

markov thebeast User Manual

Version 0.0.2

Sebastian Riedel

January 8, 2009

Contents

1	Introduction	5
2	Installation	6
2.1	Requirements	6
2.2	Compilation	6
2.3	Starting thebeast	7
2.4	Hints	7
3	The Shell	8
3.1	Architecture	8
3.2	Signature	9
3.3	Model	9
3.4	Corpora	9
3.4.1	Working Corpus	10
3.4.2	Inspection Corpus	10
3.5	Getting and Setting Parameters	10
3.6	Supported Workflows	10
3.6.1	Initialization	10
3.6.2	Training	11
3.6.3	Testing	11
3.6.4	Inspection	11
4	Signatures	12
4.1	Types	12
4.1.1	Built-In Types	13
4.2	Predicates	13
4.2.1	Hidden Predicates	13
4.2.2	Observed Predicates	14
4.2.3	Global Predicates	14
4.2.4	Built-In Predicates	14
4.3	Functions	14
4.3.1	Built-In Functions	14
4.3.2	Weight Functions	15
5	Data	16
5.1	Possible Worlds	16
5.2	Loading Data	17

Contents

5.3	Loading Global Information	17
5.4	Inspecting Data	18
5.5	Printing Data	18
5.6	Evaluation	18
6	Markov Logic Networks	20
6.1	Terms	20
6.1.1	Constants	20
6.1.2	Variables	20
6.1.3	Function Applications	20
6.1.4	Weight Terms	21
6.2	Boolean Formulas	21
6.3	Weighted Formulas	21
6.3.1	Real-Valued Formulas	22
6.3.2	Named Formulas	22
6.3.3	Acyclicity Constraints	23
6.3.4	Formula Processing Hints	23
6.3.4.1	Inference Order	23
6.3.4.2	Grounding	23
6.4	Semantics	23
6.4.1	Variable binding/replacement	24
6.4.2	Functions	24
6.4.3	Ground Markov Network	25
6.5	Global vs. Local Formulae	25
7	Weights	27
7.1	Loading Weights	27
7.2	Saving Weights	28
7.3	Inspecting Weights	28
8	Inference	29
8.1	Cutting Plane Solver	29
8.1.1	Different Processing Orders	29
8.1.2	Other Parameters	30
8.2	Base Solvers	30
8.2.1	Integer Linear Program	31
8.2.1.1	Relaxing Integer Constraints	31
8.2.2	Other Solvers	31
9	Learning	32
9.1	Instantiating Features	32
9.2	Estimating Weights	33
9.2.1	Update Rule	33
9.2.2	Loss Function	34

Contents

9.2.3	Solver	34
9.2.4	Other Options	34

1 Introduction

Markov Logic [Richardson and Domingos, 2005] is a very expressive formalism to specify large and complex Markov Networks which allow you to capture local and nonlocal correlations in your data. `markov thebeast` is a software tool that can read such specifications and provides means of inference and learning. My view on it is basically: 'something like a CRF kinda tool, but it allows to you capture a larger class of correlations, not just sequential ones'. Thus, in theory you can use in for a lot of things.

However, there are a few catches:

1. With the expressive power Markov Logic gives you it is clear that there are lot of models you can define but for which inference and/or learning are infeasible. Section 6.5 gives a short introduction into this issue and reading it is highly recommended. However, there are still numerous other issues not covered here which you might encounter when playing around with `thebeast`.
2. There are a few subtleties to take in consideration in terms of the order in which commands may be executed (see section 3.6). Thus the easiest way for you to define your own model would be to use the skeleton code in the example directory.
3. As is, the code is really hard to extend and maintain for anyone but me, at least on some levels, because I put far too much focus on premature optimization. They will be future versions (wait for the 1.x.y release line) that change this but might be initially slower.
4. Our file format is different from the format `alchemy` uses. This is pretty annoying I guess and I hope to change in some future version.

A final note: this manual is far from complete and only explains (probably poorly) a small fraction of what `thebeast` can do. Moreover, this manual does not a aim to be an introduction to Markov Logic. For this I refer the reader to the original Markov Logic work Richardson and Domingos [2005].

2 Installation

2.1 Requirements

In order to compile `thebeast` you will need to have

1. Java SDK 1.5 or higher
2. ant 1.5 or higher (compilation is possible without but slightly more complicated)

In order to run `thebeast` you will need

1. Java 5 or higher
2. `lp_solve` 5.11 (or higher) for your OS. `thebeast` comes with a version for linux and Mac OS that should work okay

Recommended:

1. ant 1.5
2. bash (to execute the main script)

In most Mac OS and Linux versions this software is installed by default or easy to get. However, for Windows machines a bit more work might be necessary (such as installing `cygwin`). It is surely possible to run and compile `thebeast` without the recommended software, albeit less convenient.

2.2 Compilation

Download the archive and extract it using

```
$ tar xvf thebeast-0.x.y
```

This will create a directory `thebeast-0.x.y` which we will refer to as `HOME`. Change into `HOME` and call

```
$ ant
```

This compiles the source.

2.3 Starting thebeast

You can now call the `thebeast` executable by calling

```
$ $HOME/bin/thebeast
```

This is the easiest way to get it running. Alternatively you can execute the corresponding class file directly. For this you should have a look at the `thebeast` script to pick the right classpath etc.

