The Alchemy Tutorial

Marc Sumner
Pedro Domingos
Department of Computer Science and Engineering, University of Washington

June 20, 2007

1 Introduction

Alchemy is a software tool designed for a wide range of users. Anyone with a need for a knowledge base with uncertainty will find Alchemy useful and this is the target audience of this tutorial. It assumes the reader has general knowledge of classical machine learning algorithms and tasks and is familiar with first-order and Markov logic and some probability theory.

Markov logic serves as a general framework which can not only be used for the emerging field of statistical relational learning, but also can handle many classical machine learning tasks which many users are familiar with. Instead of addressing only certain domains or adding ad hoc features to deal with anomalies, Markov logic presents a language to handle machine learning problems intuitively and comprehensibly. With this in mind, this tutorial looks to serve two purposes:

- Show the user how to model a few learning tasks in familiar learning representations.
- Introduce the user to learning and inference in Markov logic so that he/she has the basic tools to develop his/her own applications in this framework.

This tutorial is not meant to be exhaustive in terms of the capabilities of Alchemy. Many more learning tasks, both classical and emerging, can be handled by Alchemy in an elegant and intuitive manner. In addition, many new tasks have not been considered in the Markov logic framework; Alchemy is a work in progress and is continually being extended to meet these needs. The best catalyst for this progress is user feedback, so please tell us about any problems or limitations with Alchemy and wishes for the next version.

We start with the basics of Alchemy in the next section before moving on to more interesting tasks which can be accomplished. All of the datasets used in this tutorial are available at http://alchemy.cs.washington.edu in the “Datasets” section.

2 The Basics

Alchemy has been designed to run on the Linux platform. After downloading it, unzip and untar the file with `tar xvfz alchemy.tgz`. This creates a directory `alchemy` which contains the source code and documentation. In the file `alchemy/src/` type `make depend; make` which will produce the executables `learnstruct`, `learnwts`, and `infer` in the directory `alchemy/bin/`. If you have problems making the executables, please consult the user's manual, as Alchemy is dependent on other programs being present. Throughout the tutorial, these three commands will be used without specifying the location of them.

Alchemy can perform three basic tasks: *structure learning*, *weight learning*, and *inference*. The former two involve learning the structure or parameters of a model given a training database consisting of ground atoms. The latter involves inferring the probability or most likely state of *query* atoms given a test database consisting of *evidence* ground atoms.

The two learning commands, `learnstruct` and `learnwts` take the same basic arguments as input:

`-i <string>`	Input .mln files
`-o <string>`	Output .mln file
`-t <string>`	Training .db files

All other arguments are optional and are set to a default value if not specified. For a list of all command line options, the reader should consult the developer's manual.

To perform inference, the command `infer` requires an MLN with weights (either learned or hand-crafted), evidence on which it conditions and a set of query predicates:

`-i <string>`	Input .mln files
`-r <string>`	Output file containing inference results
`-e <string>`	Evidence .db files
`-q <string>`	Query atoms (comma-separated with no space)

Again, many other options are available which can be found in the developer's manual. One important default at this point is the type of inference being performed. Alchemy can perform MAP inference which outputs the most likely state of the query atoms or probabilistic inference which outputs marginal probabilities of the query atoms given the evidence. The latter is the default.

We have encountered two file types: `.mln` files and `.db` files. The former contains the MLN used for inference or learning, the latter contains a set of ground atoms used as training examples (when learning) or evidence (when running inference). A `.mln` file consists of two basic parts: declarations and formulas. The declaration section must contain at least one predicate, while the formulas section contains 0 or more formulas. Optionally, one can enumerate the constants of each type used in the `.mln` and `.db` files; if there is no enumeration, the set of constants is implied from all constants present in both files. A `.db` file consists of a set of ground atoms, one per line. Evidence predicates are assumed by default to be closed-world, meaning that if they are not present in the `.db` file, they are assumed false. Non-evidence predicates, on the other hand, are assumed open-world by default. The full syntax of the files can be found in the user's manual; however, the basics needed to get started should become apparent as you read this tutorial.

