

## Chapter 2

# Post-processing Data Mining Models for Actionability

Qiang Yang

**Abstract** Data mining and machine learning algorithms are, in the most part, aimed at generating statistical models for decision making. These models are typically mathematical formulas or classification results on the test data. However, many of the output models do not themselves correspond to actions that can be executed. In this paper, we consider how to take the output of data mining algorithms as input, and produce collections of high-quality actions to perform in order to bring out the desired world states. This article gives an overview on two of our approaches in this actionable data mining framework, including an algorithm that extracts actions from decision trees and a system that generates high-utility association rules and an algorithm that can learn relational action models from frequent item sets for automatic planning. These two problems and solutions highlight our novel computational framework for actionable data mining.

## 2.1 Introduction

In data mining and machine learning areas, much research has been done on constructing statistical models from the underlying data. These models include Bayesian probability models, decision trees, logistic and linear regression models, kernel machines and support vector machines as well as clusters and association rules, to name a few [1, 11]. Most of these techniques are what we refer to as predictive pattern-based models, in that they summarize the distributions of the training data in one way or another. Thus, they typically stop short of achieving the final objectives of data mining by maximizing utility when tested on the test data. The real action work is waiting to be done by humans, who read the patterns, interpret them and decide which ones to select to put into actions.

---

Qiang Yang

Department of Computer Science and Engineering, Hong Kong University of Science and Technology, e-mail: qyang@cse.ust.hk

In short, the predictive pattern-based models are aimed for human consumption, similar to what the World Wide Web (WWW) was originally designed for. However, similar to the movement from Web pages to XML pages, we also wish to see knowledge in the form of machine-executable patterns, which constitutes truly actionable knowledge.

In this paper, we consider how to take the output of data mining algorithms as input and produce collections of high-quality actions to perform in order to bring out the desired world states. We argue that the data mining methods should not stop when a model is produced, but rather give collections of actions that can be executed either automatically or semi-automatically, to effect the final outcome of the system. The effect of the generated actions can be evaluated using the test data in a cross-validation manner. We argue that only in this way can a data mining system be truly considered as *actionable*.

In this paper, we consider three approaches that we have adopted in post-processing data mining models for generation actionable knowledge. We first consider in the next section how to postprocess association rules into action sets for direct marketing [14]. Then, we give an overview of a novel approach that extracts actions from decision trees in order to allow each test instance to fall in a desirable state (a detailed description is in [16]). We then describe an algorithm that can learn relational action models from frequent item sets for automatic planning [15].

## 2.2 Plan Mining for Class Transformation

### 2.2.1 Overview of Plan Mining

In this section, we first consider the following challenging problem: how to convert customers from a less desirable class to a highly desirable class. In this section, we give an overview of our approach in building an actionable plan from association mining results. More detailed algorithms and test results can be found in [14].

We start with a motivating example. A financial company might be interested in transforming some of the valuable customers from reluctant to active customers through a series of marketing actions. The objective is find an unconditional sequence of actions, a plan, to transform as many from a group of individuals as possible to a more desirable status. This problem is what we call the class-transformation problem. In this section, we describe a planning algorithm for the class-transformation problem that finds a sequence of actions that will transform an initial *undesirable* customer group (e.g., brand-hopping low spenders) into a *desirable* customer group (e.g., brand-loyal big spenders).

We consider a state as a group of customers with similar properties. We apply machine learning algorithms that take as input a database of individual customer profiles and their responses to past marketing actions and produce the customer groups and the state space information including initial state and the next states

after action executions. We have a set of actions with state-transition probabilities. At each state, we can identify whether we have arrived at a *desired class* through a classifier.

Suppose that a company is interested in marketing to a large group of customers in a financial market to promote a special loan sign-up. We start with a customer-loan database with historical customer information on past loan-marketing results in Table 2.1. Suppose that we are interested in building a 3-step plan to market to the selected group of customers in the new customer list. There are many candidate plans to consider in order to transform as many customers as possible from non-sign-up status to a sign-up one. The sign-up status corresponds to a positive class that we would like to move the customers to, and the non-signup status corresponds to the initial state of our customers. Our plan will choose not only low-cost actions, but also highly successful actions from the past experience. For example, a candidate plan might be:

- Step 1: Offer to reduce interest rate;
- Step 2: Send flyer;
- Step 3: Follow up with a home phone call.

**Table 2.1** An example of *Customer* table

Customer	Interest Rate	Flyer	Salary	Signup
John	5%	Y	110K	Y
Mary	4%	N	30K	Y
...	...	...	...	...
Steve	8%	N	80K	N

This example introduces a number of interesting aspects for the problem at hand. We consider the input data source, which consists of customer information and their desirability class labels. In this database of customers, not all people should be considered as candidates for the class transformation, because for some people it is too costly or nearly impossible to convert them to the more desirable states. Our output plan is assumed to be an *unconditional* sequence of actions rather than conditional plans. When these actions are executed in sequence, no intermediate state information is needed. This makes the *group marketing problem* fundamentally different from the direct marketing problem. In the former, the aim is to find a single sequence of actions with maximal chance of success without inserting if-branches in the plan. In contrast, for direct marketing problems, the aim is to find conditional plans such that a best decision is taken depending on the customers’ intermediate state. These are best suited for techniques such as the Markov Decision Processes (MDP) [5, 10, 13].

### 2.2.2 Problem Formulation

To formulate the problem as a data mining problem, we first consider how to build a state space from a given set of customer records and a set of plan traces in the past. We have two datasets as input. As in any machine learning and data mining schemes, the input customer records consist of a set of attributes for each customer, along with a class attribute that describes the customer status. A second source of input is the previous plans recorded in a database. We also have the costs of actions. As an example, after a customer receives a promotional mail, the customer's response to the marketing action is obtained and recorded. As a result of the mailing, the action count for the customer in this marketing campaign is incremented by one, and the customer may have decided to respond by filling out a general information form and mailing it back to the bank. Table 2.2 shows an example of *plan trace table*.

