1 Introduction

In target-oriented Spoken Dialogue System the user-system interaction can be decomposed into three main units as shown in **Figure 1**:

1. A *Spoken Language Understanding* (SLU) unit takes user noise-corrupted waveform as input and produces a distribution over user dialogue-acts as output after processing it through Automatic Speech Recognition (ASR) and Semantic Decoding systems.
2. A *Dialogue Management* unit, which will be the main focus of this practical, consisting of two parts:
   - **Belief-state tracker**: user dialogue-act is combined with previous SLU evidence to update an internal representation of the intentions of the user, known as belief-state.
   - **Dialogue Policy Optimisation**: the current belief-state is used to produce the system dialogue act response.
3. A *Natural Language Speech Synthesis Generation* unit which generates the system dialogue act through Speech Synthesis using Natural Language Generation.

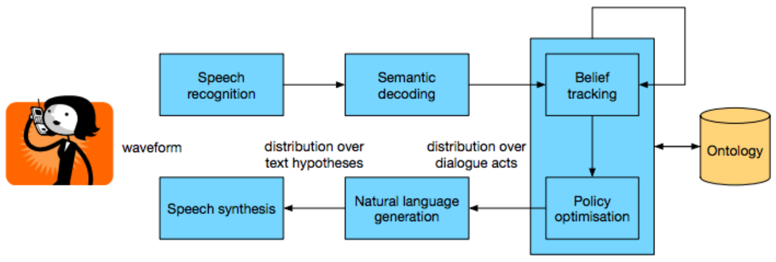

Figure 1: Modular architecture of a Statistical Spoken Dialogue System

A recent breakthrough in Statistical SDS was the reformulation of the Dialogue Management task as a Partially Observable Markov Decision Process (POMDP). This Reinforcement Learning framework allows the SLU generated uncertainty to propagate down the SDS pipeline, increasing system robustness to errors and resulting in a more efficient user-system interaction. In the following sections we will explore in detail this formulation and its implementation.

2 Dialogue State Tracker

In a Dialogue State Tracker, the dialogue-state is encoded into three components: *goals*, consisting slot and value pairs from the ontology; *requested-slots*, the belief on which slots were requested by the user; and *methods*, ways in which the system is informed by the user about requested-slots. In each turn, previous SLU evidence is combined with current observations to yield a belief-state b_t, a distribution over all possible dialogue states. Belief-state tracking is commonly done using discriminative models, where $b_t = p(s_t|\mathbf{o})$, a distribution over dialogue-states given some previous observations. These models are known to handle correlated features well. In this section, we implement and compare two discriminative models for belief-state tracking, namely, the *Baseline Tracker* and the *Focus Tracker*.

2.1 Baseline Tracker

A simple and commonly adopted discriminative method for belief tracking is the *Baseline Tracker*. At each turn, the single highest scoring hypothesis until that point, in terms of SLU evidence, is assigned to each slot. Thus, any past belief-state history is ignored and the current SLU evidence is given the same priority as those in previous turns. Consequently, the Baseline Tracker is not able to appropriately adapt to new goal constraints, resulting (as we will see) in a poor performance.

2.2 Focus Tracker

In contrast, the *Focus Tracker* accumulates SLU evidence throughout the dialogue and balances its contribution depending on current observations, in order to change the focus of attention. At each turn t, for method and goal belief-state components c, the probability that item v was most recently mentioned, $p_{c,t}(v)$, is recursively updated as a weighted sum of current SLU evidence that v was mentioned and its previous value:

$$p_{c,t}(v) = slu_{t,c}(v) + q_{c,t}p_{c,t-1}(v) \tag{1}$$
$$p_{c,t} \leftarrow normalise(p_{c,t}) \tag{2}$$

where $p_{c,1}(v) = slu_{1,c}(v)$. Here, the *scaling factor* $q_{c,t} = 1 - \sum_{v \in V_c} slu_{t,c}(v)$ represents the probability that certain component of the belief-sate c was not mentioned during the dialogue act by the user. Thus, when $q_{c,t}$ is high, $slu_{t,c}(v) \simeq 0\ \forall v$; and $p_{c,t}(v)$ does not require an update. Similarly, if there is substantial evidence that c was mentioned, $q_{c,t}$ takes a small value, causing the update of $p_{c,t}(v)$ to be fully based on current evidence $slu_{t,c}(v)$. Next, we implement this gating update rule for each of the three components of the dialogue states: Goal constraints, Methods and Requested Slots.

