Learning dialogue strategies for interactive database search

Verena Rieser\(^1\) and Oliver Lemon\(^2\)

\(^1\)Department of Computational Linguistics, Saarland University, Saarbrücken, Germany
\(^2\)School of Informatics, University of Edinburgh, Edinburgh, UK

vriesser@coli.uni-sb.de, olemon@inf.ed.ac.uk

Abstract

We show how to learn optimal dialogue policies for a wide range of database search applications, concerning how many database search results to present to the user, and when to present them. We use Reinforcement Learning methods for a wide spectrum of different database simulations, turn penalty conditions, and noise conditions. Our objective is to show that our policy learning framework covers this spectrum. We can show that even for challenging cases learning significantly outperforms hand-coded policies tailored to the different operating situations. The polices are adaptive/context-sensitive in respect of both the overall operating situation (e.g. noise) and the local context of the interaction (e.g. user’s last move). The learned policies produce an average relative increase in reward of 25.7% over the corresponding threshold-based hand-coded baseline policies.

Index Terms: Reinforcement Learning, dialogue systems, adaptive strategies, multimodality, database search

1. Introduction

In this paper we use dialogue policy learning (Reinforcement Learning) to address a complex and challenging problem for real database applications: how many database search results to present to the user, and when to present them, given the competing trade-offs between the number of results (large lists are difficult for users to process), the length of the dialogue (long dialogues are tiring, but collecting more information can result in more precise results), and the noise in the speech recognition environment (in high noise conditions accurate information is difficult to obtain). We construct the learning environment to span a wide range of applications: from very well behaved databases to databases with random behaviour, from high to low noise environments, from eyes-and-hands busy situations to situations where a large screen is available, from patient to impatient users.

There has been substantial work on using dialogue systems for accessing databases. One line of research aims to assist the user to browse the data by generating intelligent summaries at each step of the dialogue, e.g. [1, 2, 3], the other line helps the user to search for a concrete item by gathering more constraints from the user until a manageable number of results are retrieved, e.g. [4, 5]. We are focusing on the latter question and learn a dialogue strategy for multimodal output. In contrast to prior work we address the question of when to show a list of retrieved items on the screen rather than listing them verbally or providing verbal summaries.

Prior work using Reinforcement Learning (RL) for searching a database has learnt policies for only one specific application and one specific database where for representing DB hits two implicit thresholds are used; one for representing the number of database hits in the state space by quantising them (low-medium-high), and one in the reward function [4, 5]. The vast majority of dialogue policy learning research does not consider the number of database results at all, resulting in policies which are not sensitive to the current number of search results (e.g. [6]). This is obviously bad: if there is only one DB search result (or “hit”), we should tell it to the user immediately, regardless of how many information search slots have been filled in the dialogue. Conversely, if there are very many DB hits, we should persevere in getting more information (i.e. search constraints) from the user, depending on the penalty for longer dialogues (reflecting a ‘patient vs. impatient user’) and the reliability of the ASR channel (‘noise model’). The state space used for learning is quite limited. Levin and Pieraccini’s system [5] had 411 possible states, Pietquin’s [4] had \(3^7 = 2187\) states, and here we employ 102,400 states. These authors do not compare their learnt policies to any baseline. [4] does show that the reward definition has an effect on learning, but doesn’t report on statistical significance. We report significance of our results in section 5.

2. Experimental Framework

The Markov-Decision-Process (MDP) model serves as a formal representation of human-machine dialogue [7] and provides the basis for formulating strategy learning problems. Every MDP is formally described by a state space, an action set, a set of transition probabilities, and a reward function, as described below (in section 2.3). RL methods are used to determine optimal dialogue policies: mappings of dialogue states to dialogue actions.

2.1. Database simulations

To explore the space of possible policies we developed two extreme cases of database simulation (“monotonic” and “random”) for training policies, both reflecting different ways a user query can affect search results. In both cases, we assume a total database size of 100 items. We assume that presenting more than 100 results to a user is never going to be desirable, so in this paper we learn strategies where there are 100 or fewer possible answers. When the number of results is more than 100, we assume that the correct strategy is to ask for more constraints from the user.

We also assume that the user’s goal item is contained in the database. Learning when to ask for constraint relaxation is future work.

