AN ABSTRACT OF THE THESIS OF

Todd Kulesza for the degree of Master of Science in Computer Science presented on December 2, 2009.

Title: Toward End-User Debugging of Machine-Learned Programs.

Abstract approved:

__________________________________________________________________

Margaret M. Burnett

The results of a machine learning from user behavior can be thought of as a program, and like all programs, it may need to be debugged. Providing ways for the user to debug it matters because without the ability to fix errors, users may find that the learned program’s errors are too damaging for them to be able to trust such programs. We present a new approach to enable end users to debug a learned program. We then use an early prototype of our approach to conduct a formative study to determine where and when debugging issues arise, both in general and also separately for males and females. The results suggest opportunities to make machine-learned programs more effective tools.
Toward End-User Debugging of Machine-Learned Programs

by

Todd Kulesza

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 2, 2009
Commencement June 2010
Master of Science thesis of Todd Kulesza presented on December 2, 2009.

APPROVED:

____________________________
Major Professor, representing Computer Science

____________________________
Director of the School of Electrical Engineering and Computer Science

____________________________
Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

____________________________
Todd Kulesza, Author
ACKNOWLEDGEMENTS

First and foremost, I would not have reached this stage of my education without the constant, indefatigable support of my family. They instilled in me a love of academic pursuits from an early age and continued to encourage me to strive for more, even when “more” meant moving across the country to pursue a graduate degree three time-zones from home. Their continued support has kept me focused in the face of life’s unexpected detours, and I hope they know how grateful I am.

I arrived at Oregon State with a very nebulous concept of what graduate studies entailed. Dr. Margaret Burnett put the time and effort into mentoring me, guiding me in my studies and teaching me what it means to be involved in scientific research. She has been tireless in her role as my advisor, always available to discuss ideas, analyze data, and help turn my rough words into polished academic papers. Her confidence in me has bolstered my own and helped me reach the end of a process which at times seemed unattainable. For her patience, insight, and ability to push her students toward excellence in every endeavor, I am indebted to her.

The faculty and researchers Dr. Burnett has introduced me to have been a constant source of inspiration and encouragement. Dr. Andrew Ko provided many of the ideas that started us down this research path. Dr. Simone Stumpf assisted with the unenviable task of methodically coding the transcripts compiled during this study, and still had energy remaining to collaborate on the presentation of our results. Dr. Weng-Keen Wong served as the machine learning specialist, patiently helping us craft feasible
explanations that accurately reflected the logic of a learned program. Dr. Carlos Jensen provided thoughtful insights about earlier drafts of this paper. While not directly involved in this research, Dr. Emma Byrne and Dr. Yann Riche have provided much-needed perspective and encouragement when the going got frustrating.

My fellow students have been similarly supportive, both in and out of the lab. In particular, Chris Bogart, Valentina Grigoreanu, Jill Cao, and Dr. Joseph Lawrance have been invaluable for advice and assistance. Ian Oberst and Stephen Perona worked with Dr. Wong to develop the machine learning component that was involved in this work’s experiment, the preparation and running of which was assisted by Amber Shinsel, Akshay Subramanian, and Rachel White. Kyle Rector and Joe Markgraf assisted with many of the less glamorous tasks, such as transcribing recordings of each participant’s two-hour lab session.

Outside of university life, Bill and Iris McCanless’ cafe has become a home away from home; most of these words were written at one of their tables, and many of these ideas were informally discussed over coffee with people I’ve come to view as an extended family. There are far too many of you to list individually, so please accept my general appreciation and gratefulness at having all of you in my life.

Finally, without funding, none of this would have been possible. This work was supported by NSF IIS-0803487 and by the EUSES Consortium via NSF CCR-0325273.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Literature Review</td>
<td>4</td>
</tr>
<tr>
<td>Materials and Methods</td>
<td>9</td>
</tr>
<tr>
<td>Design of the Why Questions</td>
<td>9</td>
</tr>
<tr>
<td>Design Principles for End-User Debugging</td>
<td>14</td>
</tr>
<tr>
<td>Machine Learning Design Considerations</td>
<td>16</td>
</tr>
<tr>
<td>Debugging Via Our Prototype</td>
<td>18</td>
</tr>
<tr>
<td>Answering the “Why?” Questions</td>
<td>21</td>
</tr>
<tr>
<td>User Study Design</td>
<td>23</td>
</tr>
<tr>
<td>Dialogue Analysis Methodology</td>
<td>25</td>
</tr>
<tr>
<td>Results</td>
<td>28</td>
</tr>
<tr>
<td>Barriers Encountered</td>
<td>28</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Gender Differences in Barrier Encounters</td>
<td>29</td>
</tr>
<tr>
<td>Barriers and Transitions</td>
<td>32</td>
</tr>
<tr>
<td>What Information Did Users Want to Overcome Barriers?</td>
<td>37</td>
</tr>
<tr>
<td>Gender Differences in Debugging Feature Usage</td>
<td>43</td>
</tr>
<tr>
<td>Conclusion</td>
<td>46</td>
</tr>
<tr>
<td>Bibliography</td>
<td>48</td>
</tr>
<tr>
<td>Appendices</td>
<td>52</td>
</tr>
<tr>
<td>Challenges for Machine Learning</td>
<td>53</td>
</tr>
<tr>
<td>Tutorial for User Study</td>
<td>57</td>
</tr>
<tr>
<td>Questionnaires from User Study</td>
<td>66</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Screenshot of the prototype</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>Close-up of the visualization and debugging features</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>Sum of barriers encountered in all transcripts</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>Average number of barriers per session encountered by males (dark blue bars)</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>Number of transitions from design barriers to other barriers or debugging activities</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Number of transitions from selection barriers to other barriers or debugging activities</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>Number of transitions from coordination barriers to other barriers or debugging activities</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>Number of transitions from use barriers to other barriers or debugging activities</td>
<td>33</td>
</tr>
<tr>
<td>9</td>
<td>An example participant pair’s path through the debugging barriers</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>The number of times each participant implied a particular knowledge base would help them overcome their current debugging barrier</td>
<td>38</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The query grammar used to generate our Why questions</td>
</tr>
<tr>
<td>2</td>
<td>The Why questions and the query grammar productions used to generate them</td>
</tr>
<tr>
<td>3</td>
<td>Visual explanations for three Why questions</td>
</tr>
<tr>
<td>4</td>
<td>Coding scheme used to analyze our data</td>
</tr>
<tr>
<td>5</td>
<td>The knowledge bases participants requested information from</td>
</tr>
</tbody>
</table>
Toward End-User Debugging of Machine-Learned Programs

CHAPTER 1

INTRODUCTION

How do you debug a program that was written by a machine instead of a person? Especially when you don’t know much about programming and are working with a program you can’t even see?

This is the problem faced by users of a new sort of program: those generated by machine learning systems that customize themselves to a particular user. Recommender systems, SPAM filters, and handwriting recognition systems are all examples of software that uses machine learning to adapt its behavior to a specific end user’s preferences. A computer program is, at its heart, merely a collection of rules of behavior, so these machine-learned rules can be viewed as a self-contained program themselves. These learned programs often do not come into existence while the learning environment is in the hands of a machine learning specialist; they are based on a user’s actions, and thus often exist only on the end user’s computer. When one of these programs makes a mistake, the only person present to fix it is the end user. These attempts to “fix” the system can be viewed as debugging—the user is aware of incorrect system behavior, and wants to change the system’s logic so as to fix the fault responsible for the incorrect behavior.

This thesis presents an approach to support end-user debugging of machine-learned programs. Because this notion of debugging is new, an
exploration of fundamental issues and challenges is necessary. Thus, we built a prototype based upon our group’s earlier formative research and used it to investigate barriers faced by end users when they debug machine-learned programs. We also include an analysis of the different knowledge bases explanations should support to guide users past these debugging barriers.

Our prototype was an e-mail application with several predefined folders. The system used a machine-learned program to predict which folder each message in the inbox should be filed to, allowing the user to easily archive messages. End users could ask questions of the software about its decisions, and the system would respond with interactive explanations intended to both help users understand the reasoning behind the classifier’s predictions and alter the classifier’s logic to improve future predictions.

We used our prototype e-mail sorter as a research vehicle to study end users’ difficulties in debugging a learned program’s behavior. We analyzed the resulting set of barriers for sequences of their occurrence and how they related to the users’ debugging progress. Since researchers have recently found evidence of gender differences in debugging (e.g., [11, 31]), we also investigated the interaction of gender with these barriers. Our goal was to empirically discover barriers posing obstacles to participants’ debugging success, rather than empirically evaluate the design of our prototype. Our analysis thus focused on finding such barriers and eliciting the information needed to overcome them.
The main contributions of this thesis are:

• A new “why-oriented” approach to allow end users to debug the logic of a machine-learned program.
• Identification of barriers encountered by end users attempting to debug a machine-learned program.
• Identification of gender differences in the barriers encountered.
• Identification of knowledge bases that explanations of machine-learned programs should support.
CHAPTER 2
LITERATURE REVIEW

Several recent studies have highlighted the need for explanation and visualization of a machine learning algorithm’s reasoning. One [25] examines the obstacles faced by developers familiar with machine learning who need to apply machine learning to real-world problems. Another study [10] investigates the types of questions members of a research team would like to ask an adaptive agent in order to increase their trust in the agent. Similarly, researchers identified the types of information end users want context-aware applications to provide when explaining their current context, to increase both trust in and understanding of the system [20]. Finally, [33] reported that with proper explanations, end users can successfully understand how machine learning systems operate, although overcoming any preliminary faulty assumptions may be problematic. None of this research, however, supports end users actually fixing the logic of a program learned by a machine. We extend the research in this field by investigating two-way communication with end users who initially know nothing about machine learning but are required to interact with a learning system.

