POMDP-Based Interaction and Interactive Natural Language Grounding with a NAO Robot

by

Maxwell B. Forbes

Submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of

Bachelor of Science with College Honors

at the

UNIVERSITY OF WASHINGTON

December 2013

Author .....
Department of Computer Science and Engineering
December 13, 2013

Certified by .....
Rajesh P. N. Rao
Associate Professor
Thesis Supervisor

POMDP-Based Interaction and Interactive Natural Language Grounding with a NAO Robot

by

Maxwell B. Forbes

Submitted to the Department of Computer Science and Engineering
on December 13, 2013, in partial fulfillment of the
requirements for the degree of
Bachelor of Science with College Honors

Abstract

In this thesis, I investigated POMDP-based interaction and the use of sensors for natural language grounding in the NAO robot. I developed several tools throughout to aid in accelerating the experimentation pipeline. I present an overview of my work as well as preliminary results from experiments.

Thesis Supervisor: Rajesh P. N. Rao
Title: Associate Professor

Acknowledgments

The author gratefully acknowledges the continued guidance and support of Rajesh P. N. Rao, Luke Zettlemoyer, and Maya Cakmak. He also would like to extend a huge thank you to Michael Jae-Yoon Chung for his persistent mentoring and thoughtful advice throughout this thesis work.

Contents

List of Figures

Chapter 1

Introduction

The motivating question for this thesis was: how can we use multimodal interaction with a NAO robot for grounded language acquisition, and in which scenarios will the results demonstrate interesting applications?

In exploring this question, the work in this thesis spanned a number of topics.

The first central topic involved investigating the feasibility and usefulness of applying POMDPs to interactive scenarios with the NAO robot. Chapters two and three consider this topic. Chapter two involves building an end-to-end system that demonstrates the simple "Voicemail" POMDP problem[1], and then augmenting this example with multimodal input. Chapter three continues this work with the specification of a class of state-estimation problems relevant to our domain. This motivated the construction of a framework for: (a) the specification of such problems in a compressed representation, (b) the compilation to a POMDP format, and (c) the integration with an end-to-end system.

Chapter four switches from the 'interaction' task to the 'learning' task. Whereas the POMDP work in chapters two and three assume learned probability distributions of the NAO's observation capabilities, chapter four investigates the learning of such probability distributions. It looks at questions such as: which sensors of the NAO are useful for gathering data that can be utilized for grounding natural language? How

[1]The "Voicemail" problem is the "Hello World" of POMDP problems, known usually as the "Tiger Room" problem, but applied to the domain of spoken dialog systems by [23].

do the sensors behave under different conditions? What issues arise in realistically training classifiers interactively with the NAO?

Chapter five gives a broader look at the results of chapters two through four and discusses the future directions that are under consideration for this work.

Chapter 2

POMDP-Based Interaction Basics

2.1 Motivation

2.1.1 POMDPs in Spoken Dialog Systems

Partially Observable Markov Decision Processes (POMDPs) have gained recent popularity in the application of spoken dialog systems[23, 3, 22, 24, 5]. POMDPs provide several advantages in such systems, including modeling uncertainty in user speech and intention as well as reducing the number of parameters that need to be tuned to produce useful behavior. (This work assumes the reader is familiar with POMDPs. Below, we illustrate not POMDPs in general, but a specific application of POMDPs to an interactive spoken dialog problem. For a review of POMDPs, please see [9].)

First of all, interacting with a user via speech is a natural source of uncertainty because decoding speech is imperfect. Thus, the act of recording and interpreting speech alone means there is uncertainty in what the user said. POMDPs can naturally model this uncertainty in their observation function by assigning a probability distribution over possible observations when input is received from the user. This probability can be tuned based on information about the accuracy of the speech decoder. For example, if given a certain vocabulary, the decoder reports the spoken word with an accuracy of 80%, the POMDP model can give observations that involve speech a probability of 0.8 of being correct, distributing the remaining 0.2 over other

observations. (How the remaining probability is distributed might depend on other aspects of the model.)

In addition to uncertainty about what the user in a spoken dialog system actually said, there is an inherent uncertainty in what the user actually wants, or the "state" of the user. POMDPs can model this naturally as well by maintaining a list of all possible user intentions as states in the POMDP, and assuming that the user maintains exactly one state. The system does not know which state this is, and it is the 'goal' of the system to find out the correct state. To do so, the system will keep a probability distribution over all of the possible states. This is called the *belief*. As the system receives more information about its environment–for example, by decoding voice commands provided by the user–it will update its belief distribution to take into account the new data. Once it is confident enough in one of its states, which is to say that the belief of that state has crossed a certain threshold, the system will 'decide' that it knows the user's intent and proceed accordingly.

Perhaps the true power of the POMDPs use in spoken dialog systems comes in designing these thresholds. This is because one does not have to set the thresholds directly; they are set implicitly as part of the reward function of the POMDP. In the reward function, one can specify how much the system should be 'rewarded' or 'penalized' for choosing certain actions depending on the true state. For example, taking some action that matches the user's intentions would receive a positive reward, whereas taking an action that does not align with the user's desires would yield a negative reward. If there is an action that queries the user in an attempt to gather more data, this might receive a slightly negative reward; the system gains the advantage of receiving more data, but the small penalty for doing so discourages it from asking questions forever.

By tuning these reward values, we have more coarse-grained parameters through which we can control the behavior of our system. Solving the POMDP produces a policy that the interactive system will follow that dictates the thresholds in its belief distribution. For example, if the penalty for querying the user was low, and the penalty for taking a 'wrong' action was high, the resulting policy would encourage the

system to ask many questions of the user in order to be very 'sure' of its underlying state before taking a 'deciding' action. Conversely, charging the system a high penalty for querying the user would result in lower thresholds and a system that 'gambles,' making 'decision' actions after fewer queries.

2.1.2 Incorporating Sensor Data

We were interested in an interactive system which included not only speech input from the user, but also other sensor inputs. Specifically, we wanted to use an Aldebaran NAO, a humanoid robot that includes simple speech recognition and text-to-speech capabilities as well as additional sensors, including pressure sensors from its joints, sonar data, vision data, and more. We were interested in incorporating additional sensor data because it expands the opportunities for user interaction by incorporating additional aspects of the environment, which will presumably be common to the robot and the user.

The choice of a POMDP model easily allows for the incorporation of sensor data into a spoken dialog system: reading from sensors can be classified as additional actions; the sensor data that is received are used as observations; and the underlying states of the system can be modified to include both user goals and properties of the environment. The intersection of these spaces, which includes the user, robot, and their environment, provides a wide range of possible tasks, some of which we began to explore in this project.

2.2 Tasks

2.2.1 The "Voicemail" problem

The first task we addressed was a modified version of the classic POMDP problem, the "Tiger Room Problem," as described in [23] as an adaptation of this problem to the domain of a spoken dialog system. The problem approximates a voicemail system in which the user is attempting to either 'save' or 'delete' a message, and the system

attempts to determine which by querying the user with the *ask* action before making a decision to either *save* or *delete* the message.

For this task, the goal was to recreate the dialog as outlined in [23] but with a real-time interactive system using the NAO. To begin, this would involve integrating several foundational pieces of a system that uses POMDPs, including:

1. specifying the problem as a POMDP environment

2. integrating a POMDP solver to generate a policy for the specified POMDP

In order to bring the POMDP environment and policy to life, there were several other components that would need construction, including:

1. writing a parser to load a POMDP file[2] into our working language (Python)

2. writing a parser to load a generated POMDP policy file into Python

3. building an interactive system which utilizes the POMDP environment and policy, notably in choosing optimal actions and updating the belief as data is received

4. integrating NAO speech recognition (input) and text-to-speech (output)

2.2.2 Incorporating Sensor Data

The next task we addressed was to construct a POMDP problem similar to the "Voicemail" problem but that would incorporate sensor data from the NAO. We wanted to make a problem where the optimal policy would involve the system choosing some combination of user input as well as sensor input, depending on the belief.

The goal was that this behavior would 'fall out' of the POMDP definition, so to speak, simply by adjusting the reward function, and not by hardcoding threshold values. In this we hoped a POMDP would be effective in generating interactive policies without too much fine-tuning.

2.3 Results

2.3.1 The "Voicemail" problem

We were able to successfully recreate the interaction as specified in [23], matching the reported belief distribution when we took the same path in the dialog. We were able to use our own speech as input to the NAO, and it would respond with synthesized speech output. We used APPL[11] as our POMDP solver, and implemented the other components mentioned in the task.

