David Maulsby
Aurelium Inc.

Ian H. Witten
University of Waikato

Teaching Agents to Learn: From User Study to Implementation

By testing and critiquing our design ideas with human users, we were able to stay focused on our most important objective: intelligent agents that make computer-based work more productive and more enjoyable.

raphical user interfaces have helped evolve a world of easily accessible information, where computer use centers on viewing and editing, rather than on programming. Yet the need for end-user programming is becoming increasingly apparent: Users, who often find themselves performing repetitive tasks, want the ability to customize applications easily. Software developers have responded to this need with a barrage of customizable applications and operating systems. But the learning curve associated with a high level of customizability—even in GUI-based operating systems—often prevents users from easily modifying their software. So, ironically, the question becomes, "What is the easiest way for end users to program?"

Most user interfaces eschew formal notation and encourage concrete expressions of a user's ideas.[1] Perhaps the best way to customize a program, given current interface and software design, is for users to annotate tasks—verbally or via the keyboard—as they are executing them. Experiments have shown that users can "teach" a computer most easily by demonstrating a desired behavior.[2] Teaching demands less planning, analysis, and understanding of the computer's internal operations than programming does, but it raises new questions about how the system, as a learning machine, will correlate, generalize, and disambiguate a user's instructions. Human pupils can readily interact with their teachers, but a

0018-9162/97/$10.00 © 1997 IEEE

computer needs more help. In addition, a computer will likely fail in its mission if it confuses or frustrates the user.[3]

In order to understand how best to create a system that can learn, we conducted an experiment in which users attempt to train an intelligent agent to edit a bibliography. In our experiment, a researcher sat behind a curtain masquerading as the intelligent agent, which we named Turvy (after a fictional character who succeeds in life by appearing more intelligent than he actually is). This experimental setup—with a hidden researcher simulating computer behavior—is often called a Wizard of Oz scenario, the structure of which is shown in Figure 1.[4]

To make the experiment as useful as possible for constructing a real software agent, we set limits on Turvy's—the researcher's—abilities. In particular, we limited the types of instructions Turvy could understand, the features of text it could observe, and the generalizations it could learn. Users engaged in activities that ranged from easy (replacing underlining with italics) to taxing (putting all authors' given names and initials after their surnames). In the course of the experiment, Turvy had to do things like select many short strings of text, keep track of errant white space, and handle names containing "van" or "de." Users invented their own teaching methods and spoken commands to help Turvy learn even the most difficult elements of the experimental tasks.

Armed with the results of these experiments, we implemented an interactive machine learning system, which we call Configurable Instructible Machine Architecture, or Cima. Designed to acquire behavior concepts from few examples, Cima keeps users informed and allows them to influence the course of learning. We have tested Cima on transcripts from the Turvy experiment and found that it learns at roughly the same level as the researchers masquerading as Turvy.

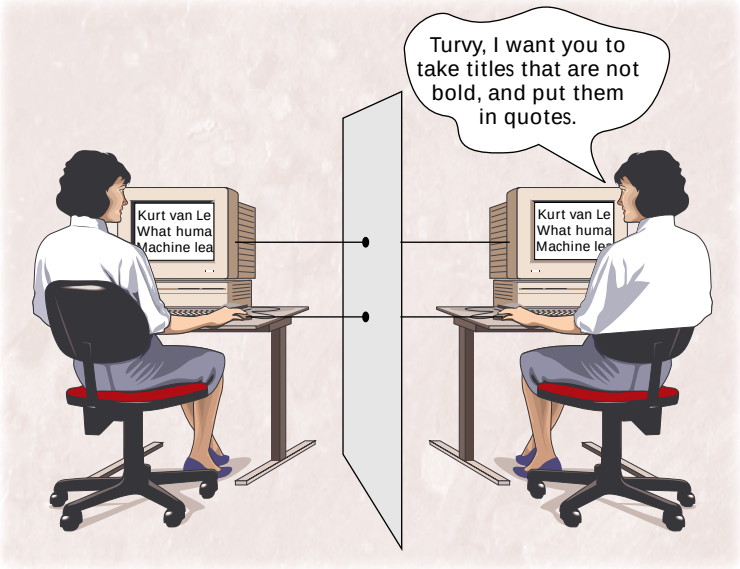

Figure 1. The experimental setup showing Turvy on the left and the user on the right. Interaction works bidirectionally in this Wizard of Oz scenario that is designed to help researchers better understand how users might work with intelligent agents.

