Where Are My Intelligent Assistant's Mistakes? A Systematic Testing Approach

Todd Kulesza[1], Margaret Burnett[1], Simone Stumpf [2],
Weng-Keen Wong[1], Shubhomoy Das[1], Alex Groce[1],
Amber Shinsel[1], Forrest Bice[1], and Kevin McIntosh[1]

[1] School of EECS, Kelley Engr. Center, Oregon State University, Corvallis, OR 97331
{kuleszto,burnett,wong,dassh,
alex,shinsela,bice,mcintoke}@eecs.oregonstate.edu
[2] Centre for HCI Design, City University London, Northampton Square, London EC1V 0HB
Simone.Stumpf.1@city.ac.uk

Abstract. Intelligent assistants are handling increasingly critical tasks, but until now, end users have had no way to systematically assess where their assistants make mistakes. For some intelligent assistants, this is a serious problem: if the assistant is doing work that is important, such as assisting with qualitative research or monitoring an elderly parent's safety, the user may pay a high cost for unnoticed mistakes. This paper addresses the problem with WYSIWYT/ML (What You See Is What You Test for Machine Learning), a human/computer partnership that enables end users to systematically test intelligent assistants. Our empirical evaluation shows that WYSIWYT/ML helped end users find assistants' mistakes significantly more effectively than ad hoc testing. Not only did it allow users to assess an assistant's work on an average of 117 predictions in only 10 minutes, it also scaled to a much larger data set, assessing an assistant's work on 623 out of 1,448 predictions using only the users' original 10 minutes' testing effort.

Keywords: Intelligent assistants, end-user programming, end-user development, end-user software engineering, testing, machine learning.

1 Introduction

When using a customized intelligent assistant, how can an end user assess whether and in what circumstances to rely on its work?

Although this may seem at first glance to be merely a matter of providing live feedback, assessment cannot be treated so superficially when the assistant is performing a critical task. Yet until now, there has been no way for end users to *systematically* assess whether and how their customized intelligent assistants need to be mistrusted or fixed. Instead, the mechanisms available for user assessment have been strictly ad hoc: users have had only their gut reactions to what they serendipitously happen to notice.

In their perspectives on the future of end-user development, Klann et al. pointed to the need both for intelligent customizations and quality control in end-user development

M.F. Costabile et al. (Eds.): IS-EUD 2011, LNCS 6654, pp. 171–186, 2011.

[15]. In addition to Klann et al.'s arguments, there are at least three reasons why end-user assessment of today's customized assistants has become of key importance. First, no machine learning technique can yet prevent an intelligent assistant from making any mistakes. Since machine learning algorithms try to learn a concept from a finite sample of training data, issues like overfitting and the algorithm's inductive bias prevent an assistant from being 100% correct over future data. For instance, in [29], a good assistant is only 80-90% accurate.

Second, today's intelligent assistants are taking on increasingly important roles—roles in which, if the assistant goes awry, the user may bear significant costs and/or risks. For example, Gmail's new priority inbox decides which e-mail messages busy people can and cannot delay reading [10]. Other kinds of emerging assistants are moving toward helping with research itself, such as qualitatively "coding" (categorizing) natural language text [18]. Assistants are even approaching intelligent "aging-in-place" monitoring of safety status to enable geographically distant caregivers to support their aging parents [26] without being personally nearby. In this paper, we focus on end users who are willing to spend a modest amount of effort to assess assistants doing these kinds of critical tasks.

Third, if an assistant is making mistakes that are critical, the user may want to fix ("debug") the assistant, but effective debugging heavily relies on effective testing—the user needs to determine *where* the assistant's mistakes are, *when* their debugging efforts have fixed the mistakes, and *when* their previous testing and/or debugging may need to be revisited. Therefore, as we will explain in the next section, this paper maps the question of end-user assessment of a customized intelligent assistant to an *end-user testing* problem.

We thus present a human/computer partnership, inspired by the What You See Is What You Test (WYSIWYT) end-user testing methodology for spreadsheets [5, 25]. In our approach (WYSIWYT/ML), the system (1) *advises* the user about which predictions to test, then (2) *contributes* more tests "like" the user's, (3) *measures* how much of the assistant's reasoning has been tested, and (4) continually *monitors* over time whether previous testing still "covers" new behaviors the assistant has learned.

This paper makes the following contributions:

- We show how end-user assessment of intelligent assistants can be mapped to testing concepts. This mapping opens the door to potentially applying prior research on software testing to assistant assessment.
- We present our WYSIWYT/ML approach for helping users find *where* their assistant's mistakes are and monitoring *when* a previously reliable assistant may have gone astray.
- We present the first empirical evaluation of an end-user testing approach for assessing a user's evolving intelligent assistant.

Our empirical results showed significant evidence of the superiority of systematic testing in terms of efficiency and effectiveness—by one measure, improving users' efficiency by a factor of 10. These results strongly support the viability of this new method for end users to assess *whether* and *when* to rely on their intelligent assistants.

2 Intelligent Assistant Assessment as a Testing Problem

In this section, we show how assessing whether and when an intelligent assistant's outputs are right or wrong can be mapped to software testing. We adopt our terminology from the general literature of software testing [3], and in particular from the formulation of previous end-user testing problems [25].

According to the latest IEEE Standard [14], *testing* is "the process of [running] a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component"—i.e., running the program in a particular way (e.g., with given inputs) and evaluating the outputs.