2.3.2 Incorporating Sensor Data

We created a POMDP environment where the goal was to determine whether an object is 'heavy' or 'light'. There are two states: {`heavy, light`}. They are both assumed to be equally likely, so the prior belief over these states is set uniformly to $[0.5, 0.5]$. There are four actions: {`askHuman, pickupObject, outputHeavy, outputLight`}. The actions `askHuman` and `pickupObject` acquire input from the user and sensors, respectively, and the final two actions are "terminal" in that they end the current interactive round. From asking the user, two observations are possible: `hearHeavy` and `hearLight`. The sensors are also discretized into a binary input, so its observations may be one of `feelHeavy` or `feelLight`.

For the observation and reward functions, we wished to model the scenario that asking a user is cheap but not very reliable, whereas picking up the object is expensive but also quite accurate. We assigned humans an accuracy of 60% with a penalty of -1, and the sensors an accuracy of 80% with a penalty of -3. The reward for choosing correctly was 10, while the penalty for choosing incorrectly was -30.

The thresholds that were generated from the optimal policy are as follows:

The hand-tuning required to generate this policy was minimal. We tried two different reward values[1] before coming up with an environment whose policy took

[1] We had originally set the `pickupObject` action a penalty of -2, and an incorrect output a penalty of -20.

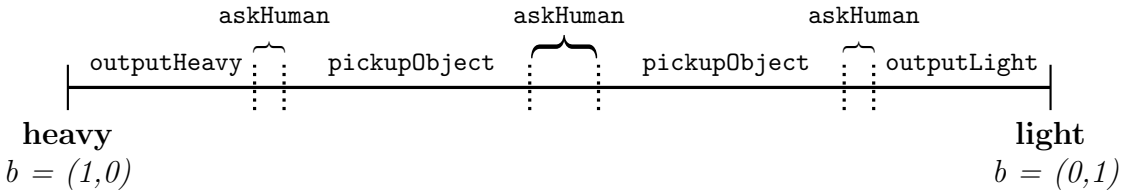

Figure 2-1: The learned belief function for the "light" vs "heavy" POMDP problem.

a	*s'*	o'			
		hearHeavy	*hearLight*	*feelHeavy*	*feelLight*
askHuman	*heavy*	0.6	0.4	0	0
	light	0.4	0.6	0	0
pickupObject	*heavy*	0	0	0.8	0.2
	light	0	0	0.2	0.8
outputHeavy	*heavy*	0.5	0.5	0.5	0.5
	light	0.5	0.5	0.5	0.5
outputLight	*heavy*	0.5	0.5	0.5	0.5
	light	0.5	0.5	0.5	0.5

Figure 2-2: The observation function $p(o'|a, s')$ for our "light" vs "heavy" POMDP problem.

a	*s*	*s'*	
		heavy	*light*
askHuman	*heavy*	1	0
	light	0	1
pickupObject	*heavy*	1	0
	light	0	1
outputHeavy	*heavy*	0.5	0.5
	light	0.5	0.5
outputLight	*heavy*	0.5	0.5
	light	0.5	0.5

Figure 2-3: The transition function $p(s'|a, s)$ for our "light" vs "heavy" POMDP problem.

a	*s*	
	heavy	*light*
askHuman	-1	-1
pickupObject	-3	-3
outputHeavy	10	-30
outputLight	-30	10

Figure 2-4: The reward function $r(a, s)$ for our "light" vs "heavy" POMDP problem.

each action at least at some point in the belief space.

On the other hand, the policy itself provides more complex behavior. We can see that this policy prefers to take the cheaper `askHuman` action in two cases. First, when it has a uniform belief over its states, it needs at least one human query and one sensor query to decide with confidence; two `askHuman`s would not be sufficient, and two `pickupObject`s would have a higher penalty. Secondly, it will execute askHuman when it is almost sure of its state, but needs a slight boost; here, the lower cost of `askHuman` is preferable. The `pickupObject` action is taken only when this action, if confirming the belief that the system currently prefers, will push it over the threshold to one output.[2]

Though the values we chose here were arbitrary, by taking measurements of real-world values in such interactions—for example, the delay when gathering each type of data, the time wasted when a wrong answer is chosen, or the accuracy of humans in classifying objects—optimal policies that emerged would satisfy actual needs.

[2]These hypotheses about the behavior of the agent using this policy were confirmed by manually testing the interactive system with this environment and policy.

Chapter 3

POMDP State Estimation

3.1 Task

The task for this section of the thesis work was to design and implement a general approach to allow the NAO to discover state. There were two main components that made this interesting:

1. The "state" of the NAO should be with respect to multiple properties, henceforth called *state dimensions*.

2. The observations the NAO receives should be from multiple sources—including natural language input—henceforth called *observation categories*.

As an example, the NAO might be trying to discover how heavy an object is that it is holding and how bright the room is that it is in (here it would have 2 *state dimensions*). Its inputs could be words input from the user, which would provide evidence for both state dimensions, as well as weight sensor and light sensor readings, which would only provide evidence for their respective *state dimensions*. Hence, there would be 3 *observation categories*.

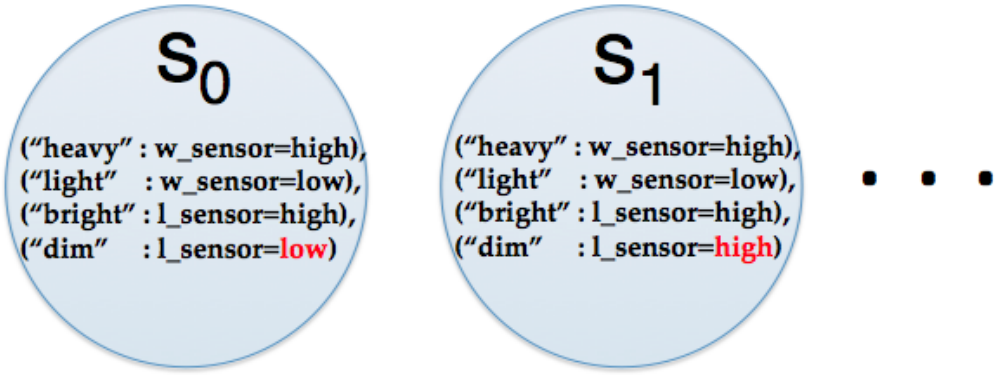

Figure 3-1: A conceptual image of successive states encoding all words in the vocabulary (4 of them) with possible sensor mappings in sensor space (binary valued). Note that here we have pre-determined which words map to which sensors; removing this assumption leads to more states.

3.2 First Approach: State Space Learning

3.2.1 Design

The first model we designed was one where learning would be defined as discovering the correct grounding (meaning) for all words in the vocabulary. These groundings would be in terms of all *state dimensions* additional to natural language input, so observations from all *observation categories* would be necessary to identify the correct state and 'learn' the meaning of all words in the vocabulary. See Figure 3-1 for a visual aid.

3.2.2 Implementation

As proof-of-concept, we created by hand POMDP environments that used this approach. The state list from one such environment is listed in Figure 3-2, and the observations and observation function for the information gathering action is shown in Figure 3-3.

3.2.3 Results

This approach provided unsatisfying results in several respects:

```
states: vHeavy-oW1_vLight-oW1   vHeavy-oW1_vLight-oW2   vHeavy-oW1_vLight-oW3
        vHeavy-oW2_vLight-oW1   vHeavy-oW2_vLight-oW2   vHeavy-oW2_vLight-oW3
        vHeavy-oW3_vLight-oW1   vHeavy-oW3_vLight-oW2   vHeavy-oW3_vLight-oW3
```

Figure 3-2: The actions of a 'state space learning' POMDP environment, with additional formatting for clarity. A single entry, for example, `vHeavy-oW3_vLight-oW2`, encodes the idea that the vocabulary word *heavy* is associated with the weight sensor value `3`, while the word *light* is associated with the weight sensor value `2`

observations: vHeavy-oW1	O: getInfo
vHeavy-oW2	0.4 0.05 0.05 0.4 0.05 0.05
vHeavy-oW3	0.4 0.05 0.05 0.05 0.4 0.05
vLight-oW1	0.4 0.05 0.05 0.05 0.05 0.4
vLight-oW2	0.05 0.4 0.05 0.4 0.05 0.05
vLight-oW3	0.05 0.4 0.05 0.05 0.4 0.05
	0.05 0.4 0.05 0.05 0.05 0.4
	0.05 0.05 0.4 0.4 0.05 0.05
	0.05 0.05 0.4 0.05 0.4 0.05
	0.05 0.05 0.4 0.05 0.05 0.4

Figure 3-3: The observations and sample observation function for a 'state space learning' POMDP environment, with additional formatting for clarity. Observations come in (vocabulary word, sensor_1, sensor_2, ..., sensor_n) tuples; in this case we have only one sensor. The observation function has states (Figure 3-2) as rows and observations as columns, and serves to weight states in a way that matches with whichever observation came in. States that match the given observation are given observation probability 0.4, whereas those that don't are given probability 0.05. Note that rows' probabilities must sum to 1.