Much of the work in explaining probabilistic machine learning algorithms has focused on the naïve Bayes classifier [2, 15] and, more generally, on linear additive classifiers [26]. Explanation of these algorithms is relatively straightforward and computationally feasible on modern hardware. More sophisticated, though computationally expensive,
explanation algorithms have been developed for general Bayesian networks [17]. All of these approaches, however, are intended to explain the reasoning of the algorithm, rather than let the user modify it.

The EnsembleMatrix system [32] provides both a visualization of a learned program’s accuracy, plus the means to adjust the program’s logic. Its audience, however, is limited to machine learning experts developing complex ensemble classifiers. It was not designed to either support or be understood by the end users working with programs powered by the resulting classifiers.

Some Programming by Demonstration (PBD) systems learn programs interactively from users’ examples via machine learning techniques (see [18] for a collection of such systems). While user feedback is, by definition, used to initially create the program, when the user is fixing bugs in the learned program this feedback is limited to only a few kinds of interaction. For example, the only part of CoScripter/Koala programs that are learned are web page objects. Users can correct misidentified objects for a particular page, but these fixes will not affect how the program identifies objects on other pages [21]. Gamut allows users to “nudge” the system, alerting it to mistakes, which then leads to addition or deletion of training examples [22]. Other systems require a familiarity with the underlying language syntax, such as Lisp (e.g., [34]). Recent work with PBD systems also relates to debugging machine-learned programs [7], but their technique allows the user to retract actions in a demonstration, which results in adding missing values to the training data rather than directly modifying the classifier’s logic.
Outside of PBD, there are a number of debugging systems that help end users find the causes of faulty program behavior. For example, in the spreadsheet domain, WYSIWYT [5] has a fault localization device that reasons about successful and unsuccessful “tests” to highlight cells whose formulas seem likely to be faulty. Woodstein [35] helps users debugging e-commerce problems, visualizing events and transactions between services. The Whyline [13] is a debugging tool aimed at event-driven programs and has been extended to help users debug the document and application state in word processors [24]. Recent work [19] has explored the usefulness of the Whyline’s approach in the domain of machine learning, but was restricted to helping end users understand very simple decision trees.

Three works of end user debugging research are of particular interest to us because they identify specific end user debugging needs. First, Ko et al. explored learning barriers novice programmers encountered when learning how to solve problems in traditional programming environments [14]. These barriers may hold implications for how to support non-traditional debugging, such as fixing machine-learned programs. Second, researchers from the WYSIWYT project have categorized the information needs of end users debugging spreadsheet programs [12]. Their results enumerate the types of information that explanations should include in order to support successful end user debugging activities. Finally, researchers have explored the types of information explanations of learned programs should include to help users understand the program [20], but they did not address the information end users need to fix a learned program.
In summary, the ability of end users to interactively debug machine-learned logic has been quite limited. Researchers have begun to investigate how such logic can be explained to end users, but user feedback, if available at all, has been heavily restricted.

Our own prior research has begun to explore end-user interactions with machine-learned programs. Using a paper prototype, we previously investigated different types of explanations machine learning systems could provide to end users, as well as user reactions to these explanations [28]. This paper prototype was also used to elicit corrective feedback from participants, allowing us to design an interactive prototype supporting the explanations best understood by participants and the feedback types they most requested [29]. The interactive prototype permitted us to run offline experiments studying the effects of rich user feedback on prediction accuracy versus traditional label-based feedback. The results suggest that when rich user feedback is incorporated into the learned program’s decision making process, it has the potential to increase the accuracy of the resulting predictions [29, 30]. Some users, however, experienced difficulty in providing useful feedback to the machine; the quality of the learned program’s predictions decreased as a result of these users’ input.

In this thesis we build upon the above foundation. The themes we previously explored were 1) three types of explanations (keyword-based, rule-based, and similarity-based) [28], 2) the types of feedback users would like to give a learned program (e.g., adjusting feature weights) [28, 29], and 3) the knowledge sources the machine would need to assimilate the desired feedback [28]. This thesis presents a new Whyline-inspired approach
to debugging learned programs. Our approach extends the successful keyword-based explanations, making them interactive so that they also serve as a feedback mechanism. Also, we used a prototype based on this approach to reveal and categorize the barriers users encountered while debugging a machine-learned program. Finally, we categorized the knowledge sources users believe will help them overcome these barriers.
CHAPTER 3
MATERIALS AND METHODS

Inspired by the success of the Whyline’s support for debugging [13, 24] and favorable user feedback regarding “why” and “why not”-style explanations [19, 20], we designed a method to allow end users to ask “why” questions of machine-learned software. Our approach is novel in the following ways: (1) it supports end users asking questions of statistical machine-learned programs, (2) the answers aim at providing suggestions for end users to debug the learned programs, and (3) users can modify these answers, which results in real-time adjustments to the learned program’s logic.

Design of the Why Questions

Our approach comprised two stages: (1) we rigorously generated the set of all possible questions via the creation of a formal grammar, and then (2) we filtered the set of questions generated by the grammar to remove impossible situations, where impossible refers to situations that the software’s user interface does not allow.

For stage (1), we began by inventorying the domain objects, such as messages and folders, that users could interact with. This inventory is enumerated in the Subjects section of Table 1. Our second step was to inventory all possible user actions our prototype supported, such as adjusting the importance of a word. Our third step was to inventory feedback effects from the system, such as the machine’s folder predictions.
and its estimation of word importance. The inventories from steps two and three together are the *Situations* section of Table 1.

Step four relates to past, present, and future system states. Thus, we enumerated three question-word phrases -- *why is*, *why did*, and *how can* -- plus the negations of these phrases. We split these into two non-terminals (*question* words and *verb* words) to make our *query* production more flexible. The *Questions* and *Verbs* sections of Table 1 generate a super-set of these types of question-word phrases. Finally, in step five we added the *Modifers* section to allow generated questions to be more specific, e.g., a question about *recently* changed predictions, rather than *all* changed predictions. *Modifers* in our environment may be temporal, related to word importance, or differentiate singular subjects from plural subjects. These five steps left us with a grammar that describes the universe of possible Why questions (Table 1).

In stage (2) we filtered the questions generated by the grammar in two ways. First, we removed questions about anything that was not possible in our prototype. For example, messages (*a subject*) can change classification (*a situation*), but words and folders (*also subjects*) can not. We further refined our list of questions by removing any questions not relating to debugging, such as “Why can important words be displayed?”. Stage (2) resulted in the nine Why questions depicted in Table 2. (While the number nine may seem small, the original Whyline required only six types of questions [13] in the complex domain of Java programming.)
Table 1: The query grammar used to generate our Why questions.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Questions</th>
<th>Verbs</th>
<th>Modifier</th>
<th>Subjects</th>
<th>Situations</th>
</tr>
</thead>
<tbody>
<tr>
<td>• [query] = [question] [verb] [subject] [situation]</td>
<td>• Why...? (ask why the system did something)</td>
<td>• To be (supports questions about current and past state of</td>
<td>• Singular (this word)</td>
<td>• Message</td>
<td>• Current classification</td>
</tr>
<tr>
<td></td>
<td>• [question] [situation] [verb] [subject]</td>
<td>• How...? (ask how the user can modify the system to do</td>
<td>• Plural (all of these words)</td>
<td>• Folder</td>
<td>• Change in classification</td>
</tr>
<tr>
<td></td>
<td>• [situation] = [situation] [subject]</td>
<td>something better)</td>
<td>• Restriction (only important words)</td>
<td>• Word</td>
<td>• Importance</td>
</tr>
<tr>
<td></td>
<td>• [subject] = [subject] [modifier]</td>
<td></td>
<td></td>
<td>• Change</td>
<td>• Availability (displayed in the UI)</td>
</tr>
<tr>
<td></td>
<td>• Why...? (ask why the system did something)</td>
<td></td>
<td></td>
<td>• End user (me)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• To be (supports questions about current and past state of the system)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• To do (supports questions about current and desired future state of the system)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Can/Make (supports questions about possible future states of the system)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 “How” is an abstraction; many of our “How” questions are expressed as “Why” questions, with the answer explaining how the end user can alter the system to achieve the desired outcome.
Table 2: The Why questions and the query grammar productions used to generate them. Color is used to map productions to English phrases.

