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

Abstract

We aim to develop a dialog simulation framework, Socrates Sim, to support task completion dialog research. The goal of the framework is to provide a set of tools that will simulate the conversation between a user simulator and a dialog agent in order to generate large training data. Traditionally, dialog researchers would use human testers and annotators to generate quality labeled training data. However, this approach is very expensive and time consuming. The hope is that this framework will help researcher augment human generated test data with synthetic data, which would be generated from simulations with a user simulator.

The Socrates Sim framework will provide a set of tools to define dialog domain, set up a user simulator, and run multiple simulations with a provided dialog agent. To demonstrate the flexibility and robustness of the framework to adapt to new domains, we will implement end-to-end simulations for the restaurant recommendation and move booking use case. The framework will be implemented in Python and made available on github at https://github.com/dhairyadalal/socrates.

Contents

Chapter 1: Introduction

Task completion dialog refers to the space of dialog activities, in which an end user engages with an interlocutor in order to complete a task or achieve a tangible goal. For example, imagine a user interacting with a concierge in order to identify and book a restaurant for dinner. In dialog systems, the human interlocutor is replaced by an artificial agent (also referred to as the system or dialog agent) that can intelligently respond and help the user achieve their goal.

Automated dialog systems are not a new concept. The have been used to support call routing (e.g. the press 1 to reach sales) in the context of customer support for banks, credit cards, flight booking, and many other commercial sectors since the 1970s. Central to any dialog system is the dialog policy. The dialog polciy informs the system on what to say and what information to collect based on the state of conversation. Traditionally dialog policies were scripted out, usually following a simple flowchart like structure. This is known as a rules based approach, where rules are written out to capture system behavior under predefined situations/states. Rules based systems are limited, as they require the user to follow a scripted path and provide the system with one piece of information at time. While effective, many user are often frustrated with rule based dialog systems and tend to prefer speaking to a human agent. Much of this frustration is caused by the slow pacing of the dialog scripts, which require the user to specify information piecemeal in specific order.

There has been significant progress in the AI and machine learning space that has led to the development of commercially viable intelligent dialog systems. In particular, the popularity of voice assistants like Amazons Alexa, Apples Siri, and Google's Assistant, have increased the demand for voice interfaces to popular applications and services. There has been a boom in the development of chatbots, third party voice skills for Alexa and Google Home, and other dialog based services. Much of this is due to the proliferation of exciting research which applies deep learning and machine learning to the dialog domain.

However, developing good voice interfaces and dialog systems is still very challenging. For example, a significant percent of the voice skills in the Alexa skill store have poor ratings Rey (2017). According to research by recode.com, 69% of skills in the Alexa skill store have a 0 or 1 reviews suggesting abysmal usage. In addition, there only a 3% chance that user will reuse a voice skill after first use, demonstrating poor retention. Underlying these poor statistics is the fundamental challenge - designing robust and usable dialog systems is very difficult.

Rules based dialogs are not scalable or optimal for more complex task completion. Researchers have moved to leveraging supervised learning (SL) methods to train dialog systems and produce more robust dialog policies. In an SL approach, the dialog policy is trained to imitate observed actions of an expert using an annotated and manually crafted datasets based real human interactions Schatzmann et al. (2006). While this approach produces better policies than a rules based approach, it is limited to quality and scope of the training data.

One of the key challenges in this space is developing quality and diverse training datasets. Producing deep annotated dataset is time consuming, expensive, and the dataset may not comprehensively cover all possible states in a policy space. Supervised learning requires large amounts of clean and annotated training data. This involves having many human testers interacting with a human expert (which proxies for the dialog agent), in order to generate what the ideal conversations would look like. In response to user questions and actions, the human expert would choose the correct policy choice from a defined action space. Over the course of the dialog, the human expert identifies all the right actions that will eventually lead to helping the user accomplish their goal. This process is repeated across many users in order to capture the different goals a user may have and how they communicate. Current data gathering practices rely on crowd-sourcing and or general human trials. While these approaches are effective at generating hight quality data, they are incredibly expensive and limited in how information they can capture.

Reinforcement learning (RL) methods are also gaining popularity. Given a reward function, the agent can optimize a dialog policy through interaction with users and learn what an optimal dialog policy should be. Reinforcement learning approaches are more robust than supervised learning techniques as the agent can explore more of the policy space.

A virtual simulator could alleviate these data needs, by simulating a user in place of using a real human user. The user simulator can be used in the context of supervised learning (SL) or reinforcement learning (RL) to train a dialog system on how to identify optimal dialog policies. In this context, the simulator would generate additional synthetic data to help augment the training data acquired through human testing. The user simulator also provides a useful starting point to train and RL based agent, which can be then further optimized in RL situation with real users Li et al. (2016). Currently, there is no open source or commercially available user simulator framework to support dialog research. The aspiration of this thesis is to develop a solution that can fill that gap.

1.1. Prior Work

The growing popularity of statistical approaches for spoken dialog systems has led to research for more optimal ways to generate training dialog data. Schatzmann and Young introduced the concept of the hidden agenda user simulation model [6], which has been foundational to conceptualizing user simulators. In their 2009 paper, Schatzmann and Young provided a formalized framework to capture user intents in stack-like structure of pending dialog acts. Bordes & Weston (2016) applied deep learning and neural models to dialog systems. They introduced a network-based end-to-end trainable dialog system, which treated dialog system learning as the problem learning to map dialog histories to systems responses and applying encoder-decoder models for training.

