A Bayesian Language for Cumulative Learning

Avi Pfeffer
Harvard University

Abstract

A cumulative learning agent is one that learns and reasons as it interacts with the world. The Bayesian paradigm provides a natural framework for cumulative learning as an agent can use its observations and prior models to reason about a particular situation, and also learn posterior models. Cumulative learning requires a rich, first-order representation language in order to handle the variety of situations an agent may encounter. In this paper, I present a new Bayesian language for cumulative learning, called IBAL. This language builds on previous work on probabilistic relational models, and introduces the novel feature of observations as an integral part of a language. The key semantic concept is a scope, which consists both of models and observations. The meaning of a scope is the knowledge state of an agent. The language is declarative, and knowledge states can be composed in a natural way. In addition to presenting a language, this paper also presents an EM based learning algorithm called functional EM for learning IBAL models.

Introduction

A cumulative learning agent is one that learns and reasons as it interacts with the world. In each encounter, it uses its accumulated knowledge and its observations to reason about the particular situation, and also updates its knowledge base by learning from its observations. The Bayesian paradigm provides a natural framework for cumulative learning. An agent's knowledge state in a particular encounter consists of its prior knowledge, which is a probability distribution over models, and its observations about the particular situation. The observations serve a dual purpose — to condition the agent's beliefs about unobserved aspects of the situation, and to update the probability distribution over models. The updated distribution can then be used as the agent's prior knowledge in its next encounter.

In addition to a framework for integrating the results of successive encounters into a learned model, cumulative learning requires a representation language that is sufficiently rich to describe the different situations the agent might encounter. In particular, the language must be modular and extensible, to allow the knowledge base to be extended and accumulated compositionally. Propositional or attribute-based representations are inadequate for this task. Inductive logic programming (MR94) has emerged as one approach to learning a rich first-order representation language, but the framework is not probabilistic.

In recent years, researchers have developed languages that integrate logical and probabilistic representations. One research direction has been to incorporate probabilities into logic programs (Mug99). Another approach extends the expressive power of Bayesian networks by integrating them with object-based and relational representations (Pfe00). There has also been work on learning such probabilistic relational models from data (FGKP99). Such languages provide a natural basis for cumulative learning. Their rich expressive power allows them to express the sort of knowledge a cumulative learner needs, while their probabilistic aspect allows them to use the Bayesian framework.

In this work, I extend the probabilistic relational modeling languages of (Pfe00) to support cumulative learning. I present a new language called IBAL (pronounced "eyeball"), standing for Integrated Bayesian Agent Language. The key difference between IBAL and previous languages is that observations are now made an integral component of the language. This means that observations can now modify the meaning of model components, which can then be used in other models. This corresponds to a learning agent modifying its models based on its observations, and using its models in future situations. In addition, since observations are an integral part of the language, each observation takes place within a certain *scope*. Each scope describes a knowledge state of an agent, consisting of the models and observations within the scope.

Thus, the language presented here is declarative. It provides a description of a cumulative agent by describing its successive knowledge states as a sequence of nested scopes. As such, it provides a language not only for designing cumulative learning agents but also for reasoning about them. In fact, the language goes beyond simply describing the knowledge states of a single learning agent. One can easily describe multiple agents in the language, having both shared and individual experiences. The knowledge states of different agents compose in a natural way.

In addition to describing a language, this paper presents a new algorithm for learning models in the language. The algorithm, called *functional EM*, is a variation on the expectation-maximization algorithm, that is specially tailored to take advantage of the structure of models represented in the language. The algorithm follows the lines of the structured probabilistic inference algorithm presented in (PKMT99). It was shown there that exploiting model structure yields great speedups for probabilistic inference, and I believe that the same will be true for learning.

Language

The basis for the IBAL language is the stochastic functional language of (KMP97). While that language is more functional in flavor than the probabilistic relational languages of (Pfe00), the expressive power is essentially the same. The ideas presented here can also be applied to probabilistic relational languages, but the key idea of using scoped observations is easiest to describe and implement in a functional framework.

