Preference elicitation for interface optimization

Preference Elicitation for Interface Optimization

Krzysztof Gajos and Daniel S. Weld
University of Washington
Seattle, WA 98195, USA
{kgajos,weld}@cs.washington.edu

ABSTRACT
Decision-theoretic optimization is becoming a popular tool in the user interface community, but creating accurate cost (or utility) functions has become a bottleneck — in most cases the numerous parameters of these functions are chosen manually, which is a tedious and error-prone process. This paper describes ARNAULD, a general interactive tool for eliciting user preferences concerning concrete outcomes and using this feedback to automatically learn a factored cost function. We empirically evaluate our machine learning algorithm and two automatic query generation approaches and report on an informal user study.

ACM Classification D.2.2 [**Design Tools and Techniques**]: User Interfaces, H1.2. [**Models and principles**]: User/Machine Systems

General Terms Algorithms, Human Factors

KEYWORDS: optimization, utility elicitation, active learning

INTRODUCTION
Recent years have revealed a trend towards increasing use of *optimization* as a method for automatically designing aspects of an interface's interaction with the user. In most cases, this optimization may be thought of as *decision-theoretic* — the objective is to minimize the expected cost of a user's interactions or (equivalently) to maximize the user's expected utility. For example, the BUSYBODY system [19] mediates incoming notifications with a decision-theoretic model of the expected cost of an interruption, which is computed in terms of the user's activity history, location, time of day, number of companions, and conversational status. The LINEDRIVE system [1], generates graphical representations of driving directions by using optimization to find the optimal balance between readability and fidelity to the original shapes, directions and lengths of the individual road segments. The RIA system [27, 28] chooses the best answer to a user query by optimizing a cost function which offsets content quality with quantity and other factors. There are many other examples [21, 16, 18, 25, 3, 10, 11], but as a final case our SUPPLE system [13, 12] uses optimization to generate concrete user interfaces from declarative specifications of a user interface, the target device and the user model.

While decision-theoretic optimization provides a powerful, flexible, and principled approach for these systems, the quality of the resulting solution is completely dependent on the accuracy of the underlying cost (utility) function. Unfortunately, defining a good cost function is a complex, time-consuming, and error-prone task. While domain-specific learning techniques have been used occasionally, most practitioners parameterize the cost function and then engage in a laborious and unreliable process of hand-tuning.

This problem is not unique to user interfaces. Economists, medical researchers, and people in the fields of decision support and artificial intelligence have developed methodologies for *utility (*or *preference) elicitation*. But this work is still young, and furthermore, there are significant differences when preference elicitation is applied to the user-interface context. One method uses *gamble queries*, asking the user whether they prefer choice X to a mixture of a probability p chance of the ideal outcome and a $(1-p)$ chance of the worst outcome. While this technique has elegant theoretical properties, we do not believe users can coherently report their preferences with respect to probability distributions over interface properties. In some approaches users are asked to pick numbers from an infinite, continuous range. For example, Horvitz and Apacible propose asking users to assign dollar amounts to quantify the cost of various outcomes [17]. We question this approach because it is so abstract; users will likely report more accurate results in a concrete setting where their context is visible. Other methods ask questions about one parameter at a time, ignoring the interactions between the parameters. Because these factors are *not* independent, undesirable outcomes are likely to occur.

This paper presents a fully implemented interactive system[1] which uses a combination of two complementary interaction techniques to solicit user feedback on concrete user interfaces or their parts. It thus frees the humans from having to reason about numerous and unintuitive parameters, probabilities or monetary values of different tradeoffs and moves the discourse into the space of concrete outcomes. We make the following contributions:

- We identify two types of interactions by which users can conveniently provide feedback regarding their preferences:

[1] ARNAULD is named after Antoine Arnauld (1612–1694), a French philosopher who first applied the principle of Maximum Expected Utility [2].

example critiquing acquires implicit feedback from certain types of user actions within the interface, and *active elicitation* explicitly asks the user to compare two alternatives.
- We investigate question-generation algorithms, proposing two solutions: one based on analyzing concrete outcomes and the other performing computation entirely in the parameter space. We evaluate both empirically, demonstrating fast learning for each.
- We describe a new machine learning algorithm, based on maximum-margin techniques, for estimating the user's cost function from the feedback provided. Our empirical evaluation shows that our method is substantially faster than a Bayesian approach yet produces equivalent results.
- We argue that ARNAULD is a general tool, applying to most optimization-based systems. We demonstrate ARNAULD's utility by integrating it with SUPPLE, and explain how ARNAULD can be used with RIA and context-sensitive notifications.
- We present the results of a preliminary user study showing that ARNAULD is a useful tool for designers with variety of backgrounds.

Our paper continues by stating our guiding desiderata and introducing SUPPLE as a running example. We discuss two types of interactions which can be used as a source of preference feedback by ARNAULD. We present an algorithm for learning cost functions from this feedback, and two algorithms for generating queries. An informal user study shows that ARNAULD largely meets the desiderata and achieves user acceptance. We argue that our approach is general, explaining how ARNAULD may be applied to a variety of other optimization-based systems. We conclude with a discussion of related work and our conclusions.

DESIDERATA

Utility theory stems from the notion of *preferences* over *outcomes* [20]. Outcomes result from the choices of the system or the user, and the user's preferences are defined in terms of an order, $\succ$, over these outcomes. This preference order can be defined in terms of a real-valued *cost function*, **$**, over outcomes — one prefers o over o' (written $o \succ o'$) if and only if o has lower cost: $\$(o) < \$(o')$. While people are capable of specifying preferences between concrete outcomes, they have difficulty articulating a real-valued cost function. Because applications typically need an actual cost function, we seek a tool which can automatically construct a good one from a set of concrete preference examples.

Depending on the application, this cost function might denote many things. In LINEDRIVE, cost measures the estimated cognitive difficulty of a user reading a map [1]. In the Automated Travel Assistant, ATA, cost reflect the undesirability of a sequence of airline flights [21]. In the Responsive Information Architect, RIA, cost measures the degree to which a user is unsatisfied with the systems answer to her query [27]. And in SUPPLE, cost measures the difficulty of using different combinations of widgets for UI tasks.

