Chapter 5

PRMs with Class Hierarchies

Here we examine the benefit that additional domain structure in the form of IS-A hierarchies provides during the construction of PRMs. We show how the introduction of subclasses allows us to use inheritance and specialization to refine our models so that they more accurately capture the dependencies in the data. We show how to learn PRMs with class hierarchies (PRM-CH) in two settings. In the first, the class hierarchy is provided, as part of the input, in the relational schema for the domain. In the second setting, in addition to learning the PRM, we must learn the class hierarchy. An important benefit provided by the class hierarchy mechanism is that PRM-CHs allow us to build models that can refer to both particular instances in our domain, and classes of objects in our domain. This capability allows us to bridge the gap between a class-based model and an attribute-value-based model.

5.1 Introduction

Consider the problem of collaborative filtering [Resnick et al., 1994, Shardanand et al., 1995, Breese et al., 1998], in which we utilize collected individual preferences to make a collaborative recommendation. Ideally, by finding individuals with similar tastes or preferences to a new user, we can use their combined preferences to make predictions for the new user's ratings for services or products. The canonical example is a movie recommender system, in which we have the rankings by large number of people on

a collection of movies. We then make recommendations by finding users who have similar taste to the new user (i.e., who have given similar rankings on some subset of the movies), and, for movies the new user has not yet seen, use the collective prediction of similar users to predict the new user's reaction.

One method for building a recommender system is to build a model of movie viewers and the movies that they enjoy. One approach, though certainly not the most common one, is to build a Bayesian network over the movies. The model represents the preferences of a single viewer, which has a random variable for each movie. Each person in the training set has a vector that represents the movies that they have watched, and their ratings for that movie. From this data set, we can learn a Bayesian network that represents the correlations between their preferences for the different films. Thus, we could learn that the user's rating of one movie, say "Forrest Gump", depends on her ratings for the movies that are its parents in the learned network, say "Big" and "Caddy Shack". Breese et al. [1998] compare this approach to other collaborative filtering approaches and show that it is superior in its ability to predict TV-show preferences.

This approach is limited in that it models only the relationships between instances of one class, the movies. We cannot model broad dependencies, such as whether a person enjoys British comedies depends on whether they like American slap-stick comedies. In addition, we cannot model relationships between people. For example, if my roommate recommends "Four Weddings and a Funeral", I may be more likely to take her advice and rent the movie (assuming that I think we have similar tastes; her recommendation may also have the opposite effect motivating me to avoid the movie entirely).

As we have seen, PRMs allow us to represent rich dependency structures, involving multiple entities and the relations between them; they allow the attributes of an entity to depend probabilistically on properties of related entities. However, PRMs model the domain at the *class* level; i.e., all instances in the same class share the same dependency model. This model is then instantiated for particular situations. For example, a person's ratings for a movie can depend both on the attributes of the person and the attributes of the movie. For a given situation, involving some set of

people and movies, this dependency model will be used several times. This allows us, for example, to use the properties and ratings of one person to reach conclusions about the properties of a movie (e.g., how funny it is), and thereby to reach conclusions about the chances that another viewer would like it.

In Chapter 3 we showed how to learn PRMs from relational data, and presented techniques for learning both parameters and the probabilistic dependency structure for the attributes in a relational model. This learning algorithm exploits the fact that the models are constructed at the class level. Thus, an observation concerning one person and one movie is used to refine the class model applied to all people and all movies, hence making much broader use of our data. This has the advantage that we can learn robust statistical models despite the fact that each individual may have seen only a few movies and each movie may have been seen by only a small number of people.

However, this class-based approach also has disadvantages: all elements of the same class must use the same model. For example, we cannot have the rating of a user for documentaries depend on one set of parents, and his ratings for comedies depend on another set of parents. Thus, we can not specialize the local probability models depending on the movie category or class. In addition, we cannot have the rating for documentaries depend on the rating for action movies (here for example, the correlation may be negative). The dependency model for these two ratings must be identical, and we cannot have the rating for a movie depend on itself. Finally, we cannot have the rating for a particular movie such as "Forest Gump" depend on the rating for another movie instance such as "Big": The dependency model for these two ratings must be identical, and we cannot have the rating for a movie depend on itself.

In this chapter, we propose methods for discovering useful refinements of a PRM's dependency model. We begin in Section 5.2 by defining *Probabilistic Relational Models with Class Hierarchies* (PRMs-CH). PRMs-CH extend PRMs by including class hierarchies over the objects. Subclasses allow us to specialize the probabilistic model for some instances of a class. For example, we might consider subclasses of movies, such as documentaries, action movies, British comedies, etc. The popularity of an

action movie (a subclass of movies) may depend on its budget, whereas the popularity of documentary (another subclass of movies) may depend on the reputation of the director. Subclassing allows us to model probabilistic dependencies at the appropriate level of detail. For example, we can have the parents of the popularity attribute in the action movie subclass be different than the parents of the same attribute in the documentary subclass. In addition, subclassing allows additional dependency paths to be represented in the model, that would not be allowed in a PRM that does not support subclasses. For example, whether I enjoy action movies may depend on whether I enjoy documentaries. PRMs-CH provide a general mechanism that allow us to define a rich set of dependencies. In fact, they provide the basic representational power that will allow us to model dependency models for individuals (as done in Bayesian Networks) and dependency models for categories of individuals (as done in PRMs).

