Multi-Domain Dialogue State Tracking

Introduction
Dialog State Tracking (DST)
    Simple Classification
    Entity Recognition
    TRAnsferable **D**ialogue stat**E** generator (**TRADE**)
TRADE Model
    Utterance Encoder
    State Generator
        1. Decoder
        2. Probability distribution over vocabulary
        3. Probability distribution over history
        4. Final probability distribution
        5. Trainable scalar to control copying vs generating ability
        6. Context vector
    Slot Gate
    Optimization
        Slot gate loss
        State generator loss
Results
    Evaluation Metrics
        Joint Accuracy
        Slot Accuracy
    Analysis
References

Introduction

Last summer our team of data scientists attended the 57th Annual Meeting of the Association for Computational Linguistics (ACL) at Florence. It was a great learning and team bonding experience for all of us(as you can see in the image below).

After learning a lot for three days in the conference these neurons just decided to dropout on the couch a.k.a. taking a quick nap.

During the conference we learned about this really interesting paper - Transferable Multi-Domain State Generator for Task-Oriented Dialogue Systems. It received two awards in the conference, an outstanding paper award the best paper award at NLP for Conversational AI Workshop.

Currently at ebot7 we are building a system (which we call Conversational Engine) for managing conversations by keeping track of dialog states and then deciding which action to take based on the current state and context. To this end we wanted to do a small proof of concept on the above mentioned paper and through this blog we will share our findings. In this first post we want to share an intuitive and detailed explanation of the proposed architecture.

Dialog State Tracking (DST)

The main goal of DST is to extract information from dialogs which will enable a dialog system to appropriately manage conversations. For instance, in task oriented systems, such as ticket booking, it is essential to gather information about *departure* *location, departure time, destination* etc to better understand a user's intention and then take appropriate action or collect more information. The job of DST is to extract this information and this extracted information is what defines the state of a dialog. We can use different formalization to represent a state eg. it can either be represented in the form of a list of `(domain, slot, value)` tuples or we can use a compact version where we convert these tuples into vectors.

Simple Classification

One way of approaching the DST problem is to transform it into a classification problem where each `(domain, slot, value)` tuple would be considered as one class. This approach is very simple and intuitive but, it has some drawbacks.

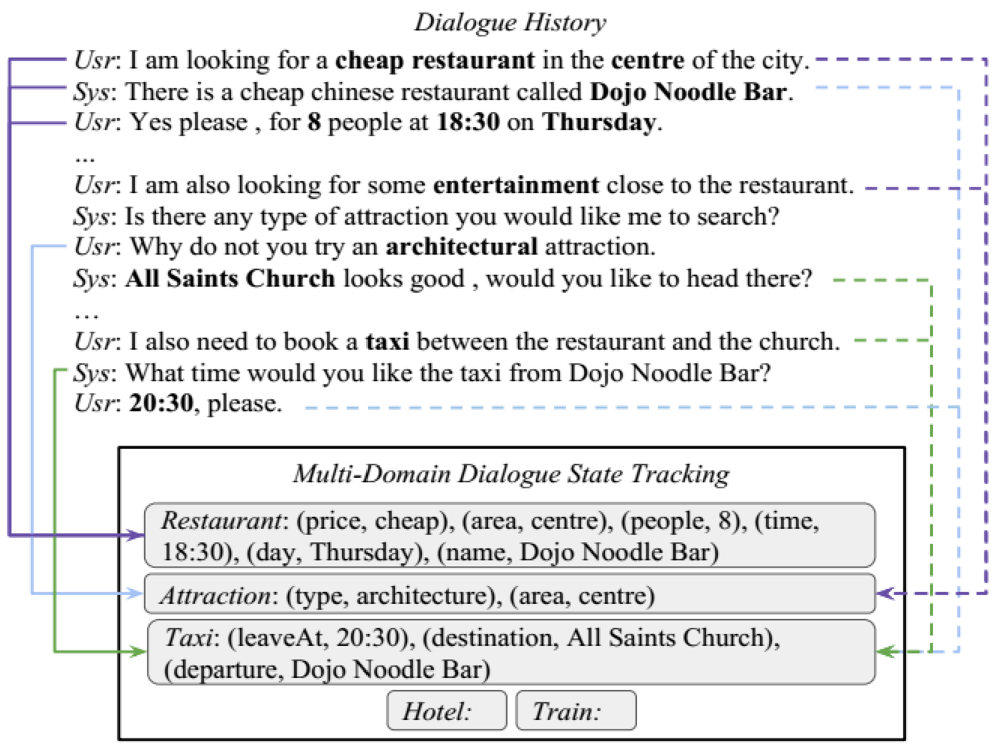
1. We need to know all the possible `(domain, slot, value)` combinations in advance. For many slots such as *departure time* it is impractical to generate all the combinations.
2. Every time we need to detect a new `(domain, slot, value)` combination we need to retrain the model.