**Table 2.2** A set of plan traces as input

Plan #	State0	Action0	State1	Action1	State2
<i>Plan</i> <sub>1</sub>	<i>S</i> <sub>0</sub>	<i>A</i> <sub>0</sub>	<i>S</i> <sub>1</sub>	<i>A</i> <sub>1</sub>	<i>S</i> <sub>5</sub>
<i>Plan</i> <sub>2</sub>	<i>S</i> <sub>0</sub>	<i>A</i> <sub>0</sub>	<i>S</i> <sub>1</sub>	<i>A</i> <sub>2</sub>	<i>S</i> <sub>5</sub>
<i>Plan</i> <sub>3</sub>	<i>S</i> <sub>0</sub>	<i>A</i> <sub>0</sub>	<i>S</i> <sub>1</sub>	<i>A</i> <sub>2</sub>	<i>S</i> <sub>6</sub>
<i>Plan</i> <sub>4</sub>	<i>S</i> <sub>0</sub>	<i>A</i> <sub>0</sub>	<i>S</i> <sub>1</sub>	<i>A</i> <sub>2</sub>	<i>S</i> <sub>7</sub>
<i>Plan</i> <sub>5</sub>	<i>S</i> <sub>0</sub>	<i>A</i> <sub>0</sub>	<i>S</i> <sub>2</sub>	<i>A</i> <sub>1</sub>	<i>S</i> <sub>6</sub>
<i>Plan</i> <sub>6</sub>	<i>S</i> <sub>0</sub>	<i>A</i> <sub>0</sub>	<i>S</i> <sub>2</sub>	<i>A</i> <sub>1</sub>	<i>S</i> <sub>8</sub>
<i>Plan</i> <sub>7</sub>	<i>S</i> <sub>0</sub>	<i>A</i> <sub>1</sub>	<i>S</i> <sub>3</sub>		
<i>Plan</i> <sub>8</sub>	<i>S</i> <sub>0</sub>	<i>A</i> <sub>1</sub>	<i>S</i> <sub>4</sub>		

### 2.2.3 From Association Rules to State Spaces

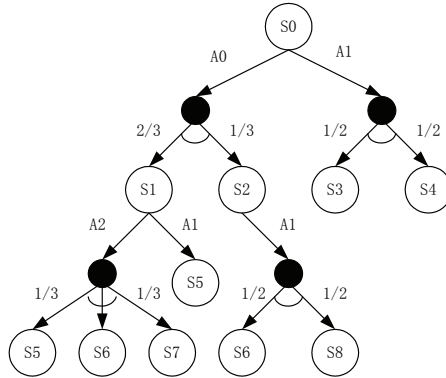
From the customer records, a can be constructed by piecing together the association rule mining [1]. Each state node corresponds to a state in planning, on which a classification model can be built to classify a customer falling onto this state into either a positive (+) or a negative (-) class based on the training data. Between two states in this state space, an edge is defined as a state-action sequence which allows a probabilistic mapping from a state to a set of states. A cost is associated with each action.

To enable planning in this state space, we apply sequential association rule mining [1] to the plan traces. Each rule is of the form:  $S_1, a_1, a_2, \dots, \rightarrow S_n$ , where each  $a_i$  is an action,  $S_1$  and  $S_n$  are the initial and end states for this sequence of actions. All actions in this rule start from  $S_1$  and follow the order in the given sequence to result in  $S_n$ . By only keeping the sequential rules that have high enough support,

we can get segments or paths that we can piece together to form a search space. In particular, in this space, we can gather the following information:

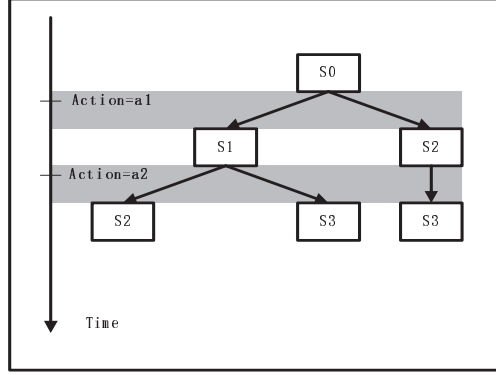
- $f_s(r_i) = s_j$  maps a customer record  $r_i$  to a state  $s_j$ . This function is known as the customer-state mapping function. In our work, this function is obtained by applying odd-log ratio analysis [8] to perform a feature selection in the customer database. Other methods such as Chi-squared methods or PCA can also be applied.
- $p(+|s)$  is the classification function that is represented as a probability function. This function returns the conditional probability that state  $s$  is in a desirable class. We call this function the state-classification function;
- $p(s_k|s_i, a_j)$  returns the transition probability that, after executing an action  $a_j$  in state  $s_i$ , one ends up in state  $s_k$ .

Once the customer records have been converted to states and the state transitions, we are now ready to consider the notion of a plan. To clarify matters, we describe the state space as an AND/OR graph. In this graph, there are two types of node. A *state node* represents a state. From each state node, an action links the state node to an *outcome node*, which represents the outcome of performing the action from the state. An outcome node then splits into multiple state nodes according to the probability distribution given by the  $p(s_k|s_i, a_j)$  function. This AND/OR graph unwraps the original state space, where each state is an OR node and the actions that can be performed on the node form the OR branches. Each outcome node is an AND node, where the different arcs connecting the outcome node to the state nodes are the AND edges. Figure 2.1 is an example AND/OR graph. An example plan in this space is shown in Figure 2.2.



