Learning to Question: Leveraging User Preferences for Shopping Advice

Mahashweta Das[‡]
Computer Science and Engineering Department
University of Texas at Arlington
mahashweta.das@mavs.uta.edu

Gianmarco De Francisci Morales
Yahoo! Research
Barcelona, Spain
gdfm@yahoo-inc.com

Aristides Gionis[‡]
Aalto University and HIIT
Espoo, Finland
aristides.gionis@aalto.fi

Ingmar Weber[‡]
Qatar Computing Research Institute
Doha, Qatar
ingmarweber@acm.org

ABSTRACT

We present ShoppingAdvisor, a novel recommender system that helps users in shopping for technical products. ShoppingAdvisor leverages both user preferences and technical product attributes in order to generate its suggestions. The system elicits user preferences via a tree-shaped flowchart, where each node is a question to the user. At each node, ShoppingAdvisor suggests a ranking of products matching the preferences of the user, and that gets progressively refined along the path from the tree's root to one of its leafs.

In this paper we show (*i*) how to learn the structure of the tree, i.e., which questions to ask at each node, and (*ii*) how to produce a suitable ranking at each node. First, we adapt the classical top-down strategy for building decision trees in order to find the best user attribute to ask at each node. Differently from decision trees, ShoppingAdvisor partitions the user space rather than the product space. Second, we show how to employ a learning-to-rank approach in order to learn, for each node of the tree, a ranking of products appropriate to the users who reach that node.

We experiment with two real-world datasets for cars and cameras, and a synthetic one. We use mean reciprocal rank to evaluate ShoppingAdvisor, and show how the performance increases by more than 50% along the path from root to leaf. We also show how collaborative recommendation algorithms such as k-nearest neighbor benefits from feature selection done by the ShoppingAdvisor tree. Our experiments show that ShoppingAdvisor produces good quality interpretable recommendations, while requiring less input from users and being able to handle the cold-start problem.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

recommendation, learning, ranking, collaborative content

1. INTRODUCTION

When shopping for a new laptop in the traditional way, customers would walk into a shop and rely on the experience of a shopping assistant to help them select the best laptop for their needs. A good shopping assistant would ask questions intelligible to non-expert users, for instance, "*Do you intend to use the laptop for playing modern computer games?*", while mapping the answers to technical product specifications, such as, "*The customer will need at least 4* GB *of* RAM." E-commerce has disrupted this custom in two ways. First, customers shop online, from their homes, without any human interaction involved. Second, catalogs of online shops are so big and with so many continuous updates that no human, however expert, can effectively comprehend the space of available products. As a consequence, the customers are left without any guidance to understand their *needs* and map them to a product.

In this paper we propose a system that addresses this need. Our system, called ShoppingAdvisor, draws inspiration from a recent marketing strategy called *"Which product should I buy?"* flowchart. Each box in these flowcharts asks the prospective shopper a question, and the sequence of answers leads the shopper to the suggested shopping option. "*Which product should I buy?*" flowcharts present two problems. First, designing such flowcharts manually is time consuming. Second, they rely on questions about technical attributes, which an average shopper might not understand. We address both problems by automatically designing flowcharts that help shoppers select the best product for their needs. An important design principle of our approach is that we distinguish between two types of features: (*i*) features in a "user space," which contain user information such as demographics, life-style preferences, and interests; and (*ii*) features in a "product space," which contain technical information of the products, e.g., the CPU speed of a laptop.

ShoppingAdvisor generates a tree-shaped flowchart, in which the internal nodes of the tree contain questions addressed to the users. These questions involve only attributes

[‡] Majority of work done while at Yahoo! Research.

KDD'13, August 11–14, 2013, Chicago, Illinois, USA.
Copyright 2013 ACM 978-1-4503-2174-7/13/08 ...$15.00.

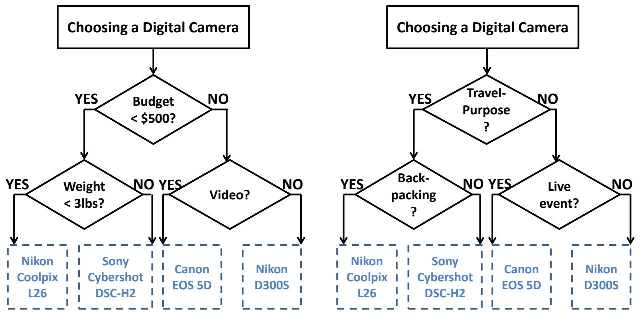

Figure 1: Example of "Which camera should I buy?" flowchart (left) and its equivalent non-technical ShoppingAdvisor flowchart (right).

from the user space, that non-expert users can understand easily. For instance, "*Are you a student?*" or "*Would you be storing videos in your laptop?*". A potential shopper, starts from the root of the ShoppingAdvisor tree, answers questions, follows the control flow, and descends towards a leaf of the ShoppingAdvisor tree, where they find a ranked list of product recommendations. In fact, a user can stop answering at any level in the tree as a ranking is available at each node, not only at leafs.

At a high-level, the ShoppingAdvisor tree resembles a *decision-tree*, and our method for learning its structure resembles decision-tree learning algorithms. However, there are important differences. First, given a set of users with similar features, our system induces a ranking on *all* the existing products, based on the attributes of the products. Thus, unlike traditional decision trees in which each node outputs a class label, we learn trees in which each node outputs a ranking of products. Second, unlike other tree-based methods, in our model the splitting and ranking domains are different. Indeed, the ShoppingAdvisor tree partitions the users on the basis of user attributes, so that similar users will end up in the same nodes, under the assumption that they will prefer similar products. However, the ranking of products at each node depends on their technical attributes. In this way, if the system learns that, say, the storage capacity of a laptop is an important feature for a particular user segment, the system will weigh this feature appropriately and will rank high other laptops with large hard disks. This identification of implicit relationships between user and product attributes lets the system deal with both new users and with new items, alleviating cold-start problems.

Figure 1 shows an example of a "*Which camera should I buy?*" flowchart with technical questions (left) and an equivalent example with non-technical questions produced by ShoppingAdvisor (right). A shopper answering `yes` to both questions "*Do you want a camera for traveling?*" and "*Do you want a camera for backpacking trips?*" is more likely to be recommended a ultra-compact camera.