<table>
<thead>
<tr>
<th>Why Questions</th>
<th>Generating Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why will this message be filed to &lt;Personal&gt;?</td>
<td>[question] [verb] [modifier] [subject] [situation] [subject]</td>
</tr>
<tr>
<td>Why won’t this message be filed to &lt;Bankruptcy&gt;?</td>
<td>[question] [verb] [modifier] [subject] [situation] [subject]</td>
</tr>
<tr>
<td>Why did this message turn red?</td>
<td>[question] [verb] [modifier] [subject] [situation]</td>
</tr>
<tr>
<td>Why wasn’t this message affected by my recent changes?</td>
<td>[question] [verb] [modifier] [subject] [situation] [modifier] [subject]</td>
</tr>
<tr>
<td>Why did so many messages turn red?</td>
<td>[question] [verb] [modifier] [subject] [situation]</td>
</tr>
<tr>
<td>Why is this email undecided?</td>
<td>[question] [verb] [modifier] [subject] [situation]</td>
</tr>
<tr>
<td>Why does &lt;banking&gt; matter to the &lt;Bankruptcy&gt; folder?</td>
<td>[question] [verb] [subject] [situation] [subject]</td>
</tr>
<tr>
<td>Why aren’t all important words shown?</td>
<td>[question] [verb] [modifier] [subject] [situation]</td>
</tr>
<tr>
<td>Why can’t make I this message go to &lt;Systems&gt;?²</td>
<td>[question] [verb] [subject] [modifier] [subject] [situation] [subject]</td>
</tr>
</tbody>
</table>

² The user interface moved the location of “I” in this question to fit the grammatical rules of English.
Our textual answers include a mixture of static and dynamic text to make clear to users that the answers relate to their current situation. For example, the answer to Table 2’s second question (with dynamically-replaced text in <brackets>) is:

*The message will be filed to <Personal> instead of <Bankruptcy> because <Personal> rates more words in this message near *Required* than <Bankruptcy> does, and it rates more words that aren’t present in this message near Forbidden. (Usage instructions followed this text.)*

In addition to the textual answers, three questions are also answered visually. These are shown in Table 3. Pink and blue bars are used to represent the importance of each word to a given folder. The location of the bars indicate the weight of each word for the machine’s predictions; the closer a word’s bar is to *Required*, the more likely it is that messages containing this word will be classified to the given folder. Conversely, the closer a word’s bar is to *Forbidden*, the less likely it is that messages containing this word will be classified to the given folder. We used bars, as opposed to points or some other representation, both because their large size makes mouse-based interaction easy, and the visual contrast between differing bars is immediately obvious. Providing the necessary dynamic content to these textual and visual explanations required support from the underlying machine learning algorithm. Details on the machine learning algorithm and how it was used to provide dynamic answers are discussed in Appendix A.
Design Principles for End-User Debugging

Many end-user debugging systems require users to have access to source code. This is problematic for machine-learned programs, since there is no obvious “source code” behind the scenes to study. Debugging also often involves inspecting concrete data about program execution. For example, debuggers provide access to the value of variables and the stack. Thus, a challenge we encountered while designing our prototype was how this distinction between source code and execution state applies to a learned program.

One principle that guided the design of our prototype was that users should be able to “debug” by directly interacting with the words in actual e-mail messages. These words and their associated predictive value for each folder (required, forbidden, or somewhere in between) can be viewed as
the source code for our e-mail-classifying learned program because they are a complete representation of the program’s logic. The execution state of the learned program is then the current set of predictions resulting from this logic, including the machine’s degree of certainty in each prediction. This results in user-manipulable source code (words and their associated importance), changes to which affect the execution state (machine predictions and their associated confidence).

A second principle of our design was that explanations should not obscure or simplify the learned program’s logic. Machine learning is a difficult topic in Computer Science, so there is a temptation to hide some detail from the end user in an effort to make the system easier to understand. Recent research [33], however, has shown that end users can understand the logic machine learning systems use to make their decisions. Furthermore, we believe that in order to successfully debug a program, the user must have the full wealth of knowledge about how the system operates; abstracting away some of this information could hamper the user’s debugging efforts.

To implement these principles, we developed an approach in which the interactive answers to the debugging questions (Table 2) are composed of a representation of the source code itself. Specifically, the visualizations (Table 3) are representations of the learned program’s rules regarding the importance of the presence or absence of words in up to two different folders. The user is able to alter the importance of these words to fix classification problems, just as they would be able to edit the source code of a Java application to resolve run-time errors. Thus, when the end user
asks the machine a question, the same visualization that supplies the answer can be manipulated to achieve the desired outcome. We discarded variants of the visualizations that omitted parts of the logic for the sake of simplicity, such as only displaying required words for a given folder.

Consistent with the notion that these visualizations are the source code, and that what the user is trying to do is fix the code, it follows that the user must be able to manipulate the visualizations. Since the visualization already displays the union of all words in all e-mail messages, we did not give users the ability to add new words; available manipulations are limited to adjusting the importance of a word with respect to a particular folder, including making it unimportant. Time constraints prevented us from implementing a method for users to delete features, but our tutorial explained that setting a word to unimportant was functionally equivalent to deleting the word from the feature set. These manipulations are the method users have to fix machine-learned bugs—they allow the user to directly change the logic the learned program will follow.

**Machine Learning Design Considerations**

For the purposes of investigating our basic approach and barriers, we decided to begin with an algorithm widely used in our study’s domain of e-mail processing. We chose naïve Bayes [27] because, first, it is a commonly used algorithm for spam filtering. Second, naïve Bayes is structured such that rich user feedback can be integrated in a straightforward manner. Third, we can readily generate rule-based explanations from the naïve Bayes classifier, and our previous work [28]
reported that rule-based explanations were the most easily understood types of explanations. Our bar graph visualization can be considered either a rule-based or a keyword-based explanation, since the rules are defined using keyword presence and absence. Fourth, when the user modifies the weight of a keyword, naïve Bayes can set the new value to be (almost) exactly the value specified by the user.

Techniques like user co-training [29], in contrast, assign a new weight that could potentially be quite different from the user-assigned value. User co-training assigns a weight that is a combination of the user-assigned value and the classifier’s internal weight. In pilot runs with user co-training, we observed that this behavior can be frustrating to users because it makes the algorithm appear to disobey the user’s change.

In our visualization, naïve Bayes does in fact make a slight modification to the user-assigned weight. We treat the user-specified folder assignment for the current e-mail as a new training data point for the classifier. Thus, in addition to the user-assigned weights, the classifier (and hence the visualization) is also changed by the new data point formed from the current e-mail and the user-specified folder assignment. This alteration makes the classifier more sensitive to user feedback in the interactive setting; without it, changing the weight of one out of 6,000 available features (as there were in our prototype) has little noticeable impact on classification.
Debugging Via Our Prototype

Figure 1 shows a bird’s eye view of the prototype we built following the above principles. It consists of the usual e-mail client elements: a folder list (A), a list of headers in the current folder (B), and the current message (C). The two bottom panes contain the textual answers (D) and interactive visualizations for debugging (E).

If at some point the user wants to know why the program is behaving in a particular way, she can ask any of the Why questions through either the global menu bar or context-sensitive menus by right-clicking on the
object (such as a specific word) she has questions about. For example, in Figure 2, the user has just asked why this message is not filed in the Systems folder. The keyword bar graph shows the system’s opinion of the importance of each word to the Resumes folder (dark pink), which is the current folder for this message, versus importance to the Systems folder (light blue). The user has decided that if the word “please” (second from left) occurs in a message, it is not likely the message belongs in the Systems folder. She then dragged the light blue bar lower; how much lower depends on her assessment of how important “please” should be to the Systems folder. The dark blue bar indicates the original importance of “please”, allowing the user to remember her change and its magnitude.
In prior empirical work [29], we learned that users wanted access to a rich set of information, possibly even the entire set of keywords that the system has available. The keyword bar graph provides this—all words are available using this graph, and each can be manipulated. User changes to each bar graph entry causes the system to immediately recalculate its predictions for every message in the inbox, allowing users to instantly see the impact of their manipulations. These changed folder predictions are listed textually next to each message header in the inbox, highlighting headers whose predictions have changed. For every manipulation, the user immediately sees both how the “source code” (in terms of the importance of the words) changes.
of words) has changed, and also how the resulting program output changes.

**Answering the “Why?” Questions**

The questions “Why will this message be filed in X?” and “Why won’t this message be filed in X?” both require dynamically generated answers that rely on support from the underlying machine learning algorithm. Before explaining how these answers are generated, we define the following notation. An e-mail message is represented as a “bag of words”, which converts the e-mail message into a Boolean vector \( W = (W_1, \ldots, W_m) \) in which \( W_i \) takes the value \( \text{true} \) if the \( i \)th word of a vocabulary of \( m \) words is present in the e-mail message and \( \text{false} \) otherwise. The vocabulary in our experiment consists of the union of the words from the following parts of all the e-mails: the e-mail body, the subject line, and e-mail addresses in the To, From and CC parts of the message header. Stop words, which are common words with little predictive value such as “a” and “the”, are not included in the vocabulary.

**Answering: “Why will this message be filed in X?”**

In previous work [28] we observed that end users understood how the presence of keywords influenced a message’s classification, but they struggled with the concept of how the absence of keywords influenced the same classification. We addressed this difficulty through the visualization of the naïve Bayes classifier, shown in the leftmost image of Table 3, in which the weight associated with each word in the vocabulary is depicted as a bar that slides between the two extremes of \( \text{Required} \) and \( \text{Forbidden} \). For
folder $f$, this weight is the probability $P(W_i = \text{true} \mid F = f)$ where $W_i$ is the random variable for the $i$th word and $F$ is the random variable for the folder. Since $P(W_i = \text{false} \mid F = f) = 1.0 - P(W_i = \text{true} \mid F = f)$, the position of the bar can be interpreted in two ways. The higher the top of the bar, the more important the presence of the word is to the prediction. Alternately, the lower the bottom of the bar, the more important the absence of the word is to the prediction.

Answering: “Why won’t this message be filed in $X$?”