**Monotonic DB simulation:** The first, “monotonic” simulation, models an extremely well behaved search tasks where each additional search constraint strictly reduces the number of search results obtained. Thus, if the user fills a search slot...
“I want a song by Arcade Fire” or “I want sushi”) the number of results returned is strictly less than in the prior state. This models boolean AND search. Conversely, if a slot becomes unfilled (e.g. by a rejected confirmation move), the number of search results will increase. Here we sample from a normally distributed database. Every search constraint (i.e. filled slot) lowers the sample mean and narrows down the standard deviation.

**Random database simulation:** For the second database model (”random”) a user-provided search constraint (slot) can either be interpreted as an AND or an OR constraint. That is, the number of DB hits can either increase or decrease, and this is approximated in a random model, sampling between 1 and 100 hits. For this model newly provided information may in fact open up new possibilities in the data. For example the user might say “how about restaurants in the Old Town?” thus shifting focus away from the current set of results and opening up a new (possibly larger) set of search results. This is not intended to be a realistic database model - but one which occupies one end of a spectrum. Our objective is to show that our policy learning framework covers this spectrum.

### 2.2. Problem representation

We represent a 4-slot information search dialogue problem as a MDP. There are 10 binary state variables (for $1 \leq N \leq 4$, fill-slotN for whether each slot number N is filled, confirm-slotN for whether each slot number N is confirmed, prv-yes for whether the last user move was “yes” and prv-no for whether the last user move was “no”), and one variable DB for the current number of DB hits, which takes integer values between 1 and 100, resulting in $2^{10} \times 100 = 102,400$ distinct dialogue states.

The following system actions are available for exploration in every state:

- **greet** e.g. “How can I help you?”
- **ask a slot (AskASlot)**, e.g. “What kind of food would you like?”
- **explicit confirm (explicitConf)**, e.g. “Did you say Italian?”
- **implicit confirm and ask a slot (implConf-AskASlot)** e.g. “Okay, an Italian restaurant. What price range?”
- **present information (presentList)** e.g. “The N items matching your query are shown on the screen.”

The action “greet” is an open-initiative question for as many search constraints as the user wishes to give: “How may help you?”, and the asked slot name (for AskASlot and implConf-AskASlot) is controlled by a process model for the domain which describes a default ordering on slots for the task (e.g. ask for music first, then artist, then album, then song title.) The user is not constrained to follow this ordering, and can also over-provide information using mixed-initiative behaviour such as over-answering. In future work we will learn the optimal ordering on how to ask for slots.

### 2.3. Reward function

We propose a novel reward function which incorporates ASR noise modelling. For each dialogue we have:

$$\text{FinalReward} = \text{completionValue} - \text{dialogueLengthPenalty} - \text{DBhitsPenalty}$$  

Where dialogueLengthPenalty penalises every system turn (via a TurnPenalty TP per turn) and DBhitsPenalty penalises every item which is presented to the user (via an itemPenalty IP per presented item).

The completionValue of a dialogue is defined as the percentage probability that the user goal is in the result set that they are presented with. For example, if we know with 100% certainty that the user wants Sushi in the Old Town (i.e. 2 slots, both confirmed at 100% probability), then we have a 100% chance of supplying the user with an item that meets their goal. On the other hand, if we are in the same situation but we are only 80% sure that they want Sushi then the probability of their goal being in the list we present them with is only $0.8 \times 1 = 80\%$.

Thus the completion value of a dialogue is directly related to the probability of search slots being correctly filled, which is in turn related to the noise conditions under which the dialogue is being conducted. Thus, where $P_c$ is the probability of a confirmed slot being correct, and $P_f$ is the probability of a filled slot being correct, where $C$ and $F$ are the number of confirmed slots and filled (but not confirmed) slots respectively, we have:

$$\text{completionValue} = 100 \times (P_c)^C \times (P_f)^F$$  

(2)

For example, in a High Noise environment, we might set $P_c = 1.0$ and $P_f = 0.5$, reflecting the fact that in a noisy environment unconfirmed slots are fairly likely to be incorrect (50% chance). In a real application domain these probabilities can be estimated from Wizard-of-Oz data [8].

### 2.4. User simulations

Exploratory trial-and-error learning with real users is an expensive, time-consuming, and sadistic procedure, so in current research user simulations are applied for learning [4, 5]. Evaluation with real users is of course preferable, and is our ultimate goal, but previous studies have shown that policy learning results obtained for simulated users do in fact carry over to results for real users [9].