Li et al. (2016) developed a framework for a user simulator and released a research proof of concept which was applied to the movie booking domain. The released proof-of-concept, TC-Bot, was written in Python 2.7 and hard-coded to support the movie booking domain. Currently, there is no open source and modern user simulator tool for task-completion dialog research. This thesis aims to adapt their framework to the restaurant domain, write it in Python 3.7.0 and apply good software engineering principles with the aspiration that Socrates Simulator can be used for other domains by the dialog research community.

Finally, Facebook recently released the beta version ParlAI. ParlAI aims to provide a standardized and unified framework for developing dialog models. Theyve released a broad set of tools to support training and development of dialog systems for the following domain areas: question answering, goal oriented dialog, chit-chat dialog, visual qa/dialog and sentence completion. ParlAI offers a simplified set of API calls to common dialog datasets (e.g. SQuAD, bAbI tasks, MCTest, etc) and provides a set of hooks to Amazons Mechanical Turk to test ones dialog model again real human testers. While, ParlAI offer an expansive set of tools and datasets, missing from its framework is a user simulator.

1.2. Project Goals

The goal of this thesis is develop a framework that will allow dialog researchers and to more easily develop user simulators and produce labeled training data for their dialog agents. The intended users of this framework would be academic researchers and data scientists or software engineers in industry. The framework will be modular and support multiple domains. Ideally, it should to be easy for the user to build and test different user simulators for their dialog agent and produce diverse dataset.

Note, this thesis will not explore the efficacy of the data produced by user simulators. Li et al. (2017) and Li et al. (2016) ran experiments in the movie booking domain and found promising results. [ADD finidings here] Additional Bordes & Weston (2016) also found results for the restaurant booking domain. EXPAND THIS SECTION WITH SUMMARY FINDINGS.

Our focus will be on the design and implementation of the Socrates Sim framework. As mentioned above, we will use as inspiration the framework architecture described in Li et al. (2016) and the configuration based approach used by Gardner et al. (2018) for AllenNLP. Significant work was done in designing a more general purpose and modular framework, in order to support a wider set of use cases. The framework will consist of the following components:

- **User Simulator**: An agenda based user modeling component that generates natural language speech utterances to simulate what an actual human would say in the context of task-completion dialog activity.

- **Dialog Agent Interface**: A set of methods to allow a researcher to plug-in their dialog agent and have it interact with the user simulator
- **Dialog Manager**: A coordinator tool that will facilitate the conversation between the user simulator and dialog agent. The simulator will save the simulated conversations in a annotated format.

Chapter 2: Design

2.1. Overview

In this chapter we will describe the architecture of the Socrates Sim framework, highlight key design choices, and describe in detail the component parts.

Our goal is develop a modularized and production grade dialog simulation framework that can be re-targeted for new domains and produce training data and train dialog agents. The overall design for the simulator was inspired by the work done by Li et al. (2016). For the user simulator, we implement the theoretical formulation of the hidden user agenda models described in Schatzmann & Young (2009). We also adopt the configuration first user access pattern used in the AllenNLP library Gardner et al. (2018).

Underlying the simulator is a framework that consists the following components:

- **User Simulator**: An agenda based user modeling component that generates natural language speech utterances to simulate what an actual human would say in the context of task-completion dialog activity.

- **Dialog Agent Interface**: A set of methods to allow a researcher to plug-in their dialog agent and have it interact with the user simulator

- **Dialog Manager**: A coordinator tool that will facilitate the conversation between the user simulator and dialog agent. The simulator will save the simulated conversations in a annotated format.

2.2. Architecture and Design

The general architectural is greatly inspired by the end-to-end neural dialog framework described in Li et al. (2017) (see Figure 2.1). In this design, there are two key components. The first is an agenda based user simulator, which has an internal natural language generation (NLG) model. The right hand side of the framework is the Neural Dialog System, which consists of a language understanding (LU) unit and a dialog management (DM) unit. The user simulator generates speech utterances using its NLG and sends to the LU unit. The LU parses the speech utterance into a semantic frame (implemented as python dictionary) and hands it the dialog management unit, which generates a system action and sends to the user simulator (as a semantic frame). The dialog manager also has a state tracker and policy learner. The policy learner is implemented as a deep Q-network (DQN).

