Overview: How Automated Mentoring Supports our Students

Our courses in software development are unique in online learning. While many e-learning programs offer learning support in various forms, our courses provide a level of immediate, robust, responsive mentoring that trains our students to be skilled, versatile problem solvers.

What Online Mentoring is Not: FAQs and Forums

Generally, when distance learning courses talk about online mentoring, they mean that answers to student questions are available from two sources: Frequently Asked Questions (FAQs), and discussion forums. A FAQ is a list of pre-established questions and answers, indexed by keywords, which can either be browsed freely or searched by entering questions into a text field. A forum is an online discussion group where students can post questions and request help from other participants in the course, or sometimes from a broader technical community.

When used to mentor novice students, both FAQs and forums have two major flaws. First, they assume that a beginner can ask the right questions; and second, they assume that the proper goal is to provide answers to those questions. Both assumptions are false.

To understand why, let's consider an example. Susan is enrolled in a standard online class for web application development that uses FAQs and forums for student support. For an assignment, Susan has just written some Java code that should allow a user to join an online multiplayer strategy board game. To test her new code, she starts her web server, opens the browser window, goes to the "Join Game" page, enters a name, and clicks "Join". Nothing happens. Now what?

Susan, like many beginners in complex technical areas such as software development, is not sure how to ask the right questions. Susan tries searching the FAQ using key words such as, "Page doesn't work." Because her question is vague and incomplete, the FAQ returns nothing. She tries and "Java doesn't save data in database." Now the FAQ retrieves a large number of answers, about Java, databases, saving files, and so on. Some are way over her head, and none appear to answer her question.

Still looking for help, Susan posts her questions to a peer forum. Of the few participants who respond, most ask for more information, and one even criticizes her for not knowing what she's talking about. Finally, a more experienced programmer says, "Please post your code." Susan does, and the expert says, "You forgot to put @Entity at the start of your class definition." Susan tries adding this entry to her code, and everything works. Problem solved! Or was it?

The problem with both FAQs and forums is that they are designed to *answer* a beginner's posted question. For novice learners like Susan, this is the wrong goal. Susan's code was fixed, but her skills were not improved. At best, she learned one technical fact about Java libraries

that will be useful for a few years, until the libraries change. Susan learned nothing about what's going on underneath, how to diagnose such mistakes in the future, or what alternative fixes could be done. That's the knowledge that would be useful in her career for years to come.

Meet AutoMentor

Let's see what our AutoMentor does for our student, Susan. As above, she has written some HTML and Java code that is supposed to allow a user to join an online multiplayer strategy board game, but when she tests her "Join Game" page, nothing happens.

At a loss for what to next, Susan decides it's time to get help. She goes to the course web page that specified the programming task she needed to do. She clicks the "I need help" button that's there.

> A page with an avatar for the AutoMentor appears. "Tell me more," the mentor says. A form appears, with Susan's name, course, and current task assignment listed.
>
> "What's the problem?" the AutoMentor asks, presenting a text input field.
>
> Susan types "My 'join game' page isn't working. I see the form and I enter a name, but when I click Submit, the form just reloads and nothing happens!"
>
> "Any code or output you can show me?" the AutoMentor responds, presenting fields for entering a labeled block of text.
>
> Susan pastes in her HTML code.
>
> "Anything else?" the AutoMentor asks.
>
> Susan decides to also paste in her Java code for the Player class.
>
> "Anything else?" Susan thinks not, so she clicks "I'm done."
>
> A short pause, and then a video of a human mentor appears, saying, in part,
>
> > "I can't tell for sure, but it may be your code isn't compiling. That's the first thing you should check, any time a program doesn't seem to work. Look through everything the compiler printed. If you find even one error message, fix it, and try again. If you have an error and your fixed code doesn't work, send me that code and the output you got. If you're not seeing any errors in the compiler output, send me the compiler output so I can check too."

On the page, along with this video, are hyperlinks to readings on how compilers work, how to read Java compiler output, and so on. There are buttons to end the dialog, and one that says “I have more to add.”

Susan goes back to her development tool. At the end of the compiler output, she sees an error message, but it makes no sense to her. She doesn’t see how it could apply to her program.

Susan returns to the AutoMentor page, and clicks “I have more to add.”

She adds the compiler output into her previous request. Then she selects the line with the last error message and attaches the note “I have no idea what this means.” She clicks “I’m done” again.

A new mentor video appears.

> “I see you have several error messages, but you’re looking at one of the later ones. I always ignore those. Often the errors at the end are caused by errors in earlier code. I look at just the first error and fix that. Then I see how many errors I still get when I try again.”