STUDYING USER BEHAVIOR

The best way to understand how we used Turvy to understand user behavior is to examine Turvy in action. Figure 2 shows a particular Turvy task: Make a heading for a bibliography item—showing the author's surname and the date of publication—by extracting the correct information from a bibliographic listing. With no background knowledge of bibliographies, names, or dates, Turvy must learn these concepts on the fly while performing this task. The relevant surname is normally the word before the first comma or colon, but sometimes includes a lowercase "van," or precedes an "ed." And in some entries the author's surname and initials are reversed. The relevant date includes the last two digits before the period at the paragraph's end, although sometimes the final period is missing. Turvy must learn special-case rules to handle these types of exceptions. A transcript from the

Philip E. Agre: The dynamic structure of everyday life: PhD thesis: MIT: 1988.	[Agre 88] Philip E. Agre: The dynamic structure of everyday life: PhD thesis: MIT: 1988.
D. Angluin, C. H. Smith: "Inductive inference: theory and methods." *Computing Surveys 3 (15)*, pp. 237-269: September 1983.	[Angluin 83] D. Angluin, C. H. Smith: "Inductive inference: theory and methods." *Computing Surveys 3 (15)*, pp. 237-269: September 1983.
Michalski R. S., J. G. Carbonell, T. M. Mitchell (eds): *Machine Learning II:* Tioga: Palo Alto CA: 1986	[Michalski 86] Michalski R. S., J. G. Carbonell, T. M. Mitchell (eds): *Machine Learning II:* Tioga: Palo Alto CA: 1986
Kurt van Lehn: *Mind bugs: the origins of procedural misconceptions:* MIT Press: 1990.	[van Lehn 90] Kurt van Lehn: Mind bugs: the origins of procedural misconceptions: MIT Press: 1990.

Figure 2. As instructed, Turvy takes information from a simple bibliography entry on the left to make a heading with an author's name and date of publication on the right. Figure 3 shows the transcript of how one user worked through this exercise.

Pam: Take the last name, before the colon.
[Pam copies "Agre," pastes it to a new paragraph, types brackets, and sets style.]
Pam: This is a list of publications. Now we'll add the last two digits of the date.
[Pam types "88."]
Pam: Do you want another demo?
[Turvy interprets Pam's question to mean the example is completed. Turvy has formed an initial generalization and now offers to edit the second entry.]
Turvy: I can try if you want.
Pam: OK.
Turvy: Looking for word before first colon in paragraph, OK?
[Turvy picks "Smith."]
Pam: No, stop! You want the first author.
Turvy: Show me please.
[Pam picks "Angluin."]
Turvy: You picked word before comma. You want word before first comma or colon?
Pam: Yes! The first comma or colon.
Turvy: [while doing actions} Copy, go to start of paragraph, OK?
Pam: OK.
Turvy: Insert new paragraph, type open bracket, paste, close bracket, change style to "citation," OK?
Pam: OK.
[Now Turvy makes a mistake with the date. When Pam typed "88" Turvy assumed the number was either a constant or input.]
Turvy: Type in blank space "88," OK?
Pam: No, the last two digits at the end.
Turvy: Show me please.
[Pam picks "83."]
Turvy: Two digits before period at end of paragraph?
Pam: Yes.
[Later, Turvy encounters author "van Lehn," mistakenly selecting "Lehn." When the user corrects the action, Turvy asks for an explanation.]
Turvy: What's different about this case?
Pam: Hmm.
Turvy: You selected these two words. Why?
Pam: Because these two words make up his last name.
Turvy: Is there some feature of this word [Turvy highlights "van"] I should look for? Lowercase word?
Pam: [hesitantly] In this case, yes.
[Turvy accomplishes the rest of the task without error.]

Figure 3. Turvy learns how to accomplish a task by watching a user's behaviors and asking questions.

videotaped record of this experiment appears in Figure 3. Pam, the user, has already completed two tasks. Prior to the experiment, a facilitator told her that Turvy is simulated by a human being, that it watches her actions, and understands a bit of English. The facilitator gave no other suggestions on how to teach Turvy.

