Socrates Sim: A Dialog Simulation Framework 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

In this thesis, we propose an end-to-end dialog simulation framework, called Socrates Sim, to support task-completion dialog research. The goal of the framework is to provide a set of tools that will simulate generate between a user simulator and a dialog agent in order to evaluate the performance of the dialog agent and generate annotated data. Specifically, the Socrates Sim framework allows the researcher to define the custom dialog domains, build user simulators, and run multiple simulations with a provided dialog agent. To demonstrate the flexibility of the framework to generalize to new domains, implement end-to-end simulations in the following domains: restaurant recommendations and movie bookings. The framework is implemented in Python 3.6 and made available on GitHub (https://github.com/dhairyadalal/socrates).

Acknowledgments

This project was able to be completed due to the support and guidance of many wonderful individuals in my life. I would like to first thank Professor Stuart Shieber for serving as my thesis adviser. I would not have able to complete this ambitious project without your support and guidance. My research advisor, Dr. Sylvain Jaume, helped edit my thesis and provided guidance in navigating the degree program.

Additionally, I would like to thank my supervisors Byron Galbraith, Paula Long, as well as the Talla organization for their patience and support as I undertook this project. I was a full-time employee for Talla while working writing and researching this thesis. Balancing the demands of work while researching was challenging. I am incredibly fortunate to work for an organization that provided me with the flexibility to take time off and prioritize the completion of my degree and this thesis. Without that support, this project could not have been completed.

The idea for this project initially arose from my time at the Allen Institute for Artificial Intelligence (AI2). Vu Ha, Amos Ng and I were part of a team that was exploring novel research areas in the dialog space. This project was initially slated to be AI2's fifth research track but was quickly decommissioned after AI2 chose to prioritize other research. I greatly appreciate the blessing from Vu Ha and AI2 to be able to continue the project independently.

This degree has taken quite a long time to complete. I started the ALM program in 2013 while working full time at Harvard. Over the course of this program,

I've changed my job four times, moved across the country to Seattle and back to Cambridge. While it has been exhausting and demanding, I have had profound emotional support from my family and friends. I am grateful to mother (Shobha), father (Bharat), and sister (Dhara) for continuing to nag, push, and of course provide generous amounts of love and encouragement. I am also thankful for my dear friend and partner Camille Shaw who sent pictures of cute animals daily to motivate me and provided so much emotional support. And finally, thank you Thakurji for blessing me and providing opportunities for success.

Contents

Table of Contents **v**

List of Figures **vii**

1 Introduction **1**

1.1 Project Goals 1

1.2 Background 3

1.3 Prior Work 6

1.4 Thesis Overview 7

2 Design **9**

2.1 Overview 9

2.2 Architecture and Design 10

2.2.1 Configuration First Design 13

2.3 Dialog Domain and Domain Knowledge Base 14

2.4 Speaker Abstract Base Class 16

2.4.1 User Simulator 18

2.4.2 Dialog Agent 21

2.5 Key Dialog Components 21

2.5.1 Dialog Action 21

2.5.2 Dialog Goal 24

2.5.3 Dialog Status 26
2.5.4 NLG and NLU Abstract Base Class 27
2.6 Dialog Manager 28

3 Implementing Socrates Sim **31**
3.1 Overview 31
3.2 General Process to Deploy Socrates Sim 32
3.2.1 Defining the Dialog Domain 32
3.2.2 User Simulator and Dialog Agent Interface 34
3.2.3 Simulation Configuration 37
3.3 Simulating Dialogs with Socrates Sim 39

4 Implementation Use Case: Restaurant Recommendation **42**
4.1 Overview 42
4.2 Defining the Dialog Domain 43
4.3 User Simulator Implementation 44
4.3.1 Simulator Logic 46
4.3.2 Natural Language Understanding 48
4.3.3 Natural Language Generation 51
4.3.4 Neural Model for NLG 53
4.4 Dialog Agent Implementation 54
4.5 Simulating Dialogs with Socrates Sim 55

5 Implementation Use Case: Movie Booking **57**
5.1 Overview 57
5.2 Defining Dialog Domain 58
5.2.1 Domain Knowledge Base 59

5.3 User Simulator Domain 60
5.3.1 NLG and NLU 61
5.4 Dialog Agent 63
5.5 Generating Sample Dialogs with Socrates Sim 63

6 Development **66**
6.1 Development Language 66
6.2 Development Tools 67

7 Results **69**
7.1 Experiment Overview and Setup 69
7.2 Runtime Performance Experiment and Results 70
7.3 Memory Consumption Experiment and Results 74

8 Conclusion **79**
8.1 Known Limitations and Suggestions for Future Work 79
8.2 Lessons Learned 80
8.3 Summary 81

References **84**

List of Figures

2.1 The user simulator architecture described by Li et al. (2016) 10
2.2 Design of the Socrates Sim framework. 12
2.3 Example yaml section. 14
2.4 Dialog domain class. 14
2.5 Domain knowledge base abstract base class and implementation. . . 16
2.6 Speaker abstract base class and implementations. 17
2.7 Example of user agenda formulation from Schatzmann & Young (2009) 20
2.8 Dialog action class. 21
2.9 Example user goal. User wants the name, address and phone number of a cheap bar in central. Schatzmann & Young (2009) 24
2.10 Dialog goal class. 25
2.11 NLU and NLG abstract base classes. 27
2.12 Dialog manager class. 28
2.13 Dialog model overview. 29
2.14 Pseudo code for conversation round. 30

3.1 Sample domain configuration file. 33
3.2 Extending Speaker class for user simulator and dialog agent. 35
3.3 Example simulation configuration file. 37
3.4 Supported simulation settings. 38

3.5 Command line parameters to run 300 simulations with TC-Bot. . . . 39
3.6 Sample run of the Socrates Sim framework. 40
3.7 Sample output after running one simulation round. 40

4.1 Restaurant dialog domain configuration file. 44
4.2 Example user goal. User is seeking the name and phone number of a cheap Chinese restaurant 45
4.3 Internal dispatch tree for the rules-simulator. 47
4.4 Intent classifier algorithm 49
4.5 Neural machine translation model architecture - Brownlee (2017) . . 51
4.6 NLU process using OpenNMT. 52
4.7 NLG template for user simulator. 53
4.8 Example goal for restaurant agent. 54
4.9 Simulation configuration file. 56
4.10 Sample dialog generated by Socrates Sim. 56

5.1 Movie booking dialog domain configuration file. 59
5.2 Example user goal for movie booking domain. 60
5.3 Internal dispatch tree for the movie booking user simulator. 61
5.4 Excerpt from nlg template for movie booking domain 62
5.5 Simulation configuration file. 64
5.6 Sample dialog generated by Socrates Sim. 65

7.1 Runtime of Socrates Sim vs TC-Bot with rule based user simulator. . 71
7.2 Runtime of Socrates Sim vs TC-Bot with model-based user simulator. 73
7.3 Runtime performance across domains. 74
7.4 Average cost to run a single simulation. 74
7.5 Memory usage of Socrates Sim vs TC-Bot with rule based user simulator. 75

7.6 Memory usage of Socrates Sim vs TC-Bot with model-based user simulator. 76
7.7 Memory usage across domains for Socrates Sim. 77

Chapter 1: Introduction

1.1. Project Goals

In this thesis, we present an end-to-end dialog simulation framework called Socrates Sim, which supports task-completion dialog research. The goal of our framework is to provide a set of tools that generate conversations between a user simulator and a dialog agent in order to evaluate the performance of the dialog agent and generate annotated data. Specifically, Socrates Sim allows researchers to define the custom dialog domains, build user simulators, and run multiple simulations with a provided dialog agent with a single unified framework.

Task-completion dialog agents have grown in popularity in the past few years. With the advent of home assistant services like Amazon's Alexa and advances in dialog research, there has been a rise in the development of dialog agents that provide various user services like flight bookings and restaurant recommendations. One of the key challenges in developing these agents is the effective training and evaluation of them. There are only a handful of tools available to support dialog research. Missing in this space is a framework that facilitates end-to-end simulation between a user simulator and dialog agent. There are several benefits of such a framework.

One of the main challenges is that it is very expensive to train dialog agents and produce quality training data. It is a very labor-intensive and human-centric process. User simulators can alleviate the need for human testers by simulating human

behavior. A framework that simulates the conversation between a user simulator and dialog agent can be used to generate large amounts of ancillary data to augment manually collected training data for model-based dialog agents. Additionally, our framework provides a standardized way to test the robustness and efficacy of a dialog agent.

The intended users of this framework are academic researchers and data scientists or software engineers in industry. The framework is modular and retargetable to support multiple domains. This makes it easy for the researcher to build and test different user simulators for their dialog agent and produce diverse datasets.

In this thesis, our focus is on the design and implementation of the Socrates Sim framework. We 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. The primary limitation of the Li et al. (2017) framework is that it tightly couples the user simulator with the development of a dialog agent. As a result, adapting the framework to a new domain requires reimplementing the entire architecture.

Our contribution is the design and implementation of a framework that generalizes to new domains in a scalable manner. We abstract and modularize key components in order to develop a framework that is domain independent. The researcher can plug in different dialog agents and user simulators in order to run simulations, evaluate the agent, and generate annotated training data. Additionally, we make the framework easier to use by utilizing a configuration-based approach, in which the user specifies the simulation parameters in an external configuration file. The user can set up multiple experiments, change domains, dialog agents, and user simulators without having to write new code or manually provide component locations as a long list of command-line parameters. Finally, our framework scales efficiently as the number of

simulations it generate increases.

The framework consists of the following components:

- **Dialog Domain**: A set of classes to represent the dialog domain and knowledge base. The dialog domain consists of all possible dialog acts, valid inform/request slots and values, and other domain specific information.

- **Speaker Interface**: A unified interface that defines a standardized protocol for the external user simulators and dialog agent to communicate with the framework.

- **Dialog Manager**: A coordinator tool that facilitates the conversation between the user simulator and dialog agent. Additionally, the dialog manager tracks simulation histories, evaluates simulated dialogs, and serializes annotated simulations to disk.

In the next section, we provide more background on the evolution of task-completion dialog research and the motivation for developing Socrates Sim.

1.2. Background

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. They have been used to support call routing (e.g.: *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 policy informs the system on what to say and what information to collect based on the state of the conversation. Traditionally dialog policies were scripted, usually following a simple flowchart-like structure. This is known as a rules-based approach, where rules are written out to capture system behavior in predefined situations/states. Rules-based systems are limited in that they require the user to follow a scripted path and provide the system with one piece of information at a time. While effective, users are often frustrated with rules-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 individually and in a 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 Amazon's Alexa, Apple's 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 proportion of the voice skills in the Alexa skills store have poor ratings. According to Rey (2017), 69% of skills in the Alexa skill store have a 0 or 1 star reviews, suggesting abysmal usage. In addition, there is only a 3% chance that a user will reuse a voice skill after the first use, demonstrating poor retention. Underlying these poor statistics is the fundamental challenge - designing robust and usable dialog systems is very difficult.

Rules-based dialog systems are neither scalable nor optimal for more complex task completion. Researchers have moved to leverage 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 the observed actions of an expert using annotated and manually crafted datasets based on real human interactions Schatzmann et al. (2006). While this approach produces better policies than a rules-based approach, it is limited by the quality and scope of the training data.

One of the key challenges in this space is developing quality and diverse training data. Producing deep annotated data is time-consuming and expensive. 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 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 general human trials. While these approaches are effective at generating high-quality data, they are expensive and limited in how they can capture information.

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 policies. The simulator would generate additional synthetic data to help augment the training data acquired through human testing. The user simulator would also provide a useful starting point to train RL based agents, which then can be further optimized in RL situation with real users Li et al. (2016). Currently there is no open source or commercially available framework to support end-to-end simulations with user simulators. The aspiration of this thesis is to develop a solution that can fill that gap.

1.3. Prior Work

The growing popularity of statistical approaches for spoken dialog systems has led to research for more optimal ways to generate training data. Schatzmann & Young (2009) introduced the concept of the hidden agenda user simulation model, which has been foundational to conceptualizing user simulators. Schatzmann and Young provide a formalized framework to capture user intents in a stack-like structure of pending dialog acts. Bordes & Weston (2016) applied deep learning and neural models to dialog systems. They introduced the concept of a neural network-based end-to-end trainable dialog system. Their approach treats dialog system learning as problem learning and attempts to map dialog histories to system responses by 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. They released a proof-of-concept framework, TC-Bot. The framework 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. Our framework is written in Python 3.6.0 and applies good software engineering principles. It is our aspiration that Socrates Sim can be used for other domains by the dialog research community.

Finally, Facebook recently released the beta version of ParlAI. ParlAI aims to provide a standardized and unified framework for developing dialog models. They have 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 Amazon's Mechanical Turk to test one's dialog model against real human testers. While ParlAI offers an expansive set of tools and datasets, missing from its framework is support for incorporating user simulators.

1.4. Thesis Overview

In this thesis, we describe the architecture of Socrates Sim, provide implementation details for two different domains, and demonstrate our framework usability through a set of runtime performance evaluations. In chapter 2, we describe the architecture of the Socrates Sim framework. In particular, we focus on modeling conversations, how to standardize communication between external dialog agents and user simulators, and the underlying dialog management framework.

In chapter 3, we describe the general implementation process for setting up and using Socrates Sim. In chapters 4 and 5, we specifically explore how Socrates Sim was adapted to support the restaurant and movie domains. We detail the development of user simulators, dialog agents, and the underlying dialog components for both domains. Additionally, we highlight the use of configuration files that allow for rapid

experimentation and easy swapping of modular components without having to modify existing code.

In chapter 6, we describe the development of Socrates Sim and cover the tools, language, and other programming-specific choices made in developing Socrates Sim. In chapter 7, we provide evidence that Socrates Sim is usable and efficient. Multiple performance tests were run to evaluate the runtime efficiency and memory consumption of Socrates Sim as it ran an increasing number of simulations. We used TC-Bot (Li et al. (2017)) as a benchmark to evaluate performance. We show how Socrates Sim scales efficiently and maintains shallow linear growth for its runtime execution.

Finally, in chapter 8, we conclude this thesis. We describe known limitations, talk about the lessons learned, and provide suggestions for future work.

Chapter 2: Design

2.1. Overview

In this chapter, we describe the architecture of the Socrates Sim framework, highlight key design choices, and describe in detail the component parts. Our goal is to develop a modularized and production-grade dialog simulation framework that can be re-targeted for new domains, produce training data and evaluate 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 by Schatzmann & Young (2009). We also adopt the configuration first user access pattern used in the AllenNLP library Gardner et al. (2018).

At a high level, Socrates Sim consists of the following components:

- **Dialog Domain**: A set of classes to represent the dialog domain and knowledge base. The dialog domain consists of all possible dialog acts, valid inform/request slots and values, and other domain-specific information.
- **Speaker Interface**: A unified interface that defines a standardized protocol for the external user simulators and dialog agent to communicate with the framework.
- **Dialog Manager**: A coordinator tool that facilitates the conversation between

the user simulator and dialog agent. Additionally, the dialog manager tracks simulation histories, evaluates simulated dialogs, and serializes annotated dialogs to disk.

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).

