Kite: Building Conversational Bots from Mobile Apps

Toby Jia-Jun Li*
Carnegie Mellon University

Oriana Riva
Microsoft Research

ABSTRACT

Task-oriented chatbots allow users to carry out tasks (e.g., ordering a pizza) using natural language conversation. The widely-used *slot-filling* approach for building bots of this type requires significant hand-coding, which hinders scalability. Recently, neural network models have been shown to be capable of generating natural "chit-chat" conversations, but it is unclear whether they will ever work for task modeling. *Kite* is a practical system for bootstrapping task-oriented bots, leveraging both approaches above. Kite's key insight is that while bots encapsulate the logic of user tasks into conversational forms, existing apps encapsulate the logic of user tasks into graphical user interfaces. A developer demonstrates a task using a relevant app, and from the collected interaction traces Kite automatically derives a *task model*, a graph of actions and associated inputs representing possible task execution paths. A task model represents the logical backbone of a bot, on which Kite layers a question-answer interface generated using a hybrid rule-based and neural network approach. Using Kite, developers can automatically generate bot templates for many different tasks. In our evaluation, it extracted accurate task models from 25 popular Android apps spanning 15 tasks. Appropriate questions and high-quality answers were also generated. Our developer study suggests that developers, even without any bot developing experience, can successfully generate bot templates using Kite.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing systems and tools**; *Natural language interfaces*; *Systems and tools for interaction design*;

ACM Reference Format:
Toby Jia-Jun Li and Oriana Riva. 2018. Kite: Building Conversational Bots from Mobile Apps. In *MobiSys '18: The 16th Annual International Conference on Mobile Systems, Applications, and Services, June 10–15, 2018, Munich, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3210240.3210339

1 INTRODUCTION

The promise and excitement around conversational chatbots, or simply bots, has rapidly grown in recent years. Besides the popularity of intelligent assistants, we are witnessing the rise of specialized bots (also called skills in Alexa and actions in Google Assistant [14, 77]). Particularly useful are task-oriented bots that act as agents on behalf of users to interact with external services to accomplish specific tasks, such as booking a cab, making a restaurant reservation or ordering food, all using natural language conversation.

Today, most task-oriented bots are built using the *slot-filling* approach [12, 42]. The user's phrase (e.g.,"I want a coffee") indicates an *intent*, an action the system supports, such as "order-coffee (type, size)". Input parameters necessary for intent execution, such as coffee type and size, are known as *slots*. A *control structure* defines the steps for a multi-turn conversation between the system and the user to collect all slots necessary to *fill* the intent.

Slot-filling has proven reliable but requires significant developer effort. First, control structures, typically in the form of finite-state automata, are hand-designed for each task. These can be complex as they need to account for many possible execution paths. Second, to support user interactions in natural language, machine learning models for understanding user questions and answers need to be trained. Training requires a large number of *utterances*, i.e, sample phrases users may use in the conversation. Even to a simple question like "What is your party size?" users may answer in many different ways such as "3", "all of us", "me and my wife" or "not sure, I'll tell you later." (and more examples in Figure 7). To train robust models, a bot developer must consider all such possible phrase variations and provide dozens of utterances for *each* slot and for *each* intent. Finally, developers need to enumerate possible values for each slot to boost slot recognition in the language understanding model. As a result, this whole process requires significant manual encoding, thus hindering scalability to new tasks and domains.

An alternative to slot-filling is a corpus-based approach where bots are automatically trained from datasets of past conversations [23, 53, 54, 70, 71, 73, 78]. This approach has shown promise for non task-oriented "chit-chat" bots [11, 65, 79], where the sole purpose of the bot is to maintain a realistic conversation. It is unclear whether this approach alone can model also task-oriented bots. The purpose of task-oriented bots is to get all the needed information for completing a task through conversation with the user. Purely corpus-based machine-learned approaches cannot guarantee critical in-task constraints are met (e.g., a user cannot reserve a restaurant without specifying a time), and they lack a model to ensure completion of an actual objective in the task [78, 85]. These systems are also difficult to train due to the scarcity of domain-specific conversation logs.

To address the tension between the hand-coded but reliable approach of slot-filling, and the automated but non-goal nature of machine-learned bots, we developed *Kite*, a practical system for quickly bootstrapping task-oriented bots leveraging both approaches. Our key insight is that while task-oriented bots encapsulate the logic of user tasks into conversational forms, existing mobile applications encapsulate it into graphical user interfaces (GUIs). For instance, an app as well as a bot for ordering coffee

*Work done while at Microsoft Research.

MobiSys '18, June 10–15, 2018, Munich, Germany

ACM ISBN 978-1-4503-5720-3/18/06…$15.00
https://doi.org/10.1145/3210240.3210339

will guide the user through the same steps of picking a coffee type, selecting a size, adding toppings, etc. From interaction traces and GUIs of mobile apps, Kite automatically infers *task models*, defined as graphs of actions (i.e., intents) and associated inputs (i.e, slots) representing a task's possible execution paths.

Kite's goal is to enable developers to bootstrap bots for a *variety of tasks*, using *existing applications*, and in an *intuitive and easy-to-learn manner*. In Kite's new programming model, bot developers define a task by *demonstrating* how typical users may carry out that task using a relevant app – apps are unmodified and could include third-party apps they did not build. Kite logs app interactions and processes such traces to extract the corresponding task model.

A core contribution of Kite is a *precise and high-coverage task model extraction* approach. Kite uses the UI tree model to represent an app's interaction data in each screen, including UI elements, their relationships and the interactions performed on them. To infer task models from app interaction traces, Kite abstracts UI screens and events into intents and slots. This process is prone to false positives and false negatives. For example, a logged UI event (e.g., user entering text in a field) is likely to indicate a slot, but UI events may also be irrelevant (e.g., due to an unrelated popup) or duplicate (e.g., when selecting an item from a list, multiple events of different types are generated), thus leading to false positives (i.e., UI events incorrectly spawning a slot). Likewise, while navigating to a new page in the app is likely to indicate a new intent (e.g., transitioning to a page showing restaurant information can be associated with the GetRestaurantInfo intent), single pages may embed dynamic sub-pages through tabs or scrolling, which, if missed, cause false negatives. In §3.1, we describe how Kite is designed to reduce false positives and false negatives through aggregation of user traces and through a conservative slot and intent classification approach.

A task model represents the logical backbone of a bot. On top of it, Kite generates a *question-answer interface* for conversation. Kite seeks an automated approach so it cannot rely solely on linguistic rules nor crowdsourcing, as in related systems [10, 47, 83]. Due to the scarcity of domain-specific conversation datasets, it also cannot rely on a pure machine-learned approach [24, 69, 84, 86, 87]. Hence, Kite adopts a hybrid rule-based and neural network approach. (1) From an app GUI and on-screen contents it infers properties and semantics of slots, so to be able to generate prompt questions by means of few semantic rules. (2) To generate questions for slots that cannot be semantically classified, and to generate utterances for user answers, Kite trains neural network transduction models [76] using a total of 6M Twitter conversation pairs. This approach was able to produce at least one appropriate question for each slot, and many high-quality and varied utterances for user answers (§5.2).

Kite is the first system to automatically bootstrap task-oriented bots from mobile apps. Kite significantly reduces the manual work required by traditional slot-filling and language model training, which are the steps of the bot building process a regular developer is least likely to be familiar with. Overall, this work makes the following contributions:

- A set of techniques to extract task models from app interaction traces, which guarantee high recall and precision (§5.1).
- A careful application of state-of-the-art techniques to generate question-answer interfaces (§3.2).

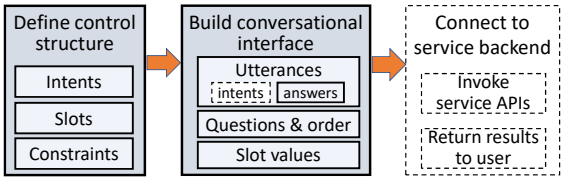

Figure 1: Bot building process. Kite addresses every step except those with dashed lines.

- A tool (Figure 6) for visualizing and editing task models as well as for iteratively testing their conversation flow, which was tested with real developers (§5.3).
- An evaluation with 25 popular Android apps from which Kite extracted high precision and high recall task models spanning 15 different types of task.

2 BACKGROUND AND MOTIVATION

In this section we motivate our work by illustrating how task-oriented bots are built today, and we outline how our system works and what it seeks to enable.

2.1 Bot building process

Task-oriented bots aim to understand a user request and execute the related task. Figure 1 shows the steps involved in building such bots and how Kite helps in this process. The first two steps are unique to bots and require knowledge and skills that a regular developer is least likely to have.

Step 1. A developer must define a **control structure** representing the kinds of action, termed *intents*, the system can support. Each intent is associated with a collection of *slots*, and may have *constraints* on other intents. For example, in a restaurant-booking bot, the intent ReserveRestaurant may have slots restaurant name, party size and time, and ConfirmReservation may have a dependency on it. Based on this structure, the bot does slot filling [42, 51, 80], i.e., collecting slots for each intent to then perform the associated action(s). The control structure is usually encoded in a finite-state automata that is hand-designed for each task by the developer. Kite helps developers automatically generate a similar control structure with *app-derived task models* (§3.1).

Step 2. A developer must define a **conversational interface** for the control structure, which can recognize user intents and slots, can query users for the required slots, and can understand user answers. For example, from a phrase "I want a table for 2" the system may recognize the intent ReserveRestaurant and the slot party-size=2, and query the user for the missing slot time. In earliest systems (e.g., [81]) this step was accomplished using semantic grammars consisting of thousands of hand-written rules. Today, developers can train machine learning models for language understanding using any of the available tools (e.g., [7, 22, 25, 41, 61]).

Figure 2 illustrates the model building process using an example with Dialogflow [22]. First, the developer supplies user utterances that map to each intent in the task (left side of the figure). Second, the developer defines slots for each intent, tags any slot value appearing in the provided utterances, provides prompt questions for each slot, and specifies the order of prompts (center of the figure).

