Tartan: A taskbot that assists with recipes and do-it-yourself projects

Li-Wei Chen
Carnegie Mellon University
Pittsburgh PA, US
liweiche@cs.cmu.edu

Ta-Chung Chi
Carnegie Mellon University
Pittsburgh PA, US
tachungc@andrew.cmu.edu

Daksh Uday Shah
Carnegie Mellon University
Pittsburgh PA, US
dakshuds@tepper.cmu.edu

Clive Gomes
Carnegie Mellon University
Pittsburgh PA, US
cliveg@andrew.cmu.edu

Jiajun Bao
Carnegie Mellon University
Pittsburgh PA, US
jiajunb@andrew.cmu.edu

Karthik Ganesan
Carnegie Mellon University
Pittsburgh PA, US
karthikg@andrew.cmu.edu

Pratik Joshi
Carnegie Mellon University
Pittsburgh PA, US
pmjoshi@andrew.cmu.edu

Sujay Suresh Kumar
Carnegie Mellon University
Pittsburgh PA, US
sujayk@andrew.cmu.edu

Dhruv Naik
Carnegie Mellon University
Pittsburgh PA, US
drn@andrew.cmu.edu

James Hagerty
Carnegie Mellon University
Pittsburgh PA, US
jhagerty@andrew.cmu.edu

Alexander Rudnicky
Carnegie Mellon University
Pittsburgh PA, US
air@cs.cmu.edu

Abstract

Tartan is a multi-domain task-oriented bot that assists users with two different tasks: 1. cooking with recipes from Whole Foods and 2. doing projects from WikiHow. The bot's system is divided into two stages. In the first stage, the bot assists the user in identifying a task that they want to do. In the second stage, the bot guides the users through sequences of step-by-step instructions. We developed several machine learning based components to guide the bot, including a domain classifier, named entity extractor, and question answering module. The core of the system is a dialogue manager. The dialogue manager guides a user's session, referencing information from the aforementioned components. Primarily, the dialogue manager is rule-based, necessitated by the lack of domain-specific data available to it. We describe our systems and provide observations on its behavior.

1 Introduction

Recent research in conversational AI has centered on two types of systems: open-domain conversational agents, and task-oriented bots. Earlier information-access systems have matured to the point that they are being successfully deployed in commercial environments. They continue to be a focus for approaches (e.g. end-to-end systems) that benefit from large corpora of training data. Task,

Preprint. Under review.

or goal-oriented systems are focused on more complex tasks that involve an extended negotiation between a user and a bot. Domains have included restaurant reservations, hotel bookings, and travel arrangements. Although machine learning approaches have been used, they also depend on training data. Non-supervised approaches are being investigated [7] as well as ones that leverage declarative information [8] More often, particularly for novel domains and task, developers have been limited to rule-based (more generally, symbolic) approaches.

Existing task-oriented bots fall into two categories: a) modular systems, and 2) end-to-end systems. A modular system is divided into several components; critical functionalities like natural language understanding, dialogue management, and response generation are implemented separately. In contrast, an end-to-end system implements an integrated machine learning model to achieve the same functionality. Those working on a modular system are able to better control and interpret their system. End-to-end systems require a significant amount of data to achieve reasonable performance. But if successful can exhibit behavior that conforms well to the requirements of a domain

Tartan is a "taskbot" that assists a user with specific tasks, here for recipes and for do-it-yourself (DIY) projects. Our general goal was to design a single architecture that capable of operating in many domains, with minimal customization. While Tartan is currently limited to recipes and projects, additional domains could be added without major changes to its architecture. Not addressed in the current work, is how a taskbot might automatically "ingest" a new domain. A complete design for a taskbot should have a component that (automatically) populates a domain representation. A complete 'solution' to the taskbot challenge would also include an ability to learn from interactions, not only on the levels of language but also on the level of task strategies. We call operations meta-tasks.