If the current message is predicted to be filed under folder $f$, the user can ask why it won’t it be filed in folder $f'$. The application answers this why question by displaying the dual-bar graph shown in the right image of Table 3. The two bars correspond to $P(W_i = \text{true} \mid F = f)$ and $P(W_i = \text{true} \mid F = f')$, respectively. The purpose of this dual-bar view is to allow the user to compare and contrast the importance of various words between the two folders. Furthermore, since the dual-bar view allows weights associated with the two folders $f$ and $f'$ to be manipulated, we can illustrate the degree that an e-mail “belongs” to either folder $f$ or $f'$ based on the magnitude of $P(F = f' \mid W_1, \ldots, W_m)$ and $P(F = f \mid W_1, \ldots, W_m)$, respectively. For instance, if folder $f$ is the originally predicted folder for the e-mail and $P(F = f' \mid W_1, \ldots, W_m) > P(F = f \mid W_1, \ldots, W_m)$ after the user interacts with the visualization, then the e-mail will be filed under folder $f'$. In the visualization, we can illustrate the degree to which an e-mail “belongs” to folders $f$ and $f'$ using the arrow shown at the top of Figure 2.
User Study Design

Using a prototype of the above approach, we conducted a formative study. Our purpose was not to validate the usefulness of our approach, but rather to investigate barriers and their impact on end users attempting to debug a machine-learned program.

The study involved a dialogue-based talk-aloud design in which a pair of users verbally expressed their thoughts to each other while collaborating on a task. This pair design encouraged participants to voice their reasoning and justifications for actions via typical social communication with their partners.

The participants consisted of six pairs of female and five pairs of male students with an even distribution of GPA, years in university, and e-mail experience across gender. All participants were required to have previous e-mail experience but could not have a computer science background. In order to eliminate a lack of familiarity with each other as a source of noise in our data, pairs had to know each other prior to the study and sign up together. Pairs also had to be same-gender, so that we could clearly identify any gender differences that might arise.

We ran the study one pair at a time. Each session started with the participants completing a questionnaire that asked for background information and gathered standard pre-session self-efficacy data [8]. We then familiarized the pair with the software and examples of classification through a 20-minute hands-on tutorial. For the main experiment task, participants were asked to imagine that they were co-workers in a corporate department at Enron. Their department included a shared e-mail
account to provide easy access to work communications that affected all of them. The premise was that new e-mail software had recently been installed, featuring the ability to learn from the users and automatically classify messages into a set of existing folders. They were told that their supervisor had asked them to get messages from the Inbox into the appropriate folders as quickly as possible, doing so in a way that would help improve later classification.

We used the publicly available Enron e-mail data set in our experiment. To simulate a shared mailbox, we combined messages from three users (farmer-d, kaminski-v, and lokay-m) that these users had originally filed into five folders (Bankruptcy, Enron News, Personal, Resumes, and Systems). At the start of the experiment, each folder held 20 messages; these were used to initially train both the classifier and the participants regarding how messages were to be filed. The Inbox contained 50 messages for the participants to work on. The amount of training data was relatively small to simulate real-world instances where users have not invested the time to label hundreds of training examples.

The pair worked on the main experiment task for 40 minutes and participants were asked to switch control of the mouse half way through the experiment. We used Morae software to capture video and audio of each user session, synchronized with their screen activity. We also logged user actions via our own instrumentation. After completing the main task, participants individually filled out a post-session questionnaire gathering feedback about the prototype and post-session self-efficacy.
Dialogue Analysis Methodology

To analyze the participant dialogue during our study, we developed two code sets (Table 4) that capture barriers and debugging activities. Regarding barriers, Ko et al. identified six types of learning barriers experienced by novice programmers using a new programming environment [14]. These barriers informed our investigation because our participants, like theirs, were problem-solving about how to make programs work correctly and were inexperienced with the provided facilities for debugging. The first five barrier names and the definitions as they apply to our environment are in Table 4. We did not use Ko et al.’s sixth barrier, searching for external validation, because all problem solving in our experiment was based on facts internal to our environment. Regarding debugging activities, previous research [9, 13] identified six common actions in fixing bugs in programming environments. We applied the two of these not involving data structuring or writing new source code, and also introduced a fault detection code. These codes are also given in Table 4.

We then applied the codes to “turns”. A turn consisted of sentences spoken by a participant until his or her partner next spoke. Speech by one participant that contained a significant pause was segmented into two turns. If the same barrier spanned multiple turns (for example, if one person was interrupted by the other), only the first occurrence of the barrier was coded. Coding iteratively, two researchers independently coded a 5-minute random section of a transcript. We calculated similarity of coding using the Jaccard index (dividing the size of the intersection of
codes by the size of the union). Disagreements led to refinements in coding rules, which were then tested in the next coding iteration. Agreement eventually reached 82% for a five-minute transcript section, followed by an agreement of 81% for a complete 40-minute transcript. Given this acceptable level of reliability, the two researchers then divided the coding of the remaining transcripts between themselves.
Table 4: Coding scheme used to analyze our data.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design Barrier</strong></td>
<td>Uncertainty regarding overall debugging strategy (i.e., designing a solution to the problem). “Can we just click File It?”</td>
</tr>
<tr>
<td><strong>Selection Barrier</strong></td>
<td>Knows what to do, but is having trouble selecting which object to change. “What kind of words should tell the computer to [file this] to Systems?”</td>
</tr>
<tr>
<td><strong>Coordination Barrier</strong></td>
<td>Doesn’t understand how changes affect the rest of the system. “Why... why it won’t go to Personal...”</td>
</tr>
<tr>
<td><strong>Use Barrier</strong></td>
<td>Trouble determining appropriate weights to use with the source code visualization. “So is [this word] ‘unimportant’?”</td>
</tr>
<tr>
<td><strong>Understanding Barrier</strong></td>
<td>Doesn’t understand the system’s feedback. “Why is ‘web’ more forbidden for [the] Systems [folder]?”</td>
</tr>
<tr>
<td><strong>Fault Detection</strong></td>
<td>Detecting an incorrect prediction by the system. “It’s going to [the] Systems [folder]; we do not want Systems.”</td>
</tr>
<tr>
<td><strong>Diagnosing</strong></td>
<td>Diagnosing the specific cause of a detected fault. “Well, ‘e-mail’ needs to be higher.”</td>
</tr>
<tr>
<td><strong>Hypothesizing</strong></td>
<td>Hypothesizing a general solution for a detected fault. “Let’s move something else, and then maybe it’ll move [the e-mail] to Systems.”</td>
</tr>
</tbody>
</table>
CHAPTER 4

RESULTS

Barriers Encountered

Participants ran into an average of 29 barriers during the 40-minute study (with a range from 7 to 66). Barriers were equally likely to be encountered at the beginning and end of the study. It is important to note that everyone hit barriers, and some encountered them very frequently, underscoring the importance of addressing barriers in fixing machine-learned programs.

As Figure 3 shows, the most frequent barriers were Selection barriers (40% of all barriers encountered). This type of barrier relates to the difficulty of selecting the right words or messages to modify in order to give feedback to the system, for example:

P712: “Then ‘news’? Well, they like team players. Contributions? That would be more that you’d use for news then Systems.”

Coordination barriers also arose frequently (28% of all barriers). Participants often wondered how the feedback they were about to give would change the system’s other predictions, as well as coordinating how

![Figure 3: Sum of barriers encountered in all transcripts.](image-url)
the system responded (or failed to respond) to their source code modifications:

\[P732: \text{"Resume? [user finds word, makes ‘resume’ required] Why} \]
\[\text{didn’t it change it? How about university?”}\]

The fact that \textit{Selection} and \textit{Coordination} barriers accounted for most observed barriers is confirmed by the questionnaires, where 16 of 22 respondents (72\%) mentioned difficulty in determining which words were important when fixing misclassified mail. The prevalence of these types of barriers suggests the need for intelligent user interfaces to be able to direct end users to the most useful places to give feedback, such as which words will have the strongest effect on message reclassification.

Participants ran into \textit{Design} and \textit{Use} barriers less frequently (14\% and 12\%, respectively). While these barriers should not be neglected, the predominance of \textit{Selection} and \textit{Coordination} barriers suggests that end users may have less trouble deciding on a strategy for \textit{how} to give feedback (\textit{Design} and \textit{Use}), than on \textit{where} to give feedback (\textit{Selection} and \textit{Coordination}).

**Gender Differences in Barrier Encounters**

Males and females did not appear to experience the same number of barriers: females encountered an average of 33.3 per session, versus the male average of 24.4 per session. This difference was despite the fact that males talked more (and thus had more opportunities to verbalize barriers) than females, averaging 354 turns per session, compared to 288 for females.

Figure 4 shows the average barrier count per session; the same differences were observed when comparing the average counts per turn.
Females experienced more barriers in almost every category; the only exceptions were *Coordination* and *Understanding*. *Selection* barriers, the most common barrier type, had a very large difference: females averaged 14 per session, about 1.5 times more than the male average of 9. This difference was statistically significant despite the small size of our sample population (Wilcoxon Rank-Sum Test: \(z=2.1044, p<0.05\)). *Design* barriers, too, exhibited a strong contrast, with females averaging 5.3 per session versus males averaging 2.8. The differences in both the total learning barriers and *Design* barriers encountered were not statistically significant, though this may be a result of our small sample size (totaling six female and five male pairs). A larger sample is needed to provide more conclusive evidence of gender differences.

One reason for these differences may be that females expected more problems due to lower self-efficacy (a form of self-confidence specific to the expectation of succeeding at the upcoming task [1]). Females began the experiment with lower self-efficacy than males, scoring an average of 38 out of a possible 50, compared to 42 for males (via [Figure 4: Average number of barriers per session encountered by males (dark blue bars) and females (light pink bars).]
a self-efficacy question set [8]). Even with our small sample, this difference was significant (Wilcoxon Rank-Sum Test: z=-2.64, p<.01). This is consistent with similar self-efficacy differences for end users engaging in other complex computer tasks [3, 11, 31]. Our results about differences in barriers are consistent with this prior research in another aspect, too: these prior works showed gender differences in both features used and the strategies by which end users tried to fix errors in spreadsheets.