Susan returns to her development tool. Scrolling all the way back in the compiler output, she finds the first error message. It’s a really basic mistake on the third line of her code. She can see how to fix that. She goes to work.

Intelligent Human-like Automated Mentoring

We are developing an online automated mentoring system (AutoMentor) that mirrors the approach of expert human mentors. Like human mentoring, the interaction is more like a conversation than a single question and answer. Like experienced human mentors, the AutoMentor understands not only what a student’s problem is, but what to say to help the student learn to solve their problem on their own.

The AutoMentor is modeled on expert human reasoning:

- An expert has a large *case library* of answers, learned from experience.
- Cases are labeled by features to look for in future situation that suggest this answer is relevant.
- When those features are seen in a new situation, the case is retrieved and applied to the new situation.

This is called *case-based reasoning*. It is a model of human reasoning that we developed that has become a standard part of human-like artificial intelligence.

The AutoMentor for a course, such as a software development course, has a case library of hundreds of responses to give a student. Responses are indexed by features of both the problem and the student, e.g., a response might be indexed by “JSON data not transferring” and “for beginner unfamiliar with JSON.”

The AutoMentor uses the case library and indexing to the responses relevant to a student’s problem, and appropriate for the student to hear.

Diagnosing a Student’s Problem

When a student says, "My program doesn’t work," a skilled human mentor responds, "OK, *show me* what you did." A good mentor knows that what a student *says* is unreliable for diagnosing the problem. Instead, the mentor asks to see what the student actually did.

In our example, when Susan clicked the "I need help" button, our AutoMentor helped her gather the data about who she is, what course and task she is working on, what code she is trying to run, and what output she sees. This data is what the AutoMentor uses to find cases with similar problems.

Index-based Reminding

Expert human mentors have learned to automatically look for specific features when students have problems. For example, they may have learned that a certain error message is very common on a particular task because there’s a simple mistake students frequently make. The task and error message are indices for retrieving advice on how to help students find that particular mistake.

We build our AutoMentor to do the same thing, by creating indexing rules that the AutoMentor runs to scan for different features in a request for help. Here are just a few examples of such indexing rules:

- Use a course information database to extract student goals from the course and task information attached to the request.
    - For example, Task 4 in the Java course might have the goal “pass JSON to web server.” Passing JSON is a feature. There are mentoring responses indexed by that feature that explain common reasons why that goal sometimes fails.
- Use text patterns to extract error messages, if any, from the output that the student sent.
    - For example, the Java error “method not found” is an index for a number of responses explaining different reasons this can happen.
- Use text patterns to extract common mistakes, if any, from the student’s code.
    - For example, the mistake “object changed but not saved” can be recognized by looking for code that changes a class but doesn’t call the **save()** method.

In our example with Susan and the AutoMentor, she originally submitted just her HTML and Java code.

- By looking up which task she is doing in a course/task database, the AutoMentor extracted the indices "new Java code written" and "sending form data to server."
- No compiler output was included, so no indices for error messages are generated.
- The HTML and Player code are scanned for common novice mistakes on this task, e.g., no @Entity annotation, but none are found, so no indices for mistakes are generated.

Using the few indices identified, the AutoMentor retrieves two sets of relevant responses.

- The feature "new Java code written" is an index for responses about code not working because the new code is incorrect. In Java such mistakes will often cause compiler error message to be printed. Among the cases retrieved are general responses about what compiling and compiler errors, and more specific tips on interpreting messages correctly.
- The feature "sending form data to server" is an index for responses on problems with form submissions. There are general responses on form submission and how to see what's being sent, and more specific responses are particular types of errors.

When Susan added the compiler output to her request, with a note attached to the last error message, the AutoMentor rules generate an additional index, "not focusing on first error." This index exists because there's a response about why errors should be fixed in the order they appear.

Choosing the Appropriate Response

The responses retrieved based on the student's code and output are *relevant* to the problem. The next step is to decide which responses are *appropriate* for this student.

Expert human mentors know that what to say depends on who you are talking to. Don't overwhelm beginners with details or advanced concepts. Give them general tutorial responses. But it's also the case you should not bore students who are farther along with tutorials. Give them short, specifically targeted responses. If they are really advanced, challenge them with alternatives things to consider. In other words, two students who present *the exact same problem* may get very different answers. This is what makes quality mentoring different from the one-size-fits-all answers provided by FAQs, and the hit-and-miss answers often provided by forums.