For **Goal-tracking**, the Focus Tracker outputs a distribution over all informable slot-value pairs in the ontology and their joint distribution of these goals. This is done in the script below:

```
# --- 1. goal --- #
for slot in set(this_u.keys() + hyps["goal-labels"].keys()):
   q = max(0.0,1.0-sum([this_u[slot][value] for value in this_u[slot]])) # clipping at zero
   if slot not in hyps["goal-labels"] :
      hyps["goal-labels"][slot] = {}
   # Update rule
   for value in (hyps["goal-labels"][slot].keys() + this_u[slot].keys()):
      hyps_old = hyps["goal-labels"][slot].get(value, 0.0)
      hyps["goal-labels"][slot][value] = this_u[slot][value] + q * hyps_old
   # normalise the score of each value in a slot
   hyps["goal-labels"][slot] = normalise_dict(hyps["goal-labels"][slot])
# ------------ #
```

The Focus Tracker outputs a distribution over all the possible methods together with the no-action method 'None', which requires a separate handling in the implementation:

```
# --- 2. method --- #
method_label = hyps["method-label"]
# your code here, modify the following update rule
q = max(0.0,1.0-sum([value for method, value in method_stats.items() if method!='none']))
for method in (method_label.keys() + method_stats.keys()):
    slu_method = method_stats.[method]
    if method in method_label.keys():
        hyps_old = method_label[method]
        method_label[method] = slu_method + q * hyps_old
    else:
        method_label[method] = slu_method
# After exiting the loop 'None' method is computed as complementary of other methods
val_other_methods = sum([value for method, value in method_label.items() if method!='none'])
method_label['none'] = max(0.0, 1.0 - val_other_methods)
# normalise the score
hyps["method-label"] = normalise_dict(method_label)
# -------------- #
```

For each Requested-slot, the Focus Tracker outputs a binary distribution with the probability p of such slot been requested. Hence, a similar implementation to Goal and Method tracking can be used in **Requested-slot tracking**, simply by setting $q = 1 - p$. It is worth noticing that once the system has informed about a particular slot, it leaves the requested-slot set and its p value is set to 0.

```
# --- 3. requested slots --- #
informed_slots = []
for act in mact :
    if act["act"] == "inform" :
        for slot,value in act["slots"]:
            informed_slots.append(slot)
for slot in (requested_slot_stats.keys() + hyps["requested-slots"].keys()):
    p = requested_slot_stats[slot]
    if slot in informed_slots:
        p = 0.0
    else:
        p += (1 - p) * hyps["requested-slots"].get(slot, 0.0)
# clip the score
hyps["requested-slots"][slot] = clip(p)
# ----------------------- #
```

The tables below show the performance of *Baseline Tracker* and the *Focus Tracker* on the `dstc2` dataset, using Accuracy, L2[1]and a ROC-curve related measure as evaluation metrics.

	Joint Goals	**Requested Slots**	**Methods**
Accuracy	0.569	0.914	0.682
L2	0.835	0.122	0.576
`roc.v2`	0.000	0.606	0.002

Table 1: Performance of Baseline Tracker

	Joint Goals	**Requested Slots**	**Methods**
Accuracy	0.739	0.919	0.684
L2	0.423	0.120	0.566
`roc.v2`	0.000	0.320	0.022

Table 2: Performance of Baseline Tracker

The Focus Tracker outperforms the Baseline Tracker across all evaluation metrics on Joint Goals. The decrease in L2 s specially relevant, as it relates to the entire true distribution of the belief-states. This goes to indicate the robustness of the Focus Belief Tracker in order to adapt to new constraint goals, as well as a better understanding of the belief state dynamics. In addition, almost no difference is observed between both belief trackers on Requested-slots and Methods. Thus, it seems (in this dataset at least) that current SLU evidence suffices to track these dialogue-state components.

3 Dialogue Policy Optimisation

In Dialogue Policy Optimisation, the current belief-state distribution b_t is used to to select the next system dialogue act. As we discussed before, This decision-making problem can be approached as a Reinforcement Learning task, where the goal is to learn *policy* (dialogue-act selection rule) that maximises a *reward function* (some dialogue-quality measure) though user simulation. In this section, we discuss two model-free methods for Dialogue Management: *Monte Carlo control Algorithm* and *GP–SARSA Algorithm.*

3.1 Monte Carlo control Algorithm

Monte Carlo control Algorithm (MCC) is model-free method that aims to learn an approximated optimal policy by updating estimates based on raw simulated experience. In particular, the Q-value function estimates are updated by averaging sample returns once the full episode (user-system dialogue) is completed. Then, policy estimates are updated with respect to Q-value function estimates in a greedy way.

