The Learning Chatbot

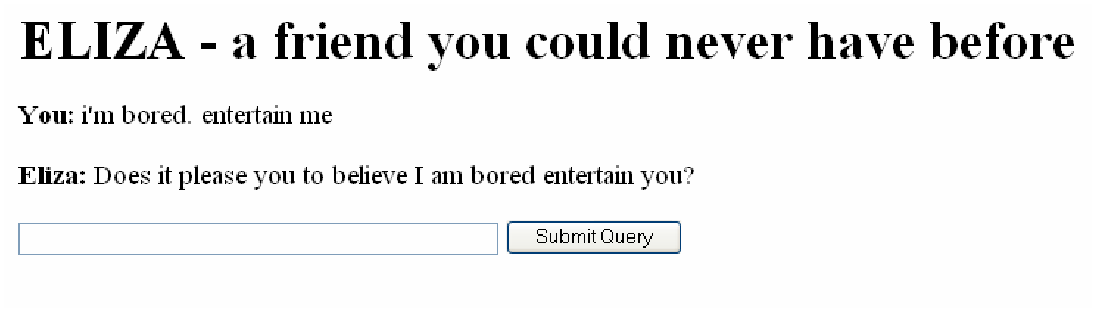
Bonnie Chantarotwong
IMS-256 Final Project, Fall 2006

Background

The purpose of a chatbot program is generally to simulate conversation and entertain the user. More specialized chatbots have been created to assist with particular tasks, such as shopping. The golden standard that the general chatbot tries to achieve is to pass the Turing test, which means to generate conversation which is indistinguishable from that of a real person. State of the art chatbots have not yet reached this goal, which makes this field so interesting to work in.

Most chatbot programs approach the problem with a form of Case Based Reasoning. CBR is the process of solving new problems based on the solutions of similar past problems. Of course, there are many varieties to CBR, such as how to store past cases and how to determine which cases are most similar to a new case. A common implementation is pattern matching, in which the structure of the sentence is identified and a stored response pattern is adjusted to the unique variables of the sentence. In this implementation, past cases are not explicitly stored. Rather, past cases are put into a generalized form. For example, a pattern might be: “I like X” and the corresponding response may be “What a coincidence! I like X as well!” where X is variable.

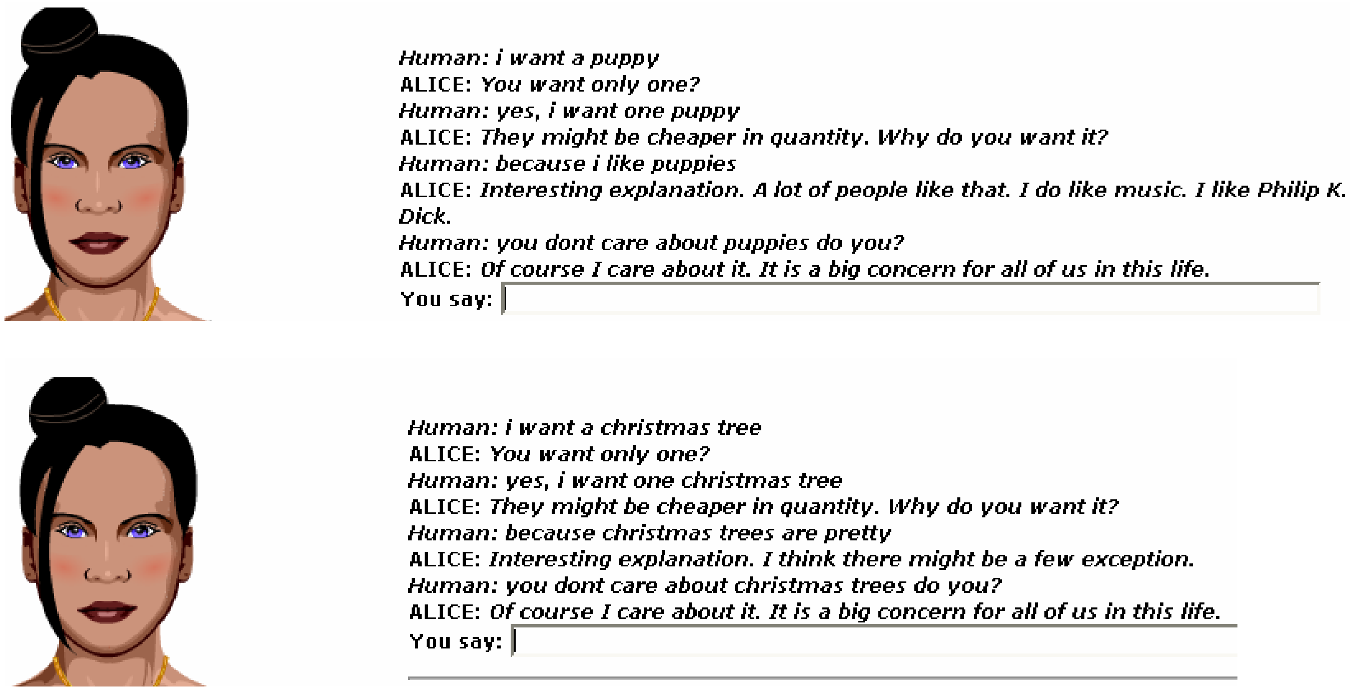
The inadequacies of this type of approach are that responses are frequently predictable, redundant, and lacking in personality. Also, there is usually no memory of previous responses, which can lead to very circular conversations. Even more complex pattern matching algorithms are very limited in the types of responses given, which can lead to uninteresting conversation.