Typically, these cost functions are defined in parameterized form, and the parameters represent various tradeoffs. Many decision-theoretic systems make strong claims to extensibility. However, while it may be easy to add new components to these systems, it is often hard to pick cost parameters which correctly integrate the new features. This creates a possible source of error that the designers of the original platform cannot control. In general, it is very hard to choose parameter values in a way which accurately orders a wide variety of outcomes. Furthermore, many of these parameters reflect *subjective* judgments, which are potentially controversial. Since these parameters need to be set by humans, certain values (or constraints on values) may be set incorrectly or inconsistently — yet the overall system must degrade gracefully in the face of such contradictions.

One of the advantages of the optimization-based approach is the potential to *personalize* the application for individual users, simply by defining separate cost functions for each user. Thus, a tool facilitating the definition of cost functions may do more than reduce the burden on developers — it could enable wide-scale personalization capabilities.

In summary, choosing the appropriate weights for a parameterized cost function has long been acknowledged to be challenging. We conclude that any tool which facilitates the elicitation problem must satisfy the following desiderata:

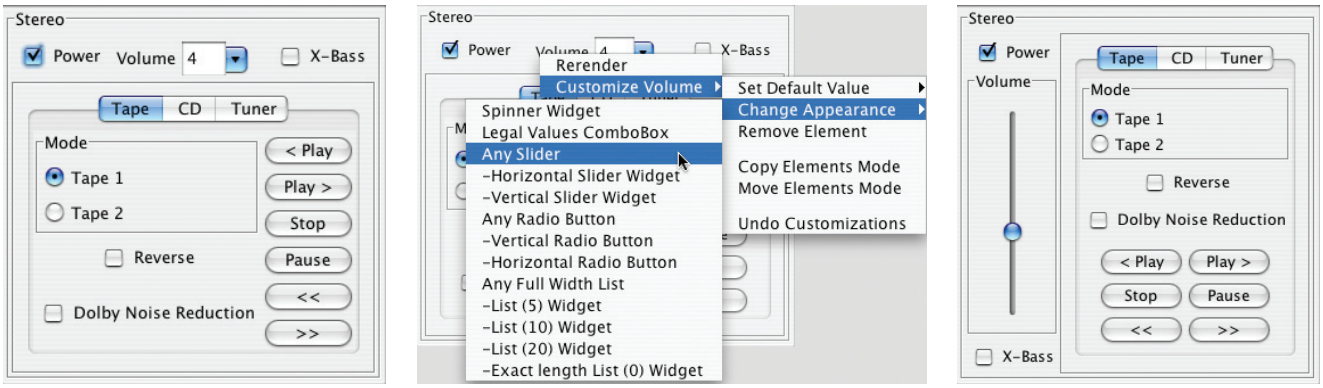
1. Make it *fast* and *easy* for both developers and end-users to find good values for the weighting parameters.
2. Output weights which are *correct* and *robust*, even in the face of erroneous or inconsistent user feedback.

RUNNING EXAMPLE

ARNAULD is a general system, which can be used to define a cost (utility) function for most optimization-based interface applications. However, in order to make our discussion concrete, we explain preference elicitation in the context of a specific application: SUPPLE.

SUPPLE is a fast and efficient UI toolkit for ubiquitous applications, that automatically generates user interfaces (at run time) which are optimized for the specific hardware platform, device characteristics and owner's individual working style.

SUPPLE takes three inputs: a *functional specification* of the interface, a *device model* and a *user model*. The functional specification defines the *types* of data that need to be exchanged between the user and the application. The device model describes which widgets are available on the device and provides a cost function, which estimates the user effort required to manipulate these widgets with the interaction methods supported by the device. Finally, a user's typical activities are modeled with a device- and rendering-independent *user trace*. SUPPLE's rendering algorithm combines constraint propagation with branch-and-bound search, to find the "optimal" concrete rendering of the interface. Optimality is decided with respect to a cost function, derived from the information contained in the device model. The original formulation of SUPPLE's cost function casts it as a weighted sum of per-element cost functions, that reflect how well a particular concrete widget supports the operations usually performed on a particular abstract element of the functional specification. The weights are derived from the user model. The per-element cost functions, however,



(a) (b) (c)

Figure 1: Example critiquing in SUPPLE. (a) Initial rendering of a stereo controller. (b) The user asks SUPPLE to use a slider instead of a combo box for controlling the volume. (c) The new rendering. Information recorded from this interaction is used by ARNAULD to further improve SUPPLE's cost function for future renderings.

were originally formulated as computationally opaque complex functions. We have recently reformulated them as sums of weighted *factors* f:

$$\$(render(e)) = \sum_{k=1}^{K} u_k f_k(render(e)) \qquad (1)$$

Here e denotes an element of the functional specification and $render(e)$ denotes the concrete widget which the rendering function $render$ has chosen for e. Lastly, $f_k(render(e))$ is one of the K factors comprising the cost function and u_k is a corresponding weight. Factors can be either binary indicator functions reflecting presence or absence of a property, or they can return a non-negative[2] real value to reflect the magnitude of a property. For example, there are binary factors to indicate that a widget is a checkbox or that it is a spinner assigned to represent a numerical value. Continuous-valued factors include, for example, the downsampling factor for images and maps, or the number of lines by which a list widget is too short to accommodate the expected number of elements.

SUPPLE has over forty factors, and the corresponding weights had to be chosen manually to yield the desired solutions. This process was tedious and error prone and had to be repeated every time a new kind of widget was added to any of the platforms supported by SUPPLE.

USER INTERACTIONS

In this section we demonstrate two classes of interactions for eliciting users preferences, which we encode as inequalities of the form $o \succ o'$, indicating that user prefers *outcome* o to o'. These preferences are used, as the next section explains, to calculate a real-valued cost function, which is maximally consistent with the preferences and which can be used to generate good interfaces in the future.

The first elicitation method, called example critiquing, is well suited for both developers and end users, because such critiques may often be derived implicitly from normal user actions. The second, active elicitation, is a computer-driven questioning process where the human operator is asked to make comparisons between different pairs of automatically chosen interfaces.