To determine what the appropriate response is, the AutoMentor needs some idea of what each response is about, and some idea of what the student is already knows.

So that the AutoMentor know what responses are about, we label each response in the case library with the concepts it *introduces*, and the concepts it *presumes*. For example,

- A response for beginners in Java explaining why a “method not found” error is occurring, might *introduce* the concepts of "method resolution" and "type matching."
- A response for more experienced students for the same error, might *presume* these two concepts and *introduce* the more advanced concept of "type promotion".

By concepts, we mean canonical unambiguous symbols, organized into an ontology in a semantic memory. A serious semantic memory has thousands of concepts, but we only need the much smaller set of technical concepts that responses either introduce or presume.

When a set of responses has been retrieved for a student’s problem, the AutoMentor needs to pick a response that does not presume any concepts that the student doesn’t know, but, as much as possible, doesn’t spend time introducing concepts the student does know.

To know what a student knows, the AutoMentor has to look at what the student said and didn’t say.

Let's look at what the student didn't say first. For example,

- If a student wrote some new Java code, but did not include any compiler output, or the student did submit compiler output, with error messages, but no notes attached to the lines with those messages, assume they do not understand how to read and use compiler output.
- If the student did attach a note to an error message, but not to the line of code that the error points to, assume they do not know how to connect error messages to specific code.

In short, failure to comment in any way on an issue can be used to infer lack of knowledge of the underlying concepts for that issue. A few simple rules can be used to infer some basic conceptual gaps such as the above.

In order to infer what a student does know, based on what they said, the AutoMentor has to understand what the student wrote. On the face of it, this might seem impossible. Understanding well written natural language that is semantically correct is still unsolved challenge for Artificial Intelligence. How can an automated mentor possibly understand student text that is fragmentary, usually ungrammatical, using vague and incorrect terms?

Memory-based Natural Language Understanding

Our answer is to use a model of language understanding that we pioneered called *Direct Memory Access Parsing* (DMAP). DMAP recognizes references to concepts without depending on grammar. This makes DMAP ideal for student texts, which are frequently grammatically challenged.

In DMAP, words and phrasal patterns are attached to the concepts of interest in the domain, e.g., "programming language" to the concept PROGRAMMING_LANGUAGE and "Java" to the concept JAVA_PROGRAMMING_LANGUAGE. There can be any number of phrases attached to the same concept, e.g., "code", "program" and "file" for the SOURCE_CODE object. Likewise, the same phrase can be attached to any number of concepts, e.g., "program" can refer to source code, compiled code, or the action of writing code.

The AutoMentor uses DMAP to scan the notes the student has written in the request for help. When DMAP sees phrases that are attached to a concept in semantic memory, DMAP collects the concept. If DMAP sees two concepts that are linked in memory, it collects the link. For example, when scanning "and then I recompiled my code," it collects the concepts for the "compile" action and for the "source code" object, and also the "object of action" link that connects those two concepts in memory.

DMAP is the perfect tool for the AutoMentor because it can identify references to concepts from phrases, without needing grammatically well-formed English.

Having inferred the concepts that the student is aware of, based on what she said, and the concepts she doesn't know, based on what she didn't say, the AutoMentor can now select the most appropriate response to help the student.

Choosing the Appropriate Response Example

In our example with Susan and the AutoMentor, the AutoMentor has retrieved relevant responses about the possible error. There are responses about compiler errors and responses about form submission errors.

Now the AutoMentor has to decide which response is most appropriate to give Susan.

- One response introduces the concept of compilation and verifying the absence of compiler errors. The others presume the student is aware of this concept.
- Similarly, there is one response that introduces the concept of form submission and how to detect form submission errors. The other presume that concept.

The AutoMentor look at what Susan has sent and written. Because she did not include her compiler output, it assumes she is unaware of the concept of checking for compilation errors. Similarly, because of what she didn't say, the AutoMentor assumes she is unaware of how to verify the absence of form submission errors. This means the tutorial responses that introduce the basic concepts of compilation and form submission are appropriate for Susan. A domain-specific tie-breaker rule says that compiler problems precede runtime problems and should be addressed first.

When Susan added the compiler output with error messages, the AutoMentor retrieved an additional response about focusing on the first error message first. This response presumes awareness of compiler errors and introduces the concept of taking them in sequence. The

AutoMentor infers Susan is now aware of compiler errors because (1) she included the compiler output, and (2) she has been shown the video on compiler errors. Therefore, the AutoMentor considers the new response is appropriate and shows it to Susan.