Interaction and Negotiation in Learning and Understanding Dialog

Tom Dean (`tld@google.com`) with a lot of help from my friends*

Abstract

Interaction and negotiation are an essential component of natural language understanding in conversation. We argue this is particularly the case in building artificial agents that rely primarily on language to interact with humans. Rather than thinking about misunderstanding—thinking the user said one thing when he said another—and non-understanding—not having a clue what the user was talking about—as a problem to be overcome, it makes more sense to think of such events as opportunities to learn something and a natural part of understanding that becomes essential when the agent trying to understand has a limited language understanding capability. Moreover, many of the same strategies that are effective in situations in which the agent's limited language facility fails also apply to the agent actively engaging the user in an unobtrusive manner to collect data and ground truth in order to extend its repertoire of services that it can render and to improve its existing language understanding capabilities. In the case of developing an agent to engage users in conversations about music, actively soliciting information from users about their music interests and resolving misunderstandings on both sides about what services the application can offer and what service in particular the user wants now is already a natural part of the conversation. Data collected from thousands or millions of users would provide an invaluable resource for training NLP components that could be used to build more sophisticated conversational agents.

Building Conversational Agents

The Zinn Project aims to develop a chatterbot capable of sustained conversation about a single topic such as popular music. I purposefully used the deprecatory term "chatterbot" to emphasize that, while we aspire to build an agent that is relatively adept at conversing in natural language, we have no pretensions that such an agent—call it "Zinn"—will *understand* the conversation in any but the most shallow sense of the word. We do, however, aspire to having Zinn perform *as if* it understands most of the time, and thus is able to deliver value in an engaging conversational manner.

Like most chatterbots, Zinn relies on keyword spotting—scanning user input searching for words that might provide clues as to the user's intentions. We use relatively sophisticated methods for spotting keywords including Google NLP technology based on deep networks. What distinguishes Zinn from other chatterbots is that, if Zinn can't find evidence to support an hypothesis about the user's current intention by scanning the last few utterances, then Zinn enlists the user's help in a side conversation with the express purpose of resolving the underlying ambiguity.

---

*Thanks to the members of the Zinn team, Johnny Chen, Sudeep Gandhe, Shalini Ghosh, Anja Hauth, Luheng He, Peter Lau, Xin Rong and Gabe Schine who are building the first instantiation of Zinn and dealing with the myriad technical challenges involved in building a real system that I was able to gloss over in presenting the ideas in this document. Many of those ideas surfaced from discussions in our daily scrum meetings, and nearly all of the hard-won insights into what works and what doesn't in developing scalable dialog systems came from those same meetings or side discussions with individuals on the team. And special thanks to Bill Byrne for the time he spent convincing me how important it is to get the language right and in sync with the user's expectations and for showing me some of his techniques for implementing natural, contextually-aware language generation.

This reliance on interactive understanding is potentially Zinn's most important contribution in terms of generally useful technology. In point of fact, it's not a new idea; researchers have long understood that exchanging information is inherently interactive if the two parties involved in the exchange are not essentially equal to begin with, in which case, what's the point? In order to understand what the user is saying, Zinn relies on the user to help it out by answering questions and supplying additional information as required for Zinn to do its job.

Engaging the user to help avoid or recover from errors in understanding has an additional benefit that is critical to the longer term goals of the Zinn Project, namely, many of strategies that work for dealing with misunderstanding, also work to learn about colloquial dialog. If Zinn doesn't understand something the user said, it can ask for a paraphrase. If it has two or more hypotheses, it can ask the user to pick one. If Zinn thinks the user might be asking it to play a song by a particular artist, it can ask if this is the correct interpretation, or start playing the music and see if the user protests.

Launching Zinn as a music entertainment application is the first step in building more sophisticated agents that rely on language models trained on extensive logs of users interacting with simpler agents like the first instantiations of Zinn. Even our first instantiation will depend on massive amounts of music and dialog data mined from news, books, interviews and technical-help chat logs. The size of this data is in the billions of words, but much of it is not well aligned with the music application and billions of words doesn't begin to do justice to the diversity of human language generation.

We are in the process of building an end-to-end music application. The actual delivery of music content will be limited and not as nuanced as we might like—we are engineers, not user-experience experts. However, if we are successful, we will have demonstrated that our approach can enhance the language understanding technologies that define the state of the art in dialog systems. This document introduces the basic strategies for resolving ambiguity and recovering from failures in understanding. The accompanying Python code illustrates how these strategies can be realized in a simple planning system.

We will also argue that the various components of the error handling sub system provide solutions to three other problems in developing dialog systems. First, the same components used to solicit clues pertaining to the user's intent can be used to evaluate Zinn's performance as a natural part of conversation. Second, such feedback can serve as a reward signal for reinforcement learning, some form of which will be needed to scale any existing application. Finally, Zinn can elicit the ground truth necessary to train deep and recurrent networks for generating proposals to help Zinn dealing with novel input.

Natural Language Generation

We'll begin the discussion by considering one of most overlooked and under-appreciated challenges in building dialog systems, namely natural language generation (NLG). While it is easy to generate basic responses by instantiating variables in predefined patterns, obviously "canned" responses wear thin in short order, and, with a little more thought, the dialog-system developer can substantially improve the user experience by producing more natural, contextually-aware responses. Next we consider the special case of natural language understanding (NLU) typically faced by a dialog system that need only know about a very limited domain of discourse. In each case, we introduce simple NLP technology that supports some degree of application dependence, while allowing the error handling sub system to be relatively domain independent. Last, we consider tools from automated planning to expedite the development of modular dialog components.

Traditional Schema Support

Schema represented as strings with schema variables and coupled with variable bindings are the mainstay of simple most chatterbot applications:

```
>> instantiate("Am I right that you like $genre?", {'genre':'jazz'})
Am I right that you like jazz?
```

To add variety to generated dialog, we use lists of strings representing linguistic variation but conveying the same general meaning called *variants*.