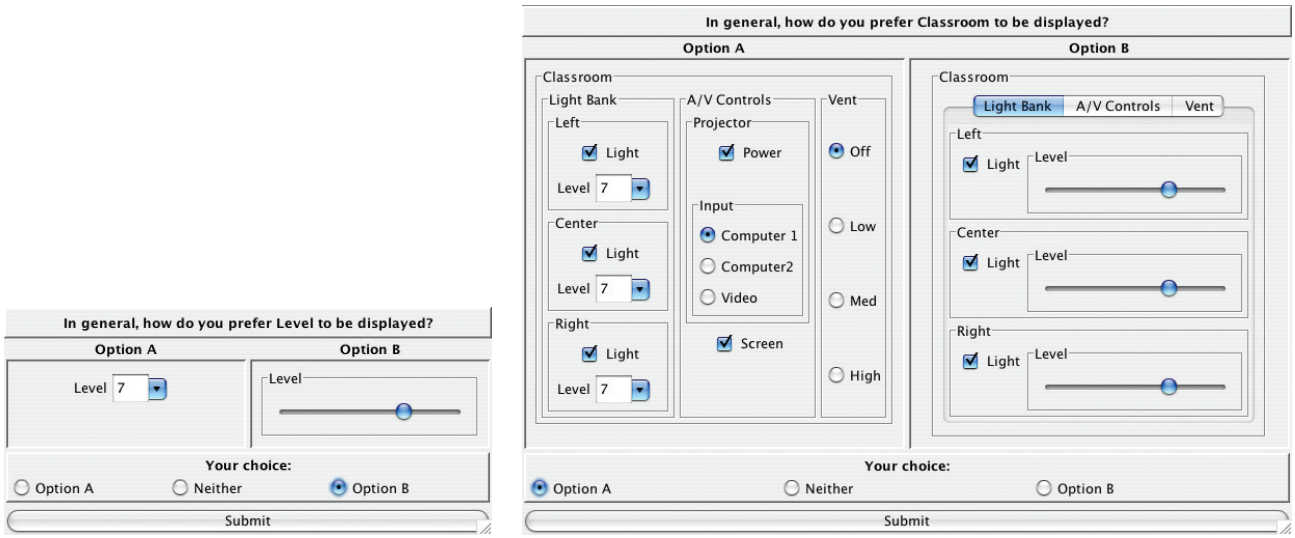
[2]Although SUPPLE requires factors to return non-negative values, ARNAULD imposes no such restrictions.

Example Critiquing

In many interfaces, the user's natural interactions provide information on their cost model; in these cases, one may learn the cost function, without imposing any additional burden on the user, using a process often called *example critiquing*. We demonstrate this approach in the context of SUPPLE, which has an extensive customization framework that allows both designers and users to modify the structure, behavior and appearance of the rendered user interfaces [12]. Amongst other things, SUPPLE's customization facility allows human operators to right click on any part of a (desktop computer) interface and choose from a selection of possible ways that element can be rendered (Figure 1). We use ARNAULD to automatically record all such customization requests and use them to further improve the parameters of SUPPLE's cost function. We note, however, that often changes to one small part of a user interface (*e.g.*, causing light intensities to be rendered with sliders instead of combo boxes) may cascade, causing changes elsewhere (*e.g.*, causing the top level pane of the interface to be split into separate tabs, thus making the navigation through the interface more difficult). Thus a local improvement might lead to a global decrease in quality. In order to correctly learn the parameters of the objective function (i.e., to account for these tradeoffs), it is important to also record whether or not the user considers the resulting interface a global improvement over the previous version. For this reason, when ARNAULD detects that critiquing has caused non-local changes to the outcome, it requests feedback on the global result in addition to the local choice.

Active Elicitation

In some cases, the user's natural actions provide insufficient feedback to learn a cost model. In these cases, the computer must facilitate preference elicitation by generating information-gathering questions to ask the user. Some researchers advocate ranking multiple outcomes, but for simplicity, we use binary queries, presenting users with pairs of outcomes and asking which of the two (if either) they prefer.



(a) (b)

Figure 2: Two consecutive steps in the active elicitation process. (a) ARNAULD poses a *ceteris paribus* query, showing two renderings of light intensity control in isolation; this user prefers to use a slider. (b) Realizing that the choice may impact other parts of the classroom controller interface, ARNAULD asks the user to consider a concrete interface that uses combo boxes for light intensities but is able to show all elements at once, and an interface where sliders are used but different parts of the interface have to be put in separate tab panes in order to meet the overall size constraints.

To keep questions simple for the user, we believe it is important to emphasize so called *ceteris paribus* (everything else being equal) queries, where users are asked to consider two small differences in isolation (Figure 2(a)). However, whenever the local change between the two options causes cascading, global changes, we also ask the user to compare the entire outcomes (Figure 2)(b).

The key challenge for active elicitation is determining the best questions to ask the user. We explain our question-generation algorithm in two sections, but first we must explain how we use the *answers* to these questions to learn a cost function.

LEARNING FROM USER FEEDBACK

Now that we have described how to elicit a set of user preferences over pairs of outcomes (*e.g.*, o and o'), we formally represent these preferences as constraints and use them to infer the best values of the cost function's underlying parameters. We observe that in every system we have studied, the cost (or utility) of any outcome can be decomposed linearly with respect to any class of parameters and consequently represented as

$$\$(o) = \sum_{k=1}^{K} u_k f_k(o) \tag{2}$$

where $f_k(o)$ is a *factor* in the cost function. These factors represent the presence, absence or intensity of various properties of the solution. For example, in SUPPLE, one factor is used to indicate that a particular widget is a spinner while another reflects how much an image was scaled from the desired size. In a probabilistic system like [17], these factors correspond to probabilities. The u_k is a non-negative weight corresponding to factor f_k — it is the values of these weights that must be estimated.

