Front. Comput. Sci., 2017, 11(1): 160–173
DOI 10.1007/s11704-016-5273-4

RESEARCH ARTICLE

Extracting optimal actionable plans from additive tree models

Qiang LU (✉)[1,3], Zhicheng CUI[2], Yixin CHEN[2], Xiaoping CHEN[3]

1 College of Information Engineering, Yangzhou University, Yangzhou 225127, China
2 Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis MO 63130, USA
3 School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China

© Higher Education Press and Springer-Verlag Berlin Heidelberg 2016

Abstract Although amazing progress has been made in machine learning to achieve high generalization accuracy and efficiency, there is still very limited work on deriving meaningful decision-making actions from the resulting models. However, in many applications such as advertisement, recommendation systems, social networks, customer relationship management, and clinical prediction, the users need not only accurate prediction, but also suggestions on actions to achieve a desirable goal (e.g., high ads hit rates) or avert an undesirable predicted result (e.g., clinical deterioration). Existing works for extracting such actionability are few and limited to simple models such as a decision tree. The dilemma is that those models with high accuracy are often more complex and harder to extract actionability from.

In this paper, we propose an effective method to extract actionable knowledge from additive tree models (ATMs), one of the most widely used and best off-the-shelf classifiers. We rigorously formulate the optimal actionable planning (OAP) problem for a given ATM, which is to extract an actionable plan for a given input so that it can achieve a desirable output while maximizing the net profit. Based on a state space graph formulation, we first propose an optimal heuristic search method which intends to find an optimal solution. Then, we also present a sub-optimal heuristic search with an admissible and consistent heuristic function which can remarkably improve the efficiency of the algorithm. Our experimental results demonstrate the effectiveness and efficiency of the proposed algorithms on several real datasets in the application domain of personal credit and banking.

Keywords actionable knowledge extraction, machine learning, additive tree models, state space search

1 Introduction

Machine learning techniques have achieved great success. Tremendous efforts have made on achieving high accuracy and efficiency. However, in many applications such as targeted marketing, users need not only prediction, but also suggestions on courses of actions to maximize profits or avert undesirable outcomes. Currently, in practice, users often need to spend much manual efforts digging out the real "nuggets" of knowledge, the actions, in order to achieve their goals.

In this paper, we study automatic extraction of actionability from certain machine learning models. Given an input and a desired output, the actionability of a model is the ability to identify a set of changes to the input features that transforms the prediction of this input to the desired output. Automatic extraction of *actionable knowledge* rather than resorting to domain experts is badly needed in many applications. We elaborate this using two real cases that people encountered in practice.

Example 1 In one of the world's largest telecommunication carriers ("Company A"), data scientists need to build models to predict customer churning. A set of more than 200 features related with clients' service plans, call quality, roaming coverage, Internet connection, and social-economic attributes are used. They use a random forest model after ex-

Received July 5, 2015; accepted January 15, 2016

E-mail: lvqiang@yzu.edu.cn

tensive testing. In addition to having a predictive model, it is needed to extract actionable knowledge to intervene with these customers with high churn probability and try to retain them. In general, it is much cheaper to retain existing customers than obtaining new ones. It is especially valuable to retain large, enterprise-level customers.

There are certain actions that Company A can take, such as making phone contacts, sending promotional coupons, offering data gifts, offering discounts for mobile devices, providing roaming services, etc. Each of these actions can change certain attributes of a customer. However, such intervention incurs costs to Company A. Therefore, for each customer, we want to extract an optimal set of actions that maximizes the expected gain minus the costs. □

Example 2 A major hospital uses machine learning models to predict sudden deterioration for hospitalized patients based on their vital signs (such as blood pressure, heart rate, oxygen saturation, etc.) [1, 2]. The current models are engineered to give alert of an impeding deterioration in the next 48 hours with pretty high precision, but cannot suggest any potential intervention.

A model with actionability can be readily translated into intervening actions to help avert potential clinical deterioration when the model predicts an undesired outcome. For example, if the model not only predicts that a patient is likely to have respiratory arrest in the next 48 hours, but also provides actionable knowledge such as increasing oxygen saturation, then the doctors can exploit the suggested actions to reduce the likelihood of sudden deterioration. □

It should be noted that there are some existing techniques to extract knowledge from complex models such as random forests or neural networks. Such techniques include feature sensitivity analysis [3] and extracting a simpler model (typically a decision tree) from a complex model [4]. In fact, a random forest itself can be used to rank the importance of features. For certain models with a continuous closed-form function such as neural networks, the direction of feature change can be extracted by a gradient-based method [5]. However, such approaches have some major drawbacks and are not suitable for our problem. First, they are derived from the entire population of training data and do not provide actionable knowledge customized for each individual. For Example 1, a customer who travels a lot may be more sensitive to roaming charges while someone who watches a lot of videos may be more interested in the data plan. For Example 2, patients have very different alert-triggering reasons, such as sepsis, respiratory, or cardiovascular problems. Automatic suggestion on potential intervention actions will provide valuable guidance to physicians and rapid response teams. Hence, extraction of individualized actionable knowledge is much needed. Thus, feature selection and ranking algorithms are not useful here because their result is not personalized for each patient. Second, these techniques did not take the cost of making changes into consideration.

In order to address these challenges, we propose an effective framework to extract actionable knowledge from additive tree models (ATMs), which encompasses some of the most popular models such as random forest, adaboost and gradient boosted trees. The reasons why we choose ATMs are: 1) In addition to superior classification/regression performance, ATMs enjoy many appealing properties that many other models lack [6], including the support for multi-class classification and natural handling of missing values and data of mixed types. 2) Often referred to as the best off-the-shelf classifiers [6], ATMs have been widely deployed in many industrial products such as Kinect [7] and face detection in camera [8], and are the must-try methods for some competitions such as web search ranking [9].

However, extracting actionable knowledge from ATMs is more difficult since the tree models are discrete in nature and we cannot directly compute the gradients. The problem becomes even harder when it is compounded with the fact that changing different features may have different costs. For example, the cost of making a change to gender should be considered enormous or unrealistic. In fact, we will show that it is an NP-hard problem.

In this article, we rigorously formulate the optimal actionable plan (OAP) problem for a given ATM, which is to extract a sequence of actions for a given input so that it can achieve a desirable output while maximizing the net profit. After proving its NP-hardness, we propose a state space graph formulation which allows us to tackle it using the state space search algorithm. Moreover, we first introduce an optimal state space search method which tries to find an optimal solution. We also propose a sub-optimal state space search algorithm with an admissible and consistent heuristic function which can remarkably improve the efficiency of the search. Experiments on four real-world datasets on personal banking and credits show that the proposed sub-optimal method is superior in both quality and efficiency as compared to other baseline methods.

To show the advantages of our scheme, we summarize our contributions as follows:

1) We start with an formulation of the OAP problem for a given ATM and prove it is NP-hard.
2) We transfer the OAP problem to a state space graph search problem and solve it by the state space search algorithm. Specifically, we first present an optimal algorithm which aims to find an optimal solution. Further more, to achieve a good balance of search time and solution quality, we propose a sub-optimal algorithm which can find a sub-optimal solution within the very limited time.
3) Our state space search algorithms can provide actionable knowledge customized for each individual and take account of the cost of making changes.

The rest of this article is organized as follows. In Section 2, we introduce the formal definition of ATMs. In Section 3, we formally define the OAP problem based on an ATM and prove the OAP problem is NP-hard. In Section 4, we propose an encoding method which can transfer the OAP problem to a state space search problem and present an optimal algorithm and a sub-optimal algorithm to solve the problem. We present the experimental results in a variety of four datasets in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 Additive tree models (ATMs)

Additive tree models (ATMs) include a wide range of tree-based models that are widely used in machine learning applications. Special cases include random forests [10] and boosted trees. For example, random forest is currently one of the most popular and best off-the-shelf classifiers.

Note that our formulation is very general and encompasses several most important and popular models for classification and regression, not only random forest but also other additive tree models, such as adaboost, gradient boosting trees. Thus, the proposed action extraction algorithm has very wide applications.

In this paper, we assume we are given a dataset $\{X, Y\}$, where $X = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N\}$ is the set of feature vectors and $Y = \{y^1, y^2, \ldots, y^N\}$ is the set of labels. Each feature vector $\mathbf{x}^i = (x^i_1, x^i_2, \ldots, x^i_M)$ has M attributes, where each attribute $x_j \in D_j$ has a finite or infinite domain D_j. x_j can be either categorical or numerical. The label y's have a finite discrete domain D_Y. In the following, we also use $\mathbf{x} = (x_1, x_2, \ldots, x_M)$ instead of $\mathbf{x}^i = (x^i_1, x^i_2, \ldots, x^i_M)$ when there is no confusion. For ease of presentation, we only consider binary output where the label $y \in \{0, 1\}$. It is straightforward to extend this work to the case of multi-way classification.

