Learning to Generalize for Complex Selection Tasks

Alan Ritter
University of Washington
Computer Science and Engineering
Box 352350, Seattle, WA 98195, USA
E-mail: aritter@cs.washington.edu

Sumit Basu
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
E-mail: sumitb@microsoft.com

ABSTRACT
Selection tasks are common in modern computer interfaces: we are often required to select a set of files, emails, data entries, and the like. File and data browsers have sorting and block selection facilities to make these tasks easier, but for complex selections there is little to aid the user without writing complex search queries. We propose an interactive machine learning solution to this problem called "smart selection," in which the user selects and deselects items as inputs to a *selection classifier* which attempts at each step to correctly generalize to the user's target state. Furthermore, we take advantage of our data on how users perform selection tasks over many sessions, and use it to train a *label regressor* that models their generalization behavior: we call this process *learning to generalize*. We then combine the user's explicit labels as well the label regressor outputs in the selection classifier to predict the user's desired selections. We show that the selection classifier alone takes dramatically fewer mouse clicks than the standard file browser, and when used in conjunction with the label regressor, the predictions of the classifier are significantly more accurate with respect to the target selection state.

Author Keywords
Interactive selection, learning by example, programming by demonstration, meta-learning, learning user models, file browsers, learning to generalize.

ACM Classification Keywords
H.5.2. User Interfaces (Interaction Styles, Evaluation) I.2.6 Machine Learning.

INTRODUCTION
A variety of common tasks require users to select multiple items from a list: picking out files to delete or copy in a file browser, selecting emails to be moved to a folder, or selecting particular cells, rows, or columns from a spreadsheet. If there are a small number of items, this is a fairly easy task. If there are many, the task may still be quick if the items group together in alphabetical or other attribute order, since many applications have facilities to ease the selection of contiguous groups. However, if the selection is more complex, e.g., the user wants all files that contain "old" in the title, she generally has little recourse but to individually select all of the files. In the case of file selection in particular, there is the option of resorting to the command line or a search interface. From there, one can use regular expressions or complex search syntax to specify an arbitrary list, but this level of complexity is beyond the reach and/or patience of many typical users.

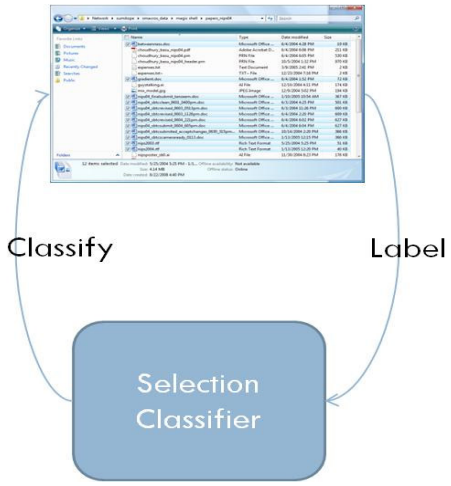

Figure 1. The smart selection scenario: the user selects or deselects a new item and thus provides a label to the classifier, which then classifies all items based on all labels thus far and updates the selection window with its results.

The goal of this work has been to see whether and how it might be possible to give users the power of complex selection directly in the GUI. In our prototype and experiments, we focus exclusively on the file selection task, but our approach generalizes to any other multiple selection scenario.

Given the specific goal of improving file selection, one option would have been to provide an "advanced search" dialog, which would allow the user to specify bounds and ranges for various attributes of the data. This would be quite onerous for the user, however, as she would have to

IUI'09, February 8–11, 2009, Sanibel Island, Florida, USA.

mentally reform her intended selection into a set of rules and conditions and then either enter it into a complex form or type a query with complex syntax. Instead, we chose to pursue an approach of automatic selection generalization or *smart selection*, wherein the user selects and deselects items as positive and negative examples. If the system is effective at generalization, the user should be able to reach her target selection after providing a small number of examples. Furthermore, we would expect that such a method would take fewer clicks than using conventionally available mechanisms in the file browser (sorting, block selection, etc.), especially for complex tasks. Figure 1 shows the workflow of the automatic generalization process. Although it would be possible to highlight the inferred selections in a different color, this would require additional UI elements for deciding when to switch over to the system's predictions. Instead we chose to make as few changes to the user's familiar workflow as possible, i.e., items are either selected or not, and the user makes use of these selections as they would today (deleting, moving, etc.) without the need for additional buttons or keystrokes. We recognize, however, that for everyday use it is likely our method would be something the user would want to turn on only for complex selection tasks or activate via a special key (e.g., alt-click vs. control-click).

We are not, of course, the first to propose the use of learning to help with selection generalization. There is some excellent work on inferring multiple text selections by Miller and Myers [10]: their system inferred regions of text to select from a few fragments selected by the user. While their results looked promising in terms of the number of required examples, the system had limitations with respect to its target function as its inference mechanism was limited to learning a single conjunction of features.

For our system, we were interested both in maintaining the standard selection UI and in handling arbitrarily complex selections. The latter was particularly important to us since item selection tasks (unlike text selection tasks) often require selecting multiple groups with distinct characteristics. As such, we wanted to first propose a learning framework that could handle such selections. We soon found that a mixture of individual classifiers was effective at expressing an appropriate degree of complexity. In our prototype system, we use limited depth decision trees as a component learners, and then used Adaboost (a form of boosting) [7] to combine them into an overall *selection classifier*. As we will later show, this classifier alone takes significantly fewer clicks than standard file browsing mechanisms for many selection tasks.