Turning Feedback Into Constraints

We can quite naturally convert a user's preferences over a pair of outcomes o and o' into a constraint by noting that if $o \succ o'$ then the cost of o' must be greater than that of o:

$$\sum_{k=1}^{K} u_k f_k(o') \geq \sum_{k=1}^{K} u_k f_k(o) \tag{3}$$

Since the factor functions always return the same values for the same outcomes, the goal is to find weight values, u_k, which satisfy the constraints. Consider the simple example of Figure 2(a). In the rendering on the left hand side, all factors are set to 0 except $f_{\textit{combo box}}$ and $f_{\textit{combo box for number}}$, which are set to 1, indicating that a combo box has been used and it was used to provide access to a numeric state variable. On the right hand side, only two factors again are set to 1: $f_{\textit{slider}}$ and $f_{\textit{horizontal slider}}$. Thus Equation 3 becomes:

$$u_{\textit{combo box}} + u_{\textit{combo box for number}} \geq u_{\textit{slider}} + u_{\textit{horizontal slider}} \tag{4}$$

While it is clear how to formally interpret user's responses to the active elicitation queries, the example critiquing feedback deserves additional attention. Note that if the user is presented with an interface such as the one in right pane of Figure 2(b) but she wishes SUPPLE would render it as shown in the left pane of Figure 2(b), she can proceed in two possible ways: she can use SUPPLE's critiquing facility (as in Figure 1) to tell it not to use the tab pane to organize the top level of the interface, or she can request that the light intensities be rendered with combo boxes instead of sliders.

In the first case, her feedback reflected her overall preference for avoiding tab panes wherever possible. In the second case, she downgraded the widget for the light intensity in order to improve the overall interface. In order to formalize

the preference as a correct constraint, ARNAULD needs to be able to determine which of the two was intended. To perform this disambiguation, we assume that the vast majority of the feedback events will be of the first kind, where the user's preference for one widget over another will apply in any situation.[3]

In some cases (*e.g.*, [9]) it may be reasonable to assume that if the user critiques an example by expressing a preference for one outcome over another, then the preferred outcome can be interpreted as being better than *all* other possible outcomes (rather than just the one it was explicitly stated to dominate). Since this interpretation is not always correct, ARNAULD takes the narrow interpretation.

Defining the learning problem

Our objective is to find values for each weight, u_i, such that all constraints (or as many as possible) are satisfied. To make sure that the cost function is robust, we also wish to maximize the degree of satisfaction of the constraint, c_i. This degree of satisfaction is called the *margin* and denoted m_{C_i}:

$$m_{C_i} = \sum_{k=1}^{K} u_k f_k(o'_{C_i}) - \sum_{k=1}^{K} u_k f_k(o_{C_i}) \tag{5}$$

One way to find the weight values is with the Chajewska, et al. approach [8], which finds the Bayesian estimate of these values given user's responses. This method relies on Metropolis sampling, which is too slow for interactive use, as we demonstrate later. Consequently, we propose a faster and more direct technique which explicitly tries to maximizes the margin through linear optimization.

Maximum Margin Learner

Our algorithm is based on maximum-margin methods from support-vector machines [6]. The maximum-margin approach finds the factor-weight values which satisfy the most constraints. Furthermore, these values are chosen so as to maximize the difference between the two sides of the constraint inequality. At the same time, explicit *slack variables* are used, which allow the optimization to succeed even in the presence of contradictory or hard-to-satisfy constraints.

Formally, we reformulate the constraints from Equation 3 to include the addition of a shared margin variable m and a per-constraint slack variable ξ_i:

$$\sum_{k=1}^{K} u_k f_k(o'_{C_i}) - \sum_{k=1}^{K} u_k f_k(o_{C_i}) \geq m - \xi_i \quad \text{for all } c_i \tag{6}$$

Following standard practice, we constrain the margin to be greater or equal than 1. Note that this causes $\sum \xi_i$ to be an upper bound on the number of violated constraints. Thus we formulate an objective function to be of the form:

$$\text{maximize} \quad m - \alpha \sum \xi_i \tag{7}$$

Where α stands for a parameter that controls the tolerance of the learner to a small number of violated constraints or constraints satisfied by a margin smaller than m.

[3]Another approach would be to provide a user interface mechanism for disambiguating the two interpretations or attempting to automatically predict the user's intention.

The set of parameters returned by this optimization satisfies the maximum number of constraints and satisfies them by the maximum possible margin. Further notice, that all the constraints and the objective function are linear and therefore we can solve this problem using very fast linear programming techniques.

Unlike a Bayesian learner [8], a max-margin algorithm can utilize prior knowledge only if it is presented in the form of constraints — it does not allow the knowledge to be summarized in the form of concise statistics. To address this shortcoming, we extend our approach so that it can take into account prior knowledge. Suppose we have a prior belief (*e.g.*, factory preset) that a weight for the k^{th} factor is u'_k, then we want any new value u_k to be *close* to the prior value. We may formalize this desire as the following optimization problem:

$$\begin{aligned} &\text{minimize} && m' + \alpha' \textstyle\sum \xi'_k \\ &\text{subject to} && |u_k - u'_k| \leq m' + \xi'_k \ \text{ for all } k \in [1 \ldots K] \end{aligned} \tag{8}$$

Since this constraint is not linear, we rewrite it as a pair of linear constraints:

$$\begin{aligned} u_k - u'_k &\leq m_{prior} + \xi'_k \\ u'_k - u_k &\leq m_{prior} + \xi'_k \end{aligned} \tag{9}$$

Finally, we combine the prior knowledge and the knowledge from the newly acquired constraints as our final objective function:

$$\begin{aligned} \text{maximize} \quad & \frac{N}{N+N'} \left(m - \frac{1}{\sqrt{N+K}} \sum_{i=1}^{N} \xi_i \right) - \\ & \frac{N'}{N+N'} \left(m' + \frac{1}{\sqrt{N+K}} \sum_{k=1}^{K} \xi'_k \right) \end{aligned} \tag{10}$$

We have replaced the previously undefined α and α' from Equations 7 and 8 with one over the square root of the total number of constraints representing user's preferences (including both newly acquired preferences and the ones contained in the prior). Additionally, we have weighted the parts of the objective function corresponding to the new constraints and those in the prior. N' stands here for the *equivalent sample size* — setting N' to 10 would mean that the knowledge included in the prior should be treated with as much weight as 10 new constraints. This allows us to encode our certainty about the knowledge represented by the prior.

