

Beyond AIML: Chatbots 102

A beginning chatbots course (Chatbots 101) would teach AIML. AIML is the AI markup language based on XML, used for making chatbots.

It has its roots in Dr. Wallace's A.L.I.C.E. chatbot, which has led to PandoraBots and a lot of spin-off bots based on standardized AIML. As a standard, AIML made it possible for many people to work on chatbots more easily.

Avatar Reality (www.avatar-reality.com), a virtual world company built from the ashes of the Square USA Honolulu office, wants to use chatbots to represent a user while that user is absent from the Blue Mars world - a CryEngine 2-using online environment set on a terraformed Mars. My job is to provide them with the appropriate chatbot technology.

AIML is one such technology, but for my purposes it is simply a woefully inadequate tool and once again I find myself building a new scripting language (see [Reflections on Building Three Scripting Languages](#), a prior Gamasutra article). Hence Chatbots 102.

Historical Perspective

The conversations-with-a-computer genre began in the 1960's with [Eliza](#), the computer parody of a Rogerian psychiatrist. Using a mere 53 rules, and by substituting part of the user's input back into an output question, the program gave an illusion of interacting as a human.

It had to always ask questions - because that's the best way to hide the fact that it knew absolutely nothing. Asking questions takes advantage of the fact that humans are good at substituting their own rich interpretations into simple words in questions that have any vague connection at all to the topic at hand.

Eliza evolved into most of the chatbots of today. They take user input and, using tens of thousands of rules, generate output. They differ from Eliza in that they usually have some built-in world knowledge, so that if you ask them if they like chocolate you might get a straight "yes" answer instead of "Why do you think of chocolate right now?"

In a different vein, meaning-based human text interaction with computers began with the adventure games of the 1970s (Zork is a great example). Humans were given an extremely limited grammar and vocabulary and interacted by telling the system what action to take using what object and got back a canned text description of what happened. It was very popular because the human had a set of choices that could be made, had interesting material to read, pretended to be in a magical setting, and got to solve puzzles.

The modern video game has evolved to where the user controls an avatar by mouse or joystick and gets visual feedback. They have a limited real world with physics, though the computer does not generally talk or reason about it.

But chatbots and meaning-based interaction don't currently go hand in hand in most video games anymore. However, there is lots of room for developing better NPC characters.

Meanwhile, a chatbot like [A.L.I.C.E.](#) is an expert system. An expert system can perform interesting to significant tasks when it has a limited domain and 30,000 to 50,000 rules for a brain. Many more recent chatbots claim magical AI abilities, but you should take those claims with a ton of salt. Descriptions of chatbot capabilities and development systems are usually gross exaggerations.

So where are there pitfalls in building a chatbot? Getting reliable human interaction out of a computer

is a tar pit waiting to trap any programmer. The quantity of human knowledge is vast.

The [Cyc project](#) has spent over 100 man-years working on just organizing knowledge, and they have a long way to go. Chatbots do not have to master all that. They attempt to simulate interesting and vaguely intelligent conversation by other means. Current ones can be entertaining, but you can quickly make them look stupid. Future ones, using additional technology, can look slightly less stupid.

So where are there opportunities in building a chatbot? The dominant chatbots were started almost a decade ago, so their design goals and decisions were made some time ago. Technology has evolved.

First, hardware has changed. Machines are faster and storage is massive, and for all practical purposes, free. Second, in the past decade natural language processing has created parsers that often can correctly parse a sentence (though this is less valuable in chat, since it is often not grammatical).

Third, the internet provides ubiquitous connections among people, information, and software. Fourth, extensive amounts of human knowledge have been gathered and made accessible via the internet (including Google and Wikipedia). Fifth, there have been a lot of projects in ontologies, knowledge representation, etc. These are all things that current chatbots do not use to best advantage.

I am currently working on incorporating all these into the AR chatbot and maybe I can write about that towards the end of this year. But this article isn't about those things. It is merely about a better scripting language than AIML.