However, while this classifier has the expressive power to handle arbitrary user selections, we were struck by the data-poor nature of the scenario: for each selection task, a new classifier was being trained using only the positive and negative examples that the user had explicitly clicked on. Surely, though, knowledge of how people tend to do selection tasks could be helpful. In other words, any classifier has its own notion of how best to generalize, but specific populations, tasks, or individuals could have typical patterns of behavior that we should be able to leverage. For example, if the system automatically selects

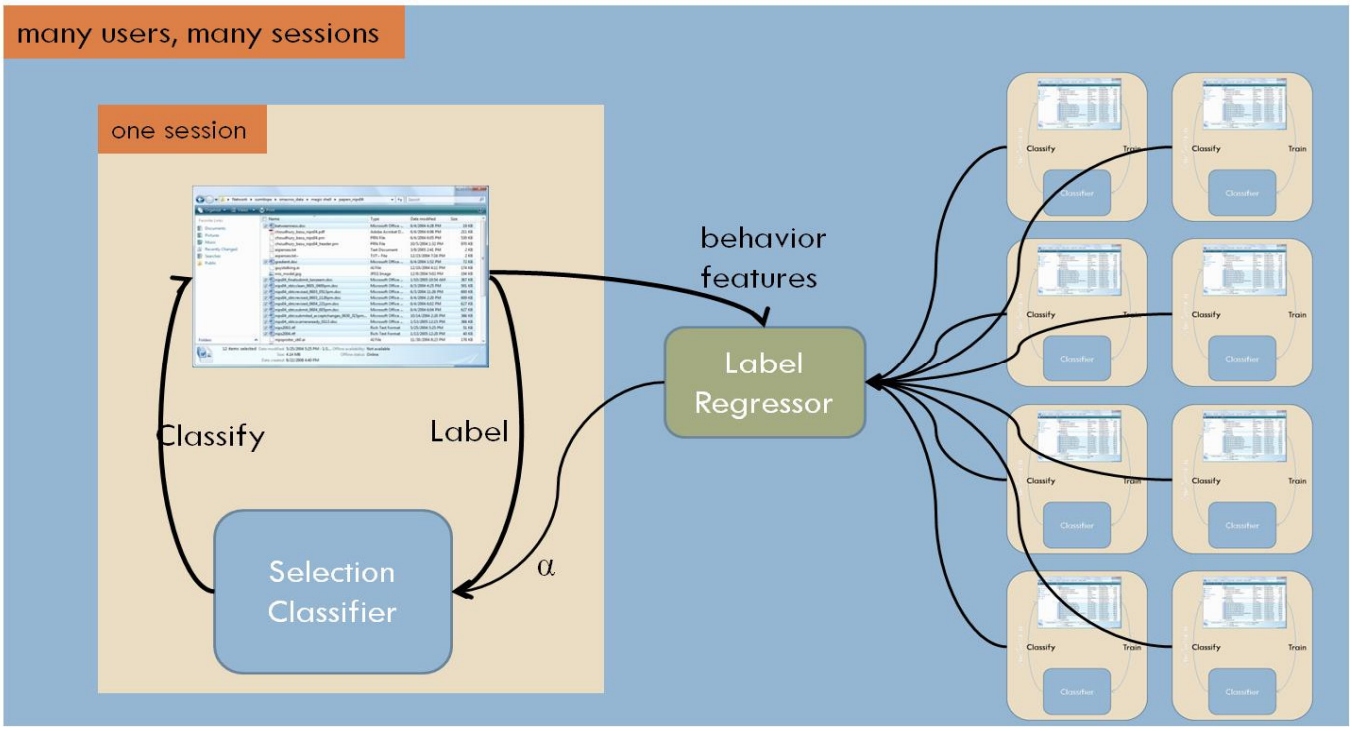

Figure 2. Learning to generalize. For a given session, the selection classifier only has labels for that particular task. However, the label regressor uses data from many sessions of users doing selection tasks in order to form a model of how to best generalize selections. The selection classifier then combines the labels from the user with the predictions of the label regressor in order to come up with the best prediction of the user's desired selection.

some items, and the user does not deselect any of them and instead selects a new example, it may well mean the user is implicitly saying that the system selected appropriate items. The selection classifier has no notion of such concepts: to it, a label is just a label.

One could introduce heuristics such as the one described above based on intuitions or observations of people's behavior, but heuristics always come with their inherent risks. Fortunately, in the smart-selection domain, it is easy to gather labeled training data from many users doing many different selection tasks, since every selection task they complete as they go about their daily business is effectively a training session. Therefore, we sought a way to incorporate this information via learning. We propose a secondary mechanism, the *label regressor*, which is trained on users' previous interactions with the selection classifier, to make probabilistic predictions about what items are likely to be selected. We then feed the soft labels from this mechanism in addition to the user's explicit examples into the selection classifier in hopes of improving performance. This process of *learning to generalize* is shown in Figure 2.

In the remainder of this paper, we will describe the details of our methods and results. We begin with a review of relevant background literature and then explain the file browser prototype we developed. We then describe the selection classifier and the label regressor in detail, and discuss the features we used for both as well as how we integrate their results. We then present the user study we ran to measure the effectiveness of our method and show an analysis of the results. We close with a brief discussion and thoughts about future directions.

RELATED WORK

There has been strong interest in recent years in aiding users with complex tasks via *interactive machine learning*, in which users interactively build classifiers by providing labeled examples or ratings to a learning algorithm. Fails and Olsen introduced this concept in [5], in which they described a framework for this process and presented the "Crayons" tool for finding image regions via interactively building decision trees. In [6], Fogarty et al. presented the CueFlik system, which helps users find images by learning a custom cost function based on the user's feedback on which images are relevant. Kristjansson et al. [8] described an interactive method for optimizing text extractors for structured text (such as addresses) in free-form documents. More specific to the file management domain, Bao and Herlocker [1] presented a method to help users choose the appropriate folder in a folder hierarchy more quickly.

