ModelTracker: Redesigning Performance Analysis Tools for Machine Learning

Saleema Amershi, Max Chickering, Steven M. Drucker, Bongshin Lee, Patrice Simard, Jina Suh
Microsoft Research
Redmond, WA
{samershi, dmax, sdrucker, bongshin, patrice, jinsuh}@microsoft.com

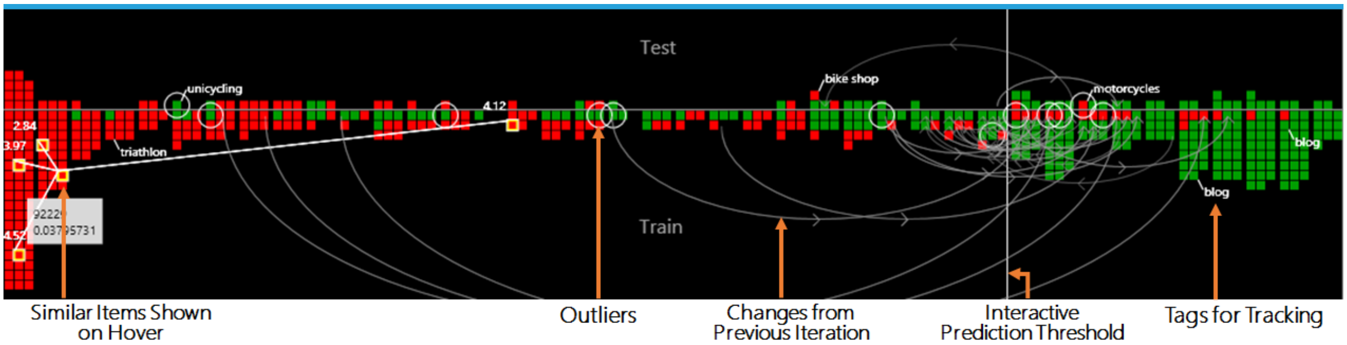

Figure 1. ModelTracker conveys overall model performance while enabling direct data inspection. Boxes represent user labeled examples and color indicates the label given (green for *positive* and red for *negative*). *Test* examples are placed at the top and *train* examples at the bottom. Examples are laid out horizontally according to the model's prediction scores, with low scoring examples to the left and high scoring examples to the right. A high performing model will have most green boxes to the right and most red boxes to the left of the display. Users can interact directly with ModelTracker to reveal additional information (e.g., hovering over an example reveals its nearest neighbors in the current feature space), inspect examples (by clicking on boxes to pull up the corresponding raw data in the display), and annotate examples for better performance tracking.

ABSTRACT

Model building in machine learning is an iterative process. The performance analysis and debugging step typically involves a disruptive cognitive switch from model building to error analysis, discouraging an informed approach to model building. We present ModelTracker, an interactive visualization that subsumes information contained in numerous traditional summary statistics and graphs while displaying example-level performance and enabling direct error examination and debugging. Usage analysis from machine learning practitioners building real models with ModelTracker over six months shows ModelTracker is used often and throughout model building. A controlled experiment focusing on ModelTracker's debugging capabilities shows participants prefer ModelTracker over traditional tools without a loss in model performance.

Author Keywords

Machine Learning; Interactive Visualization; Performance Analysis; Debugging.

CHI 2015, April 18 - 23 2015, Seoul, Republic of Korea
Copyright 2015 ACM 978-1-4503-3145-6/15/04…$15.00 .
http://dx.doi.org/10.1145/2702123.2702509

ACM Classification Keywords

H.5.2. Information interfaces and presentation (e.g., HCI): User interfaces.

INTRODUCTION

Machine learning is an iterative process. In supervised machine learning, practitioners iteratively collect and label a sample of data, create features to represent the data, train a model with the data and features, and then inspect the model's performance to determine how to proceed in the next iteration (e.g., collecting more data, adding/editing features, experimenting with a different learning algorithm). Once a model has achieved a sufficient level of performance, typically determined by its intended use, the model can be deployed in the target application (e.g., relevance ranking, activity detection, recommendations).

The performance inspection step of the machine learning process can itself be quite involved. Performance inspection typically begins with an assessment of a model's overall ability to correctly predict labels on data, often represented with summary statistics or graphs of common metrics (e.g., accuracy values, precision-recall curves). If summary metrics indicate poor performance, a practitioner may decide to continue iterating in a trial-and-error manner or debug the model by examining its behavior and trying to diagnose problems to inform the next iteration [8,18].

Debugging model performance typically requires a disruptive cognitive switch from the primary task of building a model to the task of analyzing prediction errors (i.e., examples whose user-provided labels are predicted incorrectly by the model). For example, performance analysis may involve first locating errors within large datasets via sorting and filtering (e.g., sorting by model prediction scores, filtering by errors types) and then inspecting raw data to form hypotheses about potential causes of those errors. Many tools have also been created to facilitate deeper model analysis and error debugging (e.g., dimensionality reduction [20,7,4] and scatterplot analysis [12,6]). Such tools, however, are often more complex and heavy weight than inspecting raw data [7]. Switching between these extremes of viewing summary statistics to debugging model performance behavior can result in a loss of context and disrupts the flow of model building [18]. These disruptions can be partially mitigated with integrated environments supporting both model building and error analysis tasks. However, existing integrated environments only support coarse statistics and fine-grained raw data inspection tools (e.g., [16,21]), thereby still requiring mode switches between model building and performance analysis. In practice, therefore, performance analysis and debugging is often performed only after trial-and-error strategies fail rather than to drive subsequent iterations of model building.

In this paper, we present ModelTracker (Figure 1), an interactive visualization designed to encourage a more informed approach to model building in machine learning. ModelTracker subsumes information contained within numerous traditional summary statistics and graphs while displaying example-level performance within a single compact visualization. It also supports direct error examination and debugging, reducing disruption caused by context switching from model building to error analysis.