As the transcript shows, Turvy has a conversational interface in which either party may take the lead and both are learning about one another's goals.[5] Turvy forms an initial hypothesis from the user's first example, and thereafter mixes prediction with conclusions drawn from other user input. Turvy annotates predictions with verbal feedback, encouraging the user to annotate her examples with hints. When testing a hypothesis for the first time, Turvy offers a detailed description; as it gains confidence, it reduces feedback and increases execution speed.

Although they are told to teach Turvy, subjects are likely to have other goals: to finish the session as quickly as possible, to make a good impression on the researcher, or maybe to compete with Turvy. These motivations influence the number and complexity of instructions users give. As an experimental instrument, Turvy probes for more detailed instructions than might be appropriate in a real system. To elicit as much experimental data as possible, while keeping the session on schedule, the researcher may make Turvy more passive or more proactive.

Observations and results

The Turvy experiment is an unusual example of participatory design applied to an artificial intelligence system. In fact, our observations of user interactions with Turvy led to the functional specification of our learning algorithm in Cima. We paid special attention to the forms of instruction users adopted, the ambiguities they presented, and the possible sources of disambiguation inherent in natural combinations of demonstrated action with verbal and gestural annotation. We analyzed some 20 hours of videotape to identify common forms of instruction that were both natural for users and feasible for a computer to interpret.

Four users participated in a pilot study and seven in the formal experiment. The subjects included secretaries, an office manager, a multimedia artist, and students in psychology and computer science. Their level of computer experience ranged from novice to professional. We attempted to choose a reasonable cross section of users who might routinely compose structured text documents with a computer. We also wanted to include both users who had insight into computer programming (such as computer science students) and those who would have little or no experience with programming (such as graphic artists). We were constrained in our choice of subjects by the need to keep the experiment small enough for a single researcher to document and assimilate the data. We did, however, test our observations informally on many other subjects over a course of three years.

Each subject worked with Turvy for about one hour. We videotaped each user in action and interviewed the users afterward. Although the video transcripts were analyzed quantitatively, we focused more thoroughly on users' qualitative accounts of their experiences.

Dialog styles. Users were evenly split between two dialog styles: talkative and quiet. A typical talkative user, like Pam, gives a detailed verbal description even before starting a task, to which Turvy replies, "Show me what you want." The user performs a single example and asks Turvy to try the next. If Turvy makes a mistake, the user says, "Stop," and tells it what to do. Turvy says, "Show me what you want," and the user

performs the correction, repeating the verbal hint. Turvy might ask for features that distinguish the user's correction from Turvy's attempts. Sometimes the user is puzzled, so Turvy proposes a course of action, which the user either accepts or modifies with another hint.

Quiet users work through the first task without giving hints or inviting Turvy to take over. When Turvy detects repetition, it interrupts, saying, "I've seen you do this before, can I try?" After some hesitation, the user consents. When Turvy makes a mistake, the user says, "Stop," then demonstrates how to correct the mistake. Quiet users are sometimes reluctant to answer Turvy's questions about the features distinguishing the correction from Turvy's attempt. Rather than explain to Turvy how to handle a troublesome case, a quiet user will likely tell Turvy to skip it.

Command set. All users discovered the same set of commands that Turvy could best understand. To control learning, users would say to Turvy, "Watch what I do" or "Ignore this." To control prediction, they would say, "Do the next one" or "Do the rest." The actual wording varied little, and all users adopted standard terminology once they heard Turvy use it, as when Turvy asks, "Do the next one?" Subjects used fewer forms of instruction for focusing attention than expected. They almost never volunteered vague hints like "I'm repeating actions" or "Look here," and instead mentioned specific features, as in "Look for the colon before italics."

TurvyTalk. We found that users do learn to describe concepts like titles and authors' names in terms of syntactic features—but only after hearing Turvy do so. In the pilot study preceding the formal experiment, Turvy did not verbalize its actions, so users had no idea how to answer questions like "What's different about this case?" While Turvy had the same learning abilities in the pilot study as in the formal experiment, in the pilot study Turvy did not suggest generalizations to the users. As a result, users did not learn Turvy's language and did not readily recognize the importance of giving Turvy verbal hints. Moreover, they had very little understanding of Turvy's native intelligence—the basic concepts from which it could build new knowledge. We came to understand that users learn how to talk to Turvy the way they learn how to talk to people, by mirroring speech patterns.[6]

Teaching difficulty. We've also determined that programming by demonstration should make simple tasks easy to teach and complex tasks teachable. In the postsession interviews, all subjects reported that they found Turvy easy to teach, especially once they realized that it learns incrementally, so that they need not anticipate all special cases Turvy might encounter. In fact, the ability to infer and generalize was Turvy's most popular feature.