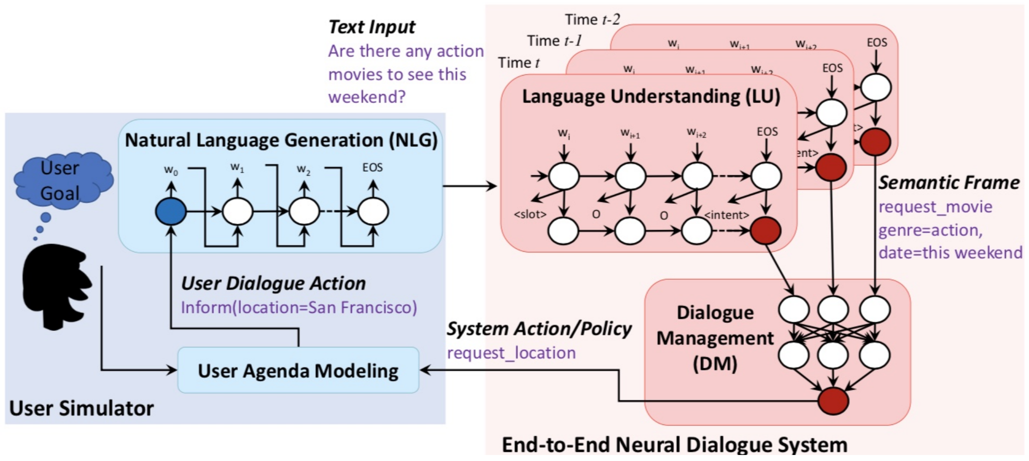

Figure 2.1: The User simulator architecture described by Li et al. (2016)

The neural dialog system uses reinforcement learning to learn optimal policies.

At the end of each simulated conversation, the dialog manager will score itself and update its policy learner. One of the key features to note is that the dialog agent is built in as a learner in the Neural Dialog System. After a set of predefined rounds, the output will be a neural dialog agent serialized as a model file. The simulated dialogs will also stored externally and can be used downstream as further training data.

The primary challenge with this framework is that it is hard to adapt to new domains. While the user simulator is a separate component and can be easily reconfigured, the dialog agent and dialog manager are tightly coupled. The dialog agent must be implemented as neural learner. To adapt this framework to a new domain would require rewriting the dialog manager, language understanding unit, and surrounded scaffolding scripts setting up and running the simulations. In the research code provided by Li et al. (2017), the domain information (movie booking) is directly hard coded into all the components of the framework. Communication between components takes place with python dictionaries, which can arbitrarily defined. This introduces an element of variability that can present challenges downstream for debugging.

The Socrates Sim framework was designed generalize to new domains and provide a consistent experience for dialog simulation and generation of labeled data. Figure 2.2 provides a high level overview of the framework. The framework is pretty basic. There are three key modules, the user simulator, the dialog agent, and the dialog manager. The researcher is responsible for implementing the user simulator and dialog agent. The dialog manager class can configured using an external configuration file, allowing for easier experimentation.

The framework needs to be modular so that it can be quickly adapted to support new dialog domains and experiments. Each major component of the framework

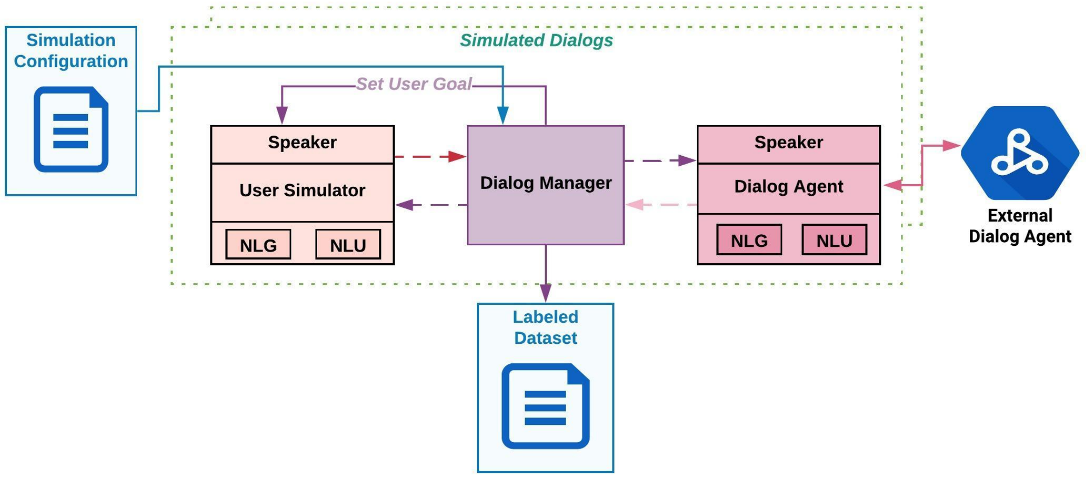

Figure 2.2: Design of Socrates Simulator Framework

is represented as a python abstract base class. Both the user simulator and agent have the same base class (Speaker) and have their own internal nlg and nlu objects. Additionally, I promoted dialog actions, goals, and domain knowledge bases to first class objects, rather than implementing them at the lower dictionary level. As first class objects, I can standardize the communication and management of these pieces and ensure more consistent behavior.

One of the key distinctions, is that the framework is agnostic to how the dialog agent is implemented. By decoupling the agent from the dialog manager, the researcher is free to test out different agents without having to rewrite the entire simulation framework. The dialog agent class provides a simple interface to allow external agents to plug into the simulation framework. This also frees up the dialog manager to provide other useful services like metrics, dialog analysis, and labeled data generation.

We want to provide as much flexibility and freedom to the researcher and limit what is hard-coded. I follow the configuration first approach leveraged in the development of AllenNLP, a deep learning for NLP tool developed at the Allen Institute for Artificial Intelligence. The code is generalized and the user defines particular implementation details in external configuration files.

To support a configuration driven approach, each configurable module will have a use defined yaml or json file. For smaller configuration details, the user will prefer using yaml which is more human readable. The yaml format is simple and has a very low learning curve. It follows a basic key value pair paradigm, where keys have clear semantic meaning and values can be represented in a variety of data structures.