ModelTracker is not algorithm or data type specific, requiring only prediction scores and user labels on data to visualize performance. It can therefore support a variety of supervised machine learning tasks and algorithms. In this paper, we examine ModelTracker within the context of a general purpose machine learning environment called ICE (Interactive Classification and Extraction) [21].

This paper makes the following contributions:

- ModelTracker, a generally applicable interactive visualization supporting performance analysis during model building in supervised machine learning while enabling direct error examination and debugging.
- Usage behavior from machine learning practitioners building real models (for research and product deployments) with ModelTracker over a six month period. Observations and feedback show that practitioners interacted with ModelTracker throughout model building, indicating that ModelTracker's support for performance analysis and direct access to data helps circumvent a disruptive context switch between summary statistics and separate data inspection tools.
- A controlled experiment comparing ModelTracker's support for direct debugging of common sources of errors against traditional summary statistics and separate debugging tools, showing that participants prefer ModelTracker over traditional tools without a loss in model performance or debugging ability.

RELATED WORK

In this section, we review common techniques for conveying model performance in machine learning and related tools for debugging performance issues.

Conveying Model Performance

Summary statistics are the most common technique for conveying model performance in machine learning. Common statistics include accuracy, precision, recall, F-scores, and area under the curve (AUC). Sometimes these are plotted over model variables (e.g., precision-recall curves are plotted over prediction thresholds (Figure 2)). Confusion matrices are another popular tool for conveying model performance by contrasting user provided labels with predicted labels (using a fixed prediction threshold) in a table, displaying the frequency of examples categorized within each cell. For example, in binary classification, confusion matrices are displayed using a 2x2 grid (Figure 2) showing the number of user-model agreements along the diagonal and false positives and negatives off the diagonal. Sometimes color or shading is also used to emphasize cell counts, drawing attention to problematic categories [22].

Most popular machine learning toolkits provide built-in functionality for computing summary statistics and confusion matrices because they provide an efficient and consistent means of conveying performance over a wide variety of tasks (e.g., Weka [11], Matlab [13], R [19]). However, while summary statistics and confusion matrices can convey the presence of errors, they do not indicate severity or the potential causes of those errors. Practitioners must therefore use separate tools to locate and examine individual errors. Summary statistics therefore lend themselves well to trial-and-error, rather than an informed approach to model building [8,18]. ModelTracker bridges the gap between performance analysis and debugging, reducing disruptions caused by context switching.

Debugging Performance Issues

Debugging model performance often involves direct examination of individual prediction errors [3,18]. Errors are typically located by sorting, filtering, and grouping operations over raw data in a separate display (e.g., sorting by prediction scores, filtering by error category). Once errors are located, they can be examined and hypotheses can be generated about potential causes. This process of locating and examining individual errors can often be tedious, particularly as the size of the data set grows. In

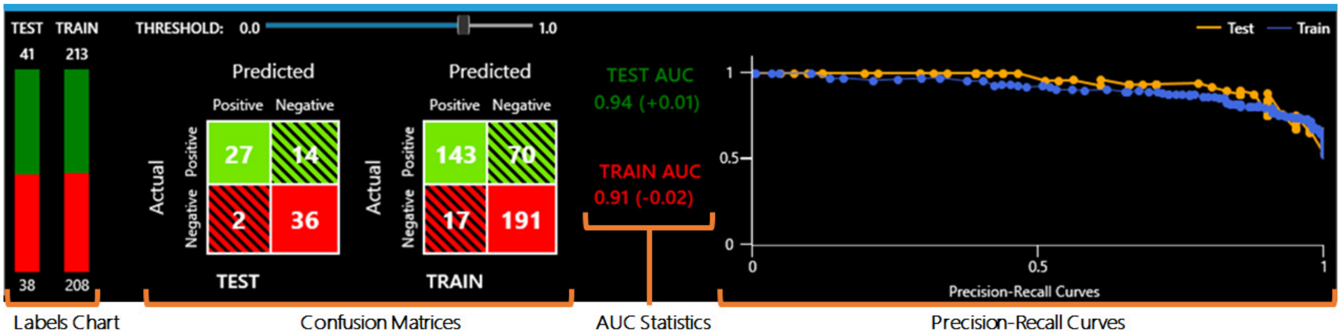

Figure 2. Common performance metrics used during model building in machine learning. Typically each metric is shown over both the *test* and *train* data. Confusion matrices contrast user labels with labels predicted by a model at a specific prediction threshold. Precision-recall curves plot precision versus recall over varying model thresholds. The area under the precision-recall curve (AUC) is often used as a summary of model performance where higher scores indicate better performance. While AUC values can be read from the corresponding curves, they are shown explicitly in the figure for emphasis. This also allows for display of change in AUC performance from one iteration of model building to the next (shown in parentheses). This figure also shows a chart illustrating the current ratio of positive and negative examples, used to encourage the user to provide balanced data.

addition, focusing on individual errors can result in a loss of context, making it difficult for users to prioritize or generalize from their observations. A similar strategy is direct *evaluation*, where users provide *new* data to the model to examine its predictions in real time (e.g., [9,10,21]). However, as with direct examination, this strategy does not facilitate prioritization or generalization of errors. In contrast, ModelTracker presents both summary and example-level performance information directly within its visualization, facilitating error identification and prioritization without the need for context switching.

Dimensionality reduction techniques such as principal components analysis, multidimensional scaling, and clustering can also be used for model debugging [20]. Dimensionality reduction projects high-dimensional data onto fewer dimensions to enable visual inspection of relationships between individual examples, often via two-dimensional scatterplots. While these techniques facilitate deeper model analysis, they can also be complex and difficult to extract insight from [7]. Multidimensional scaling, for example, requires determining a distance function to represent the similarity between data, choosing a scaling mechanism, and then interpreting results. Scatterplot analysis can itself be difficult because meaningful substructures (e.g., clusters) may not be visually salient and reduced dimensions may not correspond to real variables of the data. Projected dimensions will also typically change with each round of model building (e.g., adding new data or features), requiring reinterpretation with each iteration. In contrast, ModelTracker visualizes individual examples along consistent and real dimensions, eliminating the need to reorient and reinterpret visual dimensions after each iteration while fostering expectations about how performance should appear over time.