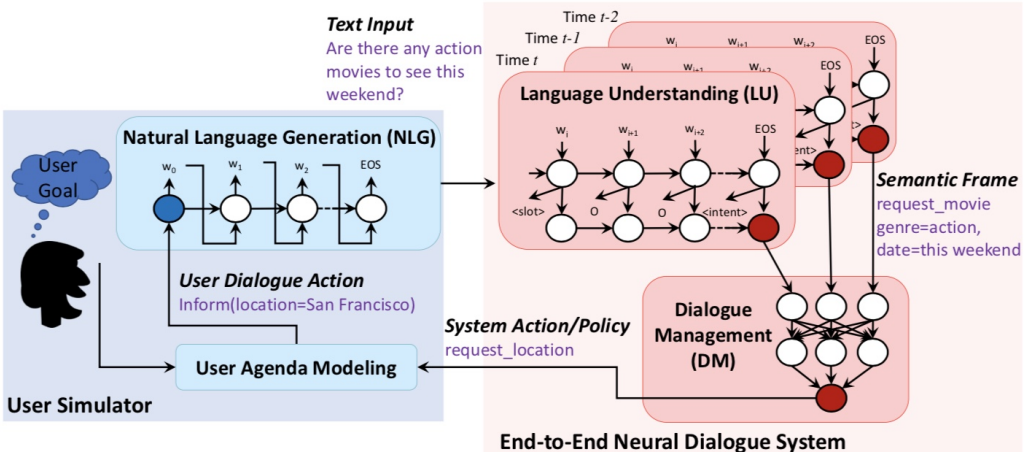

Figure 2.1: The user simulator architecture described by Li et al. (2016)

The neural dialog system uses reinforcement learning to learn optimal policies for the dialog agent. At the end of each simulated conversation, the dialog manager will score itself and update its policy learner. One of the key limitations of this approach 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 be stored externally and can be used downstream as further training data.

The primary challenge with this framework is that it is difficult to adapt it 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 a neural learner. To adapt this framework to a new domain would require rewriting the dialog manager, language understanding unit, and the surrounding scaffolding scripts which set up and run the simulations. In the research code provided by Li et al. (2017), the domain information (movie booking) is directly hardcoded into all the aspects of the framework. Communication between components takes place with Python dictionaries, which can be defined arbitrarily. This introduces an element of variability that can present challenges downstream for debugging.

The Socrates Sim framework was designed to 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 simple. There are three key modules, the user simulator, the dialog agent, and the dialog manager. We assume that the user simulator and dialog agent have been implemented externally. In order to fit into the framework, they need to be integrated into implementations of the *Speaker* abstract base class. The *Speaker* abstract base class provides a set of APIs that allow for the user simulator and dialog agent to com-

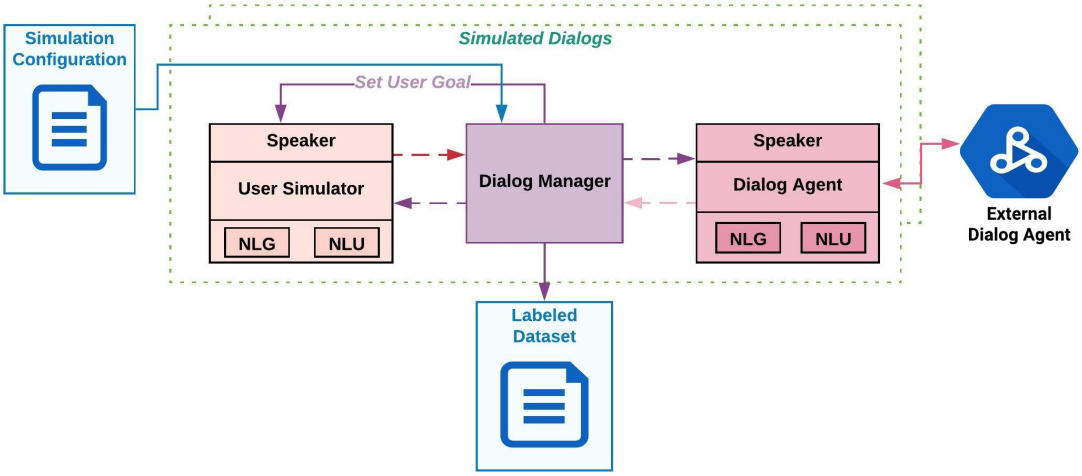

Figure 2.2: Design of the Socrates Sim framework.

municate with the framework in a standardized way. Once the classes are created, their locations are stored in a configuration file that is used by the dialog manager to load speakers and run simulations. The external configuration file supports additional settings and configurations which allow rapid experimentation and make it easy to transition to new domains.

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 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. The internal nlg and nlu objects are implementations of the *NLG* and *NLG* abstract base class. Additionally, communication between the speakers is standardized. Dialog actions, goals, and domain knowledge bases are all first-class objects. This is in stark contrast to the design choice of Li et al. (2017), where similar components are implemented as Python dictionaries. As first class objects, we can standardize the communication and management of these pieces and ensure more consistent behavior.

Another key distinction of Socrates Sim is that the framework is agnostic as 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.

2.2.1 Configuration First Design

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 objective of the approach is to separate out domain-specific logic from implementation details. The configuration file stores information about the domain and domain specific implementation choices. As a result, the implementation code can be written at a higher level and be more flexible and re-targetable.

To support a configuration-driven approach, each configurable module supports the ingestion of a user-defined yaml or json file. For smaller configuration details, the user may prefer to use the yaml format 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.

Socrates Sim is a command line tool. Once the researcher has set up 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 out-

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

Figure 2.3: Example yaml section.

put performance metrics for the dialog agent and store the generated dialogs with annotations.

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

2.3. Dialog Domain and Domain Knowledge Base

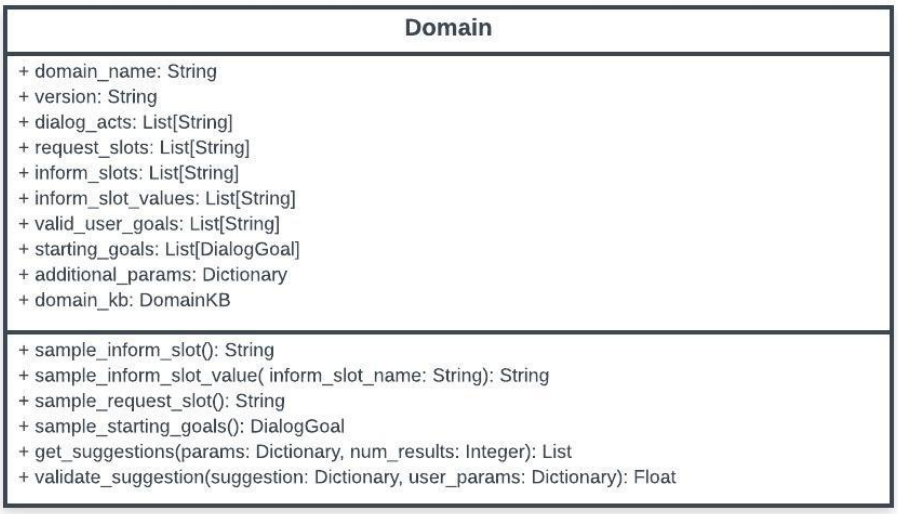

Figure 2.4: Dialog domain class.

The domain class standardizes the collection and storage of information related to the 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 the various 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 object is also provided to the user simulator and dialog agent for use.

The domain knowledge base (KB) is an abstract base class. Its purpose is to define a standardized way for a speaker to query and access the knowledge base. The primary purpose of the domain KB is to 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 the item. The researcher is free to use whichever back-end and implementation to resolve those three API calls.

For this thesis, *DomainKBtable* is an implementation of the *DomainKB* abstract base class. It loads tabular data from a csv file into a pandas data frame. The pandas data frame is a memory efficient data structure that supports querying and data manipulation. More details about the *DomainKBtable* can be found in the implementation chapter.

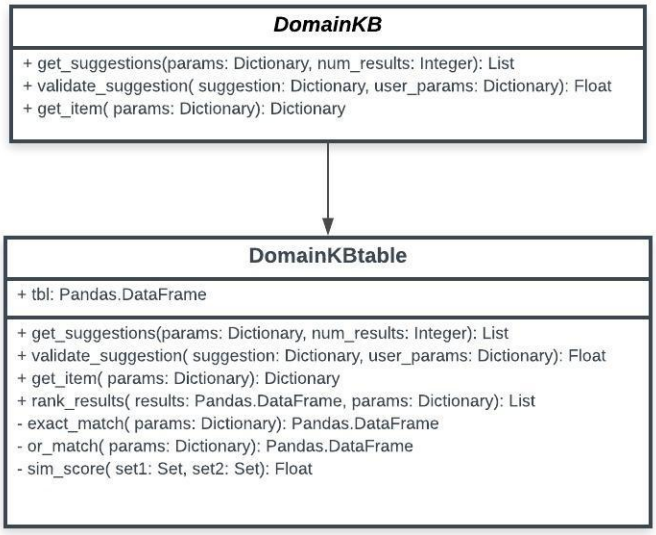

Figure 2.5: Domain knowledge base abstract base class and implementation.

2.4. Speaker Abstract Base Class

The *Speaker* abstract base class represents an actor that has the ability to speak and comprehend speech utterances. In our framework, both the user simulator and the dialog agent are represented by the same base speaker class. Both actors are conceptually identical in terms of functional behavior. They listen and comprehend speech utterances and respond in turn by speaking. Thus, 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 2.14).

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 nlg object and the nlu object are passed to the constructor. We abstract away the implementation of how the speaker speaks and parses speech in order to maintain separation of concerns and also empower the researcher to be able to experiment with multiple techniques.

The *next* method is the primary driver for 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 is external to 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 nlu objects. The primary parameter for next is the previous dialog action.

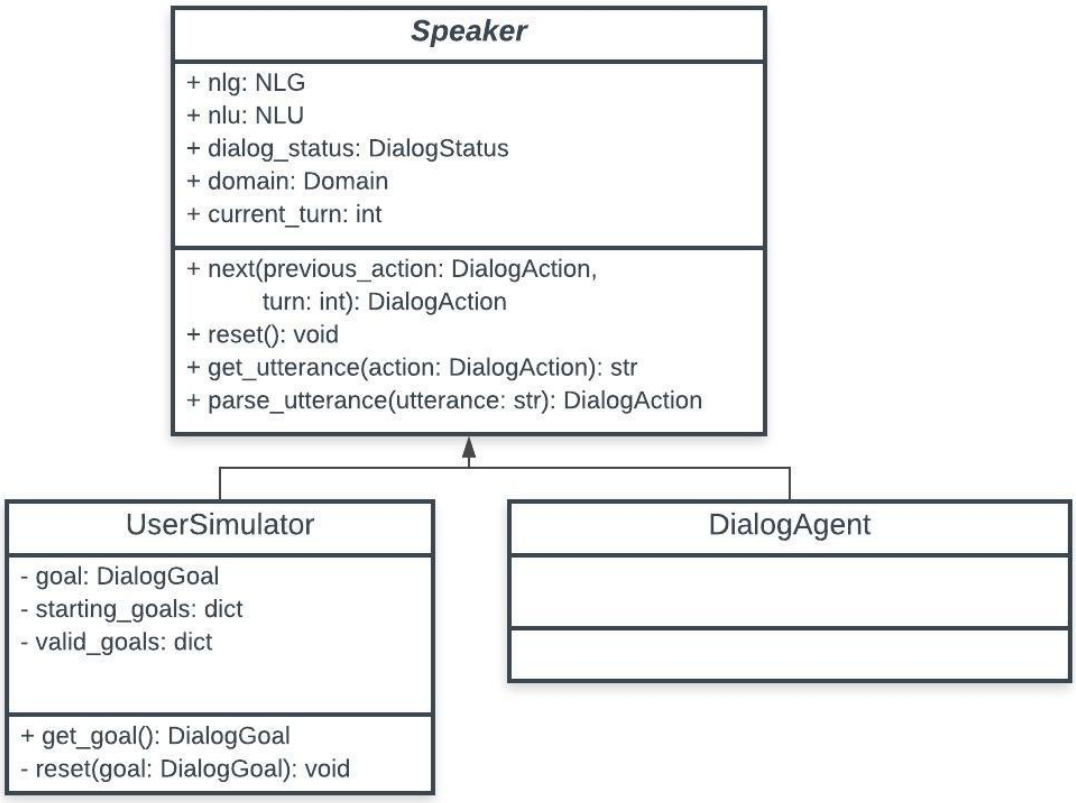

Figure 2.6: Speaker abstract base class and implementations.

2.4.1 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 to complete a 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 by Schatzmann & Young (2009).

One of the primary assumptions here is that the 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 the user's goal by asking the user a set of clarifying questions. Schatzmann and Young introduce 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.

The user agenda and user goal are first class objects and are described in further detail below. Additionally, we use the *UserSimulator* class (see Figure 2.7) as an implementation of the *Speaker* abstract base class. This user simulator is required to be more transparent and accessible in a particular manner by the dialog manager. At the beginning of each round, the dialog manager generates a user goal and updates the user simulator with that goal. Additionally, at the end of each

conversation round, the internal state of the user simulator's goal is extracted and stored in the dialog history that is being actively tracked by the dialog manager. As a result, the researcher must implement a get_goal and override the reset method in addition to the interface methods defined in the *Speaker* abstract base class. The dialog goal is formalized as a *DialogGoal* object and more details about it can be found in section 2.5.2.

User Agenda.

Schatzmann & Young (2009) define the user agenda as a *[stack] structure of pending dialogue acts [which] serve as a convenient mechanism for encoding the dialogue history and user's 'state of mind'*. Formally, at any time t, the user is in a state s_u and takes an action a_u,which transitions into an intermediate state s'_u. During this intermediate state, the user will receive an action from the system (machine), which will transition the dialog to next state s''_u and the cycle will reset. The result is a sequence of alternating turns between the user and system (i.e. $s_u \rightarrow a_u \rightarrow s'_u \rightarrow a'_u \rightarrow s''_u \rightarrow \dots$), which represents the conversation state over time *t*.

The user agenda is a 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 action is a representation of the user's 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 one of the request slots with the 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 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 C_0 (bar serving beer in central) and goals (name, address, and phone for a bar that meets the constraints in C_0). This set of inform and request slots are translated into a user action stored in A_0. When the system initiates the conversation, the user simulator pops two inform actions which translate into the user utterance *I'm looking for a nice bar serving beer.* When the system at t=1, responds *Ok, a wine bar. What price range?* the agenda is updated to include a new inform intent (inform(prange=cheap)). Also added is a negate action, as the user asked beer and not wine.

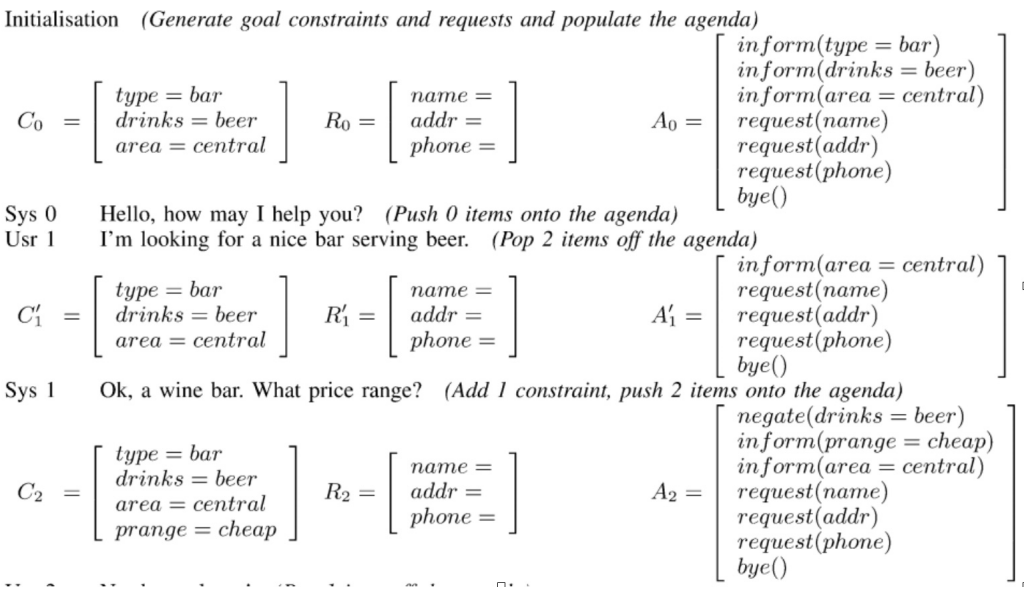

Figure 2.7: Example of user agenda formulation from Schatzmann & Young (2009)

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.