The assistant is obviously a program, but it is an unusual kind of program in that it was, in part, automatically generated. Specifically, the intelligent assistants of interest to us are text classification assistants that output a single label for each textual input—in this domain, the programming process is as follows. First, the machine learning expert writes the assistant shell and learning algorithm, and tests or otherwise validates them to his or her satisfaction. The expert then runs the algorithm with an initial set of *training data* (here, labeled text examples) to automatically generate the first version of the assistant, which is the first version of the *program*. The assistant is then deployed to the end user's desktop.

At this point, the assistant's primary job, like that of most programs, is to read inputs (here, unlabeled text) and produce output (here, a label for that text). But unlike other programs, the assistant has a second job: to gather new training data from its user's actions to learn new and better logic—the equivalent of automatically generating a *new program*. Note that the machine learning expert is no longer present to test this new program—the newest version of the assistant learned behaviors from its specific end user *after* it had been deployed to the user's desktop. Thus, the end user (acting as an oracle [3]) is the only one present to test the program.

Given such a program, many of the testing concepts defined in Rothermel et al. [25] can be straightforwardly applied to our domain. A *test case* is the combination of an input (unlabeled text) and its output (a label). Given a test case that the program has executed, a *test* is an explicit decision by the end user about whether the output is correct for that test case's input. If the user decides that the output is not correct, this is (at least in the end user's eyes) evidence of a *bug* in the assistant's reasoning.

Testing would not be viable if every possible input/output pair must be tested individually—the space of all possible inputs is usually intractably large or possibly infinite. One solution has been to use the notion of coverage [3] to measure whether "enough" testing has been done. Along this line, consider a partitioning scheme that divides inputs into "similar" groups by some measure of similarity. A test case can then be said to *cover* all current (and future) input/output pairs for which the inputs are in the same group as the test case's input, and the outputs equal the test case's output.

Given these definitions, *systematic testing* differs in two important ways from the ad hoc testing that comes by serendipitously observing correct/incorrect behaviors: systematic testing has a measure (coverage) for ascertaining how "tested" the program is, and it provides a way to identify which test cases can increase that measure. In the spreadsheet paradigm, systematic testing by end users has been shown to be significantly more effective than ad hoc testing [5].

Finally, it is worth discussing how testing and debugging, while related, are distinctly separate activities. Testing, as we have explained, evaluates whether a program's outputs are right or wrong, whereas *debugging* is the act of actually fixing the program. Even without precisely mapping debugging of assistants to classic debugging (which is beyond the scope of this paper), it is clear that testing contributes to two phases that have been identified for debugging [19]: it contributes to the debugging phase of *finding* the bug by showing an instance of how/where a program is failing, and also contributes to the debugging phase of *validation* of whether the program has now stopped failing in that way. We envision our testing approach as contributing to both of these aspects of debugging.

3 Related Work

Testing of intelligent assistants is often done pre-deployment by machine learning specialists via statistical methods [13]. Such methods do not substitute for *end users*' assessment of their assistants because pre-deployment evaluation cannot assess suitability of after-deployment customizations to a particular user.

Some statistical debugging, however, can be automatically carried out after deployment. Research in machine learning has led to *active learning*, whereby an assistant can request the user to label the most informative training examples during the learning process [28]. Although one of our WYSIWYT/ML methods (Confidence) is sometimes used in active learning, most of our methods differ from active learning's. Our approach complements debugging techniques such as active learning, allowing the user (not the intelligent assistant) to assess whether and when the assistant is reliable.

Statistical outlier finding has been used in end-user programming settings for assessment, such as detecting errors in text editing macros [22], inferring formats from a set of unlabeled examples [27], and to monitor on-line data feeds in web-based applications for erroneous inputs [24]. These approaches use statistical analysis and interactive techniques to direct end-user programmers' attention to potentially problematic values, helping them find places in their programs to fix. Our approach also uses outlier finding, but does so as just one part of a larger approach that also systematically measures how much more assessment needs to be done.

Systematic testing for end users was pioneered by the *What You See Is What You Test* approach (WYSIWYT) for spreadsheet users [25]. To alleviate the need for users to conjure values for testing spreadsheet data, "Help Me Test" capabilities were added; these either dynamically generate suitable test values [7] or back-propagate constraints on cell values [1]. WYSIWYT inspired our approach in concept, but our under-the-hood reasoning about test prioritization and coverage are based on statistical properties of the assistant's behavior, rather than WYSIWYT's "white box" use of source code structure. Also, rather than helping users conjure new values to test, our approach instead aims to help users focus on just the right fraction of existing data to find important errors quickly.

To support end users' interactions with intelligent assistants, recent work has explored methods for explaining the reasons underlying an assistant's predictions. Such explanations have taken forms as diverse as *why…* and *why not…* descriptions of the

assistant's logic [17, 20], visual depictions of the assistant's known correct predictions versus its known failures [29], and electronic "door tags" displaying predictions of worker interruptibility with the reasons (e.g., "talking detected") [31]. As a basis for creating explanations, researchers have also investigated the types of information users want before assessing the trustworthiness of an intelligent agent [9, 18]. Recent work by Lim and Dey has resulted in a toolkit for applications to generate explanations for popular machine learning systems [21], and a few systems add debugging capabilities to explanations [17, 18]. Our approach for supporting systematic assessment of intelligent assistants is intended as a complement to explanation and debugging approaches like these.

4 The WYSIWYT/ML Approach

We have explained that without systematic testing, a user is left with only the ability to assess ad-hoc the assistant's predictions that they happen to notice. Ad-hoc testing does not help the user pick *which* items to test, nor does it help the user decide *how much more* testing should be done. WYSIWYT/ML targets both issues for situations in which an assistant's mistakes carry high risks or high costs for the user.