Many tools have been proposed to aid visual exploration of structures within projections manually (e.g., [12,14]) or semi-automatically (e.g., [7,6,4]). These tools are often complex or immersive, providing functionality to support visual data mining and knowledge discovery. In practice, therefore, cognitively demanding tools are typically only used when a deeper analysis of unexpected performance is necessary rather than to drive each iteration of model building [18]. In contrast, ModelTracker's compact visualization enables it to be always accessible for lightweight performance analysis and debugging throughout the model building process.

Scatterplots are also used for displaying other properties of the data during model building. Scatterplots and scatterplot matrices, for example, are often used to display correlations between features over the current data set [20]. This can help identify feature dependencies as well as potential clusters and outliers. However, these techniques suffer from the same issues as dimensionality reduction techniques including a significant amount of user effort in interpretation and an explicit context switching from building a model to examining properties of the model to interpret errors. In another example, Patel *et al.* [17] used a scatterplot to visualize the results of multiple models simultaneously during the machine learning process. Data points are plotted over fixed axes of entropy and incorrectness. This configuration aids detection of errors potentially caused by either mislabeled data or missing features. While this visualization can also help practitioners prioritize efforts in model debugging, unlike ModelTracker, it requires a large number of models to be trained simultaneously and does not provide an at-a-glance summary of any individual model that would eventually be exported for use in an application.

Finally, several tools have been created to help practitioners build and inspect specific types of models (e.g., decision trees [1], naïve Bayes [2], support vector machines [5], and ensemble models [22]). However, the cost of learning new tools for each type of model or machine learning task can make special purpose tools prohibitive in practice.

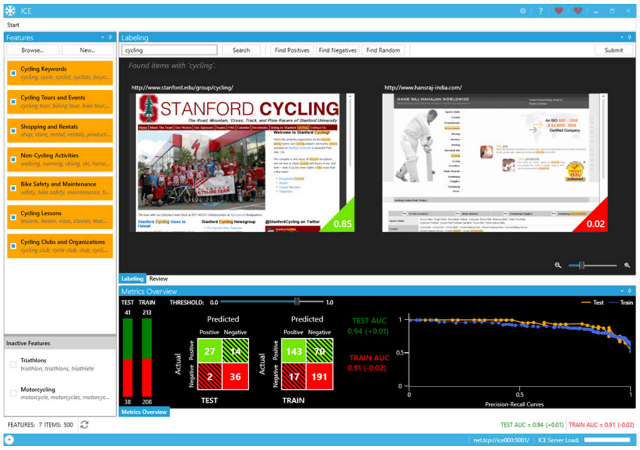

Figure 3. The ICE interface with traditional performance metrics used in machine learning (bottom panel). Users interactively label data in the labeling area (top right) and modify features in the featuring area (left). ICE automatically trains a model as a user supplies labels and features and displays its current performance in the panel at the bottom.

ModelTracker is not algorithm or data type specific, and therefore can provide a consistent means for performance analysis and debugging across a variety of supervised machine learning tasks (as discussed further in Discussion).

MODEL BUILDING REVIEW

ModelTracker was designed for use throughout model building in machine learning. While ModelTracker can be used for many supervised machine learning tasks, in the following sections we focus on its use in binary classification for ease of explanation. In this section, we briefly review the process of building a binary classifier. We then describe relevant aspects the ICE model building environment that we examined ModelTracker in [21].

Building a binary classifier typically begins with collecting and labeling data as either belonging (*positive*) or not-belonging (*negative*) to the target *class*. Next, *features* are defined to characterize relevant aspects of the data in a machine understandable representation. Features should help a classifier discriminate between the positive and negative classes as labeled by the user. Labeled data and features are then fed into a machine learning algorithm to create a model. The labeled data is typically first split into a *train* and *test* set, where the *train* set is used to fit the model and the *test* set is used to estimate model performance on new data. Sometimes a third *validation* set is also set aside for tuning model parameters. Performance is then typically displayed on both the *train* and *test* sets.

ICE is a general purpose machine learning environment for model building, supporting binary and multi-class classification as well as entity extraction. Figure 3 shows the ICE interface. Here, a user is building a binary classifier to classify web-pages as *cycling* or *non-cycling* related pages. Users find (via keyword search) and label within the main *labeling* area (top right). Users define and refine features for the classifier to use via the panel on the left of the display. For example, features that might help a classifier discriminate between *cycling* and *non-cycling* pages may be whether or not the page contains terms pertaining to *types of cycles* (e.g., "cycle" or "bicycle") or an image of a cycle. Given labels and features, ICE automatically splits the labeled data into a *train* and *test* set and then trains a model for review.

Prior to introducing ModelTracker into ICE, users would assess model performance via traditional summary statistics and graphs (Figure 2 and shown in the context of ICE at the bottom of Figure 3). These included confusion matrices, precision-recall curves, AUC statistics, and a label chart showing the balance of positive and negative labels currently provided. These metrics were computed and displayed for both the *train* and *test* sets. If performance analysis indicated problems, individual errors could be inspected by switching from *labeling* to a *review* mode. Errors could then be located via sorting and filtering and then paging through the raw data. After error inspection, users would switch back to *labeling* mode to continue (e.g., with labeling more data or adding or refining features).

MODELTRACKER

ModelTracker was iteratively designed with feedback from real users over one year to enable lightweight performance analysis and direct error examination during model building in machine learning. In this section, we explain how ModelTracker conveys model performance and how it can be used to debug common sources of errors in machine learning. Throughout this section, we also describe design decisions made based on observations and user feedback with early ModelTracker prototypes. Figure 1 shows our current version of ModelTracker (which replaces the panel of metrics in ICE, bottom panel in Figure 3) after a user has provided approximately 500 labeled examples and 7 features, and will be referenced throughout this section.