Given a modular design, a taskbot domain can be partitioned into three stages (Figure 1). The first stage does NLU; we leverage machine learning as part of it[1]) to do harmful utterance detection, domain entity extraction (e.g. dish-name extraction), and intent classification. This stage creates critical context for the next two stages. In the second stage, the taskbot selects a response generator based on this context. Motivated by Ravenclaw [1], the second stage selects the optimal response generator (Figure 8). See Section 4 for detail. In the final stage, the selected response generator emits a response. Note that response generators incorporate additional logic. While, theoretically, it is possible to include all logic in the second stage, making the third stage response generator is much simpler.We believe that a higher level abstraction for response generation better balances the workload between selecting a response strategy and generating a response.

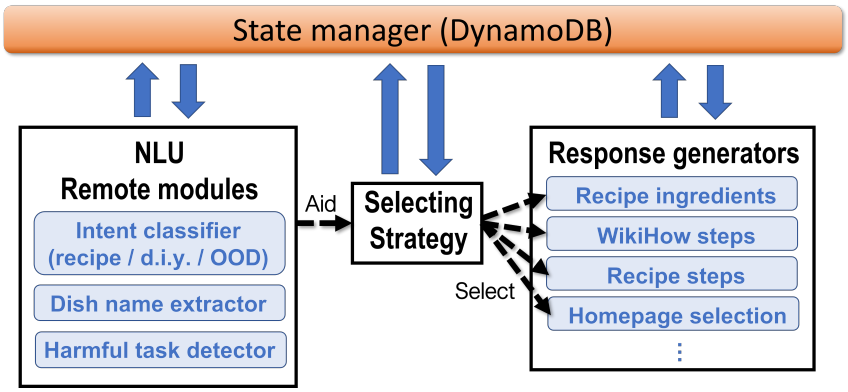


Figure 1: Abstracted overview of the Tartan taskbot. Three major components are shown. Remote modules are NN-based functions support the selection strategy in selecting appropriate response generators. All three components communicate via the state manager of cobot framework.

We also established a systematic workflow to guide development. New requirements/features came from an initial task analysis then evolved as user log data became available. After the introduction of a new feature, an initial check was made by deploying the bot in a team-only environment (BETA) and testing it internally to identify problems. This step minimized the chance that a catastrophic error might show up in the release version. If a version passed this initial screen, it became public (as a

[1]Tartan was hosted in the Amazon cloud computing environment, AWS, and used the Alexa cobot platform for development and fielding.

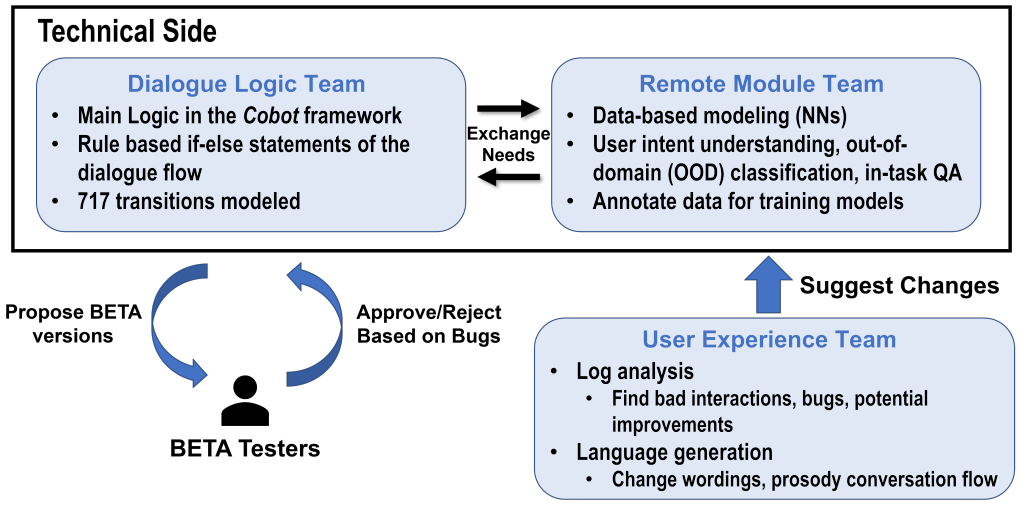


Figure 2: The system development pipeline. We divide the tasks into three groups. Each task has its primary goal and communicates with people in other ones to facilitate development.

