

Personalisation for Smart Personal Assistants

Joshua Cole

Computer Sciences Laboratory
The Australian National University
Email: Joshua.Cole@anu.edu.au

Matt Gray

Computer Sciences Laboratory
The Australian National University
Email: Matt.Gray@anu.edu.au

John Lloyd

Computer Sciences Laboratory
The Australian National University
Email: jwl@discus.anu.edu.au

Kee Siong Ng

Computer Sciences Laboratory
The Australian National University
Email: kee.siong@discus.anu.edu.au

Abstract—This paper is concerned with personalisation of smart personal assistants by machine learning techniques. The application of these ideas to an infotainment demonstrator is discussed in detail. The symbolic, on-line machine learning techniques used are described. Also experimental results, which demonstrate that a high level of personalisation can be achieved by this approach, are presented.

I. INTRODUCTION

This paper describes some recent results from the “Machine Learning for the Smart Personal Assistant” project in the Smart Internet Technology Cooperative Research Centre. The aim of the project is to apply machine learning techniques to building adaptive agents, especially user agents that facilitate interaction between a user and the Internet. This paper concentrates on the particular form of adaptation known as personalisation in which the agent adapts its behaviour according to the interests and preferences of the user. There are many practical applications of personalisation that could exploit the technology presented here.

The research is set in the context of an infotainment demonstrator that is being built in the project. This demonstrator, which is a multi-agent system, contains a number of agents with functionalities for recommending movies, TV programs, music and the like, as well as information agents with functionalities for searching for information on the Internet. This paper concentrates on the TV recommender as a typical such agent and shows how a high degree of personalisation can be achieved by symbolic machine learning techniques. The techniques can be ported comparatively easily to other agents of the demonstrator, thus providing a fully personalised infotainment system.

Section II gives an outline of the infotainment demonstrator, concentrating on its TV recommender. Section III briefly describes the approach to adaptation taken here. Section IV describes how the TV recommender is personalised to the user. Section V presents results of experiments that show the level of personalisation achieved. Section VI contains some conclusions and future directions for research.

II. AN INFOTAINMENT DEMONSTRATOR

In this section, we describe the infotainment demonstrator being developed, concentrating on the TV recommender.

The infotainment demonstrator is a multi-agent system that combines a number of related functionalities concerning information search and entertainment. The agents that comprise the demonstrator and that are at least partly implemented include a TV recommender, a movie recommender, a news agent, a search agent, and a diary agent. In addition, there is a coordinator agent that has the responsibility of handling interactions between the user and the various agents in the demonstrator. We also plan to include a music recommender because the technology that would be needed to implement it is the same as the other agents and, given the extraordinary current interest in the IPOD and similar devices, we believe it has excellent potential for commercialisation. The TV recommender also has potential for commercialisation as a personalisation component in a system such as TiVO, for example, which supports sophisticated search functions to find TV programs of interest to a user but never has any actual knowledge of the interests or preferences of the user.

A detailed description of the architecture of the TV recommender is now given. The architecture of the other agents in the infotainment demonstrator is similar. What functionality do we want the TV recommender to have? When the user first begins to use the TV recommender it clearly has no knowledge of the interests or preferences of the user. The aim is to design an adaptive architecture for the TV recommender so that within a comparatively short time, perhaps several weeks, it is able to make helpful recommendations to the user. Furthermore, it should improve its performance over longer periods and accurately track changing user interests and preferences. To get started, the agent presents a short questionnaire to the user the first time it is used. The purpose of the questionnaire is to acquire, with as little effort as possible on the part of the user, some initial idea of the user’s interests and preferences. After that, the agent collects training examples, partly by observing the user’s activities and partly by direct questions, if necessary. Over time, the agent is expected to be able to make recommendations for programs

in specified time periods (say, ‘next week’ or ‘tonight’) that the user finds helpful.

A detailed description of the most pertinent aspects of the design of the TV recommender is now given. The approach to knowledge representation taken here is covered in detail in [1]. We will need several standard types: Ω (the type of the booleans), Nat (the type of natural numbers), Int (the type of integers), and $String$ (the type of strings). The intended meaning of the constant \top of type Ω is true and that of \perp is false. Also $List$ denotes the (unary) list type constructor. Thus, if α is a type, then $List \alpha$ is the type of lists whose elements have type α .

Three domain-specific types, $Channel$, $Genre$, and $Classification$, will also be needed. Here are the data constructors for these types.

ABC, Prime, WIN, Ten, SBS, Animal_Planet, Arena, BBC_World, Cartoon_Network, Channel_V, CNBC, CNN, Comedy_Channel, Discovery_Channel, Disney_Channel, ESPN, Fox8, Fox_Classics, Fox_Footy, Fox_News, Fox_Sports_1, Fox_Sports_2, Hallmark_Channel, History_Channel, Lifestyle_Channel, Max, Movie_Extra, Movie_Greats, Movie_One, MTV, National_Geographic, Nickelodeon, Ovation, Showtime, Showtime_Greats, Sky_News, Sky_Racing, TCM, TV1, TVSN, UK_TV, W, World_Movies : Channel

Action, Adventure, Animation, Arts_and_Culture, Biography, Business_and_Finance, Cartoon, Children, Comedy, Crime, Current_Affairs, Cult, Documentary, Drama, Education, Entertainment, Family, Fantasy, Film_Noir, Game_Show, Historical, Horror, Infotainment, Lifestyle, Murder_Mystery, Music, Musical, Mystery, Nature, News, Parliament, Real_Life, Religion, Romance, Romantic_Comedy, Science_and_Technology, Sci_Fi, Shopping, Short, Sitcom, Soap_Opera, Sport, Talk_Show, Thriller, Travel, Variety, War, Weather, Western, NA : Genre

C, G, M, MA, P, PG, R, TB, NA : Classification.

