MOVIE RECOMMENDATION
EXPERT SYSTEM

I know a lot about movies...

Ashish Gagneja (ag2818)
Ben Warfield (bbw2108)
Mehmet Can Işık (mci2109)
Mengü Sükan (ms3774)
Snehit Prabhu (sap2131)
Vivek Kale (vk2217)

Monday, December 8, 2008
MOVIE RECOMMENDATION
EXPERT SYSTEM

Introduction

Our movie recommendation expert system is composed of an inference engine, a set of rules, and a working memory of movie-, actor-, and director-facts. Our engine was designed and developed to work on any rule-set and knowledge domain, but in this application it has been tailored to make movie recommendations.

We gathered data for our knowledge base from the Internet Movie Database (IMDB). Our system has information such as cast, director, release year, and genre on the top 250 movies of all time (as voted by IMDB users).

Usage of our Expert System

Using our movie recommendation system is very simple: Just provide the system with a list of movies that you like by specifying the title of the movie and the year that it was made in (there may be two movies with the same name, but made in different years).

You can use either one of the functions "show-movies-by-name" or "show-movies-by-year" to extract all the movie names in the knowledge base in a human-readable format.

CL-USER> (show-movies-by-name)

Once you browse through the list of movies that our expert system knows about, you can copy & paste the movies you like as inputs into the recommend-movies function.

Note: While this is far from an ideal user interface, we provided these functions, so that you don't have to deal with the extraneous escape characters that you see in the actual knowledge base. If you input incorrectly spelled movie names, our engine will not find a match for it in its knowledge base and ignore it. We did not implement spell-checking for user input, since we wanted to focus on building a robust inference engine rather than worrying about the User-System interface.
As we mentioned before, the "show-movies-by-name" or "show-movies-by-year" functions will print a list of movie names onto the screen. To actually get movie recommendations, you can specify your input in the following manner:

```
CL-USER> (recommend-movies '(
  "movieName1 (<year>)"
  "movieName2 (<year>)"
  "movieName3 (<year>)"
  "movieNameN (<year>)"
))
```

After calling the “recommend-movies” function with a list of your favorite movies, you will get a status message indicating that the system is “thinking” (i.e. making inferences to provide recommendations). The output will then show the top 10 movies that the system recommends based on its interpretation of your taste. It is possible to show the top “k” movies as well, and this can be configured in our movie-expert-source.lisp code. Specifically, the output will be of the form:

```
Your top ten recommended movies:
  recommendedMovieName1 (year1) score1
  recommendedMovieName2 (year2) score2
  recommendedMovieName3 (year3) score3
  
  recommendedMovieName10 (year10) score10
```

Here is an example of an actual interaction with our expert system:

```
(recommend-movies
 '(
  "Citizen\ Kane\ (1941)"
  "Day\ the\ Earth\ Stood\ Still,\ The\ (1951)"
  "Dial\ M\ for\ Murder\ (1954)"
  "Great\ Dictator,\ The\ (1940)"
  "Judgment\ at\ Nuremberg\ (1961)"
))

"Hmm, so you say you have enjoyed watching those 5 movies. Let me think...
"OK, I'm done! Based on your input, I recommend:

  Manchurian Candidate, The (1962)  148.3
  Rear Window (1954)  138.7
  Vertigo (1958)  128.5
  Witness for the Prosecution (1957)  128.1
  Man Who Shot Liberty Valance, The (1962)  128.0
  Great Escape, The (1963)  118.3
  Some Like It Hot (1959)  118.3
  Yojimbo (1961)  118.2
  Rosemary's Baby (1968)  118.0
  Laura (1944)  117.9
```

"
You will note that there is a column of scores to the right of the movie recommendations. These are the weights that were calculated to extract the top 10 recommendations. Last but not least, the movies that the user has given as input will never appear in the list of recommendations, as those movies are eliminated from the result set in our inference step.

**Workflow**

The diagram below show the different components of our expert system, and how they work together to produce recommendations for the user.

At a high-level, the process above can be described as follows:
1. User query (i.e. a movie list) and knowledge base are instantiated into objects and added to Working Memory
2. The engine starts applying inference rules to those objects to assign them scores based on how well their attributes correlate with user preferences
3. The engine stops when no more rules can be fired on objects in the working memory
4. Top ten recommendations (based on overall score) are returned to the user
Rules

Below is the list of rules that are passed to our movie recommendation engine to process (in order of priority)

1. Create (user-likes-actor ...) facts. That is, if there is movie user likes, create user-likes-actor facts for each actor in that movie. This rule is exhaustible, so it only runs once.

