MLN rules.

It is on this MRF that inference and learning take place.

Partitioning Beginning version 0.3, Tuffy performs MRF partitioning by default and then calls inference on each MRF component separately. The components are also processed in parallel, with as many threads as the number of available processors to the JVM. You can disable partitioning with the option `-nopart`. You can manually set the number of threads with `-threads`. For more details about the advantage of partitioning, please have a look at our VLDB paper [4].

4.3 Inference

Tuffy supports both MAP inference and marginal inference.

For MAP inference, there are two algorithms: WalkSAT [2] (Algorithm 3; δ-cost of an atom is the change of cost when this atom is flipped), which is a classic local search algorithm, and SweepSAT (Algorithm 4; an atom is *good* if its δ-cost is negative), which is a (somewhat ad hoc) MaxSAT algorithm that we designed for Tuffy. By default, WalkSAT is used. Beginning version 0.3, the SweepSAT algorithm is no longer available via command options. But the pseudo-code is still kept in this manual in case you find the algorithm interesting. In general, there are millions of SAT/MaxSAT heuristics out there. Tuffy does not strive to try out all of them, but rather sticks with a proven algorithm, which is WalkSAT.

For marginal inference, Tuffy implements the MC-SAT algorithm [5].

Algorithm 3: WalkSAT

Input: A: initial active ground atoms
Input: C: initial active ground clauses
Input: `MaxFlips`, `MaxTries`
Output: σ^*: a truth assignment to A

```
lowCost ← +∞, σ* ← 0
for try = 1 to MaxTries do
    σ ← a random truth assignment to A
    for flip = 1 to MaxFlips do
        pick a random c ∈ C that's violated
        rand ← random real ∈ [0, 1]
        if rand ≤ 0.5 then
            atom ← random atom ∈ c
        else
            atom ← atom in c with lowest δ-cost
        if atom is inactive then
            activate atom; expand A, C
        flip atom in σ; recompute the cost
        if cost < lowCost then
            lowCost ← cost, σ* ← σ
return σ*
```

Algorithm 4: SweepSAT

Input: A: closure of active ground atoms
Input: C: closure of active ground clauses
Input: `MaxSteps`, `MaxTries`
Output: σ^*: a truth assignment to A

```
lowCost ← +∞, σ* ← 0
steps ← 0
for try = 1 to MaxTries do
    σ ← a random truth assignment to A
    while steps ≤ MaxSteps do
        steps ← steps + 1
        numGood ← the number of good atoms
        if numGood = 0 then
            continue to next try
        flip each good atom with probability 0.5
        recompute the cost
        if cost < lowCost then
            lowCost ← cost, σ* ← σ
return σ*
```

Figure 18: MAP Inference Algorithms of Tuffy

4.4 Learning

Tuffy implements discriminative weight learning using the Diagonal Newton algorithm detailed in Lowd and Domingos [3].

References

[1] P. Domingos and D. Lowd. Markov Logic: An Interface Layer for Artificial Intelligence. 2009. 2, 20

[2] H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. *The Satisfiability Problem: Theory and Applications*, 1997. 17

[3] D. Lowd and P. Domingos. Efficient weight learning for Markov logic networks. *PKDD 2007*, 2007. 11, 18

[4] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: Scaling up Statistical Inference in Markov Logic Networks using an RDBMS. In *VLDB 2011*. 2, 5, 17

[5] H. Poon and P. Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *AAAI*, 2006. 17

[6] H. Poon, P. Domingos, and M. Sumner. A general method for reducing the complexity of relational inference and its application to MCMC. *AAAI-08*. 16, 17

[7] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 2006. 2

[8] J. Shavlik and S. Natarajan. Speeding up inference in Markov logic networks by preprocessing to reduce the size of the resulting grounded network. In *IJCAI-09*. 17

[9] M. Wellman, J. Breese, and R. Goldman. From knowledge bases to decision models. *The Knowledge Engineering Review*, 1992. 16