One such high-risk/high-cost situation is qualitative "coding" of verbal transcript data (a common HCI research task), in which empirical analysts segment written transcripts and categorize each segment. This is a labor-intensive activity requiring days to weeks of time—but what if an assistant could do part of this work (e.g., [18])? For example, suppose ethnographer Adam has an intelligent assistant that learns to code the way Adam does; the assistant could then finish coding Adam's transcripts. But Adam's research results may be invalid if the assistant's work is wrong, so he needs to assess where the assistant makes significant mistakes.

We prototyped WYSIWYT/ML as part of an intelligent "coding" assistant that classifies text messages, similar to Adam's hypothetical coding assistant. The assistant in our prototypes makes its predictions using a support vector machine, but the algorithm is not important—WYSIWYT/ML works with *any* algorithm (or accompanying feature set) that produces the information needed by the test prioritization methods described shortly.

4.1 How WYSIWYT/ML and Adam Work Together

Two Use-Cases. Given an intelligent classification assistant, WYSIWYT/ML's mission is to help the user assess its accuracy during two use cases.

Use case *UC-1*: In the assistant's early days, can Adam rely on it? After his assistant has been initially trained, Adam can use WYSIWYT/ML to decide whether it classifies messages consistently enough for his purposes. To minimize time spent finding the assistant's mistakes, WYSIWYT/ML advises him *which* messages the assistant believes it is weakest at classifying.

Use case *UC-2*: As the assistant continues to customize itself, can Adam *still* rely on it? As the assistant continues to learn and/or new messages arrive, WYSIWYT/ML keeps track of whether the assistant is working on messages very similar to (and

sharing the same output label as) those previously tested, or whether the assistant is now making predictions unlike those tested earlier. If the assistant is behaving differently than before, test coverage will be much lower and Adam might decide to systematically test some of the assistant's new work. WYSIWYT/ML helps him target these new predictions.

To support these use-cases, WYSIWYT/ML performs four functions: (1) it *advises* (prioritizes) which predictions to test, (2) it *contributes* tests, (3) it *measures* coverage, and (4) it *monitors* for coverage changes.

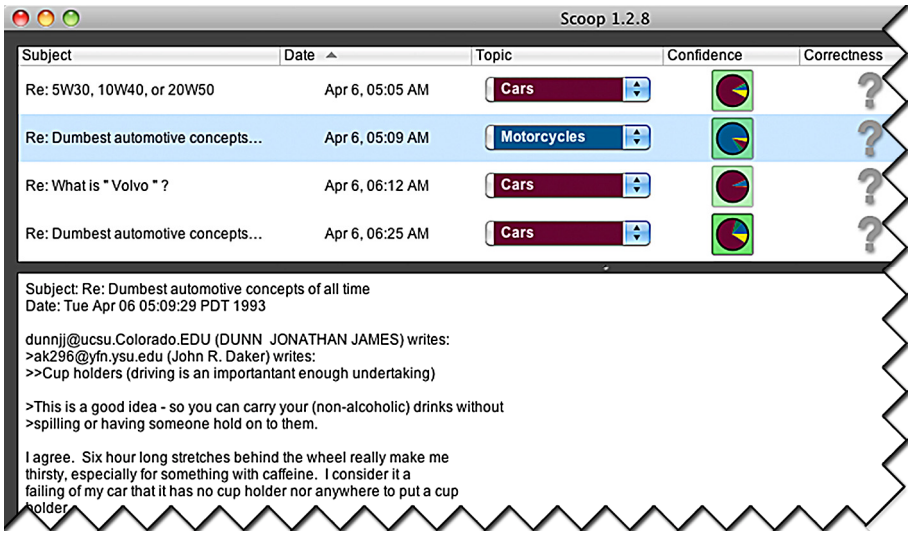

Fig. 1. The WYSIWYT/ML prototype. This variant uses the Confidence method.

WYSIWYT/ML Prioritizes Tests. WYSIWYT/ML prioritizes the assistant's topic predictions that are most likely to be wrong, and communicates these prioritizations using saturated green squares to draw Adam's eye (e.g., Figure 1, fourth message). The prioritizations may not be perfect, but they are only intended to be *advisory*; Adam is free to test any messages he wants, not just ones the system suggests.

To select prioritization methods, we first ran offline experiments using a "gold standard" oracle (rather than real users) to allow for numerous experiment runs. These experiments compared five candidate prioritization methods against randomization (where *Random* represents the statistical likelihood of finding mistakes). We selected the three best-performing methods, all of which outperformed Random: *Confidence*, *Similarity*, and *Relevance*.

The *Confidence* method leverages the assistant's knowledge of its own weaknesses, prioritizing messages based on the assistant's certainty that the topic it predicted is correct. (This is also a method used by active learning [28].) The higher the uncertainty, the more saturated the green square (Figure 1, Confidence column). Within the square, WYSIWYT/ML "explains" Confidence prioritizations using a pie chart (Figure 2, left). Each pie slice represents the probability that the message

belongs to that slice's topic: a pie with evenly sized slices means the assistant thinks each topic is equally probable (thus, testing it is a high priority).

The *Similarity* method selects "oddball" messages—those least similar to the data the assistant has learned from. The rationale is that if the assistant has never before seen anything like this message, it is less likely to know how to predict its topic. We measure this via cosine similarity [2], which is frequently used in information retrieval systems; here, it measures co-occurrences of the same words in different messages. A "fishbowl" explains this method's priority, with the amount of "water" in the fishbowl representing how unique the message is compared to messages on which the assistant trained (Figure 2, middle). A full fishbowl means the message is very unique (compared to the assistant's training set), and thus high priority.