2.4 Hints

1. You can simplify your workflow by adding `bin/` to your Path
2. You're free to move and rename HOME.

3 The Shell

Most likely you will communicate with `thebeast`¹ using a shell: a very simple scripting language and interpreter that allows to access all the essential functionality of the beast. It can be used to

- define models
- learn parameters
- do inference
- set parameters

You can start the shell by simply calling

```
$ thebeast
```

This leaves you with a prompt like

```
markov thebeast 0.x.y
#
```

Alternatively, you can save your script in a file, say `test.pml`, and execute this script directly via

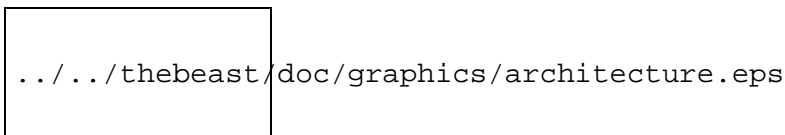
```
$ thebeast test.pml
```

In the following we will give an high level overview of the main components and commands of the shell. For details on defining models, learning, inference we refer the reader to later chapters.

3.1 Architecture

`thebeast` can be seen as a collection of components and resources. Shell commands can configure components and control them to process resources. Figure ?? gives a schematic overview of these components and resources. In the middle we see core components of the shell, the learner, solver and collector. Roughly speaking, the collector instantiates features, the learner learns weights and the solver applies a trained model to data. They all use or modify the *signature*, a collection of types, predicates and functions, the *model*

¹Alternatively, you can use the Java API.



or *Markov Logic Network*, a collection of formulas, and *weights*, a collection of real numbers that determine with how much penalty formulas can be violated. The data used for training and testing comes from the *corpora* and guess and gold atoms.

The remainder of this book will explain all these components, resources and interactions in more detail. In this chapter we will continue to give a high level overview of the components and resources.

3.2 Signature

Before doing anything, we need to define the types, predicates and functions that our model uses. All data has to adhere this signature. There are three types of definitions

Type defines a set of constants

Predicate defines a predicate over the Cartesian product of some types

Weight-Function defines a mapping from tuples to double values

The shell only maintains one single signature. Every definition is added to this signature.

Chapter 4 gives more details on signatures.

3.3 Model

Using the predicates, types and functions of the signature we can define a Markov Logic Network (MLN). An MLN consists of several formulas which assign scores (or probabilities) to substructures of a solution. As with signatures, the shell only maintains one global MLN. Each new formula is added to this MLN and all components share this model.

3.4 Corpora

The Beast needs data to learn weights from, to process during testing and for inspection and analysis of errors. This data comes from a corpus. A corpus is a sequence of databases. In the shell we have two corpora, the *working corpus* and the *inspection corpus*. Which to use depends on what you want to do with thebeast. you can find more details on corpora in chapter 5.

3.4.1 Working Corpus

The working corpus is used for training weights and testing a model, that is, applying the model to data. In general, the working corpus is saved on file and streamed in one by one, thus only needing a small amount of memory.

3.4.2 Inspection Corpus

The inspection corpus is used for analyzing the behaviour of the current model. The inspection corpus comes along with *current gold database* which can be loaded from any position in the corpus. We can seek forwards and backwards within the inspection corpus and print out the current database, apply the model it and compare the results of our model with the original gold data provided.

The inspection corpus fully resides in memory. Any database within can be randomly accessed.

3.5 Getting and Setting Parameters

As mentioned above, the shell also provides means to configure components and set parameters. This is achieved using the *set* command. Each component has name and a set of named properties. For example, the solver is named “solver” and has a parameter “maxIterations”. We can set this parameter by

```
set solver.maxIterations = 10;
```

We will give the names and parameters of components in the following chapters.

3.6 Supported Workflows

thebeast is somewhat sensitive to the order in which commands are entered or read from file. As for now, no warnings will be given when commands are executed in an unsupported order. Instead, *thebeast* is likely to fail at a seemingly unrelated execution point. To prevent this it is recommended to roughly follow the workflows implemented in the example scripts.

3.6.1 Initialization

Basically there are four workflows *thebeast* supports. The first one (initialization) prepares a corpus for training in order to improve the efficiency of online learning. This is usually achieved in a separated step because the preprocessed corpus can be reused for different training configurations as long as the MLN remains constant. This step also chooses some weights which should be fixed at zero. This improves efficiency and can also improve the learning performance. Finally, this step can collect the set of constants that are used in the training data and which should be used in later training, testing and inspection steps. An example initialization script is given in

```
examples/srl/init.pml
```

3.6.2 Training

During training the preprocessed corpus is used to estimate weights and saves those to a new file. Have a look at

```
examples/srl/train.pml
```

for an example training script.

3.6.3 Testing

In the third workflow (testing) the estimated weights are loaded and the corresponding model is applied to some test data. If gold solutions are available some performance metrics are printed out. The file

```
examples/srl/test.pml
```

should be used as starting point for test scripts.

3.6.4 Inspection

Finally, thebeast offers an inspection workflow in which the instances in a development corpus can be interactively processed and inspected. For example, during inspection we can solve an instance, print out errors, check the features active in the current solutions or the violated constraints etc. The file

```
examples/srl/inspect.pml
```

contains an example inspection script. Note that inspection would usually be done interactively, so one would include a partial version of this file in interactive mode and then go from here using the remaining commands in this file.

4 Signatures

Every model maintains a *signature*, a collection of symbols to be used in the formulas that describe the domain.

4.1 Types

TheBeast allows typed predicates and formulas. That is, constants are divided into sets (types) and predicates are defined over Cartesian Products of these types.

Say we want to perform Semantic Role Labelling. Here we are asked to label constituents of a parse tree with the semantic role these constituents play with respect to a given verb of the sentence. For example,

```
[A0He] [AM-MODwould] [AM-NEGn't] [Vaccept] [A1anything of value] from [A2those  
he was writing about] .
```

is a sentence that has been role-labelled wrt to the verb “accept”. Here the roles names are generic terms that have different meanings for different verb: A0 refers to the acceptor, A1 refers to the thing being accepted and so forth.

In this setting it will come in handy to define a type label as follows

```
type Label: A0,A1,A2,"AM-MOD","AM-NEG","somelabel";
```

Note that constants are

- either words starting with a capital letter and without any special characters or
- quoted strings

Type names have to be capitalized.

