MLSALT9
Statistical Spoken Dialogue Systems Practical

Paweł Budzianowski, Pei-Hao (Eddy) Su and Milica Gasic[*]

Due March 25th, 2017

1 Introduction

In this practical you will practice implementing two of the core components in the statistical spoken dialogue systems: **dialogue state tracker** and **dialogue policy**. The objective of this practical are as follows:

- understand the general setting of a statistical spoken dialogue system
- know how the system tracks the dialogue state
- implement a **simple dialogue state tracker**
- understand how the dialogue manager is formulated as a policy optimisation task
- implement the update rules of the **Monte Carlo** and **Gaussian process-SARSA** algorithm for dialogue manager and compare the results using a simulated user

2 CUED Python code

CUED Python Statistical Dialogue System is maintained by the Dialogue Systems Group. It provides a flexible framework for understanding how the dialogue state tracker and the dialogue policy works.

You can clone the code from the following repository with git:

https://eddy0613@bitbucket.org/dialoguesystems/cuedpython_practical.git

[*]Some parts of this document are based on the Dialogue State Tracking Challenge handbook written by Matthew Henderson, Blaise Thomson and Jason Williams

3 Dialogue State Tracking

The term 'dialogue state' loosely denotes a full representation of what the user wants at any point in a dialogue. An effective dialogue system must include a tracker which is able to accurately accumulate evidence over the sequence of tuns within a dialogue, and adjust the dialogue state according to the observations. In this section you will implement a simple dialogue state tracker and compare its performance with a baseline tracker on the Dialogue State Tracking Challenge (DSTC) dataset.

3.1 Find the Code and DSTC dataset

Relevant files and directories are shown as follows:

```
cued-python/practical1_dst/
        DST.README          # instruction and commands
        data/               # data used in DSTC2
        scripts/
           baseline.py      # starter code
           score.py         # generate output result
           check_track.py   # check output validity
           report.py        # report scores
           misc.py          # useful tools
           config/
                dstc2_data.flist # partial dstc2 data list
                ontology_dstc2.json     # ontology
```

Please refer to DST.README for more commands and requirement details.

3.2 Components of a Dialogue State

A tracker outputs for each turn a set of distributions for each of the three components of the dialogue state:

- **Goals**:
  - For each informable slot in the ontology, a distribution is spread over the all values in that slot. Note that a slot is informable if the user can inform the system about it. Any remaining probability mass is given to the hypothesis that the user has not yet mentioned this slot.
  - A distribution over joint goals. If this is omitted, then it is assumed that the goal should be composed of independent combinations of the slot distributions. The evaluation script scores the reported joint distributions as well as the distributions calculated making this assumption for reference.
- **Method** - a distribution over methods. The list of possible values is given in the ontology distributed with the data.
- **Requested slots**- for each requestable slot in the ontology, a binary distribution over whether this slot has been requested by the user and the system should inform it.

Each is described in some detail below.

3.2.1 Goals

The goal at a given turn is the user's true required value for each slot in the ontology as has been revealed in the dialogue up until that turn. If a slot has not yet been informed by the user, then the goal for that slot is 'None'.

3.2.2 Method

The 'method' of a user's turn describes the way the user is trying to interact with the system, and is one of the following values: by name, by constraints, by alternatives or finished. The method may be inferred if we know the true user's action using the following rules:

1. The method becomes 'by constraints' if the user gives a constraint by specifying a goal for a particular slot. E.g. `inform(food=chinese)`
2. The method becomes 'by alternatives' if the user issues a 'reqalts' act.
3. The method becomes 'by name' if the user either informs a value for the name slot, i.e. the user requests information about a specific named venue.
4. The method changes to 'finished' if the user gives a 'bye' act.

It is possible that the user makes no action which triggers one of the above rules at the start of the dialogue, so the method is initially labelled as "none".