General observations. After their sessions, most subjects commented that they had more confidence in Turvy's ability to understand speech because they knew Turvy was human. Even so, we found ample evidence in the video record that our subjects were interacting with Turvy as if it were a real computer agent. In particular, after the first few minutes, the subjects would commonly adopt a terse, clipped form of speech such as one might expect a machine to understand. They dispensed with social niceties of turn-taking in conversation and were often brusque and even impolite toward Turvy. Moreover, they consistently referred to the agent as "it" when talking to the researchers, and they often expressed doubt as to whether "it" would understand what they were trying to do or explain. When Turvy made a mistake, the quiet users in particular became anxious, evidently believing it would be impossible to make the agent understand what was to be done. Clearly, our users were not ascribing human-level capabilities to Turvy.

Clearly, our users were not ascribing human-level capabilities to Turvy.

All experimental subjects said they would use Turvy if it were a real system. All were concerned about completeness, correctness, and their own autonomy. They believed it would be foolhardy to leave Turvy unsupervised, but they concluded, on the basis of their experience, that using Turvy would save them time and effort, freeing them to concentrate on the more important aspects of writing.

One user refused to work with Turvy altogether. This person rejected the very idea of an instructable agent, believing that he would have to anticipate all special cases in advance, as in writing a program.

Lessons for intelligent agents

Researchers have implemented programming-by-demonstration techniques in an assortment of research projects, ranging from systems that automatically generalize user actions to ones that allow users to control generalization explicitly. None, however, provide the flexibility of interaction and learning strategies that we designed into Turvy. Based on our observations, the key learning strategies include generalizing from a single example and from multiple examples; using background knowledge to make plausible generalizations; and evoking and interpreting verbal and gestural hints from the teacher to focus the learner's attention.

We learned several things that apply to intelligent agents in general, and which existing research systems do little to address:

- Users appreciate and exploit incremental learning. They are content to teach special cases as they arise, rather than anticipating them.
- Many users want to augment demonstrations with verbal hints, and find it easy to do so.

```
repeat until all positive examples are covered
  by some rule
make a new empty rule R
inner loop: repeat until rule R covers no
  negative examples
     choose an attribute value A that
          maximizes coverage of positive
          examples not already covered by
          some rule,
          and minimizes coverage of negative
          examples
     add (conjoin) A to R
     end inner loop
prune attributes from R that were rendered
  unnecessary by those added later
     add (disjoin) R to the set of rules
```

Figure 4. Cima induces a concept using the greedy covering algorithm, which starts with a most general description and repeatedly specializes it by adding attribute-value tests, until the description covers only positive examples. It forms rules that are guaranteed to distinguish positive examples from negative, provided the language of attribute-values permits such distinctions at all.

- Users don't require static description of what the agent has learned. Concise verbal feedback, given while predicting actions, maintains user confidence.
- By using its input language in feedback, an agent helps users learn how to teach.
- To elicit a hint, it is better to propose a guess than to ask the user, "What's relevant here?"
- Both agent and user must be able to refer to past examples and instructions.

The manifest advantages of this style of interaction—and the general success of the Turvy study—led us to develop Cima, an agent implementation of Turvy's learning mechanism.

IMPLEMENTING CIMA

Cima is a machine learning system that implements Turvy's learning strategy and techniques. It is readily configurable with primitive, general knowledge of a domain (such as textual syntax) and specialized knowledge of a particular application (such as word processing). The learning algorithm in Cima forms rules for identifying patterns in data from the forms of instruction Turvy could handle: a single example, multiple examples, negative examples, and user hints.

Connected to a text editor within the Macintosh Common Lisp environment, Cima learns concepts that characterize textual structures. It learns from positive and negative examples: Positive examples are either text selections the user actually edits or predicted selections the user accepts; negative examples are predictions the user rejects. To give a "pointing hint," the user selects text and chooses "look here" from a pop-up menu. Users type verbal hints in a separate window. Our experience with Turvy demonstrates that hints are bound to be ambiguous, incompletely specified, and sometimes even misleading. Therefore Cima interprets them in light of two factors: domain knowledge about text editing and the situations already encountered in a particular user session.