Another reason for the gender dissimilarity may be due to differences in information processing. For example, work on the selectivity theory of information processing [23] has shown a number of differences in how males and females process information. According to this theory, females are more likely to work with information comprehensively, whereas males are more likely to pursue information more selectively. The following quotes illustrate the tendency of female pairs to examine several words from a message before moving on, versus males’ propensity for advancing to the next message as quickly as possible:

**Female Pair**

P1131: “So that [word is] really important. And then, um, probably ‘updates’ would be important. And then, um... [the word] ‘virus’?”
P1132: “Yeah. And then, uh, [the word] ‘login’.”

**Male Pair**

P1211: “Its [classification is] correct. It’s learned something, eh.”
P1212: “Um hmm.”
P1211: “Lets go to the next message.”

The selectivity theory is also consistent with our frequency data: females worked with a larger set of words than males did (106 unique
words for females vs. 62 for males), perhaps to perfect the algorithm’s performance. Males, conversely, may have been more inclined to move on to the next message as soon as they obtained the desired effect. This suggests that in order to support a wide range of end users, debugging features should be designed so that both of these strategies can lead to success.

**Barriers and Transitions**

When a participant encountered a barrier, what happened next? For example, did some barriers send participants spiraling into non-productive repetition? Were there male and female patterns of barrier sequences that matched gender theory predictions?

To answer these questions, we investigated sequences of transitions after participants encountered each type of barrier. We also explored differences in these sequences between male and female participants. The barriers and activities coded in participants’ verbalizations are simply states between which they can transition. To calculate the probability of each state (barrier or activity) following an initial barrier, we divided the number of occurrences of a particular subsequent state by the total number of states that followed the initial barrier. For example, if *Selection* followed *Design* once and *Diagnosing* followed *Design* twice, then the probability of *Selection* following *Design* was computed as $1/(1 + 2) = 0.33$, or 33%, and the probability of *Diagnosing* following *Design* was computed as $2/(1 + 2) = 0.66$, or 66%. We use these probabilities only for clarity. Our graphs (Figures 5, 6, 7, and 8) show the exact number of instances for
completeness. Despite these numerical summaries included for clarity, note that the lack of preconceived hypotheses make inferential statistics on these data inappropriate, and so we refrain from calculating them.

Overall, participants showed no discernible tendency to react to design barriers with a specific debugging activity, nor did they often go on to encounter a specific debugging barrier. The distribution of transitions from Design barriers (Figure 5) was the most uniform of the barriers, especially for females. Subsequent Coordination barriers were most frequent, averaging 19% over all transcripts, but Design, Selection, Fault Detection, Hypothesizing, and Diagnosing each followed this barrier at least 10% of the
time. Males, however, reacted to Design barriers with some form of debugging activity on average 70% of the time, versus 46% for females.

Selection barriers were followed by Diagnosing 40% of the time (Figure 6). The next most prevalent barrier was a second Selection code (19%), suggesting that Selection barriers were either quickly overcome and led to Diagnosing, or they cascaded, stalling participants’ debugging progress. The relatively high instance of Selection barriers stalling participants suggests the need for machine-learned programs to point out which words or features would be most likely to change the program’s behavior; we discuss how this might be done in Appendix A. These participants, for example, could have benefitted from this sort of help:

P732: “And what about ‘interview’? Oh, we just did that, so no. ‘Working’, maybe?” [finds word]
P731: “Well, no because ‘working’ could be used for anything really.”
P732: “True.”
P731: “‘Work’, no.”
P732: “What about... [scrolls left] ‘scheduling’. No, that could be News.”
P731: “That could be News, too.”
P732: “What about ‘scientist’?”
P731: “That could be Personal.”

Males had a higher tendency of Hypothesizing following a Selection barrier than females, 26% to 11%. Recall that Hypothesizing was coded when the pair discussed a possible fix but didn’t include a specific word, whereas Diagnosing indicates that the pair specified the word they intended to modify. Thus, males were more likely to follow a Selection barrier with a general solution, while females tended to first agree on a
word to alter. We know our female participants entered the experiment with lower self-efficacy than our male participants, and prior research [4] has revealed female end users to be more risk averse (in general) than male end users. Both of these situations may be alleviated by the pair coming to agreement about the best way to proceed; a participant’s self-efficacy could be boosted by discovering her partner agrees with her idea, and this improved confidence may in turn lower the perceived risk of the proposed debugging fix. The same solution proposed to help users overcome Selection barriers (directing end users toward words which will have the strongest effect on message reclassification) may help low self-efficacy users as well, by greatly reducing the choice of which words to modify down to a more manageable, less intimidating subset.

Like Selection barriers, Coordination barriers often led to Diagnosing (30%) (Figure 7). Taken together with the other two debugging actions, Fault Detection (14%) and Hypothesizing (20%), this barrier was followed by a debugging action 65% of the time. Males, however, tended to follow Coordination barriers with more Diagnosing than females (47% vs. 18% respectively), whereas females followed them with more Hypothesizing than males (29% vs. 8%). One interpretation of these results is that following confusion regarding the impact of their changes, female participants were more likely to step back and attempt to coordinate how their changes will impact the entire system, whereas males tended to stay focused on a specific failure. This would be yet another indication of the comprehensive problem-solving strategy associated with females [23], providing further evidence of the need to support both comprehensive and
Finally, Use barriers (Figure 8) were strongly tied with Diagnosing (44%); all other transitions were below 15%. It seems that when a Use barrier was encountered, our participants’ response was to adjust their specific solution, rather than move to a different problem or generalize a solution. This was equally true for males and females.

The most problematic of these transitions was Selection, which frequently led to yet another Selection barrier. This suggests that once a user first had trouble deciding where to give feedback, they became less and less able to do so. Figure 9 illustrates the problem by graphing all of the barrier transitions for one of our participant pairs (P701 and P702). The high number of incoming edges to the Selection box was typical, as is the loop from Selection back to itself. Thus, providing support for helping users to make these where decisions may be critical to their successful debugging of a machine-learned program.
Figure 9: An example participant pair’s path through the debugging barriers. The width of the arrows indicate the percentage of transitions: thinnest = 6%, thickest = 20%. Transitions accounting for 5% or less of the total are not shown.

What Information Did Users Want to Overcome Barriers?

Identifying the barriers obstructing end user debuggers shows us where and when users could benefit from additional support, but it doesn’t tell us what information said support should provide. We analyzed the previously-identified barriers in the participant transcripts to determine what information the users needed to overcome each barrier. The results establish seven knowledge bases, defined in Table 5, that explanations of machine-learned program should address. Figure 10 displays the number of times each participant implied a particular knowledge base would help them overcome their current debugging barrier.

The most frequent information requested by end users was concrete explanations about how to fix the machine’s logic. Users wanted to know specifically which word weights they should be modifying to move a
message into a specific folder, *how much* they should be adjusting the word weights, and previews of *what will happen* after adjusting the weights of particular words. This type of information alone represented nearly half of the total information requests among participants (42%), underlining the positive impact we may see by addressing this information gap.

![Figure 10: The number of times each participant implied a particular knowledge base would help them overcome their current debugging barrier.](image-url)
Table 5: The knowledge bases participants requested information from.

<table>
<thead>
<tr>
<th>Knowledge Base</th>
<th>Example User Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugging</td>
<td>Do we have to teach the system everything?</td>
</tr>
<tr>
<td>Strategy</td>
<td>Is there an easy way to get through this?</td>
</tr>
<tr>
<td></td>
<td>Can we do this quicker?</td>
</tr>
<tr>
<td></td>
<td>Should we adjust anything at all?</td>
</tr>
<tr>
<td></td>
<td>Should we file it now?</td>
</tr>
<tr>
<td>User Interface</td>
<td>Can we go back and fix bad classifications?</td>
</tr>
<tr>
<td>Features</td>
<td>Where’s the button that was supposed to show up?</td>
</tr>
<tr>
<td></td>
<td>Which color is related to which folder?</td>
</tr>
<tr>
<td>User Data</td>
<td>How many messages are going to each folder?</td>
</tr>
<tr>
<td></td>
<td>Where are similar message currently filed?</td>
</tr>
<tr>
<td></td>
<td>Is [some word] used in other messages or folders?</td>
</tr>
<tr>
<td>ML Algorithm’s</td>
<td>Where do these word weights come from?</td>
</tr>
<tr>
<td>Capabilities</td>
<td>What parts of the message are used in classification?</td>
</tr>
<tr>
<td></td>
<td>Can the system handle phrases?</td>
</tr>
<tr>
<td></td>
<td>Can we include context of keywords?</td>
</tr>
<tr>
<td></td>
<td>Does changing a word weight affect other folders?</td>
</tr>
<tr>
<td>ML Program’s</td>
<td>Why did this message turn red?</td>
</tr>
<tr>
<td>Current Logic</td>
<td>Why are more messages being filed to [folder]?</td>
</tr>
<tr>
<td></td>
<td>Why didn’t anything change after my last action?</td>
</tr>
<tr>
<td></td>
<td>What just happened?</td>
</tr>
<tr>
<td></td>
<td>Why won’t this message be filed to [folder]?</td>
</tr>
<tr>
<td>How to Fix the</td>
<td>How important should [word] be?</td>
</tr>
<tr>
<td>ML Program’s</td>
<td>Which word should we modify?</td>
</tr>
<tr>
<td>Logic</td>
<td>Which words should be ‘forbidden’?</td>
</tr>
<tr>
<td></td>
<td>What if we...?</td>
</tr>
<tr>
<td>User Action</td>
<td>What have we done?</td>
</tr>
<tr>
<td>History</td>
<td>Were our changes helpful?</td>
</tr>
<tr>
<td></td>
<td>Was one of our past actions wrong?</td>
</tr>
</tbody>
</table>
The prototype the participants worked with was designed to provide much of the background information users would need to make decisions about the learned program’s logic for themselves, such as the importance of the words in each message and a comparison of their current weights between two folders. The list of potential words, however, was the union of every word in every e-mail message (approximately 6,000), which likely explains why users wanted assistance determining which words they should work with. Sensitivity analysis [6] may be a solution to this problem by filtering out words that had little impact on a message’s classification, thus allowing the end user to focus only on words likely to fix an incorrect classification.