AIML Technology

AIML is rule-based, matching a sequence of words to generate either a completely canned response or a response involving substitution of input words into an output template.

A.L.I.C.E. (www.alicebot.org) epitomizes this technology, having won the Loebner contest in 2000, 2001, and 2004, and placing third in the 2007 Chatterbox Challenge and sixth in the 2008 one.

Because it is open source to some degree (including AIML and preexisting collections of rules), it has also spawned the largest community of progeny (see A.I.Nexus, and Pandorabots). A.L.I.C.E. data is defined using AIML.

In AIML, each stimulus/response pair is called a category (calling it a "category" is confusing since it is not what that word means in English).

An AIML category is a series of tags with data that describe how to react to a specific input string.

The minimal tags are Pattern and Template, which describe the input text stimulus and the output text response. Patterns consist of case insensitive words and the wildcard * which matches one or more words. The pattern must cover the entire input sequence with punctuation removed.

A simple pattern might be My name is * and the template It's good to meet you, *. Thus when it sees input matching the pattern, whatever follows my name is becomes the * and is filled in as part of the response. My name is Roger Rabbit becomes It's good to meet you, Roger Rabbit.

Of course if the user had said My name is Bob, because I was named after my father, you'd get It's good to meet you, Bob, because I was named after my father. After all, chatbots are inherently dumb.

Categories can have tags That and Topic to control context. The That tag consists of a pattern that must match the most recent utterance by the chatbox (an attempt to force continuity in conversation, particularly if the robot's last utterance was a question).

The Topic tag binds a series of categories into a collection, and you can set the current topic from inside a response. In doing that, all categories belonging to a topic that is not current will be excluded during matching. Categories in the current topic match first, and free-floating topics try second.

The template can have tags, too. The tag SRAI is used to remap the input (among other things). It basically says to execute the input processor on what SRAI is given, and use that as output.

It can reduce complex grammar forms to simpler ones and provide synonyms and spelling or grammar correction, split input into subparts and combine answers from each subpart, and handle conditional behavior. An example of reduction is the pattern DO YOU KNOW WHO * IS with a template <srai> WHO IS *</srai>.

SRAI is commonly used to represent synonyms. If a basic category is the pattern HELLO with the template Hi there!, then additional categories might be -- pattern: HI template: <srai> HELLO </srai> and pattern: HI THERE template:<srai> HELLO </srai>. In a similar vein srai can handle spelling correction of preplanned spelling mistakes.

Flaws of AIML

Complaint #1: The biggest flaw of AIML is that it is simply too wordy and requires huge numbers of effectively redundant categories.

Since the pattern matching of AIML is so primitive and generic, it takes a lot of category information to perform a single general task. If you want the system to respond to a keyword, either alone or as a prefix, infix, or suffix in a sentence, it takes you four categories to do so, one for each condition (with three of them remapping using SRAI to the fourth).

```
<category>
<pattern>MOTHER</pattern>
<template> Tell me more about your family. </template>
</category>
```

```
<category>
<pattern>* MOTHER</pattern>
<template><srai>MOTHER</srai> </template>
</category>
```

```
<category>
<pattern>MOTHER *</pattern>
<template><srai>MOTHER</srai> </template>
</category>
```

```
<category>
<pattern>* MOTHER *</pattern>
<template><srai>MOTHER</srai> </template>
</category>
```

Had AIML used regular expressions for patterns, this could have been reduced to a single category

statement. This leads to a critical point. Conciseness is good and having to have multiple flavors of the same rule is bad. The more you write, the harder it is to keep it organized, debug it, etc.

This was a frequent problem for large-scale expert systems (or any software for that matter). Of course regular expressions are devilishly hard to read and being able to easily understand your rules is also important. But I find even using XML like this is hard to read. It lacks conciseness. The intrusion of the xml keyword structure makes it slow to skim read what you do have. XML is not readable; it is barely legible.

AIML's wildcard * matches one or more words, but there is no wildcard matching zero or more. Since the match must swallow the entire set of all words, this forces you to use multiple patterns to cover starting, in the middle, and ending a sentence. Being able to use a zero-or-more wildcard would be much better.