An ATM consists of an ensemble of K decision trees. Each decision tree k has an output function $o_k(\mathbf{x})$ which takes an input $\mathbf{x}$ and outputs a label $y \in D_Y$. The overall output is

$$H(\mathbf{x}) = \sum_{k=1}^{K} w_k o_k(\mathbf{x}), \tag{1}$$

where $w_k \in \mathcal{R}$ are weights. $H(\mathbf{x})$ also has a probabilistic interpretation. For any label $c \in D_Y$,

$$p\Big(y = c \mid \mathbf{x}\Big) = \frac{\sum_{k=1}^{K} w_k I(o_k(\mathbf{x}) = c)}{\sum_{k=1}^{K} w_k}, \tag{2}$$

where $I(o_k(\mathbf{x}) = c)$ is an indicator function which evaluates to 1 if $o_k(\mathbf{x}) = c$ and 0 otherwise.

For a given input $\mathbf{x}$, the typical predicted label y_H by an ATM $H(\mathbf{x})$ is simply:

$$y_H(\mathbf{x}) = \underset{c \in D_Y}{\operatorname{argmax}} \;\; p(y = c|\mathbf{x}). \tag{3}$$

The ATM model in Eq. (1) is general enough to encompass the following popular models.

• **Random forests** A random forest is generated as follows [10]. For $k = 1, 2, \ldots, K$,

1) Sample n_k $(0 < n_k < N)$ instances from the dataset with replacement.
2) Train an un-pruned decision tree on the n_k sampled instances. At each node, choose the split point from a number of randomly selected features rather than all features.

The output of the random forest is simply the average from all the trees $H(\mathbf{x}) = \sum_{k=1}^{K} \frac{1}{K} o_k(\mathbf{x})$, which is a special case of the ATM in Eq. (1) with $w_k = \frac{1}{K}$.

• **Boosted trees** Boosting is a general method that ensembles multiple weak learners to make a strong final model [11]. The boosting method trains the additive model sequentially in a forward stage-wise manner. Suppose $H_k(\cdot)$ is the resulting model up to stage k. The model of the next stage is

$$H_{k+1}(\mathbf{x}) \leftarrow H_k(\mathbf{x}) + \alpha_k o_k(\mathbf{x}),$$

where $o_k(\cdot)$ is the weak learner obtained at stage k and α_k is the weight of this weak learner. The final model turns out to be a weighted sum of all trees: $H(\mathbf{x}) = \sum_{k=1}^{K} \alpha_k o_k(\mathbf{x})$, which is a special case of the ATM in Eq. (1) with $w_k = \alpha_k$. There are two common ways to train the weak learners, leading to two different models: adaboost [11] and gradient boosted trees [12].

In adaboost, at stage k, the weight α_k and the tree model $o_k(\cdot)$ are jointly optimized to minimize the loss function L of the resulting model H_{k+1}:

$$\underset{\alpha_k,o_k}{\text{minimize}} \sum_{i=1}^{N} L(y^{(i)}, H_k(\mathbf{x}^{(i)}) + \alpha_k o_k(\mathbf{x}^{(i)})), \tag{4}$$

where L is a loss function measuring the difference between the true label $y^{(i)}$ and the model $H_{k+1}(\mathbf{x}^{(i)})$. In particular, when L is the exponential loss ($L(a, b) = e^{-ab}$), there is a nice closed-form solution for α_k, and f_k can be learned by training on the *weighted* instances.

Gradient boosting [12] counts each addition to the current model $H_k(\cdot)$ as a gradient update of Eq. (4) in the function space of $H_k(\cdot)$, where α_k is the learning rate and $o_k(\cdot)$ is the negative gradient of minimizing the loss function L. Specifically, $o_k(\cdot)$ is trained so that

$$o_k(\mathbf{x}^{(i)}) \approx -\frac{\partial L(y^{(i)}, H_k(\mathbf{x}^{(i)}))}{\partial H_k(\mathbf{x}^{(i)})},$$

which is equivalent to training a tree on the original instances with new labels defined by the negative gradient.

3 Extracting optimal actionable plans from ATMs

The OAP problem can be informally described as follows. Given an ATM and an input $\mathbf{x}$, we would like to find a set of actions that, when applied to the input, change its predicted class to a desirable one. Each action changes a number of features in $\mathbf{x}$. There is a reward for reaching the target class and a cost for taking an action. The goal is to find a plan that gives the best expected net profit.

Definition 1 (Feature partitions) Given an ATM, each feature x_i, $i = 1, 2, \ldots, M$, is split into a number of partitions.

1) If x_i is categorical with n categories, then x_i naturally has n partitions.
2) If x_i is numerical, we assume each tree node branches in the form of $x_i \geqslant b$ where $b \in \mathcal{R}$ is a *splitting point*. If there are n splitting points for x_i in all the trees in the ATM model, feature x_i is naturally split into $n + 1$ partitions.

In the following, let n_i be the number of partitions for feature x_i, $i = 1, 2, \ldots, M$.

Naturally, each feature vector $\mathbf{x} = (x_1, x_2, \ldots, x_M)$ corresponds to a unique feature partition vector $(p_1, p_2, \ldots, p_M)$, where p_i is x_i's partition index. With a little abuse of notations, we still use $\mathbf{x}$ to denote the feature partition vector, since all $\mathbf{x}$'s that map to the same feature partition vector are non-distinguishable in our problem.

Definition 2 (Feature transition) Given an ATM model, a feature transition $\mathcal{T}$ is a tuple $\mathcal{T} = (x_i, p, q)$, where x_i, $i = 1, 2, \ldots, M$, is a feature, and p and q are two different partitions of x_i. A feature transition is *applicable* to a given input $\mathbf{x}$ if and only if x_i is in partition p.

Intuitively, a feature transition changes a feature x_i from partition p to another partition q.

Definition 3 (Action) An action a is a set of feature transitions, $a = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{|a|}\}$, where each feature x_i involves with at most one of the transitions. Action a is *applicable* to an input $\mathbf{x}$ if and only if all feature transitions in a are applicable to $\mathbf{x}$. Each action has a *cost* $\pi(a) > 0$.

For an input $\mathbf{x}$ with an applicable action a, we use $\mathbf{x} \oplus a$ to denote the resulting feature partition vector after applying all the transitions in a to $\mathbf{x}$.

In practice, the actions and costs are determined by domain experts. Typically, some features can be changed with a reasonable cost, such as medicine intake volumes, interest rates, and discount rates. These features are called *soft attributes* [13]. On the other hand, the values of some features, such as gender and marital status, cannot be changed with any reasonable cost. These features are *hard attributes* and we do not define any action associated with them.

A random forest H with two trees is shown in Fig. 1. Among the three features, x_1 is a hard attribute that cannot be changed with a reasonable cost. x_2 and x_3 are soft attributes that can be changed by certain actions. The split points for x_2 in the two trees are 2 and 3, respectively, leading to three partitions for this feature, i.e., $(-\infty, 2)$, $[2, 3)$ and $[3, \infty)$. Partitions are $\{married\}, \{single\}$ for x_1 and $(-\infty, 4)$, $[4, \infty)$ for x_3.

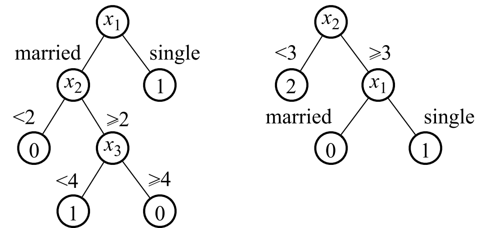


Fig. 1 An illustration of random forest containing three features

Given a feature vector $\mathbf{x}$, we use $p(\mathbf{x})$ to represent $p(y =$

$c|\mathbf{x})$.

Definition 4 (OAP problem) An OAP problem is a tuple $\Pi = (H, \mathbf{x}^I, c, O)$, where H is an ATM defined in Eq. (1), $\mathbf{x}^I$ is a given input, $c \in D_Y$ is a class label, and O is a set of actions. The goal is to find a sequence of actions $A = (a_1, a_2, \ldots, a_n), a_i \in O$, to solve:

$$\max_{A \subseteq O} \ F(A) = (p(\tilde{\mathbf{x}}) - p(\mathbf{x}^I))R_c - \sum_{a_i \in A} \pi(a_i), \quad (5)$$

$$\text{subject to: } \ p(\tilde{\mathbf{x}}) \geqslant z, \quad (6)$$

where $0 < z \leqslant 1$ is a constant, $R_c > 0$ is the reward for reaching class label c, and

$$\tilde{\mathbf{x}} = \mathbf{x}^I \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n, \quad (7)$$

is the new sample after applying the actions in A to $\mathbf{x}^I$ sequentially. Note that $F(A) \leqslant R_c$.

Intuitively, the OAP problem entails finding a set of actions to make changes to any input sample so that the predicted class membership is changed while the net profit $F(A)$ (the expected incremental profit minus the costs of action) is maximized.