1. The state space grows too quickly. If we assume we know which vocabulary words correspond to which *state dimensions*, then the size of the state space is $\mathcal{O}(|S_d|^{|D||V_d|})$, where we have D state dimensions and S_d and V_d are the sets of sub-states and vocabulary words associated with *state dimension* d, respectively . If we remove this assumption, then there are $|S|^{|V|}$ states, where S and V are the sets of all *state dimensions* and vocabulary words, respectively. The APPL POMDP solver struggled with only $\sim$ 10 states in this toy scenario; growing the vocabulary size would be nearly unthinkable.

2. The interactive program cannot directly address the sparse observation problem (curse of dimensionality) that comes with a large state space. That is because the action available to it is `getInfo`. If it continues to record observations that do not define some *state dimension*—for example, if the user never talks about how much weight is on the robot—it cannot take any actions to fill in that space.

After several attempts to use the APPL POMDP solver to generate policies for environments framed by this approach, it became apparent that heavier machinery would be needed before this approach would become feasable. There existed a very simple underlying logic to the state space, but such a logic was not captured by the "pure" POMDP environment. Instead, these environments appared to the solver as large and increasingly-intractable problems. We decided to begin a different approach to address this task, which we will call here the 'generative approach.'

3.3 Second Approach: Generative Approach

3.3.1 Design

Instead of encoding language to sensor mappings in the states, this approach assumes that the robot itself is in a single hidden state. This state then is the 'source' (it 'generates') observations, which we assume are conditionally independent given the state. See Figure 3-4 for a graphical model of this.

One key factor here is that because of our conditional independence assumption, we can write $p(word, sensor_1, sensor_2, ..., sensor_n|state) = p(word|state)p(sensor_1|state) \cdot \cdot \cdot p(sensor_n|state)$. Now, if we collect probabilities of different words and sensors for a state, we can then combine them to estimate the joint probability of the state.

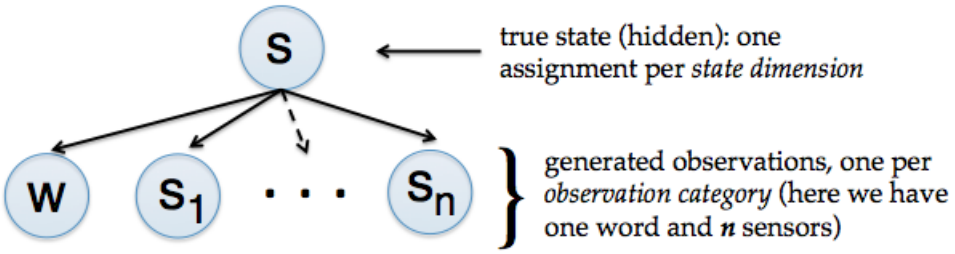

Figure 3-4: The generative model—a Naive Bayes. The true state **s** is unknown and fixed, and is discovered by getting observations through successive rounds of the POMDP.

3.3.2 Implementation

Formulation

To realize this approach, we began by writing POMDP environments by hand and manually computing probabilities and weights in order to devise a general scheme for how this approach could be implemented. An example of these calculations appears in Figure 3-5. Through this, we nailed down concrete rules for environments to implement this approach:

1. A state must be a cross product over *state dimensions* in the sense that one state must contain one "choice" per *state dimension*. A human interacting with this system would need to be able to describe any of the available *state dimensions* that the agent is in, so the state representation must accommodate that. For example, if our state dimensions represent room temperature and wall color, and the agent is in a cold room with green walls, the state must represent both of these properties rather than just one of them. This means that if there are s substates per *state dimension*, and there are d *state dimensions*, then there are s^d states in the system.

2. Some *observation categories* will provide information over multiple *state dimensions*, and so must be normalized.

3. One action can be used to get input for each *state dimension*, giving the agent

power over which *state dimensions* are updated in each time step by which action it choses.

POMDP Generator

Computing the observation matrices for multiple *state dimensions* and *observation categories* was very tedious, so we took the rules that we developed above and specified a JSON schema for which we could enter probability distributions over *observation categories* with respect to their relevant *state dimension*, which, along with some basic information about rewards and state discretizations, fully specified an instance of a problem in this approach given the assumptions of this model. We then wrote a POMDP generator which loads a JSON that follows this schema, extracts sub-states and observation classes along with the probability distributions, and creates a valid (solvable by published solvers) POMDP environment.

This POMDP generator makes up the core of our work on this approach. It hardcodes the model's details (such as having only one true state that doesn't change) and allows for concise entering of real (measured) data into a setting for this problem. Its two most important capabilities are with regard to the general requirements of this approach:

- Supports any number of *state dimensions*, and any number of substates per *state dimension* (specified by an integer). The full list of states is auto-generated.
- Supports and number of *observation categories*, any number of observations per *observation category*, and observations can span any number of *state dimensions*, even within the same *observation category*.

3.3.3 Results

Though this approach still leads to exponentially-sized state spaces, it is exponential in dimensions rather than vocabulary, which is much more reasonable. The APPL POMDP solver gives slow but usable performance for computing policies for these environments when done offline.

```
w1_r0 (note only v0-2 change from w0_r0)              non-norm  *0.5   norm
    p(v0|w0_r1) = p(v0|w0) + p(v0|r1) = 0.5 + 0    = 0.5        ->    0.25
    p(v1|w0_r1) = p(v1|w0) + p(v1|r1) = 0.2 + 0    = 0.2        ->    0.1
    p(v2|w0_r1) = p(v2|w0) + p(v2|r1) = 0.3 + 0    = 0.3        ->    0.15
    p(v3|w0_r1) = p(v3|w0) + p(v3|r1) = 0   + 0.7  = 0.7        ->    0.35
    p(v4|w0_r1) = p(v4|w0) + p(v4|r1) = 0   + 0.3  = 0.3        ->    0.15
```

Figure 3-5: This shows an example computation for one *observation category* (v: "vocabulary", i.e. natural language input) for the state `w1_r0`, which means that the `w` *state dimension* is in discretization level (sub-state) 1, and the `r` *state dimension* is in discretization level (sub-state) 0. Here, the first three vocabulary words `v0-v2` describe the `w` substate, and the last two `v3-v4` describe the `r` substate. Because this *observation category* spans multiple *state dimensions*, the probabilities must be normalized to sum to 1 for any given state, as states are a combination of one choice per *state dimension*. We currently perform unweighted normalization for simplicity.

3.4 Application

As a proof-of-concept for our second approach, we collected real data from the NAOs sensors, discretized them and extracted the probability distributions, framed them in our JSON schema, and generated and solved POMDP environments using them. Each of these steps is described in detail in the following sub sections.

3.4.1 Data collection

We collected data from the NAO robot using two different state dimensions: weight, and darkness of room. Figure 3-6 shows some of our test setups.

3.4.2 Data processing and discretization

The next step was to visualize the results of our data collection in order to determine appropriate locations to discretize the continuous space. Note that this process could be done by training a classifier on this data and then determining thresholds, but the data was separable and low dimensional enough that graphing and then eyeballing works for what we hope to achieve. The weight measurements are shown in Figure 3-7, and the darkness measurements are in Figure 3-8.

Figure 3-6: Left: Collecting weight data for the NAO without any load on it, and light data for a bright room. Center: Collecting weight data for the NAO holding a box filled with packing peanuts. Right: Collecting light data for the NAO wearing a helmet.

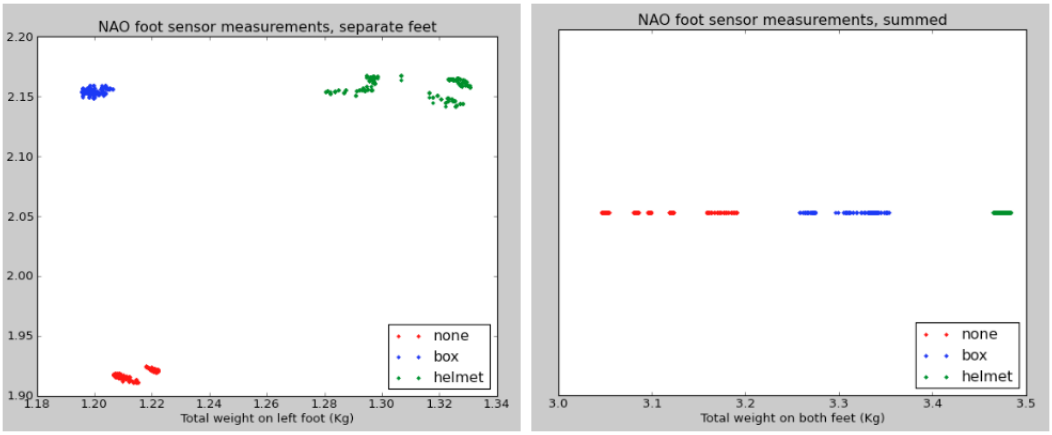

Figure 3-7: We tested the NAO with no weight (red), carrying a box (blue), and wearing a helmet (green), with 500 data points per sample. There are four foot sensors per foot on the NAO; in the left graph, we collected data from each foot (you'll notice the NAO is very unbalanced), and in the right graph, we collected the total weight. Using the 1D data, we can eyeball support vector machines and create classification boundaries at around 3.23 and 3.43 Kg