Programmable Linguistic Variation

Variants are registered in a dictionary and identified by keywords corresponding to strings consisting of uppercase letters and underscores (`[A-Z_]*`), e.g., `IS_IT_SO`. The strings that comprise a variant can be full sentences or common / idiomatic sentence fragments that can be used effectively as "subroutines" in generating dialog-appropriate language:

```
>> variant('IS_IT_SO',["am I right that","is it the case"])
>> expand("IS_IT_SO you like $genre?", {'genre':'jazz'})
Is it the case you like jazz?
```

The function `expand` splits its first argument into tokens separated by white space so that `"IS_IT_SO you like $genre?"` is analyzed as the list of tokens `["IS_IT_SO","you","like","$genre?"]`. Each token in the list is checked to see if it corresponds to a keyword in the variant registry, variable substitutions are made using `instantiate`, and additional transformations are applied recursively as needed.

The strings that comprise the list of syntactic variations associated with a given keyword can contain additional variant keywords. When a variant keyword is encountered, we choose a random selection from the variant's list of linguistic variations, apply `expand` recursively, and then substitute the result for the keyword in the expanded input that we started with. You can also embed lists of syntactic variations directly in a string:

```
>> expand("IS_IT_SO ['that',''] you like $genre?", {'genre':'jazz'})
Is it the case you like jazz?
>> expand("IS_IT_SO ['that',''] you like $genre?", {'genre':'jazz'})
Am I right that you like jazz?
```

Once a string is completely expanded having applied all transformations and made all substitutions, the resulting list of tokens is used to produce to a single string capitalizing words at the beginning of sentences and adding punctuation as needed. There is no pretense that the resulting strings do justice to the richness of natural language, but the options available to the dialog developer allow for a reasonable degree of variability with a modicum of programming complexity.

Simple Word Agreement

Linguistic variants can be extended to support simple sorts of local syntactic or semantic agreement. For example, number agreement can be implemented in an ad hoc manner as shown in the following example:

```
>> variant('IS_OR_ARE', {'function':is_or_are,'arguments':['$slots']})
>> slots = ['artist']
>> expand('The $slots IS_OR_ARE filled.','slots':join(slots,'and'))
```

```
The artist slot is filled.
>> slots = [’artist’,’album’,’genre’]
>> expand(’The $slots IS_OR_ARE filled.’,’slots’:join(slots,’and’))
The artist, album and genre slots are filled.
```

Obviously this could be extended handle a wider range of cases without necessarily solving the general problem, which is rife with edge cases in a language like English. Gender pronoun agreement can be finessed for entities in the Knowledge Graph or when a given (first) name is provided that is common enough to be included in lists for naming baby girls and boys and the given name of the person in question is in accord with common practice:

```
>> variant(’HIS_OR_HER’,’{’function’:his_or_her,’arguments’:[’$entity’]})
>> bdgs = {’entity’:’Michael Jackson’,’song’:’Thriller’}
>> expand("You mentioned $entity.  Do you like HIS_OR_HER song $song",bdgs)
>> You mentioned Michael Jackson.  Do you like his song Thriller?
>> bdgs = {’entity’:’Adele Adkins’,’song’:’Skyfall’}
>> expand("You mentioned $entity.  Do you like HIS_OR_HER song $song",bdgs)
>> You mentioned Adele Adkins.  Do you like her song Skyfall?
```

Domain Specific Language Payloads

One of our goals in building modular dialog systems with components that can be used in multiple application-specific products is to design domain-independent, function-specific components that can be reused or easily customized to work with modules that require application-specific dialog. Some degree of customization is possible in the form of linguistic variants that enhance or replace variants at the module or sub module level. Another approach is to build modules that define a default—preferably domain-independent—collection of variants, but allow substitutions from other modules at the level of particular invocations, e.g., the called module has a default of the form of the generic `"I didn’t understand what you said."` and the calling module supplies a domain-specific substitution `"I didn’t catch you mention an artist or genre."`. We support this sort of interface between a called and calling module by introducing the notion of a domain-specific *payload* that provides application-relevant language substitutions at the level of specific instances of dialog language generation. Here’s a simple payload that a music discovery dialog module might share with a general-purpose error recovery module that attempts to recover when the calling module recognizes that it has failed in an attempt to solicit information from the user to fill a slot in a schema:

```
payload(’called’:’recover’,
        ’caller’:’music’,
        ’select’:{’leader’:"I didn’t catch mention of any music.",
                  ’prompt’:"What music would you like to hear?"})
```

A payload is primarily a dictionary with keys corresponding to specific instances of dialog language generation within the called module that are mapped to linguistic variants where in the above example we show the simplest case of a single response string. There is a registry for payloads so that all the called module needs to know is the name of the calling module. Here’s a code fragment from a sub module of a misunderstanding mitigation and recovery module:

```
def recover_slots_m(state, slots, caller=None):
    if caller:
        payload = get_payload[caller]['recover']
    else:
        payload = get_payload['default']['recover']
    # normalize and expand aliases in the list of slots.
    slots = complete(slots)
    # check to see if the slots have already been filled.
    unfilled = [slot for slot in slots if not state.var['premise'][slot]]
    if not unfilled:
        return []
    else:
        # start with a leader describing the situation.
        interject(payload['select']['leader'])
        # then prompt the user for whatever is required.
        input = interact(payload['select']['prompt'])
        # see if user input could be used to fill slots.
        matches = match(input)
        ...
```

Building modules that can maintain their separation, play nicely together and not introduce too much complexity in the resulting code is difficult to say the least. Building a system with natural dialog and good coverage is hard enough on its own.