It can be tiresome to define all constants of a type in advance, especially when they are already specified implicitly in your training data. To make life easier thebeast allows you to write

```
type Label: ... ;
```

In this case the type is automatically augmented whenever a new constant is encountered. However, in order to reuse a model that has been trained using these open types one has to make sure to call

```
save types to "<filename>";
```

after training and to load the generated script file before testing with

```
include "<filename>";
```

4.1.1 Built-In Types

Thebeast comes with a set of built-in types. For now these only include an integer type

```
Int
```

and a Double type

```
Double
```

4.2 Predicates

Having defined our types, we are ready to set up predicates. In our example introduced above we could define a predicate *label* that maps constituents to labels using

```
predicate label: Int x Label;
```

Note that we use integers to represent the constituents. As the integer type contains a vast amount of constants grounding of such a predicate can become prohibitive. One could overcome this with two possible ways: have a special type with one constant for each constituent or have an additional predicate that denotes integers which are representing constituents. The first way is a bit troublesome because it requires to have a different type for each problem instance and types. However, we see types as a rather static concept, dynamic information is exclusively handled by predicates and their atoms. Thus we go for the latter option and define another predicate *candidate*:

```
predicate candidate: Int;
```

We differentiate between three types of predicates: hidden, observed and global ones.

4.2.1 Hidden Predicates

The ground atoms of *hidden predicates* are not seen during test time. Instead they have to be predicted using MAP inference (chapter 8). However, when learning weights the ground atoms of the hidden predicates are given and used to optimize parameters in order to reproduce these atoms for the given input. In our example above *label* is a hidden predicate because it is the predicate whose ground atoms we try to predict.

In order to declare a predicate to be hidden it has to be listed in the set of hidden predicates using the following command:

```
hidden: label, otherhiddenpredicate;
```

4.2.2 Observed Predicates

The ground atoms of *observed predicates* are seen both during testing and training. In our example candidate is an observed predicate because even at test and training time we always know which constituents are candidate arguments.

To declare a predicate to be hidden, use the following command

```
observed: candidate, otherobervedpredicate;
```

4.2.3 Global Predicates

Finally, we can declare a predicate to be *global*. Usually ground atoms only hold for one problem instance. For the next one we need to add all atoms from scratch. However, sometimes the same ground atoms should exist for all problem instances. These atoms and their corresponding predicates will be called global. For example, imagine we want to distinguish labels by whether they are denoting modifiers or complements. This can be done by introducing the unary predicates *modifier* and *complement* and marking them as global by writing

```
global: complement, modifier;
```

Now we can globally define which labels are complements and modifiers. This means a) less work for us (atoms have to only be added once) and b) less memory/disk usage (because we don't need to save the same atoms for each instance in a training/test set).

4.2.4 Built-In Predicates

leq(Int,Int)[<=] this predicate holds for two integers if and only if the first is less or equal to the second.

geq(Int,Int)[>=] this predicate holds for two integers if and only if the first is greater or equal to the second.

eq(*,*)[==] this predicate holds between terms if and only if both refer to the same constant.

undefined(WeightFunction(args...)) this predicate holds if the given weight function is not defined (=always returns 0.0) for the given arguments.

4.3 Functions

4.3.1 Built-In Functions

add(Int,Int)->Int[+] this function returns the sum of two integers.

minus(Int,Int)->Int[-] this function returns the subtraction of two integers.

product(Double,Double)[*] this function returns the product of two double values.

double(Int) this function casts an integer to a double.

abs(Double) this function returns the absolute value of a double.

bins(Integer,...,Integer) this function puts the last integer into one of the bins defined by $arg_0, arg_1, arg_2, \dots, arg_{i-1}$ and returns the number of this bin. If the argument to bin is negative its sign is changed and the number of the bin it falls into is multiplied by -1 before it is returned. For example, `bins(0,1,2,10,5)` returns 2 because bin 0 is $[0,1[$, bin 1 is $[1,2[$ and bin 2 is $[2,10[$ (and bin 3 is $[10,inf[$).

4.3.2 Weight Functions

The final part of a signature consists of its *weight functions*. They will be used to map (ground) formulas to weights that penalize or reward violations of these formulas. For example, it will be useful to reward or penalize the existence of particular labels in a solution. For this we could define a weight function

```
weight w_label: Label -> Double;
```

This defines `w_label` to map labels to double values. Note that weight functions, just as predicates, have to start with lowercase letters and can contain underscores.

If we were are sure that the existence of labels should only be rewarded we can write

```
weight w_label: Label -> Double+;
```

This ensures that the weight function maps labels to non-negative real values. Correspondingly,

```
weight w_label: Label -> Double-;
```

makes sure that the existence of labels are always penalized. Constraining the sign of a weight function will be very important for efficient inference. When using CPI you should make sure the weights for global formulae (see section 6.5) are signed properly.

5 Data

So far we haven't described how to feed thebeast with data. Thus, in this chapter we will give an overview how to load and save data, both for training and testing.

5.1 Possible Worlds

Data for thebeast is stored in a *possible world*. Each *possible world* contains a set of ground atoms for the predicates we have defined in our signature. During training we need to provide thebeast with ground atoms for both hidden and observed atoms. During testing the latter is sufficient – hidden atoms will be predicted by the inference method.