3.2.3 Requested slots

As mentioned above, for each turn a tracker must output a score for each requestable slot which specifies its confidence that the user has requested this slot and the system should inform it. The set of true requested slots is accumulated throughout the dialogue as follows:

1. At the start of the dialogue, the set is initialised as empty
2. If the system informs the value of s then s is removed from the set (if it is a member), as it is no longer required by the user
3. If the user requests a slot s (action contains `request(`s`)`) then s is added to the set

3.3 Dialogue State Tracker

3.3.1 Baseline tracker

One simple way to track the dialogue state, commonly used in spoken dialogue systems, is to give a single hypothesis for each slot at each turn, whose value is the top scoring suggestion so far in the dialogue. Note that this tracker does not handle the goal constraint changes well since it ignores the past states but only uses the current state.

This serves as the example tracker in the code and is used to compare with the focus tracker as in the next section.

3.3.2 Focus tracker

Compared to the baseline tracker, focus dialogue state tracker accumulates evidence and has a simple model of the state changing throughout the dialogue. Here we define how it tracks goals and the method. Define for method and goal components c, the quantity $q_{c,\,t}$ as:

$$q_{c,\,t} = 1 - \sum_{v \in V_c} slu_{t,\,c}(v)$$

Note that we should have $0 \leq q_{c,\,t} \leq 1$ as the SLU list is normalised to sum to 1. For our example turn:

$$\begin{aligned} q_{t,\,\text{method}} &= 0.0 \\ q_{t,\,\text{goal area}} &= 0.7 \end{aligned}$$

This is interpreted as the probability that the SLU did not inform the component c. Recursively define $p_{c,\,t}(v)$ for each v in V_c as:

$$\begin{aligned} p_{c,\,t}(v) &= slu_{t,\,c}(v) + q_{c,\,t}p_{c,\,t-1}(v) \\ p_{c,\,1}(v) &= slu_{1,\,c}(v) \end{aligned}$$

Thus if there is no evidence that c should be reset in the SLU, then $q_{c,\,t} = 1$, $slu_{t,\,c}(v) = 0 \;\forall v$ and $p_{c,\,t} = p_{c,\,t-1}$. If the context is sure that c is to be reset, then $q_{c,\,t} = 0$ and $p_{c,\,t} = slu_{t,\,c}(v)$. $p_{c,\,t}(v)$ may be interpreted as the probability that v is the most recently mentioned value for c, when trusting the distributions from SLU directly. The focus baseline tracker uses the $p_{c,\,t}$ distributions for its output, and does something similar for requested slots- allowing for the system action to reset the scores when a slot is informed.

3.4 Question

Implement the **focus** dialogue state tracker

1. compare its performance with the baseline tracker on DSTC **dstc2_data** dataset.
2. show up to 20 lines of the code you implement.

Please refer to DST.README for more commands and requirement details and see the file: *baseline.py*, where section *TODO* is highlighted.

4 Dialogue Policy Optimisation

For this part of the practical you are asked to implement parts of two reinforcement learning algorithms and compare their learning curves and final policy on the simulated user.

4.1 Find the code

Relevant files and directories are shown as follows:

```
cued-python/policy/
        DM.README       # instruction and commands
        MCCPolicy.py    # Monto Carlo Control poliy
        GPPolicy.py     # Gaussian Process poliy
        GPLib.py        # Gaussian Process policy library
```

Please refer to DM.README for more commands and requirement details.

4.2 Monte Carlo control Algorithm

4.2.1 Introduction

Monte Carlo control (MCC) algorithm is a model-free method for solving the reinforcement learning problem. It learns after the complete episode (dialogue) finishes (rather than step-by-step such as Q-learning) and assigns the averaging sample returns to update the Q function, and the policy is repeatedly improved with respect to the current Q function.