Entity Recognition

Another approach could be to transform this into an entity recognition problem. In this, `(slot, value)` tuple gets mapped to `(entity type, entity value)`. So, what we need to do here is, extract entity values and then map the extracted values to the appropriate tags. Although it circumvents some of the problems (e.g in some entity types we don't need all the combinations in advance) which we had in the previous approach but, it still has its own shortcomings.

1. We still need to retrain the model whenever we want to detect a new entity type.
2. We need to think of ways to uniquely map the extracted entity values to their correct slots. For instance, a location entity value can be mapped to both *destination* and *departure* slots.

TRAnsferable Dialogue statE generator (TRADE)


Dialogue History

Usr: I am looking for a **cheap restaurant** in the **centre** of the city.
Sys: There is a cheap chinese restaurant called **Dojo Noodle Bar**.
Usr: Yes please , for **8** people at **18:30** on **Thursday**.
...
Usr: I am also looking for some **entertainment** close to the restaurant.
Sys: Is there any type of attraction you would like me to search?
Usr: Why do not you try an **architectural** attraction.
Sys: **All Saints Church** looks good , would you like to head there?
...
Usr: I also need to book a **taxi** between the restaurant and the church.
Sys: What time would you like the taxi from Dojo Noodle Bar?
Usr: **20:30**, please.

Multi-Domain Dialogue State Tracking

Restaurant: (price, cheap), (area, centre), (people, 8), (time, 18:30), (day, Thursday), (name, Dojo Noodle Bar)
Attraction: (type, architecture), (area, centre)
Taxi: (leaveAt, 20:30), (destination, All Saints Church), (departure, Dojo Noodle Bar)
Hotel:
Train:

Figure 1: *An example of multi-domain dialogue state tracking in a conversation. The solid arrows on the left are the single-turn mapping, and the dot arrows on the right are multi-turn mapping. The state tracker needs to track slot values mentioned by the user for all the slots in all the domains.*

The third approach is called TRADE and it is proposed by the authors of this paper that we are reviewing. This approach is capable of tracking states from multiple domains and across multiple turns. Following are some of the main benefits of this approach.

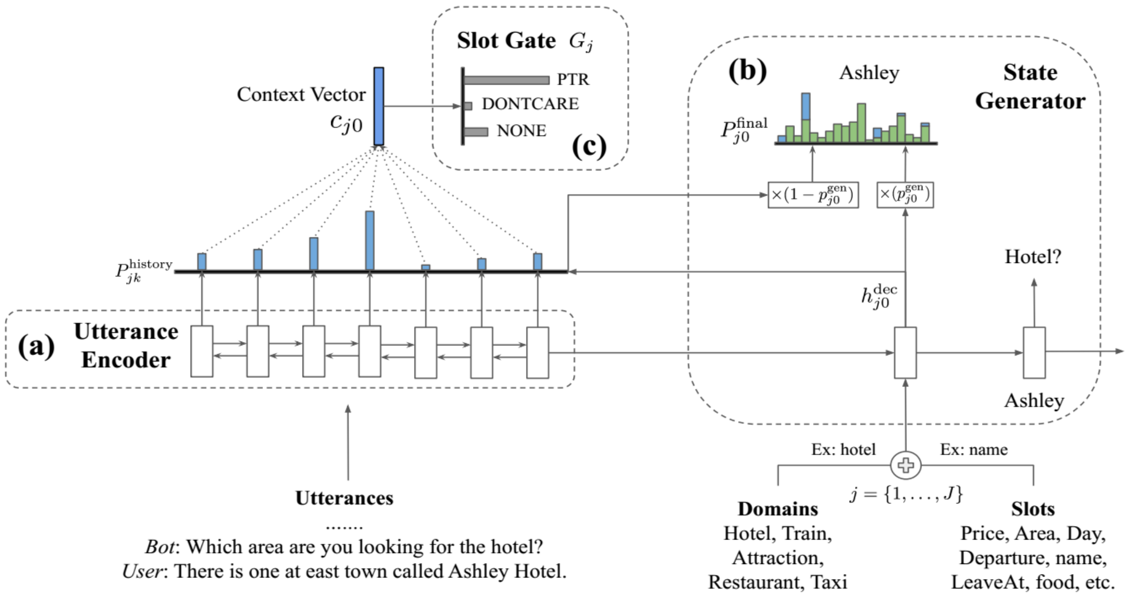
1. This framework can be easily applied on multiple domains i.e you can train the model by combining data from multiple domains and then use this single model to perform inference on dialogues from multiple domains. The motivation for doing this comes from the fact that there are many slots that are common across domains. For instance, *area* slot can exist in *restaurant* as well as in *hotel* domain. This benefit is not unique to the TRADE approach, classification and entity recognition approaches can also take advantage of this fact.