The *Relevance* method is based on the premise that messages without useful words may not contain enough information for the assistant to accurately predict a topic. In machine learning parlance, useful words have high *information gain* (i.e., the words that contribute the most to the assistant's ability to predict the topic). We used the top 20 words from the messages the assistant learned from, then prioritized messages by the *lack* of these relevant words. Our prototype uses the number of relevant words (0 to 20) to explain the reason for the message's priority (Figure 2, right), with the lowest numbers receiving the highest priorities.

In our offline tests (without users), the Confidence method was the most effective: its high-priority tests were very successful at identifying flaws in an assistant's predictions, even when the assistant was 80% accurate. The Similarity and Relevance methods did not highlight as many bugs, but they outperformed Confidence in revealing hard-to-find bugs: items the assistant thought it was right about (predicted confidently), but which were wrong. We thus implemented all three, so as to empirically determine which is the most effective with real users.

Fig. 2. The Confidence (left), Similarity (middle), and Relevance (right) visualizations.

Fig. 3. A user can mark a predicted topic *wrong*, *maybe wrong*, *maybe right*, or *right* (or "?" to revert to untested). Prior research found these four choices to be very useful in spreadsheet testing [12].

Use-Case UC-1: Adam Tests His Assistant. When Adam wants to assess his assistant, he can pick a message and judge (i.e., test) the assistant's prediction. He can pick any message: one of WYSIWYT/ML's suggestions, or some different message if he prefers. Adam then communicates his judgment by clicking a *check* if it is correct or an *X* if it is incorrect, as in Figure 3. If a topic prediction is wrong, Adam has the option of selecting the correct topic—our prototype treats this as a shortcut for marking the existing topic as "wrong", making the topic change, and then marking the new topic as "right".

WYSIWYT/ML then *contributes* to Adam's testing effort: when Adam tests a message, WYSWYT/ML automatically infers the same judgment upon similar messages. These automated judgments constitute *inferred tests*.

To contribute these inferred tests, our approach computes the cosine similarity of the message Adam just tested with each untested message sharing the same predicted topic. WYSWYT/ML then marks very similar messages (i.e., scoring above a cosine similarity threshold of 0.05) as *approved* or *disapproved* by the assistant. The automatically inferred assessments are shown with gray check marks and X marks in the Correctness column (Figure 4, top), allowing Adam to differentiate his own explicit judgments from those automatically inferred by WYSIWYT/ML. Of course, Adam is free to review (and if necessary, fix) any inferred assessments—in fact, most of our study's participants started out doing exactly this.

WYSIWYT/ML's third functionality is *measuring* test coverage: how many of the assistant's predictions have been tested by Adam and the inferred tests together. A test coverage bar (Figure 4, bottom) keeps Adam informed of this measure, helping him decide how much more testing may be warranted.

WYSIWYT/ML also allows the assistant to leverage Adam's positive tests (his "right" and "maybe right" marks) as training data—an extra benefit for Adam. (Only Adam's explicit tests become training data, not WYSIWYT/ML's inferred tests.) As previously mentioned, however, collecting a few training instances in this way is not the point of WYSIWYT/ML. Our goal is to allow Adam to effectively and efficiently assess how much he can rely on the assistant, not to collect enough training instances to fix its flaws.

Use-Case UC-2: Adam: "It was reliable before; is it reliable now?" Adam's intelligent assistant continually learns from Adam's behaviors, changing its reasoning based upon Adam's feedback. The assistant may also encounter data unlike any it had seen before. Hence, for use-case UC-2, WYSIWYT/ML's fourth functionality is to *monitor* coverage over time, alerting Adam when a previously tested assistant is exposing behaviors that he has not yet assessed.

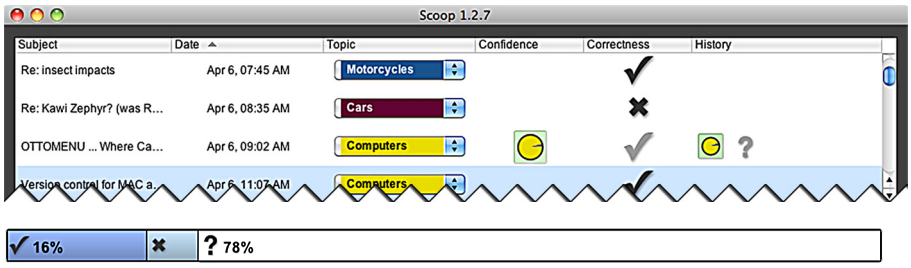

Fig. 4. (Top): The user tested three of the messages (the dark check marks and X marks), so they no longer show a priority. Then the computer inferred the third message to be correct (light gray check mark). Because the user's last test caused the computer to infer new information, the *History* column shows the prior values of what changed. (These values move right with each new change, until they are pushed off the screen.) (Bottom): A *test coverage bar* informs users how many topic predictions have been judged (by the user or the computer) to be correct (check mark) or incorrect (X mark).

Whenever Adam tests one of the assistant's predictions or new content arrives for the assistant to classify, the system immediately updates all of its information. This includes the assistant's predictions (except for those Adam "locked down" by explicitly approving them), all testing priorities, all inferred tests, and the test coverage bar. Thus, Adam can always see how "tested" the assistant is at any given moment. If he decides that more testing is warranted, he can quickly tell which predictions WYSIWYT/ML thinks are the weakest (UC-1) and which predictions are not covered by his prior tests (UC-2).