Given the input feature vector $\mathbf{x} = \{married, 1, 5\}$ and $\mathbf{x}^G = \{x|H(x) = 1\}$, the transformed OAP problem is $\Pi_{OAP}(H, \mathbf{x}^I, \text{“}good\text{”}, O)$, where $\mathbf{x}^I = \{married, 1, 5\}$ and $\mathbf{x}^G = \{x|H(x) = 1\}$. The feature partitions of variables are $\{0, 1\}$, $\{0, 1, 2\}$, $\{0, 1\}$, and $s_I = \{0, 0, 1\}$. The goal state of the plan is $s = \{0, 1, 0\}$ and the corresponding feature vector is $\mathbf{x}^G = \{married, 2, 3\}$. The actions are defined as changing the value of x_2 and x_3 according to Definition 3. A plan of the problem is (a_1, a_2), where $a_1 = \{(x_2, 0, 1)\}$ and $a_2 = \{(x_3, 1, 0)\}$.

We now show that the OAP problem is an NP-hard problem by reducing from the DNF-MAXSAT problem [14]. The DNF-MAXSAT problem is defined over a boolean formula in disjunctive normal form with M binary variables and K clauses. The problem is to determine if there exists an assignment such that there are at least m clauses that evaluate to *true*.

Theorem 1 The OAP problem is NP-hard.

Proof We reduce a well-known NP-hard problem, DNF-MAXSAT, into the OAP problem with binary class labels. Given any DNF-MAXSAT problem with K clauses and M variables, we construct an ATM model with M features and K trees where each tree represents a clause. Each literal in the clause corresponds to a node in the tree. For $k = 1, 2, \ldots, K$, the output of tree k is 1 if and only if all literals in clause k are made true. Finally, the DNF-MAXSAT problem reduces to the OAP problem with $R_c = 0$, $\pi(a) = 0$, and $z = m/K$. □

Since the OAP problem is in general an NP-hard problem, it usually does not have any efficient algorithm for optimally solving it. In this paper, we propose a sub-optimal state space search algorithm to efficiently solve this problem with high solution quality.

4 State space search for solving the OAP problem

State space search is a core technique for AI. It is widely used in domains such as automated planning, robotics, path finding, and video games. For many NP-hard combinatorial problems, state space search is the dominating solution technique. For example, in the recent International Planning Competitions[1)], the First-Prize winners are all based on state space search. Formally, state space search is usually to find the shortest path from an initial state to a goal state on a state space graph for planning and path finding problem formulations.

A key feature of state space search is that its search efficiency is largely dependent on the heuristic function guiding the search. In many cases, more accurate heuristic functions usually lead to faster search. However, more accurate heuristic functions sometimes also need more computation time [15]. Thus, we need to take a good balance between these two aspects, particularly for large-size instances. For a state space graph where all edges have positive weights, when the heuristic function is *admissible* (always underestimating the true distance), the well-known A* search can guarantee to find the optimal solution under very mild conditions.

Definition 5 (State space graph for OAP) For the OAP problem, the state space graph is a directed graph $G = (\mathcal{F}, E)$, where the vertex set $\mathcal{F}$ is the set of all possible states (partition vectors) and the edge set E satisfies that for any two states $\mathbf{x}_{i-1}, \mathbf{x}_i \in \mathcal{F}$, there exists an edge $(\mathbf{x}_{i-1}, \mathbf{x}_i) \in E$ if and only if there is an action $a \in O$ such that $\mathbf{x}_{i-1} \oplus a = \mathbf{x}_i$; the weight for this edge is

$$w(\mathbf{x}_{i-1}, \mathbf{x}_i) = \left[p(\mathbf{x}_i) - p(\mathbf{x}_{i-1})\right]R_c - \pi(a), \quad (8)$$

where $p(\mathbf{x}_i)$ is the ATM output as defined in Eq. (2).

Although the state space graph G may contain many states, it is important to note that the state space search expands

1) http://icaps-conference.org/index.php/Main/Competitions

states on-demand and typically only examines a tiny fraction of $\mathcal{F}$.

Definition 6 (Goal states) For an OAP problem in Definition 4 with a state space graph $G = (\mathcal{F}, E)$, a node $\tilde{\mathbf{x}}$ is a goal state if and only if $p(\tilde{\mathbf{x}}) \geqslant z$ where z is a constant.

Theorem 2 The OAP problem in Definition 4 is equivalent to finding a path with the largest total weights on the state space graph $G = (\mathcal{F}, E)$ from a given state $\mathbf{x}^I$ to a goal state.

Proof Let the path with the largest total weights from $\mathbf{x}$ to a goal state on G be $\mathbf{x}_0 = \mathbf{x}^I, \mathbf{x}_1, \ldots, \mathbf{x}_n = \tilde{\mathbf{x}}$, and let $\mathbf{x}_{i-1} \oplus a_i = \mathbf{x}_i$ for $i = 1, 2, \ldots, n$, the total weights of the path are

$$\sum_{i=1}^{n} w(\mathbf{x}_{i-1}, \mathbf{x}_i) = \sum_{i=1}^{n} \left[p(\mathbf{x}_i) - p(\mathbf{x}_{i-1})\right] R_c - \sum_{i=1}^{n} \pi(a_i)$$
$$= (p(\tilde{\mathbf{x}}) - p(\mathbf{x}^I)) R_c - \sum_{i=1}^{n} \pi(a_i).$$

When such a solution path is found, we know that $\tilde{\mathbf{x}}$ satisfies Eq. (6) according to Definition 6 and the path maximizes Eq. (5). The theorem can be seen from Definition 4. □

Note that the edge weight $w(\mathbf{x}_{i-1}, \mathbf{x}_i)$ might be a negative number, thus the widely used A^* algorithm for optimal state space search is not suitable for the OAP problem. Here we first introduce an optimal algorithm based on the state space search.

Algorithm 1 shows the anytime fashion optimal state space search algorithm. It maintains two data structures, a max heap (which takes ⟨value, key⟩ pairs) and a closed list, and performs the following main steps:

1) Initialize the f^* and $plan^*$. Add the pair $\langle \mathbf{x}^I, f(\mathbf{x}^I)\rangle$ to the max heap (lines 1–4).
2) Pop the state $\mathbf{x}$ from the heap with the largest $f(\mathbf{x})$ (line 7).
3) If $\mathbf{x}$ is a goal state and $g(\mathbf{x}) > f^*$, update f^* and the best plan $plan^*$ (lines 8–10).
4) If $\mathbf{x}$ is not in the closed list, add $\mathbf{x}$ to the closed list and for each edge $(\mathbf{x}, \mathbf{x}') \in E$, add $\langle \mathbf{x}', f(\mathbf{x}')\rangle$ to the max heap (lines 12–16). Otherwise the state $\mathbf{x}$ has been searched before. If it finds a path with larger total weights from $\mathbf{x}^I$ to $\mathbf{x}$, update the $f(\mathbf{x})$ in max heap and the predecessor of $\mathbf{x}$ in the closed list (lines 20–23).
5) Repeat from Step 2.

The closed list is implemented as a set with highly efficient hashing-based duplicate detection. At the core of search is an evaluation function $f(\mathbf{x})$ that decides the order of state expansions.

Algorithm 1 Optimal state space search

```
Input: G = (F, E), g, h
 1: f* ← −∞
 2: plan* ← ∅
 3: compute f(x^I) = g(x^I) + h(x^I)
 4: MaxHeap.push(⟨x^I, f(x^I)⟩)
 5: ClosedList ← {}
 6: while MaxHeap is not empty do
 7:   x ← MaxHeap.pop()  // x has the maximum f(x)
 8:   if p(x) ⩾ z and g(x) > f* then
 9:     f* ← g(x)
10:     plan* ← path(x^I, x)
11:   end if
12:   if x ∉ ClosedList then
13:     ClosedList=ClosedList ∪{x}
14:     for each (x, x′) ∈ E do
15:       compute f(x′) = g(x′) + h(x′)
16:       MaxHeap.push(⟨x′, f(x′)⟩)
17:     end for
18:   else
19:     x_old ← MaxHeap.find(x)
20:     if g(x) > g(x_old) then
21:       compute f(x) = g(x) + h(x)
22:       MaxHeap.push(⟨x, f(x)⟩)
23:       update the predecessor of x in ClosedList
24:     end if
25:   end if
26: end while
27: return plan*
```

Definition 7 (Evaluation function) For a state space graph $G = (\mathcal{F}, E)$, the evaluation function $f(\mathbf{x})$ on $\mathbf{x} \in \mathcal{F}$ has the form $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$, where $h(\mathbf{x}) : \mathcal{F} \to \mathcal{R}$ is a *heuristic function*.