Cima learns to describe a concept, such as a surname, in terms of rules that classify examples as positive (members of the concept) or negative (not a member). For instance, "surname" might be defined as follows:

An example text is a surname if and only if it satisfies one of the following rules:

1. the text is a capitalized word AND the text follows a single capital letter ending with a period; OR
2. the text comprises "van" followed by a capitalized word.

Each rule represents one "special case" of the concept, and all examples under a given case have all the attributes specified by the rule, though they may have other attributes as well. Attributes are the observable or logically deducible characteristics of an example, such as a word distinguished by its capitalization or a text string matching the pattern `<capital letter><period>`. Where each rule is a conjunction of tests joined by AND, the set of rules forms a disjunction joined by OR, the structure of which is called a Disjunctive Normal form (DNF). Such rule sets may be constructed by a "greedy" covering algorithm, as shown in Figure 4.[7]

We modified the learning algorithm shown above in two important ways. First, the algorithm's inner loop continues to add features until a classification rule not only excludes all negative examples (as in the standard algorithm) but also meets operational criteria, which set the conditions a generalization must meet if it is to be translated into a specific, concrete action when the agent makes a prediction. Second, the system selects features not only for their statistical utility, but also for a combination of other criteria: whether the user has suggested new operational criteria, whether those criteria are used in other rules, whether they contribute to operationality, and whether they are salient (a priori) to the system's domain knowledge.

Figure 5 shows a sample of Cima's background knowledge for matching, generalizing, and assessing feature relevance in the domain of text editing. (Cima is not an algorithm for learning textual patterns per se, and in fact it has been tested in other domains, including number patterns, Lego block assembly, and file management.) The system encodes knowledge in Lisp as association lists, predicates, and procedures. It includes facts about data types, generalization hierarchies, methods for matching and generalizing examples, default rankings for the salience (that is, "interestingness") of example attributes, and directed graphs that encode suggested changes in focus of attention. Details of these mechanisms, which are complex and not particularly elegant, can be found elsewhere.[8]

The best way to understand what Cima does is to see it in action. Although the system was evaluated on the Turvy tasks described above, for variety's sake we use a different example here.

Suppose the user wants to train an agent to dial phone numbers contained in a text file of addresses. When she clicks the mouse anywhere inside a phone

1. Built-in Concepts (Patterns)

Pattern-matching knowledge is declarative or procedural:

```
(declare_class 'EnclosureChar `('(' ')' '[' ']' '{' '}' '\'' '\"'))
(defmethod isa ((theToken TextToken) (eql theClass :EnclosureChar))
      (find (text theToken) EnclosureChar))
```

Generalization hierarchies are used for matching or finding a generalization:

```
(declare_hierarchy 'TokenTypes
      (:Character (:Alphanumeric (:Alphabetic (:Lowercase :Capital)))
            (:NonAlphanumeric (:EnclosureChar :Punctuation ... etc.))))
```

2. Types of Features (Attributes or Relations)

A feature is defined by its type and methods for matching and generalizing examples:

```
(declare_feature 'TextMatches :printname "Matches" :type TextTokenSequence)
(defmethod match ((f1 TextMatches) (f2 TextMatches))
      (match (tokenSequence f1) (tokenSequence f2))
(defmethod generalize ((f1 TextMatches) (f2 TextMatches))
      (generalize (tokenSequence f1) (tokenSequence f2)))
```

3. Salience of Features Relative to Actions

The salience of a feature value (or pattern) is defined relative to actions and features in which it occurs; salience of a type of feature is defined relative to actions:

```
(declare_salience EnclosureChar :action 'SelectText
      :features `(TextMatches) :score Medium)
```

4. Widening the Focus of Attention When the Current Feature Set Proves Inadequate

The knowledge engineer can specify that features be added or removed when the learner fails to find a consistent or reasonably compact description:

```
(declare_focus 'add_features :when `(special_case_disjunct_created)
      :given_feature TextBegins :add_features `(TextFollows)))
```

Figure 5. Cima's background knowledge—as encoded in Lisp—includes built-in concept patterns, attributes or relations, salience of features relative to actions, and the ability to widen the focus of attention when the current feature set proves inadequate.

number and presses the command key, she wants the agent to copy the phone number to a telephone dialer widget. Figure 6 on the next page shows some sample addresses.