Now that we know the basic commands and file types available in Alchemy, we can move step by step from simple to complex problems. We end this section with the modeling of a uniform and binomial distribution and make a natural progession to logistic regression, hidden markov models, etc. , showing real-world applications along the way.

2.1 Uniform Distribution

Now that we know the basic commands and file types used in Alchemy, we want to start with the simplest MLN one can think of: the empty MLN. Suppose we want to consider the output of a coin flip. We can state that the outcome of a flip is heads with the predicate `Heads(flip)`, where `flip` ranges from, say, 1 to 20. If `Heads(n)` is true, then flip `n` was heads; otherwise it was tails. By supplying an empty MLN with `Heads` as the only predicate:

```
flip = {1,...,20}

Heads(flip)
```

we can perform probabilistic inference to result in a uniform distribution:

```
infer -i uniform.mln -r uniform.result -e empty.db -q Heads
```

Alchemy requires a `.db` file with evidence; here, `empty.db` is an empty file. The resulting file `uniform.result` shows the marginal probabilities of each grounding of `Heads` given no other evidence. In the limit, these should approach 0.5 (note, we can trade off accuracy for speed by varying the number of samples with the option `-maxSteps`).

2.2 Binomial Distribution

We can also model a binomial distribution as an MLN. If we move up a step from the empty MLN and add the unit clause `Heads(flip)` with a weight `w` to our MLN:

```
flip = {1,...,20}

Heads(flip)

// Unit clause
1 Heads(f)
```

we have a binomial distribution with n being the number of flips (in our case 20) and $p = \frac{1}{1+e^{-w}}$, where w is the weight of the unit clause (in our case 1). We can verify this by running probabilistic inference:

```
infer -i binomial.mln -r binomial.result -e empty.db -q Heads
```

In the limit the marginal probabilities should approach $\frac{1}{1+e^{-1}} = 0.73$.

2.3 Multinomial Distribution

We can easily extend our distribution from binomial to multinomial in Markov logic. For example, we might want to model the outcome of a six-faced die over a number of throws. We enumerate the throws and the faces of the die with:

```
throw = {1,...,20}
face = {1,...,6}
```

and the outcome of each throw with the following predicate:

```
Outcome(throw, face)
```

Additionally, we want to model the fact that each throw has exactly one outcome. This entails a rule stating that at *least* one outcome must occur for each throw and one stating that at *most* one outcome must occur:

```
Exist f Outcome(t, f).
Outcome(t, f) ^ f != f' => !Outcome(t, f').
```

These rules must be modeled as hard constraints (denoted by the full stop at the end of the formula. This type of modeling can become cumbersome when we start dealing with MLNs with more formulas. Even more severe, MCMC algorithms such as Gibbs sampling will not converge with these constraints in the model. Fortunately, Alchemy allows us to declare this type of constraint in a much more compact manner. Instead of the last two formulas, we can simply declare the predicate `Outcome` with the `!` operator put on the `face` argument:

```
Outcome(throw, face!)
```

The cumbersome notation is no longer necessary and the inference and learning algorithms enforce these block constraints internally, thus alleviating the problem of convergence and making inference more efficient.

If we run probabilistic inference on this MLN, querying `Outcome`:

```
infer -i multinomial.mln -r multinomial.result -e empty.db -q Outcome
```

we find that each outcome has an (approximately) equal probability. Now, say we want to consider a biased die which does not result in each face with equal probability. In Markov logic, this requires a different weight for each grounding of the `face` variable, so we would need to write 6 formulas, each a unit clause `Outcome(t, f)`, with `f` running from 1 to 6. Again, Alchemy can help us with some user-friendly notation, the `+` operator. If we add the following formula to our MLN:

```
Outcome(t, +f)
```

then Alchemy produces the clauses for which we want to learn weights. The file `biased-die.db` contains data generated from a die biased to roll only a one or a six with equal probability; we can use this to learn weights for each outcome with:

```
learnwts -i multinomial-biased.mln -o learned.mln -t biased-die.db
  -ne Outcome -noAddUnitClauses
```

which outputs the learned MLN in the file `learned.mln`.