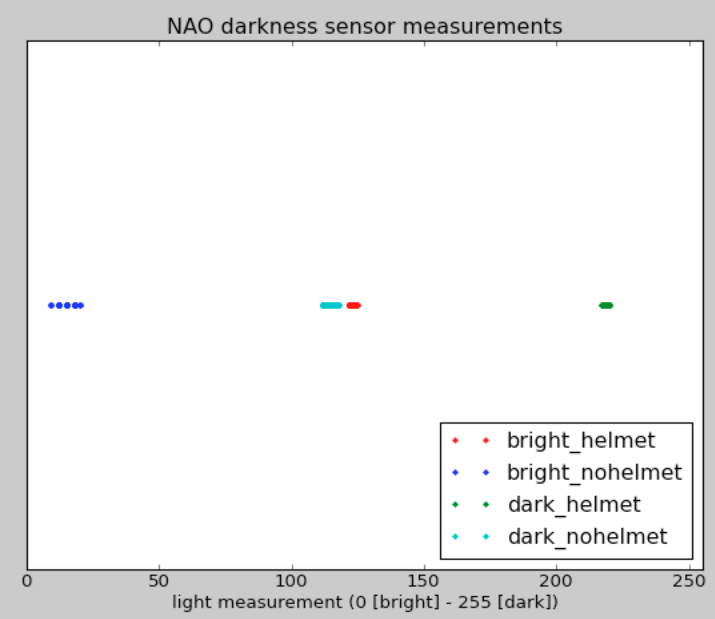

Figure 3-8: The light measurement data is also separable. We modified the environment in two ways: brightness of ambient lighting (artificial lights and blinds), and the NAO wearing the helmet or not. We gather data for all combinations of these and collected 500 data points for each type. We are left with four linearly separable chunks, and can identify where to draw classification lines.

3.4.3 Entry in JSON Spec File

At this point, we are ready to create the JSON spec file that specifies this environment. We will have to make some decisions about our vocabulary how we want to represent error probability margins, and then the rest can be encoded directly.

In the interest of space, we will not reproduce the entire JSON spec and generated POMDP file here. However, as a short example, we will provide the listings for the encoding of states and resulting generations.

For the values that we measured, there are 3 different weight 'states' within the weight *state dimension*, and 4 different light states within the darkness *state dimension*. This would be listed as follows in the JSON file:

```
1 "dimensions": [
2     {
3         "name": "weight",
```

```
        "short": "w",
        "number": 3
    },
    {
        "name": "darkness",
        "short": "d",
        "number": 4
    }
]
```

An observation specifies the *state dimension* to which it is providing data for, its name (this is not used in POMDP generation; it's for the interaction code to recognize it), and an array of values over the possible sub-states of the *state dimension*. For example, if the natural language *observation category* had the word "light," it might be listed as follows:

```
"observations": [
    {"assoc": "w", "name": "light", "values": [0.5, 0.25, 0
        .1]},
    ...
]
```

3.4.4 Generated POMDP File

To avoid listing the entire generated POMDP file, we will continue with the above example and list the generated states:

```
states: w0_d0 w0_d1 w0_d2 w0_d3
        w1_d0 w1_d1 w1_d2 w1_d3
        w2_d0 w2_d1 w2_d2 w2_d3
```

Here we see the 3 weight sub-states and 4 darkness sub-states have been combined to form 12 possible states.

The observation function generated takes the specified JSON observations and maps and normalizes them to influence the generated states whose sub-states they map to. Here is a sample of what a generated observation function might look like:

```
O: get_word
0.075 0.1 0.325 0.35 0.15 0 0 0 0
0.075 0.1 0.325 0.1  0.4  0 0 0 0
0.25  0.1 0.15  0.35 0.15 0 0 0 0
0.25  0.1 0.15  0.1  0.4  0 0 0 0
```

Here we can tell that the first five observations are words, because the other observations are zeroed out (note the observation function is for the `get_word` action).

3.4.5 Solved POMDP File

The solver, though slow, generates a reasonable policy (precision < 1.00) in under 10 minutes.

3.5 Conclusions

The task of constructing a generic system by which POMDP state discovery—though any number of *state dimensions* and *observation categories*—can be made through interaction with natural language seems to be a problem that requires a large number of states. For someone constructing such environments, this means there are several ramifications:

1. Policy computation must be done offline. This also means that the addition of, say, novel words to the vocabulary, cannot happen in an online setting if it means resolving the POMDP. Thus, alternate approaches would be necessary.

2. Though the state space may be exponential, thinking about *in which dimensions* it is exponential can have a large impact on the problem.

3. The specification of these environments themselves is tedious and error prone. It is worth the time and effort to build systems that allow for more specific, convenient specifications and auto-generate the more generalized POMDP specification.

A caveat of either approach taken is that we have assumed that the transition function T in the POMDP is stationary. That is, $T(s, a) = s$ with probability 1.0 $\forall s, a : a$ is non-final action. In real-world situations it is possible that the environment is static and the problem can be formulated this way, but in fact the power of the POMDP is that it *allows* for encoding a changing environment. The system described previously can be thought of as a cross product of several "Voicemail" POMDPs—or, more bluntly, several computationally-expensive decision thresholds.

However, despite the theoretical support of the POMDP for a changing environment, simply "encoding the state changes" in a POMDP environment is nontrivial because the probabilities of all possible transition situations must be determined beforehand. The details of all such situations might be clear once the scenario is unfolding, but beforehand it is not immediately apparent how to specify an environment that is both tractable[1], useful[2], and general[3].

These difficulties suggest that an approach which both learns and demonstrates online may be useful, as a clear separation of the two implies either (a) more information must be collected before beginning than is necessarily convenient, or (b) the "learning" stage must be continually re-entered from the "interactive" stage, which involves continuously computing policies from the POMDP.

[1] i.e. keeping the state space small
[2] i.e. not encoding all transitions in a uniform distribution
[3] i.e. the robot is not constrained to a pre-determined sequence of interactions

Chapter 4

Interactive NAO Experiments: Learning and Demonstration

4.1 Overview

This chapter outlines work from an interaction-first approach towards language grounding with the NAO. Whereas the previous two chapters focused on formulating and testing prototypes of models that could be used to model the interaction and statistics of a system, the focus of the work in this chapter was to look directly at the NAO's sensors and see what systems can be built using them.

For more details on these experiments, including annotated videos of the systems in action, please see the project website at `http://sites.google.com/site/uwaiprojects/`.

4.2 Real Time Monitoring System

Preliminary tests showed that some of the NAO's sensors could begin reading bad values during certain (seemingly unpredictable) conditions. For example, foot weight sensors would occasionally begin reading excessively large values, or record the inverse of changes that affected them (such as putting a ~ 0.3 kg object on the NAO causing a *drop* in the foot sensor reading of ~ 0.3 kg). Thus, in order to conduct

experiments with the NAO's sensors that relied on their accuracy for data collection, it was necessary to first build a monitoring system in order to tell what its sensors were measuring during the experiments.

4.2.1 Design 1: Multiprocess, `pyplot` instances

Because all previous code for modeling (POMDP) and interaction (NAO) work was done in Python, Python was continued for these experiments. This problem was fundamentally multi-threaded in nature, because a monitoring system must record input and display output from many sources, including:

- User input (typing)
- User speech commands (speech recognition)
- Querying multiple robot sensors (foot sensors, sonar readers, ...)
- Activating robot actuators (stand, sit, ...)
- Displaying visualizations of monitored sensors (graphs, numerical readings, ...)

Due to Python's lack of true threading, Python's `multiprocessing` was chosen instead to achieve true concurrency.

Though the NAO's sensor API supports concurrent access, empirical tests showed that the NAO's system uses a single global lock on all of its sensor values, and this locking mechanism favored allowing one thread (or process) to access it for several seconds. This was not suitable for a monitoring system in which multiple observers would query different aspects of the NAO's state, so locking access to the NAO's API was done manually for each `Proxy` (NAO subsystem interface).

Python's `matplotlib.pyplot`[7] library for plotting was used for displaying the results of monitoring. Given the concurrent nature of the code, and the shared use of a single library, each observer process is given its own `pyplot` instance. An outline of the first system attempt is shown in Figure 4-1.

