Toward End-User Debugging of Machine-Learned Classifiers

Todd Kulesza
Oregon State University
kuleszto@eecs.oregonstate.edu

Abstract

Many machine-learning algorithms learn rules of behavior from individual end users, such as task-oriented desktop organizers and handwriting recognizers. These rules form a generated "program" tailored specifically to the behaviors of that end user, telling the computer what to do when future inputs arrive. Researchers, however, have only recently begun to explore how an end user can debug these programs when they make mistakes. We present our progress toward enabling end users to test and debug learned programs so that everyone can benefit from intelligent programs adapted to their specific tasks and situations.

1. Introduction

An increasingly wide range of end-user applications use machine-learning techniques to adapt to the user or automate repetitive tasks. Tools such as facial recognition in photography programs, junk mail filtering, and recommendation systems all generate rules of behavior based on the specific idiosyncrasies of a particular end user. The resulting classifier can thus be thought of as a *learned program*: like traditional programs, outputs are determined by the learned program's internal logic operating on given inputs. Research into end-user programming has sought to democratize creation of computational tools in domains such as web programming and spreadsheets. Learned programs, tailored to an end user's individual behaviors, are a new generation of end-user programs.

When a learned program makes a mistake, such as misclassifying an e-mail message as junk, who can fix it? Unlike traditional software, a learned program has never been seen by a human programmer: it was *generated* by a learning algorithm. Further, this generated program is specific to an end user's behavior. Thus, the particular end user is the only oracle for judging whether the program is behaving properly, and if not, is the only person with the knowledge to properly correct it.

Thus, unless the learned program is perfect (which is unlikely), we believe the end user must be able to debug the program's faults. This presents an immediate challenge: in a learned program, what *are* faults? The generating algorithm may appear to operate exactly as intended, but the generated program may still contain logic that does not match the end user's intentions. It is in this learned logic, then, that the fault lies, and thus this logic is what the user must be able to correct. While a machine learning expert may be able to quickly scan a misclassified e-mail and conclude likely reasons for the learned program's error (such as the presence of a unique word that is frequently associated with junk mail), how can such reasoning be communicated to an end user? Furthermore, how can that user explain back to the computer 1) that what it did was wrong, and 2) *why* it was wrong, so as to avoid similar failures in the future?

Our approach to support end users debugging machine-learned programs consists of three components: user testing of the learned program to find failures (wrong behaviors and outputs), machine-generated explanations of the program's logic to uncover faults (reasons for the failures), and support for corrective user feedback to fix the detected faults.

2. Communicating with learned programs

Our research has thus far focused on the linked questions of whether a learned program can successfully explain its reasoning to an end user, and whether a user armed with such knowledge can adjust the program's logic to satisfactorily match the user's expectations. One of the central issues involved in answering these questions is determining the types of information presented to end users, and how this information should be represented.

We have run a pair of user studies with different representations of a learned program's "source code". One used an interactive bar graph showing the importance of each word to the program's prediction [1]; if participants disagreed with the machine-assigned importance, they could adjust each bar to fix the program's logic. A second study used textual descriptions of the most important words and allowed users to highlight new words (indicating importance) or remove currently important words. Neither study yielded significant improvements in classification

978-0-7695-4206-5/10 $26.00 © 2010 IEEE
DOI 10.1109/VLHCC.2010.45

IEEE computer society

accuracy: participants were only marginally more likely to make the program better as they were to harm its accuracy.

Is explaining the program's logic enough, or do other types of information lead to better debugging success? Just as professional software engineers would be at a disadvantage debugging a program via source code modifications alone (as opposed to viewing the program's runtime state with a debugger), we felt end user debuggers would benefit from knowledge of the program's execution state. The second study described above included a treatment presenting a collection of runtime information to participants, such as the machine's confidence in its predictions. Participants using this treatment *did* significantly improve the accuracy of their learned program, suggesting that "source code" representations may be better understood by users when paired with additional information about the program's current runtime state.

Our preliminary research has begun enumerating barriers that end users encounter while debugging a learned program, such as not knowing how to select specific areas of the program's logic to fix, and having difficulty coordinating how those fixes affect the program's predictions [1]. Efforts to overcome these barriers include approaches for steering users toward the sections of a program's logic most responsible for a given prediction, as well as determining in real time whether a specific user correction will help or harm the accuracy of the learned program.

3. Testing learned programs

How does a user know whether or not to trust the predictions of a learned program? Lacking an answer to this question, the user would not even know if they *should* debug the program. Without systematic testing, a user's most obvious recourse is to base trust on the number of good and bad predictions the program has made in the past, but this raises two issues: 1) such ad-hoc testing may not exercise many areas of the learned program's logic, the equivalent of (perhaps severely) incomplete software testing coverage, and 2) as the learned program continues to adapt to a user's behavior, the changes introduced may cause previously good predictions to become incorrect, and vice-versa. To come back to our e-mail example, a user needs to know whether to continually check the Junk folder for important, misclassified messages. Furthermore, every time the learned program changes, the user may need to re-evaluate whether *now* they can trust the program's output.

To address these issues, we are revisiting the What You See Is What You Test (WYSIWYT) approach, successfully used to support end-user programmers testing spreadsheets [2], to understand whether such an approach can be applied to test learned programs. A great strength of WYSIWYT is its ability to bring the benefits of structured software testing to end-user programming environments, allowing end users to directly benefit from software engineering best practices. This aligns precisely with our goal of showing end users in real time which of the program's predictions they can trust, and which may be suspect. When combined with the debugging approaches outlined above, we hope that WYSIWYT/ML will serve a dual purpose by also enabling end users to quickly find the areas of a program's logic that are responsible for erroneous predictions (i.e., the areas that most need to be debugged).

Our work on WYSIWYT/ML began with an exploration of test prioritization metrics largely unique to machine learning, such as prioritizing tests based on a classifier's confidence in each prediction (A *test* in this new domain represents the user verifying one of the program's predictions as either right or wrong). Offline tests revealed that some of our prioritizations were surprisingly accurate (up to 80%) at selecting incorrect predictions for users to test. We are now preparing a prototype for a user study to evaluate three of the most promising prioritizations: classifier confidence, Euclidean similarity of test data, and absence of words the program considers highly relevant to classification.

Test prioritization is supported by the notion of *test coverage*. We have developed a method employing cosine similarity to identify especially similar predictions. A single user test may thus cover multiple predictions, relieving the end user of the burden of examining every machine-made prediction. This aspect is critical to any testing effort because the number of eventual inputs will be infinite: a systematic way to cover multiple predictions with one test is necessary.

We soon plan to empirically evaluate our approach via the upcoming user study, the results of which will inform our attempts to meld support for end-user testing and end-user debugging of machine-learned programs. It is our hope that such support will lead to uniquely adapted machine-learned programs that end users can test, debug, and trust.

4. References

[1] Kulesza, T., Wong, W., Stumpf, S., Perona, S., White, R., Burnett, M. M., Oberst, I., and Ko, A. J. Fixing the program my computer learned: barriers for end users, challenges for the machine. In Proc. IUI, ACM (2009), 187-196.

[2] Rothermel, G., Li, L., DuPuis, C., and Burnett, M.. What you see is what you test: a methodology for testing form-based visual programs. In Proc. ICSE, ACM (1998). 198-207.