2.4.2 Dialog Agent

The framework assumes that the dialog agent is opaque and can only communicate with the framework through dialog action objects. Therefore the *DialogAgent* class in Figure 2.7 serves more as a template for implementation. Implementing the four methods defined in the *Speaker* abstract base class allows for the dialog agent to communicate with the dialog manager. We explore the implementation details in Section 3.2.2 for the *DialogAgent* class.

2.5. Key Dialog Components

In order to standardize communication, we have formalized the concepts of dialog action, dialog goal, and the nlu and nlg processes. In the sections below, we investigate them in further detail.

2.5.1 Dialog Action

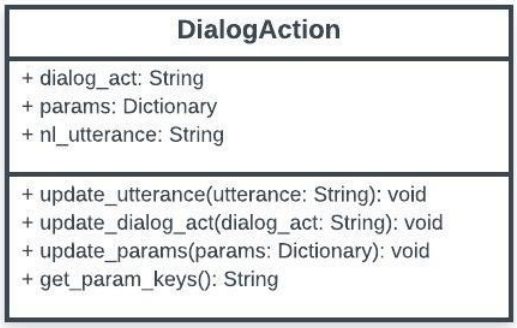

Figure 2.8: Dialog action class.

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 encodes information about a speech act in a standardized way. The dialog action consists of three key properties: the dialog act, a set of explicit dialog parameters or constraints, and the 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 speaker One of the objectives of the user simulator is to generate speech utterances based on its internal agenda. The user simulator will use the dialog act and dialog parameters popped from the user agenda in order to generate a new utterance using its 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 parameters.

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 language 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 the 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 the dialog do-

main configuration file. While this limits the possible interpretations of a speech act, a reduced dialog act space is beneficial to building effective task-completion dialog agents.

The dialog parameters property encodes entities found in the speech utterance as key-value pairs and is used to communicate or elicit the user's preferences. The key captures the entity/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 of 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, we used Python dictionaries and sets to represent the dialog parameters. However, this was sub-optimal for several reasons. First, it introduced variability and uncertainty. To capture the request parameters, we used Python sets, since the value was null and we just needed to pass along the constraints types. For all dialog actions, we had used explicit dictionaries as real values were passed along with the constraint types. However, downstream this required logic to check the class instance of the parameter variable being passed in.

Given the dynamic nature of the dialogs being generated, the simulator was rather unstable and would randomly crash when a method expecting a dictionary received a set. By enforcing the Python dictionary type and setting null values, we were able to greatly improve stability and make debugging easier by standardizing the input into functions that consumed the *DialogAction* object.

2.5.2 Dialog Goal

The user goal captures explicitly the speaker's preferences and missing information they are trying to acquire. For example, consider 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.

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

Figure 2.9: Example user goal. User wants the name, address and phone number of a cheap bar in central. Schatzmann & Young (2009)

Formally, Schatzmann and Young define the user goal 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. Figure 2.9 above shows how one could represent the goal of a user looking for a bar.

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

For the user simulator, the goal defines the hidden set of preferences and

Figure 2.10: Dialog goal class.

information needs the user has. The goal object has two properties: 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 updates 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 the *DialogGoal* object. But it made sense to allow the dialog agent to have access to its own *DialogGoal* object. The rationale behind this was two-fold. First, it provides a useful mechanism to track the state of the dialog agent as well. Like the user, the dialog agent has its own set of goals, 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 complementary to the user's inform slots. The dialog agent can elicit the user's inform slot through a series of request speech acts and then execute its service. This leads us to the second value for the dialog agent, the *DialogGoal* object provides 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 in capturing. For reinforcement learning, we can develop a loss function to minimize the open request slots in the agent's *DialogGoal* at the end of each conversation.

2.5.3 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 its dialog state. If 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 its *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.

2.5.4 NLG and NLU Abstract Base Class

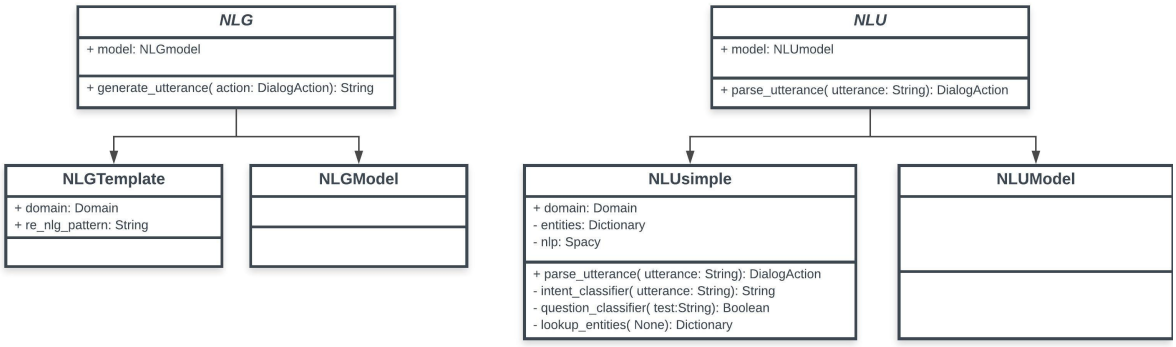

Figure 2.11: NLU and NLG abstract base classes.

The NLU and NLG interface provides a common API for parsing and generating speech utterances for each speaker. Both interfaces define a single public-facing API method: *parse utterance* and *get utterance* respectively. *Parse utterance* will take in a natural language speech utterance and return a *DialogAction* object. *Get utterance* takes in a dialog action and returns a natural language speech utterance.

The researcher has flexibility in setting up the NLU and NLG back-ends. In the implementation section, we will further detail 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 nlg are open problem spaces and there are no 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 dialog action 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.

2.6. Dialog Manager

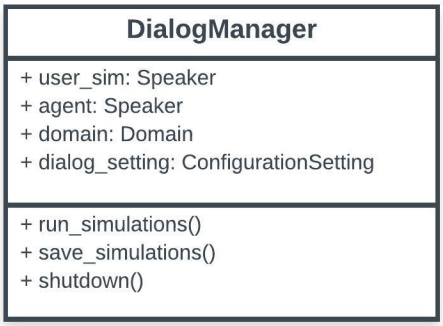

Figure 2.12: Dialog manager class.

The dialog manager 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 be modified by the researcher and is set up to implement the researcher's simulations 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 manager 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 2.13 shows at a high level what the dialog model looks like.

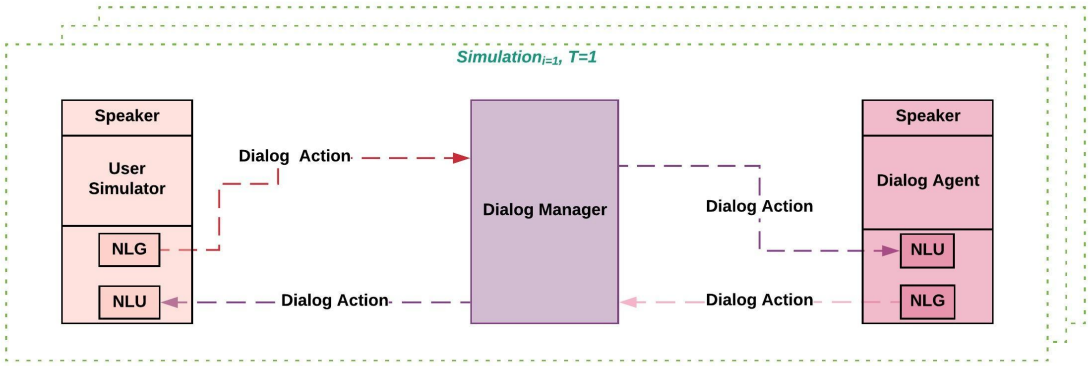

Figure 2.13: Dialog model overview.

The dialog manager follows 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 ready to interact, the simulation is initiated. The logic for conversation is quite straightforward. The speaker class has the 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 in two lines (see Figure 2.14). The dialog action object standardizes how the speakers talk to each other and detail can be found below.

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

```
# 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)
```

Figure 2.14: Pseudo code for conversation round.

Once all the simulations are run, the dialog simulator will run performance metrics to evaluate the efficacy of the dialog agent. If the user simulator was able to fill all the request slots in their hidden goal, the dialog is marked as successful. 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. After all simulations have been run, the dialog manager will serialize the output (if specified by the researcher) and exit. The goal of the output file is to provide quality annotated data that can be used as training data for supervised learning or reinforcement learning based dialog agent.

Chapter 3: Implementing Socrates Sim

3.1. Overview

In this chapter, we describe how to implement and deploy Socrates Sim. We first describe the general process of setting up Socrates Sim. In chapter 4 and chapter 5, we focus specifically on the implementation choices and details for supporting the restaurant recommendation and movie booking domains. These domains were chosen for different strategic reasons. The restaurant domain was selected due to the availability of training data from the Dialog State Tracking Challenge 2 (DSTC2). The dialog acts and slot values are well defined by the organizers of DSTC2. We had to manually scrape restaurant data from Yelp in order to build out the database of restaurants to use for recommendations. We used the training dialog data to train the neural translation models to support model based natural language understanding (NLU) and natural language generation (NLG).

The movie booking domain was selected so that we could run performance tests against the TC-Bot framework Li et al. (2017). TC-Bot was hardcoded to support the movie booking use case. While the TC-Bot code repository provided some domain configuration data, it was ultimately unusable. As a result, we defined a new but analogous domain and manually created a new movie database. As there was no dialog data available for training, we did not implement model-based NLU and NLG components for the user simulator. Instead, we used a rules-based approach.

Please note, we refer to the end user as the researcher, as that is the presumed user of our framework. We aim to demonstrate the value of Socrates Sim directly though applied examples.

3.2. General Process to Deploy Socrates Sim

The Socrates Sim framework is domain and speaker agnostic. All domain, dialog agent, and user simulator specifics have been abstracted away. Therefore the researcher must inform Socrates Sim about the dialog domain and where the speaker classes are located. Prior to running Socrates Sim, the researcher has to prepare the components required to run a simulation. At a high level, the researcher needs to:

- Define the dialog domain
- Develop the interface classes for the user simulator
- Develop the interface class for the dialog agent
- Define the simulation settings

The researcher then can invoke Socrates Sim to run multiple simulations and generate train data. Next, we will briefly explore how to implement the components above. We further explore domain specific implementation details in the proceeding chapters.

3.2.1 Defining the Dialog Domain

The first thing the researcher should do is define the dialog domain. The dialog domain captures the various dialog acts, inform and request slots, slot values, and other information pertinent to their domain. Domain information is formally captured and represented to Socrates Sim as a *domain* object (see section 2.3). This

object is made available to the user simulator, dialog agent, and dialog manager. The object is static and does not change in order to ensure consistent communication between all parties.

The researcher does not need to manually create and instantiate the domain object. All the domain information is captured in a yaml or json configuration file (e.g. Figure 3.1). The yaml format is recommended for small domains that can easily be written out by hand. For larger complex domains, the json format is recommended as it is easier to export programmatically.

```yaml
# Dialog Domain Template

# Domain Information
domain_name: restaurant
version: 1.0

# Dialog Acts
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
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

# Domain Inform Slots and values
inform_slot_values:
  pricerange: [ "cheap", "moderate", "expensive"]
  area: [  "centre", "north", "west", "south", "east" ]
  cuisine: [91 items]
```

Figure 3.1: Sample domain configuration file.

As seen above, the dialog information is captured with key-value pairs. The values in most cases are lists of items (e.g. a list of valid dialog acts). Inform and request slot values are represented as nested dictionaries. Each slot is a nested key that points to a list of valid slot values. Chapter 2 specifically outlines the various domain attributes. The one thing it does not cover is the valid user goals.

Valid user goals are an optional piece of information the researcher can provide to Socrates Sim. At the beginning of each round, the dialog manager will generate

a goal for the user simulator. This goal can be randomly selected from a preexisting list of valid goals or randomly generated from the request and inform slot information in the domain object. In the valid user goals section, the researcher lists the different combinations of valid request values that can be used when generating a goal. This is to ensure the random goal generator does not generate a user goal with no request slots (implying the user has no goal in mind) or accidentally select request slots that produce invalid goals.

Additionally, the researcher may have a knowledge base that they want to provide to the dialog agent. The knowledge base can be used by the dialog agent to make suggestions. We create a simple interface to allow the dialog agent to access information from a knowledge base in a standardized way. The *DomainKBTable* class will take in a csv file and convert it into a pandas data frame. It additionally provides a set of query and sampling methods. The *DomainKBTable* object is a public property of the domain class.

At runtime, the dialog domain information stored in the configuration file is dynamically loaded and converted into *domain* object by the run script. For local testing, the researcher can invoke the *import_domain* method from the *dialog_simulator* module to manually instantiate the object.

3.2.2 User Simulator and Dialog Agent Interface

The process for integrating an external user simulator and dialog agent into Socrates Sim is very similar. The researcher needs to create an interface class that allows Socrates Sim to communicate with both speakers in a standardized manner. This is accomplished by creating a subclass of the base *UserSimulator* and *DialogAgent* (see Figure 3.2). Both these base classes are implementations of the *Speaker* abstract base class described in section 2.4.

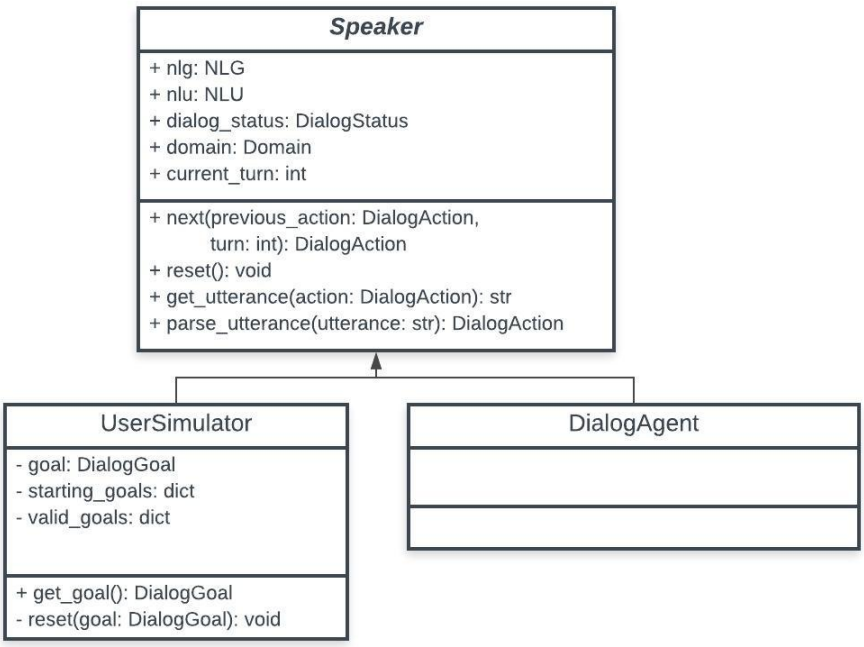

Figure 3.2: Extending Speaker class for user simulator and dialog agent.