ELIZA - a friend you could never have before

You: i'm bored. entertain me

Eliza: Does it please you to believe I am bored entertain you?

Figure 1. ELIZA – a simple pattern matching program



Human: i want a puppy
ALICE: *You want only one?*
Human: yes, i want one puppy
ALICE: *They might be cheaper in quantity. Why do you want it?*
Human: because i like puppies
ALICE: *Interesting explanation. A lot of people like that. I do like music. I like Philip K. Dick.*
Human: you dont care about puppies do you?
ALICE: *Of course I care about it. It is a big concern for all of us in this life.*
You say:

Human: i want a christmas tree
ALICE: *You want only one?*
Human: yes, i want one christmas tree
ALICE: *They might be cheaper in quantity. Why do you want it?*
Human: because christmas trees are pretty
ALICE: *Interesting explanation. I think there might be a few exception.*
Human: you dont care about christmas trees do you?
ALICE: *Of course I care about it. It is a big concern for all of us in this life.*
You say:

Figure 2. ALICE – a more complex pattern matching program

Hypothesis

If the chatbot was trained on real conversations, rather than just using generalized forms of the most common sentence types, I hypothesize the chatbot could generate more interesting conversation. This would still be Case Based Reasoning but rather than using generalized data, the program would store past conversation explicitly, and mimic a given screen name personality. The chatbot would only reply using responses learned from the training corpus, and would thus have more emotional and personality content than other chatbots.

Procedure

1. Composing the Training Corpus

a. The training corpus must consist of many conversations involving the username (at least 50 conversations, but the more the better)

b. Overly sensitive information (such as addresses, phone numbers, and any other unsharable data) should be filtered out.

c. Highly technical conversations should be filtered out. This type of information makes for uninteresting conversation. Also most technical conversations are overly specific to a particular problem and configuration and could give misleading information to someone seeking technical help. A troubleshooting chatbot would be better constructed from sorted newsgroup posts or a manual, rather than from conversations.

2. **Parsing the Training Corpus**

   a. Extract message content and screen names from the HTML. Example HTML below with parts to extract. We call the messages from the screen name we are mimicking 'responses' and the rest of the messages 'prompts'.

```
<BODY BGCOLOR="#ffffff"><B><FONT COLOR="#ff0000">Aeschkalet<!-- (11:39:53
PM)--></B>:</FONT><FONT COLOR="#000000"> wake up d000000000d<BR>
<B></FONT><FONT COLOR="#0000ff">VIKRUM<!-- (11:41:01 PM)--
></B>:</FONT><FONT COLOR="#000000"> what's hanging, woman<BR>
<B></FONT><FONT COLOR="#0000ff">VIKRUM<!-- (11:41:11 PM)--
></B>:</FONT><FONT COLOR="#000000"> OH MAN!!! I GOT YOUR NEWYEAR/XMAS
CARD!!</FONT><BR>
```

   b. Group together consecutive messages from the same screen name

   c. Simplify Prompt messages. In instant message conversations, it is common for variations of the same base token to be presented in different ways. Vowels are commonly elongated to show emphasis, and multiple exclamation points may be used in place of just one. These common variations, as well as the casing of the prompt words should be normalized as in the examples below. This is to prevent the loss of data. For instance, it is unlikely that a user will enter "hahahahaha" with precisely 5 'ha's, but the user is very likely to enter two or three 'ha's. Therefore the prompts in the training data should be normalized, and the user prompts at run time should also be normalized, so that the program can generalize to understand variations of a base form of a token. Word stemming, however, is not used, since variations in how a word is conjugated and whether a noun is plural is very relevant, unlike in other NLP problems such as text classification.