4.2 Cognitive Dimensions Analysis

We used Cognitive Dimensions [11], a popular analytical technique, to head off problems early in the design of our WYSIWYT/ML prototype. This analysis revealed four key issues for the implementation of WYSIWYT/ML, which we addressed as follows.

What just changed and how (Hard Mental Operations, Hidden Dependencies). Hard mental operations denote the user having to manually track or compute things in their head, and a hidden dependency is a link between two items that is not explicit. These dimensions revealed that a user could only answer the question "what just changed?" by scrolling extensively and memorizing prior statuses. To solve this, we added a History column showing the last two statuses of each message.

Too eager to help (Premature Commitment). This dimension denotes requiring users to make a decision before they have information about the decision's consequences. Because each user action may cause the assistant to update its predictions and WYSIWYT/ML to update its priority rankings and inferred tests, end user cannot easily guess the scope of alterations resulting from each interaction. In an early prototype, testing a prediction could cause on-screen messages to disappear from the user's view (due to the system automatically re-sorting messages based on new predictions or test priorities), making it difficult to see these consequences. Thus, we changed the prototype to only re-sort when the user asks it to (e.g., clicks the column header). This modification also helps emphasize recent changes, as other affected items in the sort "key" become visually distinct from their neighbors (e.g., now have a lower test priority than nearby messages).

Notes and scratches (Secondary Notation). Secondary notations allow users to annotate, change layout, etc., to communicate informally with themselves or with other humans in their environment (as versus communicating with the computer). We decided that secondary notation was unnecessary for end-user testing. As our empirical results will show, revisiting this decision may be warranted—some participants appeared to repurpose certain user interface affordances to indicate assistant predictions they intended to revisit later.

Communication overload (Role Expressiveness). This dimension denotes a user's ability to see how a component relates to the whole. This was initially a problem for our priority widget because it had too many roles: a single widget communicated the *priority* of assessing the message, explained *why* it had that priority, and *how* the message had been assessed—all in one small icon. Thus, we changed the prototype so

that no widget had more than one role. We added the Correctness column to show the user's (or computer's) assessment (Figure 1), the green square to represent priority, and the widgets inside to explain the reasoning behind the priority (Figure 2).

5 Empirical Study

We conducted a user study to investigate use-case UC-1, the user's initial assessment of an assistant doing important work. We attempted to answer three research questions to reveal how well ordinary end users could assess their assistants in this use-case, even if they invested only 10 minutes of effort:

RQ1 (Efficacy): Will end users, testing systematically with WYSIWYT/ML, find more bugs than via ad hoc testing?
RQ2 (Satisfaction): What are the users' attitudes toward systematic testing as compared to ad-hoc testing?
RQ3 (Efficiency): Will WYSIWYT/ML's coverage contributions to the partnership help with end users' efficiency?

We used three systematic testing treatments, one for each prioritization method (Confidence, Similarity, and Relevance). We also included a fourth treatment (Control) to represent ad hoc testing. Participants in all treatments could test (via check marks, X marks, and label changes) and sort messages by any column in the prototype. See Figure 1 for a screenshot of the Confidence prototype; Similarity and Relevance looked similar, save for their respective prioritization methods and visualizations (Figure 2). Control supported the same testing and sorting actions, but lacked prioritization visualizations or inferred tests, and thus did not need priority/inferred test history columns.

The experiment design was within-subject (i.e. all participants experienced all treatments). We randomly selected 48 participants (23 males and 25 females) from respondents to a university-wide request. None of our participants were Computer Science majors, nor had any taken Computer Science classes beyond the introductory course. Participants worked with messages from four newsgroups of the widely used 20 Newsgroups dataset [16]: *cars*, *motorcycles, computers,* and *religion* (the original rec.autos, rec.motorcycles, comp.os.ms-windows.misc, and soc.religion.christian newsgroups, respectively). This data set provides real-world text for classification, the performance of machine learning algorithms on it is well understood, and, most important, the "gold standard" topic choice (the newsgroup to which the message's author posted it) defines exactly which messages are "bugs" (misclassified by the assistant), in turn defining how many of those bugs participants found and when WYSIWYT/ML's inferred approvals went astray.

We randomly selected 120 messages (30 per topic) to train the intelligent assistant using a support vector machine [6]. We randomly selected a further 1,000 messages over a variety of dates (250 per topic) and divided them into five data sets: one tutorial set (to familiarize our participants) and four *test sets* (to use in the main tasks). Our intelligent assistant was 85% accurate when initially classifying each of these sets. We used a Latin Square to counterbalance treatment orderings and randomized how each participant's test data sets were assigned to the treatments.

Participants answered a background questionnaire, then took a tutorial to learn one prototype's user interface and to experience the kinds of messages and topics they would be seeing during the study. Using the tutorial set, participants practiced testing and finding the assistant's mistakes in that prototype. For the first main task, participants used the prototype to test and look for mistakes in a 200-message test set. After each treatment, participants filled out a Likert-scale questionnaire with their opinions of their success, the task difficulty, and the prototype. They then took another brief tutorial explaining the changes in the next prototype, practiced, and performed the main task in the next assigned data set and treatment. Finally, participants answered a questionnaire covering their overall opinions of the four prototypes and comprehension.

Table 1. ANOVA contrast results (against Control) by treatment. The highest values in each row are shaded.