Figure 2.3: Example yaml section.

```
# Dialog Simulation settings
simulation_rounds: 10
max_turns: 8
first_speaker: usersim
simulation_output_path: data/simulated_dialogs/
```

Socrates Sim is command line tool. Once the researcher has setup the user simulator and dialog agent, they can invoke the dialog manager and run simulations with the command line. At the end of the simulation, the dialog manager will output performance metrics for dialog agent and store the generated dialogs with annotations.

The remainder of the chapter will further describe in detail the different components of Socrates Sim.

2.3. User Simulator

The user simulator is responsible for imitating a real user and generating realistic speech utterances. Here we assume the user is an actor that is attempting complete task. For example, the user may want to travel to Japan and is attempting to book a flight there. That user could then interact with a travel agent chatbot in order to get assistance in identifying the appropriate flight and purchasing tickets. In order to model and represent a user, we will utilize the formalization of the hidden user agenda described in Schatzmann & Young (2009).

One of the primary assumptions here is that user has intentionally engaged with the dialog agent in order to complete their task. At the outset of the conversation, the user will have some specific goal in mind (order Indian food, book a flight to Japan, etc). The dialog agent will attempt to learn user's goal by asking the user a set of clarifying questions. Schatzmann and Young introduces the idea of a hidden user agenda as a mechanism to represent the sequence of dialog acts and utterances a user will say in the context of that conversation. At each step of a task-completion dialog, the user is either responding to the dialog agent or initiating a new conversation direction. The user agenda provides an efficient way and formal structure to represent the pending set of dialog acts the user will communicate to the dialog agent.

Socrates Simulator implements the Schatzmann and Young's concept of user agenda and user goal as first class objects. The conversion from the formal representation to code is rather straightforward as there are analogous data structures.

2.3.1 User Goals

The user goal captures explicitly both user's preferences and missing information needs they are trying to fill. For example, take a user who wants to find an Indian restaurant in Central square for dinner. We can decompose this goal into two distinct components. The first is the user's explicit preferences. In this example, their preferred cuisine is Indian. The second component is implicit and unknown to the user. They are looking for a restaurant or more specifically the name and presumably the restaurant's phone number and address. This information is unknown but can be broken down into discrete pieces of information the user will attempt to elicit from the dialog agent as a request for more information.

Formally, Schatzmann and Young defines the user goal G as G = *(C,R)*, where C consists of constraints or the user's explicit preferences and R represents the user's requests. The constraints and requests are explicitly represented as slot-value pairs. 2.4 below shows how one could represent the goal of user looking for a bar.

$$C = \begin{bmatrix} \text{type} = \text{bar} \\ \text{drinks} = \text{beer} \\ \text{area} = \text{central} \end{bmatrix} \quad R = \begin{bmatrix} \text{name} = \\ \text{addr} = \\ \text{phone} = \end{bmatrix}.$$

Figure 2.4: Example User goal. User wants the name, address and phone number of a cheap bar in central.

The concept of a goal abstractly turns out to be useful in also driving the dialog manager's simulations. We abstract the idea of goal and make it available to both the user and dialog agent api as way to track the internal state of each speaker.

2.3.2 User Agenda

Schatzmann & Young (2009) define the user agenda as a *[stack] structure of pending dialogue acts [which] serve as a convenient mechanisms for encoding the dialogue history and users state of mind* . Formally, at any time t, the user is in a state su and takes an action au,which transitions into an intermediate state su. During this intermediate state, the user will receive an action from the system (machine) am, which will transition dialog to next state su and the cycle will reset. The result is a sequence of alternating turns between the user and system (i.e. su -¿ au -¿ su -¿ am -¿ su -¿), which represents the conversation state over time t.

The user agenda A is stack-like structure which contains all pending user actions. User actions are actualized through popping the stack and the agenda is updated by pushing back onto the stack. A user act is a representation of the users intent, which will eventually be translated into a speech utterance. The stack may also contain other actions that will affect the user when popped. For example, the system can communicate a restaurant suggestion, which would fill the one of the request slots for restaurant name.

At the start of the dialog a new goal is randomly generated from the provided dialog domain. An accompanying agenda is then generated to represent the potential sequential of events.

Below is an example of the a sample user agenda that Schatzmann and Young provide in the context of a user asking the dialog system for a bar recommendation [6]. The states of the conversation are indexed by time t. Note, Schatzmann and Young use constraints C, which would be the equivalent of inform slots in our representation. In the first turn, the user simulator generates a set of constraints (bar serving beer in central) and goals (name, address, and phone for a bar that meets the constraints in C0). This set of inform and request slots are translated into a user action stored in A0. When the system initiates the conversation, the user simulator pops two inform actions which translate into the user utterance Im looking for a nice bar serving beer. When the system at t=1, responds Ok, a wine bar. What price range? the agenda updated to include a new inform intent (inform(prange=cheap)). Also added is a negate action, as the user asked beer and not wine.

Over the course of the conversation the agenda is updated, as are the request slots. The conversation ends at t=5, when bye() is popped and the agenda stack is empty. The conversation will then be evaluated based on how well the request slots were filled.