GENERATING QUERY EXAMPLES

As discussed previously, one way in which ARNAULD elicits feedback from the user is by presenting her with two alternative outcomes and asking for a statement about her preference. It should be no surprise that the choice of questions asked might affect the number of questions necessary to learn a good cost function. For example, repeatedly asking very similar questions will provide too little information for the learning algorithm to make much progress. Unfortunately, choosing the *optimal* question from the perspective of expected information gain is intractable [4]. Furthermore, complex questions are usually harder for users to answer than simple ones. Specifically, asking a user to compare

two vastly different outcomes may result in an ambivalent response ("apples *vs.* oranges"), because of the vast number of tradeoffs. Thus, we seek heuristic query-generating algorithms, that will ask simple and informative questions.

We assume that the user provides a set of *training examples*, each being a sample input to their application. In the case of learning the cost function for a SUPPLE device description, each training example is a functional specification of an interface and a screen size constraint. For LINEDRIVE, an example would be a route, and for RIA an example would be a set of information to be presented to the user. Note that each example may be decomposed into primitive *constituents*, corresponding to the smallest portion of the example that can be used as an argument to the cost function. With SUPPLE each element (node in the functional-specification tree) is a constituent (for which a widget or a layout must be assigned and costed). In order to generate simple, localized questions, we restrict attention to queries about single constituents.

Weight-Based Question Generation

Our weight-based ranking algorithm applies to any application that can enumerate the space of queries. Thus, when applied to SUPPLE, ARNAULD enumerates all pairs of different renderings of any *single* element in the interface specification of an example (*i.e.*, varying either a single widget or the layout of one portion of the interface). These queries are ranked via a computation in the space of possible factor weights, and the best is chosen.

The key observation is as follows: if the cost function is defined in terms of K weights, then the constraints so-far recorded define a sub-region of this space, and the region's centroid denotes the K weight values, which define the current, best cost function. Every candidate query, $(o \; ? \; o')$, defines a *hyperplane* in this space: the set of weight values that cause $\$(o) = \$(o')$. Hence, an answer to the query (*e.g.*, a statement of the form $o \succ o'$) transforms the query into a new constraint: the half-space to one side of the hyperplane is consistent with the constraint while the other is not. It has been shown that if a question is very good, then its corresponding hyperplane falls very close to the region's centroid. This is intuitive, because no matter what the user answers, the region containing the true cost function will be cut nearly in half [24, 26].

But there may be several hyperplanes, each passing close to the centroid, but oriented differently from each other, perhaps even orthogonal; which is best? Intuitively, one would like the constrained region to be approximately spherical. Thus, if the region is already narrowly constrained along one dimension, the next cut should be orthogonal. Put another way, on average the questions should equally constrain *every* dimension of the weight space.

Our algorithm exploits these intuitions using a heuristic encoded as a modified distance function. We rank queries by the distance between the corresponding hyperplane and the centroid, but the distance function scales each dimension by a value proportional to the number of constraints cutting along that dimension. While our method is heuristic, it is fast to compute and effective.

Outcome-Based Question Generation

This algorithm applies to applications satisfying the condition that each factor of the cost function corresponds to a single constituent (hence the cost of an example may be written as the sum of the costs of the example's constituents). For instance, in SUPPLE one may create a new example by taking a constituent (element) from one functional specification and considering that element in isolation (*i.e.*, computing the best rendering of an interface with a single widget corresponding to that element).

Our outcome-based ranking algorithm first enumerates all trainings example and uses the current cost function to choose the best outcome for each example as a whole. Next, for each constituent e it notes what partial outcome o_e was assigned to it in the solution that addressed the example as a whole. Then, it iterates over all possible partial outcomes o'_e that could have been assigned to e and computes *score*$(o'_e) = \$(o_e) - \$(o'_e)$. Note that if *score* ends up being positive, then it means that o_e is a suboptimal local outcome for e, and it was presumably chosen in order to allow for a better solution in a different part of the interface. If the *score* is negative then o_e was the better choice according to the current cost function. Our algorithm chooses query $(o'_e \; ? \; o_e)$ where *score*(o'_e) is the highest. For positive values of *score*, this challenges the tradeoff that the interface generator made. For slightly negative values of *score*, it asks about a close, second-best, local outcome — thus helping to either spot incorrect choices or widening the margin between the two options.

EVALUATION

We conducted a sequence of experimental studies to measure how well ARNAULD meets the desiderata, to evaluate our learning method, and to compare our two question-generation algorithms. Recall that we presented two requirements: 1) Make it *fast* and *easy* for both developers and end-users to find good values for the weighting parameters; and 2) Output weights which are *robust*, even in the face of inconsistent or erroneous user feedback.

Informal User Evaluation

The first desideratum concerns ease of use for both developers and end-users, which we evaluate in a pilot user study. In this study we observed four users using ARNAULD connected to SUPPLE. Two of the four users represented developers — they were experienced contributors to the SUPPLE project, but they were unfamiliar with ARNAULD. The other two users represented "sophisticated users" — experienced programmers who were unfamiliar with ARNAULD and who had only cursory knowledge of SUPPLE. While these were not typical end users, we do not believe ARNAULD is quite mature enough for use by the novices. Soon, however, we hope to perform studies on true end users.

We gave the "sophisticated user" group an overview of SUPPLE's behavior and its overall architecture followed by instruction into the operation of ARNAULD, including the ability to switch between the question generation (driven by the domain-dependent algorithm) and example critiquing modes. We then asked them to create a cost function for a new SUPPLE interface platform, using ARNAULD until they

were satisfied that SUPPLE was creating reasonable concrete user interfaces.

We treated the "developer" group slightly differently, by first asking them to *manually* assign values to all parameters in the cost function. (One of the subjects had done this in the past but for a different set of parameters describing a different interface platform.) We then instructed them in the use of ARNAULD, and asked them to use it to construct the cost function over again, from scratch. In addition to monitoring their overall use of and satisfaction with ARNAULD, we also recorded each preference response and compared them to those that their hand-crafted cost functions would have generated. Interestingly, the subjects disagreed with their hand crafted preference model 26% and 22% of the time respectively. Additionally, in a further 12% and 9% of the cases their hand crafted cost function disagreed less strongly (*i.e.*, one expressed a preference while the other considered the two options to be equivalent). This certainly shows how hard it is to manually define a cost function.

All four subjects felt that ARNAULD was a much easier and more accurate way to find a good set of cost parameters. They attributed the improvement to being able to improve the function in a rapid, iterative manner with immediately visible feedback. The "sophisticated users" felt confident that they could produce a robust set of parameters, while the developers felt that the tool let them arrive at the right result more quickly and often helped them address trade-offs, which they had missed when manually setting parameter values.