We demonstrate the effectiveness of our interactive recommendation system for two categories of products: *cars* and *cameras*. In addition to the conceptual contributions – introducing a new problem definition and developing novel methods to solve this problem – we also show how to mine publicly available data to create datasets required by our problem definition, namely, integrating information regarding users, products, and reviews.

For our first use case, cars, we extract information from Yahoo! Autos.[1] We collect car specifications (numerical, boolean and categorical attributes), as well as ratings and reviews submitted by users. Each user review includes pros and cons, short free-text snippets that highlight positive and negative aspects of the car. We employ standard text-mining techniques to extract tags such as `fuel economy`, `stylish` and `performance` from the reviews. We use those tags as user attributes: we represent each user by the set of tags that the user has used in the pros section of his reviews.

We take a different route for cameras and use data from flickr.[2] In this case the tags used by each user to describe their photos are used as a proxy to user interests. We extract the cameras used to take the photos with such tags from the metadata, and we retrieve their technical specifications from CNET.[3] Rather than using review scores from CNET, we choose to test our framework in a different setting, where review scores might not be available, and use popularity of a camera in flickr as a proxy. For each tag, we count how many pictures were taken with a given camera, and use this number to rank the cameras.

We use mean reciprocal rank (MRR) to evaluate ShoppingAdvisor, and we show how the performance increases by more than 50% along the path from root to leaf. We also show how collaborative filtering algorithms such as k-nearest neighbor benefits from *feature selection* done by ShoppingAdvisor. We obtain results of comparable quality by using only a subset of user attributes, i.e., features for k-NN, while producing more interpretable recommendations.

2. PROBLEM DEFINITION

We consider that our input data consists of three tables: a user table $\mathbf{U}$, a product table $\mathbf{P}$, and a review table $\mathbf{R}$.

The table $\mathbf{U}$ contains information about the users and it has dimensions $n_U \times m_U$. That is, we assume that we keep information for n_U users, and each user is described with m_U attributes. We do not make any explicit assumption regarding what type of information we have, it may be any kind information, for example, registration information such as demographics, or implicitly provided information, such as tags used on social-media sites, or browsing behavior. Such information expresses the interests and lifestyle preferences of users. We use the notation u_i to refer to the i-th user.

The second component of our data, the table $\mathbf{P}$, is of dimensions $n_P \times m_P$ and contains product information. We assume that we have information for n_P products, and each product is described with m_P technical attributes, typically provided by the manufacturer. For instance, for cameras the attributes contain information such as `resolution`, `zoom`, `aperture`, and `weight`. We use the notation p_j to refer to the j-th product.

The last component of our data is the review table $\mathbf{R}$ with dimensions $n_C \times 3$, and each row describing a reviewing action (u_i, p_j, s_{ij}). This row contains data about user u_i evaluating product p_j with score s_{ij}. We may think of s_{ij} as a number in an interval, say, $s_{ij} \in [1, 5]$, where $s_{ij} = 1$ reflects a negative opinion and $s_{ij} = 5$ is positive.

Examples of a user table, a product table, and review table are shown in Tables 1, 2, and 3, respectively. Note that a

[1] http://autos.yahoo.com

[2] http://www.flickr.com

[3] http://www.cnet.com

Table 1: An example user table **U**.

uid	Gender	Age	Travel	Family	Live	Video
1	female	young	1	1	0	0
2	male	old	0	1	1	0
3	female	young	0	0	1	1
4	male	teen	1	0	0	1

Table 2: An example product table **P**.

pid	Weight	Resolution
1	5.0 lbs	36.3 MP
2	1.7 lbs	10.1 MP
3	2.0 lbs	14.0 MP
4	4.5 lbs	28.1 MP

Table 3: An example review table **R**.

uid	pid	Rating
1	3	4.5
2	2	3.5
3	1	4
3	3	3

user may review more than one product, and a product may be reviewed by more than one user.

For ease of reference, we denote by $\mathcal{P}$ the set of all products and by $\mathcal{U}$ the set of all users in our system. The set of all user attributes is denoted by $\mathcal{A}$. Note that $|\mathcal{P}| = n_P$, $|\mathcal{U}| = n_U$, and $|\mathcal{A}| = m_U$.

The system we design and build in this paper, ShoppingAdvisor, is a *decision tree* that guides the users in making their shopping decisions. We denote this tree by $\mathcal{T}$. The internal nodes of the ShoppingAdvisor tree $\mathcal{T}$ correspond to user attributes in $\mathcal{A}$. A potential shopper is supposed to use the tree $\mathcal{T}$ as a *flowchart* for receiving product recommendations. A user starts from the root of the tree. The root, as any internal node, contains a user attribute $a \in \mathcal{A}$. The attribute a is perceived as a question by the user. For example, if a is a demographics attribute, the question can be *"Are you in the a demographics group?"*, while if a is a tag, the question can be *"Are you interested in tag a?"* The user, by answering this question, follows the left or right subtree, and continues recursively answering questions in internal nodes of $\mathcal{T}$until they reach a leaf node or they decide to stop.[4]

The leaf nodes of $\mathcal{T}$ correspond to an *ordering* of the products in $\mathcal{P}$. Once a shopper reaches a leaf node $\ell \in \mathcal{T}$, by following the internal nodes of $\mathcal{T}$ and answering questions, ShoppingAdvisor recommends to the shopper products in the order specified by the leaf node ℓ. In practice, we assume that ShoppingAdvisor recommends the top-k products in the ordering, although the user may select a "`more` k `products`" button. We leave such system details outside our discussion and focus on the algorithmic abstractions.

We now define the main problem we address in this paper.

Problem 1 (ShoppingAdvisor). *We are given as input a product table* **P***, a review table* **R***, a user table* **U***, and integers h and k. The task is to learn a* ShoppingAdvisor *tree $\mathcal{T}$. Each internal node of the tree contains a question formed by a user attribute $a \in \mathcal{A}$. Each tree node contains a top-k ranked list of the products $\mathcal{P}$. The height of the tree is restricted to be h.*

The objective of the tree $\mathcal{T}$ is to provide relevant recommendations on products $\mathcal{P}$. A new shopper, starting from the root, traverses down the tree $\mathcal{T}$, answering at most h questions, until reaching a leaf node, where the user receives the top-k recommendations contained at that leaf node. The quality of the learned tree $\mathcal{T}$ reflects the quality of such recommendations made to a new user/shopper.