2. The TRADE model also takes into account context from previous turns while predicting state for the current turn. The motivation for doing this comes from the fact that sometimes values of slots in the current turn can be extracted from the previous turns. For instance, in the dialogue shown in figure1 the *name* slot in the *restaurant* domain can share the same value with the *departure* slot in the

taxi domain. Hence, the value for the *departure* slot in the *taxi* domain can be extracted from the previous turns,

3. By doing multi-domain training i.e sharing model parameters across multiple domains the model is also capable of tracking states on unseen domains with the help of zero-shot learning and few-shot learning. This is definitely a really good benefit to have but unfortunately, the results for zero-shot learning and few-shot learning presented in the paper are not that impressive.
4. The model uses a combination of copy vs generate mechanism for predicting slot values. For this, it combines a probability distribution over vocabulary and a distribution over the dialogue history. This gives the model an ability to generate words for slot values even if they are not present in the dialogue history.

TRADE Model



Slot Gate G_j
PTR
DONTCARE
NONE
(c)
Context Vector c_{j0}
(b)
Ashley
State Generator
P_{j0}^{final}
$\times(1-p_{j0}^{\text{gen}})$
$\times(p_{j0}^{\text{gen}})$
P_{jk}^{history}
Hotel?
h_{j0}^{dec}
(a) Utterance Encoder
Ashley
Ex: hotel
Ex: name
$j = \{1, \ldots, J\}$
Utterances
....
Bot: Which area are you looking for the hotel?
User: There is one at east town called Ashley Hotel.
Domains
Hotel, Train, Attraction, Restaurant, Taxi
Slots
Price, Area, Day, Departure, name, LeaveAt, food, etc.

Figure 2: The architecture of the proposed TRADE model, which includes (a) an utterance encoder, (b) a state generator, and (c) a slot gate, all of which are shared among domains. The state generator will decode J times independently for all the possible (domain, slot) pairs. At the first decoding step, the state generator will take the j-th (domain, slot) embeddings as input to generate its corresponding slot values and slot gate. The slot gate predicts whether the j-th (domain, slot) pair is triggered by the dialogue or not.

After having a look at salient features of the TRADE model in the previous section let us now formally define each component of the model as shown in figure 2.

Let `math:X` be a dialogue consisting of `math: T` turns. Each turn is either a user utterance `math:U` or a system response `math: R`. Let `math:B` represent a set of dialogue states for each turn. Each state `math:B_t` is defined as a tuple of *(Domain* `math:D_n`*, Slot* `math:S_m`*, Value* `math:Y_j`*)*. There are `math:N` different domains and `math:M` different slots and `math:J` possible `(domain, slot)` combinations. As shown in the following equations *(*`math:D_n`*,* `math:S_m`*)* is the `math:j^{th}` domain-slot combination with `math:Y_j^{value}` as the true slot value(consisting of sequence of words) for this `math:j^{th}` combination.

$$X = (U_1, R_1), ..., (U_T, R_T)$$

$$B = B_1, ..., B_T$$

$$S = S_1, ..., S_M$$

$$D = D_1, ..., D_N$$

$$B_t = (D_n, S_m, Y_j^{value})$$

Utterance Encoder

The paper uses a bi-directional GRU as utterance encoder but in practice you can also use any other model. The input to the encoder is a subset of dialogue called history and denoted by `math:X_t`. It's simply a concatenation of all the words in the last `math:l` turns of utterances and responses.

$$X_t = [U_{t-l}, R_{t-l}, ..., U_t, R_t] \in \mathbb{R}^{|X_t| \times d_{emb}}$$

The value of `math:l` controls how much history we want to consider for predicting states. `math:|X_t|` denotes the number of words/tokens in the history, `math:d_{emb}` denotes word embedding size.