All users were able to produce robust cost functions in 10 to 15 minutes, while the SUPPLE developers took 20 to 40 minutes to construct cost functions by hand. All users found the tool generally easy to use, but some of them pointed out that when ARNAULD was asking them to compare more complex outcomes it was sometimes difficult to pinpoint all the differences (perhaps these should be color coded in the future?). On the other hand, less than 10% of their answers to the automatically generated queries were ambivalent, indicating that the differences between presented alternatives were usually easy to evaluate. Two of the users also commented that a closer blending of the implicit and explicit approaches would be desirable *i.e.*, being able to modify the examples provided in the active elicitation or being able to more quickly alternate between the two modes. One user requested an *undo* button which would reverse a critique or answer and also remove the corresponding input from the learner.

Comparison of Learning Methods

The first desideratum requires fast and easy operation, so we consider the speed and accuracy of our learning algorithm, comparing it against the Bayesian approach with Metropolis sampling, proposed by [8]. When the Metropolis sampling approximation was run long enough, both methods produced essentially the same parameter weights. But our maximum-margin approach not only produced exact answers, it was also much faster.

Because Metropolis sampling is an anytime, approximation method, one may trade solution quality for speed. Perhaps it generates relatively good answers quickly? Figure 3 shows

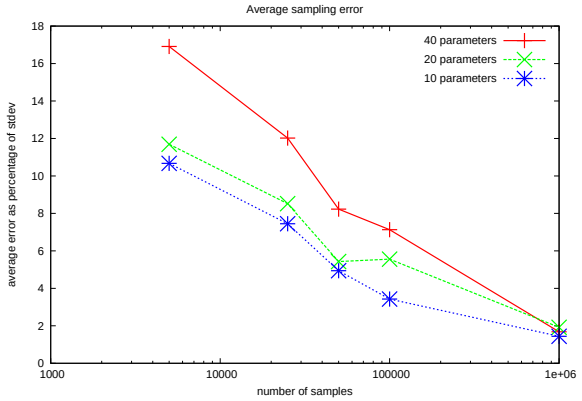


Figure 3: The sampling error of the Bayesian approach diminishes with a huge number of samples, but is thus too slow for interactive use: accurate answers require 10^6 samples (taking 40 seconds to compute), while our maximum-margin algorithm returns exact answers in less than 200ms.

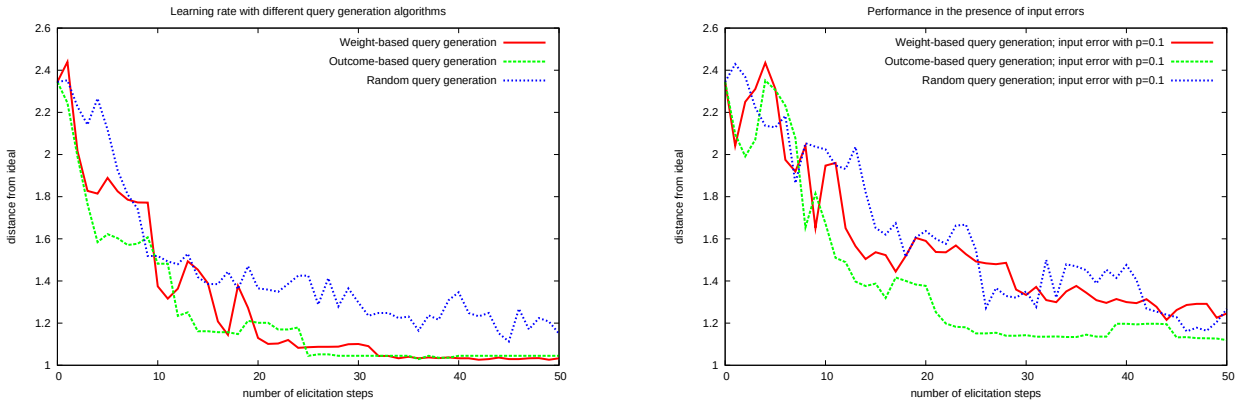
the average error in the Metropolis sampling estimates, as a percentage of the standard deviation of the underlying distribution, for the cases where we need to estimate 10, 20 and 40 variables at a time. Our implementation of the Metropolis algorithm can sample 25,000 times per second. But our maximum-margin algorithm can produce an exact solution in 40-200ms on a standard desktop computer.[4] Thus, we can draw only 5,000 samples in the time used by the maximum-margin learner. When estimating 40 variables at once (as is the case with SUPPLE), this sample-size produces an average error of 17% of the standard deviation. The error goes down to 12% and 8%, if we are willing to wait 1 or 2 seconds for the result, respectively. The error stays above 2% unless a million samples are drawn, which takes about 40 seconds. We conclude that maximum-margin is the clear choice for the fast, interactive use required by our first desideratum.

Comparison of Query-Generation Methods

Another aspect of the first desideratum concerns *how many* questions need to be answered. Thus, we evaluate our query-generation algorithms to see how many interactions are required in order to find a good set of values for the 40 parameters of the SUPPLE cost function.

In these experiments we simulate the user interacting with ARNAULD's automated querying interface (but not with example critiquing). We use a cost function defined by a previously validated set of weights (denoted the *target* function) in order to simulate a user's responses. When presented with a query, the simulator registers a preference for the option which has lower cost according to the target function. After each response, ARNAULD updates the weights of the cost function *being learned*. The quality of the learned cost function is assessed at each step by using it to generate several concrete interfaces. The target function is then used to score those interfaces. We compute

[4] Apple Dual G4, 877MHz, 1.25GB RAM



(a) (b)

Figure 4: Rate of learning (all results averaged over 10 runs). The y-axis reflects how far from the ideal are the interfaces generated using the currently learned parameter values (see text for detailed explanation) (a) for different query generation strategies; (b) in the presence of input errors: at each step there was a 0.1 chance that the simulated user would give an answer opposite from the one intended.