The main structural unit of the language is a *block*. A block consists of three kinds of statements: *model definitions*, *variable definitions*, and *observations*. A model definition describes an experiment that generates values according to some probability distribution. The definition specifies the functional form of the experiment, but the probabilities involved are learnable parameters. The form of a model definition is "*name (args) = expression*". A variable definition associates an identifier with the outcome of an experiment, and has the form "*name = expression*". A variable may only be defined once in a block.

There are several kinds of expression. An expression may simply be a symbolic value, or a variable name preceded by a $ sign. A `dist` expression defines a probability distribution over several expressions. It has the form "`dist` $p_1 : e_1, \ldots, p_n : e_n$", where each e_i is an expression, and each p_i is a positive real number. The p_i are not bounded by 1 or required to sum to 1. In fact, they do not directly represent probabilities, but rather a Dirichlet prior over which of the e_i is chosen. The values of the p_i for each of the `dist` expressions in a model are the parameters of the model.

There are several other kinds of expression: A `case` expression selects one of a number of possibilities, according to the value of some variable. An application expression applies a model to a set of arguments, each of which is an expression. The ability to pass arguments and receive values from a model are what make this language relational, and give it essentially the same expressive power as probabilistic relational models. The model to apply may itself be an expression whose outcome is stochastic — this feature implements a form of reference uncertainty, as described in (Pfe00). Finally, there is the block expression, which is a block enclosed between curly braces.

One can think of executing an expression to produce a value, just as in an ordinary programming language. The only difference is that the value produced is determined stochastically. In particular, for a *dist* expression, one of the possible subexpressions is executed, accoridng to the specified distribution.

An expression may either be *simple* or *complex*. Symbols are simple expressions, while block expressions are complex. A `case` or `dist` expression is complex if any of its possible results are complex. A model application is simple or complex depending on whether the expression defining the model is simple or complex. An *attribute chain* extracts components of a variable defined by a complex expression.

An *observation* has the form "*simple-attribute-chain* `is` *value*". In the full language, expressions are typed, and one can ensure that observations are well-typed, but I do not go into details here.

To summarize, a block consists of probability models, variables, and observations about the variables. The expressions defining the models and variables may themselves contain nested blocks, which have their own models, variables and observations. Each block defines a *scope*, consisting of the variables and models defined in the block, and all observations either directly in the block or within some nested block. Models and variables, together with any observations in their definitions, can be incorporated into a scope using an `import` statement. The import statements must be non-recursive — a block is not allowed to import itself, nor may two blocks import each other. A program, which is just a top-level block, defines a hierarchy of nested scopes.

In this language, a scope serves two purposes. There is the traditional programming language purpose, which is to determine what names are available for use inside the scope. There is also an additional purpose, which is to provide a context for the observations. If an observation occurs in a scope, it also occurs in all the scopes that contain that scope. Scopes therefore provide us with a natural way to compose experiences represented by sets of observations.

Semantically, each scope is interpreted as the knowledge state of an agent. The knowledge state consists of prior knowledge derived from the model definitions, and all the observations within the scope. Between them, the prior knowledge and observations define a posterior probability distribution over model parameters, and over the values of the variables defined in the scope.

Examples

Perhaps the IBAL language is more easily appreciated by seeing a few examples. The first example illustrates some of the basic features of the language.

```
fair() = dist 50:heads, 50:tails
```

```
biased() = dist 90:heads, 10:tails
pick() = dist 1:fair, 1:biased
pick_and_toss() = dist 1:fair(), 1:biased()
c = pick()
x = $c()
y = $c()
z = pick_and_toss()
w = pick_and_toss()
x is heads
z is heads
```

This program contains models for fair and biased coins, for the act of picking (but not tossing) a coin, and for the act of picking and tossing a coin. The variable c represents the result of picking a coin. The variables x and y represent the result of two different tosses of c. Note the $ signs and parentheses appearing in the definitions of x and y. The $ signs indicate that the model used to generate x and y is determined by the value of c. The parentheses indicate that the value of $c is computed, and then applied (with no arguments), to produce x and y. This is an example of reference uncertainty, and illustrates the power of allowing the model in an application expression to be an expression itself.