2.3.3 Dialog Manager

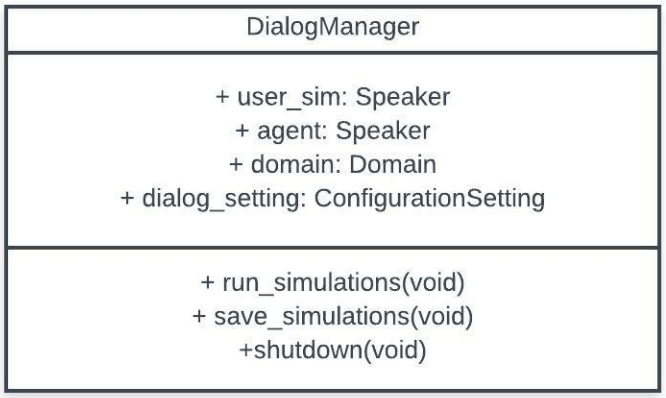

Figure 2.5: Class Definition of Dialog Manager.

The dialog manager has many responsibilities and is the primary engine of the simulation framework. Its responsibilities include loading the dialog domain, setting up user simulator and dialog agent, running the simulations, and performing post simulation metrics. The dialog manager does not need to modified by the researcher and is setup to implement the researcher's preferences via a configuration file. The configuration file is translated into a configuration object by the command line tool and sent to the dialog manager. In the configuration file, the user specifies the following:

- user simulator class
- dialog agent class
- dialog domain representation
- domain knowledge base class
- simulation settings

The dialog manger uses python dynamic loading capabilities to import the end user custom classes for the user simulator, dialog agent, and domain knowledge base. Once these are loaded into memory, the simulator runs the simulations. Figure 2.6 shows at a high level what the dialog model looks like.

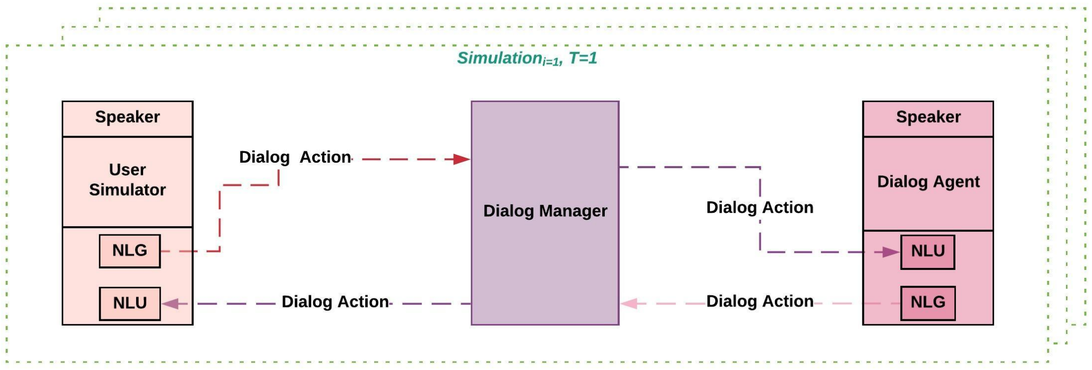

Figure 2.6: Dialog Model Overview

The dialog manager follow a basic pattern when running the simulation. It first generates a user goal, which will drive the user's hidden agenda and behavior. Next it resets the user simulator with the newly generated goal and calls the dialog agent to reset itself. Once both speakers are to ready to interact, the simulation is initiated. The logic for conversation is quite straightforward. The speaker class has next method, which takes in dialog action and returns a response dialog action. The logic for the speakers to communicate with each other can be expressed int two line (see 2.7. The dialog action object standardizes how the speakers talk to each other and detail can found below.

Finally, in order to simulate a real user, the researcher can set configure the user simulator to be corrupted in different ways. The user can change their minds and have a new preference generated from the dialog domain. To simulate faulty technical issues, the user may randomly exit the conversation. And finally the user's goals can

Figure 2.7: Psuedo code for conversation round.

```
# Assuming user speaks first
# 1. User Simulator takes turn and speaks
user_action = user_simulator.next(previous_agent_action, turn_number)
# 2. Agent takes turn and responds to user
agent_action = agent.next(previous_user_action, turn_number)
```

be corrupted resulting in an indiscernible signal from the user as to whether their goals were met or not. All these corruption are specified by the researcher with a probability value, which represents the likelihood the user simulator will act abhorrently. This behavior will help generate more diverse data by providing negative examples.

Once all the simulations are run, dialog simulator will run performance metrics to evaluate the efficacy of the dialog agent, and provide general statics about coverage of the generated data. The simulator evaluates the dialog agent in two ways. First it determines if the user was able to fill all the request slots in their hidden goal, which signals if the dialog outcome was successful or not. The researcher will be informed the average success rate of the dialog agent and can also see the round level evaluations in the generated dataset. Additionally, the simulator will produce quality score per round, which qualitatively evaluates the suggestions the agent made. The quality score captures how many of the user's preferences were satisfied by the agent's suggestion. Finally, the diaog manager will present the metrics describing the variance and distribution of the generated dataset. This way the researcher can see the coverage domain (which preference were used) and general aggregate level statistics about the dataset.