Another body of related work is from the programming by demonstration (PBD) community: as in their scenarios, our system is trying to complete a task based on the examples the user is presenting. Much of their focus, though, is around learning sequences of actions (thus *programming* by example); for instance, the work of Cypher [4] develops a method for automatically creating programs to do repetitive tasks based on examples. In our scenario, the action itself (selection) is simple, but deciding what items to apply it to is the challenge.

The most closely related work to ours is the LAPIS system from Miller and Myers referred to in the previous section [10]. This is a text-editing system which allows users to select multiple text regions for simultaneous editing. LAPIS introduced the concept of "selection guessing," in which users provided positive and negative examples by adding or removing highlighted regions of text; the system then used these to infer the user's intended selection. LAPIS automatically updates its selections after each example and thus has a very similar workflow to our system. However, their system required an option to turn off automatic selection guessing so that the user could manually complete selection tasks. This was due to its hypothesis space being limited to rules consisting of simple conjunctions, so the system was not capable of performing arbitrarily complex selections. We will discuss later why any system may want this option at times, but the potential of learning complex selections was key to our goals.

Our paper builds on the motivation of this pioneering work, but differs in that we are selecting discrete items instead of highlighting arbitrary text fields. In fact, because file selections tend to be more complex than text selections (files containing X or Y larger than Z, etc.), we strived to go beyond the simple conjunctions of the past work. Furthermore, we introduce the concept of *learning to generalize* in interactive machine learning: instead of leaving our classifier to generalize as it sees fit or choosing heuristics about appropriate generalizations, we attempt to learn the way in which the user wants the system to generalize. We do this by making use of more than just the labels the user is giving us for a particular selection task: we also use behavior data from many training instances (from one or more users) to learn a model for what the user is trying to do.

Our motivation of utilizing users' previous selection tasks to improve classifier performance on new tasks is similar to that of Multi-Task Learning [13]. For example, the AutoDJ system [12], automatically generated music playlists from users' positive and negative examples, which are provided by interactively adding or removing songs. Additionally, it uses Kernel Meta-Training to improve its recommendations by training on pre-existing album playlists from one or more users. Our work differs, however, in that we use a secondary learning mechanism which is task-independent to provide additional labeled data based on behavior data from the user.

Finally, our learning method utilizes two separate sets of features, one based on the items in the list and one based on user behavior; therefore it bears similarity to the co-training algorithm proposed by Blum and Mitchell [3]. Our method differs from co-training, however, in that both learning mechanisms are not trained on the same set of labeled data.

Instead, the label regressor utilizes task-independent features and is trained on data from previous tasks, while the selection classifier makes use of task-specific features, and is trained using labeled examples provided by the user (in addition to the outputs from the label regressor).

USER INTERFACE

In order to explore the space of smart selection, we developed a prototype "smart file browser" (Figure 3), which can automatically select files in response to the user's actions. When running in the smart selection conditions (B and C in the experiments below), the browser attempts to generalize the selection and automatically selects those items that it predicts will be selected. The user can then select more items or deselect incorrectly selected items; each such selection will cause the browser to come up with and select a new set based on these examples.

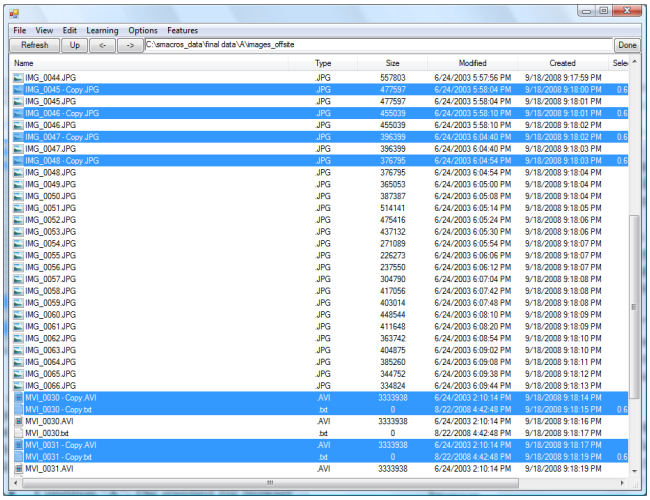

Figure 3. Smart file browser Prototype

We intentionally kept our prototype as similar as possible to the standard Windows file browser with which our subjects were already familiar. To that end, the browser provides most of the features familiar from the standard browser window: sorting by column, block selection (via shift-click, i.e., holding down the shift key and clicking on two items), and of course multiple selection (via control-click). When running in the control condition (A in our experiments below), these commands behave just as they would in the standard browser.

In the smart selection conditions, users engage the selection algorithm by control-clicking[1] on an additional item when at least one item is already selected; we require at least two items as it prevents spurious generalizations based on a single selection. While in this mode; after each example is provided, the system re-trains the selection classifier and updates its predictions on the items which have not been explicitly selected or deselected, updating their state as appropriate. If the user now clicks on an item while not holding down control, all items are de-selected (as in the standard browser), and the current auto-selection context (labeled examples and behavior features) is reset.

Furthermore, as users interact with the browser, all of their actions, in addition to the current sort order and selection state of the files, are logged to a data file. This data can later be used for training the label regressor as well as analyzing various performance metrics.

METHOD

From a high level view, we approach smart selection as a classification problem: the user provides us with explicit positive and negative labels on items, each of which has a multiplicity of features (name, extension, size, creation date, etc.), and we train a classifier based on these features and labels. If the user is unsatisfied with the result, she supplies additional labels, and so the cycle continues until she reaches the target selection state. This simplified version of things is illustrated by Figure 1.