The next-largest set of user-requested information related to the learned program’s current logic. This includes user questions about why the learned program is behaving in a particular manner, such as “Why did this message turn red?” and “Why won’t this message be filed to Systems?”.

This was the primary set of questions we hoped to answer via our Whyline-inspired debugging approach. As with the knowledge gaps regarding how to fix the machine’s logic, users appeared to want concrete information; they mentioned specific messages and folders, rather than general questions about how the machine makes predictions. This desire for concrete solution highlighted a nuance in our prototype, which coupled concrete visual explanations of word weights with non-concrete textual explanations of the machine’s general algorithm for classifying messages.
Participants frequently commented that the textual answers to their questions was the same as before, and thus unhelpful. For example:

P1001: Ok go to the why button. [clicks on the global why menu] Ask why not Systems. [asks ‘why won’t this message be filed to..?’] [hovers over choice of folder] We want it to be in Systems. [selects Systems].

P1002: It says the same thing about word importance.

Phrasing the explanation in the context of the message a user is currently working with may help end users overcome this particular knowledge gap.

When participants encountered a Design barrier, they frequently believed they could overcome it if they possessed more details about debugging strategies. These strategy questions fell into two categories: general, when the end user appeared to be searching for a different, better strategy (e.g., “Do we have to teach the system everything?”), and refinement, when the user had questions about particular aspects of their current strategy (e.g., “Should we file it now?”). Recent work has explored the possibility of using textual and video explanations to present end users with debugging strategies [12]. Such techniques may be especially applicable in this domain because of low user self-efficacy regarding debugging a machine-learned program. Watching similar people succeed at the same type of task not only helps guide users toward successful strategies, but can also increase their confidence in being capable of completing the task [1].

Users expressed an interest in overviews of the entire data set they worked with, such as “How many messages are in each folder?” and
“What other messages is [some word] used in?”. This information is independent of the machine learning system and could be gleaned by manually inspecting each message, yet all save one of our participant pairs explicitly requested easier access to support their debugging efforts. Part of this may have been due to unfamiliarity with the set of e-mail messages our participants worked with, but such an overview could be useful in other circumstances where end user debuggers are working with data that’s either new or reviewed infrequently.

A small number (7%) of user barriers were the result of imperfect knowledge of the prototype’s user interface. This is unsurprising for users working with any software for the first time, but nonetheless highlights that even if all other barriers are overcome, users may still stumble if the software is inadequately explained or if it makes debugging unnecessarily complex.

The final two knowledge gaps, machine capabilities and user action history, were rare, combining for only 5% of the total gaps encountered. This isn’t to say they are less important than the other identified knowledge gaps; a flawed understanding of what the learned program is capable of parsing and using in its predictions would obviously seriously impair one’s ability to provide useful feedback to the program, and a clear indication of the actions a user has performed over time may help end users understand the long-term effects of their changes on the machine’s predictions. The idea of displaying the history of user actions is also consistent with the Whyline approach, since each user-made change to the machine’s logic can impact the classification of multiple messages, and
similarly, a single change in classification may be the result of a collection of multiple user adjustments. Explanations of why such a change in classification occurred would be incomplete if they omitted the various user modifications contributing to the re-classification.

**Gender Differences in Debugging Feature Usage**

Previous researchers have reported gender differences in usage of debugging features in spreadsheets [3, 11, 34], and our data suggest that males and females may have used debugging features for the learned e-mail program differently as well. While not statistically significant at the \( p < .05 \) level, we observed differences that are consistent with existing gender theories and we regard these as further justification for more research in this field. Specifically, when interacting with the keyword bar graph (which was participants’ only explicit way of specifying logic changes to the classifier), there were three suggestive differences between males and females.

The first difference was identifying the fault. The primary way to pursue a fault was to ask a “Why isn’t this message in this folder?” question, since that was the way to bring up the bar graph showing the importance of words to both the faulty and the desired folder (Figure 2). Females asked slightly more of these “Why isn’t this message in this folder?” questions than males did (an average of 12 per session for females, 9 for males).

The second difference was in the comprehensiveness with which males and females considered the state of the machine-learned program. Specifically, scrolling through the bar graph (which was displayed in
partial answer to the “Why isn’t this message in this folder?” question) allowed the current weights of words, respective to folders, to be compared. Females scrolled through the bar graph more than twice as much as males (average of 189 scrolling movements for females per session vs. 89 for males), suggesting more comprehensive consideration of the words and their weights.

The third difference was in explicitly debugging the logic, i.e., adjusting the weights of words. Logs of participant actions revealed that females made slightly more edits to words and their weights in the bar graph than males (average 38 per session for females, 34 for males) in their debugging.

These differences in using specific features to effect changes to the program are corroborated by what the participants said about debugging: females’ verbalizations included more Fault Detection (average of 14.3 per session vs. 8.8 for males), more Hypothesizing (13.3 vs. 8.6) and slightly more Diagnosing (28.6 vs. 26.6) than males. These differences were despite males’ greater number of total verbalizations.

Males, on the other hand, focused more on filing messages in the present, with less regard for debugging to improve the future: they filed more messages away (average 15.4 per session) than females (average 11 per session), and dragged slightly more messages directly to folders (average 4.2 per session for males, 2 per session for females).

There are several possible explanations for these differences in debugging feature usage. Females’ interest in interacting with more of the words is consistent with the previously-mentioned Meyers-Levy’s selectivity theory [23]. Our own prior work has also found gender
differences consistent with this theory when end users debug spreadsheets [31]. Females may have also exhibited a greater responsiveness to social norms and obligations, as one participant articulated:

P512: “But if the computer does not learn from this, then the next group that comes in after us will have to do the same thing.”

An important difference between adjusting words via the bar graph (done more by females) and filing or dragging messages (done more by males) was reversibility; word adjustments could be reset to their original values with the click of a button, but, as was emphasized during the tutorial, messages could not be moved once filed. This created a risk related to filing that did not exist with manipulating words. Our findings thus are consistent with existing literature reporting that females are more risk averse than males (discussed in [3]). This suggests the need for features that lower perceived risk, such as the addition of an “Unlearn” capability.
CHAPTER 5

CONCLUSION

In this thesis we took a fresh look at end-user interactions with machine-learned programs. We began by viewing end users’ attempts to change a program’s logic as debugging, a novel perspective that emphasizes the problem-solving aspect of fixing a machine-learned program. We then developed a “Why-oriented” approach to support end users debugging these learned programs. Our approach was informed by existing end-user debugging research and may serve as a starting point for future designs in this field. Our contributions in this area include:

• A Whyline-oriented approach to the domain of end-user debugging of machine-learned programs.

• A method of presenting textual and visual explanations that both truthfully describes a naïve Bayes classifier’s current logic and allow a user to modify said logic.

Using this approach as a basis for our investigation, our study revealed barriers faced by end users debugging machine-learned programs and the information needed to methodically overcome these barriers. Our primary findings were:

• Every participant encountered barriers while debugging the learned program, demonstrating that these barriers present real obstacles for end users debugging machine-learned programs.

• Selection and Coordination barriers were the most frequent obstacles to debugging the learned program. The sheer number of these
instances strongly suggests the value of providing end users with information about where to give feedback to the machine-learned program in order to debug effectively. Sensitivity analysis may help to focus the end user on where changes should occur.

- Participants frequently believed they could overcome debugging barriers if they were provided with concrete explanations of how the machine made its predictions and indicated which word weights they should modify to achieve a particular outcome.
- Participants exhibited gender differences in the number of barriers encountered, the sequence of barriers, and usage of debugging features. These differences, alongside their ties to theories explaining possible roots, suggest that debugging tools for learned programs must support both comprehensive and non-comprehensive debugging strategies and take steps to alleviate low self-efficacy and risk-averse end users.

The generalizability of our results to the vast array of existing machine-learned programs remains an open question, which could build upon the foundation and concrete results from the work reported here. Ultimately, assisting end users so they may successfully overcome these barriers will be an important step down the path to effectively supporting end-user debugging of machine-learned programs.
BIBLIOGRAPHY


31. Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Burnett, M.,
Wiedenbeck, S., Narayanan, V., Bucht, K., Drummond, R., and Fern,
X. Testing vs. code inspection vs. ... what else? Male and female end

32. Talbot, J., Lee, B., Kapoor, A., and Tan, D. S. EnsembleMatrix:
interactive visualization to support machine learning with multiple
classifiers. Proc. CHI (2009), 1283-1292.