The key distinction between both base classes is the requirement of transparency for the user simulator. The framework assumes that the dialog agent is opaque and can only communicate with the framework through dialog action objects. Therefore the *DialogAgent* serves more as a template for implementation. In contrast, the user simulator is required to be more transparent and accessible in a particular manner by the dialog manager. At the beginning of each round, the dialog manager generates a user goal and updates the user simulator with that goal. Additionally, at the end of each conversation round, the internal state of the user simulator's goal is extracted and stored in the dialog history that is actively tracked by the dialog manager. As a result, the researcher must implement get_goal and override the reset methods in addition to the interface methods defined in the *Speaker* abstract base class. The dialog goal is formalized as a *DialogGoa*l object and more details about it can be found in section 2.5.2.

Both the user simulator and dialog agent interface subclasses must implement the following methods: *reset*, *next* *get_utterance*, and *parse_utterance*. The method signatures and return types are strictly defined and enforced to ensure that both speakers communicate with Socrates Sim in a standardized manner. As Python is a dynamically typed language, we use the type hinting feature introduced in Python 3.5 to signal to method contracts and expectations.

The *reset* method is used to reset the internal state of the speaker. That is, all inform/request slots are cleared and any internal state tracking is set back to 0. It is called by the dialog manager at the start of each simulation round. The *reset* method of the user simulator is also expected to take in a dialog goal object, which is generated anew by the dialog manager.

The *next* method is the critical method that drives the simulation. It is used to capture the current speaker's response to its interlocutor. The *DialogAction* class is central to standardizing communication between the speakers and the dialog manager. Each speaker must produce a dialog action object that captures both the raw speech utterance and its semantic representation. The specific design of the *DialogAction* class can be found in section 2.5.1. The *next* method takes in as input the dialog action of the previous speaker and the current turn number. It outputs the current speaker's response as a new dialog action object.

Finally, the *get_utterance* and *parse_utterance* methods are tied directly to the speaker's internal NLU and NLG objects. The *parse_utterance* method will take in a natural language speech utterance and pass it directly to the speaker's internal nlu object for parsing. The *parse_utterance* method will then output a new dialog action object with the parsed semantic frame representation of the utterance. The *get_utterance* does the reverse of *parse_utterance*. It generates a speech response from a provided semantic frame representation. The internals of how the nlg and nlu

objects are defined is up to the researcher. In the restaurant use case, we explore how to use a rules-based and a model-based strategy. It is important to note that the researcher can specify the *NLU* and *NLG* class implementations in the simulation file. By abstracting out the nlu and nlg process from the speaker's internal logic, we support the ability for experimentation and code reuse.

3.2.3 Simulation Configuration

After the researcher has prepared all the requisite components for using Socrates Sim, they need to capture that information in a configuration file. Specifically, the researcher has to provide Socrates Sim with the following: the path to the domain configuration file, the path to the user simulator class, the path to the dialog agent class, and the simulation settings. This information is collected in a yaml or json configuration file that is provided to Socrates Sim at runtime (see Figure 3.3).

```yaml
# Dialog Simulation Configuration

# Dialog Domain Settings
domain_config: sample_domains/restaurant/restaurant_domain.yaml
domain_kb_type: table
domain_kb_file_path: sample_domains/restaurant/restaurants_kb.json
domain_kb_file_type: json

# User Simulator location and settings
usersim_class: sample_domains.restaurant.usersim.RestaurantUserSim
nlg_class: sample_domains.restaurant.usersim_nlg.UserSimNLGTemplate
starting_goal_path: sample_domains/restaurant/sample_starting_goals.yaml
user_goal_type: template

# Dialog Agent location
agent_class: sample_domains.restaurant.agent.RestaurantAgent

# Dialog Simulation settings
first_speaker: usersim
corrupt_goal: .001
user_exit: .01
user_uncertainity: .05
max_turns: 8
simulation_rounds: 10
save_history: True
save_location: data/simulated_dialogs/
save_type: json
```

Figure 3.3: Example simulation configuration file.

The researcher also specifies the various simulation settings (e.g. number of simulations to run, where to store generated dialogs, etc). Figure 3.4 outlines out the

various settings supported by Socrates Sim.

Setting	**Description**	**Valid Values**
first_speaker	Defines who speaks first	user, agent, random
corrupt_goal	Probability the user's goal is randomly corrupted resulting in random user behavior	valid probability in range [0.0, 1.0]
user_exit	Probability user randomly exits conversations	valid probability in range [0.0, 1.0]
user_uncertainty	Probability user randomly changes mind and goal	valid probability in range [0.0, 1.0]
max_turns	Maximum amount of conversation turns before simulation is aborted.	Reasonable real integer greater than 0
simulation_rounds	Number of simulations to run.	Reasonable real integer greater than 0
save_history	Flag to indicate if generated dialogs should be saved	True or False
save_location	Location to store generated dialogs	valid save path

Figure 3.4: Supported simulation settings.

We made a deliberate choice to adopt a configuration file first approach to communicate settings and other information to our framework. We drew inspiration from Gardner et al. (2018), which uses configuration files for the communication of model details and experiment parameters to the AllenNLP framework. In traditional command line tools, parameters are passed to the tool by command-line arguments. For example, Figure 3.5 contains the command line parameters to run 300 simulations on the TC-Bot simulators.

The challenge with this approach is that command line invocation can grow more complex as more parameters are added. Long invocation commands needs to be carefully formatted to ensure there are no line breaks and proper spacing is observed to separate out the arguments. This often results in a poor user experience. It is cumbersome to remember and cleanly enter all the information required each time

```
python run.py --agt 9 --usr 1 --max_turn 40
              --movie_kb_path ./deep_dialog/data/movie_kb.1k.p
              --dqn_hidden_size 80
              --experience_replay_pool_size 1000
              --episodes 300
              --simulation_epoch_size 100
              --write_model_dir ./deep_dialog/checkpoints/rl_agent/
              --slot_err_prob 0.00
              --intent_err_prob 0.00
              --batch_size 16
              --goal_file_path ./deep_dialog/data/user_goals_first_turn_template.part.movie.v1.p
              --trained_model_path ./deep_dialog/checkpoints/rl_agent/noe2e/agt_9_478_500_0.98000.p
              --run_mode 3
```

Figure 3.5: Command line parameters to run 300 simulations with TC-Bot.

the program is run. In practice, most users end up creating a reference file from which they copy and paste the command line statement. Socrates Sim supports both json and yaml formatted configuration files. Yaml is the recommended format as it is more human-readable and is easy to write. If the researcher wants to generate configuration files as part of a larger pipeline or does not like yaml, the json format is also supported.

3.3. Simulating Dialogs with Socrates Sim

The last step in the process is actually running Socrates Sim and generating simulated dialogs. The framework is implemented as a command line tool. The user invokes a run script with the location of the simulation configuration file. If the researcher chooses to save the output, the generated dialogs will be serialized as a json file. Additionally, the researcher can signal Socrates Sim to print the simulation in a terminal using the *-pd* flag.

Behind the scenes, the run script first loads in the simulation configuration file. Next, it initiates the dialog manager and passes along all the simulation settings. The dialog manager first loads the various components (reading in the dialog domain, initializing the interface classes for the user simulator and agent, etc). Next,

```
/Users/dhairya/Documents/envs/bin/python /Users/dhairya/Documents/socrates/src/run_simulator.py -p sample_domains/restaurant/simulation_config.yml -t yaml -pd
    Running simulation 1 of 1
     0 :  usersim :  Hi! Can you help me find a restaurant?
          DialogAction:  greetings {}
     0 :  agent :  What kind of cuisine are you looking for?
          DialogAction:  request {'cuisine': None}
     1 :  usersim :  I want to eat chinese food.
          DialogAction:  inform {'cuisine': 'chinese'}
     1 :  agent :  What is your preferred area?
          DialogAction:  request {'area': None}
     2 :  usersim :  I'm looking for a restaurant in central.
          DialogAction:  inform {'area': 'central'}
     2 :  agent :  What is your price range?
          DialogAction:  request {'pricerange': None}
     3 :  usersim :  I'm looking for a expensive priced restaurant.
          DialogAction:  inform {'pricerange': 'expensive'}
     3 :  agent :  The phone number of Golden Dynasty is 343-433-4433 and the address is 1234 Street
          DialogAction:  inform {'phone': '343-433-4433', 'name': 'Golden Dynasty', 'address': '1234 Street'}
     4 :  usersim :  Thanks!
          DialogAction:  bye {}
     4 :  agent :  Thank you for using the restaurant recommend service.
          DialogAction:  bye {}
    Dialog Result:  Success
Summary
Ran 1 simulation. 100% dialogs were successful.
Successfully wrote dialog histories to data/simulated_dialogs/
```

Figure 3.6: Sample run of the Socrates Sim framework.

it starts the simulation process and facilitates the various conversations between the user simulator and dialog agent. At the end of each conversation round, the dialog manager evaluates the conversation based on the end state of the user simulator's goal object. If all the request slots in the goal object are filled, the dialog manager grades the simulation as successful.


```
[
    {
        "dialog_grade": "Success",
        "dialog_id": "58c197df-df34-48d9-a25f-e58dbfeacfca",
        "goal_grade": "Success",
        "starting_user_goal": {
            "inform_slots": {
                "area": "central",
                "cuisine": "chinese",
                "pricerange": "expensive"
            },
            "request_slots": {
                "address": "UNK",
                "name": "UNK"
            }
        },
        "turn_history": [
            {
                "first_speaker": {
                    "dialog_act": "greetings",
                    "dialog_slots": {},
                    "nl_utterance": "Hi! Can you help me find a restaurant?",
                    "speaker": "usersim"
                },
                "second_speaker": {
                    "dialog_act": "request",
                    "dialog_slots": {
                        "cuisine": null
                    },
                    "nl_utterance": "What kind of cuisine are you looking for?",
                    "speaker": "agent"
                },
                "turn": 0,
                "user_goal": {...}
            },
            {"turn": 1...},
            {"turn": 2...},
            {"turn": 3...},
            {"turn": 4...}
        ],
        "turns_taken": 5
    }
]
```


Figure 3.7: Sample output after running one simulation round.

After all simulations have been run, the dialog manager will serialize the output (if specified by the researcher) and exit. Figure 3.7 shows an example output

generated after running one simulation. The goal of the output file is to provide quality annotated data that can be used as training data for supervised learning or reinforcement learning based dialog agents. The output contains several pieces of key information. Each simulation is assigned a unique id, information about the number of turns taken, the grade of the simulation, the user's initial starting goal, and turn history, which contains the annotated history of the conversation round.

Socrates Sim is written to be extensible. Given its modular nature, the researcher has the ability to extend the framework to support new use cases. Next, we show how Socrates Sim was adapted for the restaurant and movie domains.

Chapter 4: Implementation Use Case: Restaurant Recommendation

4.1. Overview

In this chapter, we demonstrate how Socrates Sim was adapted to support the restaurant domain. We start with a dialog agent that makes restaurant recommendations. The objective of Socrates Sim will be to produce a set of simulated conversations between a user simulator and the dialog agent. The user simulator will emulate a human user and come into the conversation with a hidden goal (for example, get the name and phone number for a Greek restaurant). 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, either the user or the dialog agent can take the first turn.

As mentioned in the previous chapter, the framework requires knowledge of the dialog domain, the path locations of the user simulator and dialog agent, and the simulation settings. We will walk through how each of those components are created and then used to run simulations on Socrates Sim.

4.2. Defining the Dialog Domain

We use the Dialog State Tracking Challenge 2 (DSTC2) restaurant recommendation ontology and data for our use case. The DSTC2 was a research challenge put together by the University of Oxford and Microsoft to advance dialog research. For DSTC2, the goal was to track the state of multi-stage conversations between real humans and an expert restaurant recommender service. The recommender service provides recommendations based on the following user preferences: cuisine, area, and price range. DSTC2 provides a rich and deep set of training data that was used to train models for the natural language generation (NLG) and the natural language understanding (NLU) components of the use simulator.

The first thing we do is create the dialog domain configuration file (see Figure 4.1). This file is used to generate a domain object that will be used by the dialog manager, user simulator, and dialog agent. We use the dialog DSTC2 ontology to define the dialog acts, slots and slot values in the domain configuration file. To simplify the use case, we did narrow down the dialog act options. The request and inform slots and slot values are the same.

One important and novel feature of this file is the valid user goal templates section. It specifies the valid goals a user is allowed to have when engaging the dialog agent. That section enumerates the various combinations of required request slots that must be present in the user goal.

The knowledge base for this use case is a csv file with the various restaurants from the Cambridge (UK) 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 from Yelp.com. The restaurants' list is loaded into memory as a *DomainKBtable* object. The knowledge base is converted into a pandas

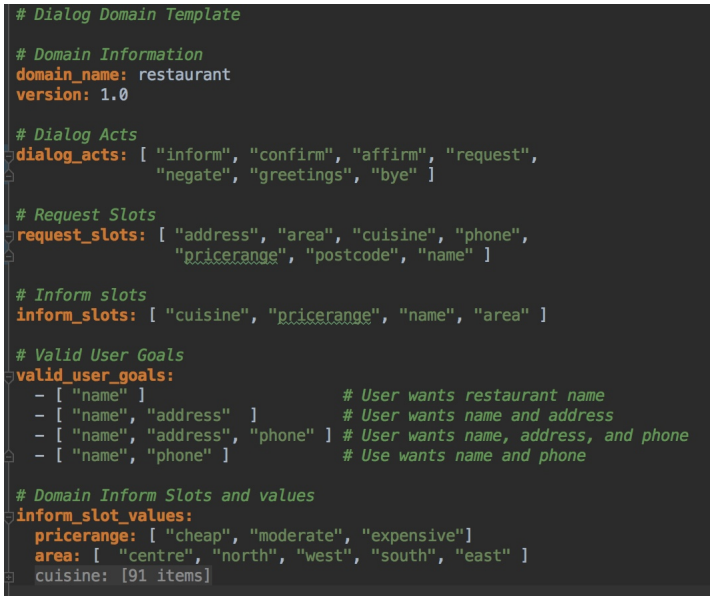

Figure 4.1: Restaurant dialog domain configuration file.

dataframe and the class provides a set of methods for querying. The knowledge base is primarily used by the dialog agent to make recommendations.

4.3. User Simulator Implementation

In this section, we describe the implementation of a rule-based user simulator. The user simulator is initialized with a set of hidden preferences (see Figure 4.2) and aims to get the name of a restaurant that satisfies those preferences from the dialog agent. Additionally, the user simulator may also want the restaurant's phone number and address.

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

Figure 4.2: Example user goal. User is seeking the name and phone number of a cheap Chinese restaurant

In order to integrate the user simulator into Socrates Sim, we define an interface class that is the descendant of the *Speaker* abstract base class. Our rules-simulator class is implemented as a subclass of the *UserSimulator* class. The rules-simulator needs to implement four key public methods: *reset*, *next*, *get_utterance*, and *parse_utterance*.

At the start of each simulation round, the dialog manager calls the user simulator's *reset* method. The purpose of the *reset* method is to set a user goal and reset the user simulator's internal state. The dialog manager passes a new *DialogGoal* object to the *reset* method. In this method, we also reset the enum value of *DialogStatus* to NOT_STARTED and current turn to -1.