However, there is a subtlety: we can't simply use any classifier that is capable of separating the data based on the labels. If the classifier has too much flexibility, for instance if it *only* selects or unselects those items that the user explicitly labels, it won't generalize the user's selections at all. While not technically wrong, this would not be very useful in the smart selection context. The goal, then, is to satisfy the user's labels while generalizing in the *right* way. A natural choice for achieving this is to restrict the complexity of the decision surface. This was the approach of the LAPIS system we described earlier [10], which restricts the set of learnable hypotheses to simple conjunctions of features. However, such an approach is too limited in the complexity of target selection states it can correctly classify. We thus developed a classification method with flexible complexity, which adds only enough complexity to satisfy the user's explicit labels. This works by using simple component classifiers (limited depth decision trees) that are combined via boosting, as we will describe shortly.

While more capable of handling arbitrary selections, we still had no reason to believe this method would generalize in the *right* way. We realized that this, too, could be cast as a learning problem. Although little labeled data is available for any given selection task, users will generally perform many selections in different directories over the course of their daily activities. Further, it would be easy to obtain labels for all of the items once these tasks are complete, since users implicitly indicate when a given selection is complete by applying some operation (copy, move, delete, etc.) to the selected files. Because these historical examples will come from different directories and/or target tasks, though, we cannot utilize the same file-attribute features as those used by the selection classifier. Instead, we extract a

[1] In a deployed system we could well use another key, for example Alt, in order to give users the choice as to when it is appropriate to use smart selection.

separate set of task-independent features based on the user's behavior. We then combine these features and labels to train a label regressor to predict the probability that a given file will be selected in the target state. We can then feed the outputs of this label regressor as soft labels to the selection classifier so that it can do a better job of generalization. This architecture is illustrated in Figure 2 above.

We now describe the details of the two components of our system: the selection classifier and the label regressor.

The Selection Classifier

For the selection classifier, we use a simple component classifier - a depth two decision tree - which is quite limited in its complexity. We then create an ensemble of these simple classifiers using the AdaBoost [7] algorithm; the resulting classifier is capable of expressing very complex decision surfaces.

During a given round of smart selection, i.e., when the user has given us a new label and we need to retrain and reclassify, we restart with an empty slate and continue adding trees to the classifier combination until all the explicit labels are correctly classified, with up to a maximum of 10 boosting iterations. The cutoff at 10 is somewhat arbitrary, and could easily be relaxed to a higher number; however, we found that this was sufficient to handle every target selection state we tried. Because we only add as many component classifiers as necessary, we found that this method was both resistant to overfitting the small number of labeled examples, and also capable of expressing complex hypotheses when there were sufficient labels to support them.

An additional benefit to this approach is that it lends itself naturally to both data and feature weighting. The data weighting is important in the context of our later integration of the label regressor: we will use the label regressor's scaled outputs to weight its own predicted labels (this process is detailed in the section describing the label regressor). Fortunately, data weighting is a natural extension to decision trees and an inherent part of boosting.

The feature weighting is important as a means of leveraging the connection between mouse position during selection and the relative importance of various features. To see this, we first need to examine the features we use for the selection classifier. While any set of features could be used, in our implementation, the following features are generated from the set of explicitly labeled examples:

- The presence of any substrings of length 3 or greater in the files' names
- The value of the file extension
- The file creation date being greater or less than/equal to each of the file creation dates of the examples
- The file size being greater or less than/equal to each of the file sizes of the examples

Each example thus *generates* several features: for example, the filename "foobar.txt" generates the features {filename contains "foo", filename contains "oob", filename contains "oba", etc.}.

To estimate the importance and generate a weight for each feature, we measured the number of times the user clicked in each of the file attribute columns when selecting or de-selecting items. We found that users frequently click with their mouse vertically placed on the item, but horizontally placed on the column they are looking at: for example, if a user is attempting to select all .zip files whose size is > 3 Megabytes, she is likely to click on that item in the size column. Each feature is assigned a weight equal to the fraction of times it was clicked out of the total. Additionally, we imposed a weak uniform prior on the feature weights by adding a count of $k = 0.1$ to each feature. Rather than simply using this column-weighting feature as a heuristic, though, we used cross-validation to determine that in fact this is a very effective mechanism for the selection classifier.

Other Techniques

We did try using various other learners before settling on boosted decision trees. We enumerate them below, and present the problems associated with each.

1. **Logistic Regression:** Although logistic regression produces accurate probability estimates, and easily allowed us to incorporate weighted training examples, we found that it had trouble learning more complex hypotheses. Additionally we could not as easily incorporate weights for the attribute column clicks in the file browser.
2. **Candidate Elimination:** This algorithm is similar to the learning mechanism used in [10], in that it is only capable of learning a single conjunction of the attributes. Although we found that it worked well in cases where the target selection could be represented in this way, it was incapable of learning more complex selections.
3. **Single Decision Trees:** In addition to using boosted decision trees, we tried using a single decision tree, however because so little labeled data was available, it was difficult to determine when to stop growing the tree to avoid overfitting.

Training Time

Previous work in interactive machine learning [5] has avoided the use of boosted classifier combinations citing slow training time. Although it is true that boosted decision trees will take longer than training a single tree on the data, in our case, training time was not a significant issue in most cases. Because directories usually contain relatively few files, both training and evaluation time are usually negligible. In directories with hundreds of files, however, our prototype implementation would sometimes take several seconds to respond after the user provided an example. Although users found this annoying, we feel it

would not be difficult to reduce training time in a more careful implementation.