The output `math:H_t` of the encoder is a set of `math:|X_t|` number of encoded words where `math:d_{hdd}` denotes the GRU output size.

$$H_t = [h_1^{enc}, ..., h_{|X_t|}^{enc}] \in \mathbb{R}^{|X_t| \times d_{hdd}}$$

State Generator

The goal of the state generator is to generate slot values either by copying words from dialogue history or by selecting words from the available vocabulary. In order to achieve this, the state generator produces two probability distributions, one over the vocabulary and another over the history and then combines them to a final probability distribution. Words are picked using this final probability distribution. State generator is applied independently on all the `math:J` *`(domain,sot)`* combinations. State generator is composed of the following 6 components.

1. Decoder

A GRU is used as a decoder and it's goal is to generate decodings which can then be used for predicting slot values for each *`(domain, slot)`* combinations. The first input denoted by `math:w_{j0}` to the decoder for the `math:j^{th}` *`(domain, slot)`* combination is the summed embedding of the domain and slot as shown in figure 2. The slot value predicted by the state generator is passed as the input to the next time step of the decoder. This is done until an end marker is reached. An important thing to note here is that the slot generator is applied independently on all the `math: J` combinations, each having it's own different number of decoding timesteps. For instance, the decoder for `math:j^{th}` combination at `math:k^{th}` decoding step takes in a word embedding `math: w_{jk}` (computed in the `math: k-1^{th}` step) as input and returns a hidden state `math: h_{jk}^{dec}` as output.

2. Probability distribution over vocabulary

The next step is to somehow map decoder hidden state `math: h_{jk}^{dec}` into the vocabulary space `math: P_{jk}^{vocab}`. This is done by learning a transformation matrix `math:E \in\mathbb R^{|V|\times d_{hdd}}`, where `math:|V|` is the vocabulary size and `math:d_{hdd}` is the hidden vector size.

$$P_{jk}^{vocab} = Softmax(E.(h_{jk}^{dec})^T) \in \mathbb{R}^{|V|}$$

In other words, `math: P_{jk}^{vocab}` effectively tells us which word to select out of the vocabulary `math: V` for the `math:k^{th}` slot value word in the `math: j^{th}` combination pair.

3. Probability distribution over history

Similarly, at the same time the hidden state `math: h_{jk}^{dec}` is also used to compute some sort of an attention distribution `math:P_{jk}^{history}` over the encoded dialogue history `math: H_t`. Intuitively, the goal here is to find words from the history that can be copied as slot values for a particular *`(domain, slot)`* combination.

$$P_{jk}^{history} = Softmax(H_t.(h_{jk}^{dec})^T) \in \mathbb{R}^{|X_t|}$$

4. Final probability distribution

Now, we compute the final probability distribution `math:P_{jk}^{final}` over the vocabulary `math: V` by computing a weighted sum of the probability distribution over vocabulary `math:P_{jk}^{vocab}` and probability distribution over history `math:P_{jk}^{history}`.

$$P_{jk}^{final} = p_{jk}^{gen} \times P_{jk}^{vocab} + (1 - p_{jk}^{gen}) \times P_{jk}^{history} \in \mathbb{R}^{|V|}$$

The scalar weight `math: p_{jk}^{gen}` controls how much importance we want to give to selecting words from the vocabulary vs copying words from the history. Due to this the model is able to generate slot values even if they are not present in the dialogue history.

5. Trainable scalar to control copying vs generating ability

The scalar weight `math: p_{jk}^{gen}` is not a hyperparameter instead it is learned during the training. Following equation tells us how to compute this scalar weight.

$$p_{jk}^{gen} = Sigmoid(W_1.[h_{jk}^{dec}; w_{jk}; c_{jk}]) \in \mathbb{R}^1$$

`math: W_1` is a trainable matrix and `math:c_{jk}` is the context vector. The value of `math:p_{jk}^{gen}` range between [0,1]. Higher value means that model is in support of generating a word from vocabulary rather than copying it from the history.

6. Context vector

The context vector `math:c_{jk}` stores a condensed form of information from the recent dialogue history. It is simply computed as a weighted sum over the encoded history `math: H_t`.

$$c_{jk} = P_{jk}^{history}.H_t \in \mathbb{R}^{d_{hdd}}$$

Slot Gate

The goal of the slot gate is to predict whether a *`(domain, slot)`* combination is present in the dialogue or not. Slot Gate `math: G` is a classifier that takes in the very first context vector as input and generates a probability distribution over three output classes *`(ptr, none, dontcare)`*.

$$G_j = Softmax(W_g.(c_{j0})^T) \in \mathbb{R}^3$$

So in practice we need to perform slot gate check only once at `math:0^{th}` timestep for all the `math: J` combinations. If the gate produces either *none* or *dontcare* then the generator values are ignored and that particular pair of `(domain,slot)` combinations is considered as being absent from the dialogue. On the other hand if the gate produces *ptr* then the words generated by the generator are considered as slot values.