In this section, you will implement the MCC algorithm for the dialogue policy update. The main idea is to maintain a dictionary which contains a set of (belief, action) pairs, where the total number is determined by a sparcification threshold ν. The algorithm is divided into two phase:

1. **Episode generation**: The goal is to generate a sequence of action to form an episode. Based on the estimated Q at any point, the model chooses an action a_t given the current belief b_t. Here the *epsilon*-greedy method is adopted to determine the probability between exploration/exploitation (or random/greedy) mode. If in greedy mode, given the current b_t, for each admissible a, it will find the grid points $\hat{\mathbf{b}}$ from the dictionary which is nearest to the considered (b_t,a) and return the corresponding Q, and then choose the action with highest Q.

2. **Q value update**: After every episode ends, the model scans each (b,a) pair in the episode and check if the distance between it and its nearest grid point in the dictionary is smaller than the threshold ν. If yes, the model will update the Q of this grid point by averaging all sampled total return starting from this point so far; if no, current (b,a) pair will be added to the dictionary $\mathcal{D}$ and the its Q value initialised with the sampled total return.

For more information on MCC algorithm in reinforcement learning, please refer to section 5 in [1]. The pseudo code is shown in Figure 1.

4.2.2 Question

Implement the **episode generation** and **Q value update** parts in Fig. 1 to optimise the system in interaction with the simulated user under concept error rate 15%.

1. examine the influence of the specification parameter ν and plot the learning curve of task success along with the training dialogues (show the averaged testing results of 100 dialogues after every 200 training dialogues, up to 2000 training dialogues).
2. show up to 20 lines of the code you implement.

Please refer to DM.README for more commands and requirement details and see the file: *MCCPolicy.py*, where the *TODO* sections are highlighted.

```
1:  Let $Q(\hat{\mathbf{b}}, a)$ = expected reward of taking action $a$ at grid point $\hat{\mathbf{b}}$
2:  Let $N(\hat{\mathbf{b}}, a)$ = number of times action $a$ is taken at grid point $\hat{\mathbf{b}}$
3:  Let $\mathcal{D}$ be a set of $(\hat{\mathbf{b}}, a)$ pairs $\leftarrow$ start from an empty list
4:  Let $\pi : \mathcal{B} \to \mathcal{A}$
5:  Let $\nu$ be the sparcification parameter
6:  repeat
7:      $t \leftarrow 0$
8:      $a_0$: initial randomly chosen action
9:      $\mathbf{b}_t = \mathbf{b}_0$: initial grid point

        (Generate an episode using $\epsilon$-greedy policy)
10:     repeat                                                   $\leftarrow$ TODO
11:         $t \leftarrow t + 1$
12:         Update belief state $\mathbf{b}_t$
13:
            $$a_t \leftarrow \begin{cases} \mathit{RandomAction} & \text{with prob. } \epsilon \\ \arg\max_a Q(\text{Nearest}((\mathbf{b}_t, a), \mathcal{D})) & \text{with prob. } (1-\epsilon) \end{cases}$$
14:         record $\langle \mathbf{b}_t, a_t \rangle$, $T \leftarrow t$
15:     until episode terminates with total return $R$

        (Scan episode and update $\mathcal{D}$, $Q$ and $N$)
16:     for $t = T$ downto 1 do                                  $\leftarrow$ TODO
17:         $(\hat{\mathbf{b}}, a_t) = \text{Nearest}((\mathbf{b}_t, a_t), \mathcal{D})$
18:         if $|(\hat{\mathbf{b}}, a_t) - (\mathbf{b}_t, a_t)| < \nu$ then     $\leftarrow$ update estimate for pt in $\mathcal{B}$
19:             $Q(\hat{\mathbf{b}}, a_t) \leftarrow \frac{Q(\hat{\mathbf{b}}, a_t) * N(\hat{\mathbf{b}}, a_t) + R}{N(\hat{\mathbf{b}}, a_t) + 1}$
20:             $N(\hat{\mathbf{b}}, a_t) \leftarrow N(\hat{\mathbf{b}}, a_t) + 1$
21:         else                                                 $\leftarrow$ create new pt
22:             add $(\mathbf{b}_t, a_t)$ to $\mathcal{D}$
23:             $Q(\mathbf{b}_t, a_t) \leftarrow R$, $N(\mathbf{b}_t, a_t) \leftarrow 1$
24:         end if
25:         $R \leftarrow \gamma R$                                  $\leftarrow$ discount the reward
26:     end for
27: until converged
```