We introduce the following type synonyms.

Date = Day \times Month \times Year
Time = Hour \times Minute
Title = String
Duration = Minute

Synopsis = String

Program = Title \times Duration \times Genre \times

Classification \times Synopsis

Year = Nat

Month = Nat

Day = Nat

Hour = Nat

Minute = Nat

Text = List String.

The agent has access via the Internet to a TV guide (for the next week or so) for all Australian channels. This database is represented by a function *tv_guide* having signature

tv_guide : Date \times Time \times Channel \rightarrow Program.

Here the date, time and channel information uniquely identifies the program and the value of the function is (information about) the program itself. The TV guide consists of (thousands of) facts like the following one.

$((tv_guide ((20, 7, 2004), (20, 30), ABC)) =$
 (“*The Bill*”, 50, *Drama, M,*
 “*Sun Hill continues to work at breaking the*
people smuggling operation”).

This fact states that the program on 20 July 2004 at 8.30pm on channel ABC has title “The Bill”, a duration of 50 minutes, genre drama, a classification for mature audiences, and synopsis “Sun Hill continues to work at breaking the people smuggling operation”.

III. ADAPTATION

The adaptation approach of this paper uses on-line learning, which works as follows. Suppose we want to learn a classification function f . For this, several ingredients are needed. First, we need an initial definition of f to get started. Second, we need an hypothesis language which is the space of all possible definitions that f could have. Finding a suitable hypothesis language is one of the key design decisions that has to be made. Finally, we need a sequence of training examples that the learning algorithm can use to move from one definition of f to another. (See Figure 1.) The intuitive idea is that the learning algorithm chooses the definition that most closely agrees with the current set of training examples. For many agent applications, certainly the ones considered in this paper, it is important that the hypotheses be comprehensible to the user of the agent. This is achieved here by the use of logic as the language in which the hypotheses are expressed. Comprehensibility ensures that the initial function can be written directly and that the definition can be inspected at any later stage to help understand (and perhaps modify) the behaviour of the agent.

A training example is simply a pair consisting of a particular individual and the class for that individual. Hypothesis languages are expressed by predicate rewrite systems as discussed

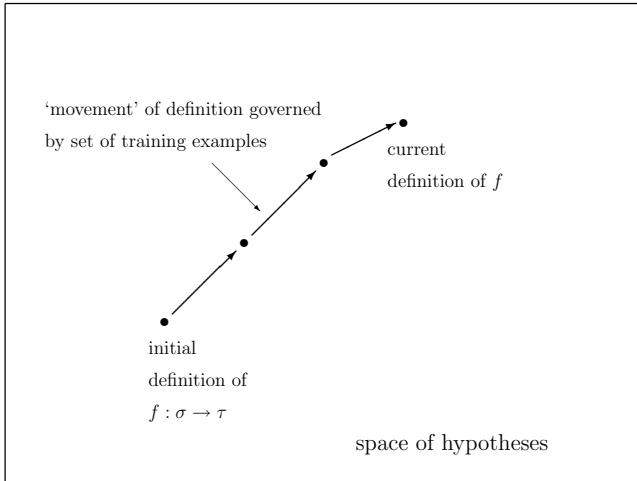


Fig. 1. Adaptation in the space of hypotheses

in [1]. Essentially, a predicate rewrite system is a grammar for constructing predicates from more basic ingredients. A predicate rewrite system for the TV recommender is given below.

The learning system we use is called ALKEMY [3] and is a decision-list learner [4] that works as follows. Starting from a set of training examples, the learner looks for a predicate such that the subset of examples whose individuals satisfy that predicate all have the same class (that is, the subset of examples is pure). The learner constructs a left child containing those examples, assigning the common class to that node. It then continues at the right child with the new set of examples obtained by removing the examples in the left child from the original set. The learner terminates when it reaches a right child with a pure set of examples. (It will also terminate if it is unable to find a predicate that splits off a pure left node.) Leaf nodes in the tree (except the last node, that is, the bottom right-hand leaf node) are labelled by the class of the examples at that node. The last node is not labelled; individuals that reach this node are not classified because we do not have enough confidence in the prediction. This means that the coverage of the learner is not 100%. If several predicates all produce a pure left node, the learner selects a predicate that produces the largest number of examples in that node.