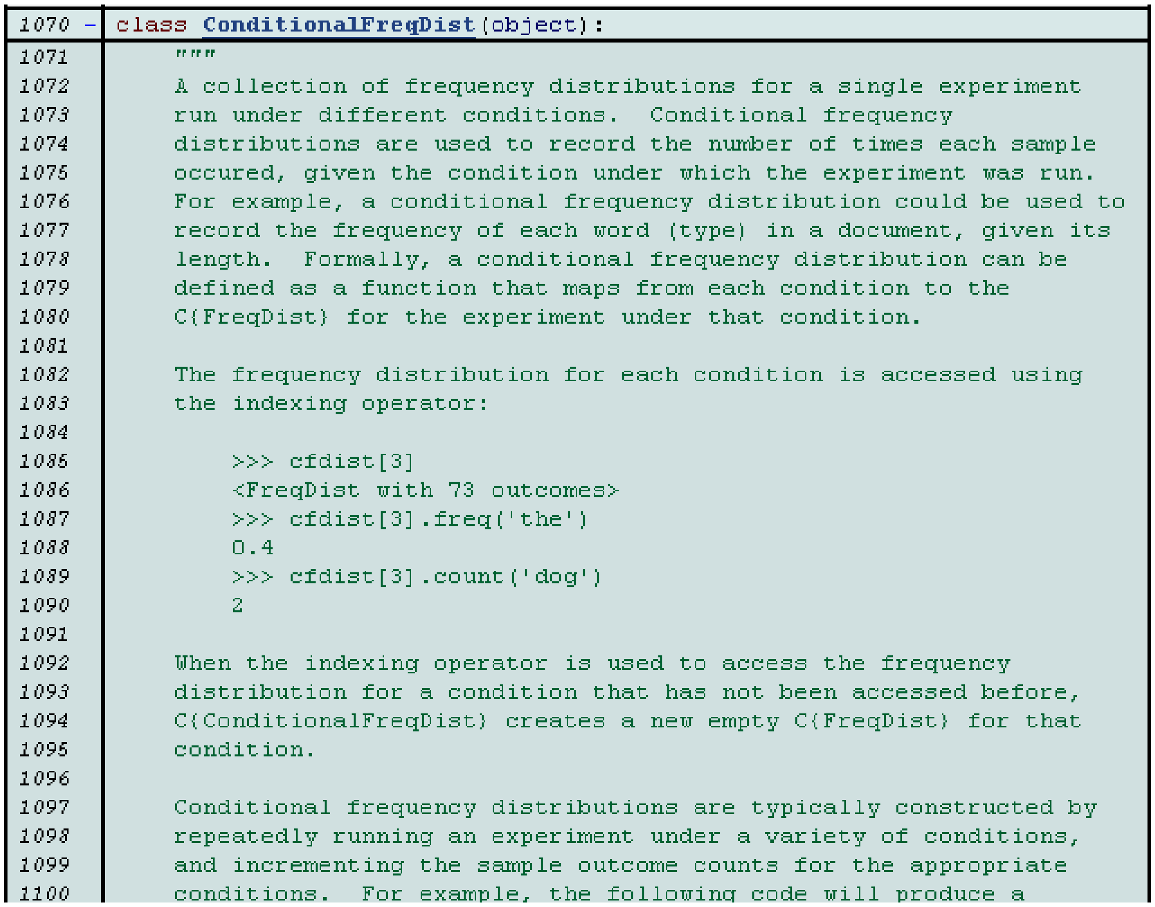
i.	!!!!!!!???????	->	!?
ii.	Ohhhhhhhhhhhh!	->	ohh!
iii.	WhATz uP??	->	whatz up?
iv.	hahahahaha	->	haha

   d. Break prompts into tokens, omitting stop words and punctuation if desired. Stop words for this problem are kept to a minimum, such as words like 'a' and 'the'. This is because most common words, such as 'you' are still important in determining the context of the phrase. The commonality of the word will be taken into account in later steps, so the use of a stop list does not really affect the results, but it can save some memory use. Also certain punctuation, such as question marks, are relevant to the content, so we don't remove those.

i.	Did you take the cat to a vet?	->	[did, you, take, cat, to, vet, ?]

3. **Constructing the Conditional Frequency Distribution**

   a. I use an instance of ConditionalFreqDist from NLTK-Lite to store the training data in a useful format.



```
class ConditionalFreqDist(object):
    """
    A collection of frequency distributions for a single experiment
    run under different conditions.  Conditional frequency
    distributions are used to record the number of times each sample
    occured, given the condition under which the experiment was run.
    For example, a conditional frequency distribution could be used to
    record the frequency of each word (type) in a document, given its
    length.  Formally, a conditional frequency distribution can be
    defined as a function that maps from each condition to the
    C{FreqDist} for the experiment under that condition.

    The frequency distribution for each condition is accessed using
    the indexing operator:

        >>> cfdist[3]
        <FreqDist with 73 outcomes>
        >>> cfdist[3].freq('the')
        0.4
        >>> cfdist[3].count('dog')
        2

    When the indexing operator is used to access the frequency
    distribution for a condition that has not been accessed before,
    C{ConditionalFreqDist} creates a new empty C{FreqDist} for that
    condition.

    Conditional frequency distributions are typically constructed by
    repeatedly running an experiment under a variety of conditions,
    and incrementing the sample outcome counts for the appropriate
    conditions.  For example, the following code will produce a
```