The primary logic of the user simulator is captured in the *next* method, which we describe in 4.3.1. The nlg and nlu processes are implemented as separate classes and described further in detail below. To demonstrate the versatility of the simulator, we implement both a rule-based and model-based approach for both the nlu and nlg. We can easily swap out implementations in the simulation configuration file, without having to modify the user simulator code directly.

4.3.1 Simulator Logic

The internal logic for the user simulator is driven by the user agenda. The user agenda captures what the user simulator will communicate to the dialog agent. For our implementation, we simply concatenate the inform and request slots stored in the user goal.

Schatzmann & Young (2009) formally define the user agenda as a stack-like structure containing the 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 on the stack, which gets popped, contains the dialog action for what the user simulator will do next. That action is passed to the simulator's internal nlg process to generate a new speech utterance.

For memory efficiency, we do not actually implement the agenda as an explicit stack. The information already exists in the user goal. Instead, we pop directly from the inform slots list in the user goal in response to the dialog acts from the dialog agent. We keep track of the conversation state by checking 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 with one of two values: NO_OUTCOME_YET or FINISHED.

The logic for how the user simulator responds to incoming speech acts from the dialog agent is handled by the *next* method.

The user simulator first parses the agent's incoming dialog action and then responds to it using an internal dispatch tree (see Figure 4.3). The dispatch tree is implemented as a Python dictionary, where the dialog acts are mapped to resolver functions. Each response function has the same method signature. The response

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

Figure 4.3: Internal dispatch tree for the rules-simulator.

functions take in a *DialogAction* object and return a new *DialogAction* object that represents the user's response to the dialog agent.

How the resolvers handle the greetings, bye, and confirm responses is straightforward. The resolver simply invokes the *NLG* object's *get_utterance* method and sends a response dialog action. In the greetings case, if the rules-simulator is the first speaker, it will randomly pop a subset of its inform slots that will be used to inform the dialog agent.

The resolver for responding to the agent's request actions is a bit more complicated. The simulator will look at the parsed request slots types the agent is asking about and attempt to find the corresponding inform slots in its *DialogGoal* object. For example, if the dialog agent asks *What cuisine do you prefer?*, the simulator will look up cuisine in its internal goal object and return the answer Chinese (based on the goal in 4.2). For the case where the 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 is 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 informed the dialog agent of all its preferences, the simulator will then pop values from its request slots and ask for a recommendation.

The dialog agent will use the inform dialog act in order to provide the user simulator with new information (e.g., a restaurant suggestion or the restaurant's phone number). If the dialog agent sends an inform action, it will be routed to the inform resolver. This method updates the user simulator's request slots with newly provided information. So for example, if the dialog agent made the suggestion, *Check out Golden Dynasty*, the *UNK* value in 4.2 would be replaced by "Golden Dynasty". If all the *UNK* values in the request slots are filled with real values, the user simulator will update its internal status to FINISHED and issue the bye action.

4.3.2 Natural Language Understanding

The goal of nlu is to parse a natural language utterance and generate a semantic frame representation for the *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. We implement two different nlu strategies to demonstrate the versatility and modularity of Socrates Sim. The first approach leverages a combination of simple NLP rules and some basic heuristics. We call this approach the rule-based approach. The second approach is model-based, where we trained a neural machine translation model to translate a natural language utterance into a semantic frame representation.

Since the nlu process is abstracted from the user simulator, we created an interface class to invoke the nlu logic for both approaches. We created the *NLUSimple* and *NLUModel* classes to support each approach accordingly. Both classes are implementations of the *NLU* abstract base class.

rule-based NLU.

For the rule-based approach, our parser has a two-part strategy. The first is to classify the intent of the utterance and map it to a dialog act. The second is to 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,we run the utterance through a set of regular expression matches and return the corresponding dialog acts (see 4.4). Given the wide range of potential string matches for inform, we use *inform* as the default dialog act.

```
Classify Intent
input: natural language 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"
```

Figure 4.4: Intent classifier algorithm

After the intent is classified, we check to see if any entities are found in the utterance that can be mapped to known entity types. To achieve this, we first look up all the slot values defined in the domain object. Next, we create a reverse map dictionary, where each unique slot value (the entity) is mapped to a slot type (e.g.,

cuisine or price range). The nlu parser tokenizes the utterance into a set of word trigrams and checks 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.

Model-based NLU.

We also implement two simple neural machine translation models for our NLU module. The DSTC2 provides a large corpus of conversations between human users and a human expert posing as the dialog agent. Between the training and development sets, there are about 2,000 annotated calls. All speech utterances (between the human and agent) are annotated to include dialog act, inform/request slots and values.

Our first neural machine translation model is adapted from Brownlee (2017). It is a simple LSTM encoder-decoder model. We use Adam for optimization and categorical cross-entropy for the loss function. The model was implemented in Keras and trained over 100,000 epochs on 1,600 input calls. Unfortunately, the model performance was rather poor. It has a .6 cross-validation accuracy and a .45 accuracy against the test set. While this model is unusable, we do use it in the runtime performance evaluation, as it was simpler to incorporate the model into the simulation workflow.

Given the poor performance of the Brownlee (2017) model, we trained a new neural-machine translation model using OpenNMT (Guillaume Klein (2017)). OpenNMT is an open source neural-machine translation tool produced and maintained by Alexander Rush and the Harvard NLP group. We trained the model on 1,600 input calls and saw significant model performance improvement (cross-validation accuracy of .7 and test accuracy of .65). One of the limitations of this approach using OpenNMT is that the tool requires you to use its command line tool for generating

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 19, 256)           105472
_________________________________________________________________
lstm_1 (LSTM)                (None, 256)               525312
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 9, 256)            0
_________________________________________________________________
lstm_2 (LSTM)                (None, 9, 256)            525312
_________________________________________________________________
time_distributed_1 (TimeDist (None, 9, 109)            28013
=================================================================
Total params: 1,184,109
Trainable params: 1,184,109
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.5: Neural machine translation model architecture - Brownlee (2017)

new predictions. The library currently does not support programmatic access to the trained model. As a result, we had to develop a rather convoluted process to incorporate it into the Socrates Sim. Our *NLUModel* class first writes the utterance to a text file. Next it invokes OpenNMT tool as a subprocess with the input text file. OpenNMT generates a prediction and writes it to an output text file. That file is read in and converted into a *DialogAction* object.

4.3.3 Natural Language Generation

The objective of the NLG module is to generate a natural language utterance, provided a dialog action. Like the nlu use case above, we implement both a template-based model and a neural model.

Since the nlg process is abstracted from the user simulator, we implemented an interface class to invoke the nlg logic for both approaches. We created the *UserSimNLGTemplate* and *UserSimNLGModel* classes to support our two different approaches. Both classes are implementations of the *NLG* abstract base class.

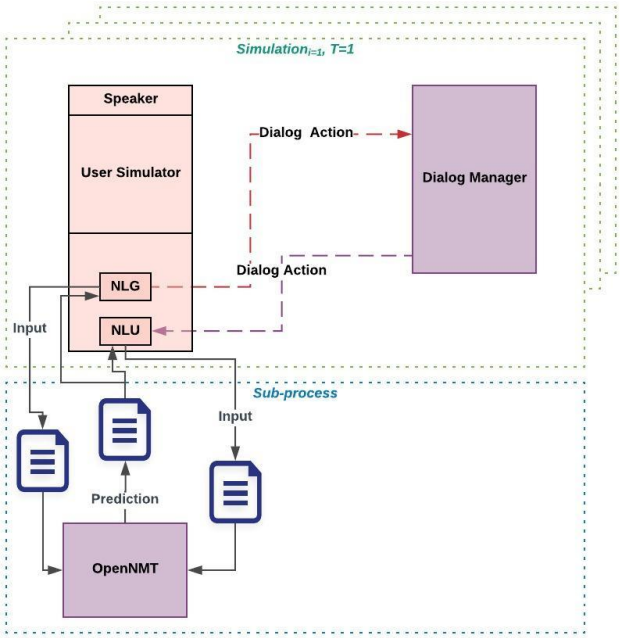

Figure 4.6: NLU process using OpenNMT.

Template-Based Model.

The template model is defined by a yaml file (see figure 4.7). 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 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.

The *get utterance* method in the user 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 through the dialog params property of the *DialogAction* object. Additionally, language templates that have multiple slot type are indexed by the combination of the slot types into a single

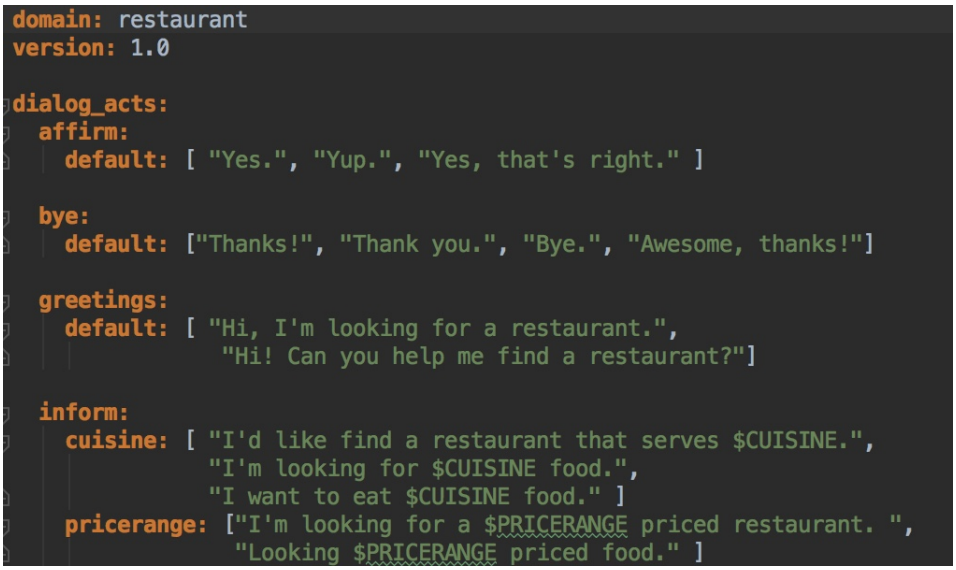

Figure 4.7: NLG template for user simulator.

string. We take the slot types, lowercase them, arrange them by alphabetical order, and concatenate them together with a 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 templates, substitute slot values, and return a generated natural language utterance.

4.3.4 Neural Model for NLG

We follow the same process for developing the neural-machine translation model for nlg as we did for the nlu model. We swap the input and target. The model takes in a dialog action and returns a natural language utterance. The Brownlee (2017) model achieved a cross-validation accuracy of .45 and had an accuracy of .34 against the test set. The OpenNMT model had an improved cross-validation accuracy of .65 and test accuracy of .56.

4.4. Dialog Agent Implementation

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 of 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 rule-based 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 own internal goal (4.8).

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

Figure 4.8: Example goal for restaurant agent.

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

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 its services
- request: the agent will ask a probing question to elicit the user's preferences
- inform: the agent will supply the user with information (usually tied to the user's 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

4.5. Simulating Dialogs with Socrates Sim

Once we have all the component pieces, we are able to run Socrates Sim. To run the simulations, we first create a simulation configuration file (Figure 4.9). As we mentioned above, the nlg and nlu processes are abstracted from the user simulator. We are able to swap in the different approaches for both nlg and nlu by modifying the *nlg_class* and *nlu_class* property of the configuration file. At load time, these classes are dynamically loaded and the user simulator is able to seamlessly use them without explicit integration code.

The user simulator and dialog agent classes are also dynamically loaded. If the researcher has alternative speaker classes, they can easily plug them in by updating the *usersim_class* or *agent_class* properties of the configuration file.

In the Dialog Simulation settings section, we lay out the different simulation settings. For this demonstration, we run a single simulation with various user simulator settings. Additionally, we save the generated dialogs to a json file. Finally, we run the simulation using the command line invocation. Figure 4.10 shows two examples of dialogs generated by the simulator.

```yaml
# Dialog Simulation Configuration

# Dialog Domain Settings
domain_config: sample_domains/restaurant/restaurant_domain.yaml
domain_kb_type: table
domain_kb_file_path: sample_domains/restaurant/restaurants_kb.json
domain_kb_file_type: json

# User Simulator location and settings
usersim_class: sample_domains.restaurant.usersim.RestaurantUserSim
nlg_class: sample_domains.restaurant.usersim_nlg.UserSimNLGTemplate
nlu_class: sample_domains.restaurant.usersim_nlu.NLUSimple
starting_goal_path: sample_domains/restaurant/sample_starting_goals.y
user_goal_type: template

# Dialog Agent location
agent_class: sample_domains.restaurant.agent.RestaurantAgent

# Dialog Simulation settings
first_speaker: usersim
corrupt_goal: .001
user_exit: .01
user_uncertainity: .05
max_turns: 8
simulation_rounds: 1
save_history: True
save_location: data/simulated_dialogs/
save_type: json
```

Figure 4.9: Simulation configuration file.

```
Running simulation 1 of 10
    0 :  usersim :  Hi! Can you help me find a restaurant?
         DialogAction:  greetings {}
    0 :  agent :  What kind of cuisine are you looking for?
         DialogAction:  request {'cuisine': None}
    1 :  usersim :  I'd like find a restaurant that serves chinese.
         DialogAction:  inform {'cuisine': 'chinese'}
    1 :  agent :  What is your preferred area?
         DialogAction:  request {'area': None}
    2 :  usersim :  I'm looking for a restaurant in central.
         DialogAction:  inform {'area': 'central'}
    2 :  agent :  What is your price range?
         DialogAction:  request {'pricerange': None}
    3 :  usersim :  Looking expensive priced food.
         DialogAction:  inform {'pricerange': 'expensive'}
    3 :  agent :  The phone number of ali baba is 343-433-4433 and
         the address is 1234 Street
         DialogAction:  inform {'address': '1234 Street', 'phone': '343-433-4433',
         'name': 'ali baba'}
    4 :  usersim :  Thank you.
         DialogAction:  bye {}
    4 :  agent :  Thank you for using the restaurant recommend service.
         DialogAction:  bye {}
  Dialog Result:  Success
```

```
Running simulation 2 of 10
    0 :  usersim :  Hi, I'm looking for a restaurant.
         DialogAction:  greetings {}
    0 :  agent :  What kind of cuisine are you looking for?
         DialogAction:  request {'cuisine': None}
    1 :  usersim :  I want to eat chinese food.
         DialogAction:  inform {'cuisine': 'chinese'}
    1 :  agent :  What is your preferred area?
         DialogAction:  request {'area': None}
    2 :  usersim :  I don't care
         DialogAction:  inform {'no_pref': 'UNK'}
    2 :  agent :  What is your preferred area?
         DialogAction:  request {'area': None}
    3 :  usersim :  up for anything.
         DialogAction:  inform {'no_pref': 'UNK'}
    3 :  agent :  What is your preferred area?
         DialogAction:  request {'area': None}
    4 :  usersim :  up for anything.
         DialogAction:  inform {'no_pref': 'UNK'}
    4 :  agent :  What is your preferred area?
         DialogAction:  request {'area': None}
    5 :  usersim :  whatever
         DialogAction:  inform {'no_pref': 'UNK'}
    5 :  agent :  What is your preferred area?
         DialogAction:  request {'area': None}
    6 :  usersim :  up for anything.
         DialogAction:  inform {'no_pref': 'UNK'}
    6 :  agent :  What is your preferred area?
         DialogAction:  request {'area': None}
    7 :  usersim :  whatever
         DialogAction:  inform {'no_pref': 'UNK'}
    7 :  agent :  What is your preferred area?
         DialogAction:  request {'area': None}
  Dialog Result:  Failed
```

Figure 4.10: Sample dialog generated by Socrates Sim.