Natural Language Understanding

Our approach to natural language understanding (NLU) is—like our approach to NLG—a study in compromise, working hard to find a middle ground between the simple chatterbot and a—yet to be realized—full AI-complete solution capable of passing itself off as a human. The two main threads of our solution consist of (a) extensions of simple keyword and sentiment spotting designed to expand and smooth out the relevant lexical space, and (b) error mitigation and recovery in which we view the process of understanding as a dialog in which we work with the user to narrow down the meaning of user input sufficiently to provide value, e.g., play music that the user enjoys.

Robust Keyword Spotting

There are obvious things that are just too easy and useful to ignore: robust, precisely-targeted matching employs sets of words and phrases that express certain categories, including but not limited to the following:

- positive and negative affect—does the user's response indicate that whatever she's talking about she seems to be positively (negatively) disposed toward it, e.g., "I love the Beatles" or "I can't stand pop music";
- quantification—does her response indicate a universal, e.g., "I like all music by Adelle", existential, e.g., "I like some jazz trumpet", or blanket rejection, e.g., "Don't ever play anything by her again";
- words indicating order or numbers that could refer to an entity introduced earlier or an item in a list offered to the user for her selection, e.g., "Who sang that last song","2", "I'll take the second one" and "How about the fourth";

The simplest approach is to generate a set of words for each distinct concept and then search for similar words in the user's input. Similarity might be measured using edit distance—using Levenshtein's algorithm, or cosine similarity—using embedding spaces and nearest-neighbor search, or special-purpose classifiers—using DNNs and RNNs for sentiment-analysis. So-called similar word expansion and re-weighting (SWEAR) methods like nearest-neighbor search in a vector-space embedding model are now commonly used in information retrieval applications. We are also looking at semi-supervised, multi-class learning approaches using label amplification by semantic similarity (LASS) for mapping utterances to specific services—"play the latest song by Adele" or "find a video of Michael Jackson singing Thriller in concert"—the dialog system is able to provide to the user.

Collaborative Language Understanding

Robust keyword spotting is a powerful method for understanding user intent—though "guessing" is a more accurate description of what the system is actually doing. But no matter how effective these methods are in dealing with the most common sort of user input, they inevitably fall short in dealing with more complicated and ambiguous locutions. Our fallback strategy is to engage the user and entice him or her to assist us: if you don't understand what someone is saying to you, ask them to repeat, rephrase, confirm or deny what you thought they said; clarify some particular aspect of what they said; select from two or more interpretations you're entertaining, or proceed as if you knew what they were talking about and count on them to correct you if you've misunderstood. We do it so often in everyday conversation that we take it for granted, and the interactive, conversational aspect of understanding is all too often overlooked.

Hierarchical Task Networks for Dialog

In the previous section we talked about modules and sub modules without any detailed description of what they are or how they're implemented. In this section, we'll define a module as a hierarchical network—directed acyclic graph—of *tasks* related to one another in that if there is an edge from task A to task B in the graph then B is said to be a sub task of A. The tasks in the network are further organized in terms of *expansions* so that a task A is mapped to—or *expanded* into—a partially ordered set of its subtasks P referred to as a *plan* for carrying out A. Intuitively, if $P = \{S_1, S_2, ...S_n\}$ together with an ordering $<$ is a plan for A, and a planner carries out the $\{S_i\}$ in the specified order, then the planner is likely to have carried out A, thereby achieving the *goal* that A was intended to accomplish.

Hierarchical Task Networks (HTNs) are commonly discussed in introductory AI textbooks [13, 5], but they also appear in books on logistics planning, operations research, and project planning. In addition to tasks, plans and goals, the other important concepts are *states*, *state variables* and *actions*. Actions that have consequences called *postconditions* corresponding to changes in the values of state variables brought about by *executing* an action. Postconditions that are contingent on antecedents called *preconditions* corresponding to specific values of state variables prior to execution. Goals are typically defined in terms of state-variable assignments. Actions are also called *primitive* tasks or *operators* in the parlance of AI planning. The term operator is usually reserved to describe the state changes brought about by executing an action.

Automated planning and dialog management have been linked to one another in AI for decades [10, 22, 3], but James Allen and his students and colleagues at the University of Rochester are probably most responsible for developing the theory of plan-based dialog management—see here for an overview his research and publications on automated planning and natural language understanding[1] In planning models of discourse, a

---

[1] Here is an excerpt from James Allen's Conversational Interactions and Spoken Dialog Research Group webpage:

> Our research in discourse is focused on two-person extended dialogs in which the speakers have specific tasks to accomplish. An emphasis in this work on developing a theory of dialogue as a collaborative problem solving activity,

task for greeting a user for the first time might be represented as `('greet', 'first')` and associated with a method that handles interaction with the user:

```
def greet_intro_m(state,turn):
    if state.user['name'] == None:
        if turn == 'first':
            input = interact("I'm Zinn. What's your name?")
        else:
            input = interact("Let's try again. Your name?")
        matches = lookup(input,'name')
        if matches:
            matches = interact('Hi {}. Did I get that right?'.format(matches))
            if matches:
                return [('commit', 'confirm', matches)]
            else:
                return [('recover', 'greet', matches)]
        elif turn == 'first':
            return [('greet', 'second')]
        else:
            return False
    else:
        return False

declare_methods('greet', greet_intro_m)
```

The method returns `False` unless the first argument representing the current state has the variable `user['name']` assigned `None`. The `greet` method then prompts the user, attempts to extract a name from the user's input, and, assuming it comes up with one or more hypotheses, tries to get confirmation. If the hypothesis is confirmed, the method generates a `commit` subtask, which is primitive and thus an action with an associated operator that, in this case, updates the current state with the user's name. In the case of a dialog system, the state records the planner's beliefs, including state variables whose values are unknown—propositions for which currently there is no evidence, premised—hypothesized but lacking reliable evidence, and confirmed—established based on reliable evidence or implied by the values of other variables for which there is reliable evidence. A more sophisticated model might include some form of probabilistic reasoning, but the simple model just introduced should suffice for the present exposition. If the user responds but the system can't confirm the name, the method spawns a recovery task, more about later.

```
>> # Error handling as part of greeting the user:
>> Zinn:  I'm Zinn.  What's your name?
```