To make the definition of our problem precise, we need to quantify how we evaluate recommendations made by the ShoppingAdvisor tree $\mathcal{T}$. For this task, we use *ten-fold cross-validation.* Users in the evaluation fold have reviewed products – their scores are contained in the review table **R** – and thus we can evaluate the quality of $\mathcal{T}$'s recommendations. As we need to evaluate a ranked list we can employ standard measures from information retrieval.

As already mentioned, we model our ShoppingAdvisor system as a decision tree. We learn the structure of the tree by partitioning the set of users in the training set recursively on the basis of available user attributes – such as, demographics, tagging behavior, and so on – and then match a test user, to the leaf node of users with whom the user shares their preferences. For each group of users, we also learn the product attribute weights and generate a ranking of top-k products in the training set, in order to identify the top products for recommendation.

3. ALGORITHMS

3.1 A general algorithmic framework

We first introduce a general algorithmic framework for solving the ShoppingAdvisor problem. Our algorithmic framework uses two functions as black boxes. The first is a payoff function, used to choose the best question to ask at any node of the ShoppingAdvisor tree $\mathcal{T}$. In other words, the function payoff determines the best user attribute $a \in \mathcal{A}$ to partition the set of users at any node of the tree $\mathcal{T}$. The second function is a rank function, used to determine the ranking of the products recommended to users. Learning the rank function can be seen as learning weights on the attributes of products, which in turn can be used to rank all products in $\mathcal{P}$; and subsequently select the top-k products.

Typically, decision trees are constructed in a top-down fashion, where each internal node splits the training instances into two or more subspaces. In the ShoppingAdvisor tree $\mathcal{T}$, each internal node of the tree corresponds to the set of users whose attributes match the attributes at all internal nodes on the path from the root to that node. Thus, we recursively partition the set of users at each internal node. The partition criterion is that the users within each side of the split should agree on their ratings on the products. The goal is to select the user attribute, so that when we perform the split based on that attribute, the uniformity criterion on the user ratings is maximized. As an example, if it so happens that avid hikers tend to prefer a certain camera, for its weight, ruggedness, and ability to take high-quality outdoors photos, then the hiking tag should be used to split the active users at that step of the construction of the tree.

Each tree node contains a ranking of products, induced from a model that uses the ratings of the users belonging to that node. Consider a shopper who has cascaded until a node q of the tree $\mathcal{T}$. If q is a leaf node, the shopper is provided with a top-k list of product recommendations. If q is an internal node, the shopper is asked the question that corresponds to q, and depending on the answer cascades to one of the two subtrees of q. The shopper is also given

[4] In this paper we focus on identifying the attribute of interest, and not on the task of formulating the question in a human-interpretable way. We assume that this problem can be solved independently, even with human supervision of the designers of the ShoppingAdvisor application.

the possibility not to answer the question and receive the current recommendation at node q. This is possible since a product ranking is available at each node. For simplicity, in the evaluation performed in this paper, we assume that shoppers navigate until tree leafs.

Learning the tree structure. Determining the structure of the tree $\mathcal{T}$ is equivalent to choosing which user attribute $a \in \mathcal{A}$ to use for a question at each node.

Formally, consider a tree node q and the set of users U_q who correspond to q, namely, the set of users whose attributes match the attributes at all internal nodes on the path from the root of the tree until q. Given a candidate splitting user attribute $a \in \mathcal{A}$, two subsets of U_q can be defined: the set of users $U_q(a)$ who match attribute a, and the set of users $U_q(\overline{a})$ who do not match attribute a. For simplicity, we assume that $U_q = U_q(a) \cup U_q(\overline{a})$ and $U_q(a) \cap U_q(\overline{a}) = \emptyset$, although it is not required as our framework can handle overlapping subsets. The root node of the tree comprises all n_U users of the user table $\mathbf{U}$.

To determine the best user attribute $a \in \mathcal{A}$ to split U_q at node q we evaluate the pay-off function associated with the sets of users resulting from the split. In particular, we consider a $\mathsf{combine}$ function that maps the pay-off of the two subsets to a single-valued measure.

$$\mathsf{payoff}(q, a) = \mathsf{combine}(\mathsf{payoff}(U_q(a)), \mathsf{payoff}(U_q(\overline{a})), |U_q(a)|, |U_q(\overline{a})|, |U_q|),$$

where the function $\mathsf{payoff}(U)$ evaluates the quality of ranking induced by a set of users U. There are a number of natural options for the $\mathsf{combine}$ function, such as *sum*, *arithmetic mean*, *geometric mean* and *harmonic mean*.

One has to consider all possible user attributes $a \in \mathcal{A}$, and choose as splitter the one that maximizes the pay-off.

$$\mathsf{splitter}(q) = \arg\max_{a \in \mathcal{A}} \mathsf{payoff}(q, a). \tag{1}$$

The idea behind such a posterior goal of maximizing payoff is to partition the set of users into two groups, which have similar preferences, and which rank the products in $\mathcal{P}$ in a similar way. Furthermore, we aim at leveraging hidden correlations among the set of user attributes and the set of product attributes. For instance, in the previous example with hikers preferring certain lightweight cameras, our tree should learn the fact that the weight of a camera is an important feature for that specific subset of the population, and thus, it should tend to rank lightweight cameras higher, even if they have not been explicitly rated.

Note that our recursive algorithm is an instantiation of a greedy heuristic. In principle, selecting the best splitter at a certain node according to Equation (1) may lead to globally suboptimal solutions. However, this is a cost we have to pay due to the **NP**-hardness of the ShoppingAdvisor problem.

Learning product rankings. We next consider the problem of learning to rank the products in $\mathcal{P}$ at a given tree node q. The input consists of users U_q belonging to node q. We also have access to the product table $\mathbf{P}$ and the review table $\mathbf{R}$. In fact, we only need the rows of the review table $\mathbf{R}$ corresponding to U_q, and denote the sub-table as $\mathbf{R}_q$.

The objective is to learn a function $\mathsf{rank} : \mathcal{P} \to \mathbb{R}$ which, given a product $p \in \mathcal{P}$ specified by its technical attributes (i.e., by a row in the product table $\mathbf{P}$), returns a value $\mathsf{rank}(p)$, which in turn can be used to induce a ranking on the set $\mathcal{P}$. We opt for a linear function, and thus the task is to learn weight coefficients for the product attributes. Note that in order to handle categorical attributes of products with our linear ranking function we convert such categorical attributes into a set of boolean attributes, and then treat them as numerical 0–1 values.