In $f(\mathbf{x})$, $g(\mathbf{x})$ is the *"current" weights* (the total weights of the path leading up to $\mathbf{x}$), while $h(\mathbf{x})$ is the *estimated "future" weights*. Let the path from $\mathbf{x}^I$ to a state be $\mathbf{x}_0 = \mathbf{x}^I, \mathbf{x}_1, \ldots, \mathbf{x}_m = \mathbf{x}$, and let $\mathbf{x}_{i-1} \oplus a_i = \mathbf{x}_i$ for $i = 1, 2, \ldots, m$, the current weights are

$$g(\mathbf{x}) = \sum_{i=1}^{m} w(\mathbf{x}_{i-1}, \mathbf{x}_i)$$
$$= \sum_{i=1}^{m} \left[p(\mathbf{x}_i) - p(\mathbf{x}_{i-1})\right] R_c - \sum_{i=1}^{m} \pi(a_i)$$
$$= \left[p(\mathbf{x}) - p(\mathbf{x}^I)\right] R_c - \sum_{i=1}^{m} \pi(a_i). \tag{9}$$

The *perfect heuristic* $h^*(\mathbf{x})$ is the path with the largest total weights from $\mathbf{x}$ to a goal state. Let the path with the largest

total weights from $\mathbf{x}$ to a goal state be $\mathbf{x}_0 = \mathbf{x}, \mathbf{x}_1, \ldots, \mathbf{x}_n = \tilde{\mathbf{x}}$, and let $\mathbf{x}_{i-1} \oplus a_i = \mathbf{x}_i$ for $i = 1, 2, \ldots, n$, the total weights are

$$\begin{aligned} h^*(\mathbf{x}) &= \sum_{i=1}^{n} w(\mathbf{x}_{i-1}, \mathbf{x}_i) \\ &= \sum_{i=1}^{n} \Big[p(\mathbf{x}_i) - p(\mathbf{x}_{i-1})\Big]R_c - \sum_{i=1}^{n} \pi(a_i) \\ &= \Big[p(\tilde{\mathbf{x}}) - p(\mathbf{x})\Big]R_c - \sum_{i=1}^{n} \pi(a_i). \end{aligned} \tag{10}$$

As everyone knows, for positive edge state space graph, when the heuristic is *admissible* ($h(\mathbf{x}^*) = 0$ and $h(\mathbf{x}) \geqslant h^*(\mathbf{x})$) and *consistent* (a monotonicity property) at any state $\mathbf{x}$, a maximal state space search will find an optimal solution [16] when $h(\mathbf{x}) = 0$. This special case is known as **A* search**. For example, a trivial admissible heuristic is $h^0(\mathbf{x}) = R_c$, in which case the algorithm is essentially the Dijkstra's algorithm. Unfortunately, since the edge weight $w(\mathbf{x}_{i-1}, \mathbf{x}_i)$ might be a negative number in the state space graph for OAP problems, Algorithm 1 cannot guarantee to find the optimal solution when $h(\mathbf{x}) = 0$. It might have to search all states to guarantee finding the optimal solution. In our experiments, it usually causes the search timeout. Thus, we propose a sub-optimal algorithm which can remarkably reduce the search time while obtaining a sub-optimal solution at the same time.

Algorithm 2 shows the sub-optimal state space search algorithm. Similar to Algorithm 1, it maintains two data structures, a min heap (which takes ⟨value, key⟩ pairs) and a closed list, and performs the following main steps:

1) Initialize the g^*, δ^*, and $plan^*$. Add the pair $\langle \mathbf{x}^I, h(\mathbf{x}^I)\rangle$ to the min heap (lines 1–4).
2) Pop the state $\mathbf{x}$ from the heap with the smallest $h(\mathbf{x})$ (line 7).
3) If $\mathbf{x}$ is a goal state and $g(\mathbf{x}) > g^*$, update g^*, δ^*, and the best plan $plan^*$ (lines 8–11).
4) If $\mathbf{x}$ satisfies one of the termination conditions ($h(\mathbf{x}) == 0$ or $p(\mathbf{x}) \geqslant p^u$ or $|ClosedList| - \delta^* > \Delta^u$), stop the search and return the best plan $plan^*$ ever found (lines 13,14).
5) Add $\mathbf{x}$ to the closed list and for each edge $(\mathbf{x}, \mathbf{x}') \in E$, add $\langle \mathbf{x}', f(\mathbf{x}')\rangle$ to the min heap if $\mathbf{x}$ is not in the closed list (lines 16–20).
6) Repeat from Step 2.

Note that p^u is the upper bound of the prediction value. δ^* records the number of searched states when finding a better plan. If the search has not found a better plan for a long time ($|ClosedList| - \delta^* > \Delta^u$), to get a good balance of efficiency and plan quality, we stop the search immediately.

Algorithm 2 Sub-optimal state space search

```
Input: G = (F, E), g, h
 1: g* ← −∞
 2: δ* ← 0
 3: plan* ← ∅
 4: MinHeap.push(⟨x^I, h(x^I)⟩)
 5: ClosedList ← {}
 6: while MinHeap is not empty do
 7:   x ← MinHeap.pop()  // x has the minimum h(x)
 8:   if p(x) ⩾ z and g(x) > g* then
 9:     g* ← g(x)
10:     δ* ← |ClosedList|
11:     plan* ← path(x^I, x)
12:   end if
13:   if h(x) == 0 or p(x) ⩾ p^u or |ClosedList| − δ* > Δ^u then
14:     return plan*
15:   end if
16:   if x ∉ ClosedList then
17:     ClosedList=ClosedList ∪{x}
18:     for each (x, x′) ∈ E do
19:       compute h(x′)
20:       MinHeap.push(⟨x′, h(x′)⟩)
21:     end for
22:   end if
23: end while
24: return plan*
```

Heuristic $h^1(\mathbf{x})$ We propose an admissible and consistent heuristic function to be used as $h(\mathbf{x})$ in Algorithm 2:

$$h^1(\mathbf{x}) = \Big[1 - p(\mathbf{x})\Big]R_c - \pi^*(\mathbf{x}), \tag{11}$$

where $\pi^*(\mathbf{x})$ is the minimum cost of any edge $(\mathbf{x}, \mathbf{x}')$. Let $A_{\mathbf{x}}$ be the set of actions on the edge from $\mathbf{x}$ to $\mathbf{x}'$. We have $\pi^*(\mathbf{x}) = \min_{a \in A_{\mathbf{x}}} \pi(a)$.

Theorem 3 Heuristic $h^1(\mathbf{x})$ is admissible and consistent.

Proof Let the path with the largest total weights from $\mathbf{x}$ to a goal state be $\mathbf{x}_0 = \mathbf{x}, \mathbf{x}_1, \ldots, \mathbf{x}_m = \tilde{\mathbf{x}}$, and let $\mathbf{x}_{i-1} \oplus a_i = \mathbf{x}_i$ for $i = 1, 2, \ldots, m$, the total weights is

$$\begin{aligned} h^*(\mathbf{x}) &= \sum_{i=1}^{m} w(\mathbf{x}_{i-1}, \mathbf{x}_i) \\ &= \sum_{i=1}^{m} \Big[p(\mathbf{x}_i) - p(\mathbf{x}_{i-1})\Big]R_c - \sum_{i=1}^{m} \pi(a_i) \\ &= \Big[p(\tilde{\mathbf{x}}) - p(\mathbf{x})\Big]R_c - \sum_{i=1}^{m} \pi(a_i) \\ &\leqslant \Big[p(1 - p(\mathbf{x})\Big]R_c - \pi^*(\mathbf{x}), \end{aligned} \tag{12}$$

which shows that $h^1(\mathbf{x})$ is admissible. Also, for any edge

$(\mathbf{x}_{i-1}, \mathbf{x}_i) \in E$, let $\mathbf{x}_{i-1} \oplus a = \mathbf{x}_i$, we have

$$\begin{aligned} w(\mathbf{x}_{i-1}, \mathbf{x}_i) + h^1(\mathbf{x}_i) &= \left[p(\mathbf{x}_i) - p(\mathbf{x}_{i-1})\right]R_c - \pi(a) \\ &\quad + \left[1 - p(\mathbf{x}_i)\right]R_c - \pi^*(\mathbf{x}_i) \\ &= \left[1 - p(\mathbf{x}_{i-1})\right]R_c - \pi(a) - \pi^*(\mathbf{x}_i) \\ &\leqslant \left[1 - p(\mathbf{x}_{i-1})\right]R_c - \pi^*(\mathbf{x}_{i-1}) \\ &= h^1(\mathbf{x}_{i-1}). \end{aligned} \tag{13}$$

The triangle inequality $w(\mathbf{x}_{i-1}, \mathbf{x}_i) + h^1(\mathbf{x}_i) \leqslant h^1(\mathbf{x}_{i-1})$ shows that $h^1(\mathbf{x})$ is consistent [16]. □

Compare to Algorithm 1, Algorithm 2 has two main differences: 1) It does not have to search all states since it will stop the search once a state $\mathbf{x}$ satisfies the termination condition (lines 13,14); 2) Since a state with the smaller h value is more close to the goal, it uses a Minheap to store the candidate states by the ascending order of the heuristic value (h) and pops the state with the minimum h value in each iteration.

