Socrates Sim: A User Simulator to Support Task Completion Dialog Research

Dhairya Dalal

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

August 2018

Contents

Chapter 1: Implementation

1.1. Overview

In this chapter, we will the deployment of the Socrates framework in two domains, restaurant recommendation and movie tickets. We will demonstrate how the framework's robustness, modularity, and ability to be re-targeted across new domains. For each domain, we describe the setup configurations used, implementation details for the user simulator, dialog agent, and the various constituent models implementations like natural language generation and knowledge base querying. Note the full implementation code can be found in the appendix. We will focus on the strategies deployed and in particular the flexibility of the framework to handle different implementations.

1.2. Restaurant Recommendations

1.2.1 Overview

The restaurant recommendation domain focuses on developing a chat-bot service that makes restaurant recommendations. The objective of the Socrates simulator is to produce a set of simulated conversations between the user simulator and Restaurant Recommender dialog agent. The dialog agent will attempt to elicit as much information about the user's preferences and then attempt to provide a useful recommendation. In this task completion exercise, both the user and the dialog agent can take the first turn.

As mentioned in the Design chapter, the framework is driven by configuration files. The researcher can easily modify and run different experiments through updated a human readable yaml file or a programmatically generated json file. For this use case, four configuration files are created to capture the following information:

- Simulation settings: contains all the details around simulation criteria (e.g. number of rounds, first speaker, save setting, etc) as well the configuration details for dialog domain, user simulator and dialog agent.

- Dialog domain: describes the dialog action space, inform and request slots, and specifies valid user goals templates.

- NLG templates for user simulator and agent: a rules based template to generate natural language utterances

Additionally, we created custom modules to implement the user simulator and dialog agent for an end-to-end dialog simulation.

We used the Dialog State Tracking Challenge 2 (DSTC2) restaurant data and service as a model for our use case. The DSTC2 is research challenge put together by the University of Oxford and Microsoft to advance dialog research. For DSTC2, the goal was track the state of multi-stage conversations between real humans and a expert restaurant recommender service. The restaurant recommender service would provide recommendations based off the following user preferences: cuisine, area, and price range. DSTC2 also provided a rich and deep set of training data that allowed to model both neural network based approaches for NLG and NLU, as well as rule based approaches.

Below we will describe our strategy for developing the user simulator and dialog agent using the Socrates Simulator Framework and implementation choices for their constituent parts.

1.2.2 Domain

Figure 1.1: Restaurant Dialog Domain

```
# Dialog Action Space
dialog_acts: [ inform, confirm, affirm, request,
                negate, greetings, bye]

# Request Slots
request_slots: [ address, area, cuisine,
                 phone, pricerange, postcode, name ]

# Inform slots
inform_slots: [ cuisine, pricerange, name, area ]

# Valid User Goals Temaplates
valid_user_goals:
- [ name ]                    # User wants restaurant name
- [ name, address ]           # User wants name and address
- [ name, address, phone ]    # User wants name, address, and phone
- [ name, phone ]             # Use wants name and phone
```

The first thing we setup is the dialog domain configuration file. This file is used to generate a domain object that will be used by both the user simulator and the dialog agent. For restaurant recommendations, we used dialog domain described in the DSTC2 ontology. The figure above details the s dialog action space, and the different inform and request slot types. The dialog domain was expressed as a yaml file, where each key captured the salient attributes of the domain.

One important feature of this file is the valid user goals templates section. It specifies the valid types of goals a user may have when engaging the dialog agent. The user simulator will use it generate random goals that can explore the preference space and test the robustness of the agent.

Note, the figure above does not include the all the possible inform and request slot values. Please see the [appendix ref] for the full configuration file.

1.2.3 Domain Knowledge Base

The knowledge base for this use case is csv with various restaurants from the Cambridge area. Each row represents a unique restaurant and the column values capture the restaurant's cuisine, price range, area, phone number, and address. The data was scraped by Yelp. For the scraping script, see the appendix. The restaurants list was loaded as DomainKBtable object. The DomainKBtable loads the csv files as in memory pandas dataframes and provides a a set of methods for querying.