3 Social Network Analysis

Now that we have seen some trivial examples, we might want to do something more practical. We start with a simple social network of friends, smoking, and cancer. The network attempts to model friendship ties between people, their smoking habits and the causality of cancer.

If we want to model "smoking causes cancer" in first-order logic, this would look like:

$$\forall \texttt{x}\ \texttt{Smokes(x)} \Rightarrow \texttt{Cancer(x)}$$

Of course, this does not hold for all smokers, so in Markov logic we can just tack a weight on to the rule, or, as we do here, learn a weight from training data. In Alchemy all free variables are universally quantified, so the formula is

```
Smokes(x) => Cancer(x)
```

Other rules we might add are "People with friends who smoke, also smoke" or "People with friends who don't smoke, don't smoke", i.e.:

```
Friends(x,y) => (Smokes(x) <=> Smokes(y))
```

Converted to CNF, this becomes the two clauses:

```
!Friends(x,y) v Smokes(x) v !Smokes(y)
!Friends(x,y) v !Smokes(x) v Smokes(y)
```

We can learn weights from the training data with the command

```
learnwts -d -i smoking.mln -o smoking-out.mln -t smoking-train.db
  -ne Smokes,Cancer
```

Here, `-d` denotes that we want to run discriminative learning, `-o` designates the output MLN, `-t` specifies the training data, and `-ne` indicates which predicates are to be viewed as non-evidence during learning. This produces the file `smoking-out.mln` with the learned weights. Using this with the test data, we can compute the marginal probabilities of each person smoking and getting cancer:

```
infer -ms -i smoking-out.mln -r smoking.result -e smoking-test.db
  -q Smokes,Cancer
```

This gives us marginal probabilities of each ground query atom. Alternatively, we might want the most likely state of the query atoms, the MAP state. In this case, we would use the `-a` option instead of `-ms`. The output contains each query atom followed by its truth value (1 = true and 0 = false).

4 Logistic Regression

One of the most wide-spread and effective classifiers in statistical learning, logistic regression, can be easily implemented in Alchemy. Here, we only deal with binary predictors and dependent variables, but the extension to these is intuitive as well.

Logistic regression is a regression model of the form:

$$\ln\left(\frac{P(Y=1|X=x)}{P(Y=0|X=x)}\right) = \alpha + \sum_{i=1}^{n} \beta_i x_i \tag{1}$$

where X is a vector of binary predictors and Y is the dependent variable. So how do we describe this model in Markov logic? If we look at the model of Markov networks:

$$P(X=x) = \frac{1}{Z}\exp(\sum_j w_j f_i(x)) \tag{2}$$

this implies we need one feature for Y and one feature for each X_i,Y in order to arrive at the model

$$P(Y=y, X=x) = \frac{1}{Z}\exp(\alpha y + \sum_j \beta_j x_j y) \tag{3}$$

resulting in

$$\frac{P(Y=1|X=x)}{P(Y=0|X=x)}) = \exp(\alpha + \sum_j \beta_j x_j)/\exp(0) = \exp(\alpha + \sum_j^{n} \beta_j x_j) \tag{4}$$

To represent this as an MLN, each parameter is represented as a weight corresponding to each formula, i.e. α Y and for each i, β_i $X_i \wedge Y$