Conveying Model Performance

Each square box in ModelTracker corresponds to one user labeled example with color indicating the label given (green for *positive* and red for *negative*). Labeled examples can be shown from two sets simultaneously. In Figure 1, *test* examples are shown on top while *train* examples are shown at the bottom of the display (separated by a horizontal line). The boxes are laid out horizontally according to the model's prediction scores (e.g., ranging from 0 to 1), where examples with low scores are to the left and those with high scores are to the right. This means that a well-performing model will have most green boxes to the right and most red boxes to the left. Conversely, errors are shown by red boxes to the right and green boxes to the left. An early ModelTracker prototype displayed items as boxes in sorted order but without distributed them according to model scores. However, user feedback revealed that while ordering examples indicated relative prediction scores (and therefore relative error severity), users also wanted to understand the magnitude of separation between individual examples to better prioritize efforts in debugging errors, leading to our current display.

As labeled examples accumulate, boxes with the same prediction score are binned and stacked away from the horizontal line (e.g., examples in the top section are stacked upwards). Boxes within each bin are sorted such that examples potentially needing attention (e.g., errors) appear closer to the horizontal line. This helps keep problematic examples visible as data accumulates. Feedback from users using ModelTracker for binary classification with datasets of about 1700 examples on average confirm that this is sufficient for providing at-a-glance performance analysis while still alerting them to potential issues needing further inspection. In the Discussion section, we discuss other techniques we employ for scaling to larger number of examples (e.g., sampling for entity extraction in ICE).

ModelTracker subsumes information contained in several traditional summary statistics and graphs while also displaying more detailed, example-level performance. Confusion matrices, for example, display the number of user-model label agreements and disagreements given a specific prediction threshold. While ModelTracker does not display numerical sums, it displays agreement and disagreement via horizontal positioning of boxes according to model scores, where agreement is indicated by green boxes to the right and red boxes to the left and disagreement by red boxes to the right (false positives) and green boxes to the left (false negatives). This has the advantage of making error severity visible, in contrast to numerical sums which treat all errors equally. This also makes ModelTracker threshold independent. A vertical line in the display (Figure 1) depicts a threshold value, but functions only as a visual aid. For example, moving the line to the right emphasizes a reduction of false positives as the threshold increases (because fewer red boxes will appear to the right of the line) while potentially increasing the number of false negatives (because more green boxes may appear to the left). To display the same information using confusion matrices would require a separate matrix for *each* threshold. Moreover, to display this information over two sets (e.g., a *test* and *train* set) would require twice the number of confusion matrices.

ModelTracker also subsumes information conveyed by precision-recall curves (Figure 2). *Precision* is the number of accurately predicted positive examples over the total number of examples predicted as positive by the model, whereas *recall* represents the number of examples accurately predicted as positive by the model out of all of the examples labeled as positive by the user. Precision-recall curves plot precision values versus recall values over various prediction thresholds. Typically, the larger the area under the precision-recall curve (AUC), the better the model. Again, while ModelTracker does not display numerical precision-recall values, it depicts precision and recall via its distribution of boxes, enabling precision and recall to be visible at all thresholds simultaneously. For any given threshold value, *precision* is illustrated by the proportion of green boxes to the right of the threshold line out of all the boxes to the right of the line. Analogously, *recall* is illustrated by the proportion of green boxes that appear to the right of the line out of all the green boxes visible. In contrast, numerical values omit this important information about the distribution of data (i.e., the same numerical value may represent vastly different distributions). ModelTracker's interactive threshold line can also help to emphasize how precision and recall vary over different thresholds. For example, moving the line to the left illustrates an increase in recall (as more green boxes will appear to the right of the line) but at a cost of precision (as more red boxes will also tend to appear to the right).

Monitoring Performance during Model Building

ModelTracker automatically updates as a user iterates in model building, adding boxes as more data is provided and rearranging boxes as prediction scores change (e.g., with new data or features). This results in a spreading and accumulating effect of the green boxes to the right and the red boxes to the left over subsequent iterations, provided the performance of the model is generally improving.

ModelTracker also emphasizes prediction score *changes* on individual examples from one iteration to next to help users better understand the effects of their actions (e.g., feature modifications tend to produce larger effects than the addition of a few more labeled examples) and alert them to potentially unexpected changes. An early ModelTracker prototype illustrated changes via rotating boxes whose prediction changed from one iteration to the next. User feedback, however, suggested a need to show not only that a prediction changed, but also the magnitude of that change. The current version of ModelTracker, therefore, displays the magnitude of score changes via directed arcs from an example's previous score location to its current score location (Figure 1). Because scores on all examples tend to shift slightly from one iteration to the next, arcs are only displayed on examples whose predicted label (computed by comparing the model's prediction score to the current prediction threshold) changed from the previous iteration (e.g., on examples that went from a positive to a negative prediction). Therefore, all arcs displayed will cross the prediction threshold line.

Observations of users interacting with ModelTracker also revealed that users often inspected and then attempted to correct errors (causing an automatic update to ModelTracker's display), but were left uncertain as to whether the error was in fact corrected as a result of their actions. To aid users in monitoring and tracking examples during model building, ModelTracker supports direct annotation of examples via bookmarking or tagging. For example, a user can tag an error and then monitor if the example was in fact corrected in the next update resulting from their action by tracking the tag in the display.

Debugging Common Errors

In this section, we describe how ModelTracker supports interactive inspection and debugging of three common sources of errors in machine learning: mislabeled data, inadequate features to distinguish between concepts, and insufficient data for generalizing from existing examples. Note that these errors may manifest in different ways depending on several factors including which set the error is in (e.g., *train* or *test*), which algorithm is being used, and whether or not the model is currently over-fitted to the data. Where possible, we explain these differences. Note also that aside from correcting mislabels, attempting to correct *test* errors is typically not recommended to prevent overfitting (maintaining a model's ability to estimate performance on new data). In the Discussion section, we suggest techniques for preventing potential overfitting in ModelTracker.