In its on-line version (in contrast to the batch version), a decision-list learner receives a sequence of training examples over time and at any time may be asked to predict the class of some unseen individual based on its current decision list. Thus it is preferable that the learner be able to build decision lists incrementally, that is, it should be able to slightly modify the current decision list based on the next training example rather than have to rebuild the decision list from scratch each time. In its deployment in the infotainment demonstrator, it is

```

function Learn( $\mathcal{E}, \succrightarrow$ ) returns a decision list;
inputs:  $\mathcal{E}$ , a set of examples;
          $\succrightarrow$ , a predicate rewrite system;

tree := single node with examples  $\mathcal{E}$ ;
S := the set of predicates defined by  $\succrightarrow$ ;
move to root node of tree;
while set of examples  $\mathcal{F}$  at node is not pure do
  foreach  $p \in S$  do
     $\mathcal{F}_+ := \{(t, v) \in \mathcal{F} \mid (p t)\}$ ;
     $\mathcal{F}_- := \mathcal{F} \setminus \mathcal{F}_+$ ;
    if  $\mathcal{F}_+$  is pure then
      create left child with examples  $\mathcal{F}_+$ ;
      create right child with examples  $\mathcal{F}_-$ ;

      move to right child;
    break;
  if no split was found then break;
label each leaf node of tree (except the last) by the class
of its examples;
return tree;

```

Fig. 2. Decision-list learning algorithm

the on-line version of ALKEMY that we use [3]. Furthermore, since a user's interests and preferences may change over time, it is likely that there are times when the current set of training examples is inconsistent, in the sense that the same individual may have two or more distinct classes in the training examples. This means that considerable care needs to be taken in deciding what should be the current set of training examples. For example, it is common to insist on some maximum size for the training set and to drop the oldest training examples as new ones are received to keep to this limit. We prefer the approach of returning the training set to consistency, even to the point of asking the user to resolve conflicts, if necessary. In this approach a training example could stay in the training set for a very long time and would only drop out if it contradicted another training example that was somehow confidently known to be correct. The current implementation uses a simple algorithm to check for consistency. More sophisticated schemes are being investigated.

As for all learning tasks, the main problem we face here is to decide which predicates should appear in decision lists, that is, what should be the hypothesis language. This problem is discussed for the TV recommender in the next section.

IV. PERSONALISATION OF THE TV RECOMMENDER

Now we turn to the personalisation aspects of the TV recommender. The key function that needs to be learned is the function *user_likes_tv_program* which takes a TV program as input and returns true if the agent considers the program to be

worth recommending to the user; otherwise, it returns false. Thus the belief base of the TV agent contains the function *user_likes_tv_program* that has signature

$$user_likes_tv_program : Program \rightarrow \Omega$$

and a definition that is a decision list of the form

$$\begin{aligned} (user_likes_tv_program\ x) = & \\ & \text{if } (p_1\ x) \text{ then } \top \\ & \text{else if } (p_2\ x) \text{ then } \perp \\ & \quad \vdots \\ & \text{else if } (p_n\ x) \text{ then } \top \\ & \text{else } \perp, \end{aligned}$$

where p_1, \dots, p_n are predicates on programs.

We now discuss the hypothesis language used by ALKEMY that contains these predicates p_1, \dots, p_n and is used to learn the definition of *user_likes_tv_program*. The approach to constructing hypothesis languages is by means of predicate rewrite systems. The basic idea is to construct predicates by composing more basic ingredients. Thus we need the composition function

$$\circ : (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$$

defined by

$$((f \circ g)\ x) = (g\ (f\ x)).$$

The basic ingredients that are composed are made out of transformations [1]. The collection of transformations used in the application is now presented.

We begin with two transformations whose definitions come from user interests and preferences, and so provide a way of personalising the hypothesis language used to learn the definition of *user_likes_tv_program*. One of these is the function *genre*. To define this, we introduce the type synonym

$$Preference = Int,$$

where it is understood that only numbers in $\{-2, -1, 0, 1, 2\}$ are to be used as constants of type *Preference*. Then

$$genre : Genre \rightarrow Preference$$

is the function that maps each genre into an integer in the range -2 to 2 , depending on how strong a preference the user has for that particular genre. A typical definition of *genre* is given in Figure 3. This definition states that the user has a modest liking for adventure programs, a strong dislike of animation programs, a modest liking for arts and culture programs, a strong dislike of cartoons, a strong dislike of children's programs, a strong dislike of cult programs, and so on. The information in this definition is obtained by an initial questionnaire completed by the user and by belief update, if the user later changes his/her preferences. Our experiments showed that the learner was able to make good use of the information given by the function *genre*.