33. Tullio, J., Dey, A., Chalecki, J. and Fogarty, J. How it works: a field
study of non-technical users interacting with an intelligent system.

34. Vander Zanden, B. and Myers, B. Demonstrational and constraint-
based techniques for pictorially specifying application objects and
308-356.

35. Wagner, E. and Lieberman, H. Supporting user hypotheses in
30-37.
APPENDICES
Supporting the Why Questions

Can any machine learning algorithm provide informative and efficiently computed answers to these or other “Why” questions? Can a general “recipe” be used by any machine learning algorithm to answer these questions? For the “Why won’t this message be filed in X?” question, the strategy employed by naïve Bayes can be easily extended to any other classifier. The “Why will this message be filed in X?” question is somewhat more restrictive. Any linear additive classifier of the form \[ \sum_i w_i f_i \], where \( w_i \) is the \( i \)th weight and \( f_i \) is the \( i \)th feature, can be visualized as a bar graph. Apart from naïve Bayes, many other machine learning algorithms are linear additive classifiers, including logistic regression, perceptrons, and linear support vector machines. The interpretation of the bar, however, will depend on the classifier, and the bar may not necessarily behave as the slider previously described. Also, the bar graph visualization is more complicated to explain if the linear additive classifier requires regularization to deal with correlated features.

In general, machine learning algorithms vary greatly in their comprehensibility, and thus the bar graphs are not a one-size-fits-all solution. For instance, classifications by decision trees are visually understood while predictions from neural networks are much more understood while predictions from neural networks are much more

---

3 Authored by Weng-Keen Wong, Oregon State University, from [16]. Included for additional background about our machine learning techniques.
difficult to explain. Furthermore, other machine learning algorithms are capable of providing much more in-depth explanations and can thus answer a greater range of “Why?” questions. As an example, Bayesian networks provide a sophisticated but computationally expensive mechanism for providing detailed explanations of how different pieces of evidence influence the final prediction made by the algorithm [17]. A current challenge for machine learning is to develop informative and efficiently computed explanations of statistical machine learning algorithms.

**Sensitivity Analysis**

One difficulty with visualizing a text classifier is that there are approximately 16,000 features involved, where each feature corresponds to a word in the vocabulary. The sheer number of features makes finding a desired word in the bar graph particularly cumbersome for a user, even though we supported sorting by weight and by alphabetical order. Furthermore, modifying the weight on many of these words produces little or no change to the final prediction.

To mitigate this problem, we plan to incorporate ideas from sensitivity analysis, which is a technique used in statistical modeling to determine the robustness of a model to changes to its parameters [6]. Chan and Darwiche [6] investigate the sensitivity of probabilistic queries on a Bayesian network in response to changes to a single parameter in the network. The authors then develop bounds on the effect of these changes to the query. In future work, we will apply these bounds to the naïve Bayes classifier,
which is a special case of a Bayesian network. The bounds will allow us to
determine if a change to a parameter (i.e., the probability $P(W_i \mid F)$) has
little or no effect on the predicted folder. We can then reduce the number of
features displayed in the visualization by not displaying the bars that
cause insignificant changes to the final prediction.

The Popularity Effect

Participants who concentrated on training the classifier to recognize e-
mails for one specific folder at a time experienced a “popularity effect” in
which the folder with the largest number of filed e-mails dominated the
classifier’s predictions for the rest of the e-mails in the inbox. This
popularity effect is primarily caused by the high dimensional nature of the
data, the relatively sparse training data, and the class imbalance of the e-
mail folders. These factors cause the classifier to overfit both the training
data and the rich user feedback for the smaller folders.

To illustrate this point, suppose the user employs such a filing strategy.
The dominant folder could be the Systems folder, which has keywords
such as “Windows” and “McAfee” that are easily identified by the user.
Once a large number of messages have been filed to the dominant folder
and the classifier learns from this set of newly acquired training examples,
the distribution for the dominant folder is accurately learned. However, the
classifier is poorly trained on the non-dominant folders. In fact, the
classifier overfits the training data for the non-dominant folders, and the
rich user feedback for these folders may even exacerbate the overfitting.
This overfitting makes all e-mails seem unlikely to be classified into the
non-dominant folders, because they must match exactly the under-smoothed distributions for these folders. As a result, the classifier files many of the e-mails in the inbox under the dominant folder.

Although this can be remedied by providing sufficient training data for all folders, there remain some practical challenges: Specialized e-mail folders tend to contain small numbers of e-mails, resulting in sparse training data for e-mail classifiers, and e-mail is known to be “bursty,” with e-mails from a small handful of folders dominating the inbox at certain times. Due to the imbalance in the number of e-mails belonging to each folder, the popularity effect thus remains a real-world problem.
APPENDIX B

TUTORIAL FOR USER STUDY

Introduction

Hi, my name is Todd Kulesza, and I’ll be walking you through our study. I’ll be reading from this script so that I’m consistent in the information I provide you and the other people taking part in this study.

Our research is studying ways to improve an automatic email-filing program. The technology we’re using is similar to that found in commercial e-mail software. I’ll take you through a short tutorial and then we’ll get started on the main experiment. Please don’t discuss this study with anyone, as we don’t want other participants to receive any advance information.

The other people involved in the study are Dr. Margaret Burnett, Dr. Simone Stumpf, Dr. Weng-Keen Wong, Dr. Andrew Ko, Ian Oberst, Amber Shinsel, Stephen Perona, Rachel White, and Akshay Subramanian.

This is what’s called a “think aloud” study; while going through your tasks, I’d like the two of you to talk to each other and think out loud--basically, verbalize your thought process and discuss ideas with each other. To get warmed up, I’d like you to try the think aloud idea with this problem: together, discuss how you would get here from the MU, including where you would meet up.

(Allow time for think aloud, encourage or interject as needed)

Great! Now please read and sign this informed consent form, and then fill out this background questionnaire.
(wait for them to fill out the questionnaire)

If you have any questions, there’s contact information available on the Informed Consent form that you signed.

[Contact: Dr. Margaret Burnett <burnett@eecs.oregonstate.edu>

Any other questions may be directed to the IRB Coordinator at (541) 737-8008]

**Tutorial**

Let’s get started on the tutorial. Please let me know if you have any questions.

(pause)

For this experiment I’d like you to imagine that you’re both employees at the Enron Corporation. Your group has a shared email address that receives messages from many different people, and each may be intended for a different person in your group. A new email application has been installed that will allow your group to file these emails into folders, so that different people can quickly find all of the messages relating to a certain topic. The application can even automatically decide where to file new messages. Your group may receive hundreds of messages in a day, so it’s important to manage them efficiently. Your boss has asked you both to spend about a half-hour configuring the system; you have too many other things to do to spend more time than that.

(pause and maximize window)

Take a look at the screen. (pause) On the left are your folders, in the middle is the list of emails, and on the right current email.
Now, find the message that has “hello” in the subject line and click on it. Notice that in the “Will be Filed to” column of this message, it says “Resumes”. Now if you go to the “Resumes” folder, you can see what types of messages belong in “Resumes”. Please take four or five minutes to try and develop an intuition for the type of email that belongs in each folder.

(5 minute time limit)

Alright, now go back to the Inbox. Incoming messages arrive here. If you look at the messages in the Inbox and where the system thinks they go, you may find that some are poor choices, while others are good. Do you see any that need adjusting? There’s no “correct” answer, but you can base your opinion on the types of messages already sorted into each folder. If you don’t feel a message belongs in any folder, pick the folder you feel is most similar.

Now, select the “hello” message again. The bar graph in the bottom shows words and their current importance to the system’s choice of folder, in this case “Resumes”. If a word is near Required, messages with that word are more likely to be filed to the current folder. If a word is near Forbidden, messages with that word are less likely to be filed to the current folder. You can click on the bars and drag them up or down to adjust the importance of a word. Try adjusting the importance of “interview” so that it’s closer to Required. This makes the system a little smarter, since
now it knows that the word “interview” means a message may belong in the “Resumes” folder.

(pause)

See these red messages? (point to them) When you change the importance of a word, it may effect how other messages will be filed. The red highlight let’s you know that because of your change, the system has changed its mind and will file the message somewhere new. You can see the new and old folders here. (point to them) These changes aren’t permanent until you click Commit (point to it). If you don’t like the effects of a change, you can click Reset (point to it) to reset all of the bars. If you select a different message before committing your changes, the system will prompt you to either commit or reset those changes before moving on. Sometimes a small change will make lots of messages turn red, especially at first. Let’s click Reset, to set the bars back to the way they were.

(pause)

At the bottom of the bar graph is a scroll-bar, because there are usually too many words to fit in the graph at once. You can sort the words so that the most important appear first, or you can sort them alphabetically. Try telling the system to sort the words alphabetically now. (pause) You can also use these buttons (point to them) to control which words appear in the bar graph. Try clicking ‘Show all words’. (pause) Notice how more words show up in the graph? (pause) Also, you can quickly find a word in the bar graph by highlighting it in the message, right-clicking it, and then choosing Find Word from the menu. Try doing this for Molly. (pause and point to the word) See how it scrolled the bar graph to Molly and highlighted the word
for you? (pause) When you right-click, you also have the option of asking why the highlighted word matters to the system. Finally, you can hover over a word to make it appear in a tooltip. Sometimes this can be easier to read. Go ahead and try that for the word “Molly”.