5 Experimental results

5.1 Experimental setup

We evaluate the proposed method on a real data set obtained from a credit card company via private communication and three datasets from the UCI Repository [4, 17, 18]. The information of these datasets are listed in Table 1. M is the number of attributes, M_s is the number of soft attributes, T is the training time for random forests in seconds, and $|O|$ is the number of actions. We randomly split each dataset into training and testing sets with a 7:3 ratio. A random forest is built on the training set using the Random Trees library in OpenCV 2.4.9.

Table 1 The detailed information of the dataset from a credit company and other three datasets from the UCI repository

Dataset	M	Sample size		Recognition rate/%		T	M_s	$\|O\|$
		Training	Testing	Training	Testing			
Adult	14	21 113	9 049	86.68	86.52	4.53	11	2 368
Bank	18	28 831	12 357	91.93	88.52	6.61	17	580
Credit	35	624 281	267 549	59.73	59.70	251.08	17	762
German	20	700	300	95.00	74.00	0.27	16	206

For each domain, we randomly choose 30 records from the testing set that are labeled in a "negative" status (i.e., "<=50K" in Adult, "no" in Bank, labeled "negative" in Credit, "bad" in German) to form OAP problem instances. We randomly generate actions to change its soft attributes. Each action is assigned a random cost in [10, 20]. R_c, the total profit of a state s in the desired status, is set to 1 000, and z is set to 0.5.

• Adult This dataset is extracted from the 1994 Census database [18]. The task is to determine whether a person makes over 50K a year. This data consists 30 162 records, and 14 attributes including 11 soft attributes. We use the 11 soft attributes to generate 2 368 actions.

• Bank This data is from direct marketing campaigns of a Portuguese bank [4]. The marketing campaigns are based on phone calls. The classification goal is to predict if the client will subscribe ("yes" or "no") a term deposit. This data consists 41 188 records, and 18 attributes including 17 soft ones. We use the 17 soft attributes to generate 580 actions.

• Credit This is a real dataset from a credit card company in the US. It has 891 831 records for customers, who have two status as bringing "positive" or "negative" profits to the company. Out of the 35 attributes, 17 can be changed values with reasonable costs. We use the 17 soft attributes to generate 762 actions.

• German This dataset classifies people as having "good" or "bad" credit scores. It consists of 1 000 records, each has 20 attributes in total and 16 soft attributes. We generate 206 actions.

We run all experiments on a workstation with an Intel Xeon 2.50GHz processor and 16GB memory. For each solver on each instance, the time limit is set to 600 seconds. If a solver does not finish in 600 seconds, we record the best solution found in terms of net profit and the total search time (600 seconds).

5.2 Adjust the tradeoff between net-profit and search time

In sub-optimal heuristic search with heuristic h^1, denoted as $NS(h^1)$, the main parameters controlling the tradeoff between net-profit and search time are p^u and Δ^u (See line 13 in Algorithm 2). Considering the search time cost, finding the good p^u and Δ^u values is very important to get a good balance between net-profit and efficiency, especially in some time-sensitive domains. Note that the best values of p^u and Δ^u are highly domain dependent.

Figure 2 shows the results of the average net-profit and search time of $NS(h^1)$ with different upper bounds p^u for each dataset. In these tests, Δ^u is set to ∞. From Fig. 2(a), we can see that the net-profit linearly increases with p^u when p^u is smaller than 0.9. However, the net-profit has a very small increase when p^u is larger than 0.9. From Fig. 2(b), we can see that the search time increases very fast when p^u is larger than a certain value (0.9 in Adult, 0.8 in Bank and Credit, and 0.95 in German). The reason is that the states with a high prediction value p are pretty rare, it will spend much more time

to find such a state. Thus, to gain the best tradeoff between net-profit and search time, the best p^u for the four datasets can not be neither too small nor too large.

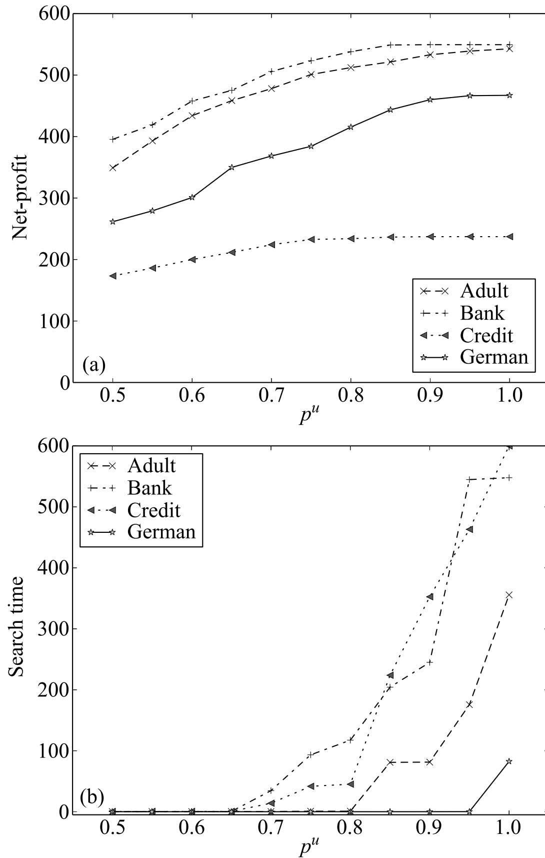


Fig. 2 The average (a) net-profit and (b) search time of $NS(h^1)$ with different upper bound p^u for each dataset ($\Delta^u = \infty$)

Figure 3 shows the results of the average net-profit and search time of $NS(h^1)$ with different upper bounds Δ^u for each dataset. In these tests, p^u is set to 1.0. From Fig. 3(a), we can see that the net-profit increases with Δ^u under a certain value (10^4 in Adult and Bank, 10^3 in Credit and German) and after that it almost keeps unchanged. Figure 3(b) shows that the search time linearly increases with Δ^u when Δ^u is larger than 10^4. We also present the net-profit and the number of searched states of all solutions found by $NS(h^1)$ for each dataset in Fig. 4. It shows that most of the solutions would be found under a certain number of searched states (10^5 in Adult, 10^4 in Bank, 10^3 in Credit and German). Thus, $NS(h^1)$ does not have to search a large number of states (larger than 10^6) to find a sub-optimal solution.

5.3 Viability of our methods

To show the viability of our algorithms, five methods are tested and compared in our experiments. We test the optimal state space search Algorithm 1 using heuristic function $h^0(\mathbf{s})$, denoted as $OS(h^0)$, and the sub-optimal state space search Algorithm 2 using the admissible heuristic $h^1(\mathbf{s})$ and upper bounds $p^u = 0.9$ and $\Delta^u = 10^5$, denoted as $NS(h^1)$. For comparison, we also implement three solvers as follows: 1) A breadth-first search using Algorithm 1 with an evaluation function $f^b = \sum_{i=1}^{n} \pi(a_i)$, denoted as $OS(f^b)$; 2) A random search using Algorithm 2 with a random heuristic h^r which is a random value from $[1, R_c]$, denoted as $NS(h^r)$; 3) An iterative greedy algorithm which chooses one action in each iteration that maximizes the net-profit $F(A)$ in Eq. (5). It keeps iterating until there is no more variables to change. We denote this algorithm as the Greedy algorithm.

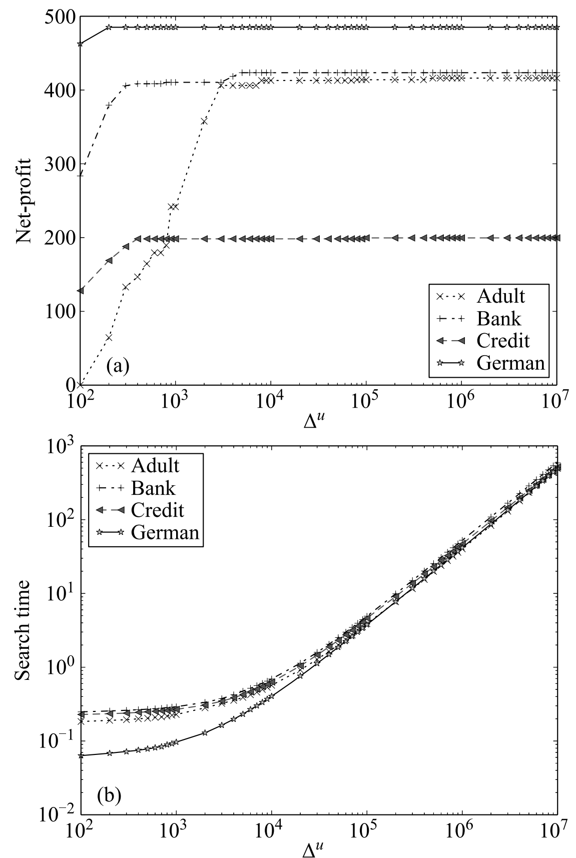


Fig. 3 The average (a) net-profit and (b) search time of $NS(h^1)$ with different upper bound Δ^u for each dataset ($p^u = 1.0$)