---

where the current problem solving situation is used to solve problems in semantic interpretation and the recognition of the intentions underlying the speakers' utterances. Highlights of work in this area include the development of the first computational model of speech acts, the development of a multi-level plan-based analysis involving discourse, and the development of an overall architecture for dialogue systems driven by a collaborative problem solving agent. While it is important for work to be formally well-defined and understood, it is equally important that computational theories can lead to effective implementations. We have demonstrated and tested our models in a wide range of different applications. Most recently, we have been focusing on task/workflow learning systems in which the the system learns a task model from a dialogue with the user that includes a single demonstration of the task. By combining deep language understanding, reasoning, learning and dialog, we can learn robust task models in a matter of minutes.

```
>> User:  It’s Al
>> Zinn:  Hi Alan.  Did I get that right?
>> User:  no
>> # Here ’greet’ generates a ’recover’ sub task:
>> Zinn:  Is it one of these:  ’Alan’ or ’Alfred’?
>> User:  it’s the second one
>> Zinn:  Thanks Alfred.  Hopefully I got it this time.
>> # This time doesn’t even recognize the name:
>> Zinn:  I’m Zinn.  What’s your name?
>> User:  Don Quixote
>> Zinn:  Let’s try again.  Your name?
>> User:  Barbie
>> Zinn:  Hi Blaise.  Did I get that right?
>> User:  nada
>> # Doesn’t get the name but salvages the dialog:
>> Zinn:  Is it one of these:  ’Blaise’ or ’Gabe’?
>> User:  nope
>> Zinn:  One more guess.  Is it Gabe?
>> User:  I said no
>> Zinn:  Rats!  Well, we can still talk about music.
```

Here’s a more complex task intended to initiate a session with a conversational music application that starts off by asking the user what sort of music she’d like to hear. This example illustrates elements of both NLG and NLU. The dialog system tries three times to get the user’s attention giving her longer to respond each time, until it finally gives up. Each time the system solicits input it uses different language reflecting its prior attempts to get a response. The dialog system expects the user to respond with something like “Can you play something by the Clash?” and, depending on what it is able to infer from the user’s response, it formulates a different plan—sequence of sub tasks—to continue the dialog.

```
def play_start_m(state,type):
    """
    Initial ’play’ task invoked by [(’play’,’start’)]
    Demonstrates the use of timers in selecting music.
    """
    if type == ’start’:
        # zinn gives the user increasingly more time.
        input = False
        if patience(expand(’ASK_WHAT_PLAY’),4,False):
            input = rewarded()
        else:
            if patience(expand(’I_CAN_PLAY’),8,True):
                input = rewarded()
            else:
                if patience(expand(’I_CAN_SUGGEST’),12,True):
                    input = rewarded()
        if input:
            matches = match(input)
            if matches['artist'] or matches['genre']:
                # user intent implied by music.
                return [('commit', 'premise', matches),
                        ('confirm', 'play'),
                        ('confirm', 'music'),
                        ('perform', 'music')]
            elif matches['play']:
                # explicit intent but no music.
                return [('commit', 'premise', matches),
                        ('confirm', 'play'),
                        ('select', 'music'),
                        ('perform', 'music')]
            else:
                # unsure user wants to hear music.
                return [('recover', 'music')]
        else:
            interject(expand('SORRY_GOODBYE'),True)
            # if no user response, terminate session.
            return False
    else:
        # if 'play' type isn't 'start', backtrack.
        return False

declare_methods('play', play_start_m)
```

Here's an example showing the method associated with the `('play', 'start')` task trying to engage a new user—the misspellings are intentional:

```
    >> # Demonstrate 'play start' patiently waiting for input:
    >> Zinn:  Tell me what you'd like to hear?
    >> User:  ....
    >> Zinn:  Maybe you have a favorite band you'd like to hear?
    >> User:  ........
    >> Zinn:  Would you like me to just play something for you?
    >> User:  plsy something slow
    >> Zinn:  I didn't catch mention of any music.
    >> Zinn:  What music would you like to hear?
    >> User:  play some sring
    >> # 'play' generates a 'confirm' following a 'commit task:
    >> Zinn:  By "play some sring" does that roughly mean "play Sting"?
    >> User:  yes
    >> Zinn:  I got that you want to hear some music.
```

Here's the method for the `('confirm', 'music')` task generated in the above bit of dialog. At the time this method is called, the state variable `'play'` is a premise and not yet confirmed, and so the method will generate a `('commit', 'confirm')` task that will set the `'play'` state to confirmed:

```
def confirm_play_m(state, type):
    if type == 'play':
        if state.var['confirm']['play']:
            # it's already confirmed!
            interject(expand('I_GOT you want me to play some music.'))
            return []
        elif state.var['premise']['play']:
            # it needs to be confirmed.
            bindings = {'this':history(), 'music':got_music(state)}
            utterance = 'WHEN_YOU_SAY "$this" IS_WAY_SAY "play $music"?'
            # ask for confirmation.
            input = interact(expand(utterance,bindings))
            if lookup(input,'positive'):
                # yay, play is confirmed.
                interject(expand('I_GOT you want to hear some music.'))
                return [('commit', 'confirm', {'play':True})]
            else:
                # uh oh, gonna be complicated.
                return [('recover', 'play')]
        else:
            # this is totally unexpected!
            return [('recover', 'play')]
    else:
        return False
```