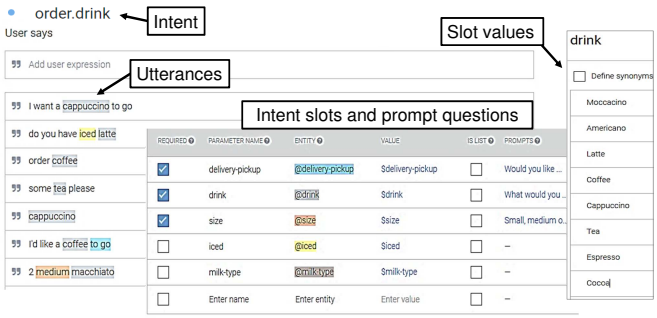

Figure 2: Building a conversation interface with Dialogflow [22]. In Dialogflow slots are called "entities".

Third (not shown), the developer repeats the same procedure to parse user answers, i.e., specifying phrases for possible user answers and tagging slot values. Finally, to facilitate slot recognition from user phrases, possible values for each slot need to provided (right side of the figure). After specifying dozens of utterance samples for each intent and for each slot (at least 30 in the Alexa guidelines [6]) and after listing slot values, a model can be trained, which should provide reasonable accuracy on inputs that *are similar* to the examples.

In reality, it is hard to train a robust language understanding model mainly because different phrasings may be missing from the provided utterances, as it is hard for a developer to enumerate them all. Kite helps developers by automatically generating utterances for answer recognition using open-domain conversational datasets (e.g., Twitter) (§3.2) and by extracting slot values from mobile apps. Kite does not yet provide utterances for intent recognition.

Step 3. The last step involves mapping intents to actual APIs and invoking a **service backend** to retrieve results. In the case of an app developer, the bot's service backend can be the app's backend. The developer needs to expose APIs for the new bot client. This step is out of scope for Kite.

2.2 Goals and overview

Bot scalability. The large amount of hand-coding and the natural language processing (NLP) skills required for building a bot partially explain why while the number of bots available on popular platforms such as Alexa, Messenger, Slack, Kik, etc. is constantly increasing [14, 75, 77], it is still tiny. For example, as of April 2018, the Google play store has over 3.2 million apps and Alexa has 31,300 skills, which is roughly the number of apps added to the play store in 2-3 weeks. Most mobile apps, even popular ones, have no bot counterpart today. From the 320 most popular Android apps (in the weather, food, entertainment, news, shopping, music, education, travel, navigation, and productivity categories) we found that only 28 (8.8%) have an Alexa skill and 14 (4.4%) have a bot in Slack. Moreover, such bots usually support a very small subset of the app's functionality [56]. For example, Starbucks has a skill for re-ordering what a user previously ordered (but a user cannot order new items). OpenTable has a skill to book a specific restaurant, but it does not support searching for new restaurants. Certainly, it is not expected that apps and bots will have identical functionality, but arguably enough existing bots are limited [34, 35, 66].

Goals. This work explores how we can reduce the developer effort required for generating task-oriented bots. Our goal is to offer a system that allows a developer to automatically generate a *bot template* specifying the intents associated with the task, their dependencies and their slots, where each slot comes with possible values, prompt questions and a set of possible user answers. We strive for a system that meets the following goals: (1) *Task coverage*: a developer should be able to use Kite to create bot templates for a variety of tasks; (2) *App flexibility*: most existing apps should be amenable to task extraction; and (3) *Low-learning barrier*: a developer should find Kite easy to use and collecting interaction traces for task model generation should be natural and intuitive.

It is *not* our goal to automatically generate a complete ready-to-use bot. The developer is expected to manually revise the generated bot templates, correct errors (e.g., missing slots), improve its natural conversation quality (e.g., selecting prompt questions other than the defaults), and use the generated user answers to train language understanding models. More importantly, in order to function, a bot needs to parse user intents and be connected to a service backend.

Kite in action. Developers use Kite by iterating through three steps: *(i) demonstration*, *(ii) template authoring*, and *(iii) conversation testing*. We illustrate this process through the example of a developer who wants to build a restaurant reservation bot and uses OpenTable for demonstration.

The first step is to demonstrate how typical users may carry out the task with the app. To obtain as complete as possible a task model, the developer can demonstrate completing the task through multiple different paths, such as selecting a restaurant from a recommended list versus searching by location. The developer can also demonstrate "sub-tasks" such as viewing restaurant reviews or menus to have them reflected in the task model. We recommend demonstrating the task about five times depending on task complexity. This programming-by-demonstration approach imposes low learning barriers for the developers, since they only need to use the app through its GUI as a regular user [18]. Prior research has also shown that even non-developers can successfully demonstrate common tasks on mobile apps [55].

After the demonstration, the developer imports the collected interaction traces into Kite (Figure 6), which, after processing, displays an interactive visualization of the computed task model and a question-answer interface. The developer can revise it, e.g., by ensuring intents and slots are correct, by selecting the most appropriate prompt from the given list, etc., and then interactively test the bot preview (Figure 6 right). Once done, the template could be further exported to one of the existing bot platforms (e.g., [5, 36, 60]).

Target developers. Developers using Kite may bootstrap bots which are similar in functionality to apps that they may *have* or may *have not* implemented. For the first class of developers, Kite could work as a *conversion tool* that can automate operations like enumerating intents, slots, slot values, and more importantly, questions and answers for the conversational interface. For the second class of developers who did *not* build the app, Kite can be used as an *exploration tool* that extracts the app's business logic and represents it into a bot format. This could be the case of a bot development

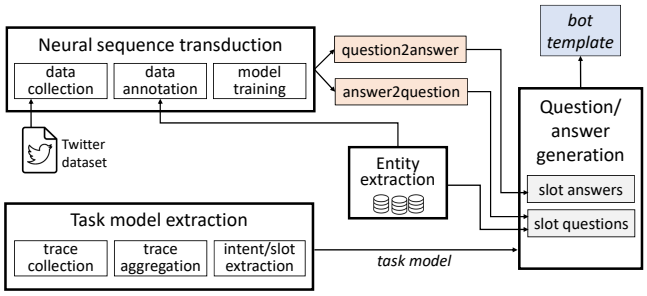

Figure 3: Kite system architecture.

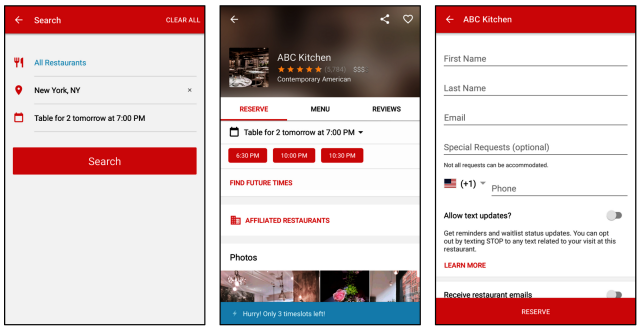

Figure 4: Example user interaction with the OpenTable Android app.

company (e.g., [1]) which builds bots on behalf of app developers or that creates personal assistants spanning multiple app domains. For example, imagine a developer designing an assistant for "planning movie-nights" where users can buy movie tickets, search for online movies, explore dining options, arrange transportation, and so forth. Kite makes it easier to use existing movie, food or cab-booking apps to explore the bot design. The bot developers who participated in our study confirmed these use cases (more in 5.3).

3 SYSTEM DESIGN

We describe how Kite was designed to meet the aforementioned goal of creating bot templates for *many* different tasks, using *existing apps*, and in an *easy-to-learn* manner. Figure 3 shows the system architecture. The two main modules are (i) task model extraction and (ii) question/answer generation.

3.1 Task model extraction

Kite's key insight is that from the interaction traces collected when performing a task with an app, it is possible to infer the logical structure of that task, known as *task model*. A task model encodes the sequence of intents and slots necessary to complete the task, as well as the dependencies between intents and slots (e.g., one cannot confirm a restaurant reservation without specifying the number of people and the time).

Before describing in details task models and how Kite constructs them, we give an intuition for why they can be extracted from mobile apps and what challenges it entails.

3.1.1 Task models in mobile apps. Compared to desktop applications, due to the phone screen size, mobile apps tend to display less content organized in separate pages that a user can interact with and navigate between. A page displays content according to its UI layout. It contains UI elements (buttons, textboxes, lists, etc.), which are organized in a UI tree (like a DOM tree), where nodes are UI elements and a child element is contained within its parent. Some UI elements (e.g., buttons) are interactable as opposed to those that just display content (e.g., text labels).

The organization of an app's content into small, distinct pages makes it easier to abstract UI interactions into intents and slots. Figure 4 shows three screens from a user interaction with OpenTable where the user searches for a restaurant (screen a), selects one from a list (not shown), views information about it (screen b), and makes a reservation (screen c). Here one can recognize intents and slots: StartRestaurantSearch (screen a), ViewRestaurantInfo, ViewRestaurantMenu and ViewRestaurantReviews (screen b) which all take as slot the restaurant name (selected in the restaurant list screen), and ReserveRestaurant (screen c) which takes as slot the reservation time (selected in screen b).

To extract intents and slots programmatically, Kite needs to translate the "app language" into the "bot language". In the app language, a user executes a task by interacting with *UI elements* and transitioning from one *page* to another. In the bot language, a user executes a task by filling *slots* and navigating a graph of *intents*. There is a correspondence between app pages and intents, and between UI elements and slots. However, this mapping is not obvious. In contrast, the translation process is prone to false negatives and false positives. Each app page could map to *one* intent, but it could also map to multiple intents (as in Figure 4b where the page corresponds to at least three different intents). Likewise, interactable UI elements, such as list items or buttons, could each map to one slot or to none (e.g., the Search button in screen a is not a slot). In general, the regular structure of distinct and nicely labeled UI elements and pages as it appears in an app's screens is not what an automated program observes at the application framework level, where (1) UI notifications may be duplicate, missing or empty, (2) UI labels appearing on the screen may not be present in the triggered UI events, and (3) UI layouts may dynamically change depending on content and user actions. Next, we describe how Kite deals with these challenges.