In a typical exercise, the user tells the agent to start watching. Then, while holding down the command key, the user clicks between the "6" and "1" in "243-6166," selects the string "243-6166," copies it, pastes it into the dialer widget, and tells the agent she is done. Cima must learn how to select these phone numbers, which it does not yet know how to parse as concepts. Moreover, Cima must learn to distinguish local numbers from long distance numbers, so that it can strip off the local area codes from local numbers.

Here we describe two scenarios for teaching Cima this task. The first uses examples only; the second uses hints. These two methods correspond roughly with the talkative and quiet user dialog styles that emerged from the Turvy experiment.

Learning by example

To provide Cima with the first example phone number, the user selects "243-6166" with the mouse. When recording this action, Cima notes the selection and its surrounding text. An operational description of this action must specify where the selection starts and ends, and so Cima forms a text-selection rule, shown as item A in Table 1 on page 43. When the user gives a second example, "220-7299," the rule is generalized to item B. Cima correctly predicts the third example, "284-4707."

When Cima predicts "255-6191," a nonlocal number, the user corrects Cima by selecting "(403) 255-6191," implicitly classifying "255-6191" as a negative example. Since no generalization covers all four positive examples while excluding the negative, Cima forms a rule with four special cases, shown in item C. Being forced to create new special-case rules is symptomatic of an inadequate attribute language; Cima therefore widens its focus of attention to include features of the surrounding text. Three of the positive examples follow "617," while the negative example does not. Using this information, Cima forms the two rules shown in item D. The string "617_" is proposed rather than merely "7_," which would discriminate equally well, because Cima is matching text at the word level. It matches at the character level when its focus of attention is directed there.

When the user selects the next positive example, "(415) 457-9138," the second rule in item D is generalized to `MATCHES (Number(length 3)) Number(length 3)-Number(length 4)`. The rules predict the remaining positive examples, except for an anomalous one, "339-8184," which lacks an area code. When the user selects this number, Cima forms the set of rules in item E. We configured Cima

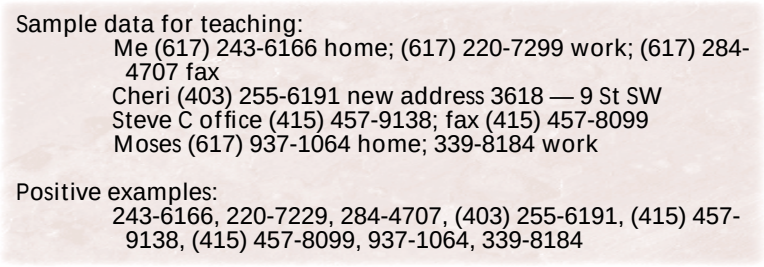

Figure 6. From a list of addresses, Cima can be trained to recognize a phone number, so that whenever the user clicks within a number, the system will select the entire number and dial it.

to maximize the similarity between rules by reusing features, which allows Cima to adopt a generalized pattern for this final phone number, even though it is the only example of the new rule.

Learning by suggestion

Now consider the same concept taught by examples and hints from the user. To point out the local area code, the user selects "(617)" and chooses "look at this" from a pop-up menu. This action directs Cima to focus on text preceding the selected phone number, and to construct a rule incorporating that text. The rule is shown in item F of Table 1. After the second positive example, the rule is generalized as shown in item G. Cima learns the other two rules as before.

Rather than point at "(617)" while selecting the first example, the user could have given a verbal hint, such as "It follows my area code." Looking in its thesaurus of feature words, Cima finds the keyword "follows," which suggests two features, `FOLLOWS` and `PRECEDES`, with preference given to the former. The system parses two lines of the address file around the example and picks out several features: the literal text, its tokenization, and the string `")_"` just before the example. Built-in knowledge (indicating that punctuation and parentheses are salient features) biases the learning algorithm to choose `FOLLOWS ")_"` as the relevant attribute. A second verbal hint, "any numbers," which the user gives while selecting the phone number, causes Cima to generalize `MATCHES`, focusing on tokens of type Number and ignoring other properties such as string value and length.