Here's a slightly different but complete version of the `recover` task similar to the one we introduced earlier in the context of in explaining payloads:

```
def recover_slots_m(state, slots, caller=None):
    # see if the calling task has an associated payload:
    if caller:
        pay = get_payload[caller]['recover']
    # normalize and expand aliases in the list of slots.
    slots = complete(slots)
    # check to see if the slots have already been filled.
    unfilled = [slot for slot in slots if not state.var['premise'][slot]]
    if not unfilled:
        return []
    else:
        # try twice to recover and then admit failure.
        if 'leader' in pay['select']:
            # if there's a domain-specific leader use it.
            interject(pay['select']['leader'])
        else:
            # otherwise display a partial-error leader.
            interject("I didn't get all of what you said.")
        # last opportunity for customizing this task.
        if 'prompt' in pay['select']:
            # if there's a domain-specific prompt use it.
            input = interact(pay[’select’][’prompt’])
        else:
            input = interact(’Give me one {}.’.format(join(unfilled,’or’)))
        matches = match(input)
        filled = [slot for slot in unfilled if matches[slot]]
        if filled:
            interject(’Thanks. I got your {}.’.format(join(filled,’and’)))
            return [(’commit’, ’confirm’, matches)]
        else:
            # try another general error prompt noting earlier.
            interject("I didn’t get that either.")
            input = interact(’Could you say that a different way?’)
            matches = match(input)
            filled = [slot for slot in unfilled if matches[slot]]
            if filled:
                interject(’I got your {} that time.’.format(join(filled)))
                return [(’commit’, ’confirm’, matches)]
            else:
                interject("Sorry. I’m totally confused.")
                return False

declare_methods(’recover’, recover_slots_m)
```

So far, the examples have only hinted at the hierarchical structure of dialog. Intuitively, in asking someone what sort of music they like, on hearing their reply you might ask them something in return to confirm what you thought you heard. There are multiple strategies for doing so. Your request for confirmation needn’t be so blunt as “Did you say Michael Jackson?” It could be in the form of a rhetorical question as in “So you like Michael Jackson. His early work with the Jackson Five or more recent solo work like Thriller?”, or it might be in the form of a statement that implicitly reveals your interpretation of what they said, and assumes your conversational partner will realize if you’ve got it wrong and point it out if necessary: “It’s a shame so many celebrities seem compelled to take drugs that their physicians are only too willing to prescribe.” Of course, this last strategy could backfire if your partner replies “What other celebrities are you referring to?”, and you’re as clueless as Zinn and haven’t the foggiest what she just said.

There are two features of HTN planning that make it well suited to handling these sort of exchanges. First, for a given task like confirming what music the user likes to listen to, you can have multiple plans for achieving confirmation. If one plan fails, then the planner will backtrack and try another. Second, a task to confirm the user’s intent, can push sub tasks on the stack to deal with different contingencies encountered in the process of executing the confirmation task. The difference between executing and evaluating plans—both of which can involve interaction with the user—is somewhat blurred in dialog planning in part due to the collaborative nature of conversation. It may be necessary to support a flow-of-control construct like the “cut” sign (!) in Prolog that precludes backtracking from a failure, which in the case of dialog planning would distinguish between “I give up. Let’s reset and get on with the main event.” and “This isn’t working. Let’s backtrack and try another plan.” A more parsimonious approach is to use backtracking for exactly what it was intended and provide plans that capture the idea of cutting one’s losses and getting on with life. We adopt the latter and provide examples subsequently.

The hierarchy of tasks for a simplified version of the play-music application described in this document is listed below. We don’t go into as much detail as we did in the earlier examples, but rather list the tasks and the sub tasks and actions they example into. There are three levels of expansion within the application portion of the network:

```
# Tasks:
# ('play',{'start','continue'})
# - initiate or continue 'play music'
# - expands to sequence of 'select', 'confirm' and 'perform' tasks

# ('select',{'music','album','artist','genre','song'})
# - identify instance(s) of 'album', 'artist', 'genre' and 'song'
# - expands to sequence of 'confirm' tasks

# ('confirm',{'album','artist','genre','song'})
# - solicit confirmation of prior 'select' task
# - expands to either 'recover' tasks or one or more 'commit' actions

# ('perform',{'music','video'})
# - handle the performance of selected and confirmed media
# - expands to a sequence of 'choose', 'queue, and 'monitor' actions

# Actions:
# ('choose',{'audio','music','video'})
# - choose specific media to perform, e.g., album track or YT video
# ('queue',{'audio','music','video'})
# - queue up the selected media on an appropriate media player
# ('monitor',{'audio','music','video'})
# - respond to interrupts and generate new tasks to obtain feedback
# ('commit','','')
# - modifies the current state variable to reflect Zinn's knowledge
```

It can be tricky building any sort of dialog system. The approach based on hierarchical task networks outlined here, simplifies some aspects of system development, but getting all the edge cases right, inferring user intent and generating appropriate responses is still challenging. There has been some progress in learning various aspects of dialog management [15, 4] that we will discuss below, but there is no completely general and effective approach available at this time and, at the very least, we will have to hand-code the most basic tasks plans and for a given application. However, there is some hope that we can build a generic error mitigation and recovery module that can be plugged into any application, possibly with the aid of some linguistic lubrication in the form of an application-appropriate payload such as we discussed earlier in this manuscript.