We demonstrate this on an example from the UCI machine learning data repository, the *voting-records* dataset, which contains yes/no votes of Congressmen on 16 issues. The class to be determined is “Republican” or “Democrat” (our Y), and each vote is a binary predictor (our X_i). So, the resulting clauses in the MLN are:

```
Democrat(x)
HandicappedInfants(x) ^ Democrat(x)
```

```
WaterProjectCostSharing(x) ^ Democrat(x)
AdoptionOfTheBudgetResolution(x) ^ Democrat(x)
...
```

This predicts whether a Congressman is a Democrat (if `Democrat(x)` is false, `x` is a Republican). Alternatively, we could have modeled the connection between the predictors and the dependent variable as an implication, i.e.:

```
HandicappedInfants(x) => Democrat(x)
```

These two models are equivalent when we condition on the predictors.

We can perform generative or discriminative weight learning on the mln, given the training data voting.db with the following command:

```
learnwts -g -i voting.mln -o voting-gen.mln -t voting-train.db -ne Democrat
```

or

```
learnwts -d -i voting.mln -o voting-disc.mln -t voting-train.db -ne Democrat
```

The weights obtained tell us the relative "goodness" of each vote as it can predict whether a Congressman is a democrat or not. Given this, we can look at a new Congressman (or several) and his/her voting record and predict whether he/she is a democrat or republican. We run inference with

```
infer -ms -i voting-disc.mln -r voting.result -e voting-test.db -q Democrat
```

This produces the file `voting.result` containing the marginal probabilities of each Congressman being a Democrat.

5 Text Classification and Information Retrieval

5.1 Text Classification and Collective Classification

Text classification is an extremely important application of machine learning. We demonstrate text classification on a subset of the WebKB dataset which contains the text of web pages from four universities. Each page belongs to one of four classes: *course, faculty, research project, or student.*

If a given page contains a given word, then the predicate `HasWord(word, page)` is true for that pair; otherwise it is false. This information is given and we wish to infer the class of a page, given by the predicate `Topic(class, page)`. Therefore, we need a rule

```
HasWord(+w, p) => Topic(+c, p)
```

for each (word, class) pair and a rule stating that, given no evidence, a page does not belong to a class:

```
!Topic(c, p)
```

That's it! In fact, we can eliminate the last rule; the addition of a unit clause to the MLN is helpful in many domains and, for this reason, Alchemy adds unit clauses to the MLN by default when weights are learned (this can be supressed with the option `-noAddUnitClauses`). If we run weight learning, we will learn a weight for every word/class pair representing how good of a predictor each word is for each class. We can extend this model by stating that each page must belong to exactly one class in the predicate declaration:

```
Topic(class!, page)
```

We run weight learning with the following command:

```
learnwts -d -i text-class.mln -o text-class-out.mln
  -t text-class-train.db -ne Topic
```

We can then use the classifier to classify test instances with

```
infer -m -i text-class-out.mln -r text-class.result
  -e text-class-test.db -q Topic
```

We can extend this model to demonstrate how to perform hypertext classification with Alchemy. WebKB also contains information on which pages are linked to each other, via the predicate `LinkTo(linkid, page, page)`. We can incorporate this into a rule to perform hypertext classification:

```
Topic(c, p1) ^ LinkTo(id, p1, p2) => Topic(c, p2)
```

We add `LinkTo` from the dataset to our evidence and this one formula extends our text classifier to a hypertext classifier. We learn weights and infer as before:

```
learnwts -d -i hypertext-class.mln -o hypertext-class-out.mln
  -t text-class-train.db,links-train.db -ne Topic -dZeroInit

infer -m -i hypertext-class-out.mln -r hypertext-class.result
  -e hypertext-class-test.db -q Topic
```

These rules represent a special case of collective classification. MLNs for many other collective classification tasks can be expressed in this manner; for example, social network modeling can be achieved by replacing `LinkTo()` with `Friends()` and determining topics of a blog in which they participate instead of the topic of web pages.

