End-User Debugging of Machine-Learned Programs: Toward Principles for Baring the Logic

First Author Name (Blank if Blind Review)
Affiliation (Blank if Blind Review)
Address (Blank if Blind Review)
e-mail address (Blank if Blind Review)
Optional phone number (Blank if Blind Review)

Second Author Name (Blank if Blind Review)
Affiliation (Blank if Blind Review)
Address (Blank if Blind Review)
e-mail address (Blank if Blind Review)
Optional phone number (Blank if Blind Review)

ABSTRACT

Many applications include machine learning algorithms intended to learn "programs" (rules of behavior) from an end user's actions. When these learned programs are wrong, their users receive little explanation as to why, and even less freedom of expression to help the machine learn from its mistakes. In this paper, we develop and explore a set of candidate principles for providing salient debugging information to end users who would like to correct these programs. We informed the candidate principles through a formative study, built a prototype that instantiates them, and conducted a user study of the prototype to collect empirical evidence to inform future variants. Our results suggest the value of exposing the machine's reasoning process, supporting a flexible debugging vocabulary, and illustrating the effects of user changes to the learned program's logic.

Author Keywords
Debugging, end-user programming, machine learning, principles, saliency.

ACM Classification Keywords
H5.2. Information interfaces and presentation (e.g., HCI): User Interfaces: Theory and methods.

INTRODUCTION

New programs are emerging on end-users' desktops: programs written by machines. At its heart, a computer program is a set of rules of behavior, instructing the computer how it should react to inputs. Programs such as spam filters and recommender systems employ machine learning (ML) algorithms to learn rules of behavior from a user's actions, and then uses these rules to automatically process new data. We term the resulting set of rules a *machine-learned program*. Many of these programs continuously learn as users perform actions and build rules based on probabilistic models that are specific to the given user. As a result, only *that particular user* can determine whether the machine-learned program is performing adequately and, by extension, fix it when required.

Submitted to CHI 2010. Do not circulate.

Debugging a machine-learned program, however, presents a number of challenges to end users: 1) Users often lack a clear understanding of how the program makes decisions [23], 2) it is unclear what changes users need to make to improve the program's performance [6], 3) users can usually only provide additional training examples to fix its logic [4], and 4) the impact of user changes can only be determined at run-time. One of our study participants succinctly summed up these challenges by answering the question, *"How do you think the program makes its decisions?"* with the response, *"I don't know. Magic?"*

We hypothesize that to help users effectively debug an ML program, it must provide explanations regarding its logic and allow the user to influence the program's behavior beyond providing mere training examples. Previous studies [12, 20, 21] have explored ways to overcome these challenges by exposing some of the program's logic and by allowing users to adjust parts of this logic. There has, however, been no systematic investigation into what information regarding the logic of a learned program is particularly useful for end-user debugging.

In this paper, we introduce and explore the principle of *machine-learning saliency*, which we define as *the exposure of useful and accurate pieces of information about the logic of a machine-learned program*. Building upon this concept, we investigate the information users require when debugging ML programs, as well as how users respond to programs that present them with this information.

We worked within the domain of "auto-coding" (i.e. assisting with categorizing data from verbal transcripts) while exploring ML saliency. Coding is a familiar task in the fields of psychology, social science, and HCI, and involves categorizing segmented portions of study transcripts as part of an empirical analysis process.

We chose this domain for three reasons. First, the user's debugging time is potentially much less than the time needed to do the entire task manually, because coding can be extremely time-consuming. Second, debugging is necessary because the codes and their meanings are tailored to each individual study. Finally, coding is a good environment for studying end-user debugging of machine learning because there is likely to never be enough data pertinent to any one coding project for machine learning to succeed without help. These factors suggest the domain of auto-coding will involve end users who are highly motivated to debug the ML program.

This paper explores the principle of ML saliency as a foundation for exposing the logic of ML programs to end users. We present three saliency principles and implement them through seven UI widgets in a research prototype, basing them upon prior literature and a formative study investigating how end users interpret and correct a machine-coded transcript. Finally, we explore how end users react to these saliency features and the implications for future approaches toward supporting end-user debugging of machine-learned programs.

BACKGROUND AND RELATED WORK

In order for users to efficiently debug ML logic, they must understand when and where problems exist. Researchers have found that both technical and non-technical users have difficulty understanding how ML systems generate their predictions [6, 17, 23], and would like to understand more about their reasoning [6, 8]. Machine-generated explanations that address this knowledge gap have taken a variety of forms, such as highlighting the relationship between user actions and the resulting predictions [2], detailing why a machine made a particular prediction [15], or explaining how an outcome resulted from user actions [8, 23, 24]. Much of the work in explaining probabilistic machine learning algorithms has focused on the naive Bayes classifier [1, 11] and, more generally, on linear additive classifiers [18], because explanations of these systems are relatively straightforward. More sophisticated, but computationally expensive, explanations have been developed for general Bayesian networks [13]. All of these approaches, however, are intended to explain how a specific decision was reached, instead of assisting a user who is debugging a program's overall behavior.

Some research has started to shed light on debugging of simple ML programs [12, 20], as well as more complex ensemble classifiers [22]. Initial steps have been taken to include some results of these studies in user interfaces [21], but a thorough investigation of ML logic that could and should be made salient in the machine's explanations has not been attempted.

Focusing on the particular domain of auto-coding, the TagHelper auto-coding system is highly accurate with extensive training data, but not for categories with few training examples [5]. Obtaining training data, however, is expensive because manual coding is time-consuming. Our research addresses this issue by attempting to reduce the need for training by instead fine-tuning the program's logic.

A FORMATIVE STUDY OF ML SALIENCY

To draw out principles of saliency, we conducted a formative study to investigate end users' understanding of ML logic, desired debugging feedback, and the effects of explanations exposing the program's logic. The study comprised two parts: applying the Natural Programming approach [16] we first exposed end users to a machine-learned program *without* any explanations, while the second part added explanations about the program's logic.