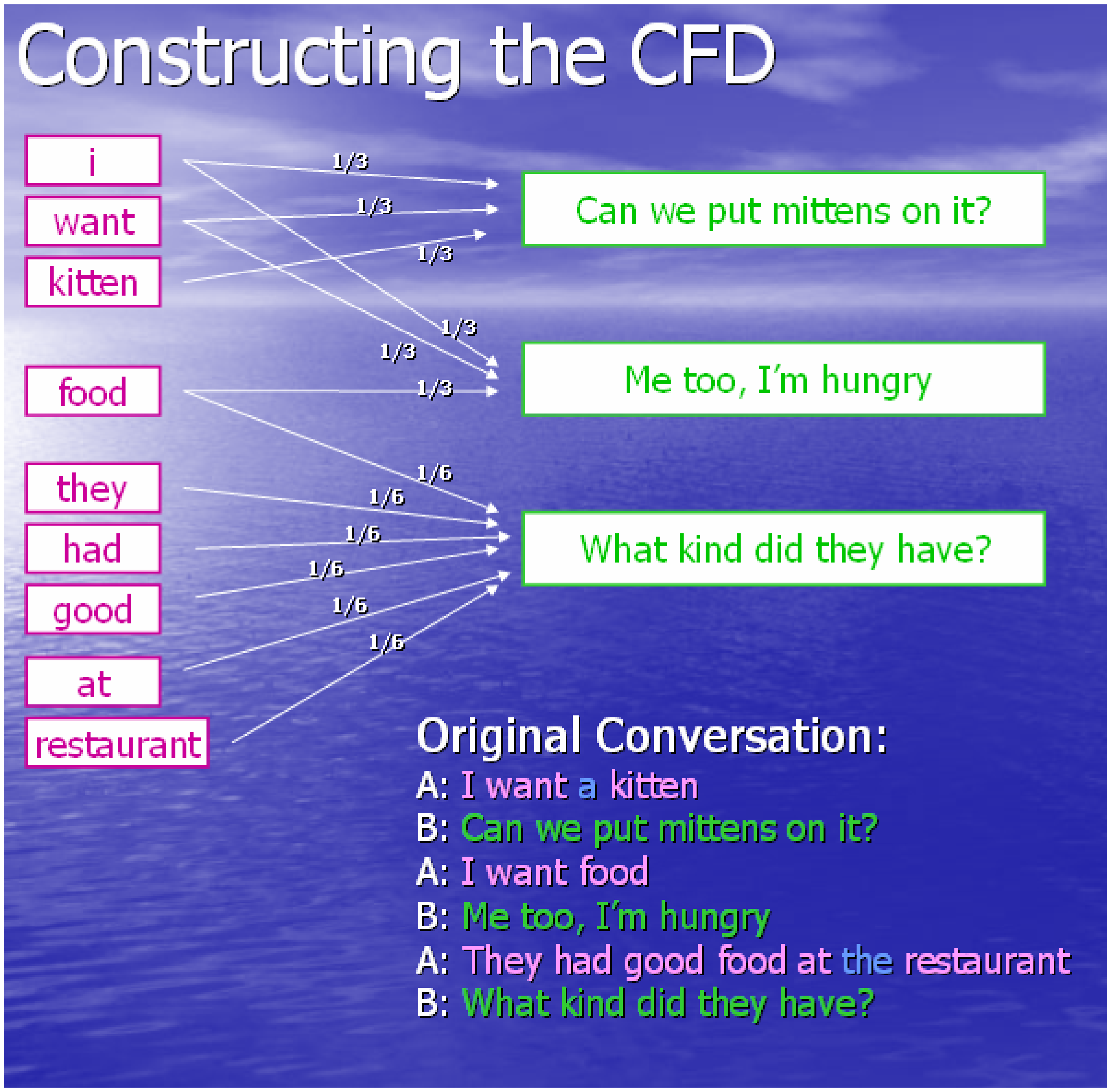
Figure 3. NLTK-Lite's ConditionalFreqDist Data Structure Description

CFDAPI: http://nltk.sourceforge.net/lite/doc/api/nltk_lite.probability-pysrc.html#ConditionalFreqDist

b. At first we take a simplified approach. If there are N tokens in the prompt for a particular response, we assume each of those prompt words was 1/N likely to have caused the response. So we insert the response, with weight 1/N, into the Frequency Distributions indexed by each prompt word. (If this response already exists in an FD, then we just increment its weight by 1/N).

```
# for each prompt words list 'sentMessageWords' and corresponding response:
    weight = 1.0 / len(sentMessageWords)

    for word in sentMessageWords:
        cfd[word].inc(response, weight)
```

Example:



Constructing the CFD

Original Conversation:

A: I want a kitten
B: Can we put mittens on it?
A: I want food
B: Me too, I'm hungry
A: They had good food at the restaurant
B: What kind did they have?