Errors Caused by Mislabeled Data

Mislabeled data (often referred to as label or class noise) frequently occurs during data collection in machine learning [15] and can be resolved by locating and correcting the incorrect labels. Egregious errors in the *train* set in ModelTracker (red boxes to the far right or green boxes to the far left) tend to be candidates for mislabeled data. This is provided the model is not over-fitted to the data (in which case mislabels may appear at any location). If a user suspects a mislabel, they can click on the corresponding box to bring up the raw data in a separate viewing area (e.g., as a popup over the labeling area in ICE). Mislabels can then be corrected and resubmitted to the system.

Errors Caused by Feature Deficiencies

Another common source of error in machine learning is caused by failing to supply a model with adequate features to accurately discriminate between target concepts. For example, a model intended to recognize *cycling* from *non-cycling* pages may confuse a page about *motorcycling* as *cycling* if only given features representing terms such as "ride" or "bike" as these may appear in both pages. Correcting this type of error often requires supplying the model with additional features (e.g., the addition of features with the terms "motorcycle" or "motorbike").

Potential feature related confusion can manifest as examples that the model views as similar given the current feature space but for which the user has given different labels. This is because the ability to generalize often implies that similar inputs (examples represented by features) have similar outputs (predicted labels). Therefore, if two examples are seen as similar by the model but different by the user (i.e., are neighbors in feature space but have different labels), this is an indication that the model is currently feature blind to some difference that the user can see. Locating such candidates for feature related errors in ModelTracker requires first hovering over errors to reveal neighboring examples (shown via connector lines between boxes, Figure 4), and then identifying neighbors with different labels (i.e., boxes with different colors connected with lines). Neighbors can be computed in many ways. We use the distance function:

$$||\boldsymbol{x} - \boldsymbol{y}||^2 = \boldsymbol{x}^T\boldsymbol{x} + \boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{x}^T\boldsymbol{y}$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are feature vectors representing two labeled examples. These are the same vectors supplied to the learning algorithm in ICE and therefore reflect the same information the classifier receives. For each example, we select the five nearest neighbors (i.e., with shortest distance to the target example) as the most similar and connect them with lines on hover. Clicking on boxes also brings up both the clicked example as well as the neighbors for further examination and feature ideation.

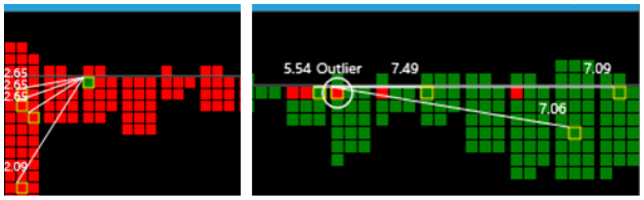

Figure 4. Connector lines showing examples considered as similar in feature space to the example hovered over. On the right, an outlier in feature space is depicted with a circle around the corresponding example.

ModelTracker can also display how features are correlated with the target class as is often examined via scatterplot analysis. For example, a positive correlation is suggested if values of a particular feature are higher for the *positive* class than for the *negative* class. Correlations can reveal feature deficiencies if unexpected trends are observed (e.g., if a feature was expected to be positively correlated but has high values in both the positive and negative classes). Feature correlation is displayed on demand in ModelTracker (triggered by hovering over a feature in ICEa) and is shown via box highlighting (Figure 5) where brighter and dimmer boxes indicate larger and smaller values of the corresponding feature, respectively.

Errors Caused by Insufficient Data

Insufficient data can also result in errors. These errors are most common in the *test* or *validation* set for regions of the input space (defined by the current set of features) that are insufficiently represented in the *train* set. Depending on the details of the modeling algorithm, however, they can also

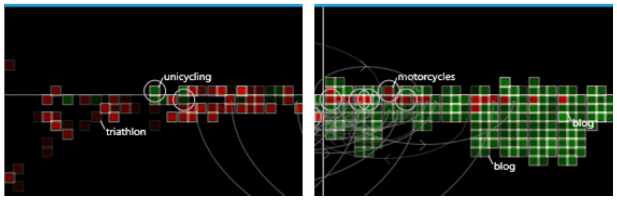

Figure 5. Correlations between features and the target class are displayed on demand via highlighting of boxes based on feature values. Brightness represents the value of the feature in the corresponding example. More bright green boxes on the right and dimmer red boxes towards the left show a feature that is highly correlated with the positive class.

occur in the *train* set when there is not yet enough examples to overcome some bias in the model (e.g., in highly-regularized linear models). Insufficient data errors often manifest as *outliers* – isolated examples distant from others in the current feature space. These errors can be corrected by providing additional data to the model.

In early ModelTracker prototypes, outliers could be identified by locating examples with distant neighbors (shown via lines between boxes). However, observations indicated that users often missed potential outliers because distance is relative to the overall distribution of examples in feature space, which is not readily apparent. To more directly draw attention to potential outliers, the current version of ModelTracker indicates outliers via circles around corresponding boxes (Figure 4).

Many techniques exist for computing outliers in statistics and machine learning. We compute outliers as examples whose nearest neighbors in feature space are greater than two standard deviations from the mean distance between all pairs of neighboring examples. By clicking on outliers, users can again inspect the corresponding example and then find additional examples similar to the outlier to supply to the model to try and correct the error.

USAGE BEHAVIOR

In this section, we report on the usage behavior of machine learning practitioners building real models (for research and product deployments) with ICE along with ModelTracker over six months to demonstrate how ModelTracker's visualization and support for direct access to data encourages frequent and regular data inspection throughout model building. We limit our analysis to binary classifiers (via logistic regression) built by people outside of our team and containing at least 400 labels. This includes 40 different models built by seven practitioners (ranging from novices to experts) over text and web page data. Example concepts these models were built to detect include *restaurant* web pages, *happy* or *angry* sentiment in text, and text containing *opinions*. These models were interactively built over an average of 5.1 hours (SD=3.8), with 1714.5 (SD=1260.2) labels and 12.1 (SD=7.7) features, and achieving final *test* AUC performances of 0.91 (SD=0.17). Note that some features such as ModelTracker's support for item tagging and identification of feature deficiencies and outliers were introduced during this period. Therefore, we only report usage statistics on features present throughout this period.