In this section, we implement MCC using a dictionary grid of belief-action pairs (b_t, a_t) as a tabular representation of the Q-value function. The dictionary size is controlled by a *sparsification threshold* ν, which will be fundametal for the performance of the system.

[1]L2 metric $= (1 - p_i)^2 + \sum_{j \neq i} p_j^2$, where p_i = probability assigned to the true label, is a quantity to minimise.

First, we implement **Episode Generation**, where given a current belief-state b_t, next action a_t is selected from the dictionary grid with respect to current Q-value estimate in ϵ-greedy fashion. We set ϵ to decay in the `.cfg` file to allow for a balanced exploration/exploitation throughout the episode. This is done in the script below:

```
Q_b = []
for a in admissible:
    closestDP, closestValue = self.findClosest(flat_belief, a)
    if closestDP in self.dictionary.keys():
        Q_b.append(self.dictionary[closestDP].Q)
    else: # closestDP == None
        Q_b.append(0.0)
index = Q_b.index(max(Q_b))
action = admissible[index]
```

In **Q-Value Update**, we add a point (b_t, a_t) to the Q-value dictionary grid if its distance to the nearest grid point is greater than the sparsification threshold ν, setting its Q-value to the `Return` obtained. Otherwise, we only update the Q-value of the nearest grid point by averaging with the sampled `Return` corresponding to (b_t, a_t). We implement this in the script below:

```
if len(self.dictionary) == 0.0:  # add (b,a) with Q=Return and N=1.0
    logger.debug('starting a new dictionary')
    DP = DataPoint(b, a)
    DPValue = DataPointValue(Q=Return, N=1.0)
    self.dictionary[DP] = DPValue
    pass
elif closestDP is None or closestValue > self.nu: # add (b,a) with Q=Return and N=1.0
    logger.debug('adding new point')
    DP = DataPoint(b, a)
    DPValue = DataPointValue(Q=Return, N=1.0)
    self.dictionary[DP] = DPValue
    pass
else: # update Q and N with monte carlo algorithm
    logger.debug('updating Q & N of the grid point')
    DPClass = self.dictionary[closestDP]
    Q_update = (DPClass.Q * DPClass.N + Return) / (DPClass.N + 1)
    N_update = DPClass.N + 1
    self.dictionary[closestDP] = DataPointValue(Q_update, N_update)
    pass
```

Figure 2, below shows the importance of the sparsification threshold ν. Large ν (e.g.: 0.1 and 0.05) results in poor system performance due to a lack of dictionary grid points to perform an adequate estimate of the Q-function. We can also observe how smaller ν values do provide greater success rates due to quality estimates of the Q-function. However, these MCCs are more expensive do to the extensive memory required.

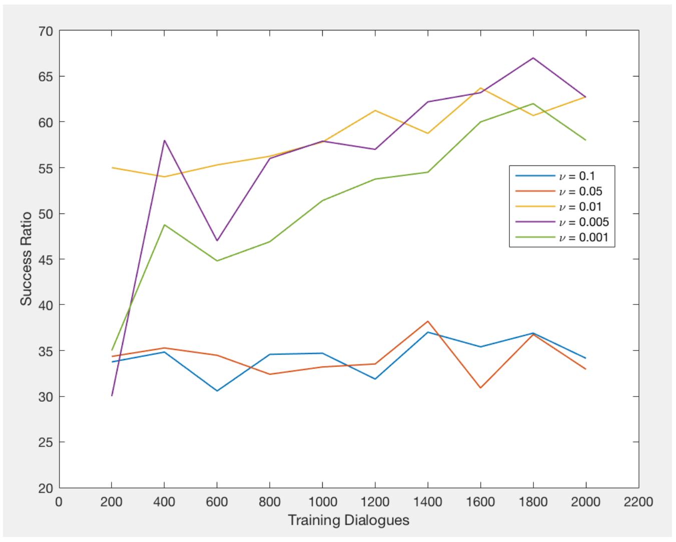

Figure 2: MCC learning curves as sparsification threshold ν varies

3.2 GP–SARSA Algorithm

GP–SARSA Algorithm non-parametric policy optimisation technique that models Q-value function as Gaussian Process:

$$Q(b,a) \sim GP(0, k((b,a),(b,a))) \tag{3}$$

where the *Kernel function* $k(\cdot,\cdot)$ is gives a measure of the correlation between different belief-action pairs (b,a). In order to compute Kernel function estimate updates throughout the learning process, we employ the Temporal Difference method, known as SARSA. Q-value function is then updated by evaluating the predictive posterior of (3) given all the belief-action pairs (b,a) and their corresponding rewards. This Q-value function estimate is significant improvement compared to nearest-neighbour interpolation employed in MCC.

However, the computation of the predictive posterior requires the evaluation of the Gram-Matrix K at (N) memorised data-points:

$$K_{ij} = k((b_i,a_i),((b_j,a_j)) = \langle \phi(b_i,a_i), \phi(b_j,a_j) \rangle$$

in terms of the product of feature vectors $\phi(b,a)$, which has a high computational cost of O(N^3). In practice, we only consider a subset of these points forming a dictionary of representative points $(\tilde{b},\tilde{a})$, which considerably reduces the computational cost involved and give a good approximate $\tilde{K}$. Similarly to MCC, we choose sparsification threshold ν to control the dictionary size. In the script below we implement Linear Gaussian kernel function

$$k((b_i,a_i),((b_j,a_j)) = \langle b_i, b_j \rangle \delta_{a_i}(a_j)$$

```
k_tilda_new = self.k_tilda(state, action, kernel)
g_new = []
if self.terminal:
    g_new = np.zeros(len(self._dictionary))
else:
    pass
    g_new = np.dot(self._K_tilda_inv, k_tilda_new)
# Modification of current_kernel and estimate_kernel
current_kernel = np.dot(kernel.Kernel(state, state), kernel.ActionKernel(action, action))
estimate_kernel = np.dot(np.transpose(k_tilda_new), g_new)
delta_new = 0.0 if self.terminal else (current_kernel - estimate_kernel)
```

A great benefit of Gaussian Processes is that they provide a measure of uncertainty in the regression of the Q-value function. GP-SARSA exploits this by directing the exploration towards belief-states with higher uncertainty, which results in more efficient learning compared to ϵ-greedy methods in MCC for exploitation/exploration. Consequently, while MCC methods require extensive training to provide with good estimates of the Q-value function, GP-SARSA allows for rapid optimal policy learning with fewer samples. We can observe this effects in Figure 3, where the best-MCC with sparsification threshold $\nu = 0.01$ is outperformed by both GP-SARSA models. No significant difference is observed between GP-SARSA models: both show a rapid learning rate, reaching values of 75% and greater success rates. In any case, GP-SARSA with Gaussian Kernel seems to perform slightly better than GP-SARSA with the linear model, due to the tuning of the length-scale parameter l in:

$$k((b_i, a_i), ((b_j, a_j)) = p^2 exp(-\|b_i - b_j\|_2/2l^2)\delta_{a_i}(a_j)$$

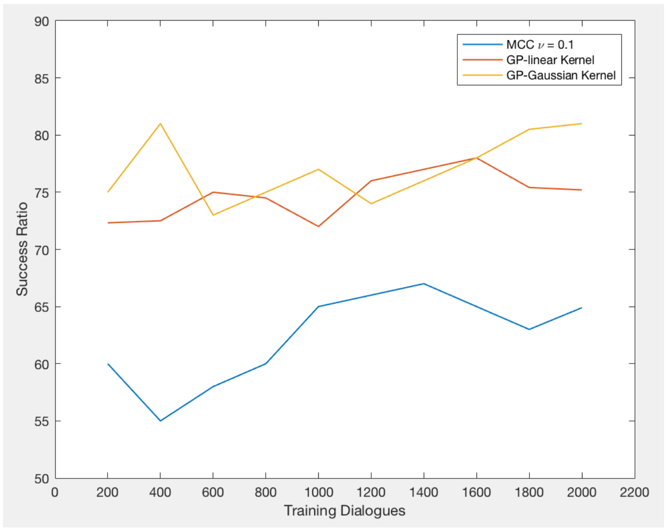

Figure 3: Learning curves of GP-SARSA with Linear and Gaussian Kernels; and best-MCC