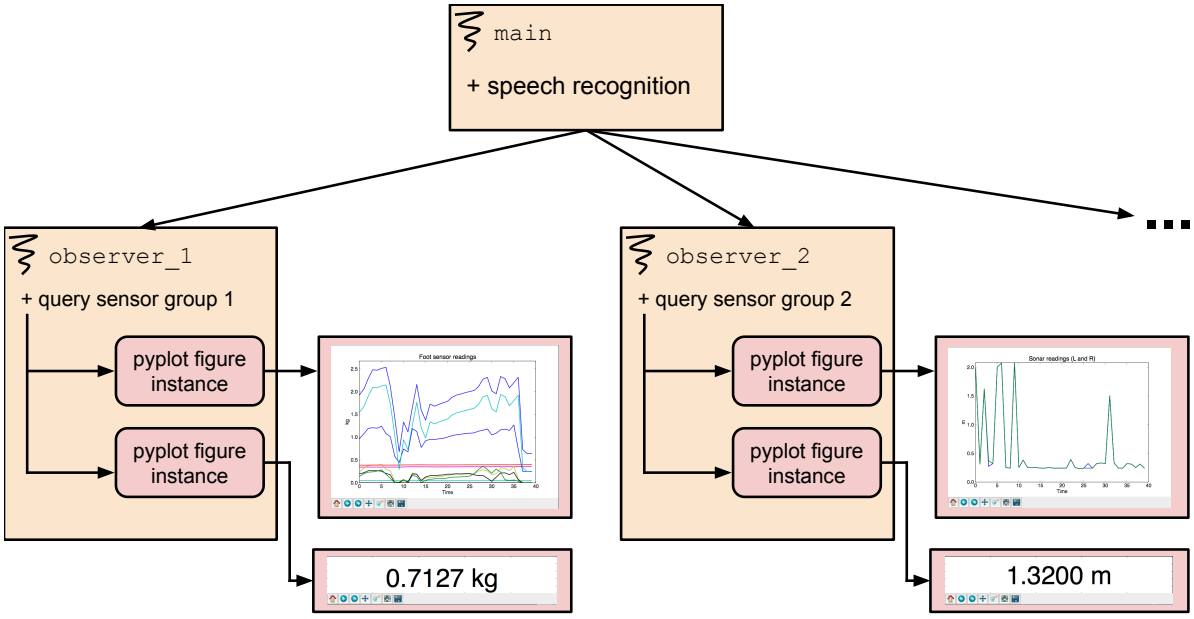

Figure 4-1: A first attempt at implementing a real-time monitoring system for the NAO. In the main process, different observer processes are spun up using Python's `multiprocessing` module. Relevant instructions are recording from the main process' speech recognition module, and multiplexed to the observer processes using `Pipes`. Each observer process is responsible for querying one logical group of the NAO's sensor values, and then rendering this information in a graph using one or more held instances of `pyplot` figure objects.

Unfortunately, this system had a flaw: after all observers correctly queried their sensors and drew their figures once, on the second round of drawing, a system error appeared which caused the Python processes to crash. The problem was related to performing GUI operations on non-main threads running on the Mac OSX system. Because the Mac was essential at this stage for experimental recording (screen capture, voice capture, video capture, video editing), I began work on a different system design.

4.2.2 Design 2: Multiprocess observers, single plotter

After the failings of the last system because of non-main threads drawing to the GUI, I designed a system where each process would again query the state of one group of the NAO's sensors, but the observations would be sent back to the main process for display. The system is outlined in Figure 4-2.

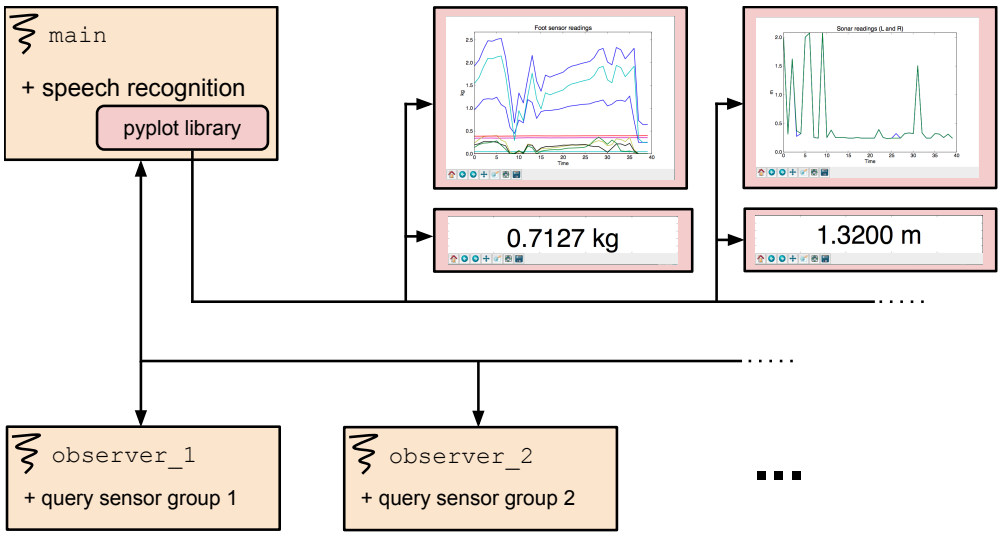

Figure 4-2: A second attempt at implementing a real-time monitoring system for the NAO. In contrast to Figure 4-1, this system uses the additional observer processes only for querying the NAO's sensor states, and performs all plotting in the main thread using calls to the `pyplot` library.

Though this system avoided the flaw of the first system, it also had a crippling problem: re-drawing multiple figures on a single thread is a CPU-bound operation. Once the system attempts to monitor even two sensors, it falls far behind; empirical tests showed the system running over 10 seconds behind real-time when rendering three graphs[1]. The result was unusable for real-time monitoring.

4.2.3 Design 3: Multiprocess, single `pyplot`

It appeared that the current task was blocked: in order to take reasonable measurements with the NAO's sensors, we needed a real-time monitoring system that would be easy to see while standing five or ten feet away, interacting with the NAO. Printing out a stream of numbers would be too small, and Python's leading graphing library was failing: either the system was 10 seconds behind real-time, or it would crash due to a system-dependent bug.

As a final design, I tried reverting to the first approach of having multiple processes

[1]Each sensor is shown by a minimum of two graphs: its immediate history plotted as a line graph, and a larger reading 'drawn' as a number.

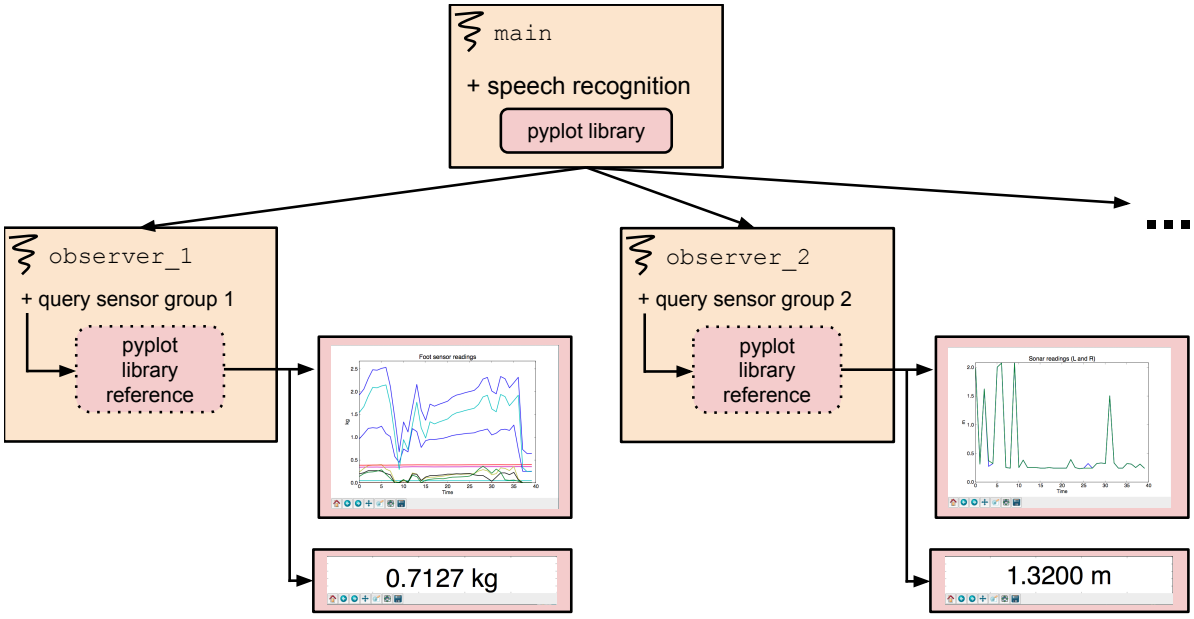

Figure 4-3: A third attempt at implementing a real-time monitoring system for the NAO. This design takes a hybrid approach of Figures 4-1 and 4-2. It performs all observations and plotting in separate processes to avoid the CPU-bound task of drawing the graphs in a single thread, but it shares a single `pyplot` instance across all processes to avoid crashing the system because of GUI/thread-related bugs.

each responsible for drawing the sensors that they monitor. This time, though, I would try using a single `pyplot` instance, created in the main process, in hopes that all processes could share it but the system would believe that all drawing was still happening as a result of the "main thread." The design is shown in Figure 4-3.