**Fig. 2.1** An example of AND/OR graph

We define the utility  $U(s, P)$  of the plan  $P = a_1 a_2 \dots a_n$  from an initial state  $s$  as follows. Let  $P'$  be the subplan of  $P$  after taking out the first action  $a_1$ ; that is,  $P = a_1 P'$ . Let  $S$  be a set of states. Then the utility of the plan  $P$  is defined recursively



**Fig. 2.2** An example of a plan

$$U(s, P) = \left( \sum_{s' \in S} p(s'|s, a_1) * U(s', P') \right) - cost(a_1) \quad (2.1)$$

where  $s'$  is the next state resulting from executing  $a_1$  in state  $s$ . The plan from the leaf node  $s$  is empty and has a utility

$$U(s, \{\}) = p(+|s) * R(s) \quad (2.2)$$

$p(+|s)$  is the probability of leaf node  $s$  being in the desired class,  $R(s)$  is a reward (a real value) for a customer to be in state  $s$ .

Using Equations 2.1 and 2.2, we can evaluate the utility of a plan  $P$  under an initial state  $U(s_0, P)$ .

Let  $next(s, a)$  be the set of states resulting from executing action  $a$  in state  $s$ . Let  $P(s, a, s')$  be the probability of landing in  $s'$  after executing  $a$  in state  $s$ . Let  $R(s, a)$  be the immediate reward of executing  $a$  in state  $s$ . Finally, let  $U(s, a)$  be the utility of the optimal plan whose initial state is  $s$  and whose first action is  $a$ . Then

$$U(s, a) = R(s, a) + \gamma \max_{a'} \{ \sum_{s' \in next(s, a)} U(s', a') P(s, a, s') \} \quad (2.3)$$

This equation provides the foundation for the class-transformation planning solution: in order to increase the utility of plans, we need to reduce costs ( $-R(s, a)$ ) and increase the utility of the expected utility of future plans. In our algorithm below, we achieve this by minimizing the cost of the plans while at the same time, increase the expected probability for the terminal states to be in the positive class.

### 2.2.4 Algorithm for Plan Mining

We build an AND-OR space using the retained sequences that are both beginning and ending with states and have high enough frequency. Once the frequent sequences are found, we piece together the segments of paths corresponding to the sequences to build an abstract AND-OR graph in which we will search for plans. If  $\langle s_1, a_1, s_2 \rangle$  and  $\langle s_2, a_3, s_3 \rangle$  are two segments found by the string-mining algorithm, then  $\langle s_1, a_1, s_2, a_2, s_3 \rangle$  is a new path in the AND-OR graph.

We use a utility function to denote how “good” a plan is. Let  $s_0$  be an initial state and  $P$  be a plan. Let  $c$  be a function that sums up the cost of each action in the plan. Let  $U(s, P)$  be a heuristic function estimating how promising the plan is for transferring customers initially belonging to state  $s$ . We use this function to perform a best-first search in the space of plans until the termination conditions are met. The termination conditions are determined by the probability or the length constraints in the problem domain.

The overall algorithm follows the following steps.

#### Step 1. Association Rule Mining.

Significant state-action sequences in the state space can be discovered through an association-rule mining algorithm. We start by defining a minimum-support threshold for finding the frequent state-action sequences. *Support* represents the number of occurrences of a state-action sequence from the plan database. Let  $count(seq)$  be the number of times sequence “seq” appears in the database for all customers. Then the support for sequence “seq” is defined as

$$sup(seq) = count(seq),$$

Then, association-rule mining algorithms based on moving windows will generate a set of state-action subsequences whose supports are no less than a user-defined minimum support value. For connection purpose, we only retained substrings both beginning and ending with states, in the form of  $\langle s_i, a_j, s_{i+1}, \dots, s_n \rangle$ .

#### Step 2: Construct an AND-OR space.

Our first task is to piece together the segments of paths corresponding to the sequences to build an abstract AND/OR graph in which we will search for plans. Suppose that  $\langle s_0, a_1, s_2 \rangle$  and  $\langle s_2, a_3, s_4 \rangle$  are two segments from the plan trace database. Then  $\langle s_0, a_1, s_2, a_3, s_4 \rangle$  is a new path in the AND/OR graph. Suppose that we wish to find a plan starting from a state  $s_0$ , we consider all action sequences in the AND/OR graph that start from  $s_0$  satisfying the length or probability constraints.

### Step 3. Define a heuristic function

We use a function  $U(s, P) = g(P) + h(s, P)$  to estimate how “good” a plan is. Let  $s$  be an initial state and  $P$  be a plan. Let  $g(P)$  be a function that sums up the cost of each action in the plan. Let  $h(s, P)$  be a heuristic function estimating how promising the plan is for transferring customers initially belonging to state  $s$ . In A\* search, this function can be designed by users in different specific applications. In our work, we estimate  $h(s, P)$  in the following manner. We start from an initial state and follow a plan that leads to several terminal states  $s_i, s_{i+1}, \dots, s_{i+j}$ . For each of these terminal states, we estimate the state-classification probability  $p(+|s_i)$ . Each state has a probability of  $1 - p(+|s_i)$  to belong to a negative class. The state requires at least one further action to proceed to transfer the  $1 - p(+|s_i)$  percent who remain negative, the cost of which is at least the minimum of the costs of all actions in the action set. We compute a heuristic estimation for all terminal states where the plan leads. For an intermediate state leading to several states, an expected estimation is calculated from the heuristic estimation of its successive states weighted by the transition probability  $p(s_k|s_i, a_j)$ . The process starts from terminal states and propagates back to the root, until reaching the initial state. Finally, we obtain the estimation of  $h(s, P)$  for the initial state  $s$  under the plan  $P$ .