To learn the function rank we can use any learning-to-rank algorithm, such as Regression, RankSVM, or GradientDescent. To employ such a learning-to-rank algorithm we need to assign a score to each product p in $\mathbf{P}$. Such a score is computed as the average rating of p in the review table $\mathbf{R}_q$, that is, the average rating of p over the set of users who correspond to the node q under consideration.

Putting the learning components together. We now discuss the interaction of the two learning ingredients: learning the tree structure and learning the product ranking. We first observe that the ranking function we learn at a tree node q depends on the set of users (and their ratings on products) who correspond q. We write rank_U to emphasize the dependance of the ranking function from a set of users U. The quality of ranking can be evaluated using a quality measure eval. Functions such as *precision*, *recall*, *normalized discounted cumulative gain* (ndcg), and *mean reciprocal rank* (mrr) can be used as eval functions. We write $\mathsf{eval}(\mathsf{rank}_U)$ to denote the quality of a rank_U ranking measured by such an eval measure. Finally, we recall that a good user attribute to split a tree node q is a user attribute that induces subpopulations with good rankings in each one. The quality of the ranking should be reflected in the payoff function used. Thus we set

$$\mathsf{payoff}(U) = \mathsf{eval}(\mathsf{rank}_U). \tag{2}$$

In the next section we discuss our considerations for the functions payoff, rank, and eval.

3.2 The LearnSATree algorithm

Our proposed algorithm, LearnSATree, for learning ShoppingAdvisor tree is an instance of the general algorithm presented in the previous section, with judicious choices for the functions payoff, rank, $\mathsf{combine}$, and eval. We now discuss more in detail those choices. We first introduce our learning-to-rank model and we define the function rank that learns weights of product attributes.

Learning to rank. The goal is to learn a weight vector $\mathbf{w} = \{w_1, \ldots, w_{m_P}\}$ for the m_P technical attributes of the products $\mathcal{P}$. As discussed in the previous section, the learn-to-rank algorithm is applied to each tree node q. Assume that for a node q we have a set $\mathbf{P}_q$ of training instances. Those training instances are a subset of products of $\mathcal{P}$ accompanied with review scores from the users associated with node q. Following the notation of the previous section, the training instances are the result of joining the product table $\mathbf{P}$ with the review sub-table $\mathbf{R}_q$. For learning the ranking function rank we employ a pairwise RankSVM method [13]. In this approach, the training instances under consideration are expanded into a set of preference pairs. Namely, we create ordered pairs of products $(\mathbf{p}_i, \mathbf{p}_j)$ where the product $\mathbf{p}_i$ has higher score than the product $\mathbf{p}_j$ from the users in the tree node q. Let us denote the set of such ordered pairs by $\mathbf{P}_q^2$. We then find the weight vector $\mathbf{w} = \{w_1, \ldots, w_{m_P}\}$ by optimizing a pairwise objective function:

$$\min_{\mathbf{w} \in \mathbb{R}^{m_P}} \left\{ \frac{\lambda}{2} \|\mathbf{w}\|^2 + \sum_{(\mathbf{p}_i, \mathbf{p}_j) \in \mathbf{P}_q^2} \mathsf{loss}\left(\mathbf{w}^T \cdot (\mathbf{p}_i - \mathbf{p}_j)\right) \right\},$$

where loss is a suitably-defined loss function, such as *hinge loss*, i.e., $\mathsf{loss}(y) = \max(0, 1-y)$. For the class of linear ranking functions, the objective of attaining an optimal ranking function rank^*, i.e., finding the weight vector $\mathbf{w}$ so that the number of inversions is minimized, is **NP**-hard [7], and the RankSVM algorithm provides an approximate solution. Once the weight vector $\mathbf{w}$ is learned, the rank function is defined as $\mathsf{rank}(\mathbf{p}) = \mathbf{w}^T \cdot \mathbf{p}$, and it induces a ranking on the *whole* set of products $\mathcal{P}$ by $\mathsf{rank}(\mathbf{p}_1) \geq \ldots \geq \mathsf{rank}(\mathbf{p}_{n_P})$.

Besides promoting the popular products belonging to the training instances for the users of node q, our method ensures that all products will be ranked, even products that have not been reviewed by the set of users in node q. This property helps us handle the cold-start problem, where a new product arrives in the system and initially there is very little feedback available for that product.

Next we evaluate the quality of the ranking generated by our method, which according to Equation (2) defines the payoff function.

Evaluating the ranking. The learning-to-rank model induces a ranking $\mathsf{rank}(\mathbf{p}_1) \geq \ldots \geq \mathsf{rank}(\mathbf{p}_{n_P})$ on the products in $\mathcal{P}$. The quality of this ranking is measured by the eval function. Our eval function measures the number of correctly-ranked pairs in the ranking generated for the products in $\mathbf{P}_q$, that is, the products in our training set in the node q. So, assuming that the set $\mathbf{P}_q$ contains n products, we have

$$\mathsf{eval}(\mathsf{rank}) = \frac{2\,|\{(\mathbf{p}_i, \mathbf{p}_j) \in \mathbf{P}_q^2 \mid \mathsf{rank}(\mathbf{p}_i) > \mathsf{rank}(\mathbf{p}_j)\}|}{n(n-1)}.$$

Note that the reason why we use this evaluation function, rather than other measures mentioned before such as precision, recall, and NDCG, is that minimizing the number of inversions is the most common way to optimize a pairwise learning-to-rank function. In other words, our choice of the eval function stems from the RankSVM approach.

Likewise, we choose to use *sum* as the $\mathsf{combine}$ function. Indeed, by summing the number of correctly ranked pairs we are guaranteed by RankSVM that the eval function is monotonically increasing with the height of the tree. In analogy with decision trees, this property allows for effective pruning strategies while building the tree.

Stopping criterion. The construction of a decision tree has another critical element: deciding when to stop growing the tree. Ideally, the algorithm will stop its recursive partitioning of the subspaces along one (or both) direction(s) if the *perfect* ranking is achieved in the left and right children nodes, i.e., the payoff associated with splitting by attribute $a \in \mathcal{A}$ for node q is 0. In reality, we first grow the tree to its entirety, and then employ post-pruning with the aim of removing sections of the tree that provide little power to capture user preferences. For example, for a node q split by tag `travel`, if its child node is split by the near-synonomous tag `vacation`, our post-pruning rules should trim the child node (or, the associated set of nodes). We employ pruning rules on the validation set. Our most significant stopping condition, in addition to the regular rules, follows from the observation that – the number of inversions due to our ranking model on a set of training instances belonging to a node montonically decreases with the decrease in size of the training set along a root-to-node path.