$$\begin{aligned} (genre\ x) = & \\ & \text{if } ((= Adventure)\ x) \text{ then } 1 \\ & \text{else if } ((= Animation)\ x) \text{ then } -2 \\ & \text{else if } ((= Arts_and_Culture)\ x) \text{ then } 1 \\ & \text{else if } ((= Cartoon)\ x) \text{ then } -2 \\ & \text{else if } ((= Children)\ x) \text{ then } -2 \\ & \text{else if } ((= Cult)\ x) \text{ then } -2 \\ & \text{else if } ((= Documentary)\ x) \text{ then } 1 \\ & \text{else if } ((= Education)\ x) \text{ then } 1 \\ & \text{else if } ((= Film_Noir)\ x) \text{ then } 1 \\ & \text{else if } ((= Game_Show)\ x) \text{ then } -2 \\ & \text{else if } ((= Historical)\ x) \text{ then } 2 \\ & \text{else if } ((= Horror)\ x) \text{ then } -2 \\ & \text{else if } ((= Infotainment)\ x) \text{ then } 1 \\ & \text{else if } ((= Nature)\ x) \text{ then } 2 \\ & \text{else if } ((= News)\ x) \text{ then } 2 \\ & \text{else if } ((= Parliament)\ x) \text{ then } -1 \\ & \text{else if } ((= Religion)\ x) \text{ then } -2 \\ & \text{else if } ((= Romance)\ x) \text{ then } -2 \\ & \text{else if } ((= Science_and_Technology)\ x) \text{ then } 2 \\ & \text{else if } ((= Sci_Fi)\ x) \text{ then } 2 \\ & \text{else if } ((= Sitcom)\ x) \text{ then } -2 \\ & \text{else if } ((= Soap_Opera)\ x) \text{ then } -2 \\ & \text{else if } ((= Sport)\ x) \text{ then } 2 \\ & \text{else if } ((= Romantic_Comedy)\ x) \text{ then } -1 \\ & \text{else if } ((= Shopping)\ x) \text{ then } -2 \\ & \text{else if } ((= Talk_Show)\ x) \text{ then } -1 \\ & \text{else if } ((= Thriller)\ x) \text{ then } 1 \\ & \text{else if } ((= Variety)\ x) \text{ then } -2 \\ & \text{else } 0. \end{aligned}$$

Fig. 3. A typical definition for *genre*

Another transformation obtained from the user is

$$classification : Classification \rightarrow Preference$$

that gives information about the user's liking for programs having a certain classification. A typical definition of *classification* could be as follows.

$$\begin{aligned} (classification\ x) = & \\ & \text{if } ((= C)\ x) \text{ then } -2 \\ & \text{else if } ((= P)\ x) \text{ then } -2 \\ & \text{else } 0. \end{aligned}$$

The information in this definition is also obtained by an initial questionnaire.

The remaining transformations that follow are generic ones which essentially come from the types employed in the application [1].

For each type α , there is a transformation $top : \alpha \rightarrow \Omega$ defined by $top\ x = \top$. The predicate top is the weakest predicate on individuals (of type α).

For each component of the type *Program*, there is an associated projection function. For example,

$$projTitle : Program \rightarrow Title$$

is defined by

$$(projTitle\ (t, d, g, c, s)) = t.$$

Similarly, there are projections *projGenre*, *projClassification*, and *projSynopsis*.

For each constant C of type *Genre*, there is a transformation

$$(= C) : Genre \rightarrow \Omega$$

defined by

$$((= C)\ x) = x = C.$$

Similarly, for each string S , there is transformation $(= S)$ that returns true iff its argument is identical to S .

For each integer N , there is a transformation

$$(< N) : Int \rightarrow \Omega$$

defined by

$$((< N)\ m) = m < N.$$

In a similar way, one can define the transformations $(> N)$, $(\geq N)$, and $(\leq N)$.

The transformation

$$StringToText : String \rightarrow Text$$

takes a string as input and returns the list of words in the order that they occur in the string (discarding white space between words). Furthermore, words in the output text are stemmed. Thus

$$(StringToText\ \text{“High Technology”}) = [\text{“high”}, \text{“technolog”}].$$

The transformation

$$listExists_1 : (String \rightarrow \Omega) \rightarrow Text \rightarrow \Omega$$

is defined by

$$listExists_1\ p\ t = \exists x.((p\ x) \wedge (member\ x\ t)).$$

The predicate $(listExists_1\ p)$ checks whether some text (that is, a list of strings) contains a string that satisfies p .

The predicate rewrite system for the function *user_likes_tv_program* is given in Figure 4. The actual strings S used in rewrites of the form