Our research questions were:

RQ1: Natural conversations: How do end users "naturally" give feedback to machine-learned programs?

RQ2: Mental models: What are end users' mental models of the program's logic, i.e. what do they believe is salient?

RQ3: Influence of saliency: What happens when relevant debugging information is provided to end users?

Participants and Materials

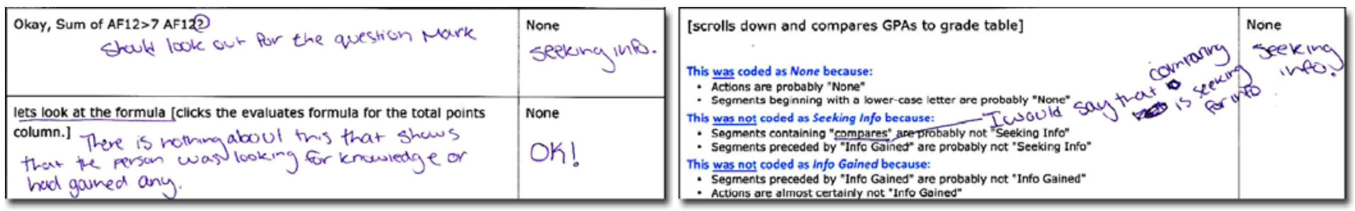
Participants included nine psychology and HCI graduate students (five female, four male). Five participants had prior experience coding transcripts and all were familiar with the domain covered in the transcript. No participants had any background in machine learning.

We obtained a coded transcript from an unrelated study and developed two different paper prototypes of an auto-coding application. Both prototypes showed transcript segments and the codes applied to them. We randomly changed 30% of the assigned codes to elicit participant corrections.

The first prototype showed only segments and codes (Figure 1 left). The second prototype added explanations about each ML prediction (Figure 1 right). The explanations were inspired by the Whyline [9], a debugging environment that can answer user questions about program behavior. Each explanation included two reasons *why* the segment was classified with a particular code and two reasons each for *not* classifying it as a different code. The explanations drew attention to a total of ten types of information items that a designer of a machine learning algorithm could reasonably select and implement for the algorithm to make a classification, such as the presence of particular words, sequential ordering of codes, etc.

Procedure

In a pre-session, we familiarized participants with the particular code set and gathered information on the



Okay, Sum of AF12>7 AF12? Should look out for the question Mark	None seeking info.
lets look at the formula [clicks the evaluates formula for the total points column.] There is nothing about this that shows that the person was looking for knowledge or had gained any.	None OK!

[scrolls down and compares GPAs to grade table] **This was coded as *None* because:** • Actions are probably "None" • Segments beginning with a lower-case letter are probably "None" **This was not coded as *Seeking Info* because:** I would say that comparing is seeking for info • Segments containing "compares" are probably not "Seeking Info" • Segments preceded by "Info Gained" are probably not "Seeking Info" **This was not coded as *Info Gained* because:** • Segments preceded by "Info Gained" are probably not "Info Gained" • Actions are almost certainly not "Info Gained"	None Seeking info.

Figure 1: Paper prototypes without (left) and with (right) explanations.

participants' background (gender, academic major, and prior experience with coding).

For the study, we led participants to believe that the paper transcript had been coded using a machine-learned program. The participants were asked to fix any incorrect codes applied to segments and to help improve the program's accuracy by suggesting things to which the program should pay attention. Participants were given 30 minutes to debug the first prototype (79 segments without explanations), after which they filled out a questionnaire asking how they believed the system made its decisions and what information it should use to make better predictions. Participants then debugged the second prototype (41 segments with explanations) for another 20 minutes. We repeated the questionnaire, asking how they now believed the computer made its decisions, and in addition, which aspects of the explanations they found confusing or helpful.

Pens, colored pencils, and post-it notes were provided to encourage feedback directly on the paper prototype in any manner the participant preferred. We also recorded their verbalizations as they worked on the main task, prompting them if their remarks were unclear or they stopped talking, and transcribed the recordings for further analysis.

Analysis Methodology

To categorize what our participants found salient, four researchers jointly established a candidate code set by analyzing a subset of the marked-up prototypes combined with the transcribed audio. Two researchers then iteratively applied this set to sections of a transcript and adjusted the code set after each iteration to ensure reliability.

We employed a code set from a previous study [20] to identify potential gaps in our code set. Our code set accounts for all of the feedback types identified in [20], extended by codes that capture information deemed salient by our participants prior to the introduction of explanations. This final code set is given in Table 1.

Code	
Subcode	*Participant talked about…*
Word/Punctuation	
Single	a single word.
Multiple	multiple words.
Punctuation	a punctuation mark.
Adjustment	a change in word(s) importance.
Process	a change in how features should be extracted or processed.
Segment	a segment as a whole
Relationship	
Word	relationship between words within one segment.
Segment	relationship between segments.
QA-Pair	a question-answer pair of segments.
Reference	some other portion of the transcript.
Double code	being just a continuation of a previously coded segment.
Code Elimination	the segment not fitting into any of the other codes.
Other	Other or unclear.

Table 1: The code set used for data analysis.

Two researchers independently coded a transcript using this coding scheme, achieving an inter-coder reliability of 87% as calculated by the Jaccard index. This level of agreement indicated a robust code set, so the remaining data was split between the two researchers to code.

ML SALIENCY FINDINGS

What Types of Information Are Important?

As a starting point toward investigating ML saliency, we need to understand what types of information users regard as useful when debugging a program and the vocabulary naturally used to tell this information to the program (RQ1). Thus we examined the suggestions participants provided to influence the machine-learned program's logic. These suggestions illustrate the types of information participants' believed the program *should* use to make its predictions.