4. EXPERIMENTS

We evaluate our ShoppingAdvisor system with both real and synthetic data. Our primary objective is to demonstrate the effectiveness of our system by comparing the quality of recommendations returned by our system with recommendations made by baseline system(s) not leveraging user attributes. We highlight how popular collaborative recommendation technique(s) benefit from our system. Besides a quantitative comparison of the recommendation quality, we also conduct a detailed use-case evaluation, where we show that recommendations with consideration of shoppers' personal preferences are superior to those by traditional state-of-art systems. We also show the scalability of our algorithm by studying the running time under varying parameters.

4.1 Datasets

Car dataset: Our first dataset, named Car, is extracted from Yahoo! Autos. We focus on new cars listed for the year 2010 spanning 34 different brands. There are several models for each brand, and each model offers several trims.[5] Since each trim defines a unique attribute-value specification, the total number of trims that we crawl are the 606 products in our dataset. The products contain technical specifications as well as ratings and reviews, which include pros and cons. We parse a total of 60 attributes: 25 numeric, and 35 boolean and categorical (which we generalize to boolean). The total number of reviews we extract is 2 180. In this dataset we do not have user information, and thus we consider that each user has reviewed only one car. Thus, the total number of users is also 2 180. Since the ShoppingAdvisor system considers tags as form of user feedback, we extract tags from the reviews using the keyword extraction toolkit AlchemyAPI.[6] We process the text listed under "pros" in each review to identify a set of 15 desirable tags such as `fuel economy`, `comfortable interior` and `stylish exterior`.

Camera dataset: Our second dataset, named Camera, is on cameras. For extracting user preferences for this dataset we take a different route. Rather than using an e-commerce site such as Yahoo! Shopping or Amazon, we use a social-content site such as flickr. Our intention is to capture user preferences by the tags that people use to describe their photos. So we assume that flickr tags such as `food`, `nature`, `animal` and `landscape` are meaningful representations of user preferences. Furthermore, we intend to leverage the hidden associations between photo tags and cameras. As an example, of the 3 000 photos in our flickr dataset tagged as `food`, Canon EOS 20D, Canon EOS 350D, Canon EOS 400D and Nikon D80 have been used 1 200 times while 550 other cameras have been used the rest of the times. T herefore, a shopper looking for a camera for taking food photos will be recommended a camera from among Canon EOS 20D, Canon EOS 350D, Canon EOS 400D and Nikon D80.

The technical specifications of the cameras are retrieved from `CNET`. Our flickr dataset sample has 135 025 photos uploaded by 37 064 users using 9 365 cameras; the tag vocabulary size is 422 240. We clean the dataset to consider only those instances in which (*i*) the cameras have well-defined set of technical specifications in `CNET` and are not phone cameras or digital camcorders; (*ii*) the tags are valid English words; and (*iii*) a user has used at least two cameras.

[5] Trims denote different configurations of standard equipment.

[6] `http://www.alchemyapi.com/api/keyword/`

Our final reduced flickr dataset has 11 468 photos and 10 845 tags from 5 647 users and 654 cameras. Since the number of tags is huge, we use latent Dirichlet allocation (LDA) technique and aggregate the tags into 25 topics based on their co-occurrence. We then express user preferences at the level of LDA topics. Thus, our Camera dataset has 11 468 instances in the review table **R**. It has 5 647 users described by 25 attributes (tag topics) in the table **U**. And it has 654 cameras in the table **P**. Some indicative user attributes, i.e., tag topics, are the following: `wildlife` (tags: `birds`, `zoo`, etc.), `food` (tags: `fruit`, `market`, etc.), `sports` (tags: `car`, `tennis`, etc.), and so on. Those tag topics correspond to the questions required to build the ShoppingAdvisor tree.

Synthetic dataset: We generate a large matrix of dimension 4 000 × 12 corresponding to the review table **R**. There are 200 products in the product table **P**, each having 20 boolean attributes and 1 000 users in the user table **U**, and 20 tags in the review table **R**. We randomly assign the products to the users for the 4 000 comments. We split the 20 independent and identically distributed attributes into four groups, where the value is set to 1 with probabilities of 0.75, 0.15, 0.10 and 0.05, respectively. For each of the 10 tags, we pre-define relations by randomly picking a set of attributes that are correlated to it. A tag is set to 1 with a certain probability if the majority of the attributes in its pre-defined relation have boolean value 1. In order to generate rating scores for each of the 4 000 user-item interactions, we randomly distribute the 1 000 users into ten categories. For each user category, we pre-define relations by randomly picking a set of tags that are preferred by the users in that category. A user rates a product high, medium or low based on the proportion of his preferred tags in the tag vector corresponding to the product she has reviewed.

4.2 Experiment setup and evaluation metrics

In order to evaluate our recommender system, we partition each of our datasets into training and test sets via ten-fold cross-validation. We use the training set for building our ShoppingAdvisor system, and measure its performance on the test set. We also consider a part of the training set as a validation set in order to optimize the model parameters, i.e., in order to post-prune the decision tree. The main evaluation metric that we use in our experiments to measure the quality of recommendations is mean reciprocal rank (MRR), which is a meaningful measure for single-item retrieval.

Mean reciprocal rank (MRR): In information retrieval MRR measures how far away from the first retrieved document the first relevant one is. In our datasets, with one relevant item per test user, we measure the recommendation quality by finding out how far from the top of the list the relevant item is. The reciprocal rank of a test user is the multiplicative inverse of the rank of the relevant item for that test user. The mean reciprocal rank is the average of the reciprocal ranks of results for all test cases

$$\mathsf{MRR} = \frac{1}{k}\sum_{i=1}^{k}\frac{1}{\mathsf{rank}_i},$$

where k is the number of instances in the test set and rank_i is the position of the i-th test user's relevant item in the ranked list of items returned. Note that, we partition our dataset into training and testing in such a way that the test set consists of users who have rated items high. In this way, for each user in the test set there is a highly relevant item and thus the MRR measure is meaningful.