Function/Operator	Description	Example	Result
+, −, *, /, %	basic math	3.2 + 4 * 2	11.2
!	factorial	4 !	24
<<, >>	bitwise shift	1 << 6	64
&, |	bitwise AND/OR	(1 << 5) | (1 << 6)	96
^	bitwise XOR	5 ^17	20
=, <> or !=, <, >, >=, <=	numeric comparisons	1 + 1 = 2	true
~	bitwise NOT	~ 1	-2
sign(x)	sign of argument (-1, 0, +1)	sign(-2.3)	-1
abs(x)	absolute value	abs(-2.3)	2.3
exp(x)	exponential	exp(1)	2.71828
ceil(x)	ceiling	ceil(1.7)	2
floor(x)	floor	floor(1.7)	1
trunc(x)	truncate toward zero	trunc(43.2)	42
round(x)	round to nearest integer	round(1.7)	2
ln(x)	natural logarithm	ln(2)	0.693147
lg(x)	base-10 logarithm	lg(1000)	3
cos(x), sin(x), tan(x)	trigonometric functions (radian)	sin(2)	0.9093
sqrt(x)	square root	sqrt(100)	10
log(x,y)	$\log_x y$	log(2,8)	3
pow(x,y)	x^y	pow(2,8)	256

Figure 19: Numeric functions

A Built-in Functions and Operators

When writing conditions, the user can use a set of common numeric/string/boolean operators and functions, as shown in Figure 19 and Figure 20. Their implementation is greatly eased by the built-in functions in PostgreSQL. We plan to support user-defined functions in future releases.

B Glossary

Assuming that the reader understands primitive concepts in first-order logic such as *constants*, *variables*, and *predicates*, we list the definitions of other terminologies used in this manual. They comply with [1].

atom
An atom, or "atomic formula," is a predicate symbol applied to a tuple of terms; e.g., `Friends(Anna, x)`.

ground atom
A ground atom is an atom with only constant arguments; e.g., `Friends(Anna, Helen)`.

literal
A literal is an atom (e.g., `Friends(Anna, x)`) or the negation of an atom ¬`Friends(x, y)`. The input to Tuffy uses "!" for the negation symbol "¬".

clause
A clause is disjunction of literals; e.g., `!Smokes(x) v Cancer(x)`, which is equivalent to `Smokes(x)=>Cancer(x)`.

Function	Description	Example	Result
`len(s)`	string length	`len("badass")`	6
`lower(s)`	convert to lower cases	`lower("BadAss")`	"badass"
`upper(s)`	convert to upper cases	`upper("BadAss")`	"BADASS"
`initcap(s)`	capitalize initials	`initcap("bad ass")`	"Bad Ass"
`trim(s)`	remove surrounding spaces	`trim(" Bad Ass")`	"Bad Ass"
`md5(s)`	md5 hash	`md5("badass")`	"f23cc..."
`concat(s,t)`	string concatenation	`concat("bad","ass")`	"badass"
`strpos(s,pat)`	offset of `pat` in `s`	`strpos("badass","da")`	3
`repeat(s,k)`	repeat string `s` for `k` times	`repeat("bd",3)`	"bdbdbd"
`substr(s,i,len)`	substring of `s`, `s` [`i`:`i` +`len`)	`substr("badass",2,3)`	"ada"
`replace(s,f,t)`	replace `f` in `s` with `t`	`replace("badass","ad","")`	"bass"
`regex_replace(s,r,t)`	replace regex `r` in `s` with `t`	`regex_replace("badass",".a","")`	"ss"
`split_part(s,d,k)`	split `s` on `d`; get `k`-th part	`split_part("badass","a",2)`	"d"
`contains(s,t)`	`s` contains `t`?	`contains("bad","ass")`	false
`startsWith(s,t)`	`s` starts with `t`?	`startsWith("badass","ass")`	false
`endsWith(s,t)`	`s` ends with `t`?	`endsWith("badass","ass")`	true

Figure 20: String functions. Note that substring indexes start from 1; matching functions returns -1 on failure.

ground clause
A ground clause is a clause without variables; e.g., `!Smokes(Tom) v Cancer(Tom)`.

grounding
Grounding is the process of substituting the variables in a clause with constants; the result is a ground clause.

possible world
A possible world is a set of ground atoms; in other words, it's a relational database. An equivalent definition is "a truth assignment to all possible ground atoms."

MLN rule
An MLN rule is a clause with a real-valued weight. If the weight is positive (resp. negative), the rule is a positive (resp. negative) rule. E.g., `0.5 !Smokes(x) v Cancer(x)` is a positive rule.

ground rule
If we ground the clause of an MLN rule and associate the same weight to the resultant ground clause, we get a ground rule. E.g., `0.5 !Smokes(Tom) v Cancer(Tom)` is a positive ground rule.

MLN program
An MLN program is a set of MLN rules.

violated ground rule
Given a possible world, a ground rule is violated if one of the following conditions holds: 1) the rule is positive and is not satisfied; or 2) the rule is negative and is satisfied.

cost
The cost of a possible world with respect to an MLN program is the sum of the absolute values of the weights of all violated ground rules. Clearly, the lower the cost, the higher the probability of this possible world.