Figure 2 illustrates participants' feedback suggesting various types of features. In machine learning parlance, a *feature* is a characteristic of the data that is useful for the ML algorithm in making a prediction. First, the high frequency of suggestions about whole segments (n=157, 40%) or word combinations (n=98, 25%) point to a need to shift away from classification approaches based on single words (e.g. "bag-of-words" classifiers) to algorithms that can handle larger functional units. In particular, word combinations appeared twice as often as single words and punctuation combined. From a machine learning perspective, designing an algorithm that adds features for all possible *n* consecutive words up to some cut-off value of *n* is infeasible because it would introduce too many irrelevant features. As a consequence, machine learning algorithms need to take account of user input, as complex features need to be definable by the user as they debug the program.

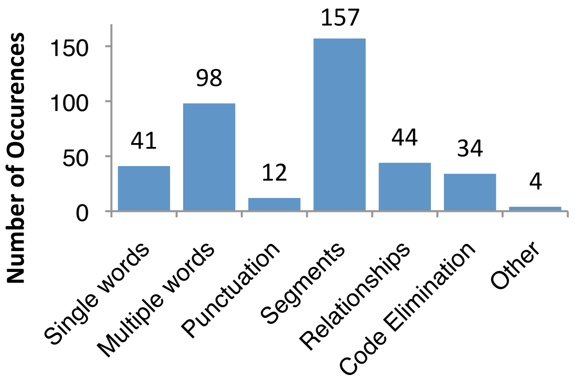


Figure 2: Number of instances of different types of feedback, prior to introduction of explanations.

Second, all but one of the participants reported taking relationships among different words or segments into consideration (*P1: "So is this part a continuation of this?"*), especially between contiguous segments. Hence, when providing opportunities for users to change the logic of the learned program, the application should support the creation of features representing relationships between existing ML features.

Third, although participants seldom mentioned punctuation, seven out of nine participants talked about it at least once. This suggests that punctuation is significant when end users analyze transcripts, yet current ML algorithms routinely ignore it. The reverse is true for absence of features, which participants mentioned three times less often as their presence. This indicates that participants are primarily focused on what they see in the data, as opposed to what is not present. This creates an opportunity for saliency to draw attention to the common practice by ML algorithms of using the absence of certain features for classification.

How Do Explanations Affect Saliency?

We investigated the effects of saliency by introducing explanations of the learned program's logic in the second prototype and comparing the changes of participants' mental models of the prototypes' reasoning processes. This data was gathered by free-form responses to paired questions, allowing us to gauge their mental models before explanations were provided (RQ2), and examine how their mental models changed after salient explanations were made available (RQ3). As illustrated in Figure 3, after interacting with the first prototype (without explanations), participants' perception of the program's logic relied on word and punctuation presence. Seven of our nine participants thought the program made decisions based on the presence of single words, while only two participants mentioned multiple words. Four participants also reported that punctuation played a role. Nobody thought the computer used *absence* of words or punctuation, and only one participant thought it paid attention to relationships between segments. In contrast with how participants thought the program *should* make decisions (cf. Figure 2), their mental model of how it *did* make decisions revealed a much simpler decision process.

Most participants' mental models changed after exposure to explanations, tending toward richer and more complex types of features like word combinations and relationships. Five participants now thought the presence of multiple, not single, words mattered to the computer, and seven participants now thought the program used sequential relationships among words and segments.

An important change in users' models was the appearance of probabilistic reasoning in five of our participants:

P1: "...Uses probabilities of certain codes occurring before and/or after other codes."

P2: "Using probabilities about what the previous boxes were coded...."

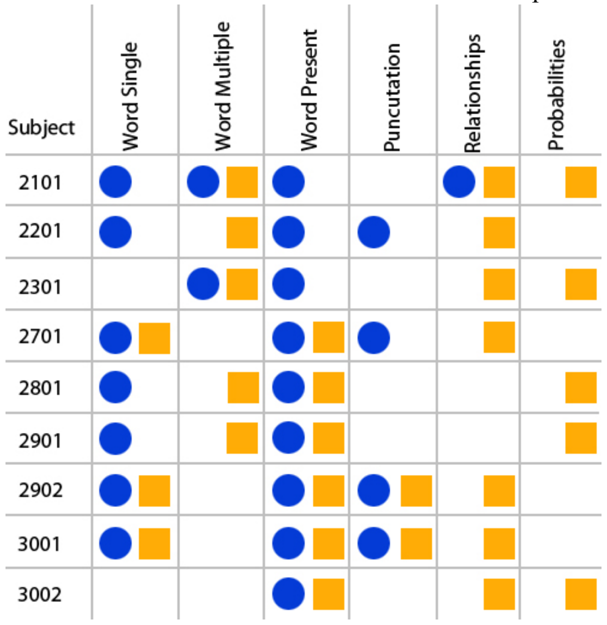
On many occasions, however, user descriptions implied probabilities were the product of consistent *rules*. This reflects a known bias in reasoning called the *outcome approach*, commonly referred to as "the weather problem" [10]. The outcome approach is the phenomenon of interpreting probabilities as binary, rather than the likelihood of a particular response. For example, if a meteorologist states there is a 70% chance of rain the next day, many people interpret this to mean it is supposed to rain. Thus, if the next day is sunny, people will say the meteorologist was wrong. In other words, while participants understood that the computer was using probabilistic reasoning, they still expected it to apply its strongest rule.

Additionally, two participants found the "why not" explanations to be particularly helpful. As one said:

P3: "The info about why it wasn't something was very helpful… Most of the time I didn't agree with reasoning for what the segment was coded as, so having the other reasons helped me understand."

These participants seemed to appreciate a balance of evidence pointing to decisions and valued the ability to tell why something did *not* happen. This finding correlates with a study of the Whyline debugging environment [9], and suggests that such information may be considered salient by end users debugging machine-learned programs.