2.3.4 Dialog Action

The fundamental unit of transaction for the simulator is the dialog action. It is an explicit semantic representation of the speech utterance that is machine readable. For example, I could represent the speech utterance, "I'm looking for a Thai restaurant", in the following way: original utterance: *I'm looking for a Thai restaurant* dialog act: *request* constraints: *cuisine=Thai* The DialogAction object standardizes and encodes information about a speech act that can be understood by both the user simulator and dialog agent. The dialog action consist of three key properties: the dialog act, a set of explicit dialog parameters or constraints, and corresponding unparsed speech utterance. I will next describe these properties in further detail.

The unparsed speech utterance is just the natural language utterance that was spoken by the feature. One of the goals of the user simulator is to generate speech utterances based off its internal user agenda. The user simulator will use the dialog act and dialog parameters popped from the user agenda to generate a new utterance using it natural language generation model. If a speaker is hearing the utterance, then the utterance will need to be parsed and broken into a dialog action and set of dialog parameters in ordered to be processed.

The dialog act property is used to capture the intent of the speech act. Common dialog acts include: inform, request, confirm, negate, and affirm. The dialog act is necessary to provide context for the dialog parameters for both the natural language generation and understanding use cases. For example, a set of dialog parameters like "cuisine=thai, area=north" can be interpreted differently in the inform vs request context. In the request context, the speech utterance could be "I'd like to find a thai restaurant in north part of town". In contrast, those same dialog parameters could be part of a suggestion in the inform context. E.g. "There is a great Thai restaurant in the north part of town". Given the variability of dialog acts and intents, the space of possible dialog acts is defined by the researcher in a configuration file. While this limits the possible interpretations of the speech act, a reduced dialog act space is vital to building effective task completion dialog agents.

As we saw above, the dialog parameters encode entities in the speech utterance as key-value pairs and are used the communicate or elicit the user's preferences. The key captures the entity type or constraint type (e.g. cuisine, area, address, etc), while the value is used to indicate the specific constraint or entity (e.g. Thai, north, 115 Way Street, etc). The parameter property is strictly typed as a python dictionary and therefore all keys must be associated with a value. In the context of request speech acts, the null value is used to indicate the speaker wants to elicit more information the provided constraint type. For example, *dialog_act=request, params={address: NULL}*, would be interpreted as a request for the address (e.g."What is the address?").

Initially, I used python dictionaries and sets to represent the dialog parameters. However, this was as an sub-optimal several reasons. First it introduced variability and uncertainty. To capture request parameters I used python sets, since the value was null and we justed needed to pass along the constraints types. For all dialog actions, I used explicit dictionaries as real values were passed along with the constrain types. However, downstream this required logic to check the class instance of the parameter variable being passed in. Given the dynamical nature of the dialogs being generated, the simulator was rather unstable and would randomly crash due when a method expecting a dictionary, received a set. By enforcing a the python dictionary type and setting null values, I was able to greatly improve stability and make debugging easier by standardizing the input into functions that consumed the DialogAction object.

2.3.5 Goals

The Goal class represents the objective of the speaker at the outset of the conversation. For the user simulator, the goal defines the hidden set of preferences and information needs the user has. The goal object has two properties, the inform slots and request slots. Both the inform and request slots are typed as python dictionaries.The inform slots capture the user's preferences that they want to communicate to the dialog agent. A well developed dialog agent should be able to elicit those preferences efficiently and ideally without needing to ask the user multiple times. The request slots capture the information user needs in order to complete their objective and task. At the start of the conversation, all the request slots values are set to null values. Over the course of the dialog, as the dialog agent responds the user simulator, the request slots may be updated with real values. A dialog is considered successful if all the request slots for the user simulator have been replaced by real values. The user simulator monitors the state of the request slots in it Goal object. If all the request slots are filled, the user simulator update its internal dialog status to the complete state and signal to the dialog manager to end the conversation.

In the first iteration, only the user simulator had a Goal object. But it made sense to include the Goal object for the dialog agent as well. The rationale behind this was two-fold. First it provided a useful mechanism to track the state of the dialog agent as well. Like the user, the dialog agent has its own set of goal, i.e. to elicit the information it needs to provide a meaningful suggestion or provide the specific service the user desires. The request slots in the dialog agent are complimentary to the user's inform slots. Ideally, the dialog agent will effectively elicit the user inform slot through a series of request speech acts and then execute it service. This leads us to the second value for the dialog agent, the Goal object provided a useful mechanism for training (especially in the reinforcement learning context). In failure cases, it signals to the researcher the information the dialog agent was ineffective at capturing. For reinforcement learning, a loss function can be developed that minimize the open request slots in the agent's Goal at the end of each conversation when the agent renders its service.

2.3.6 Dialog Status

The DialogStatus is a python enumerative object that encodes the internal state of the dialog for each speaker. Each speaker is responsible for setting its own dialog status. The valid states are not started, no outcome yet and finished. The dialog manager will probe each speaker for it's dialog state. It the dialog manager learns that the state for any speaker is set to finished, the conversation will be ended. The user simulator sets its dialog status to finished when all the requests slots in it Goal object are filled. In contrast the dialog agent may only set its state to finished after the user leaves the conversation. This way the agent does not prematurely exit the conversation before the user can complete their task.