We first show the summarized results of all methods on all instances in Table 2. 30 random cases are tested for each dataset. T is the average search time in seconds, and F(A) is the average net profit. B/EQ(f^b) is the number of cases that the $OS(h^0)$ algorithm gets better/same F(A) as the $OS(f^b)$ method out of the 30 runs. B/EQ(f^b), B/EQ(h^0), B/EQ(h^r), and B/EQ(G) are the number of cases that the $NS(h^1)$ algo-

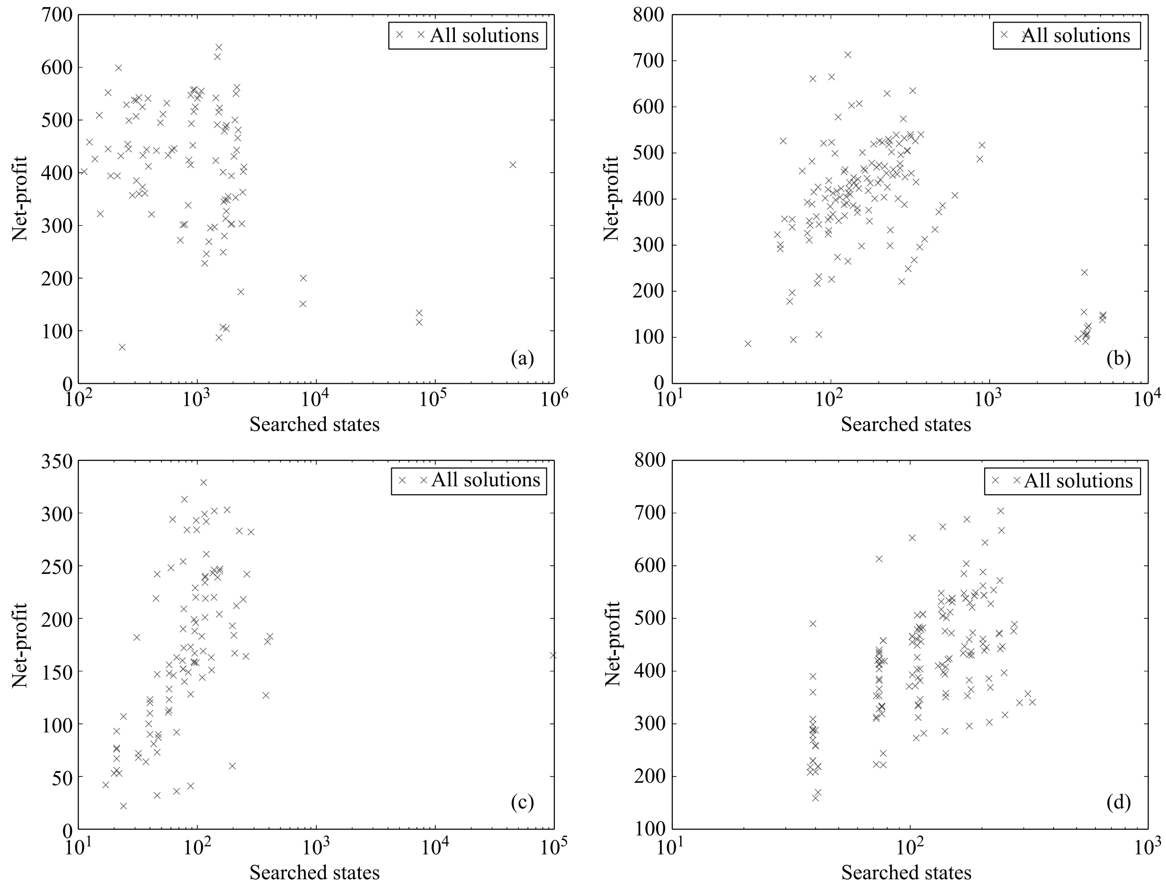


Fig. 4 The number of searched states versus the net-profit of all solutions found by $NS(h^1)$ for each dataset. (a) Adult; (b) Bank; (c) Credit; (d) German

Table 2 Summarization of experimental results of five algorithms on four datasets

Name	Optimal					Sub-optimal									
	$OS(f^b)$		$OS(h^0)$			$NS(h^r)$		Greedy		$NS(h^1)$					
	T	F(A)	T	F(A)	B/EQ(f^b)	T	F(A)	T	F(A)	T	F(A)	B/EQ(f^b)	B/EQ(h^0)	B/EQ(h^r)	B/EQ(G)
Adult	600.01	318.23	600.01	**606.43**	26/1	600.01	–96.13	0.21	482.97	3.18	**542.73**	25/2	1/7	30/0	16/0
Bank	600.01	381.47	600.01	**566.03**	30/0	600.01	–112.63	0.25	322.57	4.20	**549.47**	29/0	0/10	30/0	28/0
Credit	600.01	243.70	600.01	**261.23**	22/4	600.01	24.17	0.44	217.03	4.92	**237.20**	5/4	0/4	30/0	20/0
German	580.01	466.30	600.01	**469.60**	13/11	563.26	229.30	0.05	361.43	1.72	**467.07**	12/10	0/21	30/0	30/0

rithm gets better/same F(A) as the $OS(f^b)$, $OS(h^0)$, $NS(h^r)$, and Greedy, respectively. As shown in Table 2, the $OS(h^0)$ algorithm finds better plans than $OS(f^b)$. Although both are optimal in theory, since we set an time limit of 600 seconds and they often cannot finish the search within the time limit. As a result, we count the optimal solution found within the time limit. We can see that $OS(h^0)$ is much better than $OS(f^b)$ in terms of quality, due to its stronger evaluation function guidance.

Comparing the $NS(h^1)$ with the optimal algorithms $OS(f^b)$ and $OS(h^0)$, we can see that $OS(h^0)$ finds the best solutions in most of the instances within the given time limit. Even though $NS(h^1)$ is a sub-optimal algorithm, it finds 71 better and 16 equal solutions compared with $OS(f^b)$ and 1 better and 42 equal solutions compared with $OS(h^0)$ out of total 120 instances while spending much less search time.

Comparing the $NS(h^1)$ with the random search $NS(h^r)$ and Greedy algorithms, we can see that the Greedy algorithm is very fast, taking less than one second on average to yield a solution. However, $NS(h^1)$ finds better solutions in most of the instances within the given time limit. More precisely, it finds 120 better solutions than $NS(h^r)$ and 94 better solutions than Greedy out of total 120 instances. In many practical applications such as marketing and clinical decision-making, users are often willing to pay a few minutes of time in order to maximize their potential net benefits. Moreover, since actionability extraction can be viewed as an individualized feature selection process for selecting the most sensitive and relevant

features for a given sample, it is important to find the optimal or sub-optimal solution.

Detailed results on all instances are shown in Fig. 5. Here, we plot the search time (x-axis) versus the net profit (y-axis) for each run. From the plots, we can clearly see that the Greedy algorithm is very fast, but does not have very good solution quality. The optimal $OS\,(h^0)$ algorithm has by far the best solution quality. The speed and solution quality of $NS\,(h^1)$ are in between the Greedy and $OS\,(h^0)$ methods, which achieves a good tradeoff between solution quality and search time. $OS\,(f^b)$ and $NS\,(h^r)$ are not competitive in terms of solution quality and search time.

5.4 Comparison with ILP method

For comparison, we also consider the integer linear programming (ILP) method [19], one of the state-of-the-art algorithms for solving the OAP problem. We test our sub-optimal algorithm ($NS\,(h^1)$) and ILP method on nine benchmark datasets from the UCI repository[2)] and the LibSVM website[3)] used in ILP's original experiments. The information of the datasets is listed in Table 3. N, D, and C are the number of instances, features, and classes, respectively. To compare with ILP, for each dataset, we randomly sample 30 instances from testing set and generate 30 problems with the same parameter settings. Specifically, we choose a weighted Euclidean distance as the action cost function,

$$\pi(x_{i-1}, x_i) = \beta_i (x_i - x_{i-1})^2, \tag{14}$$

where β_i is the cost weight on variable i. β_i is randomly generated in the range between 1 and 100. Note that since ILP aims to minimize the loss function

$$\ell(\tilde{\mathbf{x}}, x^I) = \sum_{i=1}^{D} \beta_i (\tilde{\mathbf{x}}_i - x_i^I)^2, \tag{15}$$

for each dataset, we solve the 30 generated problems and report the results of the average search time (Time) and total action costs of the solutions (Cost). EQ(ILP) is the number of optimal solution found by $NS\,(h^1)$ which has equal solution cost to ILP and #instances is the number of instances in each domain.