	Mean (p-value for contrast with Control)				df	F	p
	Confidence	Similarity	Relevance	Control			
Bugs Found (max 30)	12.2 (p<.001)	10.3 (p<.001)	10.0 (p<.001)	6.5 (N/A)	3, 186	10.61	<.001
Helpfulness (max 7)	5.3 (p<.001)	5.0 (p<.001)	4.6 (p<.001)	2.9 (N/A)	3, 186	22.88	<.001
Perceived Success (max 21)	13.4 (p=.016)	13.3(p=.024)	14.0 (p=.002)	11.4 (N/A)	3, 186	3.82	.011

6 Results

6.1 RQ1 (Efficacy): Finding Bugs

Bugs Found. To investigate how well participants managed to find an assistant's mistakes using WYSIWYT/ML, we compared bugs they found using the WYSIWYT/ML treatments to bugs they found with the Control treatment. An ANOVA contrast against Control showed a significant difference between treatment means (Table 1). For example, participants found nearly twice as many bugs using the frontrunner, Confidence, than using the Control version.

Not only did participants find more bugs with WYSIWYT/ML, the more tests participants performed using WYSIWYT/ML, the more bugs they found (linear regression, $F(1,46)=14.34$, $R^2=.24$, beta=0.08, p<.001), a relationship for which there was no evidence in the Control variant (linear regression, $F(1,45)=1.56$, $R^2=.03$, beta=0.03, p=.218). Systematic testing using WYSIWYT/ML yielded significantly better results for finding bugs than ad-hoc testing.

Profile of a Hard Bug. Our formative offline oracle experiments revealed types of bugs that would be hard for some of our methods to target as high-priority tests. (Recall that, offline, Relevance and Similarity were better than Confidence in this respect.) In order to evaluate our methods with real users, we took a close look at Bug 20635, which was one of the hardest bugs for our participants to find (one of the five

least frequently identified). The message topic should have been Religion but was instead predicted to be Computers, perhaps in part because Bug 20635's message was very short and required domain-specific information to understand (which was also true of the four other hardest bugs):

> *Subject: Mission Aviation Fellowship*
> *Hi, Does anyone know anything about this group and what they do? Any info would be appreciated. Thanks!*

As Table 2 shows, nearly all participants who had this bug in their test set found it with the Relevance treatment, but a much lower fraction found it using the other treatments. As the table's Prioritization column shows, Relevance ranked the message as very high priority because it did not contain any useful words, unlike Confidence (the assistant was very confident in its prediction), and unlike Similarity (the message was fairly similar to other messages). Given this complementarity among the different methods, we hope in the future to evaluate a combination (e.g., a weighted average or voting scheme) of prioritization methods, thus enabling users to quickly find a wider variety of bugs than they could using any one method alone.

6.2 RQ2 (Satisfaction): User Attitudes

Participants appeared to recognize the benefits of systematic testing, indicating increased satisfaction over ad hoc testing. When asked "*How much did each system help you find the computer's mistakes?*" on a seven-point Likert scale, an ANOVA contrast again confirmed that responses differed between treatments (Table 1, row 2), with WYSIWYT/ML treatments rated more helpful than Control. Table 1's 3rd row shows that participant responses to the NASA-TLX questionnaire triangulate this result. Together, these results are encouraging from the perspective of the Attention Investment Model—they suggest that end users can be apprised of the benefits (so as to accurately weigh the costs) of testing an assistant that does work important to them.

Table 2. The number of participants who found Bug 20635 while working with each WYSIWYT/ML treatment

Treatment	Prioritization	Found	Did not find
Confidence	0.14	9	15
Similarity	0.58	11	14
Relevance	1.00	19	4

6.3 RQ3 (Efficiency): The Partnership's Test Coverage

Recall that when a participant tested a message, the system partnered with the user by inferring additional tests to "cover" similar messages (recall Figure 4). Coverage can be a powerful concept: it enables a user to reduce the number of items they must look over while still gaining a reasonable understanding of the assistant's reliability. It also reveals the weaknesses of an assistant's reasoning in terms of areas not yet covered by tests. In other domains, research has generally found that increased coverage increases bug finding [5, 8, 25]. Thus, in this section, we consider how much coverage the partnership achieved and how this related to participants' efficiency.

Table 3. Tests via check marks, X marks, and topic changes during a 10-minute session (out of 200 total messages per session), for the three WYSIWYT/ML treatments

	Mean √s participants entered per session	Mean Xs participants entered per session	Mean √s inferred per session	Mean Xs inferred per session	Total √s	Total Xs
Explicit	Regular: 35.0 "Maybe": 7.1	Regular: 2.4 "Maybe": 2.7	Regular: 46.4 "Maybe": 8.5	Regular: 4.7 "Maybe": 2.2	105.2	20.2
Implicit	8.2 topic changes as shortcuts for X+topic+√		N/A[1]			
Total tests	50.3	13.3	54.9	6.9		
Total messages tested [2]					117.2	

[1] Although the computer sometimes did change topics, this was due to leveraging tests as increased training on message classification. Thus, because these topic changes were not directly due to the coverage (cosine-similarity) mechanism, we omit them from this coverage analysis.

[2] *Total Tests* is larger than *Messages Tested* because topic changes acted as two tests: an X on the original topic, then a √ on the new topic.

Coverage: How much? Using WYSIYWT/ML, our participants were able to leverage their explicit tests by a factor of about 2. Together with the computer-oracle-as-partner, participants' mean of 55 test actions using WYSIWYT/ML covered a mean of 117 (60%) of the messages—thus, participants gained 62 inferred tests "for free". Table 3 shows the raw counts. With the help of their computer partners, two participants even reached 100% test coverage, covering all 200 messages within their 10-minute time limit.