The Label Regressor

Again, the purpose of the label regressor is to pool data from many selection session to learn how best to generalize based on behavior features. Its outputs are then used to generate soft-labeled data for the selection classifier, which will be weighted based on the item's confidence. As such, it is not only important that it produce accurate predictions, but that it produce accurate estimates of its confidence as well.

We therefore chose Logistic Regression [2] as our method for this stage, since it produces an estimate p_i of the posterior probability of the target being selected (i.e, ranging from 0 to 1). These probability estimates are converted to a confidence $w_i \in [0,1]$ and a label $l_i \in \{0,1\}$ as follows:

$$w_i = 2 \times abs(0.5 - p_i)$$

$$l_i = \begin{cases} 1 \text{ if } p_i > 0.5 \\ 0 \text{ otherwise} \end{cases}$$

For instance, if the label regressor predicts a value of 0.83 for a given item, its label is 1 or "selected" (since it is greater than 0.5), and its confidence is 0.33 (|0.83-0.5|).

Since the label regressor is trained from many sessions, it can't use file features directly, as these vary from session to session. Instead, it uses *behavior features*, which describe the actions of the user in a session-independent way. We describe the set of features we used below, but it is important to note that this list is not exhaustive; nor are the listed features of equal value. Because we train the label regressor from data, we can hand it an overcomplete "bag of features" that we or others have proposed; it can then sort out for itself which features are actually useful and weight them accordingly. The same applies to the selection classifier as well.

That said, our label regressor features for item *i* were:

- The number of times the user has (de)selected an item while i was (de)selected (4 features)
- Whether the last example provided by the user last changed selection state in the same round as i.
- The proximity of i (in the UI) to the last example provided by the user. Intuition: a user may click in an area where she sees many incorrect examples.
- Whether i appears in between the last two labeled examples in the UI. Intuition: if the user passes over example i with the mouse, its selection state may be correct.
- The distance (in the UI) between the last two examples provided by the user. Intuition: if the user moved a long way to find the next incorrectly labeled example, it is likely that the current selection is mostly correct.

Training the Label Regressor

In order to train the label regressor, we utilize log data from completed selection tasks. We take the selection state of each item (selected/de-selected) at the target state (i.e., when the task is complete) as its label. We then step through each round in the selection process, extracting these behavior features for the items which have not explicitly been selected or de-selected thus far. These features and labels are then used as training data.

Notice that if the user performs a selection task which takes n rounds to complete examples in a directory containing m files, we can express the number of training examples she generates in the process as:

$$\sum_{i=2}^{n} m - i$$

The label regressor thus has the benefit of a substantial number of labels from every selection session.

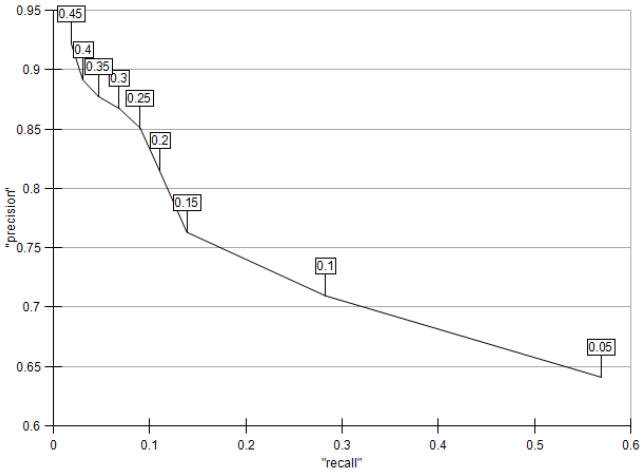

Figure 4. Precision vs. recall for the label regressor for varying levels of confidence. The confidence is shown in boxes above each data point, and represent the absolute distance of the posterior estimate from 0.5.

Because we will depend on the accuracy of the label regressor's confidence, we investigated this empirically by looking at the classifier accuracy with respect to its confidence. If its confidence predictions are accurate, then the classifier accuracy (i.e., precision) should increase with its confidence, while of course the number of items on which it is that confident (i.e., recall) will go down. This is precisely what we see in Figure 4. Furthermore, note that even at fairly low confidence levels (0.2) we have relatively high precision (> 0.8).

Integrating the Selection Classifier and Label Regressor

When operating interactively, behavior features are extracted at each round of the selection process, and the probability that each item will be selected, p_i, is estimated by the label regressor; these probabilities can be interpreted as a label and a confidence value as described above.

We now wish to incorporate these outputs into the selection classifier. Since it was a requirement from the start that the selection classifier could take in weighted data, we could simply incorporate the labels from the label regressor and use the confidences w_i as weights for all unlabeled items (with weights of 1.0 and the entered labels for the explicit selections from the user). However, it would be naïve to think that taking the raw confidence value would be the optimal weights to use in the selection classifier's boosted decision trees. Perhaps this would be more appropriate if the model explicitly took in posterior probabilities for each item, which is in fact what our label regressor produces, but in our setup that is not the case.

As a result, we decided to apply a scale factor α to the confidence of the label regressor, which we can optimize empirically such that it produces the highest *overall* selection classifier accuracy. Of course, we could apply other transformations as well, such as multiplying the value by a parametrized nonlinearity, or only using label values with the confidence above some threshold: the key is to optimize any such parameters in terms of end-to-end accuracy.