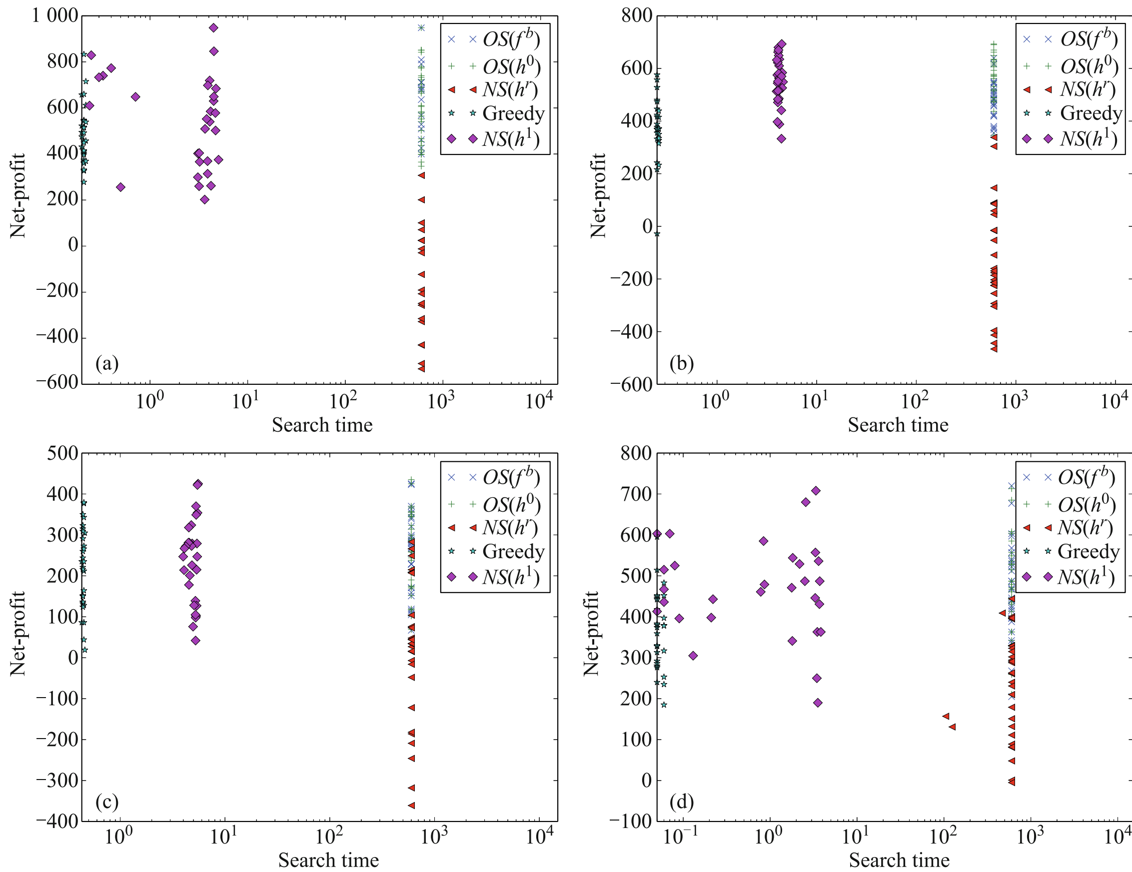


Fig. 5 Search time in seconds versus the net-profit of different methods on all test cases from the four datasets. (a) Adult; (b) Bank; (c) Credit; (d) German

2) https://archive.ics.uci.edu/ml/datasets.html

3) http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

Table 3 Comparison results of our sub-optimal search and ILP algorithms on nine datasets

Dataset	N	D	C	ILP		$NS(h^1)$				
				Time/s	Cost	Time/s	Cost	$\frac{\text{Time}(NS(h^1))}{\text{Time}(ILP)}$ (%)	$\frac{\text{Cost}(ILP)}{\text{Cost}(NS(h^1))}$	EQ(ILP)/#instances
A1a	32 561	123	2	7.55	70.9	4.05	72.32	53.64	1.02	16/30
Australian	690	14	2	108.01	0.44	1.88	0.82	1.74	1.86	23/30
Breast cancer	683	10	2	31.04	10.09	1.45	10.09	4.67	1.00	30/30
Dna	2 000	180	3	35.47	12.2	4.49	15.68	12.65	1.28	13/30
Heart	270	13	2	5.77	3.19	2.31	8.01	40.03	2.51	20/30
Ionosphere	351	34	2	48.84	18.68	2.87	25.37	5.87	1.35	3/30
Liver disorders	345	6	2	31.62	0.15	0.17	0.15	0.53	1.00	30/30
Mushrooms	8 124	112	2	3.87	22.93	2.69	23.59	69.50	1.02	27/30
Vowel	990	10	11	68.73	1.15	1.97	2.02	2.86	1.75	17/30
Total	46 014	502	28	340.90	139.73	21.88	158.05	6.41	1.13	179/270

Table 3 shows a comprehensive comparison in terms of the average search time and the solution quality measured by the total action costs. From Table 3, we can see that $NS(h^1)$ spends much less time to find a sub-optimal solution than the ILP (21.88 seconds, 6.41% of ILP's 340.90 seconds), especially in datasets australian (1.74%), breast cancer (4.67%), dna (12.65%), ionosphere (5.87%), liver disorders (0.53%), and vowel (2.86%). Moreover, $NS(h^1)$ finds 179 optimal solutions in total 270 instances and the total average solution costs (158.05) is a little larger than the ILP (139.73). Thus, our sub-optimal algorithm achieves a good balance between the search time and solution quality compared with ILP.

6 Related work

Actionable knowledge discovery has been studied in management and marketing science, where stochastic models are used to find specific rules of the response behavior of customers [20,21]. On the other hand, research on this subject is still very limited in the machine learning community. Some earlier works focused on the development of ranking mechanisms with business interests. Hilderman et al. proposed a two-step process for ranking the interestingness of discovered patterns [22]. Cao et al. proposed a two-way framework to measure knowledge actionability which not only considers technical interestingness but also domain-specific expectations [23].

Besides the ranking mechanisms, post-analysis techniques have also been studied. Liu et al. introduced an actionable knowledge discovery algorithm which could prune and summarize the learnt rules by considering similarity [24,25]. In order to expand the ability of handling different problems and applications, Cao et al. proposed domain-driven data mining, a paradigm shift from a research-centered discipline to a practical tool for actionable knowledge [26]. More specifically, meta-synthesize ubiquitous intelligence and several types of other frameworks have been involved into the mining process [27]. Techniques have also been proposed to postprocess decision tree and additive tree models to extract actionable knowledge [13,19,28]. Yang et al. used a greedy strategy to find optimal strategies on decision trees [13,28]. Cui et al. found the actions to change sample membership on an ensemble of trees using integer linear programming [19]. However, it did not consider expected net profit but only considered actions that change one attribute each time. Compared to some previous work [13,19,28], the problem we studied is much more challenging. For instance, in the dataset of "Bank Telemarketing" [4], a "contact" action may change attributes "campaign" (number of contacts performed during this campaign and for this client), "pdays" (number of days that passed by after the client was last contacted from a previous campaign), and "previous" (number of contacts performed before this campaign and for this client) at the same time. To complicate matters, one action could involve multiple attributes and one attribute changing may associate with multiple actions. In a word, these approaches can not handle the complicated situations [13,19,28]. In this paper, the proposed state space search based planning framework is capable of tackling this problem.

7 Conclusions

In this paper, we studied automatic extraction of actionable knowledge from additive tree models (ATMs), one of the most widely used and best off-the-shelf classifiers. We started from a mathematical formulation of ATM and the optimal actionable plan (OAP) problem, which is used to find the set of actions that can change an input instance's status to a desired one with the maximum net profit. Solving the OAP problem not only provides an actionable plan, but also helps

rank feature importance *individualized* for each input sample. Most feature selection algorithms are based on a given training dataset and classifier, and are not customized for an individual instance. In contrast, the output from OAP can be viewed as a result of individualized feature selection, as it identifies the few features that can most efficiently change the prediction output of a particular instance. Such individualized feature selection may find many applications, such as personalized healthcare and targeted marketing.

We then proposed a state space graph formulation to model the OAP problem as a well-studied combinatorial optimization problem that can be solved by graph search. We further introduced an optimal state space search to find the optimal solution and a sub-optimal state space search which is more effective than the optimal algorithm. We also introduced a heuristic function which can largely improve the efficiency of the sub-optimal algorithm. Extensive experimental results on real-world credit and banking data showed that the proposed sub-optimal method can efficiently and robustly solve the OAP problems. In a word, the optimal search significantly outperforms other baseline methods in terms of the solution quality with the expense of a large number of search time. Compared with the optimal algorithm, the sub-optimal search takes a good balance between the search time and plan quality. Given its efficiency and robustness, we believe that the proposed framework will become a popular method for actionable knowledge extraction on a large scope of real-world applications.