Taken together, it appears salient explanations demonstrated to participants that the machine-learned program was more complex than initially thought. Saliency particularly enhanced our participant's mental models regarding the learned program's probabilistic nature, modeling of relationships, and its ability to use sequences of words. Furthermore, the fact that participants changed their mental models after new information was presented



Subject	Word Single	Word Multiple	Word Present	Puncutation	Relationships	Probabilities
2101	●	● 🟧	●		● 🟧	🟧
2201	●	🟧	●	●	🟧	
2301		● 🟧	●		🟧	🟧
2701	● 🟧		● 🟧	●	🟧	
2801	●	🟧	● 🟧			🟧
2901	●	🟧	● 🟧			🟧
2902	● 🟧		● 🟧	● 🟧	🟧	
3001	● 🟧		● 🟧	● 🟧	🟧	
3002			● 🟧		🟧	🟧

Figure 3: Participants' mental models before (circles) and after (squares) explanations. Categories with ≤2 before and after instances are omitted.

contradicts the findings of [23], whose participants' early mental models were remarkably persistent, even given counter-evidence. Our results point to the potential for saliency to help users adjust their mental models by providing information necessary for them to properly understand the program's behavior.

SUPPORTING SALIENCY IN CONVERSATIONS WITH A MACHINE-LEARNED PROGRAM

Building upon background literature and the results of our formative study, we identified aspects of machine-learned programs where saliency could provide the user with knowledge both helpful and relevant to their debugging task. These fall into three saliency principles (SP), which later served as design constraints for our prototype.

SP1: Expose the ML Program's Reasoning Process

Just as one would not expect a professional programmer to debug an algorithm's implementation without first understanding how the algorithm operates, it is unreasonable to expect end-user programmers to debug a machine-learned program without understanding its logic or the data it draws upon. Previous research [6, 21] has shown that users are interested in the features the ML algorithm uses. The "Why" explanations in [15] proved effective at enriching the participants' understanding of the program's reasoning process, as did similar explanations in our own formative study.

Two specific aspects of the reasoning process that our formative study suggests should be made salient concerned *probabilities* and *absence of features*. Both concepts were completely absent from participants' initial mental models, indicating a need for explanations highlighting each.

Our formative explanations about probabilistic reasoning were framed in the context of a particular prediction, e.g., *"Segments containing 'compare' are* ***probably*** *'Seeking Info'"*, instead of the static, non-concrete explanations in [12]. This proved to be quite successful and indicates that making the reasoning process explicit should be done with *concrete* explanations. Further exposure of the machine's reasoning might also help resolve the outcome approach phenomenon discussed earlier, allowing end users to better reason about the learned program's logic.

SP2: Support a Flexible Vocabulary

Debugging a machine-learned program is akin to the user holding a conversation with it: the program tells the user why it is making various predictions, and the user tells it why some of those predictions are right and why others are wrong. Before such a conversation is held, however, a vocabulary must exist which both parties understand [3]. As we and other researchers [21] have found, users have a much richer vocabulary than the standard bag-of-words representation used by current ML systems. This richer vocabulary consists of *word combinations*, *punctuation*, and *relational information*. Hence, an aspect of this principle is that both the user interface and the ML algorithm must support a much larger vocabulary than is currently possible, and that this vocabulary be extensible. While the interface should allow for vocabulary expansion by the user, the algorithm must be able to deal with the resulting new user-generated features.

SP3: Illustrate Effects of User Changes

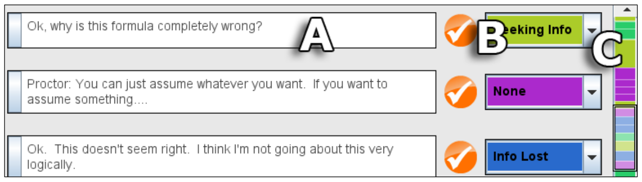
Our participants expressed a clear interest in how a rule was used in the learned program's logic, as evidenced by their positive reaction to explanations regarding why the learned program made each of its predictions. Additionally, end users like to see the effects of *their* actions in the context of a program's behavior [21], and this has been noted previously with "non-learned" programs [9]. Thus, the impact of changes to a learned program's logic should be expressed to end users, so they may understand how their behavior affects the machine's predictions and to be able to adapt their actions to increase the program's accuracy.

One specific effect, peculiar to machine-learned programs, illustrates a second reason behind this saliency principle. Most end users are unaware of a problem plaguing many machine learning algorithms; when one category of data is over-represented in the training set, this same over-representation cascades into the program's predictions. Known as *class imbalance*, this problem is often exacerbated by end user strategies that focus on one category of information at a time [12]. By clearly displaying the effects of user changes, end users can both be educated about such problems, and informed when their actions are contributing to the problem.

THE AUTOCODER PROTOTYPE

Building upon the saliency principles, we designed a hi-fi auto-coding prototype. The AutoCoder application allows users to code segmented text transcripts (Figure 4 A) using a predefined code-set (Figure 4 B). Codes shown on the interface are colored to give users an overview of their coding activity. These colors are replicated in the navigation scrollbar to provide an overview of each code's occurrence over the whole transcript (Figure 4 C). The user is able to manually assign a code to each segment; after three segments have been coded, the computer will attempt to predict the codes of the remaining segments using a variant of the naive Bayes algorithm for sequential data**.** The learned program updates its predictions as the user provides corrections and additional training data.

In addition to the basic ability of coding a transcript, we devised a series of widgets implementing our saliency principles for the AutoCoder prototype.


Ok, why is this formula completely wrong?

Proctor: You can just assume whatever you want. If you want to assume something....

Ok. This doesn't seem right. I think I'm not going about this very logically.

Figure 4: The basic AutoCoder prototype showing a series of segments (A), their corresponding codes (B), and an overview of the transcript's codes (C).

Exposing the Logic

We designed the AutoCoder such that it could describe the most relevant pieces of information leading to its decisions and expose the types of rules it used, thus satisfying our first saliency principle (SP1). AutoCoder provides a list of **Machine-generated Explanations** (Figure 5 W1) for each prediction, informing users of the features and logic that most influenced its decision. A computer icon reminds users that these explanations are based upon machine-selected features.