Chapter 5: Implementation Use Case: Movie Booking

5.1. Overview

In this chapter, we describe how Socrates Sim is adapted to support the movie domain. The goal of the movie booking agent is to help the end user purchase movie tickets. A similar dialog agent was developed for the Li et al. (2016) paper. Unfortunately, we were unable to reuse their agent and domain knowledge base. Much of their data was crowdsourced from Amazon Mechanical Turk. There was a confusing overlap between the dialog action space for the user and agent, which resulted in the user simulator generating unrealistic utterances and dialog acts. There was also a high ratio of noise and data quality issues.

Drawing inspiration from their use case, the agent we developed is simpler and modeled after something you would see on a site like Fandango. The agent will help identify movies for the user based on the user's preferences and also collect the user's payment details if the user chooses to book a ticket. As the goal for this implementation is mainly to demonstrate the end-to-end dialog simulation using a user simulator, we did not implement an actual reservation database, a payment processing system, and a large movie showtimes catalog. There are stubs for where the dialog agent would consult external resources while aiding the user.

We focus below on the implementation details for the user simulator and domain modeling exercise. One of the primary benefits of Socrates Sim is that it can

be retargeted to new domains. We do not need to reimplement the framework or dialog manager. We first define the new domain specific components (i.e., the dialog domain, user simulator, and dialog agent). Then we create a new simulation configuration file. Due to the modular nature of the Socrates Sim, we are also able to reuse much of the code developed for the restaurant domain. Most of the modules are written to be domain agnostic, and read in the domain-specific logic from external configuration files. This allows for rapid experimentation and code conservation.

5.2. Defining Dialog Domain

For the movie domain, we first set up the domain configuration file. This file is used to generate the domain object that will be used by both the user simulator and dialog agent. The inform and request slots reflect information that will be communicated and captured specifically for the movie book use case. For this use case, the user's preferences include movie genre, minimum movie rating, showtimes, and theater location. From the dialog agent's point of view, it will need to elicit the number of tickets the user wants to buy, the name of the movie, preferred showtimes, and theater location. Additionally, the agent will collect the user's credit card information to complete the "purchase" of the tickets.

The domain configuration file was designed to be flexible. Compared to the the configuration file for the restaurant domain, we introduce two new sections, *required_slots* and *random_integer_range*. The required slots property indicates the inform slots that must be included in the user's goal. We require that the user goal include credit card information in the inform slots so that it can be presented to the dialog agent for "purchasing" tickets. The *random_integer_range* is used to bound the range for generating a random positive integer. This random integer is used to indicate the number of tickets the user simulator wishes to purchase.

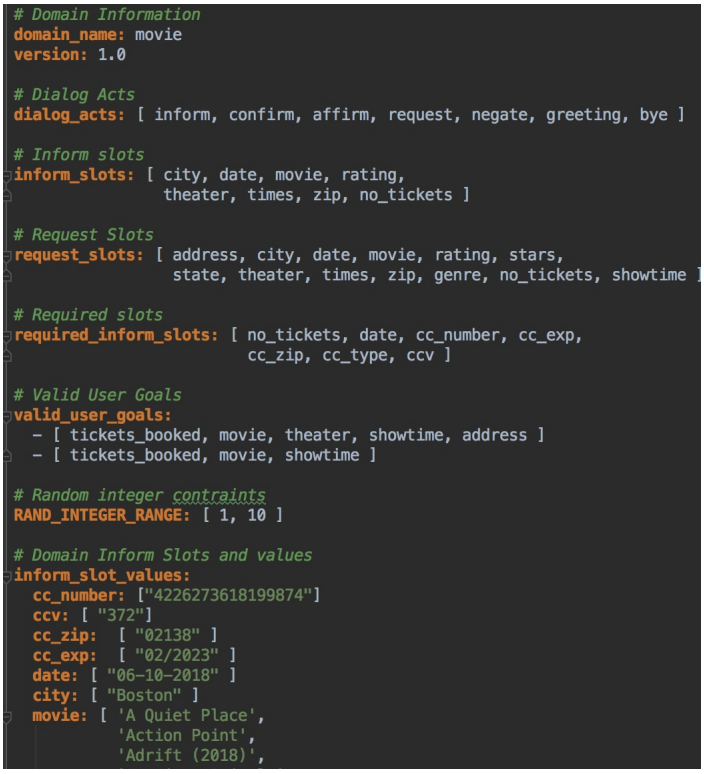

Figure 5.1: Movie booking dialog domain configuration file.

The dialog configuration file is automatically converted into a domain object when Socrates Sim is loaded. Standard domain information (dialog acts, inform and request slots, valid goals) is mapped directly as object properties. Additional information is stored in a dictionary called *additional_params* in the domain object. This design allows the user flexibility in defining the domain while maintaining a minimum level of standardization.

5.2.1 Domain Knowledge Base

The dialog agent uses a database of movie showtimes to lookup movie showings and makes movie recommendations. We generated a sample movie database by scraping Fandango movie data. Since the data is only meant to be a proof of concept, we limited scraping to a single location (movie theaters within 2 miles of Cambridge,

MA) and showings for one week. We collected show times for about nine movies across six theaters offered and all possible showtimes.

Unfortunately, Fandango does not make available genre information and star ratings about the movies on the website. We randomly assigned each unique movie genre from six genres (action, romance, comedy, adventure, thriller, and horror) and star rating score from 1 to 5. The scraped data was stored in a csv file. Like the restaurant domain, I created a *DomainKBtable* class to provide a standard interface to query and sample data from the knowledge base. The knowledge base is represented as a pandas data frame in memory.

5.3. User Simulator Domain

The underlying code and logic for the rule-based user simulator are similar to that of the user simulator used in the restaurant domain. We updated the goal space to support new goals relevant to the movie booking use case.

```
# Goal 1. Purchase 2 tickets for A Quiet Place @ Assembly Row
-
  inform_slots:
    movie: "A Quiet Place"
    date: "06-10-2018"
    theater: "AMC Assembly Row 12"
    city: "Boston"
    no_tickets: "2"
    cc_number: "4226273618199874"
    ccv:  "372"
    cc_type:  "Visa"
    cc_zip:  "02138"
    cc_exp: "02/2023"

  request_slots:
    no_tickets: "UNK"
```

Figure 5.2: Example user goal for movie booking domain.

The internal logic for the user simulator is driven by a user agenda. For our implementation, we simply concatenate the inform and request slot lists stored in the user goal. For memory efficiency, we do not actually implement the agenda as an explicit stack. That information already exists in the user goal. Instead, we

pop directly from the inform slots list in the user goal in response the dialog acts from the dialog agent. We keep track of the conversation state by checking 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 with one of two values: NO_OUTCOME_YET or FINISHED.

The logic for how the user simulator responds to incoming speech acts from the dialog agent is handled by the *next* method. We reused the resolvers described in section 4.3.1.

```
self.response_router = {
        "greetings": self._respond_general,
        "inform": self._respond_to_suggestion,
        "random_inform": self._respond_random_inform,
        "request": self._respond_request,
        "confirm": self._respond_confirm,
        "bye": self._respond_general
}
```

Figure 5.3: Internal dispatch tree for the movie booking user simulator.

The nlg and nlu processes are implemented as separate classes and described further in detail below. We can easily swap out implementations via the simulation configuration file, without having the modify the user simulator code directly.

5.3.1 NLG and NLU

For the movie domain, there were no publicly available datasets to seed a neural-based nlu model. We reused the *NLUSimple* class from the restaurant domain. No new code had to be written. The rule-based model ingests a domain object and uses the domain's inform and request slot values for parsing. It should be noted that the rule-based nlu is brittle as it will use the literal slot values for reverse match-

ing the entity. If the user simulator generates paraphrase or alters the spelling of the entity, the model will fail to map the entity to the appropriate slot type. For demonstration purposes, this is acceptable, as the goal of this use case is to demonstrate retargetability and flexibility of the framework.

We reused the *UserSimNLGTemplate* class for natural language generation. Like *NLUSimple*, the class is able to generalize to new domains by ingesting an external template file. We created a new nlg template file to support the various utterances that the user might say based on the combination of dialog act and dialog parameters.

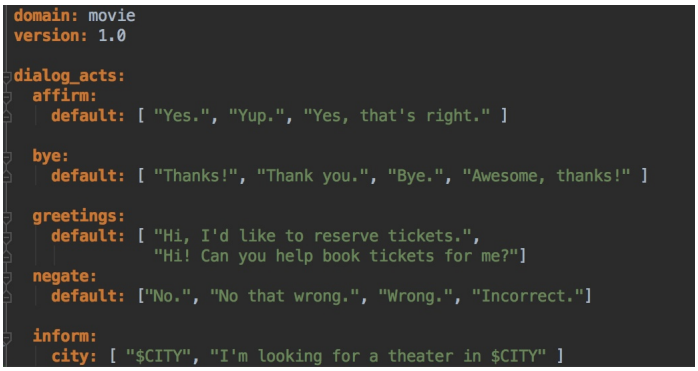

Figure 5.4: Excerpt from nlg template for movie booking domain

The key takeaway for this section is that no new code had to be written to support the nlu and nlg process for the movie booking user simulator. This demonstrates the value of creating a higher level NLU and NLG abstract base class. If the researcher wishes to build a more robust and finely tuned nlu and nlg models, they can easily plug it into the framework by extending the appropriate interface implementations of *NLUSimple*/*NLUModel* and the corresponding nlg classes.

5.4. Dialog Agent

The movie booking agent was developed to illustrate how to incorporate an external dialog agent into the simulation framework. Since we do not have an existing movie booking 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 movie showings. Like the user simulator, the public facing methods the dialog manager interacts with are *next* and *get utterance*.

For the movie booking 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.

If the movie booking agent goes first, it issues a greetings action. If the agent is not responding to the user, it will then sequentially pop one item from it request slots and issue a request dialog act. Once the movie agent has collected information from the user, it will attempt to make a suggestion for movie showings from the knowledge base (*DomainKBtable* object) stored in the domain object. This is accomplished by calling the get suggestion method provided by the domain object. If the user accepts the suggestion. The dialog agent will then ask the user for information about their credit card details.

5.5. Generating Sample Dialogs with Socrates Sim

Once we have all the component pieces, we are able to run Socrates Sim. To run the simulations, we first created a simulation configuration file (Figure 5.5). Socrates Sim dynamically loads the dialog domain, user simulator, and the dialog agent. As a result, changing from the restaurant domain to the movie domain just

requires invoking Socrates Sim with a new configuration file. In the configuration file, we update the domain_config, usersim_class, and agent_class to reflect the updated component locations.

```yaml
# Dialog Simulation Configuration

# Dialog Domain Settings
domain_config: sample_domains/movies/movie_domain.yaml
domain_kb_type: table
domain_kb_file_path: sample_domains/movies/moviekb.csv
domain_kb_file_type: csv

# User Simulator settings
usersim_class: sample_domains.movies.usersim.MovieUserSim
nlg_class: sample_domains.movies.usersim_nlg.UserSimNLGTemplate
starting_goal_path: sample_domains/movies/sample_starting_goals.yml
user_goal_type: template

# Dialog Agent settings
agent_class: sample_domains.movies.agent.MovieAgent

# Dialog Simulation settings
simulation_rounds: 3
max_turns: 8
first_speaker: usersim
reward: 10
simulation_output_path: data/simulated_dialogs/
save_history: True
save_location: "data/simulated_dialogs/"
```

Figure 5.5: Simulation configuration file.

Socrates Sim is invoked using the command-line and expects the path to the configuration file as an input. The figure below shows some example output for the sample dialogs that were generated by the simulator.

```
/Users/dhairya/Documents/envs/bin/python /Users/dhairya/Documents/socrates/src/run_simulator.py -p sample_domains/movies/simulation_config.yml -t yaml -pd
    Running simulation 1 of 1
     0 :  usersim :  Hi! Can you help book tickets for me?
          DialogAction:  greetings {}
     0 :  agent :  Which city do you want to see the movie in?
          DialogAction:  request {'city': None}
     1 :  usersim :  I'm looking for a theater in Boston
          DialogAction:  inform {'city': 'Boston'}
     1 :  agent :  Which date would you like to go?
          DialogAction:  request {'date': None}
     2 :  usersim :  Looking for movies playing on 06-10-2018
          DialogAction:  inform {'date': '06-10-2018'}
     2 :  agent :  Which movie would you like to see?
          DialogAction:  request {'movie': None}
     3 :  usersim :  I'd like to see A Quiet Place
          DialogAction:  inform {'movie': 'A Quiet Place'}
     3 :  agent :  Which theater do you want to see the movie at?
          DialogAction:  request {'theater': None}
     4 :  usersim :  I want to reserve tickets at AMC Assembly Row 12
          DialogAction:  inform {'theater': 'AMC Assembly Row 12'}
     4 :  agent :  How many tickets do you want to purchase?
          DialogAction:  request {'no_tickets': None}
     5 :  usersim :  I want to buy 2
          DialogAction:  inform {'no_tickets': '2'}
     5 :  agent :  Ready to purchase your tickets. Please provide your credit card, zip and expiration date. Respond with numbers in order and seperated by ';'. Eg. 4443444444444;02133;02/2020
          DialogAction:  request {'cc_number': None, 'cc_zip': None, 'cc_exp': None}
     6 :  usersim :  Sure. My credit card number is 4226273618199874. Expiration is 02/2023 and zip is 02138
          DialogAction:  inform {'cc_number': '4226273618199874', 'cc_zip': '02138', 'cc_exp': '02/2023'}
     6 :  agent :  Finished. You've purchased 2 for A Quiet Place playing AMC Assembly Row 12 on 06-10-2018. Your confirmation code is AAACSJ.
          DialogAction:  inform {'city': 'Boston', 'date': '06-10-2018', 'movie': 'A Quiet Place', 'theater': 'AMC Assembly Row 12', 'no_tickets': '2'}
     7 :  usersim :  Bye.
          DialogAction:  bye {}
     7 :  agent :  Thanks for using the Move Booking service!
          DialogAction:  bye {}
   Dialog Result:  Success
Successfully wrote dialog histories to data/simulated_dialogs/
```

```
Running simulation 3 of 3
     0 :  usersim :  Hi! Can you help book tickets for me?
          DialogAction:  greetings {}
     0 :  agent :  Which city do you want to see the movie in?
          DialogAction:  request {'city': None}
     1 :  usersim :  Boston
          DialogAction:  inform {'city': 'Boston'}
     1 :  agent :  Which date would you like to go?
          DialogAction:  request {'date': None}
     2 :  usersim :  Thinking 06-10-2018
          DialogAction:  inform {'date': '06-10-2018'}
     2 :  agent :  Which movie would you like to see?
          DialogAction:  request {'movie': None}
     3 :  usersim :  up for anything.
          DialogAction:  inform {'no_pref': 'UNK'}
     3 :  agent :  Which movie would you like to see?
          DialogAction:  request {'movie': None}
     4 :  usersim :  I don't care
          DialogAction:  inform {'no_pref': 'UNK'}
     4 :  agent :  Which movie would you like to see?
          DialogAction:  request {'movie': None}
     5 :  usersim :  up for anything.
          DialogAction:  inform {'no_pref': 'UNK'}
     5 :  agent :  Which movie would you like to see?
          DialogAction:  request {'movie': None}
     6 :  usersim :  whatever
          DialogAction:  inform {'no_pref': 'UNK'}
     6 :  agent :  Which movie would you like to see?
          DialogAction:  request {'movie': None}
     7 :  usersim :  I don't care
          DialogAction:  inform {'no_pref': 'UNK'}
     7 :  agent :  Which movie would you like to see?
          DialogAction:  request {'movie': None}
   Dialog Result:  Failed
   Successfully wrote dialog histories to data/simulated_dialogs/

```

Figure 5.6: Sample dialog generated by Socrates Sim.