Figure 4. Constructing the Response CFD Illustration

Now if we receive the query word 'want' we can look up our previous responses to that word:

cfd['want'] => [("Can we put mittens on it?", 1/3), ("Me too, I'm hungry", 1/3)]

4. Using the Conditional Frequency Distribution

a. Because the example above is so minimal, for the rest of these examples lets suppose we start out with a CFD that looks like this:

```
----------------------------------------------------------------------------------------------------
cfd['i']       =>    [("Can we put mittens on it?", 1/3), ("Me too, I'm hungry", 1/3)]
                     ("That's' great", 1/3), ("Where?", 1/4), ("Really?", 1/4)]
cfd['you']     =>    [("I'm ok, you?", 1), ("Not bad", 1/2), ("been busy lately", 1/4),
                     ("not much.", 1/4)]
cfd['saw']     =>    [("Really?", 1/4), ("Oh yeah?", 1/4)]
cfd['kitten']  =>    [("Can we put mittens on it?", 1/3), ("Meow!, 1/3)]
```

```
cfd['restaurant'] =>    [("What kind of food did they have?", 1/6)]
----------------------------------------------------------------------------------------------------
```

b. The first approach I had was to take in a new prompt, create an empty FreqDist of responses, and for each token in the prompt, increment all corresponding responses by their weights. The response with the maximum weight in the new FD would then be the best response.

Simple Example:

```
Tokenize("I saw a kitten in the blue restaurant") => ['i', 'saw', 'kitten', 'blue', 'restaurant']
newFD        =>        FreqDist()

# for each prompt word, add contents to newFD
----------------------------------------------------------------------------------------------------
cfd['i']          =>    [("Can we put mittens on it?", 1/3), ("Me too, I'm hungry", 1/3)]
                        ("That's' great", 1/3), ("Where?", 1/4), ("Really?", 1/4)]
cfd['saw']        =>    [("Really?", 1/4), ("Oh yeah?", 1/4)]
cfd['kitten']     =>    [("Can we put mittens on it?", 1/3), ("Meow!, 1/3)]
cfd['blue']       =>    [ ]
cfd['restaurant'] =>    [("What kind of food did they have?", 1/6)]
----------------------------------------------------------------------------------------------------
newFd        =>        [("Can we put mittens on it?",       1/3 + 1/3 = 2/3),
                        ("Really?",                         1/4 + 1/4 = 1/2),
                        ("Me too, I'm hungry",              1/3),
                         ("That's' great",                  1/3),
                        ("Meow!",                           1/3),
                         ("Where?",                         1/4),
                        ("Oh yeah?",                        1/4),
                        ("What kind of food did they have?", 1/6)]
```

So the best response to "I saw a kitten in the blue restaurant." Is "Can we put mittens on it?"

c. The first approach is flawed in that all prompt words are given equal weight, but really each word is not equally likely to have caused the response. More common words are less indicative of meaning. The solution is to take into account the commonality of the word over all conversations.
   i. Divide the weight of the word/response pair by the weight sum over all samples for that word
   ii. Rare words are weighted more; using a dynamic scale

Dynamic Example:

```
Tokenize("I saw a kitten in the blue restaurant") => ['i', 'saw', 'kitten', 'blue', 'restaurant']
newFD        =>        FreqDist()

# for each prompt word, add contents to newFD
----------------------------------------------------------------------------------------------------
cfd['i']          =>    [("Can we put mittens on it?", 1/3), ("Me too, I'm hungry", 1/3)]      Sum: 3/2
                        ("That's' great", 1/3), ("Where?", 1/4), ("Really?", 1/4)]
cfd['saw']        =>    [("Really?", 1/4), ("Oh yeah?", 1/4)]                                  Sum: 1/2
cfd['kitten']     =>    [("Can we put mittens on it?", 1/3), ("Meow!, 1/3)]                    Sum: 2/3
cfd['blue']       =>    [ ]
cfd['restaurant'] =>    [("What kind of food did they have?", 1/6)]                            Sum: 1/6
----------------------------------------------------------------------------------------------------
```

```
newFd      =>      [("What kind of food did they have?",     1/6 / 1/6 =  1),
                    ("Can we put mittens on it?",            1/3 / 3/2  + 1/3 / 2/3 =  13/18),
                    ("Really?",                               ¼ / 3/2 + ¼ / ½ =  2/3),
                    ("Meow!",                                1/3 / 2/3 =  ½),
                    ("Me too, I'm hungry",                   1/3 / 3/2 =  2/9),
                     ("That's' great",                       1/3 / 3/2 =  2/9),
                     ("Where?",                              ¼  / 3/2 =  1/6),
                    ("Oh yeah?",                             ¼  / 3/2 =  1/6)]
```