1.3. Natural Language Understanding Implementations

The goal of NLU implementation is to parse a natural language utterance into a DialogAction object. The parsed dialog action contains the intent of the utterance, i.e. the dialog act, and any entity / entity types, i.e. the dialog parameters, which are contained in the utterance. A rules based model and neural model were implemented to demonstrate the different models a researcher can use to support the natural language understanding.

Rules Based NLU. For the rules based approach, our parser has two part strategy. The first is to classify the intent of the utterance and map it to a dialog act. The second to is run an entity extraction pass and attempt to extract the entities contained in the utterance and map them to inform slot types.

The algorithm for the intent classification consists of two parts. The first is running the utterance through a question classifier (implemented from Chewning et al. (2015)). If the utterance is a question, we classify it as a request dialog act. Otherwise run though a set of regular expression matches and return the corresponding dialog acts (see 1.2). Given the wide range of potential string matches for inform, we use that as the default dialog act.

Figure 1.2: Intent Classifier Algorithm

```
Classify Intent
        input: natural languge utterance

        IF input is a question:
                return ’request’
        ELSE IF input contains [you, you want, right?]:
                return ’confirm’
        ELSE IF input contains [yes, yeah, yup, correct, right]]:
                return ’affirm’
        ELSE IF input contains [no, nope, wrong, incorrect]:
                return ’negate’
        ELSE IF input contains [hi, hello]:
                return ’greetings’
        ELSE IF input contains [bye, goodbye, thanks, thank you]:
                return ’bye’
        ELSE
                return "inform"
```

After the intent is classified, we then check to see if any entities are found in the utterance that can be mapped to entity types. To achieve this, we first lookup all the slot values defined in the domain object. Next, we create a reverse map dictionary, where each unique slot value (i.e. the entity) is mapped to a slot type (e.g. cuisine or price range). The NLU parser then tokenizes the utterance into set a of word trigrams and check if a slot type exists for the token in the reverse map. All positive matches are added to the dialog params list in the DialogAction object. Finally, the parse utterance will return a DialogAction object with parsed dialog act and a list of dialog parameters.

Neural Model for NLU. We also implemented a simple neural machine translation model for our NLU module.

The DSTC2 provides a large labeled dataset of conversations between human users and an human expert posing as the dialog agent. Between the train and development set were about 2000 annotated calls. All speech utterances (between the human and agent) were parsed and annotated. To train the NMT model, we extracted out all utterances and the corresponding parses (expressed as json string objects). The model was trained over X epochs and released.

SERT NMT model diagram.

1.4. Natural Language Generation Implementations

The objective of the NLG module is to generate a natural language utterance, provided a dialog action. For the restaurant recommendation use case, we implemented both a template based model and a neural model.

Template Based Model. The template model is defined by a yaml file (see figure 1.3). The nlg template is loaded into memory as a nested python dictionary. The first layer of keys are indexed by the dialog acts (e.g. request, inform, etc), and the corresponding values are dictionaries indexed by specific slot types. In the case where there are the are no slot types (e.g. affirm), the default value is used. The natural language templates are stored in lists at the values for the slot types.

Figure 1.3: Example NLG template for User Simulator

```
affirm:
       default: [ "Yes.", "Yup.", "Yes, that’s right." ]
greetings:
       default: [ "Hi, I’m looking for a restaurant.",
                  "Hi! Can you help me find a restaurant?" ]
inform:
       cuisine: [ "I’d like find a restaurant that serves $CUISINE.",
                  "I’m looking for $CUISINE food.",
                  "I want to eat $CUISINE food." ]
       pricerange: [ "I’m looking for a $PRICERANGE priced restaurant.",
                     "Looking $PRICERANGE priced food." ]
```

The logic then is straight forward to generating a natural language utterance.

The get utterance method in simulator will be passed a dialog action object which contains the dialog act and a list dialog parameters. We first look up the dialog act in the nlg template dictionary. Next we look up the slot values (if any) for the specific language templates. The slot values are passed in the dialog params property of the DialogAction object. Additionally, language templates that have multiples slot types, are indexed by combination of the slot types into a single string. We take the slot types, lower case them, arrange them by alphabetical order, and concatenate them together with comma separator into a single string. For example, "I want $PRICE $CUISINE" would be indexed by the string "cuisine,price". Finally, we randomly sample the list from the list of potential language template, substitute slot values, and return a generated natural language utterance.