In Section 4.6 we turn to some of the practical issues involved in learning PRMs-CH. First, we examine the case where the class hierarchy is given as input, as part of the relational schema. Our learning task is then simply to choose the appropriate level at which to model the probabilistic dependencies — at the class level, or specialized according to some subclass. We then turn to the case where the class hierarchy is not provided, and in addition to learning the probabilistic model, we must also discover the structure of the class hierarchy. In Section 5.4, we present some experimental results illustrating how we have expanded the space of probabilistic models considered by our learning algorithm, and how this allows us to learn more expressive and more accurate models.

As described in chapter Chapter 2, PRMs extend the representational power of (propositional) BNs to relational domains. In chapters Chapter 3 and Chapter 4 we examined how to learn richer probabilistic relational models. We described the semantics of the models and proposed learning algorithms for models with attribute uncertainty and models with structural uncertainty. In this chapter, we describe models that combine a class-based relational model with more traditional instance-based Bayesian networks. The tool we will use to provide this capability is a class inheritance hierarchy.

5.2 PRMs with Class Hierarchies

In this section, we describe refinements of our probabilistic model using class hierarchies. To motivate our extensions, consider a simple PRM for the movie domain. Let us restrict attention to the three classes Person, Movie, and Vote. We can have the attributes of Vote depending on attributes of the person voting (via the slot Vote.*Voter*) and on attributes of the movie (via the slot Vote.*Movie*). However, given the attributes of all the people and the movie in the model, the different votes are (conditionally) independent and identically distributed. By contrast, in the BN model for this domain, each movie could have a different dependency model; we could even have one depend on the other.

5.2.1 Class Hierarchies

Our aim is to refine the notion of a class, such as Movie, into finer subclasses, such as "action-movies", "comedy", "documentaries", etc. Moreover, we want to allow recursive refinements of this structure. So that we might refine "action-movies" into the subclasses "spy-movies", "car-chase-movies", and "kung-fu-movies".

Formally, we introduce the notion of a probabilistic class hierarchy, similar to that introduced in Koller and Pfeffer [1997, 1998]. We assume that the original set of classes define, at the schema level, the structure of an object (attributes and slots associated with it). Unlike the subclass mechanism in Koller and Pfeffer [1997, 1998], subclasses do not change this object structure.

A class hierarchy for a class X defines an IS-A hierarchy for objects from class X. The root of the class hierarchy is simply class X itself. The subclasses of X are organized into an inheritance hierarchy. The leaves of the class hierarchy describe basic classes—these are the most specific characterization of objects that occur in the database. The interior nodes describe abstractions of the base-level classes. The intent is that the class hierarchy is designed to capture useful and meaningful abstractions in a particular domain.

More formally, a hierarchy $H[X]$ for a class X is rooted directed acyclic graph defined by a subclass relation $\prec$ over a a finite set of subclasses $\mathcal{C}[X]$. For $c, d \in \mathcal{C}[X]$,

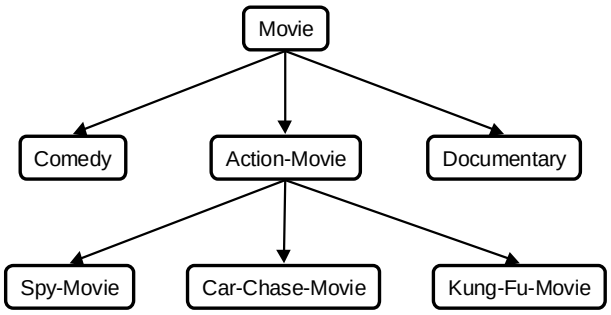


Figure 5.1: A simple class hierarchy for Movie.

if $c \prec d$, we say that X_c is a *direct subclass* of X_d, and X_d is a *direct superclass* of X_c. The root of the tree is the class X. $Class_\top$ corresponds to the original class X. We define $\prec^*$ to be the transitive closure of $\prec$; if $c \prec^* d$, we say that X_c is a subclass of X_d.

For example we may have the class Movie and its direct subclasses Comedy, Action-Movie, and Documentary. The subclass Action-Movie might, in turn, have the direct subclasses Spy-Movie, Car-Chase-Movie, and Kung-Fu-Movie. We have that Spy-Movie is a direct subclass of Action-Movie, and a subclass (but not a direct one) of the root class Movie. Figure 5.1 shows the simple class hierarchy we have just described.

We define the leaves of the hierarchy to be the *basic subclasses*, denoted $basic(H[X])$. We achieve subclassing for a class X by requiring that there be an additional subclass indicator attribute $X.Class$ that determines the basic class to which an object belongs (in theory objects could be members of a non-basic class, however we have not examined that possibility here). Thus, if c is a subclass, then $\mathcal{I}(X_c)$ contains all objects $x \in X$ for which $x.Class \prec^* c$, i.e., all objects that are in some basic class which is a subclass of c. In our example, Movie has a subclass indicator variable Movie.*Class* with the five possible values

$$\{Spy\text{-}Movie, Car\text{-}Chase\text{-}Movie, Kung\text{-}Fu\text{-}Movie, Comedy, Documentary\}$$

.