A possible world can be saved and loaded from a text file with the following format:

```
>>
>candidate
1
2
3
4

>label
2  A0
3  A1
```

Here a “>>” starts an instance and each single “>” followed by a predicate names starts a table of ground atoms for the given predicate. Each row of the table represents one ground atom, and the n-th column of the table represents the n-th argument of this atom. The table is terminated with an empty line. In the above example we have encoded the fact that the integers 1-4 are referring to candidate nodes in a parse tree and that node 2 is labelled as “A0” and node 3 as “A1”. The ground atoms this file describes are

$$candidate(1), \dots, label(2, "A0"), \dots$$

In order to save multiple possible worlds in one file (for example, if we want to give a set of training instances to thebeast) we can just concatenate multiple possible worlds by simply starting each new world with a new “>>”. For example, the following text file contains two instances:

```
>>
>candidate
```

```

1
2

>label
2   A0
>>
>candidate
3
5

>label
5   A1

```

5.2 Loading Data

In order to load data into thebeast we save the above text into a file (say, “example.atoms”) and execute

```
load corpus from "example.atoms";
```

Now the beast will use the provided data for learning and/or inference, depending on the commands that will follow.

5.3 Loading Global Information

As mentioned in the previous chapter, we can use global predicates in order to specify ground atoms that should hold in *all* possible worlds. Once we have declared a predicate to be global we can load its ground atoms using

```
load global from "filename";
```

Here the loaded file has the same format as normal data files introduced above, yet without a “>>” to indicate a new possible world to start (since there is only one global possible world this is unnecessary).

For example, if we have a global predicate *unique*, that identifies labels which can at most appear once for every verb, we would write a file containing the following lines:

```

>unique
"A1"
"A2"
"A3"

```

5.4 Inspecting Data

After loading the corpus thebeast has access to the data but it hasn't fully loaded it into memory. When training and testing this is no problem because data is processed sequentially and read in one-by-one or loaded completely, depending on the training mode given. This is automatically handled by thebeast. However, if the user wants to inspect the corpus and randomly jump around it is necessary to explicitly tell thebeast to load the full data into memory. This is done by calling

```
save corpus to ram;
```

after the corpus was loaded using the “load corpus” command. Now one can move around the corpus using the “next” command that moves the current pointer around the corpus. For example,

```
next 5;
```

moves the pointer to the current instances 5 instances further. Correspondingly,

```
next -4;
```

moves the pointer 4 instances backwards.

5.5 Printing Data

We can use thebeast to print out the current instance using

```
print atoms;
```

This renders the current instance using the format we introduced above. Alternatively, one can override the default print format with specific formats for specific tasks. This requires the implementation of a Java interface and is outside the scope of this manual for now.

5.6 Evaluation

Often we are interested in how well our model and inference method does in comparison with a gold standard. When thebeast loads in a test corpus that contains (hidden) gold atoms these can be used to compare the solutions thebeast generates to those gold atoms.

Say we have moved the cursor to a particular instance and have performed inference on this instance (more on this in chapter 8) then we can evaluate how well we do using

```
print eval;
```

5 *Data*

This prints out information such as precision, recall and F1-measure for each individual predicate as well as global versions over the ground atoms of all predicates. In addition the false negative and positive atoms for each predicate are printed.

Again we can adapt the output format and the type of information printed for different tasks. This requires Java classes to be implemented and again falls outside of the scope of this manual.

6 Markov Logic Networks

Markov Logic Networks in thebeast are slightly different to those presented in the original Markov Logic work [Richardson and Domingos, 2005]. Essentially we made the following two extensions:

- The weight of ground formula can be different for different groundings of the same formula
- The weight of a ground formula can be scaled by a numeric value

In addition we support boolean formulae that are not in traditional first order logic:

- Cardinality constraints (as a generalization of existential and universal qualifiers)
- Acyclicity constraints

In this chapter we will describe our notion of Markov Logic Networks and explain how they define a log-linear probability distribution over possible worlds. Essentially Markov Logic Networks are weighted formulae. Before we can describe these formulae in detail we need to introduce their main building blocks: terms and boolean formulas.

6.1 Terms

The most atomic building blocks of formulas are *terms*. A term is a symbol that describes an entity of the domain. Terms always have a type associated with them.

6.1.1 Constants

All constants are terms. For example, A0, A1 and “AM-MOD” are all terms.

6.1.2 Variables

Variables serve as placeholders for other terms. Their type constraints the type of terms they can be replaced with. Variables are words that begin a lowercase letter and may contain underscores.

6.1.3 Function Applications

Function applications are terms that apply a function to some argument terms. The type of a function application is the return type of the function. Thebeast comes with a set of build-in functions, such as $+$, $-$, $/$, $*$ etc that can be used with infix notation. For now no own functions can be defined. Later versions will lift this restriction.

6.1.4 Weight Terms

Finally, weight terms are weight functions applied to terms typed according to the signature of the function.

6.2 Boolean Formulas

Terms can be assembled into boolean formulas using atoms, logical connectives and quantifiers. We can write

```
candidate(c)
candidate(c) => label(c,A0)
candidate(c) & label(c,A0)
candidate(c) | label(c,A0)
```

Thebeast also allows us to use cardinal constraints (as generalizations of existential and universal quantifiers) such as

```
|Label l: label(c,l)| <= 1
```

indicating that for each candidate node there are no more than 1 label. Say we also have a hidden predicate *hasLabel* that indicates whether a node should have a label we can write

```
|Label l: label(c,l)| >= 1
```

As of now equality constraints are not allowed but can be easily encoded via a \leq and a \geq constraint.

6.3 Weighted Formulas

A Markov Logic Network is essentially a collection of weighted formulas. Each formula describes a set of similar features in a log-linear model, as we show later. It can be seen as a *parametrized* factor. A weighted formula has the following form¹

```
factor: for <QUANTIFICATION>
if <CONDITION> add [<FORMULA>] * <WEIGHT>;
```

where QUANTIFICATION is a list of variables with types, CONDITION a boolean formula that only contains observed predicates, FORMULA a boolean formula that contains at least one hidden predicate and WEIGHT a term to type *Double*. CONDITION, FORMULA and WEIGHT may not contain variables not quantified in QUANTIFICATION.