Neural Model for NLU. We also implemented a simple neural machine translation model for our NLU module.

The DSTC2 provides a large labeled dataset of conversations between human users and an human expert posing as the dialog agent. Between the train and development set were about 2000 annotated calls. All speech utterances (between the human and agent) were parsed and annotated. To train the NMT model, we extracted out all utterances and the corresponding parses (expressed as json string objects). The model was trained over X epochs and released.

SERT NMT model diagram.

1.4.1 User Simulator

We designed a rule based user simulator given the simplicity of the dialog domain. In this use case, the user has a set of hidden preferences and is looking get name of a restaurant that satisfies those preferences from the dialog agent. Additionally, the user may also want to get some the restaurant's phone number and address.

At the start of each dialog round, the user simulator will be provided a goal from the Dialog Manager. For demonstration purposes we defined both an explicit set of user goals and valid goal template for random goal generation The figure below shows what a sample goal would look like.

Figure 1.4: Example User Goal. User is seeking the name and phone number of a cheap Chinese restaurant

```
inform_slots:
        cuisine: "chinese"
        pricerange: "cheap"
request_slots:
        name: "UNK"
        phone: "UNK"
```

The rules simulator we designed is a separate python module that will be dynamically loaded by the Dialog Manager at simulation time. The rules simulator class inherits the base UserSimulator class, which in turn is a subclass of Speaker. The rules simulator will implement two key public methods defined by the Speaker, *next* and *get utterance*. From the dialog managers point of view, what the rule simulator does under the hood is completely hidden. The *next* method takes in the opposing speakers speech utterance

The user agenda, which captures what the user simulator will communicate to the dialog agent, is simply the concatenation of the inform and request slot lists stored in the user goal. Functionally the user agenda is a stack of pending dialog acts that user will say over the course of the conversation. All key-value pairs captured in the corresponding inform and request slot lists are mapped to inform and request dialog acts. Over the course of the dialog, the top item at the stack, which gets popped, contains that dialog action for what the user simulator will do next. That action would be passed the the simulator's internal natural language generation module to generate a natural language utterance.

For memory efficiency, we do not actually implement the agenda, as the information already exists in user goal. Instead we pop directly from the inform slots list or defer the action generated by the next method for responses to the dialog agent. We keep track of state of conversation by check how many of the request slots have been filled with real values. The rules simulator runs sequentially through the request slots at the end of each conversation and updates its internal DialogStatus enum object. The logic for how the user simulator responds to incoming speech acts from the dialog agent is handled by the *next* method.

Figure 1.5: Internal logic for the rules simulator.

```
response_router = { "greetings": respond_general,
"inform": respond_to_suggestion,
"random_inform": respond_random_inform,
"request": respond_request,
"confirm": respond_confirm,
"bye": respond_general}
```

The *next* is the driver of the rules simulator. It first will first attempt to parse the agents incoming dialog action and then respond to it using an internal dispatch tree. The internal logic of the *next* method is a simple dispatch dictionary, where incoming dialog acts are mapped to resolver functions. Each response function has the same method signature, which is to take in a DialogAction object and return back a DialogAction that represents the user's response to dialog agent. The dialog agent's dialog action space is limited. The agent will respond with one of the following dialog acts:

- greetings: the agent will greet the user and list it's services
- request: the agent will ask a probing question to elicit the users preferences

- inform: the agent will supply the user with information (usually tied to the user request request slots)
- confirm: the agent will ask the user to confirm if it understood the user's intent
- bye: the agent will end the conversation

The implementation of how the resolvers for the greetings, bye, and confirm responses are straight forward. The code implementation can be found in the appendix. For greetings resolver, if the rules simulator is the first speaker, it will invoke the random inform method one or several inform slots, which will be translated into an inform speech act. Otherwise, it handle the agent dialog action with the dispatch dictionary.