The values of x and y are correlated due to their mutual dependence on the identity of c. The observation of x provides evidence as to the identity of c, and therefore affects our beliefs about y. We should also update the pick, fair and biased models based on the observation of x. The variables z and w, on the other hand, represent the results of two independent experiments involving picking and tossing a coin. The observation of z provides no evidence as to the value of any other variable, but it does allow us to update the models of pick_and_toss, fair and biased.

The next example illustrates how the IBAL language can be used to represent probabilistic relational models. The example is based on the movie domain of (FGKP99). This domain is represented by a relational database containing three relations: The MOVIE relation describes movies and their properties, such as the *decade* in which they were produced, their *genre*, and whether they are in *color*. The ACTOR relation describes actors, and includes a *gender* attribute. The APPEARANCE relation relates actors to movies they appeared in. It includes a *role-type* attribute, indicating the type of role (e.g. hero, villian) played by the actor in the movie. A relational probabilistic model for this domain specifies a probability distribution over the values of attributes of tuples in each of the relations. The value of an attribute may depend on values of attributes of related tuples, so, for example, the *role-type* played by an actor in a movie may depend on the *gender* of the actor and the *genre* of the movie. A model for this domain can be defined in IBAL as follows:

```
movie() = {
 decade = dist ...
 color = case decade of ...
 genre = case decade of ...
}

actor() = {
 gender = dist ...
}

appearance(a,m) = {
 role_type = case a.gender of
  male :  case m.genre of
   western :  dist ...
   ...
  female :  case m.genre of
   ...
}
```

The above model specifies the database schema, as well as the structure of a probabilistic model over this schema. As shown below, an actual database can then be specified using variable definitions to create tuples, and observations to assert attribute values. An IBAL program can easily be produced automatically from a database using a script.

```
modern_times = movie()
modern_times.decade is thirties
modern_times.color is false
modern_times.genre is comedy
...

charlie_chaplin = actor()
charlie_chaplin.gender is male
...

charlie_chaplin_in_modern_times =
 appearance(charlie_chaplin, modern_times)
charlie_chaplin_in_modern_times.role_type
 is innocent
...
```

The third example illustrates the use of scope to capture the successive knowledge states of a cumulative learning agent. In each episode, the agent learns a new model, which is then available for use in later episodes. For simplicity, the example illustrates an agent that uses the naive Bayesian classifier for its basic representation of models. A more sophisticated representation such as probabilistic relational models could certainly be used instead.

```
agent1 = {
 f1() = {
  c = dist ...
  x1 = case c of
   v1 :  dist ...
   ...
   vm :  dist ...
  ...
  xn = ...
 }
 e1_1 = f1()
 e1_2 = f1()
 ...
 e1_1.c is v3
 e1_1.x1 is ...
}

agent2 = {
 import agent1
 f2() = { ... /* uses f1 as a feature */ }
 e2_1 = f2() ...
 e2_1.c is ...
}
...
```

Notice how each successive agent state is a distinct semantic entity. This ability of the language to declaratively describe the knowledge state of an agent has powerful implications for representing situations with multiple agents, as illustrated in the following example.

```
example_generator() = { ... }

agent1 = {
 import example_generator
 test1(x) = { ...
  outcome = ...
 }
 e1 = test1(example_generator())
 e2 = test1(example_generator())
 ...
 e1.outcome is ...
}

agent2 = {
 import example_generator
 test2(x) = { ...
  outcome = ...
 }
 e1 = test2(example_generator())
 e2 = test2(example_generator())
 ...
 e1.outcome is ...
}

agent3 = {
 import agent1
 import agent2
}
```

In this example, there is some model that generates instances. Agents 1 and 2 both have a test they can run on the instances, and both run a set of experiments using their test, but they are unaware of each other. As a result, they will learn separate models of the instance generator. Agent 3 incorporates the knowledge of agents 1 and 2, and will learn a single unified model of both tests and the instance generator.