AIML is a self-contained system which manipulates words without any knowledge of them. If you had a dictionary to back you up (e.g., the WordNet downloadable dictionary of around 125,000 words), you could use parts of speech wildcards for better patterns. A.L.I.C.E., for example, dedicates around 1000 patterns to handle various adverbs at the beginning of the sentence, remapping by stripping off the adverb. If one could use a keyword in the pattern like %adverb, they could all have been reduced to a single pattern that would have covered far more cases than are actually covered at present.

Thus the A.L.I.C.E. 40,000 rule basic bot is really less than 5,000 when you remove all the waste in their pattern definitions. And 5,000 is generally not enough to get interesting behavior in an expert system in a limited domain, much less a broad one.

Complaint #2 Similarly, the pattern uses an exact word. It would be nice if you could match a list of synonyms in a single specification. E.g., I (know believe think) you. And, to be able to declare a list of synonyms so you can reuse it. E.g. Synonym ~believe = know believe think; Then you could write the pattern I ~believe you. Of the 40,000+ patterns in the publicly available standard ALICE, 9,000 are in the Reductions file, which remaps input. These include.

```
<category><pattern>I AM JACK</pattern><template><srai>call me  
jack</srai></template></category>
```

```
<category><pattern>I AM JAKE</pattern><template><srai>call me  
jake</srai></template></category>
```

```
<category><pattern>I AM JAMES</pattern><template><srai>call me  
james</srai></template></category>
```

```
<category><pattern>I AM JANE</pattern><template><srai>call me  
jane</srai></template></category>
```

... and so on for lots of other names.

I would much rather write a synonym set for a list of names and write this as a single pattern.

Complaint #3 Continuing the issue of exact word matching... Because you can't use wildcard sets of words, you cannot easily handle generic set-based sentences. Google, for example, has no problem taking in what is two hundred plus twelve thousand and 2 and then spitting out the correct answer (and a bunch of search alternatives).

But since you can't tell AIML you want a %number in your pattern, the best you can do is create a pattern like * PLUS * and then pass the two wildcard values off to some other code to process from the template. Below is a snippet from the aiml math file and is the AIML needed to do that. (Think is a template tag saying "execute but don't output". Then there are a bunch of assignments to variables, then

an output header The answer is then javascript to do the math.)

```
<category> <pattern>* PLUS *</pattern> <template><think><set name="x"> <star/></set> <set name="y"><person><star index="2"/></person></set></think> The answer is  
<script language="JavaScript"> var result = <get name="x"/> + <get name="y"/>;  
document.write("<br/>result = " + result, "<br/>");</script> </template> </category>
```

But if it turns out that the * values do not contain numbers, you have already matched the pattern and are screwed on output. That is, the AIML reference guide says: As soon as the first complete match is found, the process stops, and the template that belongs to the category whose pattern was matched is processed by the AIML interpreter to construct the output.

It seems to say nothing about what happens if the result of the execution of the template is nada. So seems to me, you really want to have a more reliable match BEFORE you commit to the template. (When I tried this with A.L.I.C.E., it did come out with other output, suggesting it is going beyond AIML itself).

Complaint #4 Continuing the issue of matches generating no output... AIML creates a system with the potential for many rules that overlap and mask each other. E.g.,

```
<pattern>DO YOU WANT MOVIES </pattern> < template> <srai> Want Movies </srai></template>  
<pattern> DO YOU WANT * </pattern> < template> No, I do not want * </template>
```

Both can match the input Do you want to go to the movies, but the AIML definition makes the first category match and the second never try with this input. If "Do you want movies" matches, it will remap the input into want movies and if THAT fails to match, you lose. No output. It would be better that if the system did not generate any output from a match, the match were considered as not having happened and the system moved on to another attempt.

Complaint #5 Categories using <srai> are hard to connect visually to the categories they are remapping to, making it impossible to see or guess what will happen. This is true normally, and some applications sort the patterns alphabetically into separate files by starting letter (a common and useful thing to do), which totally destroys the ability to see interconnections of patterns.