Out of the 40 sessions we examined, 35 (87.5%) involved at least one interaction with ModelTracker (i.e., one box click). If people interacted with ModelTracker, they clicked on an average of 63.3 examples (4.1% of their data) during their sessions. Figure 6 shows the average distribution of ModelTracker interactions over time, indicating that people interacted with ModelTracker throughout model building, increasing interactions as they progressed.

Interestingly, of the examples clicked in ModelTracker, only 41.6% (SD=23.6%) were in error at the time they were clicked, on average. However, because errors are computed given a specific prediction threshold, this does not necessarily capture whether people were clicking on examples that were technically correct but were still potentially problematic (e.g., negative examples with relatively high scores but still less than the threshold). Figure 7 shows the average percentage of examples clicked per score bin, indicating that users tended to click more on examples at the far ends of the display, but also still clicked on examples in the uncertain middle range.

Post-deployment interviews revealed a common usage strategy that naturally arose from unguided user interaction with ModelTracker. Practitioners explained that they often ignored ModelTracker until some initial data was collected (e.g., one participant said *"about 30 or so [initial] labels"*). Then they began inspecting errors at the far ends of the visualization, often with false positives at the far right, and then working their way inwards. They explained that these errors seemed to have "easier" and more "obvious" solutions (e.g., "*I always start with the biggest outliers [at the far ends] in part because it's a sign that the label was wrong"*), whereas errors in the middle were deemed as harder cases. During error inspection, practitioners reported trying to determine causes and actions to take including "adding features," "changing labels," or "adding labels."

Practitioners also commented on ModelTracker's visual aids and diagnostic capabilities. For example, one practitioner reported initial confusion about the connector lines showing example neighbors in feature space, suggesting the need for more instructional support. Practitioners did report that the directed arcs helped to

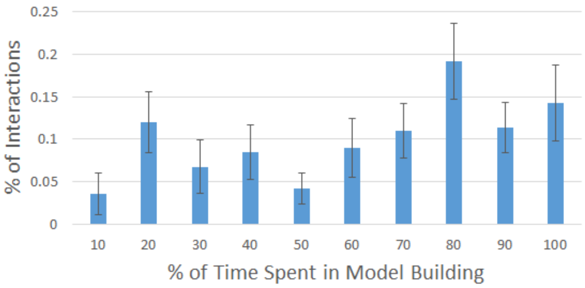

Figure 6. The average distribution of ModelTracker interactions throughout the model building process.

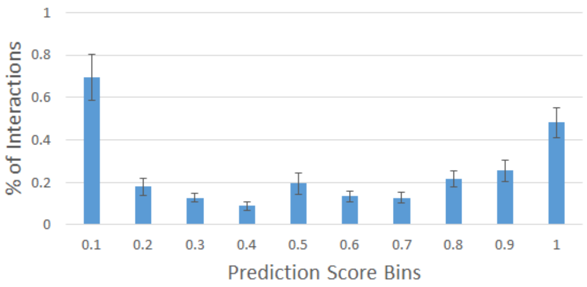

Figure 7. The average percentage of examples clicked in the ModelTracker per score bin.

verify whether an action taken resolved a suspected issue. One commented that it *"was rewarding or not [when the action did not help]."* Another commented that the arcs *"helped when it matched my expectations [about what I thought was going to happen]."* Practitioners also remarked that the arcs were less useful when there were too many of them and that the direction was sometimes hard to see.

Overall, feedback from our practitioners were positive towards ModelTracker. One practitioner commented that it is the *"most important instrument in ICE [for model building]"* and *"you can't go back [to operating without it]."* In comparing ModelTracker to summary statistics, one practitioner commented that "*[ModelTracker] gives you the complete picture. Looking at summary statistics is very good for comparing models, but I can't see what I need to do. I need to do other mini-experiments to figure that out."* However, some practitioners still desired summary statistics. One commented that *"[ModelTracker is] useful for someone new to machine learning, but I still wanted to get AUC, for people who don't use GUIs."* Interestingly, one practitioner who commented on their developed reliance on ModelTracker also requested summary statistics to help gauge improvement over time, saying "*What [the directed arcs are] getting at is a good thing, but I still want a number for whether it got better or worse overall."*

EVALUATION OF DEBUGGING CAPABILITIES

We performed a controlled experiment to examine ModelTracker's support for debugging three common errors in machine learning discussed previously (i.e., mislabeled data, feature deficiencies, and insufficient data) compared to the traditional method of using separate tools to debug. Specifically, we compared ModelTracker's interface with a *Traditional* interface, where traditional summary metrics are provided for performance analysis (Figure 2) and individual errors can then be inspected via a *review* mode. Mislabels are located via filtering by errors or error categories (false positives and negatives) and then paging through and viewing raw data. Feature deficiencies are discovered by locating errors and then clicking a button to reveal similar examples with opposite labels in the current feature space. Finally, outliers are located by filtering by outliers. Both interfaces were used within the context of ICE. The study was run using a within-subjects design, counterbalancing interface order.

Tasks

To examine debugging capabilities, we created two tasks requiring participants to fix or improve existing classifiers. These tasks occur in real scenarios such as during model maintenance (e.g., when a deployed model must be updated in light of new data) and when labels are outsourced and initial features are insufficient to model the target concept.

To create our tasks, we first built binary web-page classifiers (via logistic regression) targeting concepts related to common activities (e.g., *cooking*, *travel*). To build each classifier, we sampled data (using uncertainty-based active learning) from the Open Directory Project database (http://www.dmoz.org), an open directory of human-categorized web pages. We used the concept descriptions given in this database to guide our labeling and featuring. From these classifiers, we selected two that achieved a comparable level of performance (measured via their *test* AUC values) with 400 labels and 11 features. The classifiers we selected targeted *gardening* and *travel* related web-pages and achieved *test* AUCs of 0.95 and 0.96, respectively. We then "broke" each classifier to generate errors by first removing a random subset of 130 examples (~33% of the initial data), randomly removing 5 of the 11 features, and then randomly flipping 27 (10%) of the remaining labels. These broken *gardening* and *travel* classifiers served as the starting points for our study tasks. We also created a broken *cycling* classifier in the same way to use during the demos and practice with each condition.