The Functional EM Algorithm

I have presented a powerful language for describing complex probabilistic models and observations obtained about those models. In order to use the language effectively, we need two algorithms. One is a probabilistic inference algorithm, that allows us to compute beliefs about unobserved variables given the observed variables and particular values for the model parameters. The algorithm for this language is essentially that of (PKMT99). It is based on the variable elimination algorithm for Bayesian network inference, but also exploits the functional structure of the model.

The second algorithm needed is a learning algorithm. Ideally, we would want one that learns a posterior distribution over models given the priors and the observations, which is exactly the semantics of an agent's knowledge state. Unfortunately, as in the case of Bayesian networks with missing data, this is too difficult, since the posterior is generally not Dirichlet (Hec98). Instead, we compromise and learn the maximum a-posteriori (MAP) models.

The algorithm is based on the expectation-maximization (EM) algorithm (MK97), but like the probabilistic inference algorithm it follows the functional form of the model, so I call it *functional EM*.

Consider an agent, defined by a scope. We want to learn the agent's MAP distributions for the models defined within the scope, based on all the observations in the scope. The model parameters are the distributions associated with each `dist` expression defined in the scope. As always, the EM algorithm relies on the notion of sufficient statistics. In our case, the sufficient statistics are the number of times each of the possible outcomes of a `dist` expression is chosen. Note that the statistics are *lexically* associated with `dist` expressions. If the same `dist` expression is executed multiple times during an experiment, the choices taken are all clumped together into the same statistic.

Given the values of the sufficient statistics, the maximization step of the EM algorithm is completely standard and trivial. In fact, the MAP distribution is the sum of the Dirichlet priors and the sufficient statistics, normalized. All the work, therefore, is in the expectation step, in which the expected sufficient stastistics are computed.

The algorithm for computing expected sufficient statistics given current model parameters is recursive. Consider, first, the top-level block in a program. The definitions within the block, together with the current model parameters, define a probability distribution over the course of execution of the experiments generating the variables. The observations serve to condition this distribution, by rejecting any execution runs that produce values disagreeing with the observations. To get the expected sufficient statistics, we need to figure out the expected number of times each `dist` expression was executed, and the expected number of times each of the possible branches was chosen. To make the book-keeping simple, we create explicit variables to represent the outcomes of `dist` expressions, by converting the expression

$\texttt{dist}\, p_1 : e_1, \ldots, p_n : e_n$

into the expression

case v of $1 : e_1, \ldots, n : e_n$

where v is a new dummy variable whose definition is

$v = \texttt{dist}\, p_1 : 1, \ldots, p_n : n$

The sufficient statistics are then the number of times each of these dummy variables has a particular value.

Now consider a nested block B within a block A. B may be executed multiple times within a particular execution of block A — let us just consider one of those executions. For that particular execution, let $\mathbf{X}$ be the free variables in B whose values are defined in A, and $\mathbf{Y}$ be the variables that are defined in B and later used in A. Let $\mathbf{Z}$ be the other variables in A that are not defined in this execution of B (including those contained in other executions of B), and $\mathbf{W}$ the other variables defined in this execution of B that are not used by A.

Between them, $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$ and $\mathbf{W}$ are all the variables defined inside A, and define values for all the attribute chains defined inside A. Consider any statistic $N^{v,i}$, which is the number of times a particular dummy `dist` variable v has a certain value i. v may appear multiple times at the end of attribute chains in $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$ and $\mathbf{W}$. An assignment of values $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ and $\mathbf{w}$ defines a value for $N^{v,i}$. We can decompose $N^{v,i}$ into a sum $N_X^{v,i} + N_Y^{v,i} + N_Z^{v,i} + N_W^{v,i}$, where $N_X^{v,i}$ is the number of attribute chains in $\mathbf{X}$ ending with v that have the value i, and so on.