PROD version). Logs from all public sessions were analyzed, and specific problems were identified and routed to individual developers. Problems fell into one of two categories: a) bugs (a coding error or an oversight usually with an obvious solution) and b) missing features (which could have bridged conversation failures). Figure 2 shows the workflow.

2 Performance Analysis

A key part of our activity was maintaining ongoing performance analysis of the taskbot. This included both batch evaluation of system components and analysis of user sessions. Users sessions were self-rated and our analysis focused on these. In addition, non-rated sessions were also examined to identify trends of interest. For the period February 15th–March 15th, a total of 400 user sessions were recorded, 284 of which were rated. The majority of calls were by first-timers, although some individuals (more exactly, devices[2]) logged many calls.

We tracked performance by sessions over time, finding that chronological tracking was influenced by day-to-day variability in the number of calls and by major swings in ratings, which we believe might have been due to changes in the Alexa team's marketing strategies (we were not privy to these). Figure 3 shows bot performance over sessions, using a mean lagging window of 50 (i.e. computed over the last 50 sessions). Our selected window size appears to (subjectively) preserve a useful level of detail. For illustrative purposes we fit a line to the series. Vertical grey lines indicate major software updates, primarily linked to module updates (see below). Note that minor releases are not shown, indicating that major architecture changes happened early on. Note that changes to system logic could be made as minor updates Call volume increased significantly near the end of the series.

We monitored progress by generating visualizations and metrics of ratings over time, Figure 5. We observed that the ratings were correlated with the average number of turns users conversed with our taskbot; r=0.256 for turns and r=0.228 for duration (N=856, Spearman). Both are significant at the $p < 0.001$ level. Based on this observation, we considered and implemented features that we believed would boost the number of conversation turns (which we considered to be correlated with degree of engagement) each user would have with the taskbot. Our attempts to do this resulted in better ratings over time. Figure 5 shows that this metric, as well as the related session duration metric track ratings,

[2]There was no mechanism for identifying individuals; it was possible for a device (say in a haousehold to be used by multuple individuals

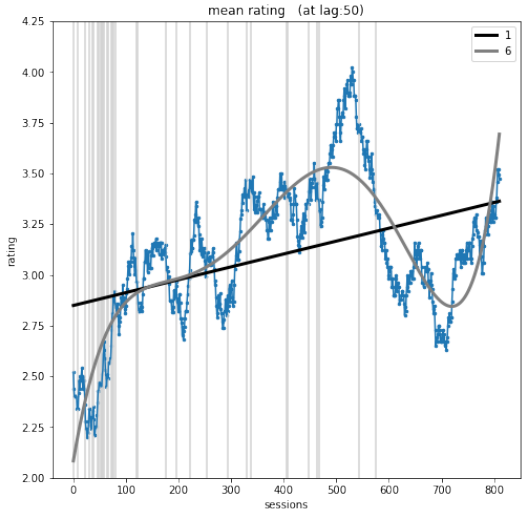


Figure 3: Ratings, computed over a lagging window of size 50, over the 800 sessions prior to the 3rd week of March 2022

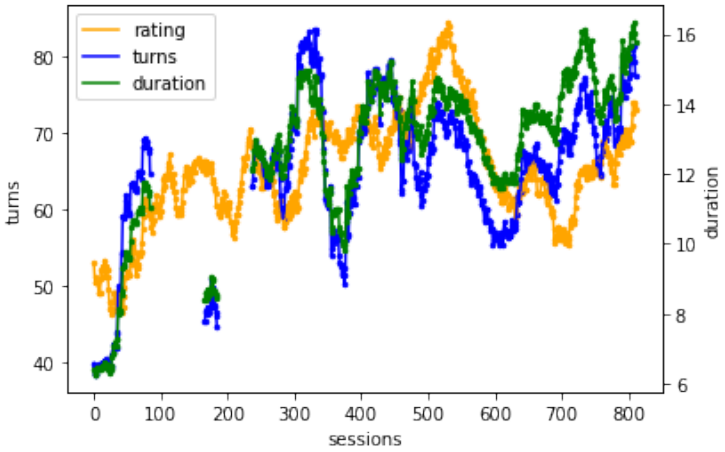


Figure 4: Combined graph of ratings, durations and turn count showing alignment. Note that tracks were manually scaled to approximately the same top value.

