Learning by Combining Observations and User Edits

Vittorio Castelli
IBM T.J. Watson Research Center
1101 Kitchawan Road – Route 134
Yorktown Heights, NY 10598

Lawrence Bergman
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532

Daniel Oblinger*
DARPA / IPTO
3701 Fairfax Dr
Arlington, VA 22203

Abstract

We introduce a new collaborative machine learning paradigm in which the user directs a learning algorithm by manually editing the automatically induced model. We identify a generic architecture that supports seamless interweaving of automated learning from training samples and manual edits of the model, and we discuss the main difficulties that the framework addresses. We describe Augmentation-Based Learning (ABL), the first learning algorithm that supports interweaving of edits and learning from training samples. We use examples based on ABL to outline selected advantages of the approach—dealing with bad data by manually removing their effects from the model, and learning a model with fewer training samples.

Introduction

Supervised learning is usually an entirely automatic process. Some of its extensions, however, are collaborative processes with a user in the loop. The best known such extensions are active learning and relevance feedback. In active learning, typically used in classical machine learning settings, the learning algorithm is allowed to ask the user for the label corresponding to a specific feature vector. In relevance feedback, common in information retrieval, the user labels the results produced by a search engine as relevant or irrelevant. The search engine uses this feedback to update the model.

To our knowledge, all extensions to supervised learning that include a user in the loop provide the user with *indirect* control of the model produced by the learning algorithm. The user can only interact with the learner by supplying new labeled samples, and the learner is responsible for appropriately updating the model. Typically, both the update process and the model produced are opaque to the user. When the model is human readable, manual editing may be supported as a postprocessing step, but with no further learning.

We propose a new approach to user-in-the-loop supervised learning that provides the user with *direct* control of the model via editing operations *as part of the learning process* (rather than as a post-processing step). In this approach, edits and automated learning can be seamlessly interwoven. Unlike the case of indirect control, direct editing requires human readability of the entire model (e.g., decision trees, rules, and linear models can be naturally represented in a human-readable form) or of selected aspects of the model (e.g., the features selected by a feature-selection algorithm). A related approach (Jensen 1992) partitions induction into tasks better suited for a human and tasks better suited for a learning algorithm; the human and the automatic algorithm then have distinct complementary roles. In contrast, we allow both user and learning algorithm to modify the model.

We have identified two main problems in supporting the seamless interweaving of editing operations and automated learning, which we call the *precedence problem* and the *consistency problem*. The precedence problem arises when the learning algorithm is allowed to "undo" the user edits using data observed before they were performed. The consistency problem arises when the data observed before the edit are inconsistent with the edited model, and the learning algorithm learns parameters of the model incorrectly as a consequence.

We propose an enhanced form of learning that addresses both these problems. We illustrate this new approach with *Augmentation-Based Learning* (ABL) (Oblinger, Castelli, & Bergman 2006), the first example of an incremental learning algorithm that allows seamless interweaving of automated learning and user editing. ABL was devised as an algorithm for programming-by-demonstration (Cypher 1993; Lieberman 2001), a class of techniques that automatically construct a program using the interaction between a user and a computer application as input. While ABL solves a specific problem, it relies on general principles to address the consistency and the precedence problems.

ABL takes as input sequences of *snapshot-action pairs*, in which the snapshot describes the content of the application GUI and is obtained just before the user performs the action. As a user demonstrates a task, ABL incrementally infers a model in real time and produces a human-readable description. ABL learns from multiple demonstrations: if a user demonstrates a path through the task that was not previously captured by the existing model, ABL produces an explanation of the deviation, and incorporates it into the model.

Unlike other incremental learning algorithms, ABL allows the user to edit the model representation at any point in time. The user can delete, move, group, or copy steps,

*This work was done while the author was with the IBM T.J. Watson Research Center.

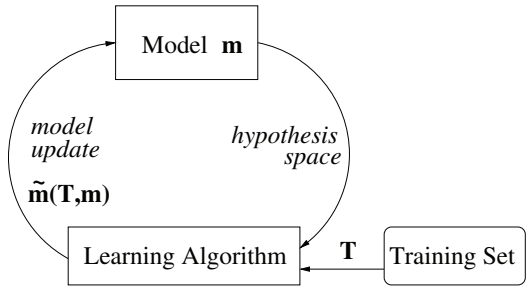

Figure 1: Iterative or incremental learning architecture.

and ABL can continue learning from demonstrations after the edits. ABL solves the precedence problem by restricting the set of operations used to incrementally update the model, and the consistency problem by selectively discarding data observed before an edit that is inconsistent with that edit.