Error Mitigation and Recovery

The `recover` task introduced earlier represented but one strategy for recovering when confirmation fails. Linguists interested in discourse have categorized the different strategies people commonly use to resolve ambiguity and recover from non-understanding, and computer scientists have implemented those strategies in various dialog management systems [11, 12, 1, 3, 7]. In this section, we briefly describe one such approach, and start by providing a set of `'recover'` tasks adapted from the RavenClaw non-understanding strategies described in [2]. The following `u_*` recovery strategies encourage the user to provide more or better information to help in recovering from Zinn's present problem of understanding the user's intention—in some cases an example domain-independent payload[2] is included for illustration purposes:

```
Tasks: ('recover', { ... see below ... }, *ARGS)

recover_u_strategies = \
    {'u_rewords':  # z asks user to rephrase her previous response:
     {'prompt':variant('U_REWORDS_PROMPT':
                       ['Could you restate that more simply.',
                        'Could you say that in another way.',
                        'Is there another way of saying that.'])},
     'u_focuses':  # z asks user to focus response on the context:
     {'prompt':variant('U_FOCUSES_PROMPT':
                       ['Could you be more explicit about {}.',
                        "I didn't get what you said about {}.",
                        'Can you tell me some more about {}.'])},
     'u_chooses':  # z asks user to select item from a short menu:
     {'prompt':variant('U_CHOOSES_PROMPT':
                       ['Please choose from the following: {}',
                        'Perhaps one of the following works: {}',
                        'Could you select from this menu: {}'])},
     'u_parrots':  # z provides examples of text it can handle:
     {'prompt':variant('U_PARROTS_PROMPT':
                       ['Here are some examples I can handle:',
                        'It would help to say something like:',
                        'Could you describe it like one of these:'])}}
```

In the `z_*` recovery strategies, Zinn takes the initiative by repeating its last utterance, thereby offering the user another chance to respond, or, alternatively, by requesting additional information from the user or, in one way or another, avoiding the problem altogether. Asking for additional information can be dangerous as as the ploy might just backfire resulting in Zinn being more confused and the user less inclined to sustain the conversation with so inept a partner. Generally, the best option is to ask one or more yes-or-no questions or allow the user to select from a short list of options as in the `u_chooses` strategy above:

```
recover_z_strategies = \
    {'z_repeats':  # repeat the last system-generated utterance:
     {},           # - no payload since last utterance == history[0]
     'z_rewords':  # reword the last system-generated utterance:
     {}            # - requires an application-specific payload
     'z_explains': # explain the misunderstanding in some detail:
     {},           # - requires an application-specific payload
     'z_advances': # zinn moves on in the dialog ignoring problem:
```

---

[2] In previous examples, the "calling" task uses `interject` to make a domain-dependent request for confirmation, e.g., `"Hi {}. Did I get that right?".format(name)`, and the domain-independent confirmation task handles the reply to the user's response, e.g., `"Great. Thanks for being patient with me"` or `"Rats! Let's forget about it for now."` We might have the application designer provide a collection of schema along with slots to be filled by the calling task, and have the confirmation task handle both confirmation requests and responses. If provided, these would override the set of application-independent schema. These schema could be as simple as `"Rats! Well at least we can have fun playing some music."` [a schema with zero slots but specific to the play-music application], or as complex as `"Nice. Looking at your profile I see you like 'Canned Heat'. How about I play 'Woodstock Boogie'?"` [schema with two slots, filled in this example].

```
      {},           # - no payload since zinn doesn’t have to respond
      ’z_silence’:  # silent wait for user to volunteer information:
      {}}           # - no payload since zinn doesn’t have to respond

def init_recover(state):
    state.nonunderstanding = {’task’:None, ’status’:None, ’input’:None}
    return state  # also intended to collect any subsequent dialog
```

Finally, here is a shorter list of `resolve` adapted from the RavenClaw misunderstanding strategies described in [2]:

```
Tasks: (’resolve’, { ... see below ... }, *ARGS)

resolve_strategies = \
    {’confirm’:  # request / obtain confirmation
     {},         # - optional continuation, mandatory recovery task
     ’assume’:   # proceed indicating assumption
     {},         # - this might require a subsequent recovery task
     ’reject’:   # reject, notify and continue
     {},         # - payload can include optional continuation task
     ’escalate’: # express, escalate, recover
     {}}         # - describe the problem and append a recovery task

def init_resolve(state):
    state.misunderstanding = {’task’:None, ’status’:None, ’input’:None}
    return state  # also intended to collect any subsequent dialog
```

Soliciting Targeted Training Data

Dialogs and the associated decision problems are represented by various formal models including finite state machines, state chard diagrams, probabilistic graphical models, Markov decision processes, partially observable Markov decision processes, hierarchical task networks, and temporal logics among others [10, 15, 14, 23]. All those mentioned are more or less up to the task with varying degrees of augmentation. Most of them have been used as the target representation in approaches to learning dialog with reinforcement learning (RL) and partially observable Markov decision process (POMDP) induction being two of the most popular approaches.

As in most learning problems, you have to supply a loss function which is generally cast in terms of a reward signal in the case of RL or a utility function in the case of POMDP learning. What might that mean in the case of dialog? One answer is that Zinn gets negative reward—punishment—when the user terminates the session since that is when Zinn’s has irretrievably failed to sustain the dialog. However, while this signal may seem like the analog of losing a game of chess or backgammon, it’s a poor analogy in this case since users will have many reasons for terminating a session that have nothing to do with Zinn’s prowess as an engaging conversationalist.

A better solution would be to periodically assess how things are going during the dialog, but this would interrupt the conversation flow and the signal will likely be noisy except in the case of very cooperative users or appropriately incentivized volunteers. Using Mechanical Turk to enlist the help of a great many paid confederates might be cost effective when amortized over the life of a successful application, but it would still require a good deal of care to ask the right questions and some additional effort to clean and interpret the data. A better approach might be to launch a limited-function, somewhat-awkward pre-release version of the application and enlist the beta testers in helping to train the system, by letting them interact normally with the application, while providing intermittent feedback in the form of helping the system to recover from errors and malapropisms.