Based on the above heuristic estimation methods, we can express the heuristic function as follows.

$$\begin{aligned} h(s, P) = & \sum_a P(s, a, s') h(s', P') \quad \text{for non terminal states} \\ & (1 - P(+|s)) \text{cost}(a_m) \quad \text{for terminal states} \end{aligned} \quad (2.4)$$

where  $P'$  is the subplan after the action  $a$  such that  $P = aP'$ . In the MPlan algorithm, we next perform a best-first search based on the cost function in the space of plans until the termination condition is met.

### Step 4. Search Plans using MPlan

In the AND/OR graph, we carry out a procedure *MPlan* search to perform a best-first search for plans. We maintain a priority queue  $Q$  by starting with a single-action plan. Plans are sorted in the priority queue in terms of the evaluation function  $U(s, P)$ .

In each iteration of the algorithm, we select the plan with the minimum value of  $U(s, P)$  from the queue. We then estimate how promising the plan is. That is, we compute the expected state-classification probability  $E(+|s_0, P)$  from back to front in a similar way as with  $h(s, P)$  calculation, starting with the  $p(+|s_i)$  of all terminal states the plan leads to and propagating back to front, weighted by the transition probability  $p(s_k|s_i, a_j)$ . We compute  $E(+|s_0, P)$ , the expected value of the state-classification probability of all terminal states. If this expected value exceeds a predefined threshold *Success\_Threshold*  $p_\theta$ , i.e. the probability constraint, we consider the plan to be good enough whereupon the search process terminates. Other-



wise, one more action is appended to this plan and the new plans are inserted into the priority queue.  $E(+|s_0, P)$  is the expected state-classification probability estimating how “effective” a plan is at transferring customers from state  $s_i$ . Let  $P = a_j P'$ . The  $E()$  value can be defined in the following recursive way:

$$\begin{aligned} E(+|s_i, P) &= \sum p(s_k|s_i, a_j) * E(+|s_k, P'), \text{ if } s_i \text{ is a non-terminal state} \\ E(+|s_i, \{\}) &= p(+|s_i), \text{ if } s_i \text{ is a terminal state} \end{aligned} \quad (2.5)$$

We search for plans from all given initial states that corresponds to negative-class customers. We find a plan for each initial state. It is possible that in some AND/OR graphs, we cannot find a plan whose  $E(+|s_0, P)$  exceeds the *Success\_Threshold*, either because the AND/OR graph is over simplified or because the success threshold is too high. To avoid search indefinitely, we define a parameter *maxlength* which defines the maximum length of a plan, i.e. applying the length constraint. We will discard a candidate plan which is longer than the *maxlength* and  $E(+|s_0)$  value less than the *Success\_Threshold*.

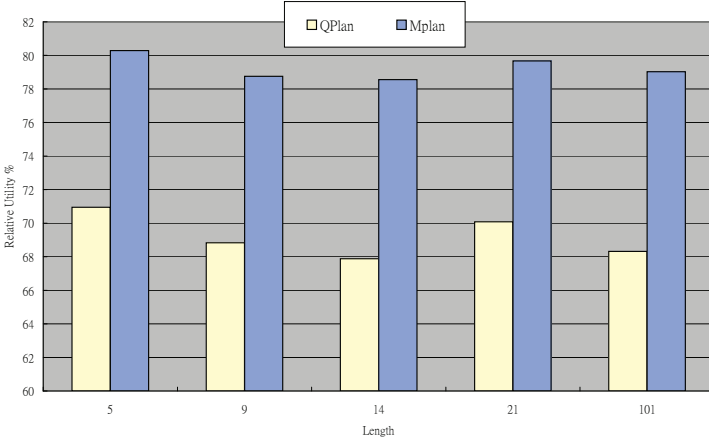
### 2.2.5 Summary

We have evaluated the MPlan algorithm using several datasets, and compared to a variety of algorithms. One evaluation was done with the IBM Synthetic Generator (<http://www.almaden.ibm.com/software/quest/Resources/datasets/syndata.html>) to generate a *Customer* data set with two classes (positive and negative) and nine attributes. The attributes include both numerical values and discrete values. In this data set, the positive class has 30,000 records representing successful customers and the negative class corresponds to 70,000 representing unsuccessful customers. Those 70,000 negative records are treated as starting points for plan trace generation. For the plan traces, the 70,000 negative-class records are treated as an initially failed customer. A trace is then generated for the customer, transforming the customer through intermediate states to a final state. We defined four types of action, each of which has a cost and associated impact on attribute transitions. The total utility of plans is  $TU$ , which is  $TU = \sum_{s \in S} U(s, P_s)$ , where  $P_s$  is the plan found starting from a state  $s$ , and  $S$  is the set of all initial states in the test data set. 400 states serve as the initial states. The total utility is calculated on these states in the test data set.

For comparison, we implemented the *QPlan* algorithm in [12] which uses Q-learning to get an optimal policy and then extracts the unconditional plans from the state space. This algorithm is known as *QPlan*. Q-learning is carried out in the way called batch reinforcement learning [10], because we are processing a very large amount of data accumulated from past transaction history. The traces consisting of sequences of states and actions in plan database are training data for Q-learning. Q-learning tries to estimate the value function  $Q(s, a)$  by value iteration. The major

computational complexity of *QPlan* is on Q-learning, which is carried out once before the extraction phase starts.

Figure 2.3 shows the relative utility of different algorithms versus plan lengths. *OptPlan* has the maximal utility by exhaustive search; thus its plan’s utility is at 100%. *MPlan* comes next, with about 80% of the optimal solution. *QPlan* have less than 70% of the optimal solution.



**Fig. 2.3** Relative utility plan lengths

In this section, we explored data mining for planning . Our approach combines both classification and planning in order to build an state space in which high utility plans are obtained. The solution plans transform groups of customers from a set of initial states to positive class states.