2. Calculate weights for (user-likes-actor ...) facts. That is, if the user likes multiple movies with the same actor, we increment the weight of that user-likes-actor fact (increments are accomplished by removing the old user-likes-actor fact and inserting a new one with the incremented weight).

3. Create (user-likes-director ...) facts. That is, if there is movie user likes, create user-likes-director facts for the director of that movie. This rule is exhaustible, so it only runs once (for a more thorough explanation of rule syntax please, refer to the appendix).

4. Calculate weights for (user-likes-director ...) facts. That is, if the user likes multiple movies by the same director, we increment the weight of that user-likes-director fact.

5. Actors that appear in fewer than 3 movies are removed as irrelevant.

6. Create user-likes-era facts for any eras in the memory which have corresponding movies in the user-likes-movie list.

7. Calculate weights of (user-likes-era ...) facts.

8. Create (user-likes-genre-name ...) facts for any genres in the memory which have corresponding movies in the user-likes-movie list.

9. Calculate weights for (user-likes-genre-name ...) facts.

10. Create (recommend-movie ...) facts for each movie fact that exists in the knowledge base.

11. Remove (recommend-movie ...) facts if movie is in user input.

12. Calculate weights for (recommend-movie ...) facts based on (user-likes-* ...) facts and their calculated weights.

13. Sort (recommend-movie ...) facts by descending weight.

Inference Engine

To actually compute the recommendations, we use an interpreter that we have written in LISP programming language. In other words, our movie inference engine is built “on top of our interpreter”. While we designed the interpreter with the movie domain in mind, we also have made sure that it can be used for other domains as well.

A high level view of our algorithm is as follows:

1. Initialize working memory by loading facts from a knowledge base and user input
2. Initialize rule objects for the specific expert system
3. Pass the working memory and the list of rule objects into the inference engine
4. Use the match function to fire rules until the terminate action is encountered or there are no more rules that match any of the facts in the working memory
   1. For each fired rule, use the substitute function on the right-hand side of the rule to fill-in action patterns using bindings that were matched by the left-hand side of the rule
   2. Modify the working memory by executing bound actions-patterns

Optimizations

We were able to speed up our expert system significantly by modifying our rule data structure to keep track of whether a given rule has been exhausted and cannot be applied any more.

Additionally, we designed the structure of our WM to break the knowledge base into separate bins for each type of fact. With this feature, patterns are only searched over the bin which contains the same type of fact as the pattern itself, making the search-space is drastically narrowed over a naive implementation that might search the entire WM for each pattern.
RPS Interpreter

Our goal in designing our RPS interpreter was to make it simple to use, but also very flexible and powerful. Our interpreter defines three different types of constructs that are used in our movie expert system: facts, actions, and rules. Facts are represented as objects in our working memory, initialized from our movie KB and user query. We define each one of the constructs in detail below.

1. Facts

Facts are constructed as LISP lists.

Syntax:

(fact-name (property1-name property1-value)
  (property2-name property2-value)
  ...
  (propertyN-name propertyN-value))

Each fact-name and property-name is a LISP symbol.
Each property-value can be symbols, numbers or strings in LISP.

Note that property-names may not be necessary. Only the fact-name must be specified.

For example, both of the following are valid facts, but they won't match each other.

(movie "Dark Night, The" (thriller 1))
(movie (name "Dark Night, The") (thriller 1))

Also, the order of the properties is important. For example,

(movie "kill-bill" (action 1)(comedy 0))
(movie "kill-bill" (comedy 0)(action 1))

are valid facts, but they won't match each other. Note that properties can be unnamed as well, since we assume that the property name can be specified based on the order in the argument list.
2. **Rules**

Syntax:

```
( <LHS>  <RHS>  <match-length(number)>  <exhaustible{T/NIL}> )
```

Our system contains a LISP class called rule. While the class itself is quite complicated (in order to keep track of various run-time parameters), the way rules are specified in our code is relatively simple. Let us examine one example.