For example

¹Lines can be arbitrarily broken

```
factor: for Int c, Label l
  if candidate(c) add [label(c,l)] * w_label(l);
```

This reads as follows:

For each label l and integer c where $candidate(c)$ holds we add the weight value $w_label(l)$ to the total score if c is labelled with l .

More general, for each variable assignment for the given quantification that satisfies both the *condition* and the *formula* we add to the total score a weight which is a function of the variables.

Note that one could potentially write

```
factor: for <QUANTIFICATION>
  add [<CONDITION> => <FORMULA>] * <WEIGHT>;
```

to get the same semantics. However the main complexity of a model comes from how hidden predicates are involved; splitting hidden and observed part of a formula as we did in the example helps thebeast to optimize inference. In future versions this splitting might be done automatically.

6.3.1 Real-Valued Formulas

It can be helpful to scale the contribution of a true ground formula with some double value that depends on the arguments of the ground formula. This is the Pseudo Markov Logic equivalent of real-valued features. In PML we write the following to achieve this:

```
factor: for <QUANTIFICATION>
  if <CONDITION> add [<FORMULA>] * <DOUBLETERM> * <WEIGHT>;
```

For example, whether a certain candidate phrase c should be labelled as role l , i.e. $label(c,l)$, depends on its distance d from the predicate verb. We incorporate this knowledge by adding the term d times a weight that depends on the label l whenever $label(c,l)$ holds:

```
factor: for Int c, Label l, Int d
  if candidate(c) & distance(d)
  add [label(c,l)] * double(d) * w_distance(l);
```

Note that we have to explicitly cast the integer distance to a double value.

6.3.2 Named Formulas

We can also name a weighted formula

```
factor <NAME>: for <QUANTIFICATION>
  if <CONDITION> add [<FORMULA>] * <WEIGHT>;
```

where NAME is a lowercase string. This helps while debugging or when some components (learner, solver, feature collector) should behave differently for different formulas.

6.3.3 Acyclicity Constraints

Some tasks (such as dependency parsing or machine translation) require a structure or graph to cycle-free. Assume you have a binary predicate such as

```
predicate link: Int x Int;
```

that defines a graph structure (each pair in the *link* relation refers to an edge). Then you can enforce acyclicity with `thebeast` using the following formula:

```
factor: link acyclic
```

This will ensure that there is no cycle in the graph defined by *link*.

6.3.4 Formula Processing Hints

Ideally `thebeast` will find the best way of inference and learning for a given model automatically. However, sometimes it will be necessary to give certain hints about how to process specific parts of the model. In the following we present how give these hints.

6.3.4.1 Inference Order

Sometimes the order in which formulas are processed during inference can make a significant difference in efficiency. The model developer can control the order in which formulas are processed by annotating the formula as follows:

```
factor[<ORDER>]: for <QUANTIFICATION>
  if <CONDITION> add [<FORMULA>] * <WEIGHT>;
```

where `<ORDER>` can be any integer value. More information on processing order can be found in chapter 8.

6.3.4.2 Grounding

In Cutting Plane Inference it can also make sense to ground certain formulas from the start. This can be achieved by writing:

```
factor[ground-all]: for <QUANTIFICATION>
  if <CONDITION> add [<FORMULA>] * <WEIGHT>;
```

6.4 Semantics

In this section we will describe how the weighted formulae of an MLN define a distribution over possible worlds.

6.4.1 Variable binding/replacement

In the following we will make extensive use variable replacement in formulae. For this we will introduce a *binding* b as a mapping from variable symbols to terms. For example, let us substitute the variables in

$$\phi = \text{pos}(x, p) \Rightarrow \text{role}(x, r)$$

with

$$b = \{x \rightarrow 1, p \rightarrow NN, r \rightarrow A0\}$$

This yields

$$\phi[b] = \text{pos}(1, NN) \Rightarrow \text{role}(1, A0)$$

Equivalently we can substitute variables in terms: with

$$t = x + 1$$

we get

$$t[b] = 1 + 1$$

6.4.2 Functions

In Markov Logic each function symbol in our signature is either bound to a built-in function or to a weight function. These function map constants of the language (or tuples of constants) to other constants (in the case of weight functions the return constants are constants that represent real values). For a given fully grounded formula (with no variables) this allows us to recursively replace all function applications with the constants with respect to the built-in function that is tied to the corresponding function symbol. We will make use to this when describing the probability distribution that an MLN describes. Note that since each function symbol is always tied to a given actual function we will often identify the symbol with the function itself.

Assume we have a ground term t , that is, a term without any variables. Assume we also have a set of weights $\Theta = \{\Theta_w\}_w$ where each Θ_w is an function that maps the argument tuples of the domain type of w to double values. Then $e_\Theta[t]$ will refer to the constant this term represents. We define $e_\Theta(t)$ recursively:

- $e_\Theta[c] = c$ for any constant c
- $e_\Theta[f(a_1, \dots, a_n)] = f(e_\Theta[a_1], \dots, e_\Theta[a_n])$ for any built-in function f
- $e_\Theta[w(a_1, \dots, a_n)] = w_\Theta(e_\Theta[a_1], \dots, e_\Theta[a_n])$ for any weight function w

If a term t does not contain any weight function we will simply write $e[t]$.

Likewise, for a ground formula (with no variables) ϕ the formula $e_\Theta[\phi]$ refers to the formula in which each function application is recursively replaced with constants according to the built-in and weight functions. Again, if a formula ϕ does not contain any weight function we will write $e[\phi]$ instead of $e_\Theta[\phi]$.