$\$(render(\textit{test interfaces}, \$'))/\$(render(\textit{test interfaces}, \$))$

where $\$$ is the target cost function and $\$'$ is the function being learned. In other words, we compute the ratio of the true cost of the rendering guided by the learned cost function, and the true cost of the rendering guided by the true cost function. This quotient reflects how far from the ideal are the interfaces generated using the currently learned parameter values[5]. To make the testing fair, we use different interface specifications to generate queries and to evaluate the results. Queries are based on the classroom and Amazon search interfaces (see [12] for screenshots of all the interfaces mentioned in this section), while evaluation was based on a stereo controller, an email client and an interactive map-based interface.

Figure 4(a) shows how the quality of the learned function improves with the number of queries issued. Our outcome-based and weight-based algorithms resulted in learned functions performing almost identically to the target function after about 25 queries. As a control, we also evaluated a random query generator, which resulted in a much slower learning rate. In summary, both of our query-generation algorithms require sufficiently short interactions to satisfy desideratum one.

Sensitivity to Input Noise

Next, we evaluate ARNAULD against our second desideratum: are the results robust even in the presence of input errors and inconsistencies? For each query generation algorithm, we repeated the experiment from the previous subsection, observing how the system behaved in the face of 5% and 10% probability of an input error at each step (*i.e.*, a situation where the response suggested by the target function was flipped to its opposite). This simulates the user providing inconsistent input or making a mistake. Figure 4(b) shows the results for the 10% error probability.

The outcome-based algorithm still generates an interaction where nearly optimal results are obtained after about 25 interactions. The weight-based algorithm tends to spread the queries uniformly among all parameter values meaning that after erroneous input is received, the system waits too long before asking more about the affected weights. This delay results in the algorithm performing noticeably poorer with input errors present compared to the error-free condition.

The user can still arrive at a good set of weights by switching to the example critiquing mode of interaction and directly addressing all the cases where the system makes mistakes. But we conclude that our outcome-ranked question generation is most consistent with our desiderata, and is the preferred approach, despite its slightly higher computational cost.

ARNAULD AS A GENERAL PLATFORM

Although we have only experimented with it in the context of SUPPLE, we've argued that ARNAULD will benefit most optimization-based interface systems. In this section we defend those claims by considering two different systems and showing how ARNAULD will facilitate their operation.

RIA, The Responsive Information Architect

RIA is a system that automatically decides first, what information to present to the user [27], and second, how to best match different pieces of information to different modalities [28]. In the first phase, RIA balances certain tradeoffs through the use of numerous, manually-tuned parameters reflecting different classes of information used to inform the final decision.

RIA's objective function is particularly complex with many of the different parameters being multiplied by each other.

[5]Since there exists a continuum of parameter values that may all produce the same concrete outcomes, we evaluate the learned cost function based on the cost of the results it produces (rather than the actual weight values)

Given ARNAULD's current restriction to linear systems, it could not be applied to estimate all classes of parameters at once. However, if all but one class of parameters are held constant, then the objective function would become a linear combination of the remaining class of parameters making it amenable to learning with our approach.

Both interaction methods would be appropriate in this case though a special interaction method would have to be provided to allow users to critique the information that was delivered through speech. Also for automated queries, speech could be simulated with text for faster interaction cycles.

Context-Sensitive Notifications

The system described in [17] is designed to deliver notifications to a user in a manner that is sensitive to the user's interruptability levels. It uses a decision-theoretic framework to estimate the utility of delivering a particular piece of information via the available modalities, conditioned on its estimate of the user's current interruptability. A primary focus of the work is innovative machine-learning algorithms for estimating the probability of the user being engaged in different classes of activities [18]. In contrast the actual cost parameters, indicating the relative level of distraction incurred by different modes of notification delivery, are obtained by asking users abstractly to assign different dollar amounts reflecting how much they would be willing to pay to avoid such a notification assuming certain level of interruptability.

The cost function driving this system relies on two classes of parameters: the costs of interrupting the user using different modalities, and the cost of delaying the delivery of a message in hope that the user will be more interruptable later. These two classes of parameters are combined in a linear manner and thus can be learned together by ARNAULD.

Currently, this system learns its sensor model by first recording the sensory inputs for a particular user and his environment, while also recording a video of everything that transpired during that time. In order to train the system, the user has to watch the video and annotate it with transitions between different attentional states or levels of interruptability. During the same interaction, ARNAULD could ask the user queries of the form: "If I were to deliver you an important message at this point, would you have preferred if I had used email or displayed a message on the screen?" or: "If I had a message to deliver for you, would you have preferred if I delivered it immediately by displaying it on the screen or waited till later and called you on the phone?".

The resulting system could be further tuned (personalized) after deployment, if the user was given the option on commenting on whatever notifications were delivered. Example critiquing could easily be supported for messages displayed on the screen but more specialized interfaces would be needed to comment on messages delivered via email or a real time phone call.

RELATED WORK

We place ARNAULD in the context of past work on preference elicitation, user interaction, utility-function representation, as well as the reasoning and query generation mechanisms.

User Interactions For Eliciting Preferences

Example critiquing has been used in several systems, notably in two trip planning assistants: the ATA system [21] and the work presented in [23]. In both cases, the computer presents the user with options based on the initial set of requirements and the user refines their preferences by narrowing their requirements, even concerning initially unspecified attributes like stop over locations or the airline. The authors argue that users will often be unaware of all the preferences they might have and will not think to specify them until confronted with a particularly disagreeable example. We adopt this example critiquing approach in ARNAULD.

The work of [23] further highlights the distinction between user's fundamental objectives and the means of achieving them, warning that if the system attributes do not reflect the fundamental objectives, users will be unlikely to accurately estimate the values of those attributes. In those cases where the parameters of the system match the user's objectives, direct manipulation approaches with instant feedback may be the right approach to explore the space of options. Thus, most of the systems, which can benefit from ARNAULD, use large sets of parameters, and the parameter semantics is often complex or unintuitive. For that reason in our work we avoid direct manipulation of the parameter values and focus on reasoning about concrete examples of system output.

Representation And Reasoning About User Preferences

Systems described in [23] use constraints to represent user preferences. In order to accommodate inconsistent requests, these systems use constraint solvers that allow partial satisfaction of constraints, but no mechanism is provided to select among different solutions at the same level of feasibility.