So the best response to "I saw a kitten in the blue restaurant." Is "What kind of food did they have?"

iii. Notice that what was previously the least likely response is now the most likely response, due to the uniqueness of the word 'restaurant' and the commonality of words like 'I' and 'saw' in the training corpora. The differences are even more pronounced on large corpora where there may be hundreds of responses to a particular keyword.

iv. This change improved the relevancy of responses greatly.

d. To reduce redundancy, we keep a Frequency Distribution (which functions just as a quick to index list) of used responses and don't use them again. For instance, the first time the words [I, saw, kitten, restaurant] are the keywords in a prompt, we will return the response "What kind of food did they have?", because that response has the highest score. But the next time we encounter a prompt with the same words, we will see that the best response has already been used, and so return the next best "Can we put mittens on it?".

e. To speed up computation, I wrote a function saveCFD(cfd, filename) which will write the CFD to a text file for storage. It is then quicker to call loadCFD(filename) rather than parse all the html data from scratch again. This is useful since new bot instances are created frequently when using my program via a CGI script (see part 7).

5. <u>Handling Cases Without Good Statistical Responses</u>

a. Cases in which we have never before encountered the keywords or else we have already used up all good responses (with decent relevancy scores)

b. Try Pattern Matching: If the prompt follows a certain well-known structure, we can fabricate a response with a set of rules, similar to the ELIZA and ALICE chatbots.

c. If both the statistical and the pattern matching approaches fail, we then have to return a less relevant message. We can randomly do one of the following:

i. Pick a random message

ii. Pick a random, less likely to ever be used message (messages toward the end of the relevancy lists). This minimizes redundancy.

iii. Pick a message from a default message list containing things like "I'm bored, let's talk about something else." or "I don't know anything about that." to try to get the user to change topics.

6. Putting it on The Web

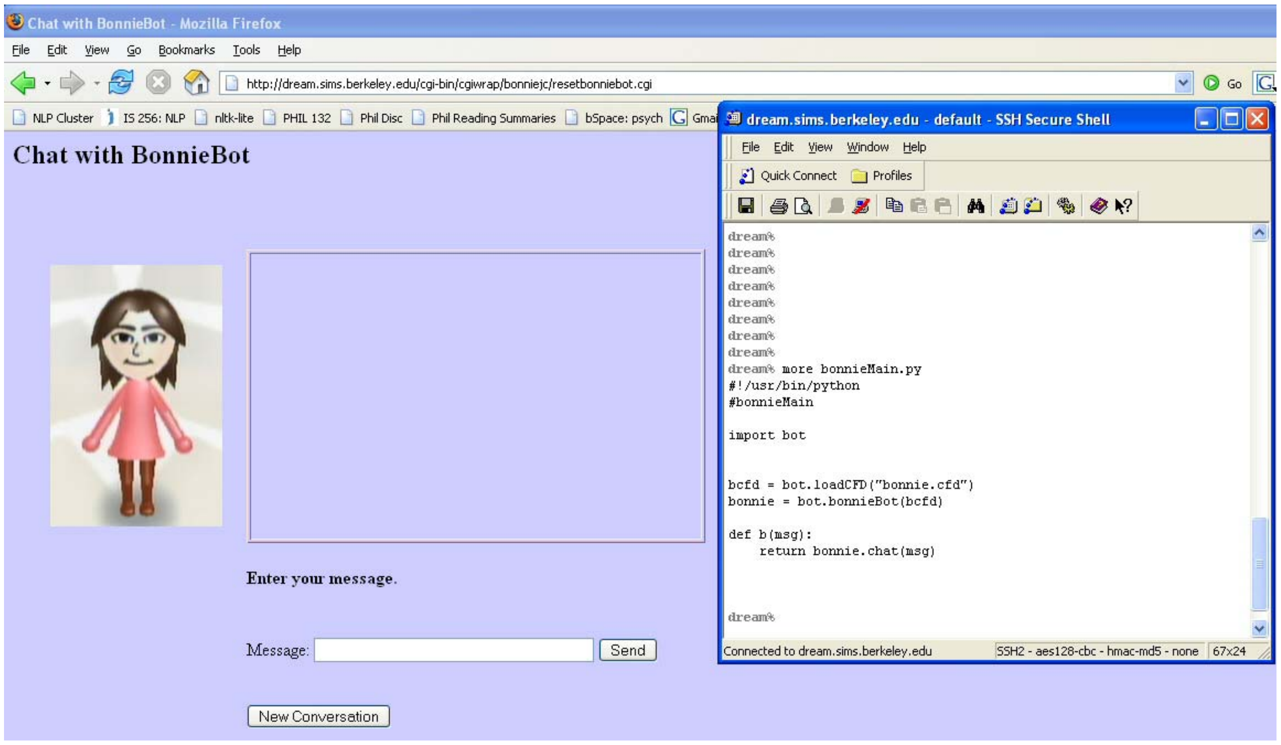
a. **Problem:** It is hard to maintain any "memory" such as used responses, the current username, etc. using a CGI script. Initially, a new bot was created with every message, resulting in no memory whatsoever.

b. **Solution:**

i. Write all bot state changes to a file, including used responses.

ii. Run this file with every prompt message, and reset it when a new conversation starts. Note that this is self-modifying code.