Thus, after one example and two hints, the system forms the rule shown in item H in Table 1. But this rule predicts a negative example, since the `FOLLOWS` pattern is too general. To eliminate the negative example, Cima specializes the `FOLLOWS` attribute value to `"617)_,"` forming the rule in item I.

A taxonomy of instructions

The various forms of examples and hints can be abstracted to three types of instruction:

- *Classify.* `(Example, {+ve,-ve}, Concept, Subset)`
- *Relevancy.* `(Feature, {relevant, irrelevant}, Concept, Subset)`
- *Consistency.* `(Rule, {correct, overgeneral, overspecific, incorrect}, Concept)`

In practice, users omit or underspecify some of the arguments related to these instructions.

Classify. The first form of instruction classifies an example as positive or negative with respect to some concept, and may indicate that it belongs with some subset of its examples, which is the usual instruction (subset omitted) given to inductive learning programs. The user implicitly classifies example data by selecting or inserting it while demonstrating a task.

Relevancy. The relevancy instruction states that an attribute, such as `FOLLOWS`, or value, such as `FOLLOWS (617)_`, is relevant or irrelevant to some subset of examples. This type of instruction is known to accelerate learning because it reduces the dimensions of the search space.[9] There are a number of different ways to deliver relevancy instructions to Cima: a pop-up menu for "pointing," verbal hints typed or spoken, and more formal partial specifications using terms like `MATCHES`. Typically, because a hint describes the attribute or value ambiguously, the learner must therefore explore several interpretations.

Consistency. The consistency instruction states whether a given rule is valid. Consistency rules have been studied in systems that learn from an informant.[10] Although not illustrated in the previous example scenario, Cima can interpret one form of this instruction, namely when the user classifies the rule as correct or incorrect through a menu command. If incorrect, the rule must contain incorrect or irrelevant features, and so Cima asks the user for relevancy instructions to guide it in generating a new rule.

EVALUATING THE IMPLEMENTATION

We tested Cima on learning the text structures—surnames, publication dates, and other bibliographic information—encountered in the Turvy tasks. For the purpose of understanding user input in the Turvy experiment, Cima coded pointing gestures as selections of text, and it coded spoken hints as text strings. Cima and Turvy use the same attribute-value language, but Cima learns DNF rules only, whereas Turvy can disjoin attribute values and thus learn simpler, more general descriptions.

By searching forward from the start of a paragraph (a feature of both rules), Turvy uses the following logic:

- Selected text `MATCHES CapitalWord or LowercaseWord "_" CapitalWord and PRECEDES ":" or ","` or
- Selected text `MATCHES "Michalski"`

In contrast, Cima learned a somewhat more complex-looking description. Searching forward from the start of a paragraph (a feature of all four rules), Cima uses the following logic:

Table 1. Three scenarios for teaching phone number tasks.

Item	Example	Class	Rule: "Search Backward from Click Location for Text That…"
	Learning from examples only:		
A)	243-6166	+	MATCHES "243-6166"
B)	220-7299	+	MATCHES Number(length 3)—Number(length 4)
C)	255-6191	-	
	(403) 255-6191	+	MATCHES "243-6166" or "220-7299" or "284-4707" or "(403) 255-6191"
D)	After change of focus to include features of neighboring text:		FOLLOWS "617)_" and MATCHES Number(length 3)—Number(length 4) or text that MATCHES "(403) 255-6191"
E)	After the final positive example:		
	339-8184	+	FOLLOWS "617)_" and MATCHES Number(length 3)Number(length 4)or text that MATCHES (Number(length 3)) Number(length 3)—Number(length 4) or text that FOLLOWS ";_" and MATCHES Number(length 3)—Number(length 4)
	Learning from examples and by pointing at "(617)":		
F)	243-6166	+	FOLLOWS "(617)_" and MATCHES "243-6166"
G)	220-7229	+	FOLLOWS "(617)_" and MATCHES Number(length 3)—Number(length 4)
	Learning from examples and through verbal hints:		
H)	243-6166	+	FOLLOWS ")_" and MATCHES Number—Number
I)	255-6191	-	FOLLOWS "617)_" and MATCHES Number—Number

- Selected text MATCHES CapitalWord and PRECEDES ":" or
- Selected text MATCHES CapitalWord and PRECEDES "," or
- Selected text MATCHES "Michalski" or
- Selected text MATCHES LowercaseWord "_" CapitalWord.