Participants and Procedure

We recruited 14 participants (10 male) from a large software company including developers, test engineers, and program managers ranging in age from 23 to 53 years old. Participants self-reported as novice or proficient in machine learning, had previously built at least one model, and were familiar with precision and recall. Participants were run in groups of at most four, each working on an identical PC running Windows 8, with a 24" 1920x1200 monitor. All relevant interface actions were time-stamped and logged.

Each session began with a review of the general model building process and a hands-on tutorial of ICE (with both the *Traditional* and *ModelTracker* panels hidden). Before each condition, the experimenter gave a tutorial on the corresponding interface and then allowed participants to practice debugging the broken *cycling* classifier with it. Prior to starting each task, the experimenter walked through the target concept description. Participants were then given 20 minutes to debug and improve their broken classifiers as much as possible. Questionnaires were distributed after each condition and at the end of the study. Each session took three hours. Participants were given $30 worth of dining coupons and a $50 prize was awarded to the person with the largest classifier improvement.

Results

We analyzed classifier performance and logged data from our study using paired-samples *t*-tests. We analyzed performance in two ways. First, we computed the AUC improvement from the beginning to end of each task on a holdout dataset comprised of the 130 examples we removed as part of breaking each classifier. Second, we computed the number of corrected mislabels out of the 27 we introduced as part of breaking each classifier. Table 1 shows all means and standard deviations of performance and the number of labels and features included in participants' final classifiers. We found no significant

	Traditional	ModelTracker
AUC Improvement	.008 (.042)	.026 (.027)
Mislabels Corrected	8.57 (6.02)	9.86 (4.89)
Final Num Labels	327.5 (28.1)	320.9 (25.5)
Final Num Features	8.64 (1.86)	8.57 (1.34)

Table 1. Mean (SD) of performance and usage statistics.

difference in participant ability to debug or improve model performance between the two interfaces.

We analyzed our post-condition Likert-scale questions using Wilcoxon signed-rank tests. Figure 8 shows preference counts and Likert-scale question medians. 10 out of 14 participants (71.4%) preferred *ModelTracker* over the *Traditional* interface. Participants favored *ModelTracker* in terms of their perceived ease of identifying and debugging both feature deficiencies (*Z*=-2.49, *p*=.013) and insufficient data (outlier) errors (*Z*=-1.99, *p*=.046). No significant differences were found in participants' stated ability to understand their classifiers' performance or their perceived ease of identifying and debugging mislabeled data.

DISCUSSION AND FUTURE WORK

Our examination of real ModelTracker usage and our controlled experiment revealed that practitioners relied on ModelTracker throughout model building and preferred it over traditional tools without a loss in performance or debugging ability. Participants commented that ModelTracker was simpler and more intuitive than the many summary statistics and graphs required to display equivalent performance information in the traditional interface. One participant said *"I thought this UI was more intuitive… I think it is great to take such a complex concept and make it accessible."* Another said *"The single visualization captures all of the segmented version and more."* Participants also commented on the reduced overhead ModelTracker provided in terms of direct error identification and reduced mode switching: *"I liked being able to focus on the data without having to go between screen[s]," "Finding issues within the set was easy,"* and *"Very actionable information."* Of the participants who preferred the traditional interface, some commented on missing the more familiar statistics: *"[Traditional] is better if you know a bit about how classifiers work."* Some also requested a hybrid view. One participant said *"I would like to see both integrated."* Another recommended to *"have [the ModelTracker] UI default, but the ability to bring up the confusion matrix and PR curves as optional UI."*

In the ModelTracker visualization, we displayed both the *train* and *test* data to users as the ICE system provides access to both sets during model building. This has the potential to encourage overfitting by tempting users to over-engineer models to fix *test* errors. However, since overfitting only happens when users edit features or adjust model parameters (not when fixing labels), one solution may be to disable inspection of *test* examples, thereby only using the *test* area to monitor generalization performance. Another solution may be to remove the *test* altogether (or replace it with a *validation* set), and rely on summary statistics to show generalization performance in a hybrid system. Further investigation is necessary to evaluate these options for reducing the potential to overfit.

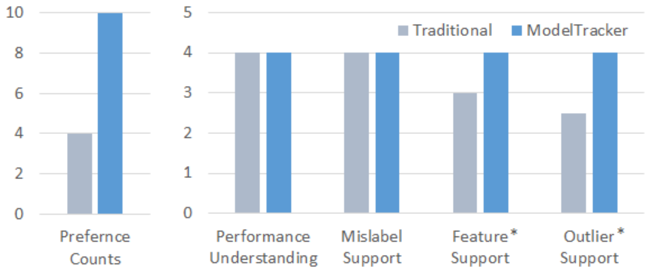

Figure 8. Overall preference counts and Likert medians. Metrics with *'s indicate significant effects were observed.

We concentrated on binary classification via logistic regression in this paper for ease of explanation. However, ModelTracker can visualize other classifiers that output probabilities (e.g., any calibrated classifier) and other supervised machine learning models. For example, within ICE [21] we also use ModelTracker for entity-extraction which uses conditional random field models to identify sequences of tokens belonging to a hierarchical set of entities (e.g., an *address* containing *street* and *city* entities). We use ModelTracker to visualize performance with respect to each entity by creating a green box for each entity (or a red box for each non-entity) labeled token or *sequence* of tokens. Prediction scores represent the relative confidence that the model assigns to these hypotheses, computed by taking the difference in probability of a parse consistent with the label and the next most likely inconsistent parse. In entity extraction, the number of boxes can grow quickly, particularly when there is one box per token. When this happens, we sample down the boxes in each score range while maintaining the same fraction of positive and negative labels.