6.4.3 Ground Markov Network

As mentioned before, an MLN M is a set of tuples $\{(q_i, \gamma_i, \phi_i, \alpha_i, w_i)\}_i$ where each tuple represents a weighted formula as described above. Here

- q_i is the set of free variables in the weighted formula
- γ_i (the *condition*) is a formula in First-Order-Logic that does not contain a hidden predicate of a weight function.
- ϕ_i (the *formula*) is a formula in First-Order Logic that does not contain a weight function.
- w_i (the *weight* term) is a function application of a weight function to some argument terms that do not contain hidden/unknown terms
- s_i (the *scale* variable) is a double variable in q_i that does not contain a weight function.

If not set, w_i is set to ∞ and s_i is set to 1.

Together with a *finite* set of constants C and a set of weights Θ , an MLN M then defines a log-linear probability distribution over possible worlds \mathbf{y} given the observation \mathbf{x} as follows

$$p(\mathbf{y}|\mathbf{x}, \Theta, M) = \frac{1}{Z_{\mathbf{x}}} \exp \left(\sum_{(q, \gamma, \phi, \alpha, w) \in M} \sum_{b \in B_q \wedge \models_{\mathbf{x}} e[\gamma[b]]} \mathbb{I}(\models_{\mathbf{y}} e[\phi[b]]) \cdot e_{\Theta}[w[b]] \cdot e[s[b]] \right) \quad (6.1)$$

where B_q is the set of possible bindings for the variables in q , the feature function $f_{\mathbf{c}}^{\phi}$ is defined as

$$f_b^{\phi}(\mathbf{y}) = \mathbb{I}(\models_{\mathbf{y}} \phi[b])$$

Z is a normalisation constant, $\mathbb{I}(true) = 1$ and $\mathbb{I}(false) = 0$.

This distribution is strictly positive and corresponds to a Markov Network which is referred to as the *Ground Markov Network*. It is also equivalent to a Weighted Satisfiability (SAT) Problem.

6.5 Global vs. Local Formulae

Local formulae are formulae which, when grounded, contain only one hidden ground atom. Since in `thebeast` formulae are divided into a condition that only contains observed predicates and an actual formula that may contain hidden predicates you can identify local formulae by simply checking the main formula. Local formulae are easy to deal with and require very few resources. Global formulae, on the other hand, require more efforts from inference and training methods and likely result in longer runtimes and higher memory requirements. Moreover, it takes some care to properly use them.

For example, assume that there exists some correlations between the roles assigned to the arguments of a predicate. In `thebeast` we could capture this by saying

6 Markov Logic Networks

```
weight w_global1: Label x Label -> Double-;
factor: for Int c1, Int c2, Label l1, Label l2
if candidate(c1) & candidate(c2) & c1 != c2
add [label(c1,l1) & label(c2,l2] * w_global1(l1,l2);
```

This is to say: whenever there are two distinct candidates arguments for a predicate labeled with l_1 and l_2 , respectively, then penalize this using the weight function $w_global1$. Note that we required the weights to be negative (via the *Double-* return type) – this is crucial for Cutting Plane Inference [Riedel, 2008]!

To understand this some knowledge of Cutting Plane Inference (CPI) is required. In each iteration CPI finds the current best (MAP) solution of a partial Ground Markov Network [Richardson and Domingos, 2005] and instantiates new parts of the network depending on which formulae are not instantiated yet but might result in a different best solution if they were.

Now assume that at a certain point CPI does find the actual gold solution. In this solution almost all groundings of the formula above are false, because for any pair of candidates there exists at most one pair of labels (l'_1, l'_2) assigned to both candidates (out of the full Cartesian product of labels with labels) – for all other pairs of labels (l_1, l_2) the formula is false. Now if this formula could have positive weights then a solution, in which all ground formulas are true which false in the current solution, would have a higher score. Thus there is potential for improvement and CPI adds all groundings to the current Network. This leads to a Ground Markov Network which is almost as large the as the complete network and is likely expensive to solve for the base solver.

Note that this restriction does not mean that we cannot include global formulae that reward the existence of certain properties. In the above case we can, for example, also add the following weight and formula:

```
weight w_global2: Label x Label -> Double+;
factor: for Int c1, Int c2, Label l1, Label l2
if candidate(c1) & candidate(c2) & c1 != c2
add [label(c1,l1) => label(c2,l2] * w_global2(l1,l2);
```

Here the existence of a certain role label (l_2) is rewarded in the presence of another (l_1) . This time the weight has to be positive, because in a gold solution most groundings of the above formulae would actually be true (because in most cases the premise of the implication is false).

To summarize: when using weighted global formulae and Cutting Plane Inference make sure to always specify whether weights should be nonnegative or nonpositive. The rule of thumb is:

- If most ground formulae are false in the actual gold solution the weight function must be nonpositive (*Double-*)
- If most ground formulae are true in the actual gold solution the weight function must be nonpositive (*Double+*)

In `examples/align/align-global.pml` you will find an example of the latter case.

7 Weights

In our model we use weight functions to allow formulas to be violated with some penalty. However, the actual mappings these weight functions describe are left unspecified. This is an important aspect of the architecture of thebeast: weights are not part of the model. When we use formulas to describe our domain this should only involve its qualitative properties. The quantitative aspect is handled by the learning algorithm that estimates the weights. This separation comes in particularly handy when we deal with millions of possible weight function arguments (i.e. features). In this case a file that contains both weights and formulas is effectively unreadable.

After we created a weight function all weights are zero, or rather, all weight function arguments are mapped to zero. We can change this in two ways, either we learn weights using data or we load weights from a file. We will describe the first way in chapter 9. Here we show how to load weights from a file.

7.1 Loading Weights

The format of a weight file is almost identical to the format of a data/ground atoms file. Simply write a “>>” (can be omitted) to begin the weight file and a “>” followed by the weight function name to start a table of weight mappings. Each row in this table represents a weight mapping. If the weight function has arity n then for $i \leq n$ the i -th column represents the i -th argument of the weight function. The $n + 1$ -th column is a double number representing the weight the argument tuple is mapped to.