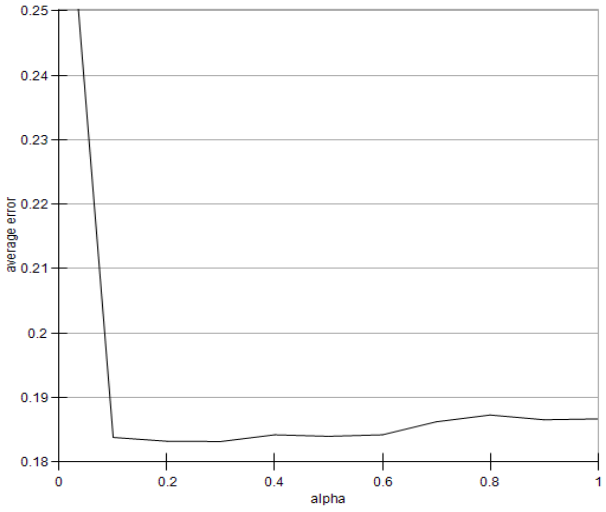

Figure 5. Selection classifier error on all pilot data tasks/sessions vs. varying values of the α parameter; α multiplies the label regressor confidence values to produce the weights for the selection classifier.

In Figure 5, we show the error rate of the selection classifier with respect to α averaged over all rounds of all of our pilot data (this is a separate population from those in our final user study). Based on this investigation, we set α to 0.2 for our final user study.

EXPERIMENTS

In order to evaluate our method, we conducted a user study with 12 subjects (11 males, 1 female), each of whom were compensated for their time with a $10 gift certificate. All were Windows users and had experience using the standard file browser to select multiple files. We asked participants to complete 8 different selection tasks (described later in this section) under 3 conditions:

- **Condition A**: standard file browser (control)
- **Condition B**: selection classifier alone (without the label regressor)
- **Condition C**: selection classifier with the label regressor

Both the order of the tasks and the conditions were randomized, and at the end of conditions B and C users were asked to provide comments and rate various aspects of the system. At the end of the study, users were asked to rank the three conditions in terms of which they would prefer if it were the *only* option for everyday usage.

During conditions B and C, we asked participants not to make any *block selections*[2] using the shift-key (note that block selections were allowed for condition A). Although this was not necessary for our framework, since we could easily accept multiple labels per round, we wanted to see how well the smart selection methods could perform when only one item was selected at a time. This also helped ensure that the test subjects couldn't default to their typical strategies and would thus be more likely to take advantage of the smart selection predictions.

Bootstrap Stages

Before we could apply condition C, we first had to train the label regressor on other users' task selection data. We obtained this data in a separate pilot study of 9 users, all using condition C, on a different but similar set of tasks. For the first 5 pilot study participants, the label regressor was trained from one of the authors performing selection tasks on a separate set of directories; for the last 4 users the label regressor was trained from the data of the first 5 users. For the final study, on 12 new subjects, the data from all 9 users was used for training.

Selection Tasks

The selection tasks and directories were designed to be a diverse mix containing widely varying directory sizes. We took care not to make any of the selection tasks too tedious when using the standard file browser, so as not to make participants overly uncomfortable during the study. Half of the tasks (0, 2, 5, and 6) were easily solved using the standard file browser via sorting and block selection. The tasks in each condition were slightly different but isomorphic so that the subjects couldn't use their memory of a previous task to help them with a different condition. One such isomorphic set of the tasks was as follows:

- **Task 0:** *Select all .py and .cs files*. This was in a large directory containing many .py files and a few .cs files. This task was easily completed in the

[2] We refer to any time the user selects a block of files which are adjacent in the UI (via shift-clicking) as *block selection*.

standard file browser by sorting and then block-selecting with the shift key.

- **Task 1:** *Select all files whose name contains the substring "OLD" or "BACKUP" in a medium sized directory.* This task was more difficult to complete with the standard file browser, as there was no way to sort the target selection into a contiguous block.
- **Task 2:** *Select all .pdf files whose size is greater than 200K.* This was in a large directory with relatively few .pdf files. This task was not possible to sort/block select, but was not too difficult with the standard file browser since there were only a few files to select.
- **Task 3:** *Select all files whose name contains the substrings "MARKF" or "JOEP" but not the string "tmp."* This task required selecting a large number of files in a large directory, and was difficult to complete using the standard file browser. Note that the string "tmp" was a substring in the filename and not the extension, which made block selection impossible.
- **Task 4:** *Select all files containing the substring "kathyp."* This task required selecting relatively few files in a medium-sized directory, and was not easily solved using the standard file browser.
- **Task 5:** *Select all files containing the substring JOEP whose size is larger than 3 Megabytes.* This task required the user to select relatively few examples in a very large directory. Block selection was possible if sorted in the right way (not all users discovered this).
- **Task 6:** *Select all .txt and .cab files* in a medium-sized directory. This task was relatively easy to complete using the standard file browser by sorting and block selecting.
- **Task 7:** *Select all files containing the substring "Copy."* This task required selecting slightly fewer than half the files in a medium-sized directory, and was not easily solved using the standard file browser.

RESULTS

We describe our results both in terms of the numbers of required examples and the accuracy of the selection classifier; we will discuss why both measures are important.

Number of Required Examples

The average numbers of user-supplied examples (selected/deselected items) required to complete each task under each condition are shown in Figure 6. Note that for condition A, one "example" might mean many labels, since the subject can hold the shift key and block-select many items via a single item selection.