Subclasses allow us to make finer distinctions when constructing a probabilistic model. In particular, they allow us to *specialize* CPDs for different subclasses in the hierarchy.

Definition 5.1: A *probabilistic relational model with subclass hierarchy* is defined as follows. For each class $X \in \mathcal{X}$, we have

- a class hierarchy $H[X] = (\mathcal{C}[X], \prec)$;
- a subclass indicator attribute $X.Class$ such that $\mathcal{V}(X.Class) = basic(H[X])$;
- a CPD for $X.Class$ (here we require that $X.Class$ has no parents.
- for each subclass $c \in \mathcal{C}[X]$ and attribute $A \in \mathcal{A}(X)$ we have either
  - a set of parents $\mathrm{Pa}^c(X.A)$ and a CPD that describes $P(X.A \mid \mathrm{Pa}^c(X.A))$; or
  - an *inherited* indicator that specifies that the CPD for $X.A$ in c is inherited from its direct superclass. The root of the hierarchy cannot have the inherited indicator. ∎

With the introduction of subclass hierarchies, we can refine our probabilistic dependencies. Before each attribute $X.A$ had an associated CPD. Now, if we like, we can specialize the CPD for an attribute within particular subclass. We can associate a different CPD with the attributes of different subclasses. For example the attribute Action-Movie.*Popularity* may have a different conditional distribution from the attribute Documentary.*Popularity*. Further, the distribution for each of the attributes may depend on a completely different set of parents. Continuing our discussion from the introduction, if the popularity of action movies depends on its budget, then Action-Movie.*Popularity* would have as parents Action-Movie.*Budget*. However, for documentaries, the popularity depends on the reputation of the director; then Documentary.*Popularity* would have the parent Documentary.*Director.Reputation*.

We define $P(X.A \mid \mathrm{Pa}^c(X.A))$ to be the CPD associated with A in X_d, where d is the most specialized superclass of c (which may be c itself) such that the CPD of $X.A$ in d is not marked with the inherited indicator.

5.2.2 Refined Slot References

At first glance, the increase in representational power provided by supporting subclasses is deceptively small. It seems that little more than an extra constructed type variable has been added, and that the structure that is exploited by the new subclassed CPDs could just as easily have been provided using structured CPDs, such as the tree-structured CPDs or decision graphs [Boutilier et al., 1996, Chickering et al., 1997]. For example, the root node in the tree-structured CPD for attribute $X.A$ can split on the class attribute, $X.Class$, and then the subtrees can defined the appropriate specializations of the CPD. In reality, it is not quite so simple; now $X.A$ would need to have as parents the union of all of the parents of its subclasses. However, the representational power is quite similar.

However the representational power has been extended in a very important way. Certain dependency structures that would have been disallowed in the original framework are now allowed. These dependencies appear circular when examined only at the class level; however, when refined and modeled at the subclass level, they are no longer cyclic. One way of understanding this phenomenon is that, once we have refined the class, the subclass information allows us to disentangle and order the dependencies.

Returning to our earlier example, suppose that we have the classes Voter, Movie and Vote. Vote has reference slots *Person* and *Movie* and an attribute *Ranking* that gives the score that a person has given for a movie. Suppose we want to model a correlation between a person's votes for documentaries and his votes for action movies. (This correlation might be a negative one.) In the unrefined model, we do not have a way of referring to a person's votes for some particular subset of movies; we can only consider aggregates over a person's entire set of votes. Furthermore, even if we could introduce such a dependence, the dependency graph would show a dependence of Vote.*Rank* on itself.

When we create subclasses of movie, we can also create specializations of any classes that make reference to movies. For example Vote has a reference slot Vote.*Movie*. Suppose we create subclasses of Movie: Comedy, Action-Movie, and Documentary. Then we can create corresponding specializations of Vote: Comedy-Vote, Action-Vote,

and Documentary-Vote. Each of these subclasses refers only to a particular category of votes.

The introduction of subclasses of votes provides us with a way of isolating a person's votes on some subset of movies. In particular, we can try to introduce a dependence of Documentary-Vote.*Rank* on Action-Vote.*Rank*. In order to allow this dependency, we need a mechanism for constructing slot chains that restrict the types of objects along the path to belong to specific subclasses. Recall that a reference slot ρ is a function from Dom[ρ] to Range[ρ], i.e. from X to Y. We can introduce *refinements* of a slot reference by restricting the types of the objects in the range.