Example rule specification:

```
{ }
{ }
{ (user-likes-western =w)
  (movie =n * *
   * * * *
   * * * *
   * * * *
   * * * (Western 1))
  (recommend-movie =n =r) }
;;;;  LHS
{ (REMOVE 3)
  (ADD (recommend-movie =n (+ (* 10 =w) =r)))
}
;;;;  RHS
2 ;;  Match-length
NIL ;;  Exhaustible
```

The rule above can be described in plain English as follows: if the user likes western movies, and if there is a western movie in the WM, and if there is already a recommend-movie fact present in the WM for that movie, for each such combination remove the fact that matched the 3rd pattern (i.e. recommend-movie =n =r) from the WM and add a new recommend-movie fact for that movie with an adjusted weight (i.e. 10 x weight-of-western movies + previous recommendation weight of that movie). Once this rule fires for a combination of user-likes-western and (movie ...) facts, it will be closed for that exact pair of facts (as indicated by match-length = 2). Note the usage of the wild card * to allow movies to have any other property as long as Western = 1.

Patterns correspond to the first two fields of the list above. The <LHS> (left-hand side of rule) is a list of object patterns, and the <RHS> (the right hand side of the rule) is a list of action patterns.

For more information on our rule/pattern syntax, please refer to the appendix.
3. **ACTIONS**

Our engine allows three actions: add, remove, and terminate.

3.1. **ADD-ACTION**

Usage:

```
(add {fact or fact-pattern})
```

The add function inserts the given fact to WM (working memory). Note that the parameter can also be a pattern instead of a flat fact. If it is a pattern, the engine makes appropriate substitutions using the bindings passed from the left-hand side of the rule.

Example:

```
(add (movie "Superman (1985)"
     1985 9.1
     (action 1)
     (comedy 0)
     (drama 0)))
```

This adds the movie superman with given properties to the WM.

3.2. **REMOVE-ACTION**

Usage:

```
(remove {n})
```

This removes the fact, which matched the n\textsuperscript{th} pattern from the LHS, from the WM.

Example:

```
(remove 2) ; removes the second object that has matched
            ; the LHS of that rule from WM.
```

3.3. **TERMINATE-ACTION**

Usage:

```
(terminate)
```

If this action is fired, it stops the termination, and the current WM is returned from the engine.
Data

The data for our expert system came in the form of flat text files that were downloaded from the Internet Movie Database (http://www.imdb.com). Before we could start developing any LISP code, our resident Perl (and LISP) expert, Ben, wrote a Perl script to construct valid facts (i.e. LISP lists) by parsing/processing those text files from IMDB.

Please note that our system scales to relatively large amounts of data.

<table>
<thead>
<tr>
<th>Total number of facts in our KB</th>
<th>3,800+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types of facts in our KB</td>
<td>20+</td>
</tr>
<tr>
<td>Number of movies:</td>
<td>250</td>
</tr>
<tr>
<td>Number of actors:</td>
<td>3,100+</td>
</tr>
<tr>
<td>Number of directors:</td>
<td>77</td>
</tr>
</tbody>
</table>

Note: A single movie can be tagged with multiple genres.
APPENDIX

Implementation Details
Rule

The RULE class defines a set of conditions to search for (specifically, a set of patterns that must match against actual facts in the working memory), and a set of actions that should be taken when those conditions are met. It also implements a number of bookkeeping features, some of which are automatic and some of which must be activated at object instantiation time.

**INITIALIZATION ARGUMENTS**

<table>
<thead>
<tr>
<th>Init key</th>
<th>required?</th>
<th>content</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>:pattern-list</td>
<td>Yes</td>
<td>A list of patterns, as described below</td>
<td>(length pattern-list)</td>
</tr>
<tr>
<td>:action-list</td>
<td>Yes</td>
<td>A list of actions, as described in the top-level README</td>
<td></td>
</tr>
<tr>
<td>:match-length</td>
<td></td>
<td>An integer between 1 and the length of the pattern list</td>
<td>NIL</td>
</tr>
<tr>
<td>:close-on-bindings</td>
<td></td>
<td>A list of bind variable names (e.g. =MOVIE)</td>
<td>NIL</td>
</tr>
<tr>
<td>:exhaustible</td>
<td></td>
<td>A T/NIL value</td>
<td>NIL</td>
</tr>
<tr>
<td>:match-once</td>
<td></td>
<td>A T/NIL value</td>
<td>NIL</td>
</tr>
<tr>
<td>:pre-bindings</td>
<td></td>
<td>A binding list (suitable for assoc)</td>
<td>NIL</td>
</tr>
</tbody>
</table>

So for example:

```
(defvar rule1 (make-instance 'RULE
   :pattern-list '((movie =mname 2006 * * * *) (user-input =mname))
   :action-list '((ADD (user-likes-new-movies)))
   :match-length 1
   :exhaustible T
   ))
```

In more detail:

**pattern-list (required)**

This a list of "patterns" defined using the syntax described below. When the working memory is searched for facts matching this rule, matches are sought for each pattern in this list.
in order, using a depth-first search: when a match for the first pattern is found, the match engine will begin to search for matches to the second pattern (if any) using the binding list produced by the first match.