Our second quantitative indicator is efficiency, i.e., running time for training; testing is fast as it only involves descending down the ShoppingAdvisor tree and receiving a recommendation at a leaf. The training time involves building the decision tree, as well as learning the ranking model for each possible split in a node. Clearly, the learning to rank module, i.e., RankSVM is an expensive operation especially since it requires to build the preference matrix each time. We employ techniques to pre-materialize part of the preference matrix, and thus reduce training time (more details in Section 4.4). We also present a detailed report of anecdotal evidence for real data in Section 4.5 as a qualitative validation of the effectiveness of the system.

Parameter settings: In our experiments, we use Quinlan's C4.5 algorithm to build the decision tree. For RankSVM, we use Olivier Chappelle's RankSVM implementation, and we set the training error penalization parameter C to 0.001 for Synthetic and Car datasets, and 0.0001 for the Camera dataset. The total number of instances in the Car dataset available for partitioning into train and test set is around 10 000. We select a subset of the total data for training and testing (maintaining ten-fold cross-validation requirements) in order to avoid running out of memory. For the Camera and the Synthetic dataset, we work with the complete data.

4.3 Quality evaluation

To validate the effectiveness of ShoppingAdvisor, we compare its performance with baseline RankSVM.

RankSVM: This is a pairwise learning-to-rank algorithm [13] that generates recommendations by learning item-feature weights. Our LearnSATree algorithm employs RankSVM for learning to rank, therefore we can consider this technique equivalent to the ranked list returned by ShoppingAdvisor at the root, before a potential shopper answers any question.

We also compare the performance of k-NN with a variant, described below in order to demonstrate how ShoppingAdvisor is useful to existing recommendation techniques:

k-NN: This is standard collaborative-filtering algorithm that matches a test user to a set of users in the training set, and returns a ranked list of items by aggregating the item lists for the top neighbor users in the training set.

SA.k-NN: Feature selection and weighting has an important role in improving the effectiveness of a k-NN learner. ShoppingAdvisor allows k-NN to select a subset of features, i.e., tags by traversing down the tree, and then perform k-NN (user-based or item-based) on the reduced feature space. It is particularly useful since collaborative recommendation algorithms assume the availability of user preferences for matching similar users with similar interests, which may not be the case in reality. Moreover, user preferences drift rapidly over time and it is better to elicit user responses before making recommendations.

Table 4 presents a comparison of recommendation quality, measured by average MRR over 10 folds, of ShoppingAdvisor with RankSVM, and k-NN with SA.k-NN for both the synthetic and the real datasets. The average height of the trees for the Car, Camera, and Synthetic datasets are 15, 19 and 7, respectively. The average number of questions users in test set answer to receive their recommendation in the leaf nodes of ShoppingAdvisor are 12, 12 and 6, respectively.

Table 4: MRR comparisons of ShoppingAdvisor with RankSVM and k-NN with SA.k-NN for Car, Camera, and Synthetic datasets.

Dataset	RankSVM	ShoppingAdvisor	k-NN	SA.k-NN
Car	0.013	0.019	0.020	0.022
Camera	0.012	0.019	0.029	0.029
Synthetic	0.060	0.231	0.604	0.580

Table 5: Training time of ShoppingAdvisor for Car, Camera, and Synthetic datasets.

Dataset	Training	Features	Materialized	Time
	Set Size	(tags)	Preference Pairs	(in sec)
Car	1900	15	36000	256
Camera	5000	25	50000	2168
Synthetic	3500	10	40000	1950

Table 6: Training time increases rapidly when exact preference matrix is considered (Synthetic dataset).

Training	Exhaustive	Train Time	MRR	Materialized	Train Time	MRR
Set Size	Preference Pairs	(in sec)		Preference Pairs	(in sec)	
50	965	0.948	0.045	483	0.726	0.045
100	3766	6.029	0.097	1883	1.472	0.093
150	8382	46.366	0.131	4191	2.549	0.128
200	14831	209.462	0.199	7416	3.375	0.196
250	23089	649.030	0.204	11545	5.927	0.203
300	33204	1526.300	0.200	16602	9.935	0.200

We observe that leveraging shopper preferences clearly yields better quality recommendations (i.e., at the leaf nodes of ShoppingAdvisor) than those returned when the user does not answer any questions (i.e., RankSVM at the root node). The increment in quality is around 50% for both real datasets from root to leaf (see last two columns in Table 4). The average MRR score for k-NN and SA.k-NN are comparable for all three datasets. This indicates that SA.k-NN returns recommendations of very high quality to the users by asking a smaller number of questions than what the k-NN method would be asking. For the Camera dataset, the k-NN method reaches a quality score of 0.029 by asking 25 questions (since the number of tag topics in the dataset is 25) while SA.k-NN achieves the same accuracy by asking only 12 questions, on average. Furthermore, an additional benefit of SA.k-NN compared to k-NN is that SA.k-NN does not require the recommender to be aware of the shopper preferences, and is therefore useful in handling new users.

Note that our ShoppingAdvisor system is also capable of handling new products, without existing reviews, by using their technical attributes, while the recommendations of k-NN are restricted to products with reviews in the training set. Finally, the tree structure in ShoppingAdvisor and SA.k-NN provides a logical explanation of the recommendations being returned, while the recommendations of k-NN are not easily interpretable.

4.4 Performance evaluation

Next, we present the time taken for training our system for synthetic and real datasets. The training time is the time taken to build the tree, which includes the time to execute RankSVM for all possible user attributes in a node. For example, in order to decide the splitting question for the root node from the pool of 15 questions in the Car dataset, our algorithm has to perform RankSVM 30 times (15 for each children node). RankSVM is an expensive operation since it builds the preference matrix from the set of training instances belonging to a node.

The first section of Table 6 shows the increase in training time, averaged over 10 folds, with increase in number of training instances for synthetic data. MRR is obtained on a set of 20 test instances sampled from the synthetic data. Employing this method for the Camera and Car datasets, which have a few thousand instances in the training set, would be very expensive. Therefore, we materialize a preference matrix for all training instances in a dataset, and use that to acquire the preference pairs to be considered for a RankSVM operation. However, such materialized preference matrices for the Camera and Car datasets would take several days to build and would be several gigabytes in size. Therefore, for each item in training set, we make a random selection of preference pairs from the space of all training instances, instead of opting for all possible pairs. The second section of Table 6 shows the train time and the recommendation quality when 50% of the preference pairs are considered from the pool of all possible preference pairs, under the same framework settings. We observe that while training time drops sharply, there is hardly any decrease in MRR. This indicates that we achieve faster training time without compromising RankSVM performance quality. When the number of training instances drops in the deeper level nodes, the number of preference pairs that can be retrieved from the materialized preference matrix may fall too, thereby requiring to generate the preference pairs at run time. On the other hand, classifying a test user is extremely fast. Table 5 shows the training time for the different datasets, along with the number of materialized preference pairs in input.