3.1.2 Trace collection. To extract information from an app, we can adopt (1) static analysis, where we examine the source code of the app without executing it, or (2) dynamic analysis, where we analyze the app by executing it. While static analysis can extract the layout specification of screens in the app [3, 68], it cannot access most of actual contents within the app, since they are usually fetched or generated at runtime. As the app's contents are essential to infer the app's task model, we opt for dynamic analysis. Dynamic analysis can be performed using a UI automation tool that automatically interacts with the app and navigates to various pages in the app (e.g., [39, 58]), but this approach would fail to explore an app in a way that mirrors human usage. Hence, we rely on human traces currently provided by bot developers. This is little effort:

5–6 traces per task which can be collected in few minutes. In the future, Kite could support user traces produced via crowdsourcing (e.g., [19, 21]) or logs collected by app developers for analytics purposes (e.g., [29, 37]).

Kite represents an interaction trace as a sequence of UI events. Each UI event is associated with the UI element on which the action was performed (e.g., clicking on a button or entering text in a text field) and with the entire UI tree including all UI elements on the screen at the moment of the interaction with their hierarchical relationships, types and contents. One way to capture UI events is to leverage accessibility services [26, 55]. However, accessibility services tend to miss UI events that are essential to Kite. For instance, the Android accessibility API provides clicks and text view changes events, but it does not provide touch events nor notifications for page transitions (i.e., startActivity() and stopActivity() notifications). Hence, Kite modifies the application framework of the OS to complement the accessibility service (more details in §4).

3.1.3 Trace processing. Kite processes the application traces provided by a developer to extract a *task model*. We define a task model as a directed graph $T = (I, D)$ comprising a set I of intents together with a set D of intent dependencies. Each graph has a start node (StartOfConversation) and an end node (EndOfConversation). Every path in the graph, from start to end node, represents a possible execution of the task. Each intent $i \in I$ contains a unique identifier, a descriptive name for the intent (e.g., "Confirm Reservation"), a list of sample user utterances for invoking the intent and an ordered list of slots (which can be empty). Each slot contains a unique identifier, a name, a probability indicating whether the slot is mandatory ($p = 1$ when the slot is mandatory), a list of observed possible slot values, a list of prompts for querying the user about the slot, and a list of sample user answers for the slot. As an example, Figure 6 gives a visualization of the task model extracted from OpenTable.

Task model extraction is a 3-step process consisting of trace aggregation, intent extraction and slot extraction. We describe each step next.

Trace aggregation. We expect a developer to demonstrate a task multiple times. Multiple traces may show completing the same task in different ways, or in the same way with different inputs. The first type of redundancy is necessary to ensure model completeness, as one trace is unlikely to capture all possible execution paths. The second type of redundancy helps with model accuracy, as we explain later. Kite first aggregates all traces from the same app at the page granularity, using the page's class name. Then, it aggregates UI events in each page. Developers do not always assign identifiers to their UI elements or, if they do, they often assign non-unique identifiers, so Kite uses a combination of factors in determining whether two target UI elements from two different UI events are the same (i.e., class name, view id, location on screen, text labels, and position in the UI tree hierarchy). During UI event aggregation, all text labels associated with individual UI elements are merged, e.g., all strings entered in a text field in separate interactions become "possible values" for the same UI event.

Intent extraction. As said above, an intuitive way of identifying intents is to map each page transition to an intent. For instance, in Figure 4, transitioning from screen b (RestaurantProfileActivity) to screen c (ConfirmReservation) may be recognized as the ReserveRestaurant intent. This approach may produce mostly correct results, but can also cause false negatives because an app page may embed multiple actions accessible by scrolling, swiping, clicking on a button or opening a menu. In Android, embedding in a single page multiple dynamic sub-pages (e.g., Fragments), such as the three tabs shown in Figure 4b, has become common practice. There are also actions that can be taken without leaving the current page (e.g., sorting results in a list). We call such multiple intents embedded within a page **sub-intents**.

To extract sub-intents from sub-pages, one option is to track sub-pages by instrumenting the application framework, but sub-page support libraries are often distributed externally or developer customized. For instance, in Android, Fragment libraries (e.g., android.support.v4 or v7 [8]) are included directly by the developer in the application package. Relying on app instrumentation is not scalable, so we take a more general approach and detect sub-intents directly from UI trees.

Our intuition is that sub-intents often hide behind UI elements whose immutable text labels indicate distinct *actions* (e.g., Menu, Reserve, Sort by price, etc.), and that once invoked the contents in the screen change significantly. Kite classifies UI elements present in the aggregated traces as sub-intents if (1) their text labels are immutable (i.e., the label is the same in all traces), (2) their labels contain verb phrases, and (3) if once interacted with they cause the UI tree and its contents to change significantly, *without* triggering a page transition. For the last condition, Kite constantly compares the current UI tree with the previously captured one in terms of size, number of images, number of text elements, and content changes in text elements. Such a strict set of conditions reduces the chances of false positives. On the other hand, because Kite performs such evaluation on aggregated traces, false negatives are also reduced. For example, during trace aggregation, the click events associated with the three tabs in Figure 4b are aggregated under the same UI element; the three conditions above are verified so they are correctly recognized as sub-intents. If those three UI events were processed individually, only the Reserve tab would be classified as a sub-intent (because the labels "Menu" and "Reviews" are not verbs).

Tracking page transitions coupled with sub-intent detection produced good results for intent extraction in our task dataset (§5.1). Intent names are currently derived from class names of the pages. For better labels, an alternative approach is to process textual contents appearing on screens.

Slot extraction. Most UI events captured in an interaction trace do *not* correspond to slots thus possibly causing false positives. Kite performs aggressive data cleaning. Kite discards UI events of the following types: (1) UI events associated with immutable content, such as buttons with static labels, that when interacted with trigger a transition to a new page or a dialog; (2) UI events associated with invisible UI elements, which can occur when the app overlays multiple layouts on top of each other – a user interacts with the visible UI element but the UI event may be reported (also) by the underlying layout which is not visible to the user. (3) UI events associated with empty content, which often occur when selecting an item in a list or a menu – this interaction fires two UI events, one associated with the parent (the list/menu container) and one with

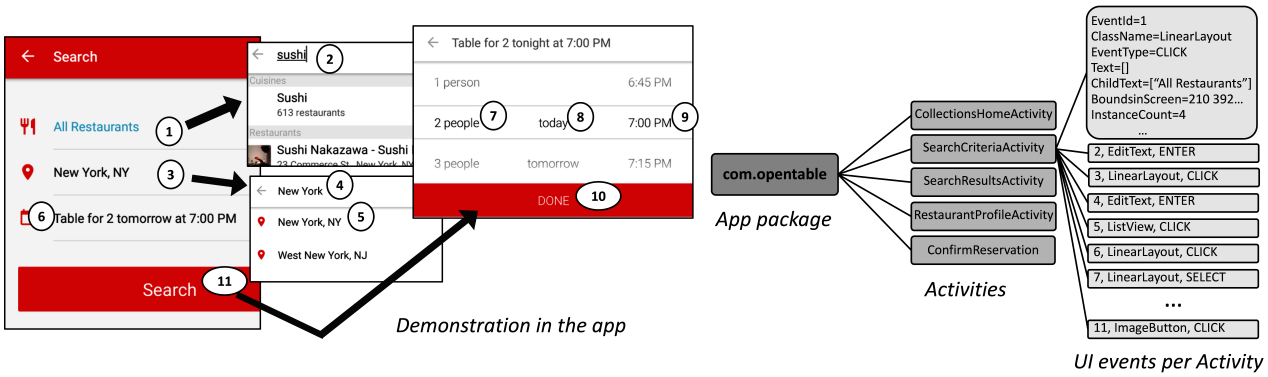

Figure 5: A simplified illustration of how UI traces are represented, using OpenTable as an example. UI events listed for SearchCriteriaActivity (with 1–11 identifiers) are drawn on the app screens shown on the left side.

the child (the actual item), but while the child contains relevant content (i.e., the selected label) the parent is empty.

To clarify, Figure 5 shows all UI events reported during a user interaction with SearchCriteriaActivity in OpenTable. Note that some UI events occur in separate dialogs, but are associated with the same page, for a total of 11 events. Out of these, only 3 events (with identifier 2, 5 and 6) are relevant to slot extraction. Events 1, 3, 10 and 11 are eliminated because of condition (1) above, events 7, 8 and 9 because of condition (2), and event 4 because of condition (3).

After data cleaning, Kite extracts slots from the remaining UI elements, and assigns them to the appropriate intent. Slots extracted from a page A are assigned to the intent extracted from the transition *from A* to a subsequent page. Slot names are extracted from accessible text labels, on-screen text labels or developer-specified identifiers of the UI element; if none of these is available or the extracted name is meaningless (e.g., button_dtp), Kite names slots with their entity type computed based on all the slot values (as described in §3.2). For example, a slot with possible values "New York" and "Miami" is labeled as "city". The values of a slot are all texts collected from the corresponding UI events (e.g., labels appearing on interacted buttons, strings entered in text fields, etc.). In the case of list-like containers, Kite extracts all items contained in the list, regardless of which were actually clicked. The order of slots is based on the order in which UI events were logged in the traces. In this way, if there is a predominant order in which users enter inputs in the app, this order will be preserved in the task model. Slot probabilities for optional slots are computed based on the number of observations where each slot is filled in the traces.

3.1.4 Example of extracted task model. Figure 6 shows the restaurant reservation model extracted from the OpenTable app, prior to developer editing. Each node in the graph represents an intent. The flow starts from StartOfConversation and ends in EndOfConversation. One can find restaurants in two ways: by viewing restaurant suggestions such as "nearby restaurants" or "outdoor sitting" (through the SearchResults intent) or by doing a custom search (through SearchCriteria and SearchResults_2[1]). For each restaurant in the results, one can retrieve profile, reservation options, menu and reviews (using the respective intents), and then proceed to make a reservation (ConfirmReservation) and submit it (ConfirmReservation_done). Slots are represented as shown at the center of the figure. A custom search (SearchResults_2) requires three slots: search query, city, and party size and time. Each slot has four fields: name, identifier, values, prompt question, and utterances for possible answers; the last 2 fields are generated as described next.