Optimization

The model is learned by optimizing for both the slot gate and the state generator. The final loss is a weighted sum of slot gate loss `math:L_g` and state generator loss `math:L_v`.

$$L = \alpha L_g + \beta L_v$$

The weights `math:\alpha` and `math:\beta` are hyperparameters.

Slot gate loss

For the slot gate loss a cross-entropy is computed using the predicted classes `math:G_j` and the ground truth one-hot labels `math:y_j^{gate}`.

$$L_g = \sum_{j=1}^{J} -log(G_j.(y_j^{gate})^T)$$

State generator loss

For the generator loss, another cross-entropy loss is computed between the final probability distribution over words `math:P_{jk}^{final}` and the true words of slot values `math: Y_j^{label}`. The inner summation is over the words of a slot value and the outer one is over all the `math:J` combinations.

$$L_v = \sum_{j=1}^{J} \sum_{k=1}^{|Y_j|} -log(P_{jk}^{final}.(y_{jk}^{value})^T)$$

Results

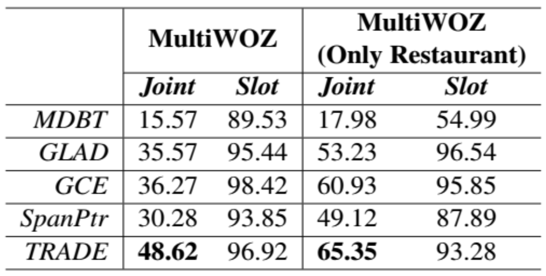
Evaluation Metrics

Joint Accuracy

It measure the predicted dialogue states to the ground truth states `math: B_t` at each dialogue turn `math: t`. A prediction is considered correct if and only if all the predicted values of all the states in a turn exactly match the ground truth values.

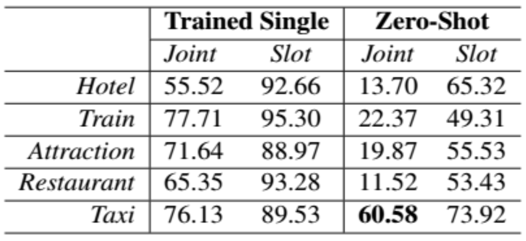
Slot Accuracy

It is a slightly lenient metric compared to the joint accuracy discussed above. In this, each `(domain, slot, value)` triplet is independently compared with its ground truth label.



	MultiWOZ		**MultiWOZ (Only Restaurant)**	
	Joint	***Slot***	***Joint***	***Slot***
MDBT	15.57	89.53	17.98	54.99
GLAD	35.57	95.44	53.23	96.54
GCE	36.27	98.42	60.93	95.85
SpanPtr	30.28	93.85	49.12	87.89
TRADE	**48.62**	96.92	**65.35**	93.28

Table 1: The multi-domain DST evaluation on MultiWOZ and its single restaurant domain. TRADE has the highest joint accuracy, which surpasses current state-of-the-art GCE model.



	Trained Single		**Zero-Shot**	
	Joint	*Slot*	*Joint*	*Slot*
Hotel	55.52	92.66	13.70	65.32
Train	77.71	95.30	22.37	49.31
Attraction	71.64	88.97	19.87	55.53
Restaurant	65.35	93.28	11.52	53.43
Taxi	76.13	89.53	**60.58**	73.92

Table 2: Zero-shot experiments on an unseen domain. In taxi domain, our model achieves 60.58% joint goal accuracy without training on any samples from taxi domain. Trained Single column is the results achieved by training on 100% single-domain data as a reference.

Analysis

1. TRADE model has the highest joint accuracy even better than the current SOTA.
2. Zero shot performance of taxi domain is pretty good but it is not so impressive for other domains.
3. All the domains in the dataset are somewhat similar to each other and also have some common slots. It is because of this reason that the taxi domain's zero-shot performance is high, since it shares similar slots with the train domain.

For more results and in-depth error analysis please refer the paper. Since the source code is publicly available we are also planning to perform our own analysis on some real world data. Stay tuned, we will be publishing the second part of this blog post with our results very soon.

References

1. https://arxiv.org/abs/1905.08743
2. https://github.com/jasonwu0731/trade-dst