5.2 Information Retrieval

With the growth of the Internet, information retrieval has grown into an important field. The task is to retrieve all documents relevant to a query of several words. A simple approach, vector-space information retrieval, can be easily implemented in Alchemy. We represent the words in a document with the `HasWord` predicate as in the previous section. Additionally, the predicate `InQuery(w)` is true iff `w` is in our query. The relevance of a page to our query is expressed by the predicate `Relevant(page)`. Our simple MLN for information retrieval looks like this:

```
InQuery(word)
HasWord(word, page)
Relevant(page)

InQuery(+w) ^ HasWord(+w, p) => Relevant(p)
```

As web search engines have shown, pages linked to relevant pages are also sometimes relevant. This is achieved by adding one formula to the MLN involving the `LinkTo` predicate:

```
Relevant(p1) ^ LinkTo(id, p1, p2) => Relevant(p2)
```

Of course, in order to scale to the internet much more work is needed in terms of indexing the documents, query processing, etc. but these two formulas represent the core of PageRank-style information retrieval.

6 Entity Resolution

Entity resolution is an important step of data cleaning and information extraction on which much research has been done. Markov logic allows an intuitive and elegant approach to this task. In order to demonstrate entity resolution with Alchemy, we take a look at the Cora dataset containing citations of computer science publications. Citations of the same paper often appear differently and the task here is to determine which citations are referring to the same paper. The model used here is based on that of [1].

We start with one basic evidence predicates, `HasToken`, telling us the actual text and the "field" (author, title, or venue) of each token in each citation, respectively. The predicate `HasToken(t, f, c)` tells us that token `t` is present in field `f` in citation `c`.

Given this evidence, we want to predict which citations are the same, indicated by the predicate `SameCitation`. We determine identical citations by looking at each of the fields author, title, and venue and determining their similarity. This is expressed by the predicate `SameField(f, c1, c2)`, where `f` is a field (author, title, or venue) and `c1` and `c2` are citations. To recap, the predicates we need are:

```
HasToken(token, field, citation)
SameField(field, citation, citation)
SameCitation(citation, citation)
```

The formulas we need to perform entity resolution are very compact thanks to the per-constant `+` operator. This can be used during weight learning to produce a separate clause (and, hence, learn a weight) for each value of the variable to which it is applied. For example, the first rule for entity resolution we want to express "If the same token occurs in the same field in two separate citations, then the field is the same"; we want to do this for each token and field pair. In Markov logic, this looks like

```
Token(+t, i1, c1) ^ InField(i1, +f, c1) ^ Token(+t, i2, c2)
   ^ InField(i2, +f, c2) => SameField(+f, c1, c2)
```

Also, we want to make the connection from same field to same citation, doing it for each field:

```
SameField(+f, c1, c2) => SameCitation(c1, c2)
```

Finally, we want to add transitivity to the model (if `c1` and `c2` are the same citation and `c2` and `c3` are the same citation, then `c1` and `c3` are the same citation):

```
SameCitation(c1, c2) ^ SameCitation(c2, c3) => SameCitation(c1, c3)
```

We run weight learning on the MLN and data with the following command:

```
learnwts -d -i er.mln -o er-out.mln -t cora-seg-train.db
  -ne SameField,SameCitation
```

which produces the clauses with learned weights in the file `er-out.mln`. We can use this to perform inference on the test data:

```
infer -ms -i er-out.mln -r er.result -e cora-seg-test.db
  -q SameField,SameCitation
```

The file `er.result` then contains the marginal probabilities of the query predicates. More refinements of this model exist which improve the results significantly; for example we could add transitivity on the `SameField` predicate. For the state-of-the-art model in Markov logic for entity resolution, see [4] and [3].

7 Hidden Markov Models

A very effective and intuitive approach to many sequential pattern recognition tasks, such as speech recognition, protein sequence analysis, machine translation, and many others, is to use a hidden Markov model (HMM). We demonstrate the modeling of an HMM on two examples. In order to convey the translation of HMMs to MLNs, we use a small traffic example. We then show a real-world example of applying an HMM in Markov logic to perform segmentation of Cora citations.