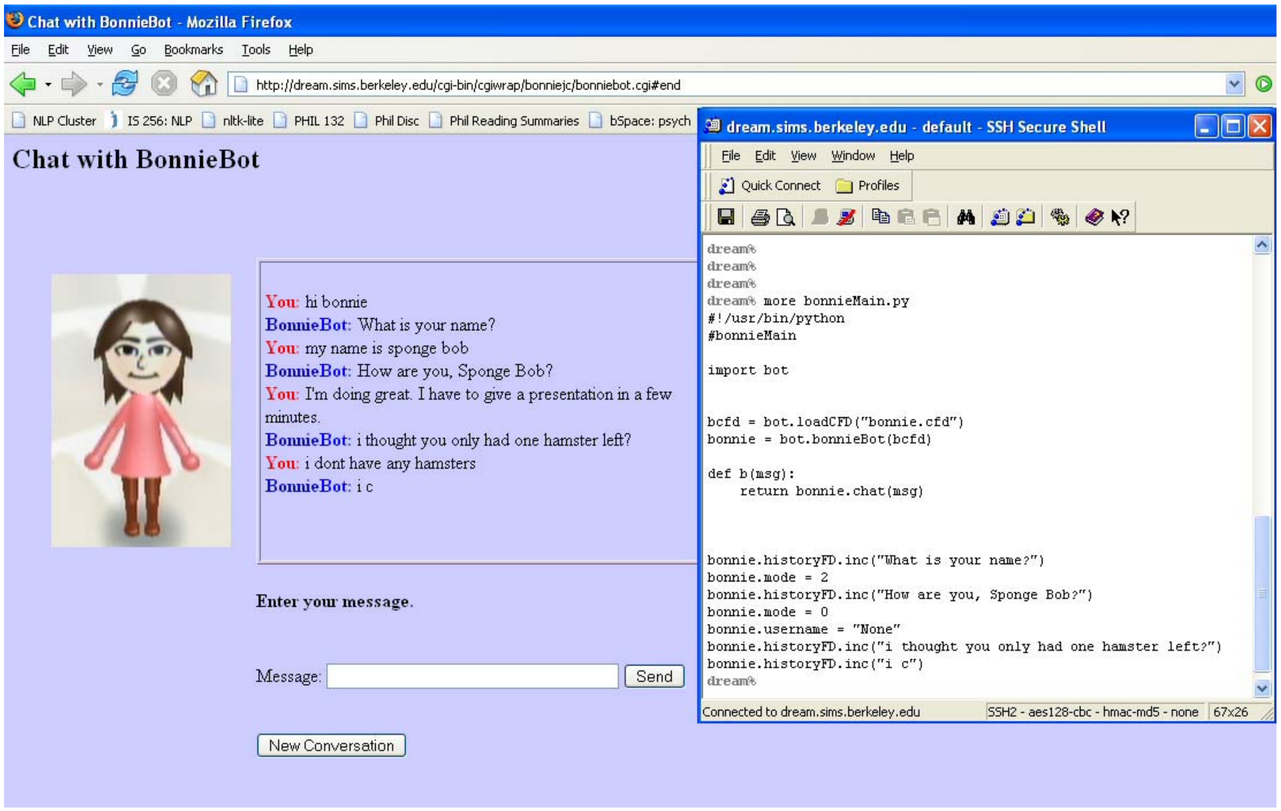
iii. The bot loads the CFD & all state changes from scratch with EVERY call. This is a performance hit, but it is worth making the application slower to retain memory.