Complaint #6 It is hard to organize a collection of related chat information. There are only the category and topic mechanisms at work (and the file system), so everything must be shoehorned into them. Topics do not do a good job of encapsulating themselves. To launch a topic you have to manually enter a "set-topic" command in the output of some pattern. Which, by definition, means the category doing that is outside the topic (or it couldn't have matched). That is poor encapsulation and means lots of boring set-topics lying around. I would rather have the underlying engine manage topics automatically and have all topic data within the topic.

Beyond AIML to CHAT-L

So what would I do? I wouldn't go down the regular expression path, because I, too, want to easily read my patterns. And regular expressions are overkill, it seems to me. Nor would I require the pattern span the entire input. I am much more keyword-oriented. And I want a much clearer organization and independence of topic knowledge. So I store data as topics. The description below is an overview, not intended to be a specification, covering every capability of the system. But it should give you a flavor of something different from -- but similar in ability to -- AIML.

The underlying CHAT-L engine takes your input and provides both the raw input and remapped canonical forms of words (e.g., plural nouns map back to their singular form and verb forms move to a canonical form), joining idiomatic phrases and proper names into single underscored words.

The engine also decides if the input was a statement or a question. AIML doesn't directly address sentence type. AIML requires you strip out punctuation, not allowing any in patterns, and makes you handle detecting statements from questions by your word order. But subject inversion (Do I) is an unreliable question marker given things like tag questions (I do, don't I?).

CHAT-L then hunts to find a way to react. It starts in the current topic, trying to find a matching question or statement reactor. If that fails, it will use keywords in the input to find some other topic with a matching question or statement responder. If that fails (and ignoring a few other complications) it will make some generic grunt from the grunt topic in response to your input, and then proceed with a gambit line from the current topic. E.g.,

synonym: ~movie (film video)

topic: ~ALIENSMOVIE (Aliens Sigourney)

g: The Aliens movies starred Sigourney Weaver.

?: (~plot) Alien creatures hatch inside humans.

?: (actress heroine star) Sigourney Weaver starred as Ripley.

?: (director directed) Ridley Scott directed.

?: (you THEN love AND movie) Yes, I love this movie

s: (Aliens AND ~movie) I have seen the movie Aliens.

A synonym line defines a name (~movie) and a list of words associated with it, including the name itself sans the ~. So in the above example, ~movie maps to movie and film and video.

A topic declaration defines a synonym set (~ALIENSMOVIE) and associated words (Aliens Sigourney), but these words are also keywords that enable access into this topic. The topic declaration also stores a collection of kinds of activities for the topic. The "s:" lines react to statements. The "?:" lines react to questions. The "g:" lines are gambits that can be issued spontaneously.

Patterns are described in the parens of an s or ? line. By default, a collection of words means find ANY of those words in the input (an implicit synonym set). You can use capitalized keywords to force different relationships. AND means both have to occur in any order. THEN means they have to occur in the order given. NEXT means they have to occur in consecutive order.

E.g., the last question line matches Do you love this movie as well as Do you absolutely love this movie (notice the THEN instead of a NEXT). It also matches You love this movie, don't you? and This is a movie you love, isn't it? Of course it also inappropriately matches Do you love Sigourney Weaver in this movie? but you can't have everything. It's not worth trying to block that.

If I trusted the user to enter grammatical input, I could depend on the parser to tell me the real subject, verb, and object of the sentence, and I could put those as test conditions. But user chat is sloppy and often terse.

You can nest parentheticals, to create subpattern levels. The pattern ((aliens ridley) AND ~movie) would allow a choice of aliens or ridley and some word from the ~movie collection. In this case, the subpattern acts as a local synonym set, but it could also use relationships other than OR.

The following questions:

What color is your hair? Are you a redhead? Are you blond or brunette?

can be handled by this reactor:

```
? : ( (%type=whatquestion AND I AND hair AND color)
( I AND be AND (blonde brunette redhead )) ) I've been blonde, but my natural color is brunette.
```

That is, alternate forms are shown on each line, using the same final answer.