Suppose, on a given day, we observe a car taking three actions: it is either stopped, driving, or slowing. We assume this is only dependent on the state of the stoplight in front of it: red, green or yellow. In a Markov process we need to model *states* and *observations* at certain points in *time*. In Alchemy, we model a state and observation with a first-order predicate and time is a variable in each of these predicates, i.e.:

```
state = {Stop, Drive, Slow}
obs = {Red, Green, Yellow}
time = {0,...,10}

State(state, time)
Obs(obs, time)
```

In Markov logic we need to explicitly express some constraints which are inherent to HMMs; in particular, we need to state that at each time step there is exactly one state and observation. This is achieved with the `!` operator:

```
State(s!, t)
Obs(o!, t)
```

Now, we need for each state and its successor, the probability of this *transition* and for each observation-state pair, we need the probability of this *emission*. In addition, we need the prior probability of each state at the starting time point. These probabilities can be represented in Alchemy in three formulas by using the `+` operator:

```
State(+s, 0)
State(+s1, t) => State(+s2, t+1)
Obs(+o, t) => State(+s, t)
```

By using the `+` operator, we generate a clause for each state/state and observation/state pair, thus enabling us to learn weights for each of these combinations. These are the observation and transition matrices of our HMM.

We can learn weights from the data with:

```
learnwts -d -i traffic.mln -o traffic-out.mln -t traffic-train.db -ne State
```

Then, we can find the most likely sequence of states by querying all `State` atoms:

```
infer -m -i traffic-out.mln -r traffic.result -e traffic-test.db -q State
```

Although this is a toy example, it showcases the intuitiveness and many of the capabilities of Alchemy: block variables, weight learning with EM and compactness of representation. We now move on to a real-world application of HMMs, segmentation of citations, and how to achieve this with Alchemy.

8 Information Extraction

In Section 6, we assumed the citation data was already segmented, i.e. it was known which text belonged to which field, making the `InField` an evidence predicate. In general, this is not the case and we need to segment the text into its fields. A common approach is to use an HMM and we demonstrate how to do this with Alchemy on the unsegmented Cora data.

Now, the `InField` predicate is non-evidence, leaving only the text, in form of the `Token` predicate as our only evidence. We model the observation matrix of our HMM with the rule

```
Token(+t, i, c) => InField(i, +f, c)
```

and the transition matrix via

```
InField(i, +f, c) => InField(i+1, +f, c)
```

We also want to impose the constraint that a position in a citation can be part of at most one field, thus:

```
!(f1 = f2) => (!InField(i, +f1, c) v !InField(i, +f2 ,c))
```

We learn the weights with the call:

```
bin/learnwts -d -i seg.mln -o seg-out.mln -t cora-unseg-train.db -ne InField
```

and use the output file to perform inference with:

```
bin/infer -ms -i seg-out.mln -r seg.result -e cora-unseg-test.db -q InField
```

Here, we have treated segmentation and entity resolution as two separate tasks. This is a valid approach; however, the information obtained from the entity resolution step can be utilized to improve the segmentation. For the state-of-the-art of this type of joint inference in information extraction using Markov logic, see [3].

9 Natural Language Processing

9.1 Statistical Parsing

Statistical parsing is the task of computing the most probable parse of a sentence given a probabilistic (or weighted) context-free grammar (CFG). The weights of the probabilistic or weighted CFG are typically learned on a corpus of texts. Here, we demonstrate how to translate a given CFG to an MLN and learn weights for the grammar on a small corpus of simple sentences.