$$top \mapsto (= S)$$

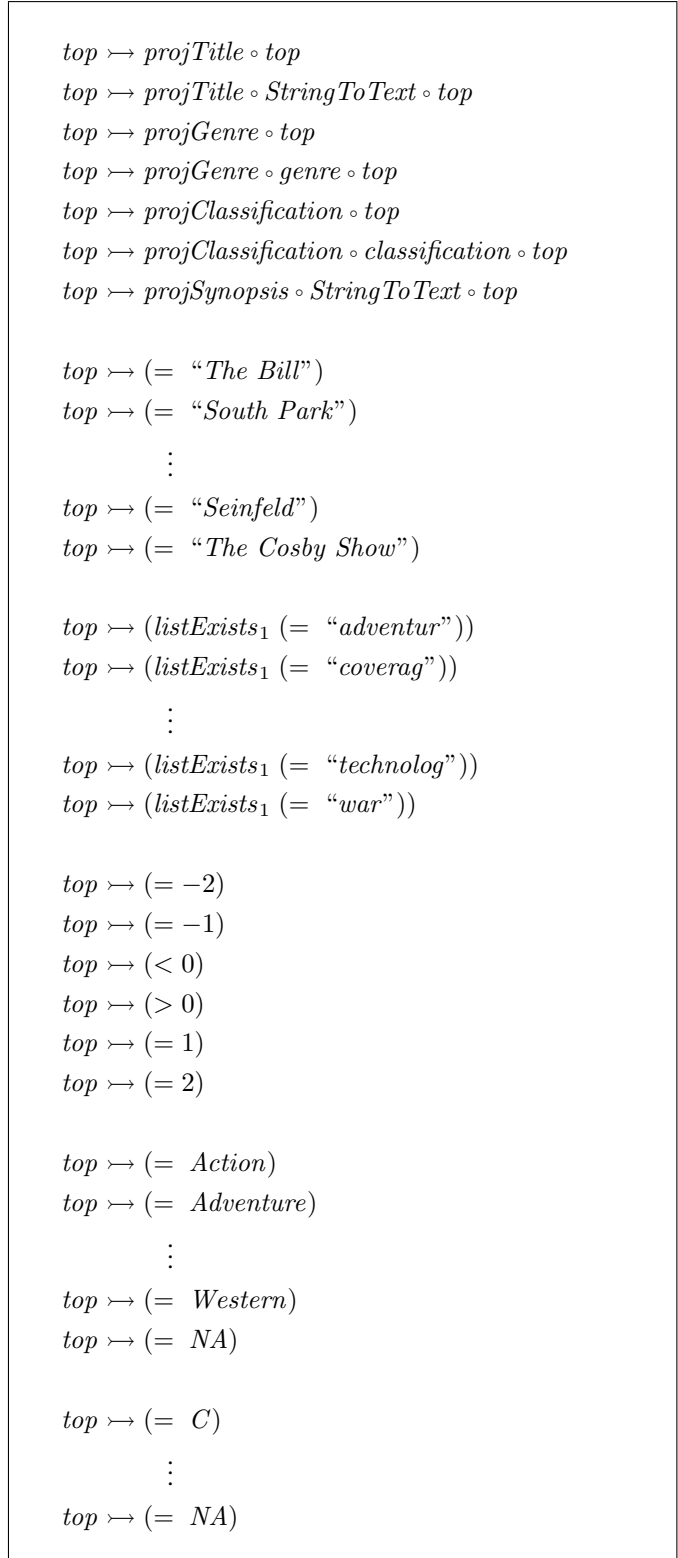


Fig. 4. The predicate rewrite system for the TV recommender

are the titles of all the programs in the (current) set of training examples. In rewrites of the form

$$top \mapsto (listExists_1 (= S)),$$

the strings S appearing are computed as follows. First, the set of all stemmed words appearing in titles or synopses of programs in the (current) set of training examples that are not stop-words is formed. Then for each word in this set we compute the ratio of the number of positive training examples (plus one) in which it appears divided by the number of negative training examples (plus one) in which it appears. The set of words is decreasingly ordered by this ratio and the top 100 are used in the rewrites. The intuition is that these 100 words are good for discriminating between positive and negative examples. Note that the predicate rewrite system is constantly changing as new training examples arrive. A typical learned definition for the function `user_likes_tv_program` is given in Figure 5.

Users also have some level of direct control over the function `user_likes_tv_program` since it is possible to add user-defined rules to the learned definition of the function. For example, a user can add a rule such as:

If the title is “Rugby Union” and the word “Australia” is in the synopsis, then true.

Since the user-defined rules are checked before the learned part of the definition, TV programs that satisfy these conditions are guaranteed to be classified in the way the user desires. As an illustration, the first two cases of the decision list in Figure 5 are rules given by the user. The remainder of the decision list was learned by ALKEMY.

V. EXPERIMENTS AND RESULTS

Here we present the results of a number of experiments measuring the performance of the TV recommender.

Three of the authors used the TV recommender over a number of days. They recorded the programs that they actually watched by supplying a positive training example for the corresponding entry in the TV guide. When the TV recommender was unable to classify a TV program it was highlighted to the user as a special case, with the understanding that they should train the recommender on such cases to improve its performance. When recommendations were requested from the TV recommender, corrective training examples could be supplied for incorrect recommendations. Reinforcing examples could also be supplied for recommendations that the recommender made that were judged correct by the user.

The quality of recommendations was measured by conducting a number of leave-one-out cross-validation experiments. The table of results in Figure 6 records the numbers of true (+) and false (−) examples for each experiment. It also records the numbers of true positive (TP), false positive (FP), false negative (FN), true negative (TN), and unclassified (NC) examples for each experiment. (These five numbers are accumulated across all folds of the corresponding experiment; for the test example in each fold, exactly one of these five outcomes is possible.) Four performance measures, cover,

recall, precision and accuracy, were calculated in the usual way from these values. Each is recorded in a separate column of the table of results. Note that the last three values are calculated on the covered examples only.

Charts of cover, recall, precision and accuracy are given in Figures 7 and 8. Figure 9 is a ROC chart. This plots the true positive rate against the false positive rate.

A. Experiments A, B and C — personalisation to different users

The first set of experiments was designed to test whether the TV recommender could personalise to different users. Starting with a generic system, the three users trained the TV recommender according to their personal TV-viewing habits and program preferences. In addition to indicating their like or dislike for individual TV programs, the users could also supply rules for recommending or not recommending a set of programs and could express a preference for TV program genres and classifications. User 1 supplied no rules. Users 2 and 3 supplied some rules. All users supplied preferences, positive or negative, towards some genres and some (fewer) classifications.

Leave-one-out cross-validation experiments were conducted as experiments A, B and C for users 1 to 3 respectively.

The results in Figure 6 show the number of training examples ranged from 158 for user 1 to 400 for user 3. Cover was in the range 91.50%–96.20%. Recall was in the range 80.00%–88.71%. Precision was in the range 79.07%–94.02%. Accuracy was in the range 87.27%–94.26%. These results indicate a high level of personalisation to each user.