The user simulations employed here are on the intention level, where the possible user acts are stochastic estimates conditioned on the previous system action to simulate a relatively collaborative user behaviour. Possible user actions are yes-answer, provide-other (e.g. “I want ABBA” when asked “What type of music do you want?”), yes-provide-asked (e.g. “yes, I want ABBA” when asked “Ok, pop music. What band do you want?”), no-answer, provide-two-slots (e.g. “I want a Radiohead song from the album OK Computer”), or remain silent. For example, if the system’s previous move was to ask for a slot, the user has a 20% chance of providing a different slot value, a 70% chance of providing the requested slot value, a 6% chance of providing two slot values, and a 4% chance of remaining silent. For system confirmation moves the likelihood of the user simulations rejecting the confirmed information is the same as the probability of a filled slot being incorrect ($P_f$). In future work we will employ a user model learnt from data [10].

### 3. The hand coded baselines

We construct a range of hand-coded policies for comparison with learned policies. These baselines are “state of the art” policies in the sense that they allow mixed-initiative interaction and use thresholds (for dialogue length and number of DB hits) chosen for the particular operating environments.
The hand-coded baseline policies follow a similar threshold based approach to that described in [2, 3]. The different baseline policies all use the following basic strategy pattern, modified by contextually appropriate global thresholds (see section 5) for dialogue length and and number of DB results:

1. Greet the user,
2. either AskASlot (if no slots need to be confirmed) or ImplConf-AskASlot (if there are remaining slots to ask and slots to confirm),
3. then repeat 2 until there are no slots left to fill,
4. then ExplConf the remaining filled slots (if any)
5. if a threshold is reached PresentInfo; present the an-

The hand-coded baselines will thus always PresentInfo if all slots are confirmed or the number of items returned from the database is less than a threshold (for example < 7) or the dialogue length exceeds a threshold (for example > 6). The different thresholds for the hand-coded policies thus determine when to stop asking for new constraints. Of course, a major difficulty with heuristic, hand-coded, threshold-based approaches is that we need to know how the thresholds should be set, and how the thresholds interact for each environment. The policy learning experiments presented below effectively learn optimal values for these global thresholds, as well as local tradeoffs that hand-coded policies are not fine-grained enough to capture. For the baseline systems, we designed four hand-coded policies with the following combinations of thresholds:

\[
\begin{align*}
H_{\text{HiTurn,HiHit}}: & \text{ short dialogues (}\leq 6\text{), short lists (}\leq 7\text{); e.g. small screen, impatient user} \\
H_{\text{LowTurn,HiHit}}: & \text{ long dialogues (}\leq 10\text{), short lists (}\leq 7\text{); e.g. small screen, patient user} \\
H_{\text{HiTurn,LowHit}}: & \text{ short dialogues (}\leq 6\text{), long lists (}\leq 14\text{); e.g. large screen, impatient user} \\
H_{\text{LowTurn,LowHit}}: & \text{ long dialogues (}\leq 10\text{), long lists (}\leq 14\text{); e.g. large screen, patient user}
\end{align*}
\]

4. Learning method

For policy learning we use the system REALL described in [11], which uses the SARSa Reinforcement Learning algorithm (with linear function approximation) to learn over the entire policy space.

In the following we report on 16 experiments where we systematically varied the database definitions and reward functions, to explore policy learning for combinations of noise level (LowNoise and HiNoise), database model (monotonicDB and randomDB), hit penalty (LowHit and HiHit), and turn penalty (LowTurn and HiTurn). The application scenarios given above (e.g. small screen, small reward, impatient user: short dialogues, short lists) are now reflected in how we formulate the overall goal of the dialogue by reward functions for policy learning (i.e. we are “programming by reward”). In particular we focus on learning different policies for different operating conditions, using the the 8 different reward structures for both monotonic and random DB models. For example:

- HiNoise,HiHit,HiTurn: 50% chance of filled slots being correct, turn penalty=-10, hit penalty=-10; e.g. noisy in-car environment, small screen, impatient user
- LowNoise,LowHit,LowTurn: 80% chance of filled slots being correct, turn penalty=-1, hit penalty=-1; e.g. in-home, larger screen, patient user