Definition 5.2:

Let ρ be a slot (reference or inverse) of X with range Y. Let d be a subclass of Y. A *refined slot reference* $\rho_{\langle d \rangle}$ for ρ to d is a relation between X and Y:

$$\text{For } x \in X, y \in Y,\ y \in x.\rho_{\langle d \rangle} \text{ if } x \in X \text{ and } y \in Y_d \text{ then } y \in x.\rho. \ \blacksquare$$

Returning to our earlier example suppose that we have subclasses of Movie: Comedy, Action-Movie and Documentary. In addition, suppose we also have subclasses of Vote, Comedy-Vote and Action-Vote and Documentary-Vote. To get from a person to their votes, we use the inverse of slot reference Person.*Votes*. Now we can construct refinements of Person.*Votes*, $Votes_{\langle \text{Comedy-Vote} \rangle}$, $Votes_{\langle \text{Action-Vote} \rangle}$ and $Votes_{\langle \text{Documentary-Vote} \rangle}$.

Let us name these slots *Comedy-Votes* and *Action-Votes*, and *Documentary-Votes*. To specify the dependency of a person's rankings for documentaries on their rankings for action movies we can say that Documentary-Vote.*Rank* has a parent which is the person's action movie rankings: γ(Documentary-Vote.*Person.Action-Votes.Rank*).

5.2.3 Support for Instance-level Dependencies

The introduction of subclasses brings the benefit that we can now provide a smooth transition from the PRM, a class-based probabilistic model, to models that are more similar to Bayesian networks. To see this, suppose our subclass hierarchy for movies is

very "deep" and starts with the general class and ends in the most refined levels with particular movie instances. Thus, at the most refined version of the model we can define the preferences of a person by either class based dependency (the probability of enjoying documentary movies depends whether the individual enjoys action movies) or instance based dependency (the probability of enjoying "Terminator II" depends on whether the individual enjoys "The Hunt for Red October"). The latter model is essentially the same as the Bayesian network models learned by Breese et al. [1998] in the context of collaborative filtering for TV programs.

In addition, the new flexibility in defining refined slot references allows us to make interesting combinations of these types of dependencies. For example, whether an individual enjoys a particular movie(e.g., "True Lies") can be enough to predict whether she watches a whole other category of movies (e.g., James Bonds Movies).

5.2.4 Semantics

Using this definition, the semantics for PRM-CH are given by the following equation:

$$P(\mathcal{I} \mid \sigma_r, \Pi) = \prod_X \prod_{x \in \sigma_r(X)} P(x.Class) \prod_{A \in \mathcal{A}(X)} P(x.A \mid \mathrm{Pa}^{x.c}(x.A)) \tag{5.1}$$

As before, the probability of an instantiation of the database is the product of CPDs of the instance attributes; the key difference is that here, in addition to the skeleton determining the parents on an attribute, the subclass to which the object belongs determines which local probability model is used.

5.2.5 Coherence of Probabilistic Model

At some level, the introduction of a class hierarchy introduces no substantial difficulties — the semantics of the model remain unchanged. Given a relational skeleton σ_r, and subclass information for each object, a PRM-CH Π_{CH} specifies a probability distribution over a set of instantiations $\mathcal{I}$ consistent with σ_r which was given by

Eq. (5.1).

As in the case of PRMs with attribute uncertainty, we must be careful to guarantee that our probability distribution is in fact coherent. In this case, while the relational skeleton specifies which objects are related to which, it does not specify the subclass indicator for each object, so the mapping of formal to actual parents depends on the probabilistic choice for the subclass for the object. In addition, for refined slot references, the existence of the edge will depend on the subclass of the object. We will indicate edge existence by the coloring of an edge: a *black* edge exists in the graph, a *gray* edge may exist in the graph and a *white* edge is invisible in the graph. As in previous chapters, we define our coherence constraints using an instance dependency graph, relative to our PRM and relational skeleton.

Definition 5.3: The *colored instance dependency graph* for a CH-PRM Π_{CH} and a relational skeleton σ_r is a graph G_{σ_r}. The graph has the following nodes, for each class X and for each $x \in \sigma_r(X)$:

- A descriptive attribute node $x.A$, for every descriptive attribute $X.A \in \mathcal{A}(X)$;
- a subclass indicator node $x.Class$.

Let $\mathrm{Pa}^*(X.A) = \bigcup_{c \in \mathcal{C}[X]} \mathrm{Pa}^c(X.A)$. The dependency graph contains four types of edges. For each attribute $X.A$ (both descriptive attributes and the subclass indicator), we add the following edges:

- **Type I edges:** For every $x \in \sigma_r(X)$ and for each formal parent $X.B \in \mathrm{Pa}^*(X.A)$, we define an edge $x.B \rightarrow x.A$. This edge is black if the parents have not been specialized (which will be the case for the subclass indicator, $x.Class$, and possibly other attributes as well). All the other edges are colored gray.
- **Type II edges:** For every $x \in \sigma_r(X)$ and for each formal parent $X.\tau.B \in \mathrm{Pa}^*(X.A)$, if $y \in x.\tau$ in σ_r, we define an edge $y.B \rightarrow x.A$. If the CPD has been specialized, or if τ contains any refined slot references this edge is colored *gray*; otherwise is is colored *black*. ∎

As before, type I edges correspond to intra-object dependencies and type II edges to inter-object dependencies. But since an object may be from any subclass, even though the relational skeleton specifies the objects it is related to, until we know the subclass of an object, we do not know which of the local probability models applies. In addition, in the case where a parent of an object is defined via a refined slot reference, we also do not know the set of related objects until we know their subclasses. Thus, we add edges for every *possible* parent and color the edges used in defining parents gray. Type I and type II edges are grayed when they are parents in a specialized CPD. In addition, type II edges may be gray if a refined slot reference is used in the definition of a parent.

