Markov TheBeast User Manual

Sebastian Riedel

May 31, 2008

Contents

1 Introduction

2 Installation

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 the THEDIR. Change into THEDIR and call

```
$ ant -f thebeast.xml
```

This compiles the source. You can now call thebeast executable by calling

```
$ $THEDIR/bin/linux/thebeast
```

if you are running linux, or

```
$ $THEDIR/bin/mac/thebeast
```

if you run a mac.

2.1 Hints

1. You can simplify your workflow by adding ..bin/linux/ or ../bin/mac to your Path
2. You're free to move and rename THEDIR.

3 The Shell

Most likely you will communicate with thebeast[1] using *theshell* (in short *the-sh*): 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-sh by simply calling

```
$ thebeast
```

This leaves you with a prompt like

```
Markov TheBeast v0.x.y
#
```

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

```
$ thebeast test.thesh
```

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 the later chapters.

3.1 Architecture

theshell can be seen as a collection of components and resources. thesh commands can configure components and control them to process resources. Figure 3.1 gives a schematic overview of these components and resources. In the middle we see core components of theshell, 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

[1]Alternatively, you can use the java API.

Figure 3.1: A schematic overview of the components and resources within thesh.

all use or modify the *signature*, a collection of types, predicates and functions, the *model,* 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 thesh 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, theshell 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.

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 constiuents of a parse tree with the semantic role these constiuents play with respect to a given verb of the sentence. For example,

> [$_{\text{A0}}$He] [$_{\text{AM-MOD}}$would] [$_{\text{AM-NEG}}$n't] [$_{\text{V}}$accept] [$_{\text{A1}}$anything of value] from [$_{\text{A2}}$those 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 adn 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 implicitely 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 buillt-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 constiuents to labels using

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

Note that we use integers to represent the constiuents. As the integer type contains a vast amount of constants grouding 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, otherobersevedpredicate;
```

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 substraction 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 arg0,arg1,arg2,argi-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 existance 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 tha 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.

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

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

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 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 explicitely 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.

5.4 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.

5.5 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 thos 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;
```

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

Once we have defined the symbols of our language we are ready to use these to make statements about the domain. We call these statements *formulas*. A collection of such statements is called a *Markov Logic Network [Richardson and Domingos, 2005]* Before we can describe these statements 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)
forall Label l: exist Int c: candidate(c) => label(c,l)
```

Thebeast also allows us to use cardinal constraints such as

```
forall Int c: candidate(c) => |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

```
forall Int c: hasLabel(c) => |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 loglinear model, as we show later. It can be seen as a *parametrized* factor. A weighted formula has the following form[1]

```
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

```
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.

[1]Lines can be arbitarily broken

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, Double d
if candidate(c) & distance(d)
add [label(c,l)] * d * w_distance(l);
```

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 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.

6.3.3.1 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.

6.3.3.2 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

We briefly described the semantics of a weighted formula in section 6.3. Let us now give a more detailed explanation. For this we will look at the task of labelling the semantic arguments of the verb "play" in the (fragment of the) sentence

"Haag plays Elianti"

and use some of the predicates formulae and defined above.

For the given sentence we will consider only two candidate constituent spans. Span 1 covers the NP "Haag" and Span 2 covers the NP "Elianti". The corresponding possible world could look like

$$\begin{array}{ll} span\,(1,0,0)\,, & candidate\,(1)\,, \\ span\,(2,2,2)\,, & candidate\,(2) \\ span\,(3,1,2)\,, & left\,(1) \end{array}$$

Note that span 3, the VP "plays Elianti", is not considered as candidate argument of "plays". The possible world also reflects that span 1 is to the left of the verb.

An MLN defines a probability distribution over possible worlds that are consistent with an observation. In our case we would like this distribution to assign a high probability to a world

$$\begin{array}{rl} span\,(1,0,0)\,, & candidate\,(1)\,, \\ span\,(2,2,2)\,, & candidate\,(2) \\ span\,(3,1,2)\,, & left\,(1) \\ label\,(1,"A0") & label\,(2,"A1") \end{array}$$

and a lower one to

$$\begin{array}{ll} span\left(1,0,0\right), & candidate\left(1\right), \\ span\left(2,2,2\right), & candidate\left(2\right) \\ span\left(3,1,2\right), & left\left(1\right) \\ label\left(2,\text{"}A0\text{"}\right) & \end{array}$$

In the following we will describe how the weighted formulae of an MLN are used to define such probability distribution.

As mentioned before, an MLN M is a set of tuples $\left\{\left(q_i,\gamma_i,\phi_i,\alpha_i,w_i\right)\right\}_i$ 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 hidden predicate
- ϕ_i (the *formula*) is a formula in First-Order Logic
- w_i (the *weight* term) is a function application of a weight function to some argument terms in q_i
- s_i (the *scale* variable) is a double variable in q_i.

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

Together with a *finite* set of constants C, an MLN M then defines a log-linear probability distribution over possible worlds $\mathbf{y} \in \mathcal{Y}_{P,C}$ given the observation $\mathbf{x}$ as follows

$$p\left(\mathbf{y}|\mathbf{x},\Theta,M\right) = \frac{1}{Z_{\mathbf{x}}} \exp\left(\sum_{(q,\gamma,\phi,\alpha,w)\in M}\ \sum_{b\in B_q \wedge \vDash_{\mathbf{x}} \gamma_i/b} f_b^{\phi}\left(\mathbf{y}\right)\cdot\Theta_{w[b]}\cdot s\left[b\right]\right) \tag{6.1}$$

where the feature function $f_{\mathbf{c}}^{\phi}$ is defined as

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

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

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. Many applications, including the two presented in this work, contain deterministic constraints. These will be realized using very large weights that result in areas with near-zero probability.

For example, with $M = \left\{\left(\phi_1, 2.5\right), \left(\phi_2, 1.2\right)\right\}$ and the finite set of constants $C = \left\{n_1, n_2, \ldots\right\}$ that represent the nodes of the parse tree, the log-linear model would contain, among others, the feature

$$f_{n_1}^{\phi_1}\left(\mathbf{y}\right) = \mathbb{I}\left(\vDash_{\mathbf{y}} agent\left(n_1\right) \Rightarrow left\left(n_1\right)\right)$$

that returns 1 if the contained ground formula holds in the possible world $\mathbf{y}$ and 0 otherwise.

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 undreadable.

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.

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 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. [get section from MLN chapter]

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. 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.

9 Learning

In Markov Logic learning can refer to three things:

1. Learning what are good formulas
2. Learning whar 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 a 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 Collecting Features

9.2 Estimating Weights

Bibliography

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.