Chajewska, et al. [7] propose treating utilities as random variables. The constraints encoding user's preferences are just a means for constructing a Bayesian model to estimate values of utilities, using the Metropolis algorithm for inference. This is the approach we have used in our Bayesian learner. It is worth noting that other, potentially faster and more accurate methods are available for inference in hybrid Bayesian networks *e.g.*, [22], but their implementational complexity renders their application problematic.

Recently, an approach similar to ours was proposed for calendar scheduling [14]. Unlike our approach, it is designed for just one application and it uses standard SVM quadratic optimization for estimating the parameter values.

Finally, [5] reasons about utilities in terms of minimizing the maximum regret while representing uncertainty about the value of utility parameters as hard intervals. Without further refinement, this approach does not allow for inconsistent responses from the user.

Generating Optimal Queries

Much of the past work on generating optimal queries is based on *value of information* (see, for example, [9, 4, 15]. In our case it is computationally prohibitive because it requires hypothesizing about how each potential query will impact the presentation of future interfaces.

Our query generation algorithms are based on the ideas of [26] and [24] who present heuristic approaches in the context of SVM training that aim to select queries which would maximally reduce the size of the remaining version space.

CONCLUSIONS

Decision-theoretic optimization has been adopted by a number of researchers to automatically generate various aspects of user interfaces. Yet in almost all cases the numerous and unintuitive parameters of the optimization's objective function are chosen by hand — thus adding an unprincipled element to an otherwise very principled approach.

In this paper we address this problem by presenting and evaluating interaction methods and algorithms that allowed us to build ARNAULD, a system that interactively elicits user preferences for the purpose of automatically learning parameters of optimization-based systems.

Our interaction mechanisms and algorithms are applicable to many systems and we intend to make ARNAULD and its learning methods available for others to use. Furthermore, we would like to ensure that it is compatible with toolkits like GADGET [11] that promise to make it easy to embed optimization in user interface systems.

Acknowledgments We thank Pedro Domingos for suggesting that we look for a discriminative approach and Deepak Verma for help in formulating the optimization problem. We also thank Mary Czerwinski and James Fogarty for useful discussion and Donald Patterson, Desney Tan, William Pentney and the anonymous reviewers for helpful comments on drafts of this paper as well as our subject who provided feedback on the system. The customization infrastructure used for example critiquing was developed by Raphael Hoffmann. This research is supported by NSF grant IIS-0307906, ONR grant N00014-02-1-0932 and by DARPA project CALO through SRI grant number 03-000225.

REFERENCES

1. M. Agrawala and C. Stolte. Rendering effective route maps: Improving usability through generalization. In *SIGGRAPH'01*, 2001.
2. A. Arnauld. *The Art of Thinking*. The Bobbs-Merrill Company, inc., 1662.
3. G. J. Badros, A. Borning, and J. Stuckey. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4), 2001.
4. C. Boutilier. A POMDP formulation of preference elicitation problems. In *AAAI/IAAI*, 2002.
5. C. Boutilier, R. Patrascu, P. Poupart, and D. Schuurmans. Constraint-based optimization with the minimax decision criterion. In *International Conference on Principles and Practice of Constraint Programming*, 2003.
6. C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, 1998.
7. U. Chajewska and D. Koller. Utilities as random variables: Density estimation and structure discovery. In *UAI*, 2000.
8. U. Chajewska, D. Koller, and D. Ormoneit. Learning an agent's utility function by observing behavior. In *ICML'01*, 2001.
9. U. Chajewska, D. Koller, and R. Parr. Making rational decisions using adaptive utility elicitation. In *AAAI/IAAI*, 2000.
10. J. Fogarty, J. Forlizzi, and S. E. Hudson. Aesthetic information collages: generating decorative displays that contain information. In *UIST*, New York, NY, 2001.
11. J. Fogarty and S. E. Hudson. GADGET: A toolkit for optimization-based approaches to interface and display generation. In *UIST'03*, Vancouver, Canada, 2003.
12. K. Gajos, D. Christianson, R. Hoffmann, T. Shaked, K. Henning, J. J. Long, and D. S. Weld. Fast and robust interface generation for ubiquitous applications. In *Proceedings of Ubicomp'05*, Tokyo, Japan, 2005.
13. K. Gajos and D. S. Weld. Supple: automatically generating user interfaces. In *IUI'04*, Funchal, Madeira, Portugal, 2004.
14. M. T. Gervasio, M. D. Moffitt, M. E. Pollack, J. M. Taylor, and T. E. Uribe. Active preference learning for personalized calendar scheduling assistance. In *IUI '05*, New York, NY, USA, 2005.
15. D. Heckerman, E. Horvitz, and B. Middleton. An approximate nonmyopic computation for value of information. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(3):292–298, 1993.
16. E. Horvitz. Principles of mixed-initiative user interfaces. In *CHI '99*, New York, NY, USA, 1999.
17. E. Horvitz and J. Apacible. Learning and reasoning about interruption. In *International Conference on Multimodal Interfaces*, 2003.
18. E. Horvitz, A. Jacobs, and D. Hovel. Attention-sensitive alerting. In *UAI-99*, San Francisco, CA, 1999.
19. E. Horvitz, P. Koch, and J. Apacible. Busybody: creating and fielding personalized models of the cost of interruption. In *CSCW '04*, New York, NY, USA, 2004.
20. R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley and Sons, 1976. Republished in 1993 by Cambridge University Press.
21. G. Linden, S. Hanks, and N. Lesh. Interactive assessment of user preference models: The automated travel assistant. In *User Modeling '97*, 1997.
22. T. P. Minka. Expectation propagation for approximate bayesian inference. In *UAI '01*, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
23. P. Pu, B. Faltings, and M. Torrens. User-involved preference elicitation. In *IJCAI'03 Workshop on Configuration*, Acapulco, Mexico, 2003.
24. G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. In *ICML*, 2000.
25. S. Shearin and H. Lieberman. Intelligent profiling by example. In *IUI '01*, 2001.
26. S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.
27. M. X. Zhou and V. Aggarwal. An optimization-based approach to dynamic data content selection in intelligent multimedia interfaces. In *UIST '04*, 2004.
28. M. X. Zhou, Z. Wen, and V. Aggarwal. A graph-matching approach to dynamic media allocation in intelligent multimedia interfaces. In *IUI '05*, 2005.