Task models of this type encapsulate the logic behind a chatbot. During execution the chatbot tries to fill the slots for a target intent by asking the prompt questions associated with each of its slots. If any of the slots are optional, the bot first asks whether the user wishes to provide that slot. After completing filling slots for an intent, if the intent has multiple possible next intents, the bot asks the user which intent she wants to execute. Otherwise, the bot directly goes to the next intent and starts filling its slots.

3.2 Question and answer generation

The previous step produces the logical flow of a bot. To navigate it using natural language, a bot needs to be able to ask questions and understand answers. Hence, Kite needs to deal with: (1) *question generation*: identifying *one* appropriate prompt question for every slot, and (2) *answer generation*: generating *large* sets of possible user answers for each prompt question so that an answer understanding model can be trained.

Automatically generating a question given unstructured text is a well-studied problem. Current approaches include rule-based techniques [4, 43, 62, 67], which are reliable but require manually-designed rules for declarative-to-interrogative sentence transformation, and corpus-based techniques [24, 69, 84, 86, 87], which are automated but are constrained by the scarcity of domain-specific conversation datasets. Either because of the extensive manual work or because of the extensive corpus of data, Kite cannot directly re-use these techniques without violating its scalability goal.

Moreover, Kite's question generation problem is different from the state-of-the-art. First, in Kite the inputs to question generation are slots, each consisting of a name, some possible values, and the UI event from which the slot was spawned. Hence, Kite' textual inputs are *not* complete sentences, but instead brief strings without

[1]The naming SearchResults_2 is to distinguish two intents both extracted from transitions to the same page (SearchResultsActivity) with different functionality: restaurant suggestions and custom search.

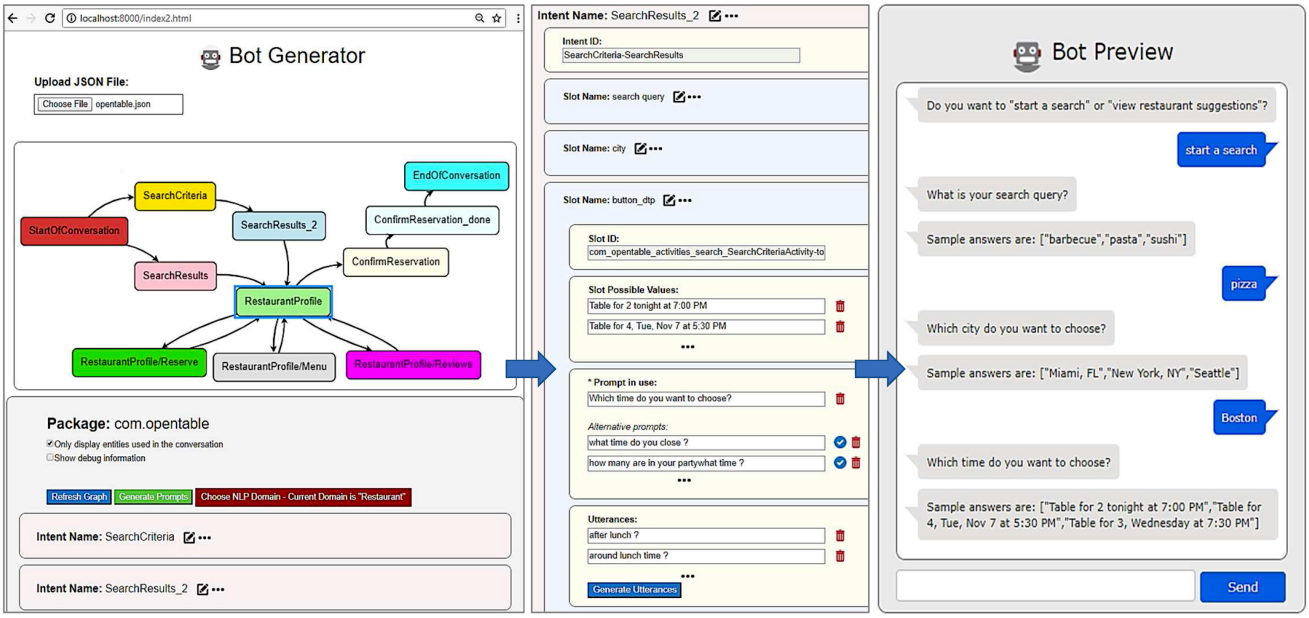

Figure 6: Kite web app for visualizing, editing, previewing, and interactively testing the bot template.

context (e.g.,"Table for 2 today at 7:00PM"). Second, Kite needs to generate a question for which all (or most) slot values represent acceptable answers. Hence, it needs to generate "non-factoid" questions. In contrast, state-of-the-art techniques usually take as input full sentences and produce factoid questions. For example, given the input "The dog is asleep on his bed" one such system would output the question "On what is the dog asleep?" [43]. In Kite, given a slot value "Table for 2 today at 7PM" we do *not* want outputs like "When was the table for 2?" or "What was at 7PM?", but instead something like "For how many people and at what time?".

To solve this problem, Kite takes a hybrid rule-based and neural network approach, exemplified in Table 1. Kite generates *wh*-questions using a few simple semantics rules based on the following elements: slot name, entity type (computed based on all slot values), and whether the slot values contain user-generated or application content. To classify the content, Kite uses the type of UI event from which the slot was spawned. For instance, a UI event of type "text-entered" is classified as user-generated, whereas a "text-click" is classified as application content. This distinction helps formulate the questions as "What is your ..." or "Do you want to select ...", respectively. The type of UI element (e.g., checkbox, list, button, etc.) could also help formulate the questions.

However, many UI elements in most apps lack descriptive text labels and so generate non-meaningful slot names, and entity extraction can fail on many fields (as in the first and third row in Table 1). This is why Kite also relies on neural network models. Recent work has attempted to use neural networks to solve general sequence-to-sequence learning problems in NLP, such as speech recognition and machine translation [44, 74]. As question answering can also be seen as mapping a sequence of words representing the question to a sequence of words representing the answer [78] (and vice versa), we train **neural sequence transduction models** using a question-response dataset of Twitter conversations. As described in detail in §4, we train two types of model: *answer2question (A2Q)* and *question2answer (Q2A)*. Figure 7 shows the top 10 questions generated for the slot with value "table for 2 tonight at 6:30PM" and the answers produced if one of the produced questions is selected. In this example, but also in general, Kite generated more relevant answers than questions (§5.2). This is an acceptable result because for a bot developer it is enough to obtain *one* good prompt question, and it is more important to obtain many diversified answers for training answer understanding models.

4 IMPLEMENTATION

App trace collection. Kite currently supports task extraction from Android apps. We modified the application framework of Android 6.0 and wrote an app for logging UI events during task demonstration. The app registers as an accessibility service listening to VIEW_CLICKED, VIEW_LONG_CLICKED and VIEW_TEXT_CHANGED events in the background. We modified the application framework to capture touch events, StartActivity and StopActivity lifecycle events, and UI tree of the current screen. By doing this, for each type of UI event listed above, the app could log information about the source UI element (e.g., text labels, accessible labels, identifier and screen locations) and the current UI tree (including all UI elements on the screen at the time of the event). A developer can either install our custom OS on their Android smartphones or run our Android X86 image in a VM hosted locally or in the cloud (we currently use Microsoft Azure VMs). A third option, that we have not yet implemented, is to make our Android VMs accessible from a web browser using existing technology (e.g., [19, 21]).

Server and web app. The rest of Kite is Android agnostic. We implemented a Java application for processing interaction traces, and a web app in AngularJS to edit bot templates and preview the

Table 1: Questions generated for the three slots in Figure 6 center (prior to developer editing).

Slot name	Slot values	Entity	User-gen	Rule-based question	Neural network model question
search query	barbecue, pasta, ...	NULL	True	***What* is your [search query]?**	What did you have?
id_list	New York, Seattle, ...	city	False	***Which* [city] do you want to choose?**	Which location?
button_dtp	Table for 2 today at 7PM, ...	time	False	*Which* [time] do you want to choose?	**How many are in your partywhat time ?**

A2Q questions for "table for 2 tonight at 6:30PM"
y all got a table for two in ten minutes?
how about now?
are you still planning to come by 15th and m today?
what time do you close?
what time are you planning on coming in?
how many are you and what time are you coming though?
how many people are you fitting in there for that?
how many people are in line?
how many are you and what time are you planning on coming in? ✓
what time are you getting there?
O2A answers for "how many are you and what time are you planning on coming in?"
6:00 pm, party of 2
how about 11?
probably around 9!
just 2
hopefully around 9.
2:45
4 people at 6pm on sat ?
not sure, but we ll keep you updated throughout the day
4 people at 6pm on saturday
4:00 pm, see you soon!

Figure 7: Example of questions and answers produced by the A2Q and Q2A models (Output limited to the top 10 results).

conversation flow, thus supporting iterative design. GoJS [72] is used for visualizing task graphs.

Entity extraction. Kite tags slots with entity types by using state-of-the-art entity extraction techniques. Given an input string (i.e., a slot value), (1) Kite determines its type by verifying whether it matches the name of an entry in two knowledge bases: OpenStreeMap for cities, addresses, buildings and various types of POIs (restaurants, schools, etc.) and a very large proprietary knowledge base for book titles, movie names, dish types, food ingredients, meal types, beverages, and few others; (2) it uses Stanford NER [27] for recognizing entities including persons, organizations, money, percents, dates and times; and (3) it classifies common structured data (e.g., URLs, email addresses, phone numbers) using regular expressions. If a slot has inconsistent entity types across its possible values, a NULL type is returned.

Neural sequence transduction models. For training we used the recently-released Transformer [76] in TensorFlow [2]. Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output, without using recurrent neural networks (RNNs) or convolution. Before adopting Transformer we tested other models, and found that Transformer models outperformed RNN models with long short-term memory (LSTM) [40] and gated recurrent units (GRUs) [16] in terms of loss, BLEU [63] score, training time and human judgment on the output quality, which confirms the findings in [76].