In the case of the approach outlined in this document, imagine that every 1 in a 1000 or 10,000 times Zinn asks some question. There are three types of question Zinn might ask depending on the context:

1. yes-no questions asking if two utterances or utterance-fragments are semantically equivalent or if an utterance is appropriate in a given context, e.g., `"is {} a good way of asking {}"`, `"does it make sense to substitute {} for {} in {}"`, `"would you to say {} in response to my saying {}"`;
2. multiple choice questions asking the user to select the best way of saying something or identify which one of the choices is least like the others, e.g., `"suppose I said {} which of the following would be the best response: {}"`, `"suppose you were me and someone asked you {} if you had to pick one of the following as your reply which one would it be: {}"`;
3. requests for example utterances demonstrating how the user would say something or what the user would answer if she were Zinn and asked a question, e.g., `"how would you say {}"`, `"how would you have answered {}"`, `"can you rephrase {}"`, `"what does {} mean"`, `"when you say {} does that mean the same as {}"`, `"what would you say to exit gracefully from this situation"`;

Answers to such questions would help Zinn to evaluate what it said, train classifiers to categorize what the user said, and enhance its language understanding and generation capabilities by learning to interpret what the uses wants, better communicate what Zinn needs to say, and improving its vocabulary and overall linguistic competence. This discussion just begins the exploration of how we might program Zinn to learn from its conversational partners. Appendix A provides some more ideas on how we might proceed to implement such a capability.

Figuring out what and when Zinn should communicate to the user in these side conversations will require a small team consisting of user-experience experts, linguists schooled in discourse analysis and speech act theory, psychologists adept at experimental design, and machine learning experts well versed in solving NLP problems. Some of these same experts will be important in designing the limited-function, pre-release version of the target application. This is just another step toward building chatterbots that can hold up their end of a conversation with a human. If successful—and I believe it will be if we assemble a talented team—then Zinn will enhance several existing products, enable new applications that would not have been possible otherwise, and help to bootstrap more ambitious projects like SmallTalk by providing a platform on which to deliver targeted training data.

References

[1] Dan Bohus. *Error Awareness and Recovery in Conversational Spoken Language Interfaces*. PhD thesis, Carnegie Mellon University, 2007.

[2] Dan Bohus and Alexander I. Rudnicky. Error handling in the ravenclaw dialog management architecture. In *In Proceedings of the Human Language Technology Conference Conference on Empirical Methods in Natural Language Processing*, 2005.

[3] Dan Bohus and Alexander I. Rudnicky. The RavenClaw dialogue management framework: architecture and systems. *Computer Speech & Language*, 23:332–361, 2009.

[4] Ananlada Chotimongkol and Alexander I. Rudnicky. Acquiring domain-specific dialog information from task-oriented human-human interaction through an unsupervised learning. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 955–964. Association for Computational Linguistics, 2011.

[5] Thomas Dean, James Allen, and Yiannis Aloimonos. *Artificial Intelligence: Theory and Practice*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[6] Sergio Guadarrama, Lorenzo Riano, Dave Golland, Daniel Gouhring, Yangqing Jia, Dan Klein, Pieter Abbeel, and Trevor Darrell. Grounding spatial relations for human-robot interaction. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1640–1647, 2013.

[7] Matthew Henderson. *Recovering From Errors in Conversational Dialogue Systems*. PhD thesis, University of Edinburgh, 2011.

[8] Percy Liang, Michael I. Jordan, and Dan Klein. Learning semantic correspondences with less supervision. In Keh-Yih Su, Jian Su, and Janyce Wiebe, editors, *Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing*, pages 91–99, 2009.

[9] Percy Liang, Michael I. Jordan, and Dan Klein. Learning dependency-based compositional semantics. *Computational Linguistics*, 39:389–446, 2013.

[10] Diane J. Litman and James F. Allen. A plan recognition model for subdialogues in conversations. *Cognitive Science*, 11:163–200, 1987.

[11] Susan McRoy and Graeme Hirst. Misunderstanding and the negotiation of meaning. Technical report, AAAI Technical Report FS-93-05, 1993.

[12] Susan W. McRoy. Misunderstanding and the negotiation of meaning using abduction. *Knowledge-Based Systems*, 8:126–134, 1995.

[13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Second edition, Prentice Hall, Upper Saddle River, NJ, 2003.

[14] Jost Schatzmann, Karl Weilhammer, Matt Stuttle, and Steve Young. A survey of statistical user simulation techniques for reinforcement-learning of dialogue management strategies. *The Knowledge Engineering Review*, 21:97–126, 2006.

[15] Satinder P. Singh, Michael J. Kearns, Diane J. Litman, and Marilyn A. Walker. Reinforcement learning for spoken dialogue systems. In *Advances in Neural Information Processing Systems 12*, pages 956–962, 1999.

[16] J.M. Siskind. Grounding language in perception. *Artificial Intelligence Review*, 8:1995, 371-391.

[17] J.M. Siskind. A computational study of cross-situational techniques for learning word-to-meaning mappings. *Cognition*, 61:1996, 39-91.

[18] Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, Stroudsburg, PA, USA, 2013. Association for Computational Linguistics.

[19] Richard Socher, Eric H. Huang, Jeffrey Pennington, Andrew Y. Ng, and Christopher D. Manning. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 801–809. MIT Press, 2011.

[20] Richard Socher, Brody Huval, Christopher D. Manning, and Andrew Y. Ng. Semantic compositionality through recursive matrix-vector spaces. In *Proceedings of the 2012 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2012.

[21] Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642. Association for Computational Linguistics, Stroudsburg, PA, USA, 2013.

[22] Marilyn A. Walker, Diane J. Litman, Candace A. Kamm, and Alicia Abella. PARADISE: A framework for evaluating spoken dialogue agents. In *35th Annual Meeting of the Association for Computational Linguistics*, pages 271–280, 1997.

[23] S. Young, M. Gasic, B. Thomson, and J.D. Williams. POMDP-based statistical spoken dialog systems: A review. *Proceedings of the IEEE*, 101:1160–1179, 2013.