2.3.7 Speaker

The speaker represents an actor that has the ability to speak and comprehend speech utterances. In our framework, the both user simulator and the dialog agent are represented by the same base speaker class. Both actors conceptually are identical in terms of functional behavior, in that they both listen and comprehend speech utterances and in turn respond by speaking. As such, both the user simulator and dialog agent can be represented in the same way to the dialog manager. In fact, the entire conversation round can be expressed in two lines (see Figure 2.7).

The speaker class has four basic functions (*next, reset, get utterance, and parse utterance*) and three properties (*nlg model, nlu model, and dialog status*). When the speaker is initialized, the natural language object and the natural language understanding object are passed to the constructor. We abstract away the implementation of how the speaker speaks and parses speech in order maintain separation of concerns and also empower the researcher to be able to experiment with multiple techniques.

The *next* method is the primary driver how the speaker behaves. For the dialog agent class, the next method functions as an API to the simulator. It is assumed that the dialog agent will live and operate external the simulator. The researcher can define how the dialog agent will interact with the user simulator here. For the user simulator, the bulk of the logic will reside here. The *get utterance* and *parse utterances* methods are simply wrappers for the speaker's nlg and nlg objects. The primary parameter for next is the previous dialog action.

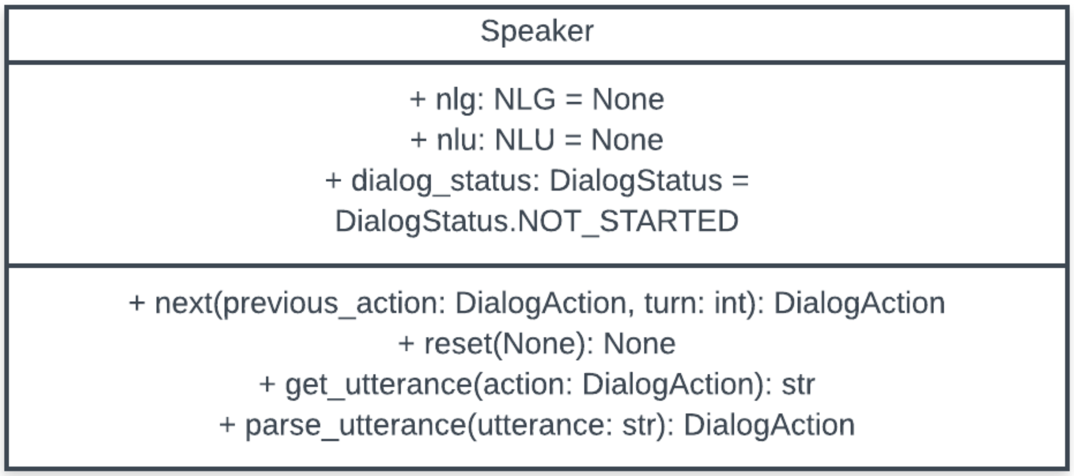
Speaker
+ nlg: NLG = None
+ nlu: NLU = None
+ dialog_status: DialogStatus = DialogStatus.NOT_STARTED
+ next(previous_action: DialogAction, turn: int): DialogAction
+ reset(None): None
+ get_utterance(action: DialogAction): str
+ parse_utterance(utterance: str): DialogAction

Figure 2.8: Definition of the Speaker class and methods.

2.3.8 Dialog Domain and Domain Knowledge Base

The domain class standardizes the collection and storage of information related to domain of the services provided by the dialog agent. The domain class is initialized by a configuration file defined by the researcher and provides a set of APIs to access dialog domain elements. The domain class consists of the following key properties: dialog acts, request slots, inform slots and inform slot values, valid user goals templates, sample starting goals list, and a domain knowledge base object. Additionally the following key API methods are made available: sample inform slots and inform slot values, sample request slots, get valid user goals, and get suggestions (from domain knowledge base) and validate suggestions.

The primary consumer of the domain class is the dialog manager, which uses the domain information to generate new user goals or sample user goals from a pre-existing list of starting goals. If the dialog manager is generating novel user goals, it will use the valid user goals template to create a new goal and sample the inform slots to generate the user's preferences for that goal.

The domain knowledge base (KB) is modeled as a distinct class and serves as an interface to a knowledge base / database the dialog agent would have access to. The primary purpose of the domain KB is store all the suggestions that a dialog agent would make based on the various preferences of the user. The domain KB provides an interface with the following three methods; get suggestions, validate suggestions, and get item. The researcher is free to use whichever back-end and implementation to resolve those three API calls. For this thesis, I chose to use a Pandas dataframe which is an efficient in memory data table that can queried and manipulated. In the implementation section, I'll further describe my approach for resolving user queries and providing simple suggestions with greedy preference matching.

2.3.9 Natural Language Understanding (NLU) and Natural Language Generation (NLG)

The NLU interface provides a common API for the parsing of speech utterances for each speaker. It is an interface with one API method: parse utterance. Parse utterance will take in a natural language speech utterance and return a DialogAction object. The researcher has flexibility in setting up the NLU back-end. The NLG module is a complimentary interface, and also contains a single API method: get utterance. Get utterance takes in a DialogAction object and will generate a new speech utterance. In implementation section, we will further details the simple rules based approach and neural machine translation implementations for both the nlg and nlu back-ends.

Upon initialization, the speaker class will set an internal nlg and nlu object. The speaker's *get_utterance* and *parse_utterance* will directly call the correspond methods in the nlg and nlu object. This way the dialog manager does not need to know the internals of each speaker's nlg and nlu objects.