At this point, the problem with our instance dependency graph is that there are some edges which are known to occur (the black edges) and some edges that may or may not exist (depending on the subclass of an object). How do we ensure our instance dependency graph is acyclic? In this case, we must ensure that the instance dependency graph is acyclic for any setting of the subclass indicators. Note that this is a probabilistic event. First, we extend our notion of acyclicity for our colored instance dependency graph.

Definition 5.4: A colored instance dependency graph is acyclic if, for any instantiation of the subclass indicators, there is an acyclic ordering of the nodes relative to the black edges in the graph. Given any a particular assignment of subclass indicators, we determine the black edges as follows:

- Given a subclass assignment $y.Class$, all of the edges involving this object are colored either black or white. Let $y.Class = d$. The edges for any parent nodes are colored black if they are defined by the CPD $\mathrm{Pa}^d(X.A)$, and white otherwise. In addition, the edges corresponding to any for any refined slot references, $\rho_{\langle d \rangle}(x, y)$, are set: If $y.Class = d$, the edge is colored black, otherwise it is painted white. ∎

Based on this definition, we can specify conditions under which Eq. (5.1) specifies a coherent probability distribution.

Theorem 5.5: *Let Π_{CH} be a PRM with class hierarchies whose colored dependency structure $\mathcal{S}$ is acyclic relative to a relational skeleton σ_r. Then Π_{CH} and σ_r define a coherent probability distribution over instantiations $\mathcal{I}$ that extend σ_r via Eq. (5.1).*

Proof: The probability of an instantiation $\mathcal{I}$ is the joint distribution over a set of random variables defined via the relational skeleton. We have the following variables:

- We have one random variable $x.A$ for each $x \in \sigma_r(X)$ and each $A \in \mathcal{A}(X)$.
- We have one random variable $x.Class$ for the class indicator variable for each $x \in \sigma_o(X)$.

Let $V_1, \ldots, V_N$ be the random variables defined above. We show that because the instance dependency graph is acyclic for any instantiation of the subclass indicators, we can construct an ordering $V_1, \ldots, V_N$ that is a topological sort of the instance dependency graph; *but* this ordering will be constructed dynamically, once we know the subclass information.

As in the proof of Theorem 2.5 our proof relies on the fact that the probability of any instantiation $\mathcal{I}$ is the product of legal conditional distributions, hence the product is a well-defined joint distribution.

The distribution is defined via Eq. (5.1). We must ensure that the conditional distribution for each random variable is well-defined and can be determined by the time that it is required in the product. Because the CPDs come directly from the PRM-CH, the first requirement is satisfied trivially. So it remains to check that the variable ordering and choice of CPDs can be determined at the point at which they are needed.

The definition of acyclicity provides us with the necessary procedure for constructing the variable ordering. Because the subclass indicator variables do not have parents, we can place all of the subclass indicators, $x.Class$, for the objects at the start of our order. Once their values have been probabilistically chosen, we are in the position to color all of our edges either black or white. And once we know all of the black edges, becuase of the acyclicity, we know there is some topological ordering of the nodes that is consistent with the black edges. This provides us with our variable ordering.

Because the instance dependency graph is acyclic and each CPD in Eq. (5.1) can be determined and is well-defined, Eq. (5.1) is a well-defined joint distribution. ∎

As in the previous case of PRMs with attribute uncertainty and PRMs with link uncertainty, we want to learn a model in one setting, and be assured that it will be acyclic for any skeleton we might encounter. Again we achieve this goal through our definition of class dependency graph. We do so by extending the class dependency graph to contain edges that correspond to the edges we defined in the instance dependency graph.

Definition 5.6: The *class dependency graph* for a PRM with class hierarchy Π_{CH} has the following set of nodes for each $X \in \mathcal{X}$:

- For each subclass $c \in \mathcal{C}[X]$ and attribute $A \in \mathcal{A}(X)$, a node $X_c.A$;
- A node for the subclass indicator $X.Class$.

and the following edges:

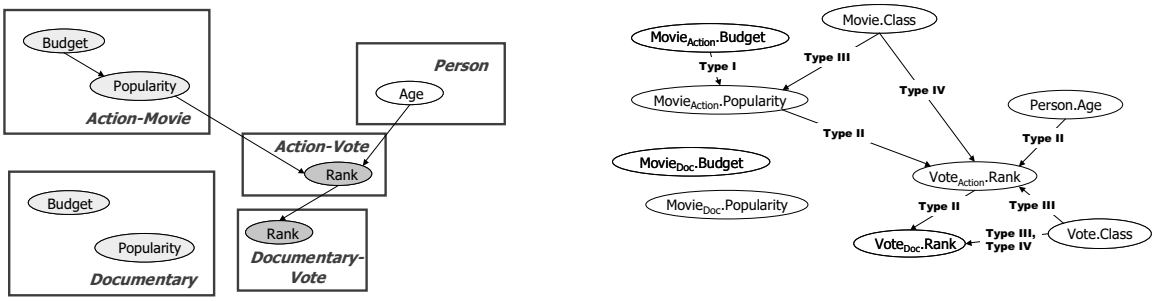
- **Type I edges:** For any node $X_c.A$ and formal parent $X_c.B \in \mathrm{Pa}^c(X_c.A)$ we have an edge $X_c.B \rightarrow X_c.A$.
- **Type II edges:** For any attribute $X_c.A$ and formal parent $X_c.\rho.B \in \mathrm{Pa}^c(X_c.A)$, where $\mathrm{Range}[\rho] = Y$, we have an edge $Y.B \rightarrow X_c.A$.
- **Type III edges:** For any attribute $X_c.A$,

  we have an edge $X.Class \rightarrow X_c.A$.

∎

Figure 5.2 shows a simple class dependency graph for our movie example. The PRM-CH is given in Figure 5.2(a) and the class dependency graph is shown in Figure 5.2(b).

It is now easy to show that if this class dependency graph is acyclic, then the instance dependency graph is acyclic.



(a) (b)

Figure 5.2: (a) A simple PRM with class hierarchies for the movie domain (b) The class dependency graph for this PRM.

Lemma 5.7: *If the class dependency graph is acyclic for a PRM with class hierarchies* Π_{CH}*, then for* any *relational skeleton* σ_r*, the colored instance dependency graph is acyclic.*

Proof: There are two parts two this proof. First, we show that our class dependency graph construction is such that for any instance dependency graph, we can determine the colors (black or white) of edges at the time in which they are needed. Next, we show that the construction is such that the black edges in the instance dependency graph will be acyclic.

The type III, type IV and type V edges are the edges that ensure that for any ordering consistent with the class dependency graph, the class is known before it is needed to determine the parents of an object in the colored instance dependency graph. In the instance dependency graph, the analogous edges are all black. We need to make sure that there is an ordering of the nodes such that at any point there is a node all of whose parents are determined (i.e., has no gray edges), and all of whose parents have been visited. Suppose at some point there is no such node. However, since all of the nodes black parent's have been visited, this contradicts our assumption, thus if the class dependency graph is acyclic, there must always be a node that we can visit next in order of the nodes in the instance graph.

The type I and type II edges in the class dependency graph enforce the acyclicity

of the black edges in the instance dependency graph. As in the proof of Theorem 2.7, if there is cycle among the black edges in the instance dependency graph, then we can construct a cycle in the class dependency graph, which contradicts our initial assumption. ∎

Note that our construction is such that if this class dependency graph is acyclic, then the colored instance dependency graph is acyclic. However there may well be acyclic colored instance dependency graphs for which the class dependency graph constructed above is not acyclic.

And again we have the following corollary:

Corollary 5.8: *Let Π_{CH} be a PRM with class hierarchies whose class dependency structure $\mathcal{S}$ is acyclic. For for any relational skeleton σ_r, Π_{CH} and σ_r define a coherent probability distribution over instantiations $\mathcal{I}$ that extend σ_r via Eq. (5.1).*

5.3 Learning PRM-CHs

As in previous chapters, we separate the learning problem into two basic questions: how to evaluate the "goodness" of a candidate structure, and how to search the space of legal candidate structures. We consider each question separately. We examine two scenarios: in one case the class hierarchies are given as part of the input and in the other, in addition to learning the PRM, we also must learn the class hierarchy. The learning algorithms use the same criteria for scoring the models, however the search space is significantly different.

5.3.1 Class Hierarchies Provided in Schema

We now turn to learning PRMs with class hierarchies. We begin with the simpler scenario, where we assume that the class hierarchy is given as part of input.

As in Chapter 3, we restrict attention to fully observable data sets. Hence, we assume that, in our training set, the class of each object is given. Without this assumption, the subclass indicator attribute would play the role of a hidden variable, greatly complicating the learning algorithm.

As discussed above, we need a scoring function that allows us to evaluate different candidate structures, and a search procedure that searches over the space of possible structures.

The scoring function remains largely unchanged. For each object x in each class X, we have the basic subclass c to which it belongs. For each attribute A of this object, the probabilistic model then specifies the subclass d of X from which c inherits the CPD of $X.A$. Then $x.A$ contributes only to the sufficient statistics for the CPD of $X_d.A$. With that recomputation of the sufficient statistics, the Bayesian score can now be computed unchanged.

Next we extend our search algorithm to make use of the subclass hierarchy. First, we extend our phased search to allow the introduction of new subclass. Then, we introduce a new set of operators. The new operators allow us to refine and abstract the CPDs of attributes in our model, using our class hierarchy to guide us.