## 2.3 Extracting Actions from Decision Trees

### 2.3.1 Overview

In the section above, we have considered how to construct a state space from association rules. From the state space we can then build a plan. In this section, we consider how to build a decision tree first, from which we can extract actions to improving the current standing of individuals (a more detailed description can be found in [16]). Such examples often occur in customer relationship management (CRM) industry, which is experiencing more and more competitions in recent years. The battle is over their most valuable customers. An increasing number of customers are switching from one service provider to another. This phenomenon is called customer “attrition” , which is a major problem for these companies to stay profitable.

It would thus be beneficial if we could convert a valuable customer from a likely attrition state to a loyal state. To this end, we exploit decision tree algorithms.

Decision-tree learning algorithms, such as ID3 or C4.5 [11], are among the most popular predictive methods for classification. In CRM applications, a decision tree can be built from a set of examples (customers) described by a set of features including customer personal information (such as name, sex, birthday, etc.), financial information (such as yearly income), family information (such as life style, number of children), and so on. We assume that a decision tree has already been generated.

To generate actions from a decision tree, our first step is to consider how to extract actions when there is no restriction on the number of actions to produce. In the training data, some values under the class attribute are more desirable than others. For example, in the banking application, the loyal status of a customer “stay” is more desirable than “not stay”. For each of the test data instance, which is a customer under our consideration, we wish to decide what sequences of actions to perform in order to transform this customer from “not stay” to “stay” classes. This set of actions can be extracted from the decision trees.

We first consider the case of unlimited resources where the case serves to introduce our computational problem in an intuitive manner. Once we build a decision tree we can consider how to “move” a customer into other leaves with higher probabilities of being in the desired status. The probability gain can then be converted into an expected gross profit. However, moving a customer from one leaf to another means some attribute values of the customer must be changed. This change, in which an attribute  $A$ ’s value is transformed from  $v_1$  to  $v_2$ , corresponds to an action. These actions incur costs. The cost of all changeable attributes are defined in a cost matrix by a domain expert. The `leaf-node search` algorithm searches all leaves in the tree so that for every leaf node, a best destination leaf node is found to move the customer to. The collection of moves are required to maximize the net profit, which equals the gross profit minus the cost of the corresponding actions.

For continuous attributes, such as interest rates that can be varied within a certain range, the numerical ranges can be discretized first using a number of techniques for feature transformation. For example, the entropy based discretization method can be used when the class values are known [7]. Then, we can build a cost matrix for each attribute using the discretized ranges as the index values.

Based on a domain-specific cost matrix for actions, we define the net profit of an action to be as follows.

$$P_{Net} = P_E \times P_{gain} - \sum_i COST_i \quad (2.6)$$

where  $P_{Net}$  denotes the net profit,  $P_E$  denotes the total profit of the customer in the desired status,  $P_{gain}$  denotes the probability gain, and  $COST_i$  denotes the cost of each action involved.

### 2.3.2 Generating Actions from Decision Trees

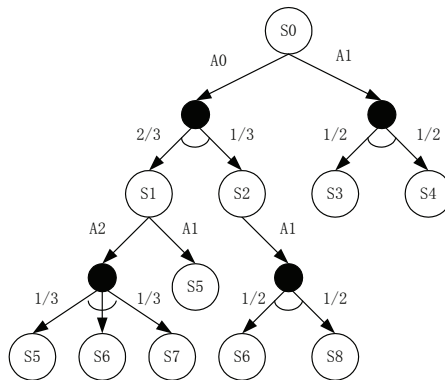
The overall process of the algorithm can be briefly described in the following four steps:

1. Import customer data with data collection, data cleaning, data pre-processing, and so on.
2. Build customer profiles using an improved decision-tree learning algorithm [11] from the training data. In this case, a decision tree is built from the training data to predict if a customer is in the desired status or not. One improvement in the decision tree building is to use the area under the curve (AUC) of the ROC curve [4] to evaluate probability estimation (instead of the accuracy). Another improvement is to use Laplace Correction to avoid extreme probability values.
3. Search for optimal actions for each customer. This is a critical step in which actions are generated. We consider this step in detail below.
4. Produce reports for domain experts to review the actions and selectively deploy the actions.

The following `leaf-node search` algorithm for searching the best actions is the simplest of a series of algorithms that we have designed. It assumes that there is an unlimited number of actions that can be taken to convert a test instance to a specified class:

Algorithm `leaf-node search`

1. For each customer  $x$ , do
  2. Let  $S$  be the source leaf node in which  $x$  falls into;
  3. Let  $D$  be a destination leaf node for  $x$  the maximum net profit  $P_{Net}$ ;
  4. Output  $(S, D, P_{Net})$ ;



**Fig. 2.4** An example of action generation from a decision tree

To illustrate, consider an example shown in Figure 2.4, which represents an overly simplified, hypothetical decision tree as the customer profile of loyal customers built from a bank. The tree has five leaf nodes (A, B, C, D, and E), each with a probability of customers' being loyal. The probability of attritors is simply 1 minus this probability. Consider a customer Jack who's record states that the Service = Low (service level is low), Sex = M (male), and Rate=L (mortgage rate is low). The customer is classified by the decision tree. It can be seen that Jack falls into the leaf node B, which predicts that Jack will have only 20% chance of being loyal (or Jack will have 80% chance to churn in the future). The algorithm will now search through all other leaves (A, C, D, E) in the decision tree to see if Jack can be "replaced" into a best leaf with the highest net profit.

Consider leaf A. It does have a higher probability of being loyal (90%), but the cost of action would be very high (Jack should be changed to female), so the net profit is a negative infinity. Now consider leaf node C. It has a lower probability of being loyal, so the net profit must be negative, and we can safely skip it.