Table 2: Dataset sizes and BLEU [63] scores for Q2A and A2Q models. 10 models in total.

Domain	# of pairs	A2Q BLEU	Q2A BLEU
Restaurants	381, 221	**1.00**	0.33
Movies	54, 423	0.92	**0.69**
Libraries	23, 103	0.35	0.24
Coffee-shops	105, 723	0.56	0.16
Multi-domain	5, 443, 096	**1.62**	**0.74**

We collected 5 training datasets of 2-turn Twitter conversations from 2012 to 2016 filtered in the following way. Initially, we produced one multi-domain dataset of over 5M conversations filtered by keywords related to specific entities (food, recipes, movies, etc.) and tasks (booking, cooking, etc.). Then, for 4 domains (restaurants, movies, libraries and coffee-shops), we collected 4 smaller datasets by filtering by business name. We obtained Twitter handles of relevant businesses (e.g., @Starbucks for coffee shops and @McDonalds for restaurants). Then, for every domain, from all pairs of tweets $\{Q, A\}$, where A is a reply to Q and Q contains a question mark, we selected those where either Q or A contained either a mention ("@") or a hashtag that matches the business. Dataset sizes are reported in Table 2. Before training we performed entity extraction on each Q and A, and appended any entity type we could recognize in it with high confidence (e.g., "it opens at 7pm | TIME"). We trained a question2answer and an answer2question model for each of the 5 datasets. All models were trained on two NVIDIA Tesla K40 GPUs, for 250,000 steps. Each model training took from 2 to 7 days, depending on the dataset size.

The 10 trained models are hosted on a GPU-equipped HTTP server. To generate prompts for a slot, the Kite web app sends a request including the domain and the slot values (along with their entity types, if available). To generate max likelihood prompts, the GPU server invokes the answer2question model of the corresponding domain and returns the top-N results. Likewise, the question2answer model can be queried with a question associated to a slot to produce the top-N answers. To provide diversity in the results, Kite removes duplicate results by computing the word-level Levenshtein edit distance, and omitting any utterance u with distance d less than $0.1 \times max(u.length, r.length)$ for any utterance r already in the top-N results.

5 EVALUATION

We evaluate Kite based on (1) How well does it extract task models from a sample of representative apps? (2) How well does it generate questions and answers for bot conversations? (3) How usable and useful is it for developers?

Table 3: The 25 apps on which we tested Kite along with their category, tested task and number of collected traces. For the intents and slots in the extracted task model, we report their count, precision and recall.

App	Category	Task	Traces	Intents			Slots		
				Total	Precision	Recall	Total	Precision	Recall
Open Table	Food	Reserve restaurant	8	9	0.89	1	13	1	0.93
AllRecipes	Food	Find recipe	11	9	1	0.90	9	0.80	1
KCLS	Books	Hold book	8	10	1	0.83	11	1	1
Dunkin Donuts	Food	Order coffee	3	5	1	1	7	1	0.88
Flixster	Entert.	Find movie	5	4	1	1	4	1	1
Zagat	Food	Find restaurant	5	3	1	1	3	1	1
Trip Advisor	Travel	Reserve restaurant	4	5	1	1	8	1	1
RecipeBook	Food	Find recipe	5	3	0.67	1	4	1	1
Big Oven	Food	Find recipe	8	8	0.88	1	6	1	1
Princeton Lib	Books	Hold book	8	8	1	1	7	1	0.88
Cincinnati Lib	Books	Hold book	5	7	1	1	4	1	0.57
Tim Hortons	Food	Order coffee	4	2	1	1	8	0.88	0.78
McDonald's	Food	Order coffee	6	5	1	1	11	0.82	0.82
Starbucks	Food	Order coffee	5	6	1	0.86	7	0.86	0.6
AMC Theaters	Entert.	Buy movie ticket	4	4	1	1	5	1	0.83
Cinemark	Entert.	Buy movie ticket	7	7	1	1	5	1	0.71
BBCNews	News	Find news article	7	9	1	1	6	1	1
Kayak	Travel	Search for flights	4	5	1	1	5	1	0.71
Booking.com	Travel	Book hotel	3	10	1	1	14	0.93	0.81
Papa Johns	Food	Order pizza	3	6	1	1	10	0.83	0.89
NHL	Sports	Check game score	5	4	1	1	3	1	1
Amazon Music	Music	Find song	9	3	1	1	6	0.86	0.83
Zillow	House	Find house listing	5	7	0.71	1	8	0.89	0.7
Comcast Xfinity	Tools	Pay cable bill	5	4	1	1	10	1	1
WebMD	Health	Look up symptoms	7	9	1	1	16	0.89	0.93
Average (σ)			5.8 (2.1)	6.1 (2.4)	0.97 (0.09)	0.98 (0.05)	7.6 (3.4)	0.95 (0.06)	0.88 (0.13)

5.1 Accuracy of task models

We selected 31 popular Android apps from various domains which are relevant to task-oriented bots such as food, books, entertainment, travel, etc. Apps were selected based on popularity in the Google Play Store or relevance in the task domain.[2] Out of these 31 apps, 5 were used for designing and debugging Kite (top 5 rows in Table 3), 6 were excluded due to technical limitations of our trace collection mechanism (details in § 7), and the remaining 20 were used for evaluation (i.e., Kite was executed on them without any further modifications). Reasons for the excluded apps included: prevalence of web views for which Kite cannot collect type, UI tree, and other necessary information (Nook, IMDB); app crashes due to our modified Android framework (Hipmunk, Dunn Bros, Spotify), unusually high UI refresh rate (Yummly) or unusually large UI layouts with over 5000 UI elements per page (IMDB) that made our trace collector unstable – an app usually has 200–300 UI elements per page. Hence, 25 apps were used in total.

We selected one task from each of the 25 apps and collected multiple (avg=5.8, σ=2.1) interaction traces. For apps where the same task could be executed in many different ways we collected more traces. For instance, in AllRecipes, a recipe can be selected in four ways (keyword search, dinner spinner, recommended recipe, similar recipe), so covering all these paths required at least 4 traces. During trace collection, we also tried to use different values for the task parameters so to "exercise" the app's options as much as possible. During app interaction, we created a "ground truth task model" by manually recording whether each UI interaction we performed represented an intent or a slot.

As reported in Table 3, Kite generated task models for all 25 apps. We evaluated task models in terms of *precision* and *recall* of intent and slot extraction. Intent precision is computed as the number of correct intents out of all extracted intents, while intent recall as the number of correct intents out of all intents that could have been extracted from the traces (i.e., based on the ground truth task model). Similar definitions hold for slot recall and slot precision. Across all apps, intent extraction achieved precision and recall close to 100% (for 19 out of 25 apps they were both 100%). Slot extraction achieved 95% precision and 88% recall.

False negatives in slot extraction (i.e, missed slots) occurred when the user input was provided through UI widgets such as sliders or incremental counters (i.e., where the value is based on the number of clicks), which Kite does not capture. False positives occurred in slot extraction because (1) some fields representing the same slot in different screen layouts were not combined, resulting in duplicate slots, or (2) UI elements such as buttons, instead of having immutable labels, had dynamic labels reflecting the status of the system (e.g., "View 88 Results"), thus being incorrectly classified as slots. Finally, false positives occurred in intent extraction because of advertising activities that were recognized as intents (RecipeBook and BigOven) or because of duplicate intents caused by limited

[2]For instance, we selected Starbucks for its popularity, and Tim Hortons and Dunkin Donuts because they are popular coffee-shop chains [64].

visibility of the app structure (OpenTable and Zillow). In general, these kinds of false positives should be easily identifiable by the developer in the Kite web tool.

Overall, Kite was able to work successfully with a random set of popular Android apps. 6 out of 31 apps were excluded due to tractable engineering issues. Overall, Kite produced accurate intent graphs, and most errors could be fixed by extending UI event capture to additional UI widgets.

5.2 Question/answer relevance and quality

Automatic evaluation. As reported in Table 2, we computed BLEU scores[3] for our 10 neural network models. The BLEU scores may seem low, which however, is not untypical of neural conversation models (e.g., in [30, 52, 84], also using Twitter datasets, BLEU scores ranged from 0.44 to 1.66). Most likely due to the dataset sizes, the multi-domain model yielded the highest BLEU scores, while the coffee-shops and library domains yielded the lowest. We complemented automatic evaluation with human evaluation.

Human evaluation. We extracted a subset of questions and answers generated for our test tasks, and asked three independent raters to judge them based on relevance and quality.

Question evaluation. The question dataset was produced as follows. We selected 5 tasks ("make a restaurant reservation", "find a recipe", "hold a book", "order coffee", and "buy a movie ticket"), and for each one we selected an app (the first in alphabetical order in that task domain) and the corresponding task model. We then extracted all questions that Kite generated for all slots in these 5 task models, thus obtaining a dataset of 1562 questions (1093 unique) for 38 slots. Three independent raters were shown a screenshot of the screen from which the slot was extracted and the slot's possible values, and were asked to identify whether each of the generated questions was appropriate on a binary scale for the given slot in the context of the task.

For all slots in the 5 tasks, at least one generated question was identified as "appropriate" by all three raters. In total, 96 questions were identified as appropriate by all three raters. Among them, 49 were generated using the answer2question models[4], 37 using rules based on the entity type, and 10 using rules based on the UI element's text label (these methods were described in §3.2). The inter-rater agreement [28] was $\kappa = 0.91$, suggesting excellent agreement [48]. Although the accuracy of the generated questions may seem low (9%), for the purpose of Kite, it is far more important to ensure at least one appropriate question in the generated set as only one question is needed for conversation. In contrast, Kite needs to generate as many appropriate answers as possible to train robust language understanding models.

Answer evaluation. For each slot in the previously described question dataset, we selected one appropriate question from the generated questions. Among those, there were three questions that were overly general (e.g., "Which one do you want to choose?"), so to obtain more realistic results, we modified them to be more specific (e.g., "Which size do you want to choose?"), as an actual developer would do in Kite. 11 questions were also duplicate across slots (e.g. common ones such as "What time do you want to choose"). Finally, we excluded 5 questions of the type "What's your name?" or "What's your phone number?" for which answer generation is not necessary. Overall we obtained 22 unique questions.