5.3.2 Introducing New Subclasses

New subclasses can be introduced at any point in the search. We may construct all the subclasses at the start of our search, or we may consider introducing them more gradually, perhaps at each phase of the search. Regardless of when the new subclasses are introduced, the search space is greatly expanded, and care must be taken to avoid the construction of an intractable search problem. Here we describe the mechanics of the introduction of the new subclasses.

For each new subclass introduced, each attribute for the subclass is associated with a CPD. A CPD can be marked as either 'inherited' or 'specialized'. Initially, only the CPD for attributes of $X_\top$ are marked as 'specialized'; all the other CPDs are 'inherited'. Our original search operators — those that add and delete parents — can be applied to attributes at all levels of the class hierarchy. However, we only allow parents to be added and deleted from attributes whose CPDs that have been specialized. Note that any change to the parents of an attribute is propagated to any descendents of the attribute whose CPDs are marked as inherited from this attribute.

Next, we introduce operators Specialize and Inherit. If $X_c.A$ currently has an inherited CPD, we can apply Specialize($X_c.A$). This has two effects. First, it recomputes the parameters of that CPD to utilize only the sufficient statistics of the subclass c. To understand this point, assume that $X_c.A$ was being inherited from X_d prior to the specialization. The CPD of $X_d.A$ was being computed using all objects in $\mathcal{I}(X_d)$. After the change, the CPD will be computed using just the objects in $\mathcal{I}(X_c)$. The second effect of the operator is that it makes the CPD modifiable, in that we can now add new parents or delete them. The Inherit operator has the opposite effect.

In addition, when a new subclass is introduced, we construct new refined slot references that make use of the subclass. Let D be a newly introduced subclass of Y. For each reference slot ρ of some class X with range Y, we introduce a new refined slot reference $\rho_{\langle D \rangle}$. In addition, we add each reference slot of Y to D, however we refine the domain from Y to D. In other words, if we have the new reference slot ρ', where $\mathrm{Dom}[\rho'] = D$ and $\mathrm{Range}[\rho'] = X$.

5.3.3 Learning Subclass Hierarchies

We next examine the case where the subclass hierarchies are not given as part of the input. In this case, we will learn them at the same time we are learning the PRM.

As above, we wish to avoid the problem of learning from partially observable data. Hence, we need to assume that the basic subclasses are observed in the training set. At first glance, this requirement seems incompatible with our task definition: if the class hierarchy is not known, how can we observe subclasses in the training data? We resolve this problem by defining our class hierarchy based on the standard class attributes. For example, movies might be associated with an attribute specifying the genre — action, drama, or documentary. If our search algorithm decides that this attribute is a useful basis for forming subclasses, we would define subclasses based in a deterministic way on its values. Another attribute might be the reputation of the director. The algorithm might choose to refine the class hierarchy by partitioning sitcoms according to the values of this attribute. Note that, in this case, the class hierarchy depends on an attribute of a related class, not the class itself.

We implement this approach by requiring that the subclass indicator attribute be a deterministic function of its parents. These parents are the attributes used to define the subclass hierarchy. In our example, Movie.*Class* would have as parents Movie.*Genre* and Movie.*Director.Reputation*. Note that, as the function defining the subclass indicator variable is required to be deterministic, the subclass is effectively observed in the training data (due to the assumption that all other attributes are observed).

We restrict attention to decision-tree CPDs. The leaves in the decision tree represent the basic subclasses, and the attributes used for splitting the decision tree are the parents of the subclass indicator variable. We can allow binary splits that test whether an attribute has a particular value, or, if we find it necessary, we can allow a split on all possible values of an attribute.

The decision tree gives a simple algorithm for determining the subclass of an object. In order to build the decision tree during our search, we introduce a new operator Split$(X, c, X.\tau.B)$, where c is a leaf in the current decision tree for $X.Class$ and $X.\tau.B$ is the attribute on which we will split that subclass.

Note that this step expands the space of models that can be considered, but in isolation does not change the score of the model. Thus, if we continue to use a purely greedy search, we would never take these steps. There are several approaches for addressing this problem. One is to use some lookahead for evaluating the quality of such a step. Another is to use various heuristics for guiding us towards worthwhile splits. For example, if an attribute is the common parent of many other attributes within X_c, it may be a good candidate on which to split.

The other operators, Specialize and Inherit, remain the same; they simply use the subclasses defined by the decision tree.

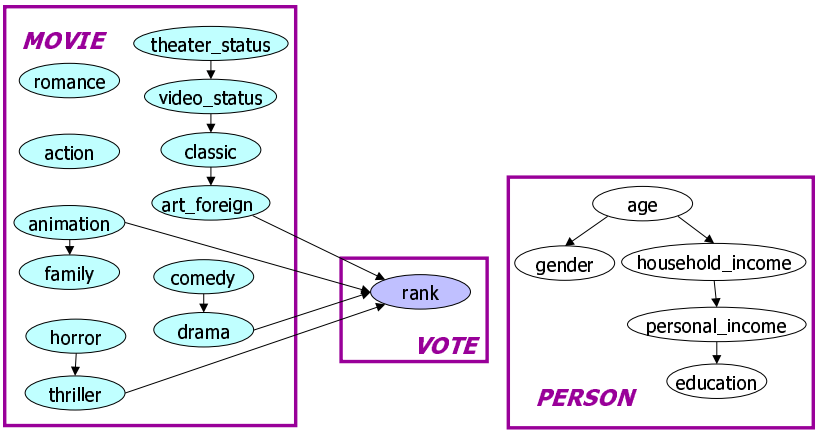


Figure 5.3: A PRM for the Movie domain.