NLU and NLOG is an open problem space and there are no single universal solutions. In abstracting the NLU and NLG interfaces, the researcher has more flexibility in training and experimenting with their dialog agent. Since the user simulator will always return the machine readable DialogAction with the generated speech utterance, the researcher has the ability to train the NLU for their dialog agent to support more robust NLU use cases.

Chapter 3: Implementation

3.1. 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.

3.2. Restaurant Recommendations

3.2.1 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.

3.2.2 Domain

Figure 3.1: 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.

3.2.3 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.

3.2.4 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 3.2). Given the wide range of potential string matches for inform, we use that as the default dialog act.

Figure 3.2: 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.

3.2.5 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 3.3). 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 3.3: 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.

3.2.6 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.

3.2.7 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 3.4: 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 3.5: 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 3.5). 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 3.5 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.

3.2.8 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 (3.6).

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 3.6: 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.

3.3. Movie Tickets

3.3.1 Overview

The goal of the Movie Booking agent is help the end user purchase movie tickets. A similar domain agent was developed for the Li et al. (2016) paper. Unfortunately, I was unable to use their agent and domain knowledge base in the context of the Socrates User Simulator. Much of their data was crowd sourced from Amazon Mechanical Turk and their use cases were a bit convoluted. There was a confusing overlap between the dialog action space for the user and agent which resulted in the user simulator making nonsensical and unrealistic utterances and dialog acts. There was a hight ratio of nose and data quality issues which would have made it difficult to replicate.

Drawing inspiration from their use case, the agent we developed is a bit more refined and modeled after something you would see on a movie booking site like Fandango. The agent will help identify movies for the user based on user preferences and also collect the user's payment details if the user choses to book a ticket. As the goal for this implementation is mainly to demonstrate the end to end dialog simulation framework and the user simulator, we did not implement an actual reservation database, a payment processing system, and a large move show times catalog. There are stubs for where the dialog agent would theoretically consult external resources in the aiding the user. We focus below on the the implementation details for the user simulator and domain modeling exercise.

3.3.2 Domain

3.3.3 Domain Knowledge Base

3.3.4 Natural Language Understanding

3.3.5 Natural Language Generation

3.3.6 User Simulator

3.3.7 Dialog Agent

Chapter 4: Development

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

4.1. 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 production python products. Most major data science and machine learning research python libraries no longer support Python 2. Python 3 provides many useful features and performance upgrade that make writing and deploying python projects more efficient and effective. In addition to updated syntax that allows for more expressive coding, the standard library was extended to support new data types (ordered dictionaries, enumerated types, data classes). Additionally, Python 3.5 introduced type annotation, which allows for the writing of cleaner, better documented, and unambiguous code. We describe type annotations further below.

The framework was written to adhere to PEP8 standard and all method signatures have type annotation. The hope is that good software documentation and standard coding styles will allow future contributers to easily debug, modify, and build new modules for the framework.

4.2. Development Tools

The framework was developed using the PyCharm IDE. Pycharm was selected for its ease of use for rapid development, python debugging tools, and integration with Github. Various python libraries were used in the development of the framework. The following third party python packages had significant impact on the development of the framework:

- yaml: The yaml library provides a set of function to read and write yaml files. It was primarily for the loading and saving of the various configuration files.
- json: The json library provides a set of functions to read and write json files. It was primarily used for the loading and saving of various configuration files and the serialization of the simulated dialogs.
- spacy and nltk: Spacy and NLTK are natural language processing libraries that provides standard NLP tools like part of speech tagging, dependency parsing and, named entity recognition. They were used primarily for developing the base natural language understanding and natural language generation models.
- pandas: The pandas library provides robust, efficient, and flexible data structures that can be used for data science. The pandas dataframe was used to represent the in memory knowledge base and execute various logical queries.
- keras: The keras library is high level front for develop neural network models for deep learning. Keras compiles the neural architectures using either a Tensorflow, Theano, or Microsoft CTNK back-end. Keras was used to develop and deploy the neural machine translation models used for the natural language understanding and natural language generation modules.

The code base is stored on Github.com and uses git for version control and bug tracking. Github is an online, cloud based, software platform used for sharing and hosting code bases. Github provides various collaboration and version tracking. It is one of the most popular platforms for code sharing and collaboration.

The Sphinx tool was used for documentation generation. Sphinx automatically extracts method docstrings and generates code documentation as rendered markdown files. For hosting, I used the open source and free Read the Docs service, which hosts code documentation websites.

4.3. Unit Testing

I used the Python unit testing library (unittest) to create standard performance and integration tests. The test used for benchmarking the Socrates Simulator framework performance and also testing code stability. Additionally, a simple set of methods are available to the end user to write custom unit tests to validate their user simulators and dialog domain classes.

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.

Gardner, M., Grus, J., Neumann, M., Tafjord, O., Dasigi, P., Liu, N. F., Peters, M. E., Schmitz, M., & Zettlemoyer, L. (2018). Allennlp: A deep semantic natural language processing platform. *CoRR*, abs/1803.07640.

Li, X., Chen, Y., Li, L., & Gao, J. (2017). End-to-end task-completion neural dialogue systems. *CoRR*, abs/1703.01008.

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.