For each question, we invoked two question2answer models (the domain-specific model[5] and the multi-domain model, as listed in Table 2) and limited the response to a maximum of 20 answers per model per question. 807 answers were generated. Three independent raters rated each answer in terms of quality and relevance on a five point scale (5 is better), where quality focuses on whether the answer is grammatically correct and fluent, and relevance evaluates whether the answer appropriately responds to the question. Table 4 summarizes the results. The average Spearman correlation coefficients were $\rho = 0.83$ for relevance and $\rho = 0.78$ for quality for all pairs of raters, suggesting high inter-rater reliability among the three raters.

Table 4: Human evaluation for generated answers (5-point scale). We report number of answers, avg (and σ) of relevance and quality.

Task	Multi-domain models			Domain-specific models		
domain	#	Relev	Quality	#	Relev	Quality
Restaurants	103	2.9 (1.3)	4.5 (1.1)	136	**4.2** (0.9)	**4.9** (0.3)
Movies	80	4.1 (1.2)	4.9 (0.5)	80	**4.2** (1.4)	**4.9** (0.3)
Libraries	40	2.5 (1.6)	**4.8** (0.7)	40	**3.1** (1.9)	3.5 (1.6)
Coffee	60	2.8 (1.0)	**5.0** (0.1)	73	**3.6** (1.4)	4.9 (0.3)
Recipes	100	3.5 (1.5)	4.5 (1.2)	95	**3.9** (1.2)	**4.8** (0.4)

The generated questions have high quality, indicating that they are mostly grammatically correct. Relevance is lower, but still acceptable for our use, as a developer can discard inappropriate answers. The domain-specific models outperformed the multi-domain model in relevance for all 5 tasks, which confirms our hypothesis that using domain-specific data helps generate more relevant utterances. The human judgment of relevance for answers generated by the domain-specific models correlates with their BLEU scores.

5.3 Developer study

We conducted a preliminary in-lab developer study to gauge the usability and usefulness of Kite for developers. We recruited 10 participants aged 22-37 (mean = 27.4, SD = 4.6), 7 males and 3 females. All but one participant considered themselves experienced software developers. Three had bot development experience, two of whom were professional bot developers in a large tech company.

Each 1-hour study session consisted of a system overview, a walk-through tutorial, a bot development task, a questionnaire, and an informal discussion session. During the overview, we explained the goals of Kite and how it fits into the bot development process. In the tutorial, we showed how to use Kite to create a template for a coffee-ordering bot using traces from Dunkin Donuts. Each participant was then asked to use Kite to create a template for a restaurant reservation bot using traces we had previously collected for OpenTable. To get familiar with OpenTable, we asked them to use it once to search for a restaurant and pretend making a

[3] using the BLEU implementation in Moses [46] using n-grams up to $N = 4$, which is consistent with [63].

[4] Question generation used the domain-specific models listed in Table 2 for all tasks except for the recipe task which used the multi-domain model. The model's output was limited to a maximum of 10 results per slot value.

[5] For the recipes task the restaurants specific model was used.

Table 5: Participant feedback on the usefulness and usability of Kite (5-point Likert scale)

Statement	Mean	SD
I find the system helpful in bot development.	4.5	0.85
This system would help me understand the flow of the task that I'm building a bot for.	4.8	0.42
I feel the system is easy to use.	4.2	0.79
This system can help me save my time.	4.7	0.67
The task model extracted from the app is useful.	4.8	0.42
The slots extracted for intents are useful.	4.7	0.48
The prompts generated for each slot are useful.	4.1	0.74
The sample utterances generated for slots are useful.	3.6	0.97
I'm satisfied with my experience using this system.	4.5	0.71
I'd want to use this system if I need to build a chatbot.	4.6	0.70

reservation. Then, we asked them to iteratively edit and test the bot template using the Kite web app and the bot preview feature (Figure 6), until they were satisfied with its quality.

After the task, each participant completed a questionnaire on the usefulness and usability of the system. The questions asked participants to rate 10 statements on a 5-point Likert scale from "strongly disagree (1)" to "strongly agree (5)". Finally, we had a discussion session with the participant to solicit their questions and feedback on the system.

Results. All 10 participants successfully created a template for the restaurant bot. The average task on time was 21.4 minutes (SD = 4.0). As shown in Table 5, participants found Kite useful in the bot development process, and were satisfied with their experience.

Although we demonstrated Kite in the tutorial, some participants took longer to figure out how the structure of the task model mapped to the actual conversation in the bot preview tool. This, however, did not slow them down for too long, as they quickly understood this mapping by iteratively trying using different values for intent names, slot names and prompts for generating bot previews. One way to accelerate the construction of such mental maps could be adding a preview of sample conversation snippets on the selected slot or intent when the developer edits the task model.

Some participants found it insufficient to rely only on the extracted possible values and the generated names in the Kite web app to understand the purpose of slots and intents. They referred back to the OpenTable app to help understand the task model, which could also be a result of their lack of familiarity with the app and the task used in the study. This could be addressed by attaching app screenshots to the corresponding parts in the task model.

Participants considered the interactive visualization of the task model particularly helpful in the bot development. A participant, who is a professional bot developer, commented that she usually spent lots of time manually coming up with these graphs, so the generated visualization of extracted task models in Kite was very helpful. Prompt generation was considered useful too, but a participant also said that writing a prompt manually is not too hard either. As for the sample utterance generation, participants with experience in bot development could immediately appreciate its value. The others recognized its usefulness after we explained the current practice of training such models. The bot preview feature was also heavily used in the study.

Our participants were very impressed that Kite enabled them to generate a bot prototype with correct and reasonable conversational flows in a short amount of time with little developer effort required. However, they also expressed the need to modify the app-derived models to better fit the conversational experience. For example, in the Dunkin Donuts app, the generated bot would ask 5 consecutive questions (size, roast type, flavor, topping and sweetener) about customizing the coffee. A participant considered these questions to be too lengthy and unnecessary, expecting a better-designed bot to only ask about these options if the user proactively asked for, or to ask first a higher-level question (e.g., "Do you want any adds-on?"). This finding confirms the differences between designing for GUIs and designing for chatbots, reported in guidelines on conversational UX design (e.g., [6]): bot conversations should be brief, as going through 5 questions in conversation (either by text or by speech) would take longer and require higher cognitive load than skimming through 5 sets of options in a GUI.

Although our focus so far has been on quickly bootstrapping bot prototypes with correct and reasonable conversational flows, future work will look into helping developers to enable more natural conversations. Extending task model extraction to real user traces will also allow us to generate task models that highlight user usage so to identify, for instance, mandatory and optional slots, default and preferred slot values, or most common execution paths.

Overall, participants agreed that Kite is helpful in bot development, since it automates a big portion of what would otherwise need to be done manually. The results also suggest that Kite has a low learning barrier.

6 RELATED WORK

Conversational bots. Most *task-oriented bots* are built using slot-filling, which is hard to scale. Bot tools like Dialogflow [22] and LUIS.ai [61] offer basic templates for frequent use cases like weather, calendar, music, etc., but they are rather limited and only support single-turn conversations. Kite goes beyond single-turn conversations by modeling complete task workflows, and can support many types of task. Existing tools for utterance generation [31, 33] are limited to simple string permutations given a dictionary of terms [32]. Kite's neural transduction models can generate more varied utterances and take only one string as input.

Non-task oriented bots can be implemented using rule-based [17, 82] or corpus-based approaches [23, 53, 54, 70, 71, 73, 78]. The latter has been successfully used in social media [65, 79] and movie corpus [11]. It is yet to demonstrate whether this approach can work for multi-turn task-oriented bots [13, 30], which is Kite's target. It is also limited by the scarcity of large in-domain corpus.

Hybrid Code Networks (HCNs) [85] is an approach to make machine-learned systems practical by combining a recurrent neural network with developer hand-coded rules. HCNs reduce the amount of training data to the expense of developer effort. In contrast, Kite reduces both by leveraging app GUIs. Yet, HCNs can evolve over time through supervised or reinforcement learning based on conversation logs collected while the system is in use. Kite could be used to bootstrap HCNs with less developer effort. Another hybrid approach is the "knowledge-grounded" conversation model proposed in [30], which injects knowledge from textual

data (i.e., restaurant reviews on Foursquare) into models derived from conversational data (Twitter) to generate informative answers. We borrow from [30] the idea of combining conversational (Twitter) and non-conversational data (mobile apps). However, [30] only models single-turn responses and depends on in-domain knowledge sources such as Foursquare; Kite is not automated as [30], but targets multi-turn conversations in many domains.

Natural language interfaces for apps. NLify [38], Speechify [45] and PixelTone [49] add natural language interfaces to existing app GUIs, allowing users to issue single commands to an app by voice. In contrast, Kite targets completing tasks independently from app GUI with multiple dialog turns, all using natural language. Sugilite [55], Epidosite [57] and CoCo [50] allow users to demonstrate a task with an Android app or a website, tag it with a single natural language command, and later re-execute it automatically. We share with these systems the goal of helping users complete their tasks using natural language and learning a task from user demonstration. However, these are effectively record-and-replay systems with little task parametrization. Kite extracts fully-parametrized task models and goes beyond single-turn interactions.

Mining of human-generated app traces. Interaction traces generated using UI automation tools (e.g., [15]) are not suitable for extracting task models. Kite relies on app exploration driven by humans. ERICA [21], Rico [19] and ZIPT [20] mine human-generated app traces to support UX designers. ERICA processes interaction traces to detect UI elements and layouts representing common "user flows". ZIPT allows for testing of UI designs at scale to identify usability issues. Rico is a repository of mobile app designs. Kite shares with these systems a similar approach for collecting and representing app traces, but application traces are abstracted at two different levels. Kite uses app traces to learn app functionality represented as functions with inputs; ERICA, Rico and ZIPT study app traces to learn UI patterns. In Kite a "task" represents a long sequence of interactions to complete a transaction such as booking a restaurant; in ERICA, Rico and ZIPT a task is a short interaction representing a single action such as adding, searching or composing.