Chapter 6: Development

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

6.1. Development Language

The framework was written in Python 3.6, which at the time of submission is the latest supported Python release. It is worth noting that the research implementation of the user simulator described in Li et al. (2016) was written in Python 2. Python 3 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 upgrades 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.

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

6.2. Development Tools

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

- yaml: The yaml library provides a set of functions 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 provide 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 knowledge base in-memory and execute various logical queries.
- OpenNMT: OpenNMT is Python library used for neural machine translation. OpenNMT was used to train the nlu and nlg models.
- memory-profiler: This library was used to measure the memory consumption of the framework.

The code base is stored on Github 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 features.

Chapter 7: Results

This chapter discusses the performance tests and results of the Socrates Simulator.

7.1. Experiment Overview and Setup

Socrates Simulator is novel as an open source end-to-end dialog simulation framework. As a result, it was challenging to develop a meaningful benchmark for evaluation. We ultimately chose the TC-Bot framework, which is the research implementation of end-to-end neural dialog framework described by Li et al. (2017). We measured two aspects of the Socrates simulator, its scalability and its memory consumption over the course of running multiple simulations. The goal was to show that Socrates Sim is generally usable and that overall performance does not significantly degrade with expanded usage across different domains and increased simulation runs.

TC-Bot is the academic inspiration for this project. TC-Bot shares similar objectives for the training and evaluation of task-completion dialog agents. However, there are several key distinctions between TC-Bot and Socrates Sim. First, the neural architecture for training the dialog agent is tightly coupled with the overall framework for TC-Bot. Second, TC-Bot was hardcoded for a limited movie booking use case. Finally, the training data used by TC-Bot was collected with Amazon Turk and not publicly released. They did provide some of the intermediate representations (Python

2.7 encoded pickle files) of the data (e.g., dialog acts, slot values, etc). However, these intermediate representations were optimized for their specific use case and were unusable for general usage. Adapting TC-Bot to a new domain would have required non-trivial work rewriting the neural framework, which is outside of the scope of this project. To test TC-Bot, we ran the framework as is, with no changes to the domain or underlying framework.

TC-Bot's performance relies on GPU support for its neural-architecture. In order to ensure a fair comparison, all performance tests were therefore run on a gpu enabled server. Specifically, the server was running Ubuntu 14.04, with a core i9 3.3 ghz 10 core cpu and 64 gb of RAM. Additionally, the server had two gpus, a Titan Pascal and a Titan XP 12 gb GPU.

A master script was written to coordinate iteratively calling each framework with a variable simulation count parameter. Each time, the framework is called as an isolated subprocess of the master script to ensure more accurate measurements.

7.2. Runtime Performance Experiment and Results

The goal of these tests was to measure how effectively the framework's runtime scaled with increased simulation rounds. Runtime is defined here as the elapsed time taken to run n simulations. Starting with one simulation, we iteratively increased the simulations by 500, until we ran a total of 50,000 simulated dialogs. We do not capture the write time to save the simulated dialogs to disk. This is usually a linear cost proportional to the size of the stored dialogs in memory.

In total, we ran six different performances tests. Four tests were run on the Socrates Sim framework and two on the TC-Bot framework.

For the first two tests, we compare the performance of end-to-end dialog simulations using a rules-based approach. For both frameworks, we use the movie booking

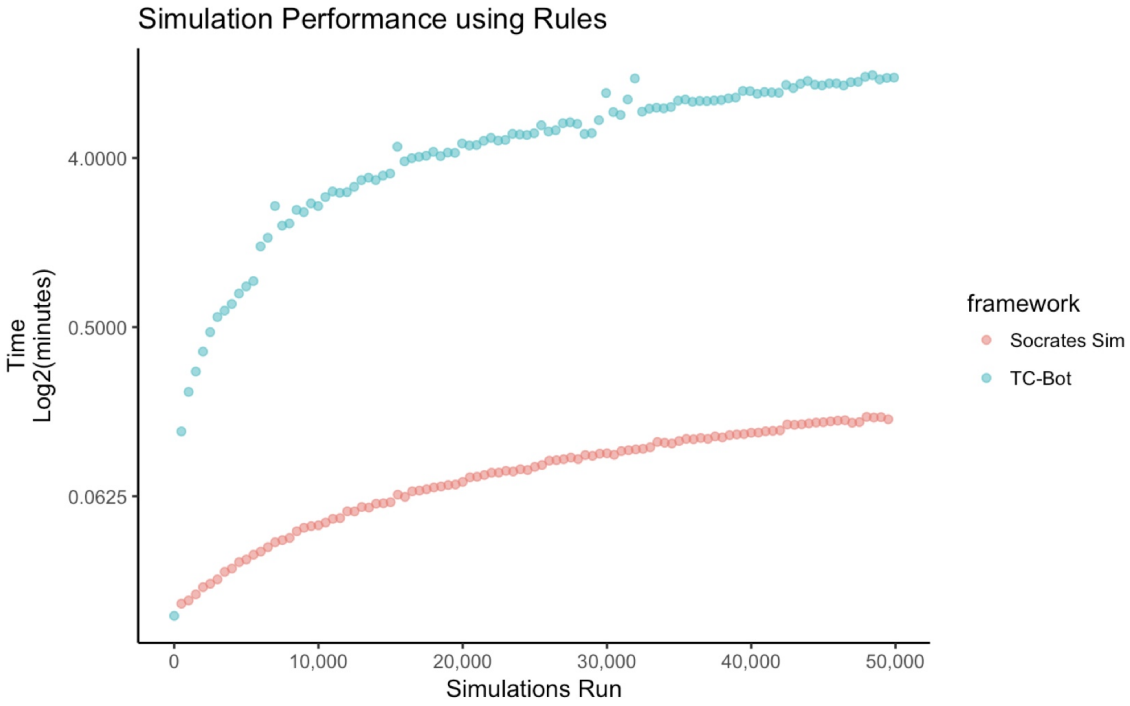

Figure 7.1: Runtime of Socrates Sim vs TC-Bot with rule based user simulator.

use case. In the Socrates Simulator, both the dialog agent and user simulator are rules-based speakers and have rules-based nlu and nlg modules. In TC-Bot, the user simulator is rules-based and the dialog agent is a DQNN neural model that is being trained on a reward signal after each simulation round. Both frameworks have a *O(N)* runtime as the simulation sizes grew, as shown in Figure 7.1. However, Socrates Sim performed overall more efficiently, taking on average about 1 minute and 51 seconds to run 50,000 simulations. In contrast, TC-Bot took on average 10 minutes and 44 seconds. A large part of this can be attributed to the fact that TC-Bot tightly couples training the dialog agent with the dialog simulation. That is, after each round, TC-Bot runs stochastic gradient descent on the reward function in order to inform the dialog agent's action in the proceeding round.

For the next two tests, we compare the performances of both frameworks using

model-based actors. As with the first two experiments, we are evaluating in the movie booking use case. For Socrates Sim, we initially use the OpenNMT trained nlu and nlg module for the user simulator and dialog agent. Since OpenNMT does not support programmatic access, our initial performance tests were very poor. It took Socrates Sim about 50 minutes to run 1,000 simulations. Each call to OpenNMT required launching a new subprocess, invoking OpenNMT anew, and running the prediction.

A preloaded model in memory would be more efficient. Therefore we switched to evaluating the performance of Socrates Sim using the Brownlee (2017) neural machine translations models. The benefit of this model is that it can be invoked programmatically. At the very initial load, neural models (serialized as hd5 files) are loaded into memory. Each call to the model involved a small preprocessing cost and an additional cost to do a forward pass through the network. On average, it took about 1 minute and 21 seconds to run 1,000 simulations. In contrast, we find that TC-Bot does perform much more efficiently. While its growth is still linear, it can run 1,000 simulations in about 30 seconds. For running neural models, TC-Bot is about 3 times faster.

Finally, our last set of experiments involve testing runtime performance across domains. As TC-Bot is hard-coded for the movie booking use case, it was not included in our performance tests. In Figure 7.3, we see that there is variation between domains. Both have linear growth, though the movie domain performs more efficiently, taking about 30 seconds to generate 50,000 simulations. What we find is that the overall performance of the framework is tied to the complexity of the domain and speaker classes. From a general usability point of view, we assert that framework scales predictably and is usable. End-to-end, it takes about two minutes to run 50,000 simulations for the restaurant domain, which is reasonable.

Growth is linear across all frameworks and domains. In the table below, we

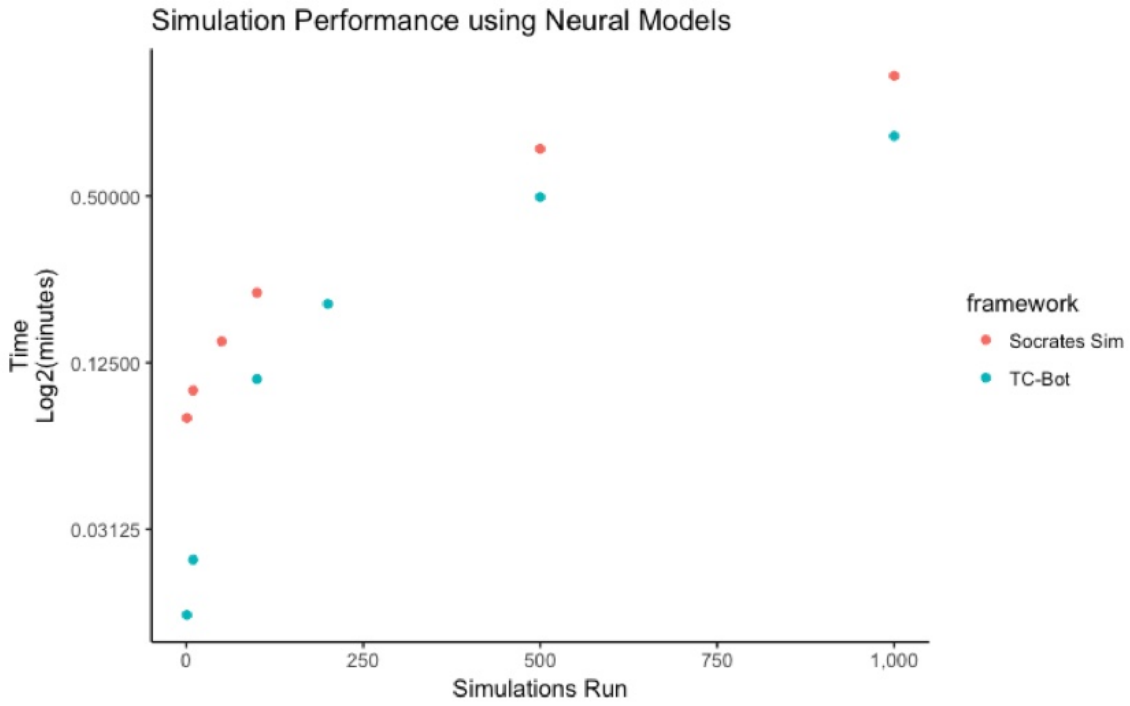

Figure 7.2: Runtime of Socrates Sim vs TC-Bot with model-based user simulator.

measure the average cost per simulation and rank all the framework/domain combinations from fastest to the slowest. The rules-based models for Socrates Sim significantly outperforms TC-Bot. In contrast, the model-based TC-Bot is about three times faster that Socrates Sim. Finally, rules-based approaches, in general, are more efficient to run.

There is an opportunity to further improve overall performance. As the simulations can run independently, we can introduce multi-threading and leverage the multi-core cpus for parallel processing. The dialog manager can be extended to support a thread pool and spawn multiple threads to run simulations in parallel. This could significantly improve performance if the researcher needs to generate hundreds of thousands or millions of simulations.

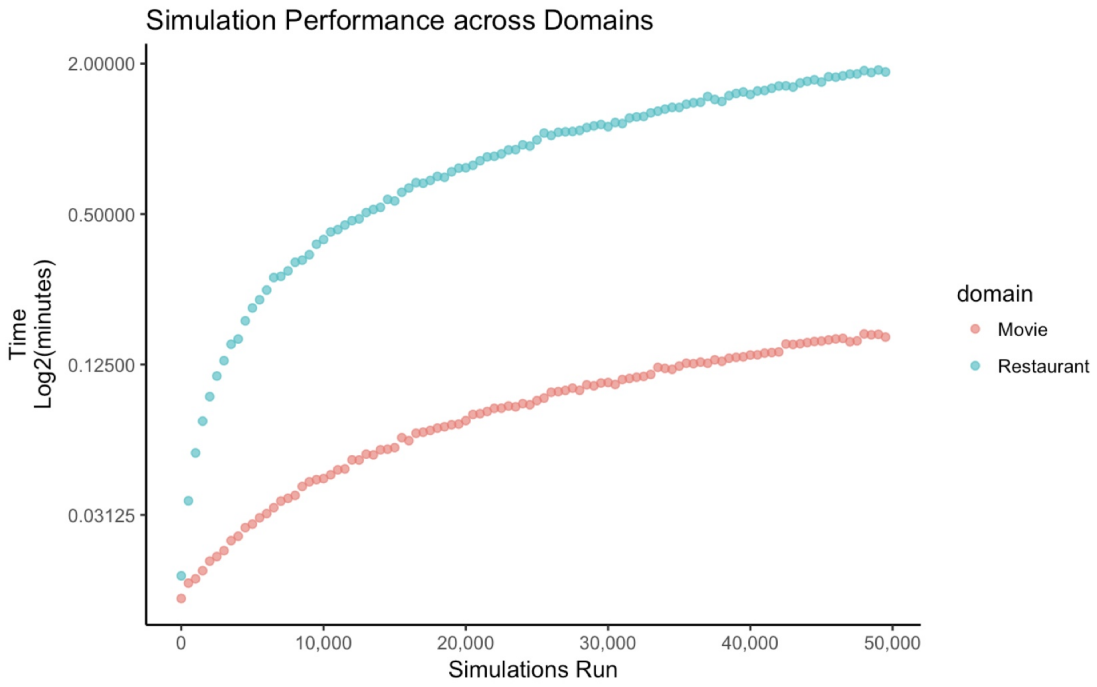

Figure 7.3: Runtime performance across domains.

Framework	Domain	Model Type	Avg Cost / simulation (seconds)
Socrates Sim	Movie	Rules	0.0002
Socrates Sim	Restaurant	Rules	0.0131
TC-Bot	Movie	Rules	0.0218
TC-Bot	Movie	Neural	0.2168
Socrates Sim	Restaurant	Neural	0.6722

Figure 7.4: Average time to run a single simulation.

7.3. Memory Consumption Experiment and Results

The next set of tests involve measuring maximum memory consumption of the framework when generating multiple dialog simulations. The goal is to show that Socrates Simulator consumes memory reasonably and is usable. Like the performance tests above, we ran a total of 6 tests to evaluate memory consumption. Four tests

were on the Socrates Sim framework and two on the TC-Bot framework.

To measure memory consumption, we used the Python based *memory-profiler* tool. The tool is invoked via command lined and runs the program to be evaluated as an internal sub-process. Over the duration of the observed program's runtime, *memory-profiler* uses the *psustil* tool to probe the operating system for information about CPU usage, running processes, and resource utilization. *Memory-profiler* will log memory usage at predefined increments until the program finishes running.