B. Experiments C, D and E — the effect of learning

The second set of experiments was designed to test the effect of learning on the quality of recommendations made by the TV recommender. User 3’s training data from experiment C was used to perform two comparative experiments.

In experiment D learning was switched off. The performance of the TV recommender was then measured in a leave-one-out cross-validation experiment where the prediction was made using only the set of recommendation rules supplied by the user. This set contained nine rules. The results show that the TV recommender achieved perfect recall, precision and accuracy on the test set without using the learner. However, the rules only covered 25.00% of the examples. This indicates that the user-defined rules were in fact confirmed by the test data, but were not general enough to obtain an acceptable cover.

In experiment E the user-defined rules were switched off. The performance of the TV recommender was then measured in a leave-one-out cross-validation experiment where the prediction was made only on the basis of the learned recommendation function. The results show that the TV recommender achieved a similar performance to that obtained in experiment C where use was made of the user-defined rules. Cover increased from 91.50% to 95.25%. Recall decreased from 88.71% to 85.19%. Precision decreased from 94.02% to

92.00%. Accuracy decreased from 94.26% to 92.13%. These results indicate that the learned recommendation function covered most of the examples covered by the user-supplied rules. While the learned rules were not identical to the user-defined rules, one approximated the other.

C. Screen shots

Figures 10 and 11 are screen shots from the TV recommender in action. They show extracts from the TV guide. Program titles highlighted in green are recommended to the user. Titles presented in amber are of programs for which the TV recommender requires further training before it can make a prediction. The tick and cross buttons next to a program description are used to generate a positive or negative training example for that program. Pressing the query button displays a short explanation of why a recommendation was made.

Figure 10 shows recommendations for two different users for the same time slot. Note that the recommendations are quite different. Both users confirmed that the quality of recommendations is high.

Figure 11 shows an extract from the TV guide for the same time period on consecutive days. For a single user, the recommendation for similar programs on different days changed from “Unsure” to “Recommended” after the user generated a training example on the first day by clicking the tick button next the program tagged as “Unsure”. This illustrates the TV recommender’s capacity to track the user’s interests.

D. Further discussion

While the number of training examples here is relatively high, we believe it is reasonable to expect numbers of this magnitude in a real application integrated into a consumer device like TiVo. Positive training examples could be collected in an automatic way by recording what the user actually watches. Negative examples would still need to be collected directly from the user. This should be made as painless as possible.

In the second set of experiments, only nine user-defined recommendation rules were supplied by the user. We acknowledge that a more determined user could supply more rules and so could, in principle, obtain good recommendations without recourse to learning. In practice, the test users were not motivated to devise more complicated sets of rules. The use of machine learning simplifies the task of the user to clicking “Yes” or “No”. Often a single training example is sufficient for the TV recommender to learn a user’s like or dislike for a particular TV program.

The quality of the metadata about individual programs in the TV guide affects the performance of the TV recommender. In the prototype the TV guide was scraped from a publicly accessible web site. The data have a number of shortcomings. Many entries have an empty synopsis entry. Some synopses are very brief and omit important program details. Recommendation quality could be improved with better data.

Finally, we note that the distribution of TV programs in the TV guide itself is not necessarily the same as the distribution of TV programs in the test data of these experiments. The latter is determined by the user, while the former is determined by the broadcasters. Hence these experiments probably overestimate the performance in day-to-day use.

VI. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we described the application of symbolic on-line learning to personalisation of an infotainment demonstrator, concentrating particularly on a TV recommender. With the qualification that more experiments are needed particularly in realistic deployments of the demonstrator with typical users, the results suggest that a high level of personalisation can be achieved.

In future work, we propose to apply the same techniques to personalising other agents, including the music and movie recommenders. Note that the TV recommender would be able to make excellent use of the movie recommender, which would have access to much more information about movies than is given in the TV guide.

We also plan to greatly extend the general approach of this paper. It turns out that there is a unified view of learning and belief revision that makes each of them special cases of a general view of belief acquisition. To implement this unified view, a theorem prover for modal higher-order logic is needed, such as the one described in [2]. In this view, there is no (essential) difference between a user-defined rule and a training example. Furthermore, the ALKEMY learning algorithm presented above can also be used to do belief revision. This approach appears to have great promise for designing architectures for adaptive agents.

ACKNOWLEDGMENT

This research was supported by the Smart Internet Technology Cooperative Research Centre.