**action-list (required)**

A list of actions to be taken when the rule is successfully matched. Valid actions are as described in the top-level documentation.

**match-length**

:match-length 3 ; defaults to the length of the pattern-list

All rules have a closed list of sets of facts on which they have already matched. For some rules, not all of the facts matched may be significant (this is frequently the case in rules designed to increment counters, since there is no UPDATE action), and a simple closed-list allows the rule to match on that set of facts only once.

The :match-length parameter may be set during object instantiation. If it is set to some value less than the length of the pattern-list, only that number of facts will be considered when checking the closed list.

**close-on-bindings**

:close-on-bindings ' (=ACTORNAME =GENRE)

Yet another way in which a rule can be restricted from matching too many times: this argument tells the rule-processing engine that this rule should only match once per valid combination of values for the bind variables supplied. So in the example above, we might have a rule that matched on actors, movies and genres, but should only be allowed to fire once per distinct (actor, genre) pair. The variables listed in this argument must be bound by the pattern-list for this rule.

**exhaustible**

Certain rules (many, in the case of the movie recommendation system) act only on facts that are created by higher-priority rules. This implies that if the rule ever begins to fire, all of the facts that it may ever match are already in the working memory. This allows for several important optimizations. Principally, however, it means that if a match on this rule is ever attempted and then fails, it means that all subsequent attempts to match the rule will fail: the rule is considered "exhausted".
Setting this flag at instantiation time represents a promise from the rule-writer to the engine that valid matches for this rule will never be added to the working memory by lower-priority rules: if it is set, match performance is dramatically improved in some cases, but facts added after the rule first fires may not be successfully matched.

**match-once**

Certain rules may match multiple times but should fire at most one time. If the :match-once parameter is set to a non-NIL value at instantiation, the rule will be marked as un-matchable after the first time it matches.

**pre-bindings**

`:pre-bindings '(=MIN-COUNT . 3)

Since our match syntax does not allow for comparisons against fixed values, but only against bound variables, this facility is supplied to allow rule writers to pre-bind certain values. The example above could be used in a pattern like `(user-likes-actor =actorname >min-count).

**INTERFACE**

**Accessors**

- pattern-list
- closed-list
- action-list
- match-length
- exhausted
- close-on-bindings
- match-once
**Methods**

- (exhaust rule)
  Marks a rule as exhausted, if it is an exhaustible rule. In any case, returns NIL.

- (add-to-closed rule result)
  Add the fact list in this result (or a prefix of that list, if match-length is set to something less than the maximum) to the closed list for this rule.

- (closedp rule fact-list)
  Test if this list of facts is in the closed list (shortcuts quickly for lists of the wrong length).

**Matching Syntax**

The syntax for patterns is derived from the syntax defined in assignment #1 of this course. There are three important modifications:

1. All patterns must be, at the top level, proper lists, with a symbol or string as their first element:
   1. valid: '(movie =name ...)
   2. valid: '("moviescores" (1 2 3))
   3. INVALID: 'some-fact
   4. INVALID: '((nested list) (other nested list))

2. The character * may be used within a pattern as a wild-card to match any fact (subject to the restriction above that patterns must be proper lists at the top level)

3. As discussed on the first mid-term, sub-patterns with the structure (\| p1 p2 p3) may be used in the same manner as (& p1 p2 p3): they match if any, rather than all, of the subpatterns successfully match.
Expert-WM

The working memory is an opaque collection of facts in the form `'(SYMBOL [arbitrary lisp data structure]* )`. The "type" of a given fact is the first symbol: the movie recommender, for example, begins with facts of type "movie", "actor", "director" and "era"; its final product is facts of type "recommend-movie".

METHODS

- `(add-fact wm fact)`
  Add a new fact, prepending it to the existing list of facts of the same type. This is effectively a constant-time operation (linear in the number of fact types currently in the memory).

- `(delete-fact wm fact)`
  Deletes a fact, if it is present in the working memory. This takes time linear in the number of facts of the same type as the one being deleted.

- `(candidate-list wm fact-type)`
  Returns a list of facts that is guaranteed to include facts of the given type (e.g. all movies or all recommendations).