A context-free grammar in Chomsky normal form consists of rules of the form:

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \end{aligned}$$

where A, B, and C are non-terminal symbols and a is a terminal symbol. Here, we want to construct a grammar for parsing simple English language sentences. One such simple grammar might look like this:

$$\begin{aligned} S &\rightarrow NP\,VP \\ NP &\rightarrow Adj\ N \\ NP &\rightarrow Det\ N \\ VP &\rightarrow V\ NP \end{aligned}$$

Here, S stands for sentence, NP for noun phrase, VP for verb phrase, Adj for adjective, N for noun, Det for determiner, and V for verb. To translate this into an MLN, we first encode each production rule in the grammar as a clause:

```
NP(i,j) ^ VP(j,k) => S(i,k)
Adj(i,j) ^ N(j,k) => NP(i,k)
Det(i,j) ^ N(j,k) => NP(i,k)
V(i,j) ^ NP(j,k) => VP(i,k)
```

Here, `i` and `j` indicate indices between the words, including at the beginning and at the end of the sentence. So, a sentence with n words has $n + 1$ indices. For example, the words in the sentence "The dog chases the cat" would be indexed as $_0$ The $_1$ dog $_2$ chases $_3$ the $_4$ cat $_5$.

In addition, we need a lexicon for our text telling us the parts of speech of each word. For our small corpus, it looks like this:

```
// Determiners
Token("a",i) => Det(i,i+1)
Token("the",i) => Det(i,i+1)
```

```
// Adjectives
Token("big",i) => Adj(i,i+1)
Token("small",i) => Adj(i,i+1)

// Nouns
Token("dogs",i) => N(i,i+1)
Token("dog",i) => N(i,i+1)
Token("cats",i) => N(i,i+1)
Token("cat",i) => N(i,i+1)
Token("fly",i) => N(i,i+1)
Token("flies",i) => N(i,i+1)

// Verbs
Token("chase",i) => V(i,i+1)
Token("chases",i) => V(i,i+1)
Token("eat",i) => V(i,i+1)
Token("eats",i) => V(i,i+1)
Token("fly",i) => V(i,i+1)
Token("flies",i) => V(i,i+1)
```

If we left it at this, then two problems would occur. First, if there are two or more production rules with the same left side (such as $NP \rightarrow Adj\ N$ and $NP \rightarrow Det\ N$), then we need to enforce the constraint that only one of them fires. This is achieved with a rule of the form:

```
NP(i,k) ^ Det(i,j) => !Adj(i,j)
```

Basically, this is saying "If a noun phrase results in a determiner and a noun, it cannot result in and adjective and a noun". The second problem involves ambiguities in the lexicon. If we have homonyms belonging to different parts of speech, such as `Fly` (noun or verb), then we have to make sure that only one of these parts of speech is assigned. We can enforce this constraint in a general manner by making mutual exlcusion rules for each part of speech pair, i.e.:

```
!Det(i,j) v !Adj(i,j)
!Det(i,j) v !N(i,j)
!Det(i,j) v !V(i,j)
!Adj(i,j) v !N(i,j)
!Adj(i,j) v !V(i,j)
!N(i,j) v !V(i,j)
```

We now have a complete MLN (consisting of the grammar and the lexicon) for which we can learn weights based on training data. We treat each sentence as a separate database, thus the call is

```
learnwts -d -i cfg.mln -o cfg-out.mln -t s1-train.db,s2-train.db,...,
  s12-train.db -ne NP,VP,N,V,Det,Adj -multipleDatabases
```

This produces the weighted CFG in `cfg-out.mln` with which we can parse our test sentences:

```
infer -m -i cfg-out.mln -r parse.result -e test/s1-test.db
  -q NP,VP,N,V,Det,Adj
```

The extension to weighted definite clause grammars is intuitive because Markov logic can capture relations between parts of speech by simply adding an argument to the predicate. For example, if we want to enforce noun-verb agreement, we would add the number to the noun phrase and verb phrase predicates, i.e. `NP(pos, pos, num)` and `VP(pos, pos, num)` and our top-level rule would become `NP(i,j,n) ∧ VP(j,k,n) => S(i,k)`.