(pause)

In addition to coaching the system by changing the importance of words, you can drag messages directly to the desired folder. While this will move a message to a folder, the system won’t learn from it. Try this out by selecting the EOL Application ID and Password message (point to it), and dragging it to the Systems folder.

(pause)

A better method of moving messages to folders is the ‘File It’ button (point to it). The selected message will be filed to the folder listed in the ‘Will be Filed to’ column. Select as many messages as you like by holding down the ‘Control’ key while you select. This lets you file many messages at once. Unlike when dragging, the system will increase the likelihood that similar messages will be filed to the destination folder. As an example, select the Hello and Resume messages (point to them), and click File It (point to it). (pause) See how messages turned red? That’s because the system learned from the messages you filed.

(pause)

The numbers next to the folder names tell you how many messages are inside each folder (point to them). The more email in a folder, the higher the chance of new messages being filed to it.
So, to review, you can manipulate the bar graph to coach the system, making its guesses about where to file each new email more accurate. Then, you can use the File It button to move messages into the appropriate folders, which has the added benefit of teaching the system what types of messages belong in those folders. If your coaching makes the system’s guesses really accurate, then when someone in your group checks this email account, they will be able to hit File It when new e-mail arrives and be confident that the messages ended up in the right folders.

Do you have any questions about the features we’ve learned?

(pause for any questions about the features)

OK, now please switch places so that you each have a chance to use the application.

(pause while they switch seats)

In addition to the features we’ve already covered, this email application allows you ask it several different questions. For example, select the first “Please Print” message (pause). It seems like a personal message, but it’s not being filed to the Personal folder. Now click on the Why… menu. Read each question out loud (pause). Do any of the questions look like they’d help us teach the system that this message belongs in Personal?

(pause)

Let’s ask why this message wouldn’t be filed to the Personal folder (pause). The system’s answer to your question appears in the bottom of the screen (point to it). Go ahead and read it out-loud (pause). If you are ever curious about something the software is doing, feel free to click the Why
menu and see if your question is listed—the list of questions depends on the computer’s recent actions, so the available questions change. If you have questions about a specific word or email, be sure to select it before clicking the Why menu. The answers to these questions may be able to increase your efficiency and accuracy.

(pause)

Now, let me draw your attention to the bar graph again. As part of the answer to your question, it’s displaying two sets of bars, one for the importance of words to the folder this message will currently be filed to (in this case, Resumes), and another showing the importance of those words to the Personal folder. You can adjust these bars to change a word’s importance. For example, if you move a bar near Required, that word’s presence in a message increases the chance the system will file it to this folder. Conversely, if you move a bar near Forbidden, that word’s presence in a message decreases the chance the system will file it to this folder. The bars are color-coded to match the two folders listed here (point to the folder names); pink bars affect Resumes, while blue bars affect Personal. Also, while we’re on this screen, notice the arrow that points between the two folders? (point to it) The more certain the system is that the selected message belongs in one of these two folders, the more the arrow will point to that folder. If the arrow is pointing straight up, the system is very uncertain about where to file the email.

(pause)

Sometimes you may find that the system becomes a bit stubborn, meaning that your changes don’t seem to have any effect. Let’s pretend
that’s happened now. To start, let’s move the pink Please bar down to the unimportant range. (pause) We’ve made a change, but it didn’t have any effect. Now ask the “Why can’t I make this message go to Personal?” question (pause). Read through the answer out-loud. (pause). Let’s click the “Just Move It” button to force the system to learn that this message belongs in Personal (pause). Sometimes you may need to do this to a few messages. If you find that even using the “Just Move It” button isn’t helping, it may be best to move on to messages that belong in a different folder, such as Enron News (point to it).

You have a couple minutes to practice coaching the system. You can’t make any changes to messages that were already filed, either via dragging or the File It button. Let me know if you have any questions.

(Give them 2 minutes)

**Main Task**

Now it’s time for you to start the main task. We’re going to erase the changes you’ve made so far, so you’ll be starting fresh. There’ll also be a new batch of e-mail in the Inbox. (Restart in NB mode) You’ll have forty minutes to work. During that time new email may arrive in your Inbox. I’m going to leave the room; if you need anything, you can instant message me via Google Talk (point to the GTalk window). Please remember to think-aloud while you work—let us know your thought process. Your job is to help your group get emails into the right places with minimal time investment. Do you have any questions?

(Give them 20 minutes. Leave the room.)
(After 20 minutes, re-enter room.)

OK, I’d like you both to switch places so that each of you has the chance to directly use the application. Also, please use the “Why” menu at the top of the screen to ask a question of the system. Any available question is fine.

(Give them 20 minutes. Leave the room.)

**Post-Session Questionnaire**

Okay, your time is up! Please do not touch the computer anymore or log out. Here is a questionnaire that I’d like you to fill out—it should only take a few minutes.

**Compensation and Thank You**

Now I need you to sign the receipt and collect your money. Thank you for participating, and let me know if you have any questions!

(Hand out receipt and pay them)

COLLECT SIGNED RECEIPT!
APPENDIX C

QUESTIONNAIRES FROM USER STUDY

Background Questionnaire

Subject number

Gender
Male  Female

Major

Current year
Freshmen
Sophomore
Junior
Senior
Post-Bac
Graduate

Cumulative GPA (roughly)

Is English your primary language?  Yes  No
If not, how long have you been speaking English?  Years

What have you used computers for? (please check all that apply)
For high school  How many years?
For college  How many years?
For professional use  How many years?
For personal use  How many years?

How many years have you been using email?

Do you use folders to categorize your email?
Have you ever knowingly used automatic junk-mail filtering software?

The following questions ask you to indicate whether you could use a new email classification system under a variety of conditions. For each of the conditions, please indicate whether you think you would be able to complete the task.

Given an email program that attempts to automatically categorize new messages into folders, such as “Work” and “Personal”, I could help train the computer to categorize mail accurately:

<table>
<thead>
<tr>
<th>... if there was no one around to tell me what to do as I go.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>... if I had never used a similar email program before.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>... if I had only the software manuals for references.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>... if I had seen someone else doing it before trying it myself.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>... if I could call someone for help if I got stuck.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>... if someone else had helped me get started.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>... if I had a lot of time to complete the task.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>... if I had just the built-in help facility for assistance.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>... if someone showed me how to do it first.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>
**Post Session Questionnaire**

Subject number

1. Please check one box for each of the following questions:

<table>
<thead>
<tr>
<th>MENTAL DEMAND</th>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>How much mental and perceptual activity (e.g. thinking, deciding, calculating, remembering, looking, searching etc.) was required to do the task?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEMPORAL DEMAND</th>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>How much time pressure did you feel due to the rate or pace at which the emails appeared?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PERFORMANCE</th>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>How successful do you think you were in doing the task?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EFFORT</th>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>How hard did you have to work (mentally or physically) to accomplish your level of performance?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FRUSTRATION LEVEL</th>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel while doing the task?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. Which features of the email software helped you figure out why messages were classified in a given folder?

3. Which features of the email software helped you figure out how to classify a message to a different folder?

4. If an e-mail is automatically classified in the wrong folder, what can you do to fix the situation?
The following questions ask you to indicate whether you could use a new email classification system under a variety of conditions. For each of the conditions, please indicate whether you think you would be able to complete the task.

Given an email program that attempts to automatically categorize new messages into folders, such as “Work” and “Personal”, I could help train the computer to categorize mail accurately:

<table>
<thead>
<tr>
<th>... if there was no one around to tell me what to do as I go.</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>... if I had never used a similar email program before.</td>
<td>Strongly Disagree</td>
<td>Disagree</td>
<td>Neither Agree Nor Disagree</td>
<td>Agree</td>
<td>Strongly Agree</td>
</tr>
<tr>
<td>... if I had only the software manuals for references.</td>
<td>Strongly Disagree</td>
<td>Disagree</td>
<td>Neither Agree Nor Disagree</td>
<td>Agree</td>
<td>Strongly Agree</td>
</tr>
<tr>
<td>... if I had seen someone else doing it before trying it myself.</td>
<td>Strongly Disagree</td>
<td>Disagree</td>
<td>Neither Agree Nor Disagree</td>
<td>Agree</td>
<td>Strongly Agree</td>
</tr>
<tr>
<td>... if I could call someone for help if I got stuck.</td>
<td>Strongly Disagree</td>
<td>Disagree</td>
<td>Neither Agree Nor Disagree</td>
<td>Agree</td>
<td>Strongly Agree</td>
</tr>
<tr>
<td>... if someone else had helped me get started.</td>
<td>Strongly Disagree</td>
<td>Disagree</td>
<td>Neither Agree Nor Disagree</td>
<td>Agree</td>
<td>Strongly Agree</td>
</tr>
<tr>
<td>... if I had a lot of time to complete the task.</td>
<td>Strongly Disagree</td>
<td>Disagree</td>
<td>Neither Agree Nor Disagree</td>
<td>Agree</td>
<td>Strongly Agree</td>
</tr>
<tr>
<td>... if I had just the built-in help facility for assistance.</td>
<td>Strongly Disagree</td>
<td>Disagree</td>
<td>Neither Agree Nor Disagree</td>
<td>Agree</td>
<td>Strongly Agree</td>
</tr>
<tr>
<td>... if someone showed me how to do it first.</td>
<td>Strongly Disagree</td>
<td>Disagree</td>
<td>Neither Agree Nor Disagree</td>
<td>Agree</td>
<td>Strongly Agree</td>
</tr>
</tbody>
</table>
... if I had used similar programs before this one to do this same task.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree Nor Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>