5.4 Experimental Results

5.4.1 Model Comparison

We begin with some preliminary results testing whether the increased representational power allowed in PRM-CHs in fact improves the quality of the models we learn. We compared the utility of models with subclasses (Π_{CH}) to our standard PRMs that do not support the refinement of class definitions (Π). Here we are given the structure, we do not allow inheritance of CPDs and we simply learn the parameters. We compare the log-likelihood of a test set for each model.

We present results for the Each Movie dataset[1]. Recall that this dataset contains information information about people's ratings of movies. We extended the demographic information we had for the people by including census information available for a person's zip-code. The three classes are Movie, Person, and Votes. The training set contained 1467 movies, 5210 people and 243,333 votes.

We defined a rather simple class hierarchy for Votes, based on the genre of the Movie, Action-Votes, Romance-Votes, Comedy-Votes and Other-Votes. We learned two

[1] http://www.research.digital.com/SRC/EachMovie

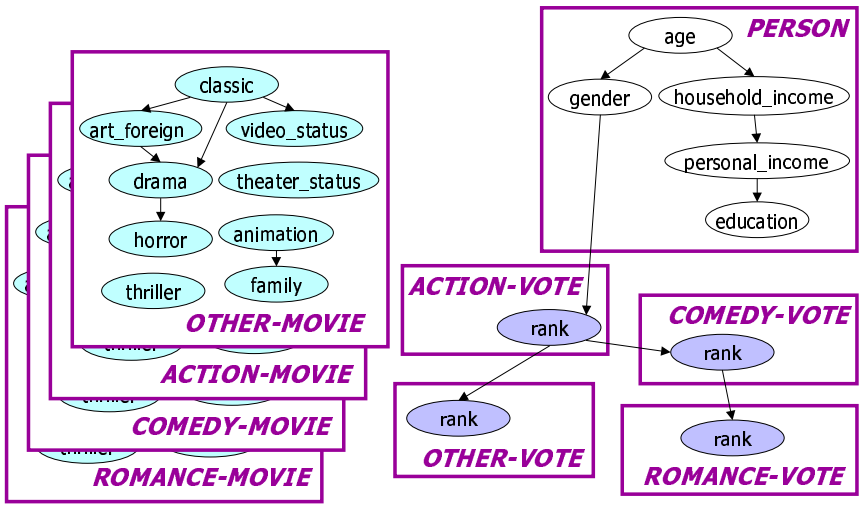


Figure 5.4: Π_{CH}. The links between vote rankings follows a slot chain, from a person's ranking on one class of movies to the person's ranking on another class of movies.

different models, one that made use of the class hierarchy (Figure 5.4) and one that did not (Figure 5.3). We then evaluated the models on five different test sets. Note that, in relational data, different test sets have markedly different structure, so trying the model on different test sets might result in very different answers. Each test set had 1000 votes, and approximately 100 movies and 115 people. The average log-likelihood of the test set for Π was -12079 with a standard deviation of 475.68. The model with class hierarchies, Π_{CH}, performed much better, with average log-likelihood of -10558 and a standard deviation of 433.10. Using a standard t-test, we obtain that Π_{CH} is better than Π with well over 99% confidence interval.

Looking more closely at the qualitative difference in structure between the two models, we see that the PRM-CH is a much richer model. For example the dependency model for $\mathsf{Vote}.Rank$ cannot be represented without making use of the class hierarchy to both refine the attributes and refine the allowable slot chains. For example, we learn a dependence of $\mathsf{Vote}_{Romance}.Rank$ on $\mathsf{Vote}_{Romance}.Person.Comedy\text{-}Votes.Rank$, whereas $\mathsf{Vote}_{Action}.Rank$ depends on $\mathsf{Vote}_{Action}.Person.Gender$. Not only are these two dependency models different, but they would be cyclic if interpreted as a standard

PRM. Note that in the PRM shown in Figure 5.3, there is no dependency between Vote.*Rank* and attributes of Person, so the PRM that uses class hierarchies allows us to discover dependencies on properties of Person that we could not uncover before.

5.5 Conclusion

In the chapter, we have proposed a method for making use of class hierarchies while learning PRMs. Class hierarchies give us additional leverage for refining our probabilistic models. They allow us to automatically disentangle our dependency model, allowing us to construct acyclic dependencies between elements within the same class. They also allow us to span the spectrum between class level and instance level dependency models.

However, using class hierarchies significantly expands an already complex search algorithm. The search space for PRMs-CH is much larger. In this chapter, we describe a general search algorithm. However, a key to the success of the algorithm is the discovery of useful heuristics to guide the search. In future work, we intend to explore the space of possible heuristics, and to test empirically which heuristics work well on real-world problems.