This system functioned correctly. Because of the shared use of the `pyplot` library, the GUI drawing operations did not crash, and because of the use of multiple processes for plotting, the graphs update in real-time. Because up to two graphs—one sensor—can be monitored in real-time per thread, the current system running on an eight core machine should support up to sixteen graphs—eight sensors—which should be enough to fill all viable screen real estate. An example of six graphs (three plots) monitoring eleven sensor values in real-time is shown in Figure 4-4.

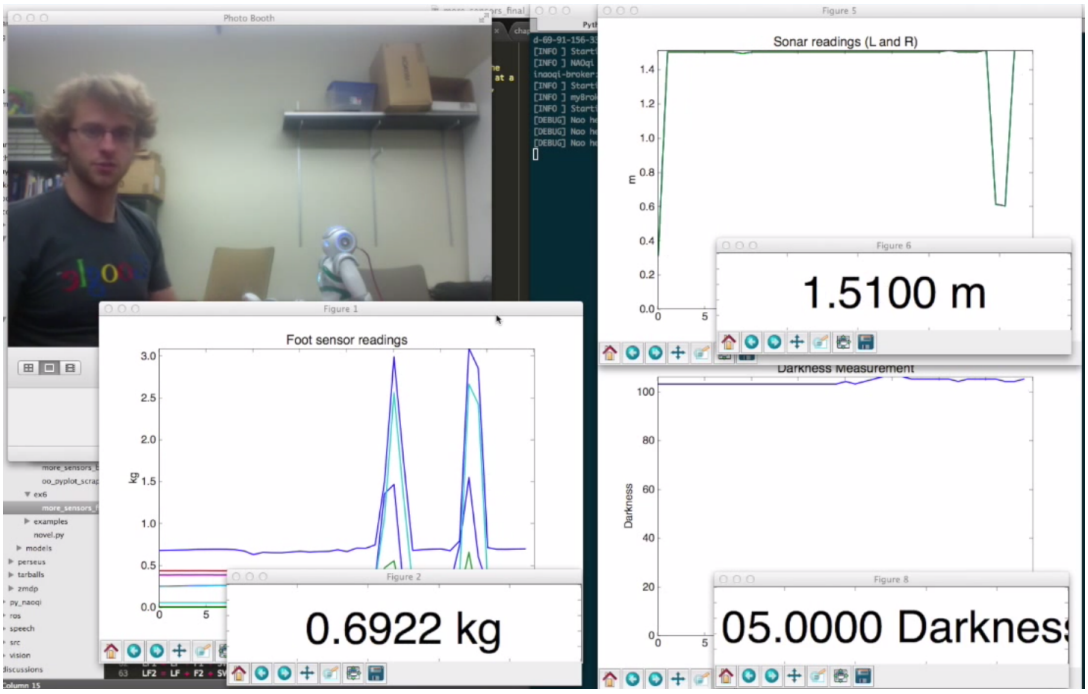

Figure 4-4: A screen capture of the final, working version of the real-time sensor monitoring system for the NAO. **Upper left:** Video feed. **Bottom left:** Foot sensor readings. **Upper right:** Sonar readings. **Bottom right:** Darkness (from camera) readings.

4.3 Foot Sensor Experiments

4.3.1 Overview

With the monitoring system in place, the first sensor domain that I wished to address was foot sensor readings because this would be a straightforward way to measure weight on the NAO. The simple goal of teaching the NAO 'heavy' versus 'light' through examples would be achievable using just the foot sensors.

I programmed the NAO's vocabulary to recognize the class label words "heavy" and "light," as well as two words to control the flow of the experiment: "predict" and "done." I trained a Gaussian Naive Bayes classifier[2] using the spoken class labels with the feature vector consisting purely of the foot sensor readings.

An interaction would run as follows. First, there is a training phase, where the user

[2]From the Python package scikit-learn [15].

places various objects on the NAO's outstretched arms and says "heavy" or "light" to provide it training data. When the user is done doing so, the user says "predict," to enter the prediction phase of the interaction. Then, the user can load however much weight she desires on the NAO, and then say "predict" again. The NAO outputs the most probable explanation (MPE) of its foot sensor readings, speaking the class label it has the highest belief over in the posterior. The user may change weight and say "predict" as many times as they wish. When the interaction is over, the user says "done."

The video proceedings of this experiment are viewable on the project website[3]. A screen capture of this experiment is shown in Figure 4-5.

4.3.2 Results

Given clean data and consistent labeling, this technique works: the Gaussian Naive Bayes classifier can correctly correlate regions of the sensor space with class labels. In practice, however, there were several problems:

1. **The NAO's foot sensors are faulty.** This simple conclusion was the result of multiple experiments which I will not detail in this paper. Briefly, there are several failure modes of the NAO's sensors, in which they will seemingly arbitrarily start out at high and incorrect values, slowly raise or lower their values over time (with no change in the NAO's stance or weight), read values opposite to those provided (removing objects increases the weight), provide varyingly noisy readings (light to severe) in general, and sometimes even jump to "off the chart" readings after which the NAO must be reset. These issues pose a problem for reliably collecting data.

2. **Gaussian Naive Bayes is "too confident."** Rarely would the classifier be anything but 100% sure that one class was the correct label, even if the provided weight was somewhere in the middle of an unseen region between training data[4].

[3]The link is at the beginning of this chapter

[4]We consider addressing this problem by providing a prior over the variance of the model.

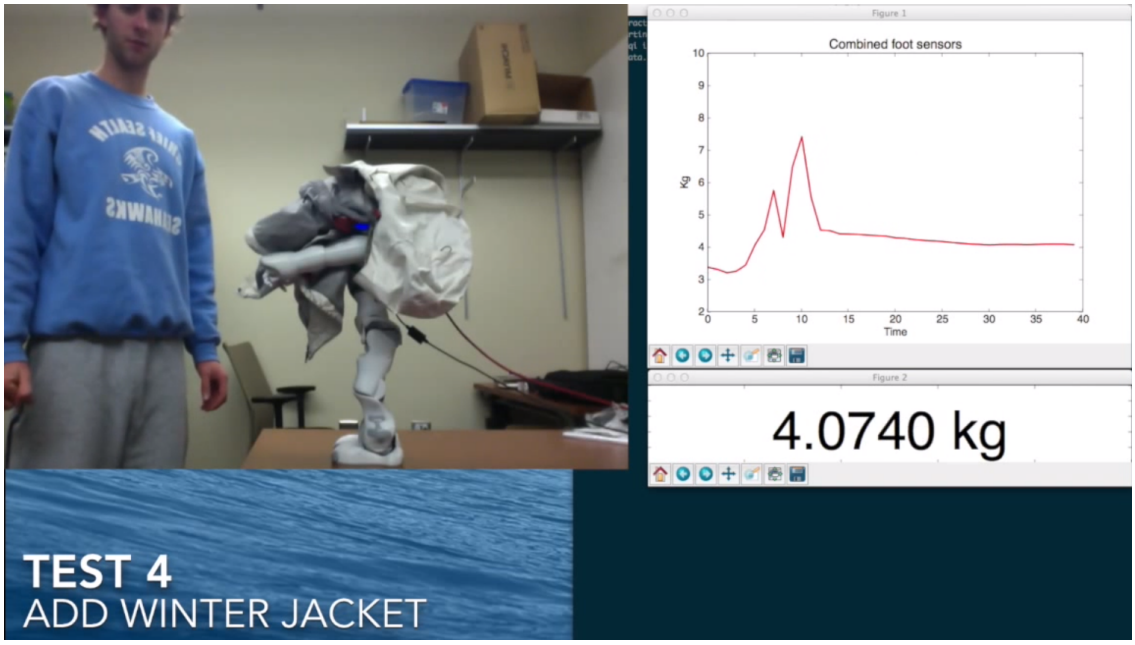

Figure 4-5: A screen capture of the experiments run to test the NAO's foot sensors. **Upper left:** The video feed is shown, with the experimenter on the left and the NAO on the right. The NAO has a winter jacket placed on it to measure its weight difference, and it is connected on its back by a red Ethernet cable and a black power cable. **Bottom left:** the annotation on the video annotates what phase of the experiment is in progress. **Upper right:** One graph from the real-time monitoring system, displaying combined foot sensor readings (there are 8 foot sensors total; this value is their sum), where the most recent observation is on the right side of the graph, and the graph 'moves' leftwards. **Bottom right:** Also from the real time monitoring system, this displays the exact number of the most recent observation of the sensor in question: the combined foot sensors.