Analytics frameworks [9, 29, 37, 59] collect user data at scale and provide developers with insights, but the analysis is generally UI-focused, not at the level of app business logic. In the future, Kite could process traces collected for analytics purposes.

7 LIMITATIONS

Platform support. Kite extracts task models from Android apps, but its design does not preclude other platforms such as iOS, web, or IoT apps. Our main assumption is that a GUI interaction fires an event that can be captured and analyzed. The complexity of extending Kite to other platforms is a function of *i)* how extensively UI event capture is provided by the application framework, for example through accessibility services; and *ii)* how much app contents are organized into distinct pages (i.e., the less content in a page, the easier it is to classify at the intent level).

Trace collection. Our current prototype has some limitations which forced us to exclude 6 apps in our evaluation. It does not recognize gestures and sensory inputs: these are more common in games, maps, or image editing apps, which are not the focus of task-oriented bots. Kite tracks embedded web views in apps, but only at a coarse granularity because accessibility APIs do not report a web view's content type (e.g., text field or button) nor its UI tree. Traces are currently demonstrated manually by developers. In the future, we plan to use real user traces, collected for analytics or via crowdsourcing. We also plan to incorporate UI automation (e.g., [39]) to automate the trace collection to further reduce the developer effort.

Natural interaction. Our goal for bots was to generate appropriate and grammatically-correct dialogs. We did not aim for dialogs that sound natural by referring to previous terms, avoid repetition, or promote variation. As in prior work [47], crowdsourcing could refine our questions. We also did not test the quality of language understanding models trained with our generated answers. Kite-generated bots do not include intents for actions that are not directly accessible in the corresponding app GUI. Kite also targets bots that use a flow chart with a clearly-defined control structure to guide users towards task completion. Next-generation bots will make such flow less explicit and aim for more fluid interactions that can better resemble human conversations. However, like others [78, 85], we believe that future bots will still require some form of task model, to ensure interaction consistency and to satisfy in-task constraints.

Integration with bot platforms and external APIs. Kite does not generate ready-to-use bots, but rather bot skeletons. An operational bot needs at least: (1) connection to APIs of external services, (2) a UI for communicating API results, and (3) the training of language understanding models for answers and intents. Kite could be integrated with existing bot frameworks [5, 36, 60], which already provide such functionality. For language understanding, Kite provides sample utterances for answers. With existing NLP tools [22, 61] a developer can quickly train a model. However, Kite does not yet provide utterances for training intent recognition, which we leave for future work. If specifications of external APIs for intent execution are available, future versions of Kite could also automatically construct API calls based on slot filling and validate user inputs based on constraints of the API parameters.

8 CONCLUSIONS

Existing approaches to building bots require either extensive manual work or an extensive corpus of data, both of which hinders scalability. Kite does not generate fully-functional bots, but it does automatically provide a bot template starting from few application traces and deals with the part of making a bot which a regular developer is least likely to be familiar with. We have used Kite with 25 apps in a variety of domains, and showed that our templates are comprehensive and precise. Users who interacted with Kite found it useful and easy-to-use. We envision Kite becoming part of existing bot platforms, where developers can convert their apps or explore other apps to create conversational experiences.

ACKNOWLEDGMENTS

We thank our shepherd, Yubin Xia, and the anonymous reviewers for their feedback. We also thank Chris Brockett and Michel Galley for their help in training the neural network transduction models, and Brad Myers for his feedback on drafts of this paper.

REFERENCES

[1] 2017. BotsCrew. http://botscrew.com/.

[2] MartÃŋn Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 265–283.

[3] Khalid Alharbi and Tom Yeh. 2015. Collect, Decompile, Extract, Stats, and Diff: Mining Design Pattern Changes in Android Apps. In *Proc. of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI '15)*. ACM, 515–524. https://doi.org/10.1145/2785830.2785892

[4] Husam Ali, Yllias Chali, and Sadid A Hasan. 2010. Automation of question generation from sentences. In *Proc. of the Third Workshop on Question Generation, QG2010.*

[5] Amazon Alexa. 2017. Alexa Skills Kit. https://developer.amazon.com/alexa-skills-kit.

[6] Amazon Alexa - Voice Design Guide. 2017. What Users Say - Making sure Alexa understands what people are saying. https://developer.amazon.com/designing-for-voice/what-users-say/.

[7] Amazon AWS. 2017. Amazon Lex. https://aws.amazon.com/lex/.

[8] Android. 2017. Support Library. https://developer.android.com/topic/libraries/support-library/index.html.

[9] Appsee. 2017. https://www.appsee.com/.

[10] Bahadir Ismail Aydin, Yavuz Selim Yilmaz, Yaliang Li, Qi Li, Jing Gao, and Murat Demirbas. 2014. Crowdsourcing for Multiple-choice Question Answering. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI'14)*. AAAI Press, 2946–2953. http://dl.acm.org/citation.cfm?id=2892753.2892959

[11] Rafael E. Banchs. 2012. Movie-DiC: A Movie Dialogue Corpus for Research and Development. In *Proc. of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2 (ACL '12)*. Association for Computational Linguistics, 203–207.

[12] Daniel G. Bobrow, Ronald M. Kaplan, Martin Kay, Donald A. Norman, Henry Thompson, and Terry Winograd. 1977. GUS, a Frame-driven Dialog System. *Artif. Intell.* 8, 2 (April 1977), 155–173.

[13] Antoine Bordes and Jason Weston. 2016. Learning End-to-End Goal-Oriented Dialog. *CoRR* abs/1605.07683 (2016). http://arxiv.org/abs/1605.07683

[14] Business Insider. 2017. Amazon's Alexa has gained 14,000 skills in the last year. http://www.businessinsider.com/amazon-alexa-how-many-skills-chart-2017-7.

[15] Pei-Yu (Peggy) Chi, Sen-Po Hu, and Yang Li. 2018. Doppio: Tracking UI Flows and Code Changes for App Development. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 455, 13 pages. https://doi.org/10.1145/3173574.3174029

[16] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv:1412.3555 [cs]* (Dec. 2014). http://arxiv.org/abs/1412.3555 arXiv: 1412.3555.

[17] Kenneth Mark Colby, Sylvia Weber, and Franklin Dennis Hilf. 1971. *Artificial Paranoia*. Vol. 2. 1 – 25 pages.

[18] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.

[19] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proc. of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, 845–854. https://doi.org/10.1145/3126594.3126651

[20] Biplab Deka, Zifeng Huang, Chad Franzen, Jeffrey Nichols, Yang Li, and Ranjitha Kumar. 2017. ZIPT: Zero-Integration Performance Testing of Mobile App Designs. In *Proc. of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, 727–736. https://doi.org/10.1145/3126594.3126647

[21] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proc. of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, 767–776. https://doi.org/10.1145/2984511.2984581

[22] Dialogflow. 2017. Build natural and rich conversational experiences. https://dialogflow.com/.

[23] Jesse Dodge, Andreea Gane, Xiang Zhang, Antoine Bordes, Sumit Chopra, Alexander H. Miller, Arthur Szlam, and Jason Weston. 2016. Evaluating Prerequisite Qualities for Learning End-to-End Dialog Systems. In *Proc. of ICLR*. http://arxiv.org/abs/1511.06931

[24] Xinya Du, Junru Shao, and Claire Cardie. 2017. Learning to Ask: Neural Question Generation for Reading Comprehension. In *Proc. of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1342–1352. http://aclweb.org/anthology/P17-1123

[25] Facebook. 2017. Wit.ai. https://wit.ai/.

[26] Earlence Fernandes, Oriana Riva, and Suman Nath. 2016. Appstract: On-the-fly App Content Semantics with Better Privacy. In *Proc. of the 22nd Annual International Conference on Mobile Computing and Networking (MobiCom '16)*. 361–374.

[27] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. In *Proc. of the 43rd Annual Meeting on Association for Computational Linguistics (ACL '05)*. Association for Computational Linguistics, 363–370.

[28] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[29] Flurry. 2017. Flurry Analytics. https://y.flurry.com/.

[30] Marjan Ghazvininejad, Chris Brockett, Ming-Wei Chang, Bill Dolan, Jianfeng Gao, Wen-tau Yih, and Michel Galley. 2017. A Knowledge-Grounded Neural Conversation Model. *CoRR* abs/1702.01932 (2017). http://arxiv.org/abs/1702.01932

[31] GitHub. 2017. alexa-js/alexa-utterances. https://github.com/alexa-js/alexa-utterances.

[32] GitHub. 2017. Code example: miguelmota/intent-utterance-generator. https://lab.miguelmota.com/intent-utterance-expander/example/.

[33] GitHub. 2017. Code: miguelmota/intent-utterance-generator. https://github.com/miguelmota/intent-utterance-generator.

[34] Gizmodo. 2017. Facebook Chatbots Are Frustrating and Useless. https://gizmodo.com/facebook-messenger-chatbots -are-more-frustrating-than-h-1770732045.

[35] Gizmodo. 2017. The Amazon Echo Now Has 10,000 Mostly Useless 'Skills'. https://gizmodo.com/the-amazon-echo-now-has-10-000-mostly-useless-skills-179269536.

[36] Google. 2017. Actions on Google. https://developers.google.com/actions/.

[37] Google. 2017. Google Analytics. https://analytics.google.com/.

[38] Seungyeop Han, Matthai Philipose, and Yun-Cheng Ju. 2013. NLify: Lightweight Spoken Natural Language Interfaces via Exhaustive Paraphrasing. In *Proc. of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '13)*. ACM, 429–438. https://doi.org/10.1145/2493432.2493458

[39] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proc. of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. ACM, 204–217. https://doi.org/10.1145/2594368.2594390

[40] Sepp Hochreiter and JÄijrgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780.

[41] IBM. 2017. Watson. https://www.ibm.com/watson/developer/.