REFERENCES

- [1] J.W. Lloyd. *Logic for Learning*. Cognitive Technologies. Springer, 2003.
- [2] J.W. Lloyd. Modal higher-order logic for agents. submitted for publication, 2004.
- [3] K.S. Ng. Alkemy: A learning system based on an expressive knowledge representation formalism. submitted for publication, 2004.
- [4] R. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

APPENDIX

```

(user_likes_tv_program x) =
  if (projTitle ◦ (= "NFL Football") x) then ⊤
  else if (projTitle ◦ (= "English Premier League") x) then ⊤
  else if (projGenre ◦ genre ◦ (< 0) x) then ⊥
  else if (projGenre ◦ (= Drama) x) then ⊥
  else if (projGenre ◦ (= Comedy) x) then ⊥
  else if (projTitle ◦ (= "Sky RaceNight") x) then ⊥
  else if (projTitle ◦ StringToText ◦ (listExists1 (= "sport")) x) then ⊤
  else if (projSynopsis ◦ StringToText ◦ (listExists1 (= "war")) x) then ⊤
  else if (projGenre ◦ (= Current_Affairs) x) then ⊥
  else if (projGenre ◦ (= Business_and_Finance) x) then ⊥
  else if (projTitle ◦ (= "2004 ICC Championship Trophy") x) then ⊤
  else if (projTitle ◦ (= "Every Game of the 2004 AFL Finals Series") x) then ⊥
  else if (projSynopsis ◦ StringToText ◦ (listExists1 (= "technolog")) x) then ⊤
  else if (projTitle ◦ (= "2004 AFL All Australian Awards") x) then ⊥
  else if (projTitle ◦ (= "White Line Fever") x) then ⊥
  else if (projTitle ◦ (= "2004 Finals") x) then ⊥
  else if (projSynopsis ◦ StringToText ◦ (listExists1 (= "renown")) x) then ⊤
  else if (projGenre ◦ (= Music) x) then ⊥
  else if (projSynopsis ◦ StringToText ◦ (listExists1 (= "mani")) x) then ⊤
  else if (projSynopsis ◦ StringToText ◦ (listExists1 (= "open")) x) then ⊤
  else if (projSynopsis ◦ StringToText ◦ (listExists1 (= "american")) x) then ⊤
  ⋮
  else ⊥.

```

Fig. 5. A typical definition for *user_likes_tv_program*

Experiment	+	−	TP	FP	FN	TN	NC	Cover	Recall	Precision	Accuracy
A	68	90	54	5	11	82	6	96.20	83.08	91.53	89.47
B	87	199	68	18	17	172	11	96.15	80.00	79.07	87.27
C	143	257	110	7	14	235	34	91.50	88.71	94.02	94.26
D	143	257	33	0	0	67	300	25.00	100.00	100.00	100.00
E	143	257	115	10	20	236	19	95.25	85.19	92.00	92.13