3. **The NAO's motors get "hot."** After one or two minutes of experiments, the NAO will say "Motor hot," after which there is a limited time period before it relaxes the stiffness of all of its joints and falls. After this occurs once, future experiments have even less time before it happens again.

Despite these issues, the system functioned in general, and the next step was to broaden the scope and challenge by looking at training more classifiers using more sensors.

4.4 Multi-Class Experiments

4.4.1 Approach

Having achieved success with the NAO predicting "heavy" or "light" based on the single value sum of foot sensor weights, our next goal was to expand this model to include more antonym word pairs, as well as more input sensors.

A simple approach would be to train n different classifiers by feeding in the relevant sensor data to the classifier. So, for example, the "heavy" versus "light" classifier would be given only foot sensor data, the "bright" versus "dark" classifier would be given only brightness measurement data, and so on.

However, this is not as interesting as a scenario where the robot could actually determine which sensors were relevant for each word description pair. In such a setup, each classifier would be given the full range of sensor data whenever a training instance is recorded, and would need to learn to weight relevant features higher. This is the approach that was taken.

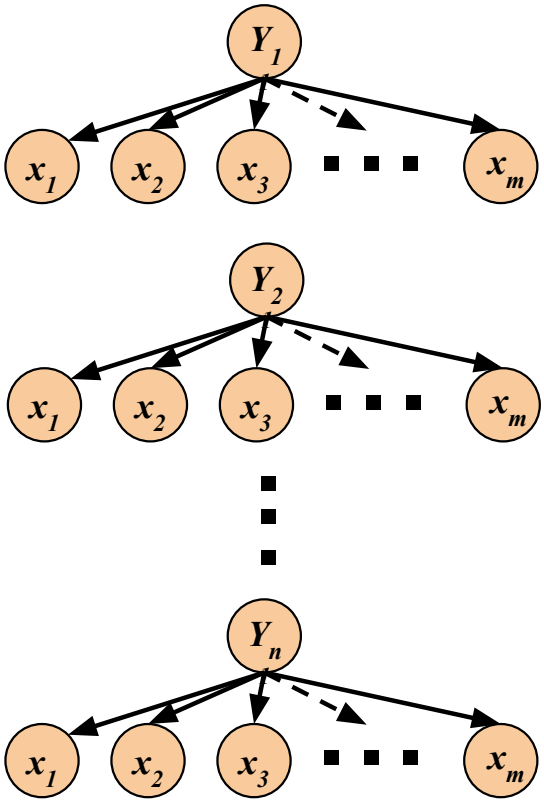

Figure 4-6: Training n different binary classifiers each on the full set of m features. We use Naive Bayes to model the relationship between the human-provided word and all of the robot's sensor features.

4.4.2 Method

To do this, n different Naive Bayes classifiers were to be trained in each interaction, one for each word pair. All m sensor features ($\mathbf{X}$) along with the truth (Y) are provided to a classifier each time a training instance that is is relevant to the classifier is given. A training instance is relevant to a classifier if it contains one of the two words relevant to the classifier. For example, when "light" is said, the values $Y =$ `light` and $\mathbf{X} = x_1, x_2, ..., x_m$ is given to the classifier which determines "light" vs "dark", where x_i is the value of the ith feature extracted from the robot's sensors. A visual representation of this is given in Figure 4-6.

For our first trial of this technique we used $n = 3$, where the three classifiers were:

1. "heavy" vs "light"
2. "bright" vs "dark"
3. "near" vs "far"

For the features, we used $m = 6$, where the six features were:

1. The sum of the four weight sensors on the left foot
2. The sum of the four weight sensors on the right foot
3. The sum of 1. and 2.
4. The left sonar distance reading
5. The right sonar distance reading
6. The darkness level reading from the camera

4.4.3 Results

At the time of writing, we have performed several preliminary experiments using the outlined approach and method, and have seen moderate success.

Given that each classifier is given no prior information about its relevant features, it is important to provide training data that sufficiently covers the feature space so as to dispel fallacious learned correlations. For example, if "heavy" is always trained by being near the robot, it might pick up the proximity of the experimenter with its sonar sensors. In that case, "heavy" examples without a sonar impedance are necessary for accurate learning.

The problem of Naive Bayes classifiers always being 100% sure of the predicted category lingers. We have yet to experiment with seeding the variance of the model to be higher in order to force more uncertain predictions, which would seem more correctly Bayesian.

For our immediate future work, we would like to try adding more sensor features, such as dominant color or size, and word options to match. Further possibilities are discussed in the next chapter.

Chapter 5

Discussion and Extensions

5.1 POMDP

5.1.1 Discussion

The POMDP had a number of strengths and weakness as a component of an integrated robotic system.

The strengths of the POMDP's use in an integrated robotic system are largely its 'advertised' strengths: that given accurate parameters, a policy emerges that is optimal[1]. Rather than pre-programming behaviors to achieve some goal, the POMDP allows an agent behave in a way that maximizes long-term expected reward. For a robotic system with a clear success metric, such as time taken to learn concepts, number of questions asked, or number of mistakes made, the POMDP will demonstrate behavior that optimally achieves this goal, rather than requiring the designers to hand engineer the behavior. As a side effect, the POMDP forces the designer to consider such success metrics and precisely define them, rather than simply guess at desired behavior. Put another way, the POMDP creates a minimum restriction on the formality of a system, which has the benefit of allowing optimal solutions, as well as benefits that come with any formal specification, such as the precision and clarity of all parameters.

[1] Or near-optimal, for point-based solvers.

With that said, using the POMDP in an integrated robotic system does have its drawbacks. When developing a system that is expected to demonstrate certain interesting or useful behavior, the true success metric of the system is not a goal-based reward but rather the behavior. Thus, the designers must craft reward functions that, when a policy is solved, encourage the desired behavior in the robot, which is the kind of reverse engineering that the POMDP was meant to solve in the first place. If a clear success metric is, for example, "seconds taken to learn a concept," and the time of all robot actions are measured and values are set in the reward function accordingly, it might be the case that the robot really ought to only perform one simple, uninteresting action over and over (such as reading from its sensors) in order to maximize its expected reward. If this is the case, the reward function must then be tuned so that the robot does other actions, like "ask human."

Furthermore, other drawbacks of using the POMDP in an integrated robotic system come exactly from the formality that provides its benefits. For simple tasks, the specification of the POMDP format is far more tedious than simply hand-coding the desired behavior. This is especially true because tools must be developed that parse and interpret the POMDP's environment and policy separate from the 'task' itself. As the tasks get more difficult, the POMDP has a greater advantage in generating complex behavior simply from solving it, but its manual specification becomes more and more tedious and error-prone. This means that problem-specific, compressed POMDP specifications must be developed, as well as tools to generate the 'true' POMDP specification from these compressed formats. Using the POMDP in a system that learns means updating the POMDP, which means writing one or more tools to convert all data in the reverse direction as well.

Finally, some shortcomings of a POMDP in an integrated robotic system stem from the nature of the POMDP itself. Logical structure within the state space is not naturally representable in a POMDP, so a task as simple as "Fly from X to Y" where X and Y must be determined from n cities, has a potentially massive and intractable state space size of $O(n^2)$. In practice, even state spaces of eight or ten caused the point-based solver to go from near-instantaneous execution to running for

multiple minutes to generate a close-to-optimal policy. Perhaps most importantly, for the classes of robotic interaction problems that we explored, the behaviors that emerge from the POMDP's optimal policies are simple thresholds on the belief space; information-gathering actions happen at unsure belief states, where more expensive actions that gather more information are more viable when the confusion over the true state is high.

5.1.2 Extensions

Several researchers have investigated methods for adapting POMDPs to overcome difficulties—some of which are presented above—that arise in their use. What follows is a selection of these methods.

Jason Williams and colleagues have done much work on POMDPs' use in spoken dialog systems. In [23], they factor the state space so that each state S is composed of n sub-states $S_1 ... S_n$; in particular, they use $n = 3$ for the user's intention, previous utterance, and the dialog history. This allows for a more natural representation of complicated composite state spaces. In [22] they address large state spaces by factoring the POMDP state representation so that a belief is held over top-k states plus a summary state that represents all very unlikely states. They also address logical structure between states in [24] by partitioning the state space into equivalence classes based on user intent. Though these techniques were applied to spoken-dialog systems, the issues they address have natural analogues in many domains of robotics.

POMDPs have been applied in the robotics domain for other purposes than dialog management. Rosenthall and colleagues use POMDPs in [16, 17] to model human robot helpers in order to estimate their accuracy and availability, better allowing mobile robots to judiciously ask for assistance from humans.