The resolver for responding to the agents request actions is a bit more involved. The simulator will looked at the parsed request slots types the agent is asking about and attempt to find corresponding inform slots in it goal object. For example, if the dialog agent asks "What cuisine do you prefer?", the simulator will look up cuisine in its internal goal and return the answer Chinese (i.e based on the goal in 1.5). In the case where requested slot type is not found in the user's inform preferences, the simulator will respond with either "I don't know" or "I don't care". This null response can be configured in the simulation configuration file, where the response either set to one of those two options or randomly chosen. Additionally, to simulate a realistic user, at configuration time, the researcher can also set the probability with which the simulator will "lie" or "change its mind" about its preferences. In those cases, the simulator will randomly sample the provided slot values in the dialog domain object and return a different slot value.

If rules simulator has exhausted the informing the dialog agent of all its preferences, the simulator will then pop values from its request slots list and ask for a recommendation. If the dialog agent sent an inform action, the inform resolver method would update the request slot with new provided information. So for example, if the dialog agent made the suggestion, "Check out Golden Dynasty", the "UNK" value in 1.5 would be replaced by "Golden Dynasty". If all the "UNK" values in the request slots were filled with real values, the user simulator would update its internal status to complete and issue the bye action.

As described above, the nlu and nlg models were set by the researcher in the simulation configuration file. By inheriting the UserSimulator class, the get utterance and parse utterance methods are also inherited and available to the researcher. Both method essentially call the corresponding methods in nlg and nlu objects.

1.4.2 Dialog Agent

The restaurant agent was developed to illustrate how to incorporate an external dialog agent into the simulation framework. Since we do not have an existing restaurant recommendation agent, we developed a simple rule based agent. The goal of the agent is to capture all the user's preferences and then make a suggestion from its knowledge base of Cambridge restaurants. Like the user simulator, the public facing methods the dialog manager interacts with are *next* and *get utterance*.

For the restaurant agent, we follow a simple rules approach. The agent expects to interact with the following dialog acts: greetings, affirm, negate, request, inform, bye. In situations where the agent encounters an unknown dialog act, it will repeat its last dialog act. At the beginning of conversation round, dialog agent internally resets its goal (1.6).

If restaurant agent goes first, it issues a greetings action. If the agent is not responding to user, it will sequentially pop one item from it request slots and issue a request dialog act. Once the restaurant agent has collected information from the user,

Figure 1.6: Restaurant Agent Goal

```
inform_slots: None
request_slots:
        cuisine: UNK
        area: UNK
        pricerange: UNK
```

it will attempt to make a suggestion from the knowledge base stored in the domain object. This is accomplished by calling the get suggestion method provided by the domain object.

1.5. Movie Tickets

1.5.1 Domain

1.5.2 User Simulator

NLG and NLU

1.5.3 Dialog Agent

NLG and NLU

Chapter 2: Development

This chapter discusses the tools and methodologies employed in the code development of this system.

2.1. Development Language

The framework was written in Python 3.7, which at the time of submission is the latest python version. It is worth noting that the research implementation of the user simulator described in Li et al. (2016) was written in Python 2. Python 3x is the preferred version for

2.2. Unit Testing

2.3. Development Tools

References

Bordes, A. & Weston, J. (2016). Learning end-to-end goal-oriented dialog. *CoRR*, abs/1605.07683.

Chewning, C., Lord, C. J., & Yarvis, M. D. (2015). Automatic question detection in publication classification natural language.

Li, X., Lipton, Z. C., Dhingra, B., Li, L., Gao, J., & Chen, Y. (2016). A user simulator for task-completion dialogues. *CoRR*, abs/1612.05688.

Rey, J. D. (2017). Alexa and google assistant have a problem: People aren't sticking with voice apps they try.

Schatzmann, J., Weilhammer, K., Stuttle, M. N., & Young, S. J. (2006). A survey of statistical user simulation techniques for reinforcement-learning of dialogue management strategies. *Knowledge Eng. Review*, 21, 97–126.

Schatzmann, J. & Young, S. J. (2009). The hidden agenda user simulation model. *IEEE Transactions on Audio, Speech, and Language Processing*, 17, 733–747.