Fig. 6. Table of results

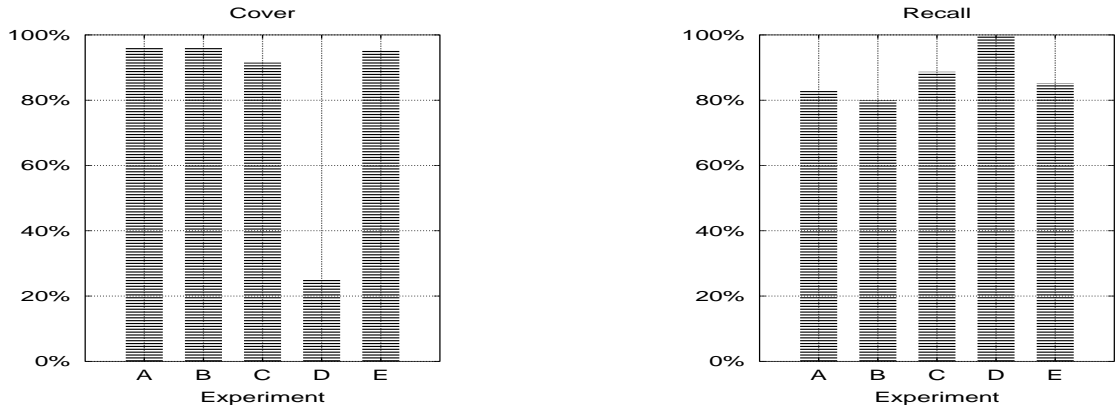


Fig. 7. Cover & recall

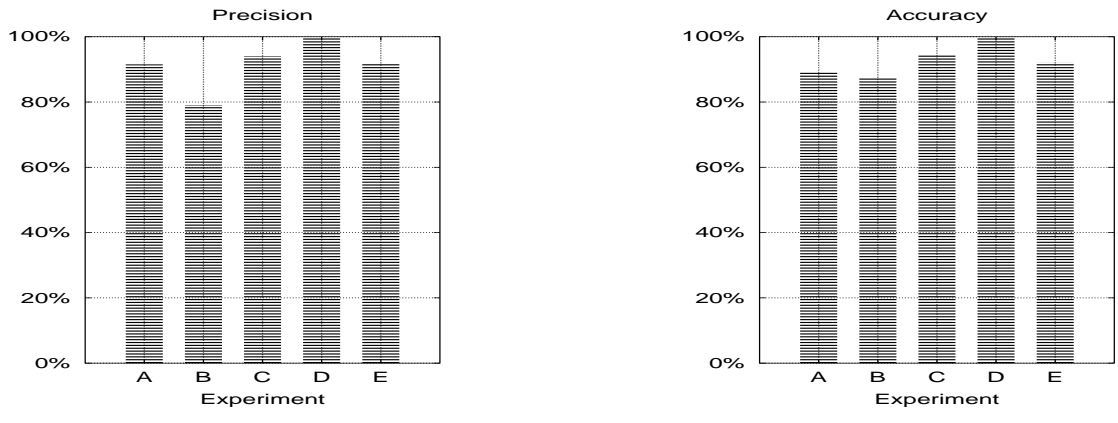


Fig. 8. Precision & accuracy

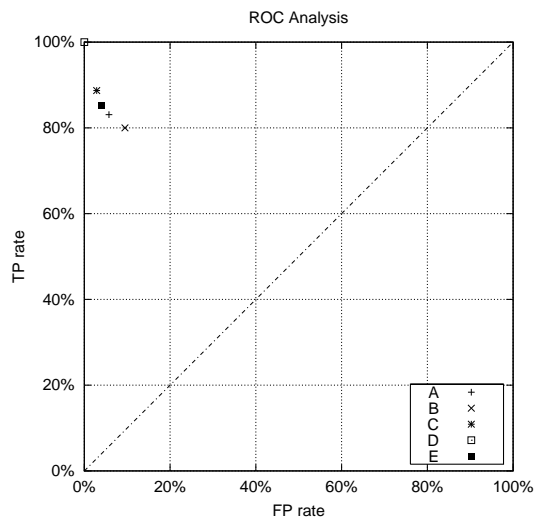


Fig. 9. ROC analysis

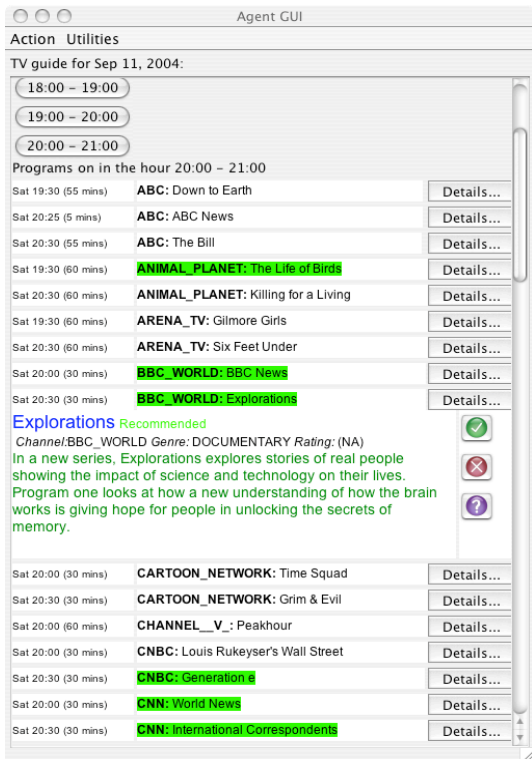


Fig. 10. Recommendations presented to two different users for the same time slot.

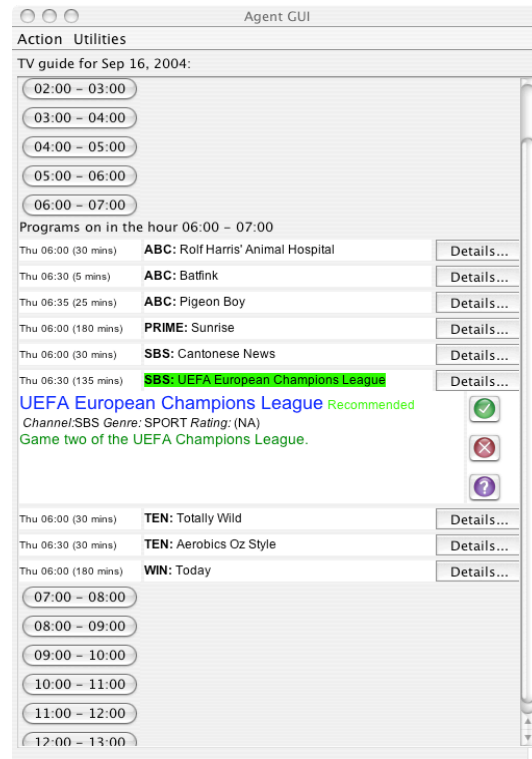
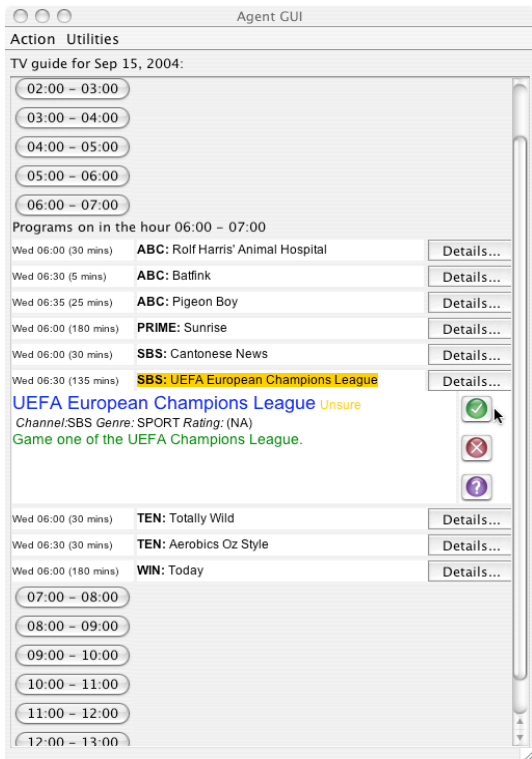


Fig. 11. Recommendations for similar programs on consecutive days changing from “Unsure” to “Recommended” after positive training.