In the terminology of (PKMT99), $\mathbf{X}$ and $\mathbf{Y}$ are the *interface* of (this execution of) B, and $\mathbf{W}$ is d-separated from $\mathbf{Z}$ by $\mathbf{X}$ and $\mathbf{Y}$. Therefore $P(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{w}) = P(\mathbf{x})P(\mathbf{y} \mid \mathbf{x})P(\mathbf{w} \mid \mathbf{x},\mathbf{y})P(\mathbf{z} \mid \mathbf{x},\mathbf{y})$. It follows that the expected sufficient statistic $E[N^{v,i}]$ can be decomposed as follows:

$$\begin{aligned}&E[N_X^{v,i} + N_Y^{v,i} + N_Z^{v,i} + N_W^{v,i}] = \\ &\quad E[N_X^{v,i}] + \textstyle\sum_{\mathbf{x}} P(\mathbf{x})E[N_Y^{v,i} \mid \mathbf{x}] + \\ &\quad \textstyle\sum_{\mathbf{x},\mathbf{y}} P(\mathbf{x})P(\mathbf{y} \mid \mathbf{x})(E[N_W^{v,i} \mid \mathbf{x},\mathbf{y}] + E[N_Z^{v,i} \mid \mathbf{x},\mathbf{y}])\end{aligned}$$

Now, in this expression, the terms $E[N_Y^{v,i} \mid \mathbf{x}]$, $P(\mathbf{y} \mid \mathbf{x})$, and $E[N_W^{v,i} \mid \mathbf{x},\mathbf{y}]$ are computable completely within B. This in fact holds not just for this particular sufficient statistic $N^{v,i}$ but for all sufficient statistics. The functional EM algorithm is therefore recursive, computing as much as possible within nested blocks. At the core of the algorithm is a function **SolveBlock**, taking the following arguments:

A block B.

A subset $\mathbf{Y}$ of the variables defined inside B.

Values $\mathbf{y}$ for $\mathbf{Y}$.

SolveBlock returns the following:

A set of free variables $\mathbf{X}$ in B whose values are needed in order to compute $\mathbf{Y}$.

For each such variable X, a range of values $Dom[X]$ that is consistent with the value of $\mathbf{Y}$ being $\mathbf{y}$.

$P(\mathbf{y} \mid \mathbf{x})$ for each $\mathbf{x}$ in $Dom[\mathbf{X}]$.

For each sufficient statistic $N^{v,i}$, the expected contribution to $N^{v,i}$ from variables defined inside B, given each $\mathbf{x}$ and $\mathbf{y}$.

SolveBlock works in two phases. In the first phase, it computes the following information, by processing each of the variable definitions from the bottom up:

For each variable Z, a set $Needed[Z]$ is computed, consisting of simple attribute chains beginning with Z that are relevant to the observations.

For each attribute chain $Z.\sigma \in Needed[Z]$, we compute a set of values $Dom[Z.\sigma]$ that are consistent with the observations.

If Z is a simple variable, the parents $\mathbf{U}$ of Z and $P(Z = z \mid \mathbf{U} = \mathbf{u})$, for all $Z \in Dom[z]$ and $u \in Dom[U]$ is computed from the expression defining Z.

If Z is complex, let B be the block defining Z, and let $\mathbf{Y} = Needed[Z]$. For each value $\mathbf{y} \in Dom[\mathbf{Y}]$, a recursive call to **SolveBlock** is performed with arguments B, $\mathbf{Y}$ and $\mathbf{y}$. The result of each such recursive call is a set of attribute chains $\mathbf{X}$ used in B, possible values $\mathbf{x}$ for those chains, $P(\mathbf{y} \mid \mathbf{x})$ for each $\mathbf{x} \in Dom[\mathbf{X}]$, and for each sufficient statistic $N^{v,i}$ and values $\mathbf{x}$ and $\mathbf{y}$, $E[N_Z^{v,i} \mid \mathbf{x},\mathbf{y}]$, which is the contribution to the statistic resulting from the execution of Z, given that $\mathbf{X}$ and $\mathbf{Y}$ have the given values.