In the next section we describe a learning paradigm that supports the interweaving of learning from samples and manual editing, and discuss the precedence problem and the consistency problem. We then provide examples of the behavior of ABL, the first algorithm that supports this paradigm. Finally, we give a high-level description of the ABL algorithm and discuss how it solves the precedence and consistency problems.

Combining Learning with Manual Editing

We consider a broad class of learning algorithms (learners), schematically described in Figure 1, that includes: 1) algorithms where training consists of multiple iterations using a fixed training set; 2) incremental learning algorithms, where a new sample is added at each iteration; and 3) relevance feedback algorithms, where new positive and negative examples are added at each iteration. During each iteration, the learner replaces the existing model $\mathbf{m}$ with a new model $\widetilde{\mathbf{m}}(\mathbf{t}, \mathbf{m})$ selected using the training set $\mathbf{t}$. If $\mathbf{T}$ denotes the collection of all possible training sets, then the *hypothesis space* of the learner given the current model is the set of all $\widetilde{\mathbf{m}}(\mathbf{t}, \mathbf{m})$'s selected by the learner as $\mathbf{t}$ varies over $\mathbf{T}$.

Figure 2 illustrates how to extend these learning algorithms to support interweaving of iterative learning from data and manual edits of the model. We assume that a human-readable representation of the model is available to the user (the *author*) and that the model is editable. For example, the model produced by the k-means algorithm could be shown as a scatterplot where the centroids are clearly marked, and editing could be accomplished by changing the coordinates of the centroids. An ABL model is displayed as a *script* editable via a simplified text editor.

Supporting the combination of manual edits and learning is not straightforward; in particular, two new problems arise: the *precedence* and the *consistency* problems.

The Precedence Problem When learning and manual edits are interwoven, a learning algorithm might automatically reverse the effects of the edits. This is acceptable if the data used to reverse an edit is observed after the edit operation. However, if the data predates the edit, the reversal is undesirable—this is the precedence problem.

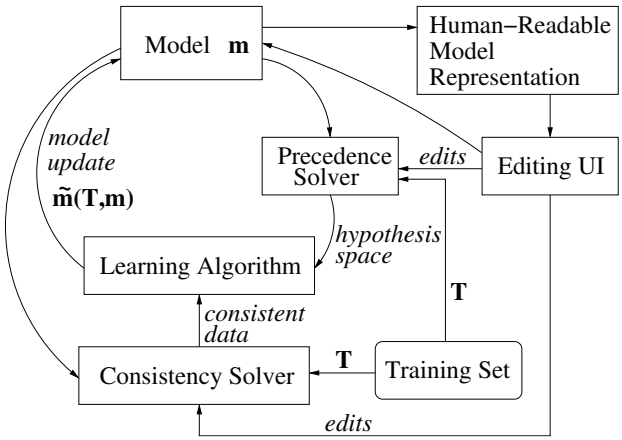

Figure 2: Architecture for supporting interwoven editing and learning from data.

Consider, for example, a k-means extension that allows the author to inspect the models produced during specific iterations. Assume that the author edits the model by setting all the coordinates of the centroid closest to the origin to zero. During the subsequent iteration, the classic k-means algorithm would update all the centroids, including the one edited by the author, thereby undoing the edit.

A solution to the precedence problem consists of appropriately restricting the hypothesis space of the learner to exclude models that 1) could be selected using data observed before the user edit, and 2) counteract that edit. In the k-means example, the hypothesis space would be restricted to the set of k centroids in which k-1 are arbitrary and one is fixed at the origin of the feature space.

Figure 2 explicitly contains a component, the *Precedence Solver* that restricts the hypothesis space using the current model, the author edits, and the training data. In practice, the precedence solver need not be a separate component and this functionality could be built into the learning algorithm itself (as, for example, in ABL).

Recall that the k-means learning algorithm alternates partitioning the training set using the centroids, and recomputing the centroids using the partitioned training set. In the example, the precedence solver for the k-means algorithm would mark the centroid edited by the author as fixed so that it would not be recomputed.

The Consistency Problem When an author edits a component of the model, it is possible that part of the data is no longer appropriate to train that component; if a learner used these data, the result would be an incorrect model—this is the consistency problem.

Consider, for example, an author who edits a decision tree after the splitting is complete and before pruning starts. Assume that splitting and pruning are done with the same data set (e.g., using Quinlan's error-based pruning or pessimistic-error pruning strategies), and that splits are binary. Editing consists of the split variable or the split value of any node.