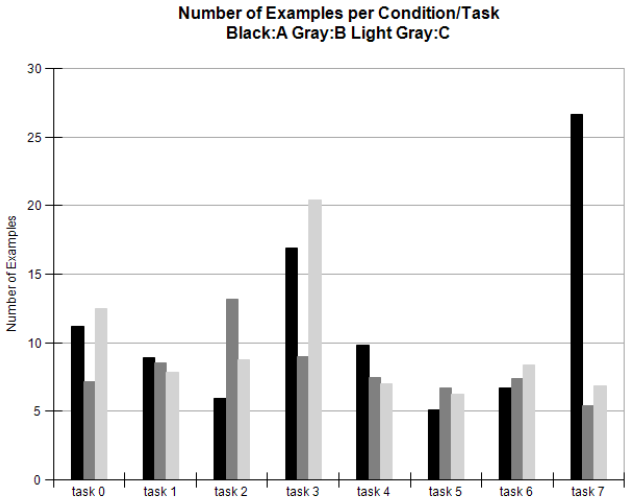

Figure 6. Average number of examples (number of items clicked by the users) for completing each task in each condition.

As expected, the tasks that were easy to do with the standard file browser (0, 2, 5, and 6) show minimal improvement or more often a slight increase in selections for conditions B and C - this is due primarily to the fact that we didn't allow users to use block selection for these conditions. The harder tasks, though, (1,3,4, and 7) almost uniformly show marked improvements for the smart selection conditions. The mean time for each condition is shown in Table 1.

Condition	Mean number of examples to complete task	Median
A	11.4	9
B	8.09	7
C	9.47	8

Table 1. Number of examples for each condition averaged across tasks. All differences are significant as illustrated in Table 2.

We performed a two-way analysis of variance (ANOVA) test with the number of examples as the dependent variable and with condition and task as the two independent variables; the results are displayed in Table 2 below. This means both smart selection conditions (B and C) required significantly fewer examples/selections to reach the target state. Condition C (using the label regressor) did require slightly more examples than B (selection classifier alone), but had other benefits which will become clear in the accuracy results below.

In summary, auto-selection generally requires fewer explicit selections and de-selections than the standard file browser in tasks where it is not possible to sort the files in such a way that the targets appear in one or more contiguous blocks. When such sorting is possible, users

can use block selection in condition A to complete the task in a few clicks. Furthermore, we noticed that all of the users in our study sorted first when block selection was possible, regardless of the condition. Although they weren't allowed to block select in the B and C conditions, they still preferred to sort the target selections into a block before selecting them.

Comparison	p-value
A vs B	1.338×10^{-8}
A vs C	3.196×10^{-3}
B vs C	2.102×10^{-2}

Table 2. Comparing number of examples required between conditions by a 2-way ANOVA across tasks and conditions

Selection Classifier Accuracy

In addition to the total number of examples, another key metric for our system is the accuracy of the selection classifier. This is important since we want to generalize in the *right* way as we described above – ideally, at every step, the new items that the system selects and deselects should be as close as possible to what the user intends to do. The less that this is the case, the more likely it is that the user will get confused and annoyed with the erroneous selections being presented to them.

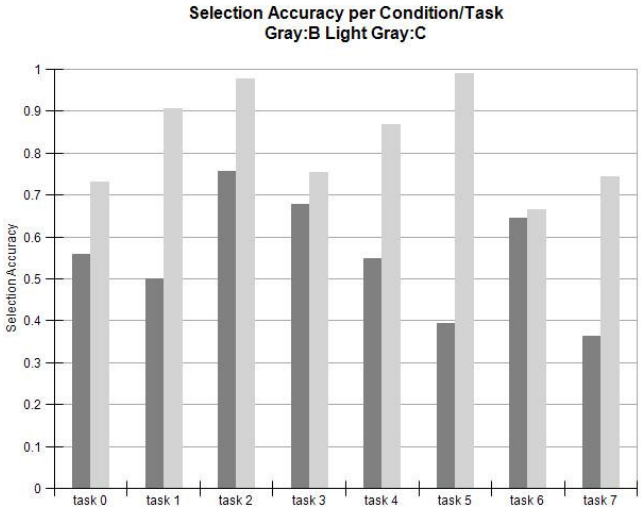

Figure 7. Average selection accuracy for each task under smart selection conditions B (no label regressor, dark gray) and C (using label regressor, light gray).

In Figure 7, we compare the accuracy of the selection classifier in conditions B and C for each task, averaged over all users. C is substantially more accurate in every case. Overall, this represents an average increase in accuracy of 49% when using the label regressor (see Table 3), a result that is significant at the 95% confidence level. This means that using the label regressor reduces the relative error rate by a full 60% (from 45% error to 18% error).

Condition	Average task accuracy	Standard Deviation
B	0.555	0.136
C	0.820	0.122

Table 3. Accuracy of the inferred selections in both auto-selection conditions averaged across tasks. In a 2-way ANOVA across condition and task, C is more accurate than B with a p-value of 0.04681.

The reader may wonder at this point how such a large improvement in accuracy doesn't also result in a significant reduction in the number of clicks. The reason is that when using the label regressor, the selection classifier gets *close* to the right answer much more quickly: however, getting the selection 100% correct takes about the same number of examples. In other words, while the label regressor won't get to the right answer more quickly, it will get there in more of the *right way*. We show this on an example task (task 4) in Figure 8, with the accuracies at each round averaged over all users. Notice that in only four clicks, the classifier using the label regressor (C) has the selection more than 95% correct on average, while without it the performance lags at less than 80%.