Notice that in the above example, the actions suggested for a customer-status change imply only correlations rather than causality between customer features and status.

### 2.3.3 The Limited Resources Case

Our previous case considered each leaf node of the decision tree to be a separate customer group. For each such customer group, we were free to design actions to act on it in order to increase the net profit. However, in practice, a company may be limited in its resources. For example, a mutual fund company may have a limited number  $k$  (say three) of account managers, each manager can take care of only one customer group. Thus, when such limitations exist, it is a difficult problem to *optimally* merge all leave nodes into  $k$  segments, such that each segment can be assigned to an account manager. To each segment, the responsible manager can several apply actions to increase the overall profit.

This limited-resource problem can be formulated as a precise computational problem. Consider a decision tree  $DT$  with a number of source leaf nodes that correspond to customer segments to be converted and a number of candidate destination leaf nodes, which correspond to the segments we wish customers to fall in.

A solution is a set of  $k$  targetted nodes  $\{G_i, i = 1, 2, \dots, k\}$ , where each node corresponds to a 'goal' that consists of a set of source leaf nodes  $S_{ij}$  and one designation leaf node  $D_i$ , denoted as:  $(\{S_{ij}, j = 1, 2, \dots, |G_i|\} \rightarrow D_i)$ , where  $S_{ij}$  and  $D_i$  are leaf nodes from the decision tree  $DT$ . The goal node is meant to transform customers that belong to the source nodes  $S$  to the destination node  $D$  via a number of attribute-value changing actions. Our aim is to find a solution with the maximal net profit.

In order to change the classification result of a customer  $x$  from  $S$  to  $D$ , one may need to apply more than one attribute-value changing action. An action  $A$  is defined

as a change to an attribute value for an attribute  $Attr$ . Suppose that for a customer  $x$ , the attribute  $Attr$  has an original value  $u$ . To change its value to  $v$ , an action is needed. This action  $A$  is denoted as  $A = \{Attr, u \rightarrow v\}$ .

To achieve a goal of changing a customer  $x$  from a leaf node  $S$  to a destination node  $D$ , a set of actions that contains more than one action may be needed. Specifically, consider the path between the root node and  $D$  in the tree  $DT$ . Let  $\{(Attr_i = v_i), i = 1, 2, \dots, N_D\}$  be set of attribute-values along this path. For  $x$ , let the corresponding attribute-values be  $\{(Attr_i = u_i), i = 1, 2, \dots, N_D\}$ . Then, the actions of the form can be generated:  $ASet = \{(Attr_i, u_i \rightarrow v_i), i = 1, 2, \dots, N_D\}$ , where we remove all null actions where  $u_i$  is identical to  $v_i$  (thus no change in value is needed for an  $Attr_i$ ). This action set  $ASet$  can be used for achieving the goal  $S \rightarrow D$ .

The net profit of converting one customer  $x$  from a leaf node  $S$  to a destination node  $D$  is defined as follows. Consider a set of actions  $ASet$  for achieving the goal  $S \rightarrow D$ . For each action  $Attr_i, u \rightarrow v$  in  $ASet$ , there is a cost as defined in the cost matrix:  $C(Attr_i, u, v)$ . Let the sum of the cost for all of  $ASet$  be  $C_{total, S \rightarrow D}(x)$ .

The BSP problem is to find best  $k$  groups of source leaf nodes  $\{Group_i, i = 1, 2, \dots, k\}$  and their corresponding goals and associated action sets to maximize the total net profit for a given test dataset  $C_{test}$ .

The BSP problem is essentially a maximum coverage problem [9], which aims at finding  $k$  sets such that the total weight of elements covered is maximized, where the weight of each element is the same for all the sets. A special case of the BSP problem is equivalent to the maximum coverage problem with unit costs. Thus, we know that the BSP problem is NP-Complete. Our aim will then be to find approximation solutions to the BSP problem.

To solve the BSP problem, one needs to examine every combination of  $k$  action sets, the computational complexity is  $O(n^k)$ , which is exponential in the value of  $k$ . To avoid the exponential worst-case complexity, we have also developed a greedy algorithm which can reduce the computational cost and guarantee the quality of the solution at the same time.

Initially, our greedy search based algorithm Greedy-BSP starts with an empty result set  $C = \emptyset$ . The algorithm then compares all the column sums that corresponds to converting all leaf nodes  $S_1$  to  $S_4$  to each destination leaf node  $D_i$  in turn. It found that  $ASet_2 = (\rightarrow D_2)$  has the current maximum profit of 3 units. Thus, the resultant action set  $C$  is assigned to  $\{ASet_2\}$ .

Next, Greedy-BSP considers how to expand the customer groups by one. To do this, it considers which additional column will increase the total net profit to a highest value, if we can include one more column. In [16], we present a large number of experiments to show that the greedy search algorithm performs close to the optimal result.

## 2.4 Learning Relational Action Models from Frequent Action Sequences

### 2.4.1 Overview

Above we have considered how to postprocess traditional models that are obtained from data mining in order to generate actions. In this section, we will give an overview on how take a data mining model and postprocess it into a action model that can be executed for plan generation. These actions can be used by robots, software agents and process management software for many advanced applications. A more detailed discussion can be found in [15].

To understand how actions are used, we can recall that automatic planning systems can take formal definitions of actions, an initial state and a goal state description as input, and produce plans for execution. In the past, the task of building action models has been done manually. In the past, various approaches have been explored to learn action models from examples. In this section, we describe our approach in automatically acquiring action models from recorded user plans. Our system is known as ARMS , which stands for *Action-Relation Modelling System* ; a more detailed description is given in [15]. The input to the ARMS system is a collection of observed traces. Our algorithm applies frequent itemset mining algorithm to these traces to find out the collection of frequent action-sets. These actions sets are then taken as the input to another modeling system known as weighted MAX-SAT, which can generate relational actions.