[42] Daniel Jurafsky and James H. Martin. 2017. *Speech and Language Processing* (3rd ed.). Chapter 29. Draft available online at https://web.stanford.edu/~jurafsky/slp3/29.pdf.

[43] Saidalavi Kalady, Ajeesh Elikkottil, and Rajarshi Das. 2010. Natural language question generation using syntax and keywords. In *Proc. of the Third Workshop on Question Generation, QG2010.*

[44] Nal Kalchbrenner and Phil Blunsom. 2013. Recurrent Continuous Translation Models.. In *EMNLP*. ACL, 1700–1709.

[45] Tejaswi Kasturi, Haojian Jin, Aasish Pappu, Sungjin Lee, Beverley Harrison, Ramana Murthy, and Amanda Stent. 2015. The Cohort and Speechify Libraries for Rapid Construction of Speech Enabled Applications for Android. In *Proc. of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*. Association for Computational Linguistics, 441–443. http://aclweb.org/anthology/W15-4661

[46] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open Source Toolkit for Statistical Machine Translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions (ACL '07)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 177–180. http://dl.acm.org/citation.cfm?id=1557769.1557821

[47] Igor Labutov, Sumit Basu, and Lucy Vanderwende. 2015. Deep Questions without Deep Understanding. In *Proc. of the 53rd Annual Meeting of the Association for Computational Linguistics, ACL 2015.*

[48] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[49] Gierad P. Laput, Mira Dontcheva, Gregg Wilensky, Walter Chang, Aseem Agarwala, Jason Linder, and Eytan Adar. 2013. PixelTone: A Multimodal Interface for Image Editing. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, 2185–2194.

[50] Tessa Lau, Julian Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey P. Bigham, and Jeffrey Nichols. 2010. A Conversational Interface to Web Automation. In *Proc. of the 23Nd Annual ACM Symposium on User Interface Software and Technology (UIST '10)*. ACM, 229–238. https://doi.org/10.1145/1866029.1866067

[51] Oliver Lemon, Kallirroi Georgila, James Henderson, and Matthew Stuttle. 2006. An ISU Dialogue System Exhibiting Reinforcement Learning of Dialogue Policies: Generic Slot-filling in the TALK In-car System. In *Proc. of the 11th Conference of the European Chapter of the Association for Computational Linguistics: Posters*

& *Demonstrations (EACL '06)*. Association for Computational Linguistics, 119–122.

[52] Jiwei Li, Michel Galley, Chris Brockett, Georgios Spithourakis, Jianfeng Gao, and Bill Dolan. 2016. A Persona-Based Neural Conversation Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 994–1003. https://doi.org/10.18653/v1/P16-1094

[53] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. 2016. Deep Reinforcement Learning for Dialogue Generation. In *In Proc. of EMNLP*.

[54] Jiwei Li, Will Monroe, Tianlin Shi, Alan Ritter, and Dan Jurafsky. 2017. Adversarial Learning for Neural Dialogue Generation. In *In Proc. of EMNLP*.

[55] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proc. of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, 6038–6049.

[56] Toby Jia-Jun Li, Igor Labutov, Brad A. Myers, Amos Azaria, Alexander I. Rudnicky, and Tom M. Mitchell. 2018. An End User Development Approach for Failure Handling in Goal-oriented Conversational Agents. In *Studies in Conversational UX Design*. Springer.

[57] Toby Jia-Jun Li, Yuanchun Li, Fanglin Chen, and Brad A. Myers. 2017. Programming IoT Devices by Demonstration Using Mobile Apps. In *End-User Development*, Simone Barbosa, Panos Markopoulos, Fabio Paternò, Simone Stumpf, and Stefano Valtolina (Eds.). Springer International Publishing, Cham, 3–17.

[58] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proc. of the 39th International Conference on Software Engineering Companion*. IEEE Press, 23–26.

[59] Localytics. 2017. http://www.localytics.com/.

[60] Microsoft. 2017. Bot framework. https://dev.botframework.com/.

[61] Microsoft - Cognitive Services. 2017. Language Understanding Intelligent Service. https://www.luis.ai.

[62] Ruslan Mitkov and Le An Ha. 2003. Computer-aided Generation of Multiple-choice Tests. In *Proc. of the HLT-NAACL 03 Workshop on Building Educational Applications Using Natural Language Processing - Volume 2 (HLT-NAACL-EDUC '03)*. Association for Computational Linguistics, 17–22. https://doi.org/10.3115/1118894.1118897

[63] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proc. of the 40th Annual Meeting on Association for Computational Linguistics (ACL '02)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[64] Ranker. 2017. Coffee Shop Chains That Make Mornings Bearable. https://www.ranker.com/list/best-coffee-shop-chains/chef-jen.

[65] Alan Ritter, Colin Cherry, and William B. Dolan. 2011. Data-driven response generation in social media. In *Proc. of EMNLP*. Association for Computational Linguistics, 583–593.

[66] Search Engine Roundtable. 2017. Google: Chatbots Don't Make Your Pages Better. https://www.seroundtable.com/google-on-chatbots-seo-24494.html.

[67] Vasile Rus, Brendan Wyse, Paul Piwek, Mihai Lintean, Svetlana Stoyanchev, and Cristian Moldovan. 2010. The First Question Generation Shared Task Evaluation Challenge. In *Proc. of the 6th International Natural Language Generation Conference (INLG '10)*. Association for Computational Linguistics, 251–257. http://dl.acm.org/citation.cfm?id=1873738.1873777

[68] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 2013. Insights into Layout Patterns of Mobile User Interfaces by an Automatic Analysis of Android Apps. In *Proc. of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*. ACM, 275–284. https://doi.org/10.1145/2494603.2480308

[69] Iulian Vlad Serban, Alberto García-Durán, Çaglar Gülçehre, Sungjin Ahn, Sarath Chandar, Aaron C. Courville, and Yoshua Bengio. 2016. Generating Factoid Questions With Recurrent Neural Networks: The 30M Factoid Question-Answer Corpus. In *Proc. of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016*. http://aclweb.org/anthology/P/P16/P16-1056.pdf

[70] Iulian V. Serban, Alessandro Sordoni, Yoshua Bengio, Aaron Courville, and Joelle Pineau. 2016. Building End-to-end Dialogue Systems Using Generative Hierarchical Neural Network Models. In *Proc. of the 30th AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 3776–3783.

[71] Lifeng Shang, Zhengdong Lu, and Hang Li. 2015. Neural Responding Machine for Short-Text Conversation. In *Proc. of ACL*. 1577–1586.

[72] Northwoods Software. 2017. GoJS Diagrams for JavaScript and HTML. https://gojs.net/

[73] Alessandro Sordoni, Michel Galley, Michael Auli, Chris Brockett, Yangfeng Ji, Margaret Mitchell, Jian-Yun Nie, Jianfeng Gao, and Bill Dolan. 2015. A Neural Network Approach to Context-Sensitive Generation of Conversational Responses. In *Proc. of NAACL-HLT*. 196–205.

[74] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proc. of NIPS*. Montreal, CA. http://arxiv.org/abs/1409.3215

[75] TechCrunch. 201. Kik users have exchanged over 1.8 billion messages with the platform's 20,000 chatbots. https://techcrunch.com/2016/08/03/kik-users-have-exchanged-over-1-8-billion-messages-with-the-platforms-20000-chatbots.

[76] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *arXiv:1706.03762 [cs]* (June 2017). http://arxiv.org/abs/1706.03762 arXiv: 1706.03762.

[77] Venture Beat. 2017. One year later, here's the state of the chatbot economy. https://venturebeat.com/2017/06/11/one-year-later-heres-the-state-of-the-chatbot-economy.

[78] Oriol Vinyals and Quoc V. Le. 2015. A Neural Conversational Model. In *Proc. of ICML Deep Learning Workshop*.

[79] Hao Wang, Zhengdong Lu, Hang Li, and Enhong Chen. 2013. A Dataset for Research on Short-Text Conversations. In *Proc. of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 935–945.

[80] Zhuoran Wang and Oliver Lemon. 2013. A Simple and Generic Belief Tracking Mechanism for the Dialog State Tracking Challenge: On the believability of observed information. In *Proc. of the SIGDIAL 2013 Conference*. Association for Computational Linguistics, 423–432.

[81] Wayne Ward and Sunil Issar. 1994. Recent Improvements in the CMU Spoken Language Understanding System. In *Proc. of the Workshop on Human Language Technology (HLT '94)*. Association for Computational Linguistics, 213–216.

[82] Joseph Weizenbaum. 1966. ELIZA—a Computer Program for the Study of Natural Language Communication Between Man and Machine. *Commun. ACM* 9, 1 (Jan. 1966), 36–45. https://doi.org/10.1145/365153.365168

[83] Johannes Welbl, Nelson F. Liu, and Matt Gardner. 2017. Crowdsourcing Multiple Choice Science Questions. *CoRR* abs/1707.06209 (2017). http://arxiv.org/abs/1707.06209

[84] Tsung-Hsien Wen, Milica Gašić, Dongho Kim, Nikola Mrkšić, Pei-Hao Su, David Vandyke, and Steve Young. 2015. Stochastic Language Generation in Dialogue using Recurrent Neural Networks with Convolutional Sentence Reranking. In *Proc. of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL)*. Association for Computational Linguistics.

[85] Jason D. Williams, Kavosh Asadi, and Geoffrey Zweig. 2017. Hybrid Code Networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. In *Proc. of the 55th Annual Meeting of the Association for Computational Linguistics (ACL 2017)*.

[86] Xingdi Yuan, Tong Wang, Çaglar Gülçehre, Alessandro Sordoni, Philip Bachman, Sandeep Subramanian, Saizheng Zhang, and Adam Trischler. 2017. Machine Comprehension by Text-to-Text Neural Question Generation. *CoRR* abs/1705.02012 (2017). http://arxiv.org/abs/1705.02012

[87] Qingyu Zhou, Nan Yang, Furu Wei, Chuanqi Tan, Hangbo Bao, and Ming Zhou. 2017. Neural Question Generation from Text: A Preliminary Study. *CoRR* abs/1704.01792 (2017). http://arxiv.org/abs/1704.01792