For example, for a model that contains the weight function

```
weight w_label: Label -> Double;
```

the following text represents a weight mapping

```
>>
>w_label
A0  0.123
A1  -0.41
```

If we save this text in a text file, say “example.weights”, we can use

```
load weights from "example.weights";
```

in order to load this mapping.

7 Weights

Alternatively we can load weights from a binary format. Files of this format can be generated with thebeast, for example are training in order to reuse them later. Loading from the binary format is much faster, but such weight files cannot be created manually. In case we have a binary weight file “example.dmp” we can use

```
load weights from dump "example.dmp";
```

7.2 Saving Weights

One of the most common use cases of thebeast is to learn weights using a training set. If we want to reuse these weights on a test set later on we need to be able to save them after training. This is done by

```
save weights to "learnt.weights";
```

This writes the weights to a file “learnt.weights” using the text format introduced above. Alternatively one can store to a binary file using

```
save weights to dump "learnt.dmp";
```

These files are faster to load and save but can’t be manually inspected. However, one can inspect them using thebeast once their are loaded.

7.3 Inspecting Weights

Often it is helpful to see the numerical values of some weights. This can be done using

```
print weights;
```

which prints out all weight mappings of all weight functions. As there might be millions of these mappings printing all of them can become prohibitive. Instead one can use the name of a particular weight function to only get the mappings of this weight function

```
print weights.w_label;
```

Sometimes one is actually looking for the weight of a one particular argument tuple. In this case one can use the predicate name and the tuple in question:

```
print weights.w_label(A0);
```

Note: If a constant is quoted, such as “A0” then it needs to be referred to using single quotation marks, i.e. 'A0', in the above command.

8 Inference

There are several types of Inference one can perform for Markov Logic Networks. One is to calculate the probabilities of certain ground atoms to be true given some evidence atoms. Another is to find the most likely possible world given some evidence. This is usually referred to as Maximum A Posteriori Inference (MAP). So far only MAP inference has been implemented for thebeast.

MAP Inference in thebeast should mainly performed using Cutting Plane Inference [Riedel, 2008]. In a nutshell this mode of inference incrementally instantiates fractions of the complete Markov Network the Markov Logic Network describes. When used in combination with Integer Linear Programming this yields exact solutions that are often found in a very short amount of time. In the following we briefly describe this inference method. This will help the user to fine tune the efficiency of thebeast in case runtimes become too large.

8.1 Cutting Plane Solver

Cutting Plane Inference proceeds as follows:

1. Solve an initial part of the MLN
2. Find all ground formulas that do not contribute maximally to the total probability of the solution, if none are found return the solution produced during all iterations that has the maximal score
3. Add these formulas to the former partial model
4. resolve the new partial problem and return to 2.

The algorithm acts as a *meta* method that uses another solver to solve the actual partial problems. We will refer to this solver as *base solver*. Provided that two solvers have the same accuracy, it does not matter which one to use. Moreover, if the used solver is exact Cutting Plane Inference is exact, too.

8.1.1 Different Processing Orders

It often makes sense to first enforce formulas that make the solution small (such as formulas including \leq constraints). In general thebeast will try to predict a good order for a formula but it can be necessary to manually fine tune orders.

Say you have a formula that would be violated often if another constraint has not been enforced before. In this case it makes sense to give this formula a higher order (meaning

that it will be taken into consideration only after the formulae of lower order have been processed. By default the order of a formula is 0 thus by giving a formula the order 1 or higher it will be considered after all other formulae.

Assume we have a deterministic formula

```
factor: for Int c if candidate(c): |Label l: label(c,l)|<=1;
```

which ensures that a candidate has not more than one label. By default this formula has the order 0. Now assume another formula

```
factor[1]: for Int c1, Int c2, Label l1, Label l2
if candidate(c1) & candidate(c2) & c1 != c2
add [label(c1,l1) & label(c2,l2) * w_global(l1,l2);
```

Here we assigned this formula the order 1 (in the square brackets) it is less likely to be violated once we have ensured that no candidate has more than one label.

8.1.2 Other Parameters

maxIterations [100] This is an integer parameter that controls the maximum number of Cutting Plane iterations the solver will perform. By default this parameter is set to 100 for testing and to 20 for training. If your model needs more iterations you might want to try and improve the local part of your model.

integer [false] A boolean parameter which, if set to true, causes the solver to incrementally add integer constraints for those variables which are fractional in the current solution. If set to false integer constraints are never added. Note that this does not mean that results are necessarily fractional – if the propositional solver is configured to always return integer solutions this parameter has no effect on whether results are fractional or not.

checkScores [false] If the cutting plane algorithm converges and uses an exact base solver the last result will always be optimal. However, when we stop before convergence or when we are using an approximate base solver this is not guaranteed. For this case any solution produced along the way can be optimal and we have to check their scores to determine which one it is. This boolean parameter controls whether this checking is performed or whether the last solution is returned.

8.2 Base Solvers

The Cutting Plane Solver creates a Ground Markov Network with increasing size during inference. There are different types of propositional models that can be used to represent a Ground Markov Network. For example, one can represent Ground Markov Networks

using a Weighted SAT Problem. Likewise, Integer Linear Programs can represent Ground Markov Networks, too.

thebeast allows to define what kind of propositional models the Cutting Plane Solver should create and what kind of Inference method should be used for the specified type of models. For example, if we choose Weighted Sat as representation we can choose MaxWalkSAT as inference method. If we choose Integer Linear Programs as propositional model we can use the (free, open source) ILP solver *lp_solve* as inference method. In the following we describe how to configure the propositional model.