4.5 Examples of ShoppingAdvisor trees

Figures 2 and 3 present snapshots of ShoppingAdvisor trees built for the datasets Car and Camera, respectively. In Figure 2, a user is asked if they would like to buy a stylish car. A yes takes her to the next question about the interior of the car. If the user does not express any requirement for a comfortable and roomy interior, ShoppingAdvisor specifically asks if they are interested in a great audio system inside the car. On the other hand, a user who wants a car that has good fuel economy is immediately asked if acceleration is important, since fuel-efficient cars have slower acceleration.

Table 7 presents the top recommendations at three nodes of the tree shown in Figure 2. We observe that inclusion of the fuel economy condition brings in a hybrid car and an ecoboost car in the top recommendations, while exclusion of requirements of a fuel-efficient car makes ShoppingAdvisor recommend a Audi, which is known to compromise mileage for performance. Again, the presence of Jeep Grand Cherokee SRT-8 in all three nodes reflects its popularity in 2010.

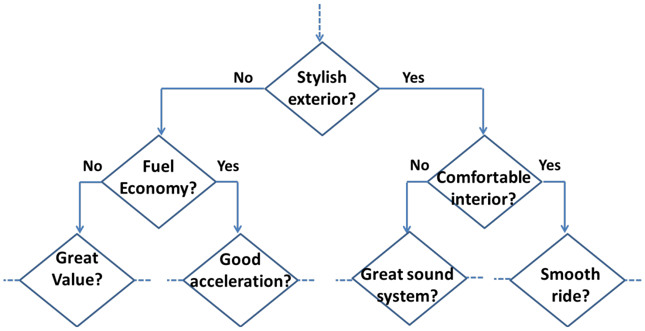

Figure 2: ShoppingAdvisor for the Car dataset.

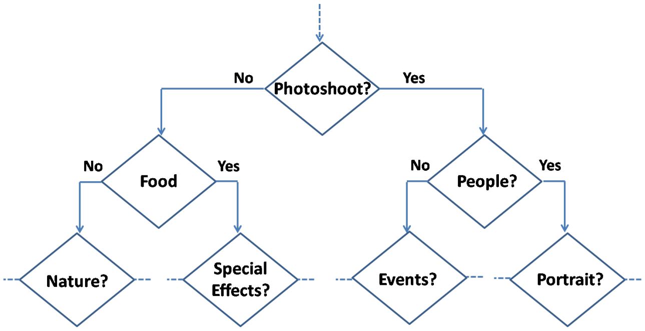

Figure 3: ShoppingAdvisor for the Camera dataset.

Table 7: Example of top car recommendations at three nodes of tree in Figure 2.

ShoppingAdvisor	Top Cars For Recommendation
Stylish Exterior = NO	Jeep Grand Cherokee SRT-8 Dodge Challenger SRT-8 Volvo XC60 AWD
Stylish Exterior = NO Fuel Economy = NO	Jeep Grand Cherokee SRT-8 Dodge Challenger SRT-8 Audi Q5 Premium quattro Tiptronic
Stylish Exterior = NO Fuel Economy = YES	Lincoln MKS 3.5L EcoBoost AWD Jeep Grand Cherokee SRT-8 Ford Escape Hybrid

Table 8: Example of top camera recommendations at three nodes of tree in Figure 3.

ShoppingAdvisor	Top Cameras For Recommendation
Photoshoot = YES	Canon EOS Digital Rebel XS Nikon D80 Fujifilm FinePix S3

Photoshoot = YES
People = NO
	Canon EOS Digital Rebel XS
	Olympus E-3
	Nikon D50

Photoshoot = YES
People = YES
	Canon EOS Digital Rebel XS
	Canon EOS 30D
	Nikon D80

Similarly in Figure 3, a user is asked if they are interested to buy a camera for photoshoot purposes, and if the photoshoot to be conducted is for people, and if it is focused on face shots. Thus, we see that ShoppingAdvisor narrows down the preferences of the shopper, and helps recommend cameras tailored to her needs. The camera recommendations at three nodes of the tree are shown in Table 8. For example, one of the top cameras recommended to her, Canon EOS 30D, introduced an auto image rotation feature in order to make better use of the LCD display especially during portrait-orientated shots. Again, for the shopper who is looking for a camera for shooting events and not people, ShoppingAdvisor recommends Olympus E-3 which happens to be a lightweight digital SLR camera. The presence of the Canon EOS Digital Rebel XS in all three nodes reflects its popularity among flickr members.

5. RELATED WORK

Our research is motivated by the observation that people with limited domain knowledge feel better supported when presented with qualitative product information rather than technical details [2]. This fact is even more important when the complexity of the product calls for an expert-driven approach, which identifies the needs of the user rather than proposing options based on product features [9].

From a technical point of view, our recommender system can be thought of as a mixture of collaborative filtering, interactive eliciting of user preferences, and learning-to-rank. Recommendation is traditionally formulated as the problem of estimating ratings for items that have not been seen by a user [16]. Once these ratings are estimated, a recommendation is built by picking the items with highest rating. However, we are not actually interested in the estimated ratings, but only in the induced *ranking*. A more recent formulation makes this assumption explicit, and casts the recommendation task as a ranking problem [3]. Given this formulation, techniques from the learning-to-rank literature can be applied to learn personalized ranking functions.

Learning to rank. Our framework uses a learning-to-rank algorithm as a basic building block to generate recommendations. We chose to use SVM-rank [13], but other algorithms like RankBoost [8] would be equally viable.

Probabilistc Boost Tree (PBT) [19] builds a tree in which each node is an instance of a boosting algorithm. The tree grows by splitting the training instances based on the classification performed at each node. While PBT performs classification, our algorithm performs ranking. TreeRank [6] uses a tree-based algorithm to solve the bipartite ranking problem, where each element has a binary relevance label and the goal is to rank all relevant items on top. Such binary relevance is not adequate to express complex user preferences.