Entity-extraction problems also have similar characteristics to multi-class classification problems, in that both can involve multiple prediction categories (hierarchical entities and multiple classes, respectively). In both cases, we can view the one-vs-remaining performance for each category in ModelTracker. An opportunity remains to integrate between-class confusability information into a single visualization to better support light-weight performance analysis and debugging in multi-class scenarios.

ModelTracker may have some benefits as a stand-alone static visualization to replace traditional performance statistics and graphs. Clearly, however, many of ModelTracker's benefits increase when it is integrated into an environment supporting an iterative process of model building and error analysis. As discussed earlier, when ModelTracker is linked back to the individual examples, mislabeled data can be quickly discovered, examples with few neighbors can be brought forward and examined, and situations where new features need to be identified can be called out. Even closer integration allows overall model performance as well as performance on individual examples to be monitored as the model is iteratively retrained. Further investigations are therefore necessary to understand the benefits of ModelTracker as a stand-alone tool.

CONCLUSION

We introduced ModelTracker, a generally applicable interactive visualization for performance analysis and debugging in machine learning. Our examination of ModelTracker usage by machine learning practitioners building real models with ModelTracker for research and product deployments over a six month period demonstrates that ModelTracker is used regularly throughout model building. Our controlled experiment further shows that ModelTracker is preferred over traditional tools for performance analysis and debugging for its simplicity and intuitiveness without a loss in model performance. This suggests that ModelTracker can replace current tools for performance analysis and debugging, encouraging a more informed approach to model building in machine learning.

REFERENCES

1. Ankerst, M., Elsen, C., Ester, M., and Kriegal, H. Visual Classification: An Interactive Approach to Decision Tree Construction. *Proc. KDD 1999,* ACM Press (1999), 392-396.
2. Becker, B., Kohavi, R., and Sommerfield, D. Visualizing the Simple Bayesian Classifier. *Information Visualization in Data Mining and Knowledge Discovery.* Fayyad, U., Grinstein, G.G., and Wierse, A. (eds). Morgan Kaufmann Publishers, 2001, 237-249.
3. Bird, S., Klein, E., and Loper, E. *Natural Language Processing with Python*. O'Reilly Media, 2009.
4. Broekens, J., Cocx, T., and Kosters, W. Object-Centered Interactive Multi-Dimensional Scaling: Ask the Expert. *Proc. BNAIC 2006*, 59-66.
5. Caragea, D., Cook, D., and Honavar, V. Gaining Insights into Support Vector Machine Pattern Classifiers Using Projection-Based Tour Methods. *Proc. KDD 2001*, ACM Press (2001), 251-256.
6. Chan, Y., Correa, C., and Ma, K-L. Flow-based Scatterplots for Sensitivity Analysis. *Proc. VAST 2010*, IEEE (2010), 43-50.
7. Choo, J., Hanseung, L., Liu, Z., Stasko, J., and Park, H. An Interactive Visual Testbed System for Dimension Reduction and Clustering of Large-Scale High-Dimensional Data. *Proc. SPIE Electronic Imaging 2013,* 865402–865402–15.
8. Domingos, P. A Few Useful Things to Know about Machine Learning. *CACM 55,* 10 (2012), 78-87.
9. Fails, J.A. and Olsen, D.R. Interactive Machine Learning. *Proc. IUI 2003,* ACM Press (2003), 39-45.
10. Fiebrink, R., Cook, P.R., and Trueman, D. Human Model Evaluation in Interactive Supervised Learning. *Proc. CHI 2011*, ACM Press (2011), 147-156.
11. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I.H. The WEKA Data Mining Software: An Update. *SIGKDD Explorations 11*, 1 (2009).
12. Hao, M.C., Dayal, U., Sharma, R.K., Keim, D.A., and Janetzko, H. Variable Binned Scatter Plots. *Information Visualization 9*, 3 (2010), 194-203.
13. MATLAB 9.0 and Statistics Toolbox Release 2014a, The MathWorks, Inc., Natick, Massachusetts, United States, http://www.mathworks.com/products/statistics, 2014.
14. Mayorga, A. and Gleicher, M. Scatterplots: Overcoming Overdraw in Scatter Plots. *IEEE TVCG 19*, 9 (2013), 1526-1538.
15. Nettleton, D. F., Orriols-Puig, A., and Fornells, A. A Study of the Effect of Different Types of Noise on the Precision of Supervised Learning Techniques. *AI Review 33*, 4 (2010), 275-306.
16. Patel, K., Bancroft, N., Drucker, S.M., Fogarty, J., Ko, A., and Landay, J.A. Gestalt: Integrated Support for Implementation and Analysis in Machine Learning Processes. *Proc. UIST 2010*, ACM Press (2010), 37-46.
17. Patel, K., Drucker, S.M., Fogarty, J., Kapoor, A., and Tan, D.S. Using Multiple Models to Understand Data *Proc. IJCAI 2011*, AAAI Press (2011), 1723-1728.
18. Patel, K., Fogarty, J., Landay, J.A., and Harrison, B. Examining Difficulties Software Developers Encounter in the Adoption of Statistical Machine Learning. *Proc. AAAI 2008*, AAAI Press (2008), 1563-1566.
19. R Core Team, "R: A Language and Environment for Statistical Computing," *R Foundation for Statistical Computing*, http://www.R-project.org, 2013.
20. Rossi, F. Visual Data Mining and Machine Learning *Proc. ESANN 2006*, 251-264.
21. Simard, P., Chickering, D., Lakshmiratan, A., Charles, D., Bottou, L., Suarez, C.G.J., Grangier, D., Amershi, S., Verwey, J., and Suh, J. ICE: Enabling Non-Experts to Build Models Interactively for Large-Scale Lopsided Problems. 2014, arXiv:1409.4814.
22. Talbot, J., Lee, B., Kapoor, A., and Tan, D. EnsembleMatrix: Interactive Visualization to Support Machine Learning with Multiple Classifiers. *Proc. CHI 2009*, ACM Press (2009), 1283-1292.