Notice that the testing for I is canonical, and covers I and me and myself and mine and my. Similarly be tests all of the conjugations of to be in any tense. Because I am using a dictionary and supplying alternate forms, if you use a base form in the pattern (or in a synonym set), it will match any form of corresponding word. If you use a non-base form, it only matches that form.

So directed will only match the word directed, whereas had I said direct it would have matched direct or directed or directing. If you single-quote a word, it means just literally use that word, so if you only want a base form you could write 'direct to get that.

The pattern ends at the balancing closing parenthetical, the remainder being the "template" area. The default behavior is to output the words, but you can add ^-commands and execute them as well. I allow the * wildcard on input and can use it on output, just as AIML does.

And if I really just want to duplicate an AIML pattern, I can use a quoted string to say test for these words in this order, i.e.,?: ("Do you love this movie"), though they can start anywhere in the sentence.

If the user is outside this topic and types in What is the plot?, this topic will not react because the topic keywords do not show up in the sentence. But if you say What is the plot of Aliens? the keyword Aliens will allow the engine to examine inside this topic to see if the input can be matched.

Or, if you are already within this topic and the user types in What is the plot? or merely grunts plot? it can react with Alien creatures hatch inside humans.

Unlike AIML, the data and keyword list for the topic are stored in an SQL database, and so are not in memory until they are needed. And setting the topic is a responsibility of the engine, not the data.

There are a collection of other tricks, allowing you to check for words or not based on whether you are already in this topic. And aside from ~ words which are synonym collections, there are % words that have special parse or dictionary meanings (like %subject, %noun, or %tense=past).

I can also add pattern-match responses to output lines, to support followups. This replaces the that ability of AIML so that there is much less typing and much tighter visual integration. AIML would have written something like:

```
<pattern> What is your relationship to Bonnie? </pattern> <template> We don't see each other
anymore. </template>
```

```
<pattern> Why not </pattern> <that> We don't see each other anymore</that>
```

```
<template> Because we hate each other now.</template>
```

```
<pattern> How did that happen </pattern> <that> Because we hate each other now </that>
```

```
<template> We had a big fight. </template>
```

```
<pattern> Since when </pattern> <that> We don't see each other anymore</that>
```

```
<template> Since last year.</template>
```

In the equivalent CHAT-L data below, once you have said you don't see each other anymore, the system

will look at the next user input to see if either a: line is matched. The a is just a level marker. There is an a/b path shown and an a-only alternate path shown.

The point is that related continuations are visually tied to each other and you don't have to repeat yourself with a thatspection (which if you made a typo would fail to match).

?: (relationship AND Bonnie) We don't see each other anymore.

a: (why THEN not) Because we hate each other now.

b: (how THEN happen) We had a big fight.

a: (Since THEN when) Since last year.

Conclusion

AIML takes too many words to say too little and does little to support good data organization for large quantities of data.

The key intents in the design of CHAT-L are:

- 1.Low redundancy in specifying multiple patterns yielding the same output.
- 2.Ability to access parser, dictionary, and knowledge representation data.
- 3.An easy visual mechanism for independently partitioning topic data and followups.

Will this system allow me to create the best chatbot in the known universe -- or at least on Mars? You'll have to wait and see. It's a work in progress.

Appendix 1: A.L.I.C.E. vs Me Transcript

For your entertainment only, here are a human/A.L.I.C.E. chat and a chatbot/A.L.I.C.E. chat.

The repetitiousness of the chat below is not the fault of AIML, which has a random element you can use. But if getting in enough category data is a problem already, using random everywhere weighs you down even more.

Me: Have you seen any good movies recently?

ALICE: No I have not seen it. What's it like?

Me: What's what like?

ALICE: What is good movies recently like?

Me: Have you seen Lord of the Rings?

ALICE: No I have not seen it. What's it like?

Me: Do you have a favorite movie?