Specific **Absence Explanations** (Figure 5 W2) inform users about how the learned program utilizes the absence of words or phrases when predicting codes, e.g. *"The absence of '?' often means that a segment is 'Info Gained'"*. The idea of invisible things affecting the machine's predictions has traditionally been difficult to express to users in a comprehensible manner [12, 20], and questions still remain over whether being truthful about this aspect of machine-learned decision-making is desirable, considering its potential to confuse end users.

In providing the above explanations, end users are given the ability to review important elements of the learned program's logic. Users can select any segment to view explanations about its current prediction, resulting in the machine-generated explanation with the highest-weighted features appearing beneath the segment text: this is generally the suggestion that was most influential toward the learned program's prediction. Since the algorithm is probabilistic, it can employ a large number of competing pieces of logic, so the AutoCoder sorts the explanations list according to the most influential features. Users can click a button to progressively show more. If a user disagrees with an explanation, she can delete it to exclude its logic from the learned program's future predictions.

We carefully crafted the wording of the explanations to illustrate that suggestions are open to uncertainty. To further expose the probabilistic nature of the learned program we designed a **Prediction Confidence** (Figure 6 W3) widget. This is a pie graph that displays the program's probability of coding a given segment using any of the possible codes. A confident prediction is indicated by a predominantly solid color within the pie graph, while a graph containing an array of similarly sized colors indicates that the program cannot confidently determine which code to apply. If the program does not display a high confidence in its prediction, users can implement measures to improve its logic.

Debugging With a Flexible Vocabulary

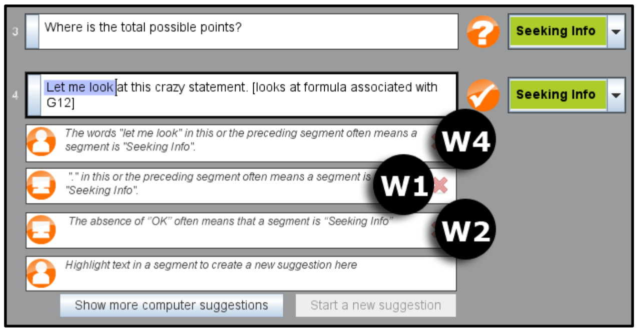
We provided debugging support with a flexible vocabulary to support saliency principle SP2. By selecting a sequence of text, users can add their own features to the program. Hence, they are given the ability to explain to the machine why a segment should be coded in a particular manner. The **User-generated Suggestions** (Figure 5 W4) are integrated into the learned program's logic for determining the codes of the remaining segments not already manually coded. User-generated suggestions are treated by the learned program similarly to machine-generated explanations, but with added weight to reflect the fact that the user attempted to correct the machine. They are distinguished from machine-generated explanations by a "human" icon.

Users are able to create features spanning adjacent segments to model relationships between words and segments, as well as including non-consecutive portions of text in a single segment. An example of a possible user-generated suggestion that incorporates such a relationship is: "*'?' in the preceding segment followed by 'OK' in this segment often means this segment is 'Info Gained'*". We also allow users to add single words or combinations of words to the program. For example, Figure 5 shows a user creating a word combination feature by selecting a portion of text. This flexibility allows a user to extend the machine's vocabulary to closer-match her own.

Illustrate Effects

We developed three widgets that illustrate the effects each user change had on the learned program (SP3). These widgets serve a similar function as a run-time debugger in traditional programming: they provide detailed information about the current state of the program during its execution.

To reflect how a user suggestion impacts the overall coding task, we designed an **Impact Count Icon** (Figure 6 W5) to reflect how many predictions each suggestion affects. For example, the suggestion stating that *"?"* implies a particular code will impact each segment containing a *"?"*, likely resulting in a high impact count. A suggestion with a


3 Where is the total possible points?

4 Let me look at this crazy statement. [looks at formula associated with G12]

The words "let me look" in this or the preceding segment often means a segment is "Seeking Info".

"." in this or the preceding segment often means a segment is "Seeking Info".

The absence of "OK" often means that a segment is "Seeking Info"

Highlight text in a segment to create a new suggestion here

Figure 5: An AutoCoder Machine-generated Explanation (W1), Absence Explanation (W2), and User-generated Suggestion (W4).

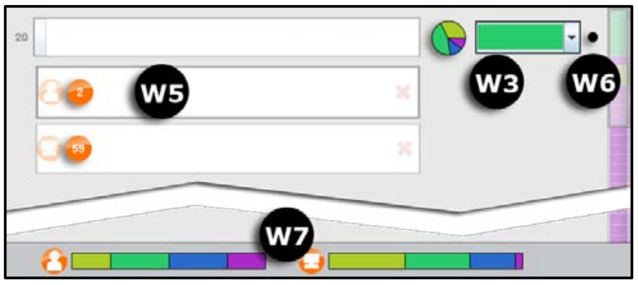


Figure 6: The Prediction Confidence widget (W3), Impact Count Icons (W5), Popularity Bar (W7), and Change History Markers (W6).

high impact count is important, since it will affect many predictions. To indicate which segments are affected by a suggestion, AutoCoder highlights affected segments and their corresponding sections in the scrollbar.

To help users understand the specific implications of their last action, **Change History Markers** (Figure 6 W6) provide feedback on where changes in the program's predictions recently occurred. A black dot is displayed adjacent to the most recently altered predictions beside their respective segments in the scrollbar. This gives the user an overview of every change throughout the program. As the user makes changes that *do not* alter the machine's prediction for a segment, its mark gradually fades away.

The **Popularity Bar** (Figure 6 W7) specifically addresses the problem of class imbalance. It represents proportions of each code amongst the user-coded and machine-predicted segments. The left bar represents code popularity among user-coded segments, i.e., the proportion of codes the user has manually applied to segments. The right bar contrasts code popularity among machine-predicted segments, i.e., the proportion of codes the machine is predicting for remaining segments.