8.2.1 Integer Linear Program

It is recommended to use Integer Linear Programs (ILPs) as propositional representation for Ground Markov Networks. Thus, by default, thebeast uses ILPs. However, if this configuration was changed by the user at some point and should be changed back one can write

```
set solver.model = "ILP";
```

8.2.1.1 Relaxing Integer Constraints

The ILP represents ground atom states using 0-1 variables. By default these are set to be integers.¹ However, sometimes it can be okay to remove the integer constraints (or add the later). This can be achieved by writing

```
set solver.model.initIntegers = false;
```

To change it back use “true” instead of “false”.

8.2.2 Other Solvers

thebeast supports ILP and WSP representations of Ground Markov Networks. However, for the time being ILP is highly recommended as inference because the only WSP solver implemented, MaxWalkSAT, is not as accurate.

¹ This holds only for the test solver, during training we relax this constraint by default to gain efficiency.

9 Learning

In Markov Logic learning can refer to three things:

1. Learning what are good formulas
2. Learning what are good weights (or more general, weight functions) for formulas
3. Learning both good formulas and good weights at the same time

However, as of now thebeast only supports the second case. We assume that the user has designed a good set of formulas (aka feature templates in other frameworks) and has some training corpus at hand that can be used to optimize the weights for each formula.

There is a multitude of possible training methods to pick from. Yet, many of these require the problem to have a particular structure in order to be efficient. Online Learning methods, on the other hand, only need efficient MAP inference in order to be applicable. Since Cutting Plane Inference can provide exact and efficient inference for several interesting problems for the time being thebeast's only learning method is Online Learning.

An Online learner roughly works as follows:

1. set number of epochs to 0.
2. for each instance x_i, y_i of the training corpus do
 - a) run inference to calculate $\hat{y} = \arg \max_y s(x_i, y)$
 - b) update current weights w_i by comparing \hat{y} to y_i
 - c) if averaging: add w_i to global weight vector w
3. if number of epochs is larger than some predefined value go to 4), otherwise increase number of epochs and go to 2).
4. If averaging return $w/epochs/instances$ otherwise return last solution

9.1 Instantiating Features

Each weighted formula ϕ has a weight function w associated with it. Initially this function maps each argument tuple to 0. After training we expect some all or of these arguments to be mapped to nonzero values. However, sometimes it helps the learner (and improves memory requirements) to restrict the set of weights which can be nonzero. For example, McDonald et al. [2005] fixes weights of certain feature type to zero if they (or rather, their

arguments) are not used in the training data (or less than k times). This is sometimes known as feature instantiation: we instantiate a subset of features of a certain feature type or template. We will refer to this step as collection step because we iterate over the corpus and collect features (or their counts).

In thebeast collection is triggered by calling

```
collect;
```

By default this will instantiate all features (weight arguments) which are seen at least once in the training set. If we want to allow all arguments of a weight function w to have nonzero weights we will need to state

```
set collector.all.w = true
```

before we start the collection process. If we want to restrict the set of arguments with nonzero weights to those seen at least 2 times we would write

```
set collector.cutoff.w = 2
```

again before we start collection.

9.2 Estimating Weights

As mentioned above we currently only support online learning as method to estimate weights. However, we do support different update rules and loss functions to pick from.

9.2.1 Update Rule

MIRA (default) By default thebeast uses 1-best MIRA [Crammer and Singer, 2001] (similar to the work of McDonald et al. [2005]). That is, we use the best solution given the current model and update the weights using a quadratic program that takes the feature vector of this solution, the ground truth (gold) feature vector and a loss into account.

```
set learner.update = "mira";
```

Perceptron Plain perceptron with 1.0 learning rate and decay

```
set learner.update = "perceptron";
```

To change the learning rate of the perceptron write

```
set learner.update.rate = 0.5
```

Likewise, to change the decay write

```
set learner.update.decay = 3.0
```

9.2.2 Loss Function

Mira can use different loss functions. There are a few built-in loss functions to try:

GlobalNumberOfErrors (default) counts the number of false positive and false negative ground atoms for all hidden atoms

```
set learner.update.loss = "globalNumErrors";
```

AverageNumberOfErrors counts false positive and negative atoms per hidden atoms and averages this number over all hidden predicates

```
set learner.update.loss = "avgNumErrors";
```

GlobalF1 calculates 1.0-F-score using total precision and recall of all hidden predicates

```
set learner.update.loss = "globalF1";
```

AverageF1 calculates 1.0-F-score per hidden predicate and averages the result

```
set learner.update.loss = "avgF1";
```

ZeroOne if the solution is completely correct a zero loss is returned, otherwise a 1.0 loss.

```
set learner.update.loss = "exact";
```

9.2.3 Solver

The online learner maintains an own solver which can be configured as described in chapter 8. However, all properties of the solver have to be qualified using the term "learner.". For example, if we write

```
set solver.maxIterations = 10;
```

for the test-time solver we would write

```
set learner.solver.maxIterations = 10;
```

for the solve used during online learning.

9.2.4 Other Options

Average averages all weight vectors after training

```
set learner.average = true;
```

Bibliography

Koby Crammer and Yoram Singer. Ultraconservative online algorithms for multiclass problems. In *14th Annual Conference on Computational Learning Theory, COLT 2001 and 5th European Conference on Computational Learning Theory, EuroCOLT 2001, Amsterdam, The Netherlands, July 2001, Proceedings*, volume 2111, pages 99–115. Springer, Berlin, 2001.

R. McDonald, K. Crammer, and F. Pereira. Online large-margin training of dependency parsers. In *43rd Annual Meeting of the Association for Computational Linguistics, 2005*, 2005.

Matthew Richardson and Pedro Domingos. Markov logic networks. Technical report, University of Washington, 2005.

Sebastian Riedel. Improving the accuracy and efficiency of map inference for markov logic. In *UAI '08: Proceedings of the Annual Conference on Uncertainty in AI, 2008*.