E-commerce. Application of recommender systems to e-commerce dates back to the '90s [17]. While early works often mined transaction logs, more recent works focus on user ratings, especially for digital media like movies and news. Adomavicius and Tuzhilin [1] provide an overview of recommender systems and their categorization into content-based, collaborative filtering and hybrid.

Amazon's "customers who bought" feature is based on co-buying information [14]. Similarly, Hsu et al. [11] create a recommender system for e-commerce based on mining transaction logs that uses a probabilistic graphical model to handle skew and sparseness in data. Our work does not rely on transaction logs and only takes into account implicit or explicit user feedback.

Our system bears resemblance to Bayesian networks, where each node is a decision tree and each edge represents user details [5]. In our case the network is a decision tree with constraints; user details are elicited explicitly via questions; nodes are instances of a learning-to-rank algorithm.

As the recommendations provided by our system derive from eliciting user preferences, they are easy to explain to the user. Explanation of recommendation has been shown to be effective in increasing the acceptance of recommendations [10]. Our recommendation system naturally lends itself to a keyword-style explanation, which has been found to be the most effective kind of explanation [4].

User preferences. The cold start problem of collaborative filtering is usually addressed by eliciting user preferences [15]. The state-of-the-art is *example critiquing* [20]: a user is presented an example recommendation and asked to critique it by operating on the attributes of the item. However, while example critiquing works for suggesting items whose domain is common knowledge, it implicitly assumes that the user understands the domain and all its attributes. Our system avoids this assumption by eliciting preferences via an interactive questionnaire on lifestyle questions about the user rather technical questions about the item.

Stolze and Ströbel [18] present an interactive system that is similar to ours in spirit. The system recommends technical products by a asking a series of questions that start from high-level needs of the user and transition to technical features of the item. Users are clustered into "target groups" according to their similarity in evaluation structure. While the idea is similar to ours, the authors only describe the interaction between the system and the user without discussing how how such a system would be built and which algorithm would power the recommender system. Our system is also closely related to the problem of constraint-based recommender systems [12], which interact with users to collect user preferences before making recommendations. However, users typically submit their preferences in the form of explicit requirements or scoring rules on technical dimensions. Indeed, our system is able to automatically build a non-technical question tree.

6. CONCLUSION

We have proposed a novel recommender system that helps users to shop for technical products. Our system, ShoppingAdvisor, draws inspiration from manually-created flowcharts used to guide shoppers in their purchases. The fundamental design principle is to learn a flowchart that contains questions that are understandable to non-expert users. We thus aim to elicit answers from a user feature space rather than a product feature space.

The first goal of ShoppingAdvisor is to automatically build such a flowchart, where each node is a question. The user follows a path by answering those questions, and stops upon reaching a recommendation. We modeled our problem as a decision tree, and showed how to build the flowchart by adapting the classical approach used for decision trees.

The second goal of ShoppingAdvisor is to produce a suitable recommendation at each node of the flowchart, so that the user may stop answering at any time. We employed a learning-to-rank approach to provide a list of top-k recommendations for the user given the current answers. This approach allows our framework to implicitly leverage the correlations between user preferences and product attributes, thereby providing a solution for the typical cold-start problem found in pure collaborative filtering. Furthermore, it provides a simple way to interpret the recommendations.

We compared our system with a baseline, and demonstrated the effectiveness of our approach. We showed how collaborative filtering methods such as k-NN benefits from feature selection by ShoppingAdvisor. We also provided examples that confirm the easy interpretability of the recommendations provided.

We intend to evaluate the applicability of our proposed framework to other novel applications, e.g., explore hashtags in tweets to guide news article recommendation, etc.

7. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *TKDE*, 17(6):734–749, 2005.

[2] L. Ardissono and A. Goy. Tailoring the Interaction with Users in Web Stores. *UMUAI*, 10(4):251–303, 2000.

[3] S. Balakrishnan and S. Chopra. Collaborative ranking. In *WSDM*, pages 143–152, 2012.

[4] M. Bilgic and R. J. Mooney. Explaining Recommendations: Satisfaction vs. Promotion. In *Beyond Personalization, workshop*, 2005.

[5] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *UAI*, pages 43–52, 1998.

[6] S. Clémençon and N. Vayatis. Tree-based ranking methods. *T-IT*, 55(9):4316–4336, 2009.

[7] W. W. Cohen, R. E. Schapire, and Y. Singer. Learning to order things. *J. Artif. Int. Res.*, 10(1):243–270, 1999. ISSN 1076-9757.

[8] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *JMLR*, 4:933–969, 2003.

[9] R. T. Grenci and P. A. Todd. Solutions-driven marketing. *CACM*, 45(3):64–71, 2002.

[10] J. L. Herlocker, J. A. Konstan, and J. Riedl. Explaining collaborative filtering recommendations. In *CSCW*, pages 241–250, 2000.

[11] C.-N. Hsu, H.-H. Chung, and H.-S. Huang. Mining Skewed and Sparse Transaction Data for Personalized Shopping Recommendation. *Machine Learning*, 57(1):35–59, 2004.

[12] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich. *Recommender Systems: An Introduction.* Cambridge University Press, 2010. ISBN 9780521493369.

[13] T. Joachims. Training linear SVMs in linear time. In *KDD*, pages 217–226, 2006.

[14] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing*, 7(1):76–80, 2003.

[15] B. Peintner, P. Viappiani, and N. Yorke-Smith. Preferences in interactive systems: Technical challenges and case studies. *AI Magazine*, 29(4):13–24, 2008.

[16] P. Resnick, N. Iacovou, M. Sushak, P. Bergstrom, and J. Riedl. GroupLens: An open architecture for collaborative filtering of netnews. In *CSCW*, pages 175–186, 1994.

[17] J. B. Schafer, J. Konstan, and J. Riedi. Recommender systems in e-commerce. In *EC*, pages 158–166, 1999.

[18] M. Stolze and M. Ströbel. Recommending as Personalized Teaching. In *Designing Personalized User Experiences in eCommerce*, volume 5, pages 293–313. Springer, 2004.

[19] Z. Tu. Probabilistic Boosting-Tree: Learning Discriminative Models for Classification, Recognition, and Clustering. In *ICCV*, pages 1589–1596, 2005.

[20] P. Viappiani, B. Faltings, and P. Pu. Preference-based search using example-critiquing with suggestions. *JAIR*, 27(1):465–503, 2006.