In our current work, we have not considered the scalability of the ATM models. When learning on a very large data set, models like random forest can be extremely large and deep. It may increase the time of computing the heuristic function h^1 and slow down the sub-optimal search. One interesting future work is to simply the ATM and use a smaller model to extract plans [29].

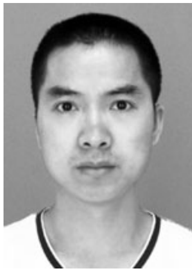
Acknowledgements This work was supported in part by China Postdoctoral Science Foundation (2013M531527), the Fundamental Research Funds for the Central Universities (0110000037), the National Natural Science Foundation of China (Grant Nos. 61502412, 61033009, and 61175057), Natural Science Foundation of the Jiangsu Province (BK20150459), Natural Science Foundation of the Jiangsu Higher Education Institutions (15KJB520036), National Science Foundation, United States (IIS-0534699, IIS-0713109, CNS-1017701), and a Microsoft Research New Faculty Fellowship.

References

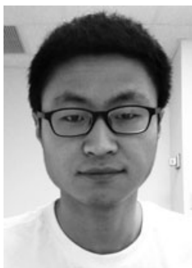
1. Mao Y, Chen W L, Chen Y X, Lu C Y, Kollef M, Bailey T. An integrated data mining approach to real-time clinical monitoring and deterioration warning. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2012, 1140–1148
2. Bailey T C, Chen Y X, Mao Y, Lu C Y, Hackmann G, Micek S T, Heard K M, Faulkner K M, Kollef M H. A trial of a real-time alert for clinical deterioration in patients hospitalized on general medical wards. Journal of Hospital Medicine, 2013, 8(5): 236–242
3. Cortez P, Embrechts M J. Using sensitivity analysis and visualization techniques to open black box data mining models. Information Sciences, 2013, 225: 1–17
4. Moro S, Cortez P, Rita P. A data-driven approach to predict the success of bank telemarketing. Decision Support Systems, 2014, 62: 22–31
5. Szegedy C, Zaremba W, Sutskever I, Bruna J, Erhan D, Goodfellow I, Fergus R. Intriguing properties of neural networks. 2013, arXiv preprint arXiv:1312.6199
6. Friedman J, Hastie T, Tibshirani R. The elements of statistical learning. Volume 1. Springer Series in Statistics Springer, 2001
7. Shotton J, Sharp T, Kipman A, Fitzgibbon A, Finocchio M, Blake A, Cook M, Moore R. Real-time human pose recognition in parts from single depth images. Communications of the ACM, 2013, 56(1): 116–124
8. Viola P, Jones M J. Robust real-time face detection. International Journal of Computer Vision, 2004, 57(2): 137–154
9. Mohan A, Chen Z, Weinberger K Q. Web-search ranking with initialized gradient boosted regression trees. Journal of Machine Learning Research, Workshop and Conference Proceedings, 2011, 14: 77–89
10. Breiman L. Random forests. Machine Learning, 2001, 45(1): 5–32
11. Freund Y, Schapire R E. A decision-theoretic generalization of online learning and an application to boosting. Journal of Computer and System Sciences, 1997, 55(1): 119–139
12. Friedman J H. Greedy function approximation: a gradient boosting machine. The Annals of Statistics, 2001, 29: 1189–1232
13. Yang Q, Yin J, Ling C X, Chen T. Postprocessing decision trees to extract actionable knowledge. In: Proceedings of the 3rd IEEE International Conference on Data Mining. 2003, 685–688
14. Manindra A, Thomas T. Satisfiability Problems. Technical Report. 2000
15. Cai S W. Balance between complexity and quality: local search for minimum vertex cover in massive graphs. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence. 2015, 747–753
16. Russel S, Norvig P. Artificial Intelligence: A Modern Approach. 2nd Ed. Upper Saddle River: Prentice-Hall, 2003
17. Bache K, Lichman M. UCI Machine Learning Repository. Technical Report. 2013
18. Kohavi R. Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. In: Proceedings of the International Conference on Knowledge Discovery and Data Mining. 1996, 202–207
19. Cui Z C, Chen W L, He Y J, Chen Y X. Optimal action extraction for random forests and boosted trees. In: Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2015, 179–188
20. DeSarbo W S, Ramaswamy V. Crisp: customer response based iterative segmentation procedures for response modeling in direct marketing. Journal of Direct Marketing, 1994, 8(3): 7–20
21. Levin N, Zahavi J. Segmentation analysis with managerial judgment. Journal of Direct Marketing, 1996, 10(3): 28–47
22. Hilderman R J, Hamilton H J. Applying objective interestingness mea-

sures in data mining systems. In: Proceedings of the European Symposium on Principles of Data Mining and Knowledge Discovery. 2000, 432–439

23. Cao L B, Luo D, Zhang C Q. Knowledge actionability: satisfying technical and business interestingness. International Journal of Business Intelligence and Data Mining, 2007, 2(4): 496–514
24. Liu B, Hsu W. Post-analysis of learned rules. In: Proceedings of the National Conference on Artificial Intelligence. 1996, 828–834
25. Liu B, Hsu W, Ma Y. Pruning and summarizing the discovered associations. In: Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 1999, 125–134
26. Cao L B, Zhang C Q, Yang Q, Bell D, Vlachos M, Taneri B, Keogh E, Yu P S, Zhong N, Ashrafi M Z, Taniar D, Dubossarsky E, Graco W. Domain-driven, actionable knowledge discovery. IEEE Intelligent Systems, 2007, 22(4): 78–88
27. Cao L B, Zhao Y C, Zhang H F, Luo D, Zhang C Q, Park E K. Flexible frameworks for actionable knowledge discovery. IEEE Transactions on Knowledge and Data Engineering, 2010, 22(9): 1299–1312
28. Yang Q, Yin J, Ling C, Pan R. Extracting actionable knowledge from decision trees. IEEE Transactions on Knowledge and Data Engineering, 2007, 19(1): 43–56
29. Zhou Z H, Jiang Y. Nec4. 5: neural ensemble based c4. 5. IEEE Transactions on Knowledge and Data Engineering, 2004, 16(6): 770–773



Qiang Lu is currently an assistant professor in the College of Information Engineering at Yangzhou University, China. He received the BE and PhD degrees from the School of Computer Science and Technology, University of Science and Technology of China (USTC), China in 2007 and 2012, respectively. He received the National Natural Science Foundations of China and Jiangsu Province, Joint PhD Training Scholarship from the China Scholarship Council, and the China Postdoctoral Science Foundation. He has published more than ten papers in journals and conference proceedings, including the ACM TIST, IEEE TSC, EAAI, AAAI'13, ICAPS'11, CloudCom'11, and IPC'11. He is a member of the ACM and the CCF. His research interests include data mining, automated planning and scheduling, parallel and distributed computing, and cloud computing.



Zhicheng Cui is now a second year PhD candidate in the Department of Computer Science and Engineering at Washington University in St Louis (WUSTL), USA, supervised by Prof. Yixin Chen. Prior to joining WUSTL, he received his BE in computer science from University of Science and Technology of China (USTC), China in 2014. His research interests are data mining and machine learning, in the area of large scale time series analysis.

Yixin Chen is an associate professor of computer science at the Washington University in St. Louis, USA. He received the PhD degree in computer science from the University of Illinois at Urbana-Champaign, USA in 2005. His work on planning has won First-Class Prizes in the International Planning Competitions (2004 and 2006). He has won the Best Paper Award in AAAI (2010) and ICTAI (2005), and Best Paper nomination at KDD (2009). He has received an Early Career Principal Investigator Award from the Department of Energy (2006) and a Microsoft Research New Faculty Fellowship (2007). Dr. Chen is a senior member of IEEE. He serves as an associate editor on the IEEE Transactions on Knowledge and Data Engineering, and ACM Transactions on Intelligent Systems and Technology. His research interests include nonlinear optimization, constrained search, planning and scheduling, data mining, and data warehousing.

Xiaoping Chen is a full professor with the School of Computer Science and Technology and the Directors of the Robotics Lab and the Center for Artificial Intelligence Research at University of Science and Technology of China (USTC), China. He received his PhD in computer science from USTC in 1997. He established and has led the USTC Robotics Lab and its robot team, WrightEagle, which won 7 champions and 11 runners-up in RoboCup world championships. Prof. Chen found and has led the KeJia Project, which won the Best Autonomous Robotics Award at the IJCAI 2013 Video Competition and the First Prize for General Robot Skills at the IJCAI 2013 Robot Competition. He published about 130 papers, including some appeared in AIJ, IJCAI, AAAI, AAMAS, KR, UAI, ICLP, ICAPS, IJHR, IROS, and JHRI. In 2010, he won the USTC President Award for Research Excellence, which has been the topmost research award at USTC with 1 or 2 scientists being presented annually. Prof. Chen has been working in the fields of artificial intelligence and intelligent service robotics. His current research interests include problem-solving with open knowledge, situated NLP, semantic perception, and automated planning.