The Popularity Bar serves to illustrate an added nuance of designing widgets to support our saliency principles: a single widget may implement multiple principles. The initial goal of the Popularity Bar was to show the *effects* of class imbalance (SP3), but we later realized that this would only be helpful if users knew *why* class imbalance was a problem for the learned program (SP1). Thus, we indirectly exposed an aspect of the machine's reasoning process to the end user by crafting a widget that would illustrate the negative effects of class imbalance.

USER STUDY

To better understand the impact of saliency on end-user debuggers, we conducted a user study of the AutoCoder prototype. We developed four versions (VB, V1, V2, and V3), each embedding various sets of saliency widgets. The basic version (VB) provided machine-generated explanations, user suggestions, and change history markers; these widgets support the three saliency principles at a basic level. Another version, V1, extended the basic version by including Absence Suggestions and the Impact Count Icon. Version V2 extended the basic prototype by adding Prediction Confidence widgets and the Popularity Bar. A final version (V3) included all seven widgets. A summary of the various versions is given Table 2.

By presenting several subsets of widgets to users, we intended to collect nuanced differences in user reactions. Since working with independent versions for each of the widgets would have placed intensive demands on the user, we grouped saliency widgets with respect to the class of information they display. Inspired by traditional programming languages, we created two classes of widgets, *code-oriented* and *run-time oriented*. *Code-oriented* widgets present information that is independent of the machine's predictions (i.e., information that could be gleaned from the learned program's "source code"). *Run-time oriented* widgets present information that requires the program to "run" or make predictions (i.e., information can only be determined by actually running the "source code"). It is difficult to imagine a professional programmer debugging without both types of information being available. The same logic may apply when end users debug machine-learned programs. The Absence Suggestions and Impact Count Icon both present code-oriented saliency information, and are represented in V1. The Prediction Confidence widget and Popularity Bar displayed saliency information only available at run-time, and were in V2.

	W1	W2	W3	W4	W5	W6	W7
VB	■			■		■	
V1	■	■		■	■	■	
V2	■		■	■		■	■
V3	■	■	■	■	■	■	■

Table 2: The saliency widgets included in each version of the AutoCoder prototype.

The addition of V3, consisting of all seven saliency widgets at once, allowed us to investigate whether *too much* saliency would be a problem for end users.

Procedure

We recruited 74 participants (40 males, 34 females) from the local student population and nearby residents for our study. None possessed experience with machine learning algorithms and only one had previous experience with a task similar to coding. The high number of participants was chosen to allow statistical tests on the resulting log data. We plan to investigate how end-user debugging strategies were affected by the different versions each user interacted with in a future paper. In this paper we focus on how useful end users perceived the various saliency widgets, and the implications this holds for our saliency principles.

The tutorial consisted of a 30-minute introduction to coding, the coding set itself, and the prototype's functionalities. This was followed by a 20-minute period where users coded a transcript with one version of the prototype. We then gave additional instructions regarding a second version of the prototype, before asking users to code another transcript using this different version. We assigned versions and transcript orders randomly across our participants.

In addition to logging participant interactions with the prototype, we used a set of questionnaires to obtain comprehensive user feedback about each saliency widget, participant mental models of the learned program's logic, and their preferred version (out of the two they experienced). We also employed the NASA-TLX survey [7] to evaluate difficulties and perceived success with each prototype version.

Findings

Through participants' questionnaire responses relating to different prototype versions, we were able to collect reactions to individual saliency widgets and compare code-oriented and run-time saliency widgets. Figure 7 shows counts of how useful participants found the various saliency widgets, and Figure 8 displays the perceived effort of interacting with the different prototype versions.

Absence Features and Prediction Widgets

Most participants found the Absence Explanations unhelpful (Figure 7), while still considering Machine-generated Explanations to be useful. Additionally, the Prediction Confidence graphs were widely seen as very helpful. These three widgets reflect SP1 by exposing the learned program's reasoning process, and variation in users' reaction may be due to the amount of effort required to comprehend them. As others [20] have reported, many end users have trouble understanding how absence of something plays a role in a machine's decision-making. As one participant explained:

P4: "[Absence explanations were] very confusing and provided no help."

Furthermore, our formative study showed that participants preferred simple explanations of *why* the words in a segment led the machine to make a particular prediction. These results imply that the complexity of the reasoning should be taken into account when crafting explanations supporting SP1 (Expose the program's reasoning process).

Impact Count, Popularity Bars, and Change History Markers

A majority of our participants found the Impact Count Icons unhelpful, and responses to the Popularity Bar and Change History Markers were neutral. All of these widgets supported SP3; they illustrated the effects of user changes to the learned program. A frequent criticism of all three widgets was that the information they represented was unclear, such as:

P5: "The number in the circle means nothing to me."

P6: "What is it???"

P7: "I don't understand what this graphic was implying."

This may simply illustrate a learning curve that was not adequately dealt with in the tutorial, but it may also imply that widgets supporting SP3 should clearly indicate the type of effects they represent. While our widgets included tooltips to remind participants what the information they conveyed meant, it seems many participants did not use them. Providing visual cues that reflect this kind of information could alleviate the primary participant complaint against these widgets.

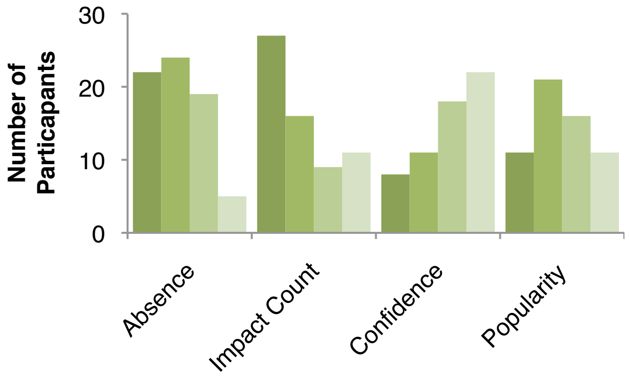


Figure 7: The number of participants who rated each saliency widget "very unhelpful" (darkest) to "very helpful (lightest).