Despite such differences, Cima achieved 95 percent of Turvy's predictive accuracy, measured over the course of both learning and performing tasks. In normal use, learning and performance are of course inextricably interleaved. Cima's performance on some tasks fell short of Turvy's, due to the way Cima primitively interprets hints. Cima merely spots keywords; Turvy parses sentences. Even so, we concluded that Cima satisfies our design requirements for the concept learning part of the system, and that the agent's user interface should be designed to ensure that users always associate hints with examples, in order to reduce the set of plausible interpretations that an agent might rely upon.

Capturing a user's intentions is one of the most difficult problems in programming intelligent, user-friendly agents. By paying very close attention to the forms of instruction that people naturally adopt when communicating their intentions, we believe that we can greatly refine the structures computer agents use to interpret intentions. Such a conclusion may seem obvious, but machine learning systems have not completely exploited this observation.

Our experience with both Turvy and Cima demonstrates that even ambiguous hints improve learning from examples, provided the learning system uses other sources of knowledge to interpret the ambiguous hints and measure their credibility.

The potential applications of agents that learn from their users are legion. Most interactive tasks involve a degree of repetition that can become intensely annoying to users. Programming by demonstration offers a general solution to reduce boring, repetitive work. We have explored one domain—repetitive text editing—and believe that the ideas presented here apply to other applications as well, such as spreadsheets, scheduling programs, and personal digital assistants.

Designers of intelligent agents have tended to focus on technology, assuming that any intelligent agent will be easy for humans to deal with. Perhaps the most important lesson we learned in this research, however, is the value of involving users in the design process. By testing and critiquing our design ideas, they keep us focused on our objective: agents that make computer-based work more productive and more enjoyable. ❖

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada and Apple Computer. Many thanks to Craig Nevill-Manning and the anonymous reviewers, whose suggestions helped us improve the presentation of this work.

References

1. D.C. Smith, *Pygmalion: A Creative Programming-Environment*, Birkhäuser Verlag, Basel, Switzerland, 1977.
2. A. Cypher, ed., *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, Mass., 1993.
3. B. Shneiderman, "Beyond Intelligent Machines: Just Do It!," *IEEE Software*, Jan. 1993, pp. 100-103.
4. D. Maulsby, S. Greenberg, and R. Mander, "Prototyping an Intelligent Agent through Wizard of Oz," *Proc. InterCHI*, ACM Press, New York, 1993, pp. 277-285.
5. S.E. Brennan, "Conversation As Direct Manipulation: An Iconoclastic View," in *The Art of Human-Computer Interface Design*, B. Laurel, ed., Addison Wesley, Reading, Mass., 1990, pp. 383-404.
6. R.G. Leiser, "Exploiting Convergence to Improve Natural Language Understanding," *Interacting with Computers*, No. 3, 1989, pp. 284-298.
7. D. Maulsby, *Instructible Agents*, doctoral dissertation, Univ. of Calgary, Dept. of Computer Science, 1994.
8. J. Cendrowska, "PRISM: An Algorithm for Inducing Modular Rules," *Int'l J. Man-Machine Studies* 27, 1987.
9. D. Haussler, "Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework," *Artificial Intelligence*, No. 2, 1988, pp. 177-221.
10. D. Angluin, "Queries and Concept Learning," *Machine Learning* (2), 1988, pp. 319-342.

David Maulsby is an independent consultant and product designer. He is interested in moving programming-by-example technology out of the laboratory and into commercial applications for personal finance and computer-based learning. Maulsby received a PhD in computer science from the University of Calgary. He is coeditor of the book Watch What I Do: Programming by Demonstration *(MIT Press, 1993).*

Ian H. Witten is a professor of computer science at the University of Waikato. He is interested in machine learning, adaptive text compression, and user modeling. Witten is the author of nearly 200 refereed papers on machine learning, speech synthesis, signal processing, text compression, hypertext, and computer typography He has written six books, the latest being Managing Gigabytes: Compressing and Indexing Documents and Images (*Van Nostrand Reinhold, 1994*).

Contact Maulsby at Aurelium Inc., 430, 820-89 Ave. SW, Calgary T2V 4N9, Canada; maulsbyd@aurelium.com. Contact Witten at the Department of Computer Science, University of Waikato, Hamilton, New Zealand; ihw@cs.waikato.ac.nz.