Further, coverage scaled well. In an offline experiment, we tried our participants' explicit tests on the *entire* set of Newsgroup messages from the dates and topics we had sampled for the experiment—a data set containing 1,448 messages. (These were tests participants explicitly entered using either WYSIWYT/ML or Control, a mean of 55 test actions per session.) Using participants' 55 explicit tests (mean), the computer inferred a mean of 568 tests per participant, for a total coverage of 623 tests (mean) from only 10 minutes of work—a 10-fold leveraging of the user's invested effort.

Participant and WYSIWYT/ML Approvals vs. Disapprovals. As Table 3 shows, participants approved more messages than they disapproved. When participants approved a message, their topic choice matched the 20-Newsgroup "gold standard" (the original newsgroup topic) for 94% of their regular checkmarks and 81% of their "maybe" checkmarks (the agreement level across both types of approval was 92%). By the same measure, WYSIWYT/ML's approvals were also very accurate, agreeing with the gold standard an average 92% of the time—exactly the same level as the participants'.

Participants' regular X marks agreed with the gold standard reasonably often (77%), but their "maybe" X marks agreed only 43% of the time. Informal pilot interviews revealed a possible explanation: re-appropriation of the "maybe" X marks for a subtly different purpose. When unsure of the right topic, pilot participants said they marked it as "maybe wrong" to denote that it *could* be wrong, but with the intention to revisit it later. This indicates that secondary notation (in addition to testing notation)—in the form of a "reminder" to revisit instead of a disapproval—could prove useful in future prototypes.

Perhaps in part for this reason, WYSIWYT/ML did not correctly infer many bugs—only 19% of its X marks agreed with the gold standard. (The computer's regular X marks and "maybe" X marks did not differ—both were in low agreement with the gold standard.) The problem cannot be fully explained by participants repurposing "maybe" X marks—WYSIWYT/ML's regular inferred X marks were just as faulty. However, this problem's impact was limited because inferred X marks only serve to highlight possible bugs. Thus, the 81% failure rate on WYSIWYT/ML's average of seven X's per session meant that participants only had to look at an extra five messages/session. Most inferred tests were the very accurate *positive* tests (average of 55 per session), which were so accurate, participants could safely skip them when looking for bugs.

7 Discussion

Will end users *really* explicitly and systematically test an intelligent assistant? Although we did not test this question in our lab study, theory suggests that they will when the perceived benefits of doing so outweigh the costs [4]. Until this question can be investigated empirically, we target the subset of end users who are willing to expend at least modest effort to assess assistants on tasks in which mistake types and frequencies *must* be understood before the user would be willing to rely on them, such as with Adam's intelligent qualitative coding assistant.

Our current similarity-based notion of coverage also warrants further empirical investigation. It worked well for approvals, but a smaller threshold for disapprovals may result in fewer false bug identifications. In the future, we plan a systematic evaluation of this threshold and its impact on different aspects of WYSIWYT/ML.

Finally, we emphasize that finding (not fixing) bugs is WYSIWYT/ML's primary contribution toward debugging. Although WYSIWYT/ML leverages user tests as additional training data, simply adding training data is not an efficient method for *debugging* intelligent assistants. To illustrate, our participants' *testing* labeled, on average, 55 *messages,* which increased average accuracy by 3%. In contrast, participants in another study that also used a subset of the 20 Newsgroup dataset spent their 10 minutes *debugging* by specifying *words/phrases* associated with a label [32]. They entered only about 32 words/phrases but averaged almost twice as much of an accuracy increase (5%) in their 10 minutes. Other researchers have similarly reported that allowing users to debug by labeling a word/phrase is up five times more efficient than simply labeling training messages [23]. Thus, rather than attempting to replace the interactive debugging approaches emerging for intelligent assistants (e.g., [17, 18, 22, 30]), WYSIWYT/ML's bug-finding complements them. It provides the missing testing piece, suggesting where important bugs have emerged and when those bugs have been eradicated, so that end users need not debug blindly.

8 Conclusion

With the increase in intelligent assistants helping with critical tasks comes the need to rethink the nature of how end users can assess whether and when to rely on their assistants' help. WYSIWYT/ML is the first work to address this need.

WYSIWYT/ML is a human/computer partnership that enables end users to assess intelligent assistants systematically. The human's role is to approve or disapprove (i.e., test) portions of the assistant's work. The computer's role is to *advise* the user about testing priorities, *contribute* additional tests similar to the user's (which the user may verify), *measure* how much of the assistant's reasoning has been assessed, and *monitor* the need for additional assessment as the assistant evolves over time.

Our empirical evaluation showed that systematically testing with WYSIWYT/ML resulted in a significant improvement over ad hoc methods in end users' abilities to assess their assistants: our participants found almost twice as many bugs with our best WYSIWYT/ML variant as they did while testing ad hoc. Further, the approach scales: participants covered 117 messages in the 200-message data set (over twice as many as they explicitly tested) and 623 messages in the 1448-message data set (over 10 times as many as they explicitly tested)—all at a cost of only 10 minutes work.

Thus, systematic assessment of intelligent assistants was not only effective at finding bugs—it also helped ordinary end users assess a reasonable fraction of an assistant's work in a matter of minutes. These findings strongly support the viability of bringing systematic testing to this domain, empowering end users to judge whether and when to rely on intelligent assistants that support critical tasks.