Figure 1: Monte Carlo Control Algorithm

4.3 GP-Sarsa algorithm

4.3.1 Introduction

Given an observed (b_t, a_t) pair at time-stamp t, unlike MCC algorithm which determine the Q value by its nearest discrete grid points $(\hat{\mathbf{b}}, a)$, GP-Sarsa exploits the use of *kernel function* to define this observation's correlation with its every collected points to help estimating the Q value.

In GP-Sarsa algorithm the unknown Q-function is modelled as a Gaussian process with zero mean and kernel function $k(\cdot, \cdot)$, $Q(b, a) \sim \mathcal{GP}(0, k((b, a), (b, a))$. This Q-function is then updated by calculating the posterior given the collected (b,a) pairs and their corresponding rewards [1]. This knowledge of the distance between (b,a) pairs in the observation space highly increases the policy learning speed. For each update the current **s**tate, current **a**ction, **r**eward, next **s**tate and next **a**ction is used, thus named **SARSA**.

However, in practice, since the computation complexity will increase linearly if every data point is memorised, sparse methods are used. A kernel function can be thought of as the dot product of a set of feature functions $k((b, a), (b, a)) = \langle\, \boldsymbol{\phi}(b, a), \boldsymbol{\phi}(b, a)\rangle$ where $\phi(b, a)$ is the feature vector. Any linear combination of feature vectors $\left\{\boldsymbol{\phi}(b^0, a^0), ..., \boldsymbol{\phi}(b^t, a^t)\right\}$ given a set $\left\{(b^0, a^0), ..., (b^t, a^t)\right\}$ is called the *kernel span*. The aim then is to find the subset of points $\left\{(\widetilde{b}^0, \widetilde{a}^0), ..., (\widetilde{b}^t, \widetilde{a}^t)\right\}$ that approximates this kernel span. These points are called the representative points which form a dictionary $\mathcal{D}$.

Similar to MCC algorithm in section 4.2, a sparcification threshold ν is set to control the dictionary size. To avoid adding the current visiting point (b^t, a^t) to the dictionary, the following approximate linear dependence (ALD) condition should be satisfied:

$$min_{\boldsymbol{g}_t} || \sum_{j=0}^{m} \boldsymbol{g}_{tj} \boldsymbol{\phi}(\widetilde{b}^j, \widetilde{a}^j) - \boldsymbol{\phi}(b^t, a^t)||^2 < \nu$$

where $\boldsymbol{g}_t$ is the coefficient vector, and the dictionary is of size m. This is equivalent [3] to:

$$min_{\boldsymbol{g}_t} \left(k((b^t, a^t), (b^t, a^t)) - \widetilde{\boldsymbol{k}}_{t-1}(b^t, a^t)^\top \boldsymbol{g}_t)\right) < \nu \tag{1}$$

where $\widetilde{\boldsymbol{k}}_{t-1}(b^t, a^t) = \left\{k((b^t, a^t), (\widetilde{b}^0, \widetilde{a}^0)), ..., k((b^t, a^t), (\widetilde{b}^m, \widetilde{a}^m))\right\}^\top$. Also, the left side of Eqn. 1 is minimised when $\boldsymbol{g}_t = \widetilde{\boldsymbol{K}}^{-1} \widetilde{\boldsymbol{k}}_{t-1}(b^t, a^t)$, where $\widetilde{\boldsymbol{K}}$ is the Gram matrix of the current set of representative points. This constitutes the sparsification criterion and it allows the posterior to be approximated. For more details on the posterior calculation of Q value, please refer to [2]. The pseudo code of GP-SARSA is shown in Figure 2.

Here you will be implementing a part of the GP-SARSA and understanding the core idea of its benefit.