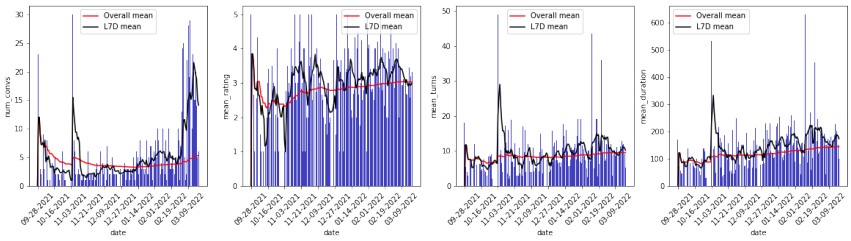


Figure 5: Plots we used to analyze progress over time, with indicators such as ratings, number of turns, duration per turn, number of conversations per day.

2.1 Session Log Analysis

Traffic volume for the taskbot was moderate for most of its public deployment. As a result, we were able to examine the log for every rated session (as well as some unrated ones).

An important aspect of making improvements was being able to analyze taskbot actions as well as patterns in caller behavior. These analyses were used to identify failure points, which were then distributed to members of the development team. To more easily analyze session data, we generated logs using a structured format. These logs allowed us to track down issues that degraded taskbot conversations in the context of multiple sessions. Changes that we implemented based on these analyses include: improving ambiguous response language, modifying dialog logic to reflect user expectations, and improving the naturalness of response wording and prosody. Analysis allowed us to identify and address problems in dialog flow. For example, identifying "vicious cycles" (where the user and bot start saying the same thing to each other over multiple turns) and "escape rooms" (where the caller does not know what to say to move on the the next state). Many of these problems made clear the importance of tracking context, managing self-awareness, and keeping the user informed about the taskbot's internal state.

From Figure 5, we observe that the average number of turns in a conversation has been consistently around 10 for the past 5 months. Moreover, the average duration a user spends on a conversation is approximately 160 seconds (2.5 minutes). These metrics indicate that the majority of users do not spend time actually working on projects or cooking with the bot step-by-step procedure; most users appeared to only browse through recipes and DIY projects. We shifted the focus of our team to design a better recommendation and browsing environment for the users. In the session logs, we note that 10% of the users use the word "next" more than 7 times (Up to 50 times in extreme cases), to navigate to the next step. For larger recipe and DIY instruction sets, the users began to get frustrated and ended the conversation. We conjecture that step-by-step was not appropriate for browsing (and modified presentations accordingly). We further conjecture that step sequences should be presented at different levels of detail through summarization; we did not have the opportunity to explore this.

2.2 Conversation Graph Analysis

We examined how conversational flows play out - whether the taskbot ends up in a "stuck" state and how often the taskbot ends up in a problematic state. To analyze the conversational flows, we generated conversational graphs to better understand these issues. We also used these graphs to determine how bug fixes reduce the amount of times the taskbot enters a problematic state.

Figure 6 shows conversation graphs. We designated certain states/responders as *Normal states*; we wanted the conversations to remain in these states. A set of *Problem states*, states/responders that correspond to undesirable points in the conversation (here the bot needs to to resolve and return to *Normal states* promptly). For "stuck" states, we can see in Figure 6a that the user attempts 4 times to escape the "*DIDN'T HEAR CLEARLY*" state (likely due to an ASR error). In contrast, we can also observe good conversational flows in Figures 6b and 6c.