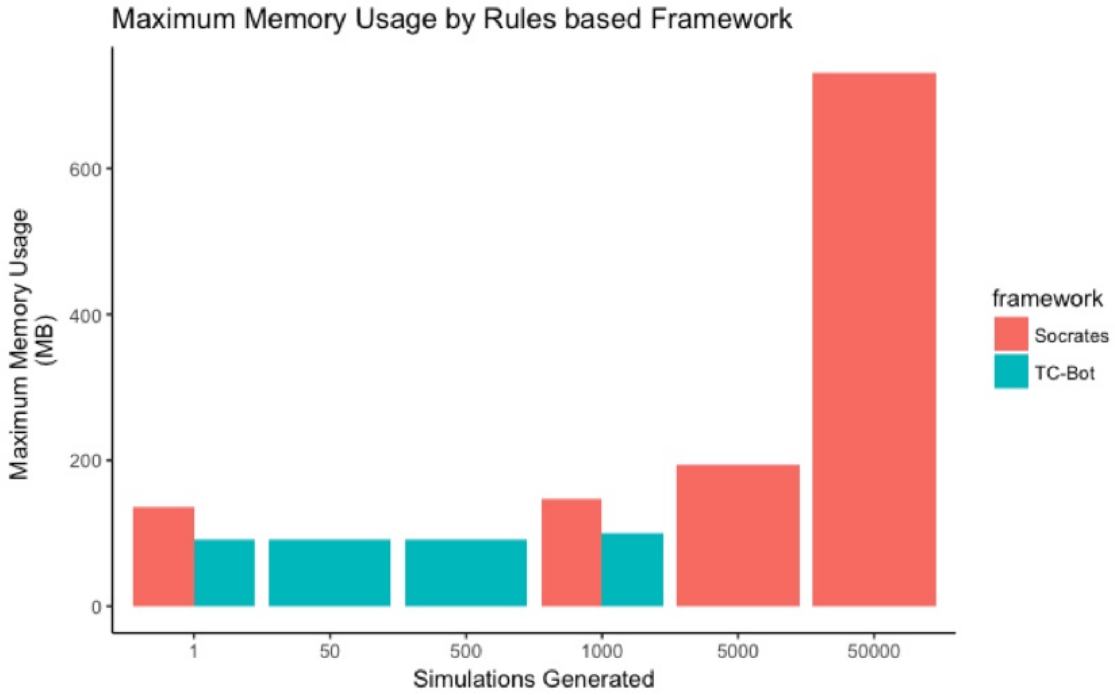

Figure 7.5: Memory usage of Socrates Sim vs TC-Bot with rule based user simulator.

For the first test, we looked at the maximum memory usage of both frameworks when using rule-based actors for the movie domain. Figure 7.5 shows relative performance of both frameworks. TC-Bot performs significantly better, capping at about 150 MB of overall memory usage over the course all simulations run. Running up to 5,000 simulations, Socrates Sim uses under 200 MB of memory. At the 50,000 simulations, memory usage does significantly increase to about 700 MB. It is

worth noting that this much memory is only utilized at the tail end of Socrates Sim's runtime. This likely due to the fact that the dialog manager has not serialized the generated dialogs and is holding them in memory until all simulations have run.

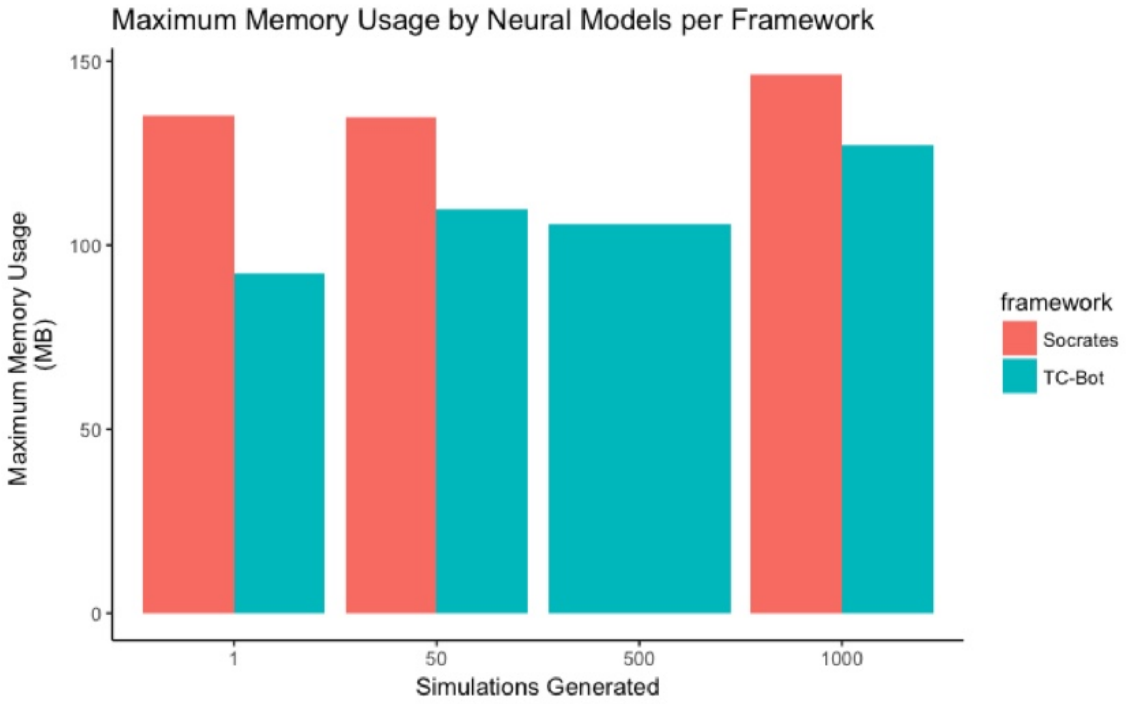

Figure 7.6: Memory usage of Socrates Sim vs TC-Bot with model-based user simulator.

In Figure 7.6, we see the results for memory consumption when using neural models. Note, that we used the restaurant domain for reasons explained above. Overall, both TC-Bot and Socrates Sim use a little under 150 MB. While runtime performance is drastically different, with TC-Bot running more efficiently overall, both frameworks have similar memory usage patterns.

Finally, in Figure 7.7, we see the results for memory usage across the restaurant and movie domains for Socrates Sim. Under 10,000 simulations, we found total memory usage caps around 250 MB.

All the tests reveal a small oversight in the Socrates Simulator design. The

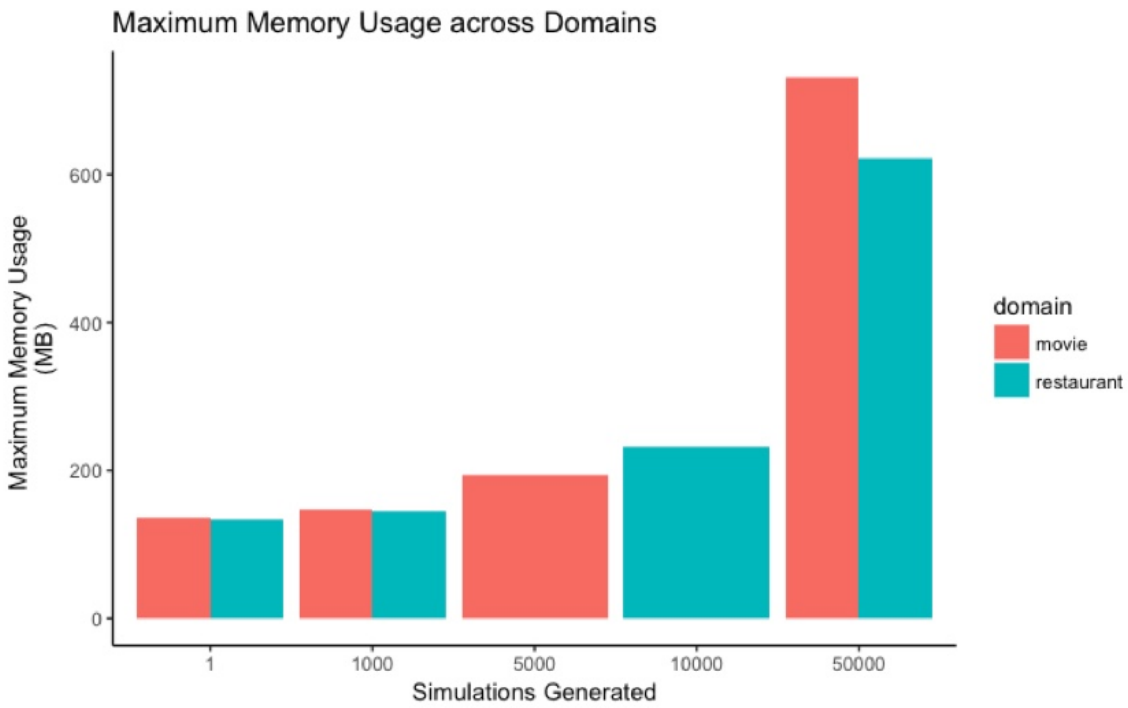

Figure 7.7: Memory usage across domains for Socrates Sim.

tests show that memory usage grows linearly with the number of simulations run. This is due to the fact the dialog manager does not serialize the generated dialogs until all simulations are run. This is a minor design choice, which can be updated to have the dialog manager periodically serialize the queue of generated dialogs and reset the queue.

Accounting for the serialization issue, the overall memory usage for Socrates Sim is reasonable. In small simulations (under 50,000), Socrates Sim has a similar memory usage profile to TC-Bot. We believe Socrates Simulator is usable and provides predictable performance and memory utilization. For rules-based use case, Socrates Sim is quite efficient. Accounting for the complexity of the rules, the researcher is able to run thousands of simulations in a matter of minutes. Overall, both the performance tests and memory utilization tests show that Socrates Simulator can

be used for most reasonable use cases.

Chapter 8: Conclusion

8.1. Known Limitations and Suggestions for Future Work

Overall, the framework works as intended. One key limitation was discovered during performance testing. We found that our serialization strategy causes the framework to consume memory linearly as the number of simulations increase. This occurs because the dialog manager waits until all simulations are run before dumping the generated dialog history. As a result, the in-memory json representation grows linearly with each incremental simulation run. The remedy for this is straightforward. The dialog manager can store the dialog history in a fixed size buffer that serializes to storage when it is full.

We also saw limitations around neural mode runtime performance and general qualitative performances in the context of nlg and nlu. The focus of this project was the engineering and development of the simulation framework rather than neural architectures to support particular use cases. This is an exciting and active area of research and offers many interesting avenues for further inquiry.

In a related vein, we were unable to support reinforcement learning and incorporate online training of the dialog agent in our framework. There is fascinating value to extend the framework to provide reward signals and support live training of the dialog agent based off of the feedback from the user simulator.

Finally, there is an opportunity to extend Socrates Sim by adding in a visual

interface. Currently, it is command line driven. The framework is developed with modularity in mind. We have abstracted the domain logic from the implementation layer. As a result, a visual front end can be developed for Socrates Sim without modifying the underlying framework.

8.2. Lessons Learned

This thesis was our first foray into task completion dialog research and statistical language understanding. While we ultimately approached this domain from an engineering perspective, we did gain a better understanding of the active research challenges and model-based approaches. Our understanding of the capabilities and limitations of neural network-based approaches was likewise enriched.

From a software engineering perspective, the greatest challenge we ran into was walking the fine line between defining rigid conventions to ensure predictable usage versus allowing for more flexibility to empower the user. Often Socrates Sim was written and rewritten to accommodate marginal edge cases because our design choices tilted too far on the flexible side. Other times, in an effort to reduce manual coding for the end user, we tilted too far in the opposite direction and defined very rigid conventions. Our end takeaway was that it often helps to take a step back and list out the various use cases. If you can make assumptions that captures 60% to 80% of those use cases, then developing defined conventions is very valuable. In fact, it reduces the friction and learning curve to understand and use your tools.

Additionally, this project was a great deep dive into using the Python language for a larger scale project. One of the persistent challenges with Python is due to dynamic typing. In large projects, it is hard to track bugs, as objects and variable are mutable and not strictly typed. The ability to explore Python 3.5's type hinting capabilities was useful and illuminating, especially since type hinting provides a way

to raise the quality of your Python code base to meet software engineering best practices and norms.

8.3. Summary

In summary, we have demonstrated the design and implementation of an end-to-end dialog simulation framework that supports task-completion dialog research. In chapter 2, we detailed a modular architecture that can be re-targeted to new domains and scaled efficiently. Central to this design were four key components. The first was the speaker abstract base class, which provides a unified interface for external user simulators and dialog agents to communicate with the framework in a standardized manner. The second was the codification of the dialog domain, where dialog acts, inform and request slots and slot values, and other pertinent information to the domain is captured. The dialog domain object is made available to the dialog manager, dialog agent, and user simulator. The third important component was the standardization of ancillary communication components like the dialog action object. Specifically, we extend and formally implement as classes the user goal and user agenda described in Schatzmann & Young (2009) to support the development of a user simulator. The final component is the dialog manager, which tracks, evaluates, and serializes simulated dialogs.

In chapter 3, we described the general process of implementing Socrates Sim. We highlighted our configuration-first philosophy, inspired by Gardner et al. (2018), which allowed for the framework to more easily generalize to new domains. Specifically, we called out the use of the simulation configuration file. The configuration file allows the researcher to integrate an external user simulator, dialog agent, and domain into the framework. We also demonstrated modularity by supporting the dynamic loading of the nlg and nlu modules.

In chapters 4 and 5 we explored in detail how Socrates Sim was adapted for the restaurant recommendation and movie booking domains. We detailed the development of dialog domain, dialog agent, and user simulators for both domains. We showed specifically how Socrates Sim was able to support new domains and nlg/nlu models with the use of configuration files and require no code update to the underlying framework and dialog manager. In doing so, we were able to demonstrate that the framework is retargetable.

In chapter 6, we described how Socrates Sim was developed. We highlighted the tools, programming language, and other development specific choices made while developing Socrates Sim. In chapter 7, we provided evidence that Socrates Sim is usable and scale efficiently. Multiple performance tests were run to evaluate the runtime efficiency and memory consumption of Socrates Sim as it ran an increasing number of simulations. We used TC-Bot Li et al. (2017) as a benchmark to evaluate performance.

The testing confirmed that Socrates Sim scales efficiently and predictably running up to 50,000 simulations. While Socrates Sim does scale linearly as the required number of simulations increase, performance does not degrade until you exceed 100,000 simulations. The average cost of running rules-based user simulators in Socrates Sim is between .0002 to .0131 seconds. You are able to simulate 50,000 dialogs in under two minutes, which far exceeds the performance of TC-Bot. We also find there is an opportunity for improvement in improving model-based simulations in Socrates Sim. Objectively, the average cost for a model-based simulation is .6722 seconds and it takes about one hour to run 50,000 simulations. In contrast, the average cost per simulation for TC-Bot's .2168 seconds. TC-Bot is able to run 50,00 simulations in about 30 seconds.

In conclusion, we have developed a framework that supports multi-domain

task completion research. We have demonstrated the framework's ability to support new domains without changing the underlying code. Additionally, our performance tests reveal that Socrates Sim scales efficiently and is generally usable. Our hope is this project can provide the foundations to support end-to-end task completion research and can be extended to provide new value by the community.

References

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

Brownlee, J. (2017). *Deep Learning for Natural Language Processing*.

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.

Guillaume Klein, Yoon Kim, Y. D. J. S. A. M. R. (2017). Opennmt: Open-source toolkit for neural machine translation.

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.