9.2 Other NLP tasks

The field of NLP concerns itself with many other tasks besides parsing, such as word sense disambiguation, semantic role labeling, etc. Alchemy lends itself well to many of these; however, extensive research has not been done yet in applying Alchemy to these areas.

10 Bayesian Networks

Bayesian networks are one of the most popular and widespread graphical models and many people from fields other than AI and machine learning are familiar with them. This framework can be easily implemented in Markov logic; we show how this is done on the classic ALARM Bayesian network used to monitor patients in intensive care units. It contains 37 nodes and 46 arcs.

In a Bayesian network, nodes represent discrete variables and arcs the dependencies between them. A conditional probability table (CPT) is associated with each node indicating the probability distribution for the variable conditioned on its parents. We want to represent every variable in a Bayesian network as a predicate; thus, we declare for each variable `Var` a unary predicate `Var(varValue!)`, indicating that each variable can only take on one of its values (i.e. if `CVP` is `LOW`, it cannot be `NORMAL` or `HIGH`). This results in the following predicate declarations:

```
HISTORY(historyValue!)
CVP(cvpValue!)
PCWP(pcwpValue!)
HYPOVOLEMIA(hypovolemiaValue!)
LVEDVOLUME(lvedvolumeValue!)
LVFAILURE(lvfailureValue!)
STROKEVOLUME(strokevolumeValue!)
```

```
ERRLOWOUTPUT(errlowoutputValue!)
HRBP(hrbpValue!)
HREKG(hrekgValue!)
ERRCAUTER(errcauterValue!)
HRSAT(hrsatValue!)
INSUFFANESTH(insuffanesthValue!)
ANAPHYLAXIS(anaphylaxisValue!)
TPR(tprValue!)
EXPCO2(expco2Value!)
KINKEDTUBE(kinkedtubeValue!)
MINVOL(minvolValue!)
FIO2(fio2Value!)
PVSAT(pvsatValue!)
SAO2(sao2Value!)
PAP(papValue!)
PULMEMBOLUS(pulmembolusValue!)
SHUNT(shuntValue!)
INTUBATION(intubationValue!)
PRESS(pressValue!)
DISCONNECT(disconnectValue!)
MINVOLSET(minvolsetValue!)
VENTMACH(ventmachValue!)
VENTTUBE(venttubeValue!)
VENTLUNG(ventlungValue!)
VENTALV(ventalvValue!)
ARTCO2(arco2Value!)
CATHECOL(cathecolValue!)
HR(hrValue!)
CO(coValue!)
BP(bpValue!)
```

As shown in [2], we can convert a Bayesian network to a weighted CNF expression (an MLN) by producing a clause for each line and value of the variable and enforcing mutually exclusive and exhaustive constraints on the variables (we have already achieved this with the `!` operator. Each clause contains the negation of each variable in the row of the CPT and the weight is $-\ln(p)$, where p is the corresponding probability in the CPT. Entries with zero probability cannot be translated in this manner; however, this problem can be solved by making these lines of the CPT hard clauses and not negating the variables. For our example we arrive at the MLN found in `alarm.mln`.

Note, these weights could easily be learned from training data, either discriminatively or generatively. In the case of Bayesian networks, the partition function is known (it is 1) and the weights can be computed exactly.

References

[1] A. McCallum and B. Wellner. Conditional models of identity uncertainty with application to noun coreference. In *Proceedings of the NIPS 04*, 2004.

[2] James D. Park. Using weighted max-sat engines to solve mpe. In *Eighteenth National Conference on Artificial Intelligence*, pages 682–687, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[3] H. Poon and P. Domingos. Joint inference in information extraction. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence*, Boston, MA, 2007. AAAI Press. To appear.

[4] P. Singla and P. Domingos. Entity resolution with markov logic. In *Proceedings of the Sixth IEEE International Conference on Data Mining*, pages 572–582, Hong Kong, 2006. IEEE Computer Society Press.