Consider an example input and output of our algorithm in the Depot problem domain from an AI Planning competition [2, 3]. As part of the input, we are given relations such as (clear ?x:surface) to denote that ?x is clear on top and that ?x is of type "surface", relation (at ?x:locatable ?y:place) to denote that a locatable object ?x is located at a place ?y . We are also given a set of plan examples consisting of action names along with their parameter list, such as drive(?x:truck ?y:place ?z:place), and then lift(?x:hoist ?y:crate ?z:surface ?p:place). We call the pair consisting of an action name and the associated parameter list an *action signature*; an example of an action signature is drive(?x:truck ?y:place ?z:place). Our objective is to learn an *action model* for each action signature, such that the relations in the preconditions and postconditions are fully specified.

A complete description of the example is shown in Table 2.3, which lists the actions to be learned, and Table 2.4, which displays the training examples. From the examples in Table 2.4, we wish to learn the preconditions, add and delete lists of all actions. Once an action is given with the three lists, we say that it has a complete action model. Our goal is to learn an action model for every action in a problem domain in order to "explain" all training examples successfully. An example output

from our learning algorithms for the load(?x ?y ?z ?p) action signature is:

action load(?x:hoist ?y:crate ?z:truck ?p:place)

pre: (at ?x ?p), (at ?z ?p), (*lifting* ?x ?y)

del: (*lifting* ?x ?y)

add: (at ?y ?p), (in ?y ?z), (available ?x), (clear ?y)

**Table 2.3** Input Domain Description for Depot Planning Domain

domain	Depot
types	place locatable - object depot distributor - place truck hoist surface - locatable pallet crate - surface
relations	(at ?x:locatable ?y:place) (on ?x:crate ?y:surface) (in ?x:crate ?y:truck) (lifting ?x:hoist ?y:crate) (available ?x:hoist) (clear ?x:surface)
actions	drive(?x:truck ?y:place ?z:place) lift(?x:hoist ?y:crate ?z:surface ?p:place) drop(?x:hoist ?y:crate ?z:surface ?p:place) load(?x:hoist ?y:crate ?z:truck ?p:place) unload(?x:hoist ?y:crate ?z:truck ?p:place)

As part of the input, we need sequences of example plans that have been executed in the past, as shown in Table 2.4. Our job is to formally describe actions such as lift such that automatic planners can use them to generate plans. These training plan examples can be obtained through monitoring devices such as sensors and cameras, or through a sequence of recorded commands through a computer system such as UNIX domains. These action models can then be revised using interactive systems such as GIPO.

### 2.4.2 ARMS Algorithm: From Association Rules to Actions

To build action models, ARMS proceeds in two phases. *Phase one* of the algorithm applies association rule mining algorithms to find the *frequent action sets* from plans that share a common set of parameters. In addition, ARMS finds some frequent relation-action pairs with the help of the initial state and the goal state. These relation-action pairs give us an initial guess on the preconditions, add lists and delete lists of actions in this subset. These action subsets and pairs are used to obtain a set of constraints that must hold in order to make the plans correct.

In *phase two*, ARMS takes the frequent item sets as input, and transforms them into constraints in the form of a weighted MAX-SAT representation [6]. It then solves it using a weighted MAX-SAT solver and produces action models as a result.



**Table 2.4** Three plan traces as part of the training examples

	Plan1	Plan2	Plan3
Initial	$I_1$	$I_2$	$I_3$
Step1	lift(h1 c0 p1 ds0), drive(t0 dp0 ds0)	lift(h1 c1 c0 ds0)	lift(h2 c1 c0 ds0)
State		(lifting h1 c1)	
Step2	load(h1 c0 t0 ds0)	load(h1 c1 t0 ds0)	load(h2 c1 t1 ds0)
Step3	drive(t0 ds0 dp0)	lift(h1 c0 p1 ds0)	lift(h2 c0 p2 ds0), drive(t1 ds0 dp1)
State	(available h1)		
Step4	unload(h0 c0 t0 dp0)	load(h1 c0 t0 ds0)	unload(h1 c1 t1 dp1), load(h2 c0 t0 ds0)
State	(lifting h0 c0)		
Step5	drop (h0 c0 p0 dp0)	drive(t0 ds0 dp0)	drop(h1 c1 p1 dp1), drive(t0 ds0 dp0)
Step6		unload(h0 c1 t0 dp0)	unload(h0 c0 t0 dp0)
Step7		drop(h0 c1 p0 dp0)	drop(h0 c0 p0 dp0)
Step8		unload(h0 c0 t0 dp0)	
Step9		drop(h0 c0 c1 dp0)	
Goal	(on c0 p0)	(on c1 p0) (on c0 c1)	(on c0 p0) (on c1 p1)

$I_1$  : (at p0 dp0), (clear p0), (available h0), (at h0 dp0), (at t0 dp0), (at p1 ds0), (clear c0), (on c0 p1), (available h1), (at h1 ds0)

$I_2$  : (at p0 dp0), (clear p0), (available h0), (at h0 dp0), (at t0 ds0), (at p1 ds0), (clear c1), (on c1 c0), (on c0 p1), (available h1), (at h1 ds0)

$I_3$  : (at p0 dp0), (clear p0), (available h0), (at h0 dp0), (at p1 dp1), (clear p1), (available h1), (at h1 dp1), (at p2 ds0), (clear c1), (on c1 c0), (on c0 p2), (available h2), (at h2 ds0), (at t0 ds0), (at t1 ds0)

The process iterates until all actions are modeled. While the action models that ARMS learns are deterministic in nature, in the future we will extend this framework to learning probabilistic action models to handle uncertainty. Additional constraints are added to allow partial observations to be made between actions, prove the formal properties of the system. In [15], ARMS was tested successfully on all STRIPS planning domains from a recent AI Planning Competition based on training action sequences.