Evaluation: We test each (learned and baseline) policy in each condition by running 550 test dialogues in simulation. We compared all pairs of learned and hand-coded policies in respect of their average final reward per dialogue over the test runs. We then perform a T-test (with Bonferroni correction) on the final rewards, to determine statistical significance.¹

5. Results

The results produced by the learnt policies (denoted RL) and the different hand-coded baselines \((H_{\text{HiTurn,HiHit}} \ldots H_{\text{LowTurn,LowHit}})\) for 2 of the 16 operating conditions (Noise, DB, HitPenalty etc.) can be seen in table 1. These are two extreme cases for learning: a policy trained with a monotonic DB, for HiNoise, LowTurn, HiHit conditions (left); and a policy trained with a random DB, for LowNoise, HighTurn, LowHit conditions (right) versus the hand-coded policies. The strategy on the left can be employed in a noisy environment where only a small screen is available and the dialogue length is not very important. This policy learns to reduce the number of presented items to 5 in 8.6 turns on average while filling and confirming all the slots (see example 3). The strategy on the right (table 1) is for a less noisy environment with a large screen, but time is crucial. This strategy learns to present a list of 29 items on average after only 3 turns while confirming slots is not crucial (see example 4).

In the LowNoise case, the learner has settled on a policy of confirming fewer slots than in the high noise case (because filled slots are more reliable in low noise). Interactions are thus shorter, and more reward is gained.

(3) S: ‘Hello, how can I help you?’
U: ‘I want a Radiohead song’
provide_info(artist)
implConf(artis)AskASlot(album)(hits:77)
U: ‘Paranoid Android from OK Computer’
provide_info(song_title,album)
S: ‘Did you say Paranoid Android from OK Computer?’
explConf(song_title,album)(hits:1)
U: ‘Yes’
S: ‘The requested item is now shown on the screen.’
presentList(artist,album,song_title)(hits:1)

(4) S: ‘Hello, how can I help you?’
U: ‘I want a song from OK Computer’
provide_info(album)
S: ‘I found 12 items now shown on the screen.’
presentList(album)(hits:12)

In general all policies perform worse in the more challenging random DB case, as expected, but even for this extreme case we can learn a good strategy which outperforms handcrafted ones. For example, the learned policy for the random case has learned that in some cases not all filled slots need to be confirmed (presumably because sometimes users provide new information when asked for confirmation, which can increase the number of DB hits in the random model).

Across all the results, the learned policies produce an average relative increase in reward of 25.7% over the corresponding threshold-based hand-coded baseline policies. In 93% of test runs the learnt policies significantly \((p < .001)\) outperformed the hand-coded policies (the other cases also performed better, but not significantly so).

¹Complete results are available at www.coli.uni-saarland.de/~vrieser/when-to-present-list.html.
5.1. Discussion

The above results show that, for a complex decision problem, policy learning is able to find more effective local trade-offs which are globally optimal than the hand-coded strategies implemented by global threshold setting. In addition, in practice, when hand-coding a policy to exactly fit the application requirements all the parameters would have to be known in advance, as well as how these parameters relate to specific threshold settings. In contrast, when “programming by reward” for a real application (which is somewhere in the range of the extreme cases explored here) the reward definition reflects the overall goal of the dialogue (e.g. being short). The specific parameters of the reward definition for a particular application can be estimated from data [8].

6. Conclusion

We used dialogue policy learning (Reinforcement Learning with the SARSA algorithm and linear function approximation) to address a wide range of complex problems for real dialogue applications: how many database search results to present to the user, and when to present them, given the competing trade-offs between the length of the list, the length of the dialogue, and the noise in the speech recognition environment.

We showed that we can learn strategies for this complex problem which are significantly better than a variety of hand-coded baseline policies, for a wide range of noise conditions, types of DB, and turn-penalties. The learned policies produce an average relative increase in reward of 25.7% over the corresponding threshold-based hand-coded baseline policies. In 93% of the cases the learnt policies perform significantly better than the hand-coded ones (p < .001). We can show that even for the difficult cases of the spectrum (such as a random DB model) RL significantly outperform hand-coded policies tailored to the different operating situations.

In future work we will apply this general framework to a concrete application, using a real database and a reward function estimated from Wizard-of-Oz data. We also plan to compare the learned strategies to those of human “wizards” from our data collection [12], and to test the policies with real dialogue system users. In addition, different ways for presenting a list of items (e.g. using verbal summaries) will be explored.

7. References