In the second phase of **SolveBlock**, the information computed in the first phase is used to compute sufficient statistics. First, for each simple dummy variable v, we compute the distribution over the value of v using standard BN inference, and add $P(v = i)$ to $E[N^{v,i}]$. Then, for each complex variable Z, we have already computed expected sufficient statistics $\mathbf{N}_Z$ resulting from the computation of Z given each possible set of inputs $\mathbf{u}$ and observations $\mathbf{y}$ on the output. We therefore compute $P(\mathbf{u},\mathbf{y})$ using standard BN inference, and add

$$\sum_{\mathbf{u} \in Dom[\mathbf{U}], \mathbf{y} \in Dom[\mathbf{Y}]} P(\mathbf{u},\mathbf{y})E[\mathbf{N}_Z \mid \mathbf{u},\mathbf{y}]$$

to the expected sufficient statistics.

This is the core of the learning algorithm. Note that the same advantages that were obtained from using a structured algorithm for probabilistic inference are also achieved by functional EM. There are two advantages in particular. The fact that the computation of statistics for nested scopes is performed via a recursive call exploits the fact that most of the internal details of the called scope are encapsulated from the calling scope. In addition, if the same block is called from multiple places with the same possible values for relevant inputs and outputs, the same set of nested sufficient statistics will be returned. The nested sufficient statistics therefore only need to be computed once, and reused each time the block is called. Both these advantages were shown to produce major speedups for inference in (PKMT99), and I believe the same will be true for learning.

Discussion

IBAL is a work in progress. I am currently developing an implementation, which will hopefully be made publically available for research purposes. In addition to observations, I intend to make decisions and utilities basic components of the language — hence the "integrated" part of the name. My hope is that it will develop into a system that can be used for rapidly designing new representations, agent architectures and inference algorithms, and can be applied to a wide variety of domains.

With regard to learning, this paper can be extended in several ways. The functional EM algorithm learns a single agent's models. The language allows us to describe a cumulative learning agent, or multiple agents with different knowledge states, and one would like to learn the models of the different agents in a compositional manner. It is possible, though it remains to be seen whether this will work out in practice, that learning the scopes in a program from the bottom up will turn out to work well. The idea is that if each of the nested scopes contains observations, the observations in the innermost scopes can be used to quickly learn good probability models in the inner scopes. Even though these models may need to be updated based on observations in higher level scopes, they should provide a good starting point for learning on the higher level.

The functional EM algorithm presented here assumes that the recursive procedure eventually terminates. This is the case for the languages discussed in (PKMT99) and (FGKP99). However, in (PK00), we allow for the possibility of infinite recursion by developing an iterative anytime approximate inference algorithm. It should be possible to extend functional EM to deal with the infinitely recursive case. EM is also an iterative algorithm. This raises the intriguing possibility of interleaving the two iterative processes into a single lazy process.

Finally, the language and algorithm presented here only allows for learning the model parameters, and not model structure. Because of the ability of the language to represent uncertainty over which of several models gets executed (as in the first example), even this limited language is quite powerful. However, it would be interesting to try to learn model structure as well, as in (FGKP99).

References

N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proc. IJCAI*, 1999.

D. Heckerman. A tutorial on learning with Bayesian networks. In M. I. Jordan, editor, *Learning in Graphical Models.* MIT Press, Cambridge, MA, 1998.

D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *Proc. AAAI*, 1997.

G.J. McLachlan and T. Krishnan. *The EM Algorithm and Extensions.* Wiley Interscience, 1997.

S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

S. Muggleton. Stochastic logic programs. *Journal of Logic Programming*, 1999. Accepted subject to revision.

A.J. Pfeffer. *Probabilistic Reasoning for Complex Systems.* PhD thesis, Stanford University, 2000.

A. Pfeffer and D. Koller. Semantics and inference for recursive probability models. In *Proc. AAAI*, 2000.

A. Pfeffer, D. Koller, B. Milch, and K.T. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Proc. UAI*, 1999.