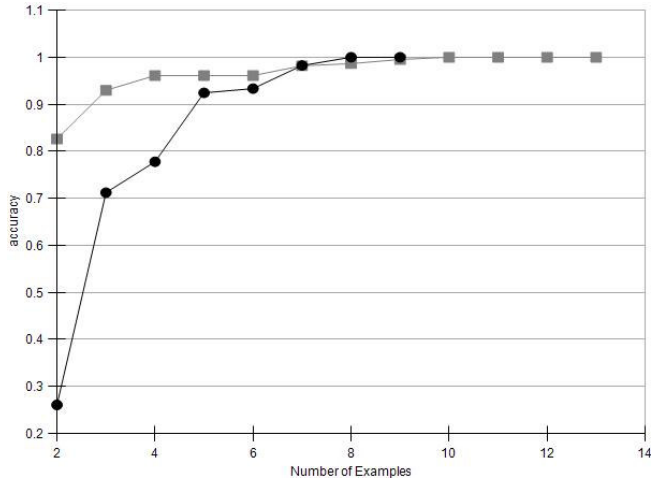

Figure 8. Average selection accuracy for each round in task 4 for condition B (black, round dots) and C (dark gray, square dots), averaged over all subjects. Note that when using the label regressor (condition C), the system gets closer to the correct answer much more quickly, though both take about the same number of examples to reach the target exactly.

From a user's point of view, this poses a significant advantage: the system is more quickly approaching the right answer. This is particularly the case in a real deployment where at any time the user could switch from smart selection to standard selection (e.g., control-click vs. alt-click) and quickly highlight the few remaining examples.

The comments from our users supported this notion: the following are a few examples from the response section of the study:

> "*The 2nd method* (B) *seemed a more "aggressive" version of method 1* (C). *However the UI presentation i.e. the selection and deselection of large numbers of files strained my eyes and annoyed me.*"
>
> *"Selecting more files than desired can seem dangerous in some situations - especially when selecting files to delete or modify."*

Overall 7 out of the 12 participants indicated that they preferred condition C (i.e., using the label regressor) over condition B; we expect if we allowed them to switch to manual selection in the middle of the task even more would have preferred condition C.

Personalization

In addition to benefiting from learning how users as a whole prefer to generalize, we hypothesized that we might get further gains from training on a particular user's data in order to capture their individual styles of generalization. Indeed, during the user study, we observed that users attempted a wide range of strategies to communicate their indented selection. For example, when possible, several users tried sorting, then selecting the top and bottom items of blocks, whereas others clicked on incorrectly selected items from top-to-bottom order.

Although we did not have users generate their own training data during the study, we did a post-study analysis to evaluate the potential benefits of personalization. We used cross-validation to compare the accuracy of the selection classifier used during the study with simulated accuracy after training on the target user's data (but excluding data from the test task). Overall we saw a very modest (2%) improvement in overall accuracy (significant with a p-value of 0.068). Thus, while there is a small benefit from personalization in this scenario, by far the most significant gains were from learning generalization strategies from the user population as a whole.

DISCUSSION

We have presented two primary contributions in this paper. The first is a selection classifier that is flexible enough to achieve complex selection targets in a small number of examples; it is in fact significantly faster (in terms of number of clicks) than the standard file browser on a variety of tasks, even without block selection. The second is a general method for learning to generalize, which leverages many sessions of user tasks in order to learn a model of how users prefer to generalize selections; we also present a means to integrate the results from this learner into the selection classifier. The result of this integration is a substantial increase in the selection accuracy of the classifier, which means the system is more likely to generalize the user's selections in the *right* way.

In future work, we hope to develop a deployable system with this technology that will allow users to easily switch between smart selection and standard selection; this will let them more easily take advantage of the rapid convergence towards the target state seen in Figure 8.

In closing, we would like to emphasize that the particular task and the particular features we used in this work are merely an illustration of our method: future applications using this method could add arbitrary sets of features for their selection tasks and let the classifiers select which ones are worth emphasizing.

ACKNOWLEDGEMENTS

Many thanks to Krzysztof Gajos and Alice Zheng for helpful references and insightful discussions.

REFERENCES

1. Bao, X., Herlocker, J. and T. Dietrich. "Fewer Clicks and Less Frustration: Reducing the Cost of Reaching the Right Folder." *Proc of IUI 2006.*
2. Bishop, C. "Linear Models for Classification." In *Pattern Recognition and Machine Learning.* Springer, 2006.
3. Blum, A. Mitchell, T. "Combining Labeled and Unlabeled Data With Co-Training." In *Proc. of COLT 1998.*
4. Cypher, A. "Eager: Programming Repetitive Tasks by Demonstration." In A. Cypher, ed., *Watch What I Do: Programming by Demonstration*, pp.205-218. MIT Press, 1993.
5. Fails, J.A. and Olsen, D.R. Jr. "Interactive Machine Learning." In *Proc. of IUI 2003.*
6. Fogarty, J. Tan, D. Kapoor, A. and Winder, S. "CueFlik: Interactive Concept Learning in Image Search." In *Proc. of CHI 2008.*
7. Freund, Y. and Schapire, R.E. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting." *Journal of Computer and System Sciences 1997.*
8. Kristjansson, T. Culotta, A. Viola, P. McCallum, A. "Interactive Information Extraction with Constrained Conditional Random Fields." In *Proc. of AAAI 2004.*
9. Maynes-Aminzade, D. Winograd, T. Igarashi, T. "Eyepatch: Prototyping Camera-Based Interaction Through Examples." In *Proc. of UIST 2007.*
10. Miller, R.C. and Myers, B.A. "Multiple Selections in Smart Text Editing." In *Proc. of IUI 2002*.
11. Mitchell, T.M. "Version Spaces: A Candidate Elimination Approach to Rule Learning." In *Proc. of IJCAI 1977.*
12. Platt, J.C. Burges, C.J.C. Swenson, S. Weare, C. Zheng, A. "Learning a Gaussian Process Prior for Automatically Generating Music Playlists." In *Proc. of NIPS 2002.*
13. Thrun, S. "Is Learning the n-th Thing Any Easier Than Learning the First?" In *Proc. of NIPS 1996.*