4.3.2 Question

Implement the sparcification criterion part in Figure 2 to complete the GP-SARSA algorithm with the threshold $\nu = 0.01$ under concept error rate 15%.

1. Use linear kernel function $k((\mathbf{b}, a), (\mathbf{b}', a')) = \langle \mathbf{b}, \mathbf{b}' \rangle \, \delta_a(a')$, $\sigma = 5$ and compare the its success rate learning curve to the MCC algorithm (show the averaged success rate results of 100 dialogues after every 200 training dialogues, up to 2000 training dialogues).
2. (Optional) Use Gaussian kernel function $k((b, a), (b', a')) = p^2 \exp(-\frac{||\mathbf{b}_i - \mathbf{b}_j||_2^2}{2l^2})\delta_a(a')$, by tuning parameters p and l and compare its success rate learning curve results with the linear kernel GP-SARSA and MCC algorithm.
3. show up to 20 lines of the code you implement.

- δ is Kronecker delta function.

Please refer to DM.README for more commands and requirement details and see the file: *GPLib.py*, where the *TODO* sections are highlighted.

```
1: Initialise $\tilde{\boldsymbol{\mu}} \leftarrow []$, $\tilde{\mathbf{C}} \leftarrow []$, $\tilde{\mathbf{c}} \leftarrow []$, $d \leftarrow 0$, $1/v \leftarrow 0$, $\pi \leftarrow random$
2: for each episode do
3:     Initialise $\mathbf{b}$
4:     if initial step then
5:         Choose $a$ arbitrary, $\mathcal{D} = \{(\mathbf{b}, a)\}$, $\tilde{\mathbf{K}}^{-1} \leftarrow 1/k((\mathbf{b}, a), (\mathbf{b}, a))$
6:     else
7:         Choose $a \leftarrow \pi(\mathbf{b})$
8:     end if

       (Given initial step, check whether current (b,a) should be added as a support point)
9:     $\tilde{\mathbf{c}} \leftarrow \mathbf{0}$ size $|\mathcal{D}|$, $d \leftarrow 0$, $1/v \leftarrow 0$
10:    $\mathbf{g} \leftarrow \tilde{\mathbf{K}}^{-1}\tilde{\mathbf{k}}(\mathbf{b}, a)$, $\delta \leftarrow k((\mathbf{b}, a), (\mathbf{b}, a)) - \tilde{\mathbf{k}}(\mathbf{b}, a)^{\mathsf{T}}\mathbf{g}$        ▷ TODO
11:    if $\delta > \nu$ then
              Add current point (b,a) to the dictionary $\mathcal{D}$.
              Update the parameters.
12:    end if

       (Given next step, check whether next (b',a') should be added as a support point)
13:    for each step in the episode do
14:        Take $a$, observe $r'$, update $\mathbf{b}'$

15:        if non-terminal step then
16:            Choose new action $a' \leftarrow \pi(\mathbf{b}')$
17:            $\mathbf{g}' \leftarrow \tilde{\mathbf{K}}^{-1}\tilde{\mathbf{k}}(\mathbf{b}', a')$, $\delta \leftarrow k((\mathbf{b}', a'), (\mathbf{b}', a')) - \tilde{\mathbf{k}}(\mathbf{b}', a')^{\mathsf{T}}\mathbf{g}'$
18:        end if

19:        if $\delta > \nu$ and non-terminal step then
              Add current point (b',a') to the dictionary $\mathcal{D}$.
              Update the parameters.
20:        end if

           (move on to next b, a)
21:        if non-terminal step then
22:            $\mathbf{b} \leftarrow \mathbf{b}'$, $a \leftarrow a'$
23:        end if
24:    end for
25: end for
```

Figure 2: Sparse Approximation for Episodic GP-Sarsa

References

[1] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction.* 1998.

[2] Milica Gasic and Steve Young. *Gaussian processes for POMDP-based dialogue manager optimisation.* TASLP, 2014.

[3] Yaakov Engel. *Algorithms and Representations for Reinforcement Learning.* PhD thesis, 2005.