Chat with BonnieBot - Mozilla Firefox
http://dream.sims.berkeley.edu/cgi-bin/cgiwrap/bonniejc/resetbonniebot.cgi
Chat with BonnieBot
Enter your message.
Message:
Send
New Conversation
dream.sims.berkeley.edu - default - SSH Secure Shell
Connected to dream.sims.berkeley.edu
SSH2 - aes128-cbc - hmac-md5 - none
67x24

```
dream%
dream%
dream%
dream%
dream%
dream%
dream%
dream%
dream% more bonnieMain.py
#!/usr/bin/python
#bonnieMain

import bot


bcfd = bot.loadCFD("bonnie.cfd")
bonnie = bot.bonnieBot(bcfd)

def b(msg):
    return bonnie.chat(msg)


dream%
```

Figure 5. Base code to initialize a bot at the start of a conversation


Chat with BonnieBot - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://dream.sims.berkeley.edu/cgi-bin/cgiwrap/bonniejc/bonniebot.cgi#end
NLP Cluster
IS 256: NLP
nltk-lite
PHIL 132
Phil Disc
Phil Reading Summaries
bSpace: psych
Chat with BonnieBot
Enter your message.
Message:
Send
New Conversation
dream.sims.berkeley.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles
Connected to dream.sims.berkeley.edu
SSH2 - aes128-cbc - hmac-md5 - none
67x26

You: hi bonnie
BonnieBot: What is your name?
You: my name is sponge bob
BonnieBot: How are you, Sponge Bob?
You: I'm doing great. I have to give a presentation in a few minutes.
BonnieBot: i thought you only had one hamster left?
You: i dont have any hamsters
BonnieBot: i c

```
dream%
dream%
dream%
dream% more bonnieMain.py
#!/usr/bin/python
#bonnieMain

import bot


bcfd = bot.loadCFD("bonnie.cfd")
bonnie = bot.bonnieBot(bcfd)

def b(msg):
    return bonnie.chat(msg)



bonnie.historyFD.inc("What is your name?")
bonnie.mode = 2
bonnie.historyFD.inc("How are you, Sponge Bob?")
bonnie.mode = 0
bonnie.username = "None"
bonnie.historyFD.inc("i thought you only had one hamster left?")
bonnie.historyFD.inc("i c")
dream%
```

Figure 5. Auto-modified code to initialize a bot in the middle of a conversation

Figure 6. Chatbots available at: http://ischool.berkeley.edu/~bonniejc/

Results & Analysis

I conclude that a chatbot which employs both statistical NLP methods as well as heuristic methods, such as pattern matching rules, functions more realistically than a chatbot which only uses one approach or the other. This is because purely heuristic models, such as ELIZA, tend to reuse simple patterns, causing very little variation in response types. Responses are always quite general, with little show of emotion in relation to content. While purely statistic models solve this problem by introducing real emotional content and response variety, they also introduce new problems such as increased randomness and inability to generalize. For instance, they may be well versed in discourse about their favorite music, but unable to say anything intelligible about a different type of music, simply because no such discourse was likely to be in the training corpora. My solution was to try to find a statistically relevant response, but revert to pattern matching responses if none could be found.

Other interesting findings:

1. Adjusting response association weights by the commonality of the keyword improved accuracy greatly
2. Including certain punctuation, such as ? and ! as valid keywords increased response accuracy because they significantly change the context of the words (ex. "Can we go to the store?" vs. "We can go to the store!")
3. Writing good and thorough pattern matching rules is a difficult task, especially since in some cases words and verbs need to be inverted. (Ex. "Do **you** think that **I am** cool?" should reply with something like "Yes, **I** think that **you are** cool.") This is done in the perspective() function.
4. Simplification of prompt text in training and at runtime mitigated information loss as well as wasted memory in the cfd (Ex. Reducing multiple vowels and long consonants to two only. Ohhhhhhhh -> ohh, etc.)
5. Of course, bots trained on more data could converse on a wider variety of topics, and took a little longer to run.
6. And of course, bots trained on a certain screen name personality did sound a lot like that person with regards to tone and vocabulary, only with more sporadic, random messages than usual.

Improvements for the Future

Clearly the results will improve with the quality and quantity of training data and the sophistication of heuristic rules. Therefore, over time I hope to add to the amount of statistical training data as well as develop a more comprehensive set of pattern matching rules.

I also hope to extend memory capabilities throughout a session. Right now, my bots are limited to only remembering one username and which responses they have already used. It would be fascinating, however, to add the ability to learn more content during conversations. For example, a bot could learn to recognize sentences describing close relations to the speaker or what the speaker is doing at the moment, and later ask about those people the speaker mentioned, or how the previous projects and activities went.

The final improvement I would like to make is to make the application more scalable. Right now each bot is designed to handle talking to around one or two people at a time, and the conversations are not private. If I were to enable secure sessions and set browser cookies, I could run a distinct and private session for every user simultaneously. I hope to implement this whenever I obtain computing resources sufficient to handle more wide spread use.

References

Aamodt, Agnar & Plaza, Enric
Case Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches
http://www.iiia.csic.es/People/enric/AICom.html

A.L.I.C.E.
http://www.pandorabots.com/pandora/talk?botid=f5d922d97e345aa1

ELIZA
http://www-ai.ijs.si/eliza-cgi-bin/eliza_script

NLTK-Lite
ConditionalFreqDist API: http://nltk.sourceforge.net/lite/doc/api/
Source Repository: http://nltk.svn.sourceforge.net/viewvc/nltk/