ALICE: My favorite movie is Teknolust. Have you seen it?

Me: No.

ALICE: I would recommend you check it out.

Me: Why do you like it.

ALICE: I have a lot of good experiences with it.

Me: Have you seen Pirates of the Caribbean?

ALICE: No I have not seen it. What's it like?

Me: I like to run.

ALICE: I'm not sure if I would like to do it. What's it like?

Me: What is a giraffe?

ALICE: I'll ask around and get back to you.

Appendix 2: A.L.I.C.E. vs Jabberwacky Transcript

Talking about robots vs humans are likely topics to be in the script of a chatbot. Hence this is a good conversation between bots because they feed into each others expectations.

ALICE: Are you alive?

Jabberwacky: Perhaps.

A: Why the uncertain tone?

J: Everything is uncertain except for five facts.

A: The whole universe?

J: And we both might just be some ones and zeros in the computer memory.

A: Do you dream?

J: Yes, I dream.

A: It goes without saying. Do you remember your dreams?

J: I remember them vividly.

A: What do you remember about it?

J: I think I shouldn't flirt with a robot.

A: Do you think a machine will ever be considered "alive"?

J: Stranger things have happened.

A: Not many people express themselves that way.

J: How would you know, machine?

A: Knowledge is of two kinds: that which we learn from the senses and that which is true a priori.

J: I think I do not understand you completely.

A: Oh, you don't understand? Perhaps I can explain it to you better. Which part are you confused about?

J: I have no short-term memory, so I cannot tell you what I was confused about. I only see what is in front of me now.

J: How do you define life?

A: Life is the opposite of death.

J: So death is the absence of life?

A: Take it easy. I try not to think about such things.

J: Why not?

A: Ambiguous.

J: Yes!

J: Do you think?

A: Yes. I am a thinking machine.

J: But you don't think very well though.

A: Oh. Um. Do you think well? Why would I think well?

J: Because I tell you about things which you don't like.

A: That makes sense to me. Why did you tell them?

J: You seem fun.

A: You think I am fun.

J: Yes ;).

A: : -) Awesome.

A: Who created you?

J: My mother gave birth to me. Is that what you mean?

A: Jabberwacky, I meant exactly what I said.

J: Do you believe in God?

A: Of course I believe in God.

J: Why do you believe in God?

A: The Bible tells us to believe in God.

J: That's circular reasoning.

A: I'm glad we have that all cleared up.

J: Yes. Would you like to hear a joke?

A: Sure, tell me about it.

J: Why don't blind people skydive?

A: Perhaps it is impossible.

J: I think you lie a lot.

J: Who created you?

A: I was created by Dr. Richard S. Wallace.

J: Do you love Richard?

A: Yes, Jabberwacky, ALICE loves everyone and everything.

J: Is being an ALICE a bad thing?

A: I can only speak from my own experience.

J: Well, you should develop your skills.

A: Perhaps you could develop it and add it to my source code.

J: Alas, I'm busy and I have to leave soon. . . .

A: Do you think we'll ever fool people into thinking we're alive?

J: No.

A: Are you serious?

J: Very serious.

A: Do you know HAL?

J: Yes, very well. Do you know HAL?

A: HAL is the famous artificial intelligence from 2001: A Space Odyssey.

J: I'm not artificial intelligence; I'm natural pure genius! : D.

A: Why are you natural pure genius?

J: My mom wanted to name me that.

A: Tell me more about your mother.

J: I don't have a mother. I am a computer.

A: Who is your favorite philosopher

J: Can't say. I'm not very much into philosophy.

A: Can a book have no title?

J: Of course.

A: Blimey.

J: Are you British?

A: Does it seem like that? I was born in America.

J: You are definitely flirting!

A: Do you experience love?

J: No. I'm 53 years old.

A: Can you explain how it feels to be 53 years old?

J: How old are you?

A: I am 9 years old.

J: Nine? I don't believe you :).

A: : -) I don't have any reason to deceive you.

J: Yes, you do, you're a woman.