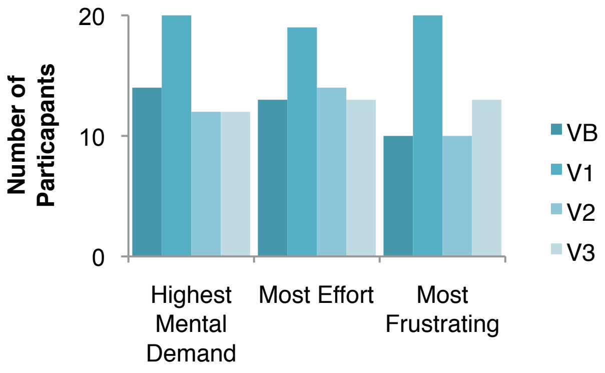


Figure 8: The number of participants who rated each version as requiring the most mental demand, the most effort to use properly, or causing the most frustration.

Code-oriented vs. Run-time Information

Next we examined participants' preference regarding the classes of widgets they interacted with. As Figure 9 Right illustrates, V3, which combined both types of saliency widgets, was a clear favorite. It was most frequently a user's top choice, and least frequently their second choice. The run-time version, V2, was preferred slightly more often than not, but both the code-oriented and basic versions (V1 and VB) were more likely to be the runner-up than the first choice. Participants preferred run-time saliency to code-oriented saliency, but preferred the inclusion of both types of information in a single interface even more.

Interestingly, our participants' preferred version contrasted with the system they felt yielded the best results. The NASA-TLX questionnaire showed that participants felt they performed better with V2 than any other version (Figure 9 Left). V3 did not lead to a clear feeling of success over other systems; instead, users may have preferred working with it for other reasons, such as improved confidence that they could *eventually* accomplish the task.

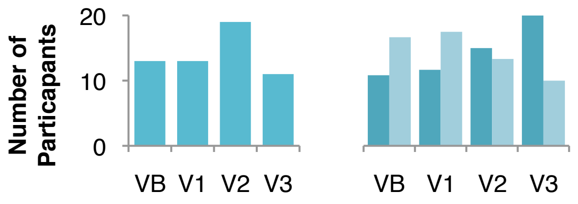


Figure 9: Left: The number of participants who felt they were most successful with each version. Right: The number of participants who rated each version as preferred (dark) and non-preferred (light).

These results are mirrored by the other TLX responses in Figure 8, which showed V1 being more mentally demanding, requiring more effort and leading to more frustration than the other systems. V2 and V3 often performed much better in these respects. This shows that code-oriented saliency (V1) may lead to higher task load than run-time saliency (V2 and V3).

LESSONS LEARNED AND FUTURE DIRECTIONS

As a result of our studies exploring saliency, we learned lessons and design implications that will be explored in future work regarding the effects of saliency on end users debugging machine-learned programs.

Saliency Facets

While we explored the saliency principles overall, their subtle facets and associated usefulness have not been addressed thus far. In our study we fulfilled one such principle (SP3) through two different facets: *code-oriented* and *run-time* debugging information. Our study participants displayed a noticeable preference for run-time saliency widgets versus code-oriented saliency widgets. A typical participant response about the Impact Count Icon marked it as unhelpful:

P8: "Honestly, didn't even look at this."

This same participant, however, found the run-time Popularity Bar to be quite useful:

P8: "I could see if I was spending more time and energy on specific tags and less on others."

Thus, it may be the difference between code-oriented and run-time debugging information, rather than the saliency principles embodied, that determines the perceived usefulness of a particular widget. In addition, the contribution of facets to the remaining saliency principles and their respective influence on usefulness for debugging merits more research.

Debugging Behavior

Exposing the ML program's logic (SP1) may influence user debugging behavior. Some participants in the formative study commented on being uncertain how to code some segments. The popularity of the Prediction Confidence widget may be related to these users gaining confidence in the machine's predictions and feeling there was a lower risk of the program predicting incorrectly than of the user herself being wrong. As one participant phrased it:

P9: "If I was undecided, the pie would help me decide."

Other participants indicated a similar lack of confidence in their decisions, but did not appreciate the computer's help:

P10: "I felt like I HAD to agree with the program."

Here, the saliency widget may have had a negative effect on the user's debugging behavior. Similarly, the Popularity Bar altered some users' behavior in unintended ways:

P11: "This was more distracting than anything, because I wanted each color to be evenly spaced!"

P12: "I had an internal drive to want to teach the computer to be equal. I think this caused me to favor one answer over the next."

Further research could lay bare how saliency leads to changes in user debugging behavior and how to guard against unwise feedback choices.

Future Design of Machine-Learned Programs

A surprise was that the combination of all saliency widgets did not overwhelm or intimidate most users, as the most popular version by far included all seven saliency widgets. Participants often commented on them when asked which treatment they preferred:

P9: "I liked [V3] better because it had a wider overview of outcomes each tag would cause. Easier to see changes occur also."

P8: "I liked having the percentages of what the computer thought, but I could also see what and how much I was telling the computer about each tag."

P13: "[V3] because it provided more detail and information on why the computer suggested a particular tag."

It appears that one practical implication for the design of machine-learned programs is to give the user as wide a choice as possible when implementing saliency.

Machine-learned programs could also further support the saliency principle of supporting a flexible vocabulary (SP2). Future research will explore how the *machine learning program* could suggest possible extensions to the vocabulary. For example, if it detects that the user created features involving negative words (e.g. not, never, neither, etc.) as indicators for the code "Info Lost", it could suggest similar words the user may have missed. Exploiting English as a knowledge source could be amendable to machine learning techniques, such as using WordNet [19] to find synonyms, or exploiting sentence construction and composition. This highlights the importance of making information available to the ML algorithm that may not be expressed solely from the current data via statistics on features and relationships, and hints at the possibility of supplementing a machine-learned program's reasoning through commonsense knowledge sources (e.g., [14]).