Other work [8, 18, 19] has addressed the issue of pre-specifying parameters of the POMDP. They use Bayesian methods to estimate the transition and observation functions, learning these parameters from the dynamics of a system. In [4], Finale Doshi-Velez proposes methods for also learning the POMDP's reward function and in [6], she proposes an "Infinite" POMDP that does not require knowledge of the

state space and instead models visited states as it goes along, effectively learning the state space. Doshi-Velez has also addressed the issue of POMDP's slow solving time in interactive systems in [5] with an efficient incremental update algorithm, allowing improvement without long delays in the interaction.

Overall, it is unclear to me whether the use of a POMDP is advantageous in an integrated robotic system. For smaller problems and more restricted domains, the cost of developing the infrastructure to integrate a POMDP and specify the problem in its terms might be higher than its benefit. For larger problems and more open-ended domains, the intractability of POMDPs is exposed, and more advanced techniques presented above should be incorporated to account for these difficulties. With that said, these concerns largely regard engineering cost, which is distinct from the theoretical usefulness of a POMDP.

5.2 Natural language grounding

5.2.1 Discussion

Whereas the methods in Chapters 2 and 3 focus on scenarios where grounding has been learned and the robot attempts to 'discover' its true state with a POMDP, the preliminary results presented in Chapter 4 address the question of learning such groundings.

Our work in this area at the time of writing is in progress, and there are many directions that could be taken to enhance the performance of the current system as well as extend its functionality to new domains. Before discussing results from other authors in the field, the following are some extensions to the current project that we are considering.

One natural extension is adding more words to the options of each classifier. For example, in a "heavy" versus "light" classifier, learning groundings for the word "medium" or the phrase "very heavy" would be a natural extension. This would of course require more training data during interactions, as well as extending the current

speech recognition functionality which operates on single words.

In addition to adding words to existing classifiers, adding new features to the observation feature vector would open up new avenues for additional dimensions of observation, such as color and size. Some work could be done to abstract, say, an image taken from the NAO's camera to just a dominant color. With these first two extensions, the NAO's potential learned vocabulary size would increase significantly from where it stands.

Several system components could be improved for better interactions. Existing classifiers or past training data could be saved and restored in future interactions, allowing the robot to benefit from past interactions and grow its knowledge over time[2]. The speech recognition component could be expanded to allow more than single word utterances, which would allow for significantly more natural interactions with humans, even via simple language input techniques like template matching. In addition, the word recognizer could be tuned to reject results below a certain likelihood threshold[3].

Nontrivial expansions of the system would involve changes to the model with which words are learned altogether. Learning qualifiers such as "very" poses an interesting task; take, for instance the relationship between "heavy" and "very heavy," and conversely, the relationship between "heavy" and "not very heavy." Removing the hard-coded structure of the provided vocabulary[4] would also give much more interesting results if the vocabulary could still be successfully learned. We have considered various approaches for doing this fully-naive learning of words, including attempting "continuousification" of words (mapping words to real-value numbers) and learning via dimensionality reduction and clustering.

[2]It is debatable whether this is currently a good idea given the noise of some of the sensors.

[3]The speech recognizer currently gives the most likely word whenever it hears a sound, which could have been a true utterance, or it could have been the shifting of a chair.

[4]By this, we are referencing the fact that words are currently directed to their relevant classifier via hard-coded routes in the system.

5.2.2 Extensions

Examples from recent published work describe areas of robotic grounded language acquisition that are relevant to consider when determining future work direction. What follows is a small sampling of a large body of such work.

Crowdsourcing is emerging as a viable method of data collection for robotics domains. Stefanie Tellex and colleagues [20] applied crowdsourcing to the domain of natrual language commands given to a robot by collecting a large corpus of commands and then applying a probabilistic graphical model over a command by decomposing it based on its semantic structure.

Asking targeted clarifying questions is another important area of research for robotic interaction with natural language. Tellex et al. extended the work previously mentioned in [21] where they propose an information-theoretic strategy for asking such questions, helping robotic systems to cope with natural language ambiguity and their limited perception of the environment. Robots asking targeted clarifying questions is not limited to natural language grounding; for example, Cakmak et al. use this approach—termed Active Learning—in [1] to assist programming by demonstration, where a robot is taught actions and then asks questions to recover relevant characteristics of its training data.

Another vein of research involved in helping robots learn natural language focuses on gleaning more information out of data collected than just words. In [10], Kollar and colleagues use a linguistically-motivated framework termed "Logical Semantics with Perception" that facilitates the interactive learning of language through vision, gesture, and basic language semantic analysis. They work also with natural language generation, which as a note of interest also has seen some overlap in work with dialog management systems [12]. Matuszek et al. have looked at robotic grounded language acquisition in the navigational domain through supervised learning [14], and augmenting vision with language to jointly learn a language and perceptive model [13]. I am particularly interested in this domain, where I believe that by utilizing exiting work in computer vision, a robot can determine a strong prior over words for use in its

speech recognition system. Applying this technique might help research where vast amounts of data manually collected through physical interactions must be thrown away due to speech recognition errors (e.g. in [10]).

Bibliography

[1] M. Cakmak and A. L. Thomaz. Designing robot learners that ask good questions. In *Proceedings of the International Conference on Human-Robot Interaction (HRI)*, 2012.

[2] Anthony R. Cassandra. *Tony's POMDP File Format Description*, 1999. http://www.pomdp.org/pomdp/code/pomdp-file-spec.shtml.

[3] F. Doshi and N. Roy. Efficient model learning for dialog management. In *Human-Robot Interaction (HRI), 2007 2nd ACM/IEEE International Conference*, pages 65–72, 2007.

[4] Finale Doshi, Joelle Pineau, and Nicholas Roy. Reinforcement learning with limited reinforcement: Using bayes risk for active learning in pomdps. In *Proceedings of the 25th international conference on Machine learning*, pages 256–263. ACM, 2008.

[5] Finale Doshi and Nicholas Roy. Efficient model learning for dialog management. In *Human-Robot Interaction (HRI), 2007 2nd ACM/IEEE International Conference on*, pages 65–72. IEEE, 2007.

[6] Finale Doshi-Velez. The infinite partially observable markov decision process. In *NIPS*, 2009.

[7] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[8] Robin Jaulmes, Joelle Pineau, and Doina Precup. Learning in non-stationary partially observable markov decision processes. In *ECML Workshop*, 2005.

[9] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

[10] Thomas Kollar, Jayant Krishnamurthy, and Grant Strimel. Toward interactive grounded language acquisition. 2013.

[11] Hanna Kurniawati, David Hsu, and Wee Sun Lee. SARSOP: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. *Proc. Robotics: Science and Systems*, 2008. http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/.

[12] Oliver Lemon. Learning what to say and how to say it: Joint optimisation of spoken dialogue management and natural language generation. *Computer Speech & Language*, 25(2):210–221, 2011.

[13] Cynthia Matuszek, Nicholas FitzGerald, Luke Zettlemoyer, Liefeng Bo, and Dieter Fox. A joint model of language and perception for grounded attribute learning. *arXiv preprint arXiv:1206.6423*, 2012.

[14] Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. Learning to parse natural language commands to a robot control system. In *Intnl. Symp. on Experimental Robotics (ISER)*, 2012.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[16] Stephanie Rosenthal and Manuela Veloso. Modeling humans as observation providers using pomdps. In *RO-MAN, 2011 IEEE*, pages 53–58. IEEE, 2011.

[17] Stephanie Rosenthal, Manuela M Veloso, and Anind K Dey. Learning accuracy and availability of humans who help mobile robots. In *AAAI*, 2011.

[18] Stephane Ross, Brahim Chaib-draa, and Joelle Pineau. Bayes-adaptive pomdps. In *Advances in neural information processing systems*, pages 1225–1232, 2007.

[19] Stephane Ross, Brahim Chaib-draa, and Joelle Pineau. Bayesian reinforcement learning in continuous pomdps with application to robot navigation. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 2845–2851. IEEE, 2008.

[20] Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R Walter, Ashis Gopal Banerjee, Seth J Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation.

[21] Stefanie Tellex, Pratiksha Thaker, Robin Deits, Thomas Kollar, and Nicholas Roy. Toward information theoretic human-robot dialog. In *Robotics: Science and Systems*, 2012.

[22] Jason D Williams, Pascal Poupart, and Steve Young. Factored partially observable markov decision processes for dialogue management. In *4th Workshop on Knowledge and Reasoning in Practical Dialog Systems, International Joint Conference on Artificial Intelligence (IJCAI)*, pages 76–82, 2005.

[23] Jason D. Williams and Steve Young. Partially observable markov decision processes for spoken dialog systems. *Computer Speech and Language*, 21:393–422, 2007.

[24] Steve Young, Jost Schatzmann, Karl Weilhammer, and Hui Ye. The hidden information state approach to dialog management. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 4, pages IV–149. IEEE, 2007.