The algorithm starts by initializing the plans by replacing the actual parameters of the actions by variables of the same types. This ensures that we learn action models for the schemata rather than for the individual instantiated actions. Subsequently, the algorithm iteratively builds a weighted MAX-SAT representation and solves it. In each iteration, a few more actions are explained and are removed from the incomplete action set  $\Lambda$ . The learned action models in the middle of the program help reduce the number of clauses in the SAT problem. ARMS terminates when all action schemata in the example plans are learned.

Below, we explain the major steps of the algorithm in detail.

### Step 1: Initialize Plans and Variables

A plan example consists of a sequence of action instances. We *convert* all such plans by substituting all occurrences of an instantiated object in every action instance with the variables of the same type. If the object has multiple types, we generate a clause to represent each possible type for the object. For example, if an object  $o$  has two types *Block* and *Table*, the clause becomes:  $\{(?o = \text{Block}) \text{ or } (?o = \text{Table})\}$ . We then extract from the example plans all sets of actions that are *connected* to each other; two actions  $a_1$  and  $a_2$  are said to be *connected* if their parameter-type list has non-empty intersection. The parameter mapping  $\{?x_1 = ?x_2, \dots\}$  is called a connector.

### Step 2: Build Action and Plan Constraints

A weighted MAX-SAT problem consists of a set of clauses representing their conjunction, where each clause is associated with a weight value representing the priority in satisfying the constraint. Given a weighted MAX-SAT problem, a weighted MAX-SAT solver finds a solution by maximizing the sum of the weight values associated with the satisfied clauses.

In the ARMS system, we have four kinds of constraints to satisfy, representing three types of clauses. They are action, information and plan and relation constraints.

*Action* constraints are imposed on individual actions. These constraints are derived from the general axioms of correct action representations. A relation  $r$  is said to be *relevant* to an action  $a$  if they are the same parameter type. Let  $pre_i, add_i$  and  $del_i$  represent  $a_i$ 's precondition list, add-list and delete list.

### Step 3: Build and Solve a Weighted MAX-SAT Problem

In solving a weighted MAX-SAT problem in Step 3, each clause is associated with a weight value between zero and one. The higher the weight, the higher the priority in satisfying the clause. ARMS assigns weights to the three types of constraints in the weighted MAX-SAT problem described above. For example, every action constraint receives a constant weight  $W_A(a)$  for an action  $a$ . The weight for action constraints is set to be higher than the weight of information constraints.

## 2.4.3 Summary of ARMS

In this section, we have considered how to obtain action models from a set of plan examples. Our method is to first apply association rule mining algorithm on the plan traces to obtain the frequent action sequences. We then convert these frequent action sequences into constraints that are fed into a MAXSAT solver. The solution can

then be converted to action models. These action models can be used by automatic planners to generate new plans.

## 2.5 Conclusions and Future Work

Most data mining algorithms and tools produce only statistical models in their outputs. In this paper, we present a new framework to take these results as input and produce a set of actions or action models that can bring about the desired changes. We have shown how to use the result of association rule mining to build a state space graph, based on which we then performed automatic planning for generating marketing plans. From decision trees, we have explored how to extract action sets to maximize the utility of the end states. For association rule mining, we have considered how to construct constraints in a weighted MAX-SAT representation in order to determine the relational representation of action models.

In our future work, we will research on other methods for actionable data mining, to generate collections of useful actions that a decision maker can apply in order to generated the needed changes.

## Acknowledgement

We thank the support of Hong Kong RGC 621307.

## References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases(VLDB'94)*, pages 487–499. Morgan Kaufmann, September 1994.
2. Maria Fox and Derek Long. PDDL2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.
3. Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Dan Weld, and David Wilkins. PDDL—the planning domain definition language, 1998.
4. Jin Huang and Charles X. Ling. Using auc and accuracy in evaluating learning algorithms. *IEEE Trans. Knowl. Data Eng.*, 17(3):299–310, 2005.
5. L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
6. Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996)*, pages 1194–1201, Portland, Oregon USA, 1996.
7. Ron Kohavi and Mehran Sahami. Error-based and entropy-based discretization of continuous features. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 114–119, Portland, Oregon USA, 1996.

8. D. Mladenic and M. Grobelnik. Feature selection for unbalanced class distribution and naive bayes. In *Proceedings of ICML 1999.*, 1999.
9. M.R.Garey and D.S. Johnson. *Computers and Intractability: A guide to the Theory of NPCompleteness*. 1979.
10. E. Pednault, N. Abe, and B. Zadrozny. Sequential cost-sensitive decision making with reinforcement learning. In *Proceedings of the Eighth International Conference on Knowledge Discovery and Data Mining (KDD'02)*, 2002.
11. J.Ross Quinlan. *C4.5 Programs for machine learning*. Morgan Kaufmann, 1993.
12. R. Sun and C. Sessions. Learning plans without a priori knowledge. *Adaptive Behavior*, 8(3/4):225–253, 2001.
13. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
14. Qiang Yang and Hong Cheng. Planning for marketing campaigns. In *International Conference on Automated Planning and Scheduling (ICAPS 2003)*, pages 174–184, 2003.
15. Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted max-sat. *Artif. Intell.*, 171(2-3):107–143, 2007.
16. Qiang Yang, Jie Yin, Charles Ling, and Rong Pan. Extracting actionable knowledge from decision trees. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):43–56, 2007.



<http://www.springer.com/978-0-387-79419-8>

Data Mining for Business Applications

Cao, L.; Yu, P.S.; Zhang, C.; Zhang, H. (Eds.)

2009, XX, 302 p., Hardcover

ISBN: 978-0-387-79419-8