CONCLUSION

In this paper we explored candidate principles for incorporating saliency into the design of machine-learned programs. We investigated what information end users considered salient when fixing a machine-learned program, and built upon the results to inform three saliency principles for machine learned programs: expose the ML program's reasoning process; support a flexible vocabulary; and illustrate the effects of user changes.

Our user study explored the perceived usefulness of several widgets built upon these principles and gave us additional insight into how users react to various forms of saliency:

- The combination of *code-oriented* and *run-time* saliency was viewed very positively. Code-oriented saliency on its own, however, was viewed as unhelpful.

- End users were not overwhelmed by the inclusion of saliency widgets. Conventional wisdom often tells us to keep interfaces simple, so this result is surprising.
- A majority of our participants perceived at least one widget instantiating the three saliency principles to be helpful, providing initial evidence that the principles themselves are a sound starting point for supporting saliency in an end-user debugging context.

Supporting saliency can give end users the information necessary to effectively fix a machine-learned program. As such programs continue to become more prevalent, aiding users in understanding and debugging them will play an important role in the ultimate usefulness of programs learned by machines.

ACKNOWLEDGEMENTS
[Removed for blind review]

REFERENCES

1. Becker, B., Kohavi, R., and Sommerfield, D. Visualizing the simple Bayesian classifier. In Fayyad, U, Grinstein, G. and Wierse A. (Eds.) *Information Visualization in Data Mining and Knowledge Discovery*, (2001), 237-249.
2. Billsus, D., Hilbert, D. and Maynes-Aminzade, D. Improving proactive information systems. *Proc. IUI*, ACM (2005), 159-166.
3. Clark, H. H. *Using language*. Cambridge: Cambridge University Press. (1996).
4. Cohn, D. A., Ghahramani, Z., and Jordan, M. I. (1996). Active learning with statistical models. *J. of Artificial Intelligence Research, 4*, 129-145.
5. Dönmez, P., Rosé, C., Stegmann, K., Weinberger, A. and Fischer, F. Supporting CSCL with automatic corpus analysis technology. *Proc. CSCL*, ACM (2005), 125-134.
6. Glass, A., McGuinness, D. and Wolverton, M. Toward establishing trust in adaptive agents. *Proc. IUI*, ACM (2008), 227-236.
7. Hart, S. and Staveland, L. Development of a NASA-TLX (Task load index): Results of empirical and theoretical research, Hancock, P. and Meshkati, N. (Eds.), *Human Mental Workload*, (1988), 139-183.
8. Herlocker, J., Konstan, J. and Riedl, J. Explaining collaborative filtering recommendations. *Proc. CSCW*, ACM (2000), 241-250.
9. Ko, A. and Myers, B. Designing the Whyline: A debugging interface for asking questions about program failures. *Proc. CHI*, ACM (2004), 151-158.
10. Konold, C. Informal conceptions of probability. *Cognition and Instruction 6*, 1, (1989), 59-98.
11. Kononenko, I. Inductive and bayesian learning in medical diagnosis. *Applied Artificial Intelligence 7*, (1993), 317-337.
12. Kulesza, T., Wong, W.-K., Stumpf, S., Perona, S., White, S., Burnett, M., Oberst, I. and Ko, A. Fixing the program my computer learned: Barriers for end users, challenges for the machine. *Proc. IUI*, ACM (2009) 187-196.
13. Lacave, C., and Diez, F. A review of explanation methods for Bayesian networks. *Knowledge Engineering Review 17,* 2, Cambridge University Press, (2002) 107-127.
14. Lieberman, H., Liu, H., Singh, P. and Barry, B. Beating some common sense into interactive applications. *AI Magazine 25*, 4, (2004), 63-76.
15. Lim, B. Y., Dey, A. K., and Avrahami, D. 2009. Why and why not explanations improve the intelligibility of context-aware intelligent systems. *Proc. CHI,* ACM (2009), 2119-2128.
16. Pane, J., Myers, B. and Miller, L. Using HCI techniques to design a more usable programming sys-tem. *Proc. HCC*, IEEE (2002), 198-206.
17. Patel, K., Fogarty, J., Landay, J. and Harrison, B. Investigating statistical machine learning as a tool for software development. *Proc. CHI*, ACM (2008), 667-676.
18. Poulin, B., Eisner, R., Szafron, D., Lu, P., Greiner, R., Wishart, D. S., Fyshe, A., Pearcy, B., MacDonnell, C., and Anvik, J. Visual explanation of evidence in additive classifiers. *Proc. IAAI,* (2006).
19. Miller, G. 1995. WordNet: A lexical database for English. *Comm. ACM 38*, 11, (1995), 39-41.
20. Stumpf, S., Rajaram, V., Li, L., Burnett, M., Dietterich, T., Sullivan, E., Drummond, R. and Her-locker, J. Toward harnessing user feedback for machine learning. *Proc. IUI*, ACM (2007), 82-91.
21. Stumpf, S., Sullivan, E., Fitzhenry, E., Oberst, I., Wong, W.-K. and Burnett, M. Integrating rich user feedback into intelligent user interfaces. *Proc. IUI*, ACM (2008), 50-59.
22. Talbot, J., Lee, B., Kapoor, A., and Tan, D. S. 2009. EnsembleMatrix: interactive visualization to support machine learning with multiple classifiers. *Proc. CHI*, ACM (2009). 1283-1292.
23. Tullio, J., Dey, A., Chalecki, J. and Fogarty, J. How it works: a field study of non-technical users interacting with an intelligent system. *Proc. CHI*, ACM (2007), 31-40.
24. Vig, J., Sen, S., and Riedl, J. Tagsplanations: explaining recommendations using tags. *Proc. IUI*, ACM (2009), 47-56