Appendix A. Collaborative Understanding

We've already seen examples of how to extract potentially useful training data from the user as a natural part of collaborative understanding, e.g., `WHEN_YOU_SAY "$input" IS_WAY_SAY "play $music"?`. We also routinely get distal training when the user confirms our expectations either explicitly or implicitly. Any approach will likely involve more of the same. From a user-interface perspective, the trick will be to interject questions intended gather training data at natural opportunities in the dialog, e.g., when a request for confirmation facilitates the current task of, say, finding out what the user would like to hear, or when there has been an obvious misunderstanding and by answering a question the user helps to ensure that Zinn won't make the same or similar mistake in the future. I imagine that there will be more than enough such opportunities if we able to draw on as many users as we hope to attract. Since the natural opportunities will involve ambiguities and misunderstandings, we will also be able to collect "hard" negative examples corresponding to utterances on which Zinn failed.

Here are some more ways of soliciting useful information including several variations on earlier mentioned one that Bill provided: `Is $this the best way to say $that?`, `Which of $these is the best way to say $that?`, `Can you say $this in reply to $that?`, `Does it make sense to say $this in response to $that?`, `Which of $these is the least like the others?`, `If someone tells me $this does that mean I can say $that?`, `Does $this mean roughly the same as $that?` `Is $this a good analogy for $that?`, `Does` X `relate to` Y `in the same way that` W `relates to` Z`?`[3] The next question concerns what the schema variables, `$this`, `$that`, etc, are bound to. Complete short sentences are obvious candidates. These could be user or Zinn utterances, or paraphrases of the same. But I expect sentence fragments, e.g., verb and noun phrases, will be just as important if not more so when we get to working with more verbose user input as we might expect for a speech-input application. The sentence `Why`

---

[3] For more of these idiomatic ways of posting questions about language see this list of GMAT idioms, i.e., ways of phrasing questions that GMAT uses regularly in its tests. I'm not really recommending these particular idioms, rather I'm simply pointing out that others have thought about this and we can probably take advantage of what they have learned.

`don't you play the third cut included with the CD` is potentially baffling to Zinn in several ways. Zinn might ask one of the following: `When you say "third cut" do you mean "third song"?` or `Does "included with the CD" mean roughly the same as "on the album"?` but probably not `When you say "Why don't you play the third cut included with the CD" do you mean "Play the first song on the album"` unless Zinn was reasonably sure of all the pieces.

If we're to focus on smaller pieces of text than whole sentences, we'll need a good part-of-speech tagger and phrase-structure parser—what Jonni Kanerva was calling a *constituency parser* in Friday's Descartes-Zinn meeting—to identify phase boundaries and select text. This involves some relatively heavy duty NLP technology but I suspect we can get some value by using such tools sparingly and without going as far as Descartes does in mapping surface text to its underlying semantic frame language and back. Speaking of NLP technology, having extracted a bunch of phrases all of which ostensibly mean `song on the album`, how are we going to generalize? Here again, judicious use of tools borrowed from Brain and Descartes, e.g., the LSTM-plus-MCR technology that Lukasz Kaiser described Friday—if you're not familiar with the term Modular Composite Representation, see my introduction from last month.

We're not interested in learning to comprehend arbitrary topics and language, and so I started to write down the sort of knowledge that the Zinn 'Play Music' application needs to carry out its job. I broke it down into four categories characterized by four questions that Zinn might "ask of itself" as it were. The are as follows:

- What can I do? — aside from idle chitchat, there's no sense talking about anything unless Zinn can do something about it. This dismisses a large fraction of natural discourse, but I think we're fine if Zinn is an idiot savant as long as it provides a useful service in an engaging manner that sustains user interest;

  > PLAY — play a song CHOOSE — select a song from {`artist,album,genre`}, choose an artist from {`list_of_artists`}, etc.
  > ASK — what's your favorite {`artist,album,genre`}, would like me to play another selection from {`artist,album,genre`}, etc.
  > INFORM — here's song from {`artist,album`}, i'll play another {`genre`} song, i understand you don't like {`artist,album,song,genre`}, etc.

- What can I know? — again, pragmatically speaking, Zinn is limited to a pretty simple ontology including artists, albums, songs, genres, a few basic relationships, a limited conception of time events, perhaps a superficial knowledge of human activities and moods, and the notion of user preferences;

  > RELATIONSHIPS — `song [by] artist`, `song [in] genre`, `song [on] album`, `genre [has] list_of_artists`, `artist [has] list_of_genres`, etc.
  > PREFERENCES — what —> {`list_of_artists, list_of_genres`}, when —> {`current_activity, how_often_play, current_mood, time_of_day`}, etc.

- What should I do? — of all the things Zinn is capable of at any given times, exactly what should it do now—including the option of asking questions to collect information, e.g., preferences, in order to enable future planning and decision making;

  > GOALS — what do you like to hear, when do you like to hear it, what do you want to hear now, what is your current {`activity, mood`}, etc.

- How can I do it? — not really an issue yet, but one can imagine having a choice to play a song from the user's locker, from a YouTube music video, from a shared list of favorites, from the Play Store if it is free or there's an available sample excerpt from a recording on sale at Google Play or iTunes;

As for related work I asked Gary Marcus a professor at NYU who did his PhD work with Steven Pinker it he knew of any credible computational theories of knowledge acquisition. And he replied disparagingly of the current state of affairs in developmental linguistics. I did find some interesting papers from Ray Mooneys language learning group at UT Austin: (URL), and I took a look at Dan Kleins Stanford PhD supervised by Chris Manning and Daphne Koller (PDF) as well as his more recent work on different aspects of language acquisition [9, 6, 8].

Dan's work is primarily about learning syntax and I found Richard Socher's thesis (PDF) more useful in thinking about the structure of language—as well as visual scenes [18, 19, 20, 21]. I've always found Jeff Siskind's work on grounding language in perception [17] (PDF) and modeling lexical acquisition [16] (PDF) intriguing, and Kevin Murphy indicated that Jeff has some new work along similar lines that might be worth looking into—Kevin is running AAAI Spring or Fall Symposium on a related topic.