Acknowledgements. We thank Jeremy Goodrich, Travis Moore, Shalini Shamasunder, Nicole Usselman, Chaoqiang Zhang, and our study participants for their help. This work has been supported in part by NSF 0803487.

References

[1] Abraham, R., Erwig, M.: AutoTest: A tool for automatic test case generation in spreadsheets. In: Proc. VL/HCC, pp. 43–50. IEEE, Los Alamitos (2006)
[2] Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
[3] Beizer, B.: Software Testing Techniques. International Thomson Computer Press (1990)
[4] Blackwell, A.: First steps in programming: A rationale for attention investment models. In: Proc. HCC, pp. 2–10. IEEE, Los Alamitos (2002)
[5] Burnett, M., Cook, C., Rothermel, G.: End-user software engineering. Comm. ACM 47(9), 53–58 (2004)
[6] Chang, C., Lin, C.: LIBSVM: A library for support vector machines (2001), `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
[7] Fisher, M., Cao, M., Rothermel, G., Brown, D., Cook, C., Burnett, M.: Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. ACM Trans. Software Engineering and Methodology 15(2), 150–194 (2006)
[8] Frankl, P., Weiss, S.: An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Trans. Software Eng. 19(3), 202–213 (1993)
[9] Glass, A., McGuinness, D., Wolverton, M.: Toward establishing trust in adaptive agents. In: Proc. IUI, pp. 227–236. ACM, New York (2008)
[10] Gmail Priority Inbox: Get through your email faster, `http://google.com/mail/help/priority-inbox.html` (accessed September 16, 2010)
[11] Green, T., Petre, M.: Usability analysis of visual programming environments: A cognitive dimensions framework. J. Visual Languages and Computing 7(2) (June 1996)

[12] Grigoreanu, V., Cao, J., Kulesza, T., Bogart, C., Rector, K., Burnett, M., Wiedenbeck, S.: Can feature design reduce the gender gap in end-user software development environments? In: Proc. VL/HCC, pp. 149–156. IEEE, Los Alamitos (2008)
[13] Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning. Springer, Heidelberg (2003)
[14] IEEE, IEEE Standard Glossary of Software Engineering Terminology (IEEE Std610.12-1990) (1990)
[15] Klann, M., Paterno, F., Wulf, V.: Future perspectives in end-user development. In: Lieberman, H., Paterno, F., Wulf, V. (eds.) End-User Development. Springer, Heidelberg (2006)
[16] Kniesel, G., Rho, T.: Newsgroup data set (2005), http://www.ai.mit.edu/jrennie/20newsgroups
[17] Kulesza, T., Wong, W., Stumpf, S., Perona, S., White, R., Burnett, M., Oberst, I., Ko, A.: Fixing the program my computer learned: Barriers for end users, challenges for the machine. In: Proc. IUI, pp. 187–196. ACM, New York (2009)
[18] Kulesza, T., Stumpf, S., Burnett, M., Wong, W., Riche, Y., Moore, T., Oberst, I., Shinsel, A., McIntosh, K.: Explanatory debugging: Supporting end-user debugging of machine-learned programs. In: Proc. VL/HCC. IEEE, Los Alamitos (2010)
[19] Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., Fleming, S.: How programmers debug, revisited: An information foraging theory perspective. IEEE Trans. Software Engineering (2011)
[20] Lim, B., Dey, A., Avrahami, D.: Why and why not explanations improve the intelligibility of context-aware intelligent systems. In: Proc. CHI, pp. 2119–2128. ACM, New York (2009)
[21] Lim, B., Dey, A.: Toolkit to support intelligibility in context-aware applications. In: Proc. Int. Conf. Ubiquitous Computing. ACM, New York (2010)
[22] Miller, R., Myers, B.: Outlier finding: Focusing user attention on possible errors. In: Proc. UIST, pp. 81–90. ACM, New York (2001)
[23] Raghavan, H., Madani, O., Jones, R.: Active learning with feedback on both features and instances. JMLR 7, 1655–1686 (2006)
[24] Raz, O., Koopman, P., Shaw, M.: Semantic anomaly detection in online data sources. In: Proc. ICSE, pp. 302–312. IEEE, Los Alamitos (2002)
[25] Rothermel, G., Burnett, M., Li, L., Dupuis, C., Sheretov, A.: A methodology for testing spreadsheets. ACM Trans. Software Engineering and Methodology 10(1) (January 2001)
[26] Rowan, J., Mynatt, E.: Digital family portrait field trial: Support for aging in place. In: Proc. CHI, pp. 521–530. ACM, New York (2005)
[27] Scaffidi, C.: Unsupervised inference of data formats in human-readable notation. In: Proc. Int. Conf. Enterprise Integration Systems, pp. 236–241 (2007)
[28] Settles, B.: Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison (2009)
[29] Shen, J., Dietterich, T.: Active EM to reduce noise in activity recognition. In: Proc. IUI, pp. 132–140. ACM, New York (2007)
[30] Talbot, J., Lee, B., Kapoor, A., Tan, D.: EnsembleMatrix: Interactive visualization to support machine learning with multiple classifiers. In: Proc. CHI, pp. 1283–1292. ACM, New York (2009)
[31] Tullio, J., Dey, A., Chalecki, J., Fogarty, J.: How it works: A field study of non-technical users interacting with an intelligent system. In: Proc. CHI, pp. 31–40. ACM, New York (2007)
[32] Wong, W.-K., Oberst, I., Das, S., Moore, T., Stumpf, S., McIntosh, K., Burnett, M.: End-user feature labeling: A locally-weighted regression approach. In: Proc IUI. ACM, New York (2011)