Each node of the tree has an associated subset τ of the training set, which is partitioned according to the split variable and split value into a right set τ_R and a left set τ_L. These sets are used during training to construct the right and left branch of the subtree rooted at the node, and during pruning to prune them. If the author edits this node, τ_R and τ_L need not be consistent with the new split and using them to prune might yield an overpruned or underpruned model.

To address the consistency problem, Figure 2 contains the *Consistency Solver*, which passes only data consistent with the model and the author edits to the learner. Thus, the Consistency Solver acts as a filter that prevents the use of inconsistent data during induction.

In the decision tree example, the consistency solver would recursively split the training subset τ associated with the edited node using the new split variable or split value; it would produce $\tau_R^{'}$ and $\tau_L^{'}$ for the children of the node and further partition them at each descendant node, stopping at the leaves. Pruning is then performed with these data.

Examples

We base our examples on ABL, the heart of an Eclipse-based system called DocWizards (Bergman *et al.* 2005; Prabaker, Bergman, & Castelli 2006), which captures procedure models for use in automating or documenting tasks. In this section, we show how manual editing operations can be used to remove the effects of "bad" data from the model, and to help the learning algorithm induce a model with fewer examples.

Removing the effects of bad data In this example, the author records a procedure that adds the "add-javadoc" tag to all the projects in an Eclipse workspace, where the projects are represented as tree items in the "Package Explorer" tree.

In the first demonstration, the author starts recording the procedure without ensuring that the initial state of the application is the desired one. Consequently, she sees that the "add-javadoc" tag is already associated with the first project's properties. The author removes the tag from the properties to correctly initialize the state, and then shows how to add it. The resulting script is:

```
1 Select toolbar item "Java"
2 Select tree item "project1"
3 Select menu item "Project-Properties"
4 Select tree item "Java Task Tags"
5 Select table cell "add-javadoc" // spurious
6 Click "Remove"                  // spurious
7 Click "New..."
8 Enter text "add-javadoc" into "Tag:"
9 Click "OK"
```

where step numbering and comments (introduced by "//") have been manually added to improve readability. The procedure now contains two spurious steps (5 and 6) that the author does not want as part of the procedure. The user edits the script and deletes these steps, then continues to demonstrate the task, showing the same operations on the second project, but without the two spurious steps. ABL correctly identifies a loop over the projects, but does not use the demonstrations to reintroduce the removed steps. The resulting script is the following:

```
1 Select toolbar item "Java"
2 Foreach Tree item in 'Package Explorer'
3     Select tree item Tree item
4     Select menu item "Project-Properties"
5     Select tree item "Java Task Tags"
6     Click "New..."
7     Enter text "add-javadoc" into "Tag:"
8     Click "OK"
```

This demonstrates the ability of ABL to correctly perform inference after steps have been removed from the model.

Learning a model with fewer examples The following example describes a case in which the author manually edits the script both to reduce the number of demonstrations needed to build a complete procedure model, and to help the learning algorithm produce an efficient (i.e., small) model.

The task used in this example ensures that two projects (test1 and test2) are in the workspace before proceeding with further operations. The first demonstration is performed with neither project in the workspace, and the second with both present. The resulting script is:

```
 1 If Tree item "test1" does not exist, then
 2     Right-click on "Package Explorer"
 3     Select popup menu item "Import..."
 4     Select "Existing Project into Workspace"
 5     Open "test1"
 6     Click "Finish"
 7     Right-click on "Package Explorer"
 8     Select popup menu item "Import..."
 9     Select "Existing Project into Workspace"
10     Open "test2"
11     Click "Finish"
12 Select tree item "test1"
```

Step 12 is the first step of the rest of the task, which is omitted for sake of brevity. The first demonstration contains all the actions in the script (Steps 2-12); the second demonstration starts at Step 12. ABL learns a decision point (Step 1) and the condition under which Steps 2 to 11 are executed.

The third demonstration is performed with test2 in the workspace, but not test1. The resulting script, which combines the three demonstrations, is:

```
 1 If Tree item "test1" does not exist, then
 2     Right-click on "Package Explorer"
 3     Select popup menu item "Import..."
 4     Select "Existing Project into Workspace"
 5     Open "test1"
 6     Click "Finish"
 7     If Tree item "test2" does not exist, then
 8         Right-click on "Package Explorer"
 9         Select popup menu item "Import..."
10         Select "Existing Project into Workspace"
11         Open "test2"
12         Click "Finish"
13 Select tree item "test1"
```

This partial model is not incorrect, and with a sufficient number of additional demonstration ABL would correctly

infer a model that covers all possible initial states. However, the author decides to manually edit the script to move the second conditional outside the "then" part of the first, and produces the following model

```
 1 If Tree item "test1" does not exist, then
 2     Right-click on "Package Explorer"
 3     Select popup menu item "Import..."
 4     Select "Existing Project into Workspace"
 5     Open "test1"
 6     Click "Finish"
 7 If Tree item "test2" does not exist, then
 8     Right-click on "Package Explorer"
 9     Select popup menu item "Import..."
10     Select "Existing Project into Workspace"
11     Open "test2"
12     Click "Finish"
13 Select tree item "test1"
```

The example shows the power of combining learning with editing. The final model was produced with a small number of examples and minimal user intervention—three demonstrations and one drag-and-drop operation. Note that after the edit, additional learning is supported.

Augmentation-Based Learning

ABL is invoked incrementally whenever the author performs an action. If the existing model explains the action, no modification is required. Otherwise, ABL computes a set of possible *augmentations* of the model to incorporate the new observation. An augmentation is any modification of an existing model that preserves all steps and all sequentiality relations between steps. Figure 3 contains two examples of augmentations: the introduction of a new path (a loop, in this case) and the introduction of a new step. The original sequencing is preserved by both transformations; in particular, the model resulting from the second augmentation still contains a (conditional) direct path from Step1 to Step2. Figure 4 shows a change that does not preserve the direct path between Step1 and Step2 and thus is not an augmentation.

Assume that a user can edit a script by deleting, moving, copying, grouping, and ungrouping steps, all of which alter the structure of the script. The precedence problem arises if a learning algorithm undoes the structure produced by edits using data observed before these editing operations. An algorithm that never modifies the structure produced by an edit using data observed before that edit solves the precedence problem. By using only augmentations, which always preserve the structure of the model being modified, ABL satisfies this requirement. When the author edits a procedure

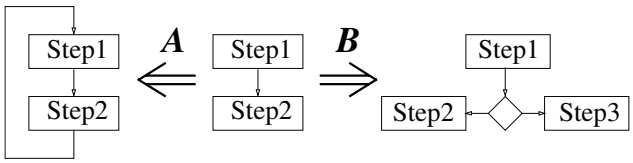

Figure 3: Augmentations. **A**, introducing a new path; **B**, introducing a new step.

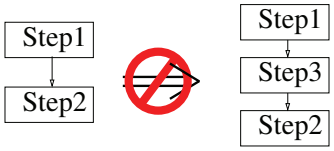

Figure 4: A transformation that is NOT an augmentation.

model, some of the prior observations can become inconsistent with the resulting procedure model, leading to incorrect inference of loop and branch predicates. This is the *consistency problem*. Consider a user interface containing two check boxes labeled *A* and *B*. In the first user's environment both check boxes are unchecked. He mistakenly checks A and correctly checks B. The resulting script is:

```
1 check A
2 check B
```

The user edits the script by removing the first statement. A second user enhances the procedure by providing a new demonstration. In her environment, A is unchecked and B is checked, so she does not perform any actions on the checkboxes. Using all the data, a learning algorithm could produce either of the following two scripts:

```
1 if A is checked, then        1 if B is unchecked, then
2    check B                   2    check B
```

The leftmost model is a result of the inconsistency between the first user's data and the model resulting from the edit; the rightmost model is the desired one.

In this example, ABL selectively discards data inconsistent with the user edit, to ensure that the correct branch predicate is inferred. This illustrates ABL's general strategy for solving the consistency problem: filtering preexisting data to retain only observations that are consistent with the current model structure.

References

Bergman, L. D.; Castelli, V.; Lau, T.; and Oblinger, D. 2005. DocWizards: a system for authoring follow-me documentation wizards. In *Proc. 18th ACM Symp. on User Interface Software and Technology, UIST2005*, 191–200.

Cypher, A., ed. 1993. *Watch what I do: Programming by demonstration*. Cambridge, MA: MIT Press.

Jensen, D. D. 1992. *Induction with Randomization Testing: Decision-Oriented Analysis of Large Data Set*. Ph.D. dissertation, Washington University.

Lieberman, H., ed. 2001. *Your Wish is My Command: Programming By Example*. Morgan Kaufmann.

Oblinger, D.; Castelli, V.; and Bergman, L. D. 2006. Augmentation-based learning: Combining observations and user edits for programming-by-demonstration. In *Proc. 2006 Int. Conf. on Intelligent User Interfaces*, 202–209.

Prabaker, M.; Bergman, L. D.; and Castelli, V. 2006. An evaluation of using programming by demonstration and guided walkthrough techniques for authoring and following documentation. In *Proc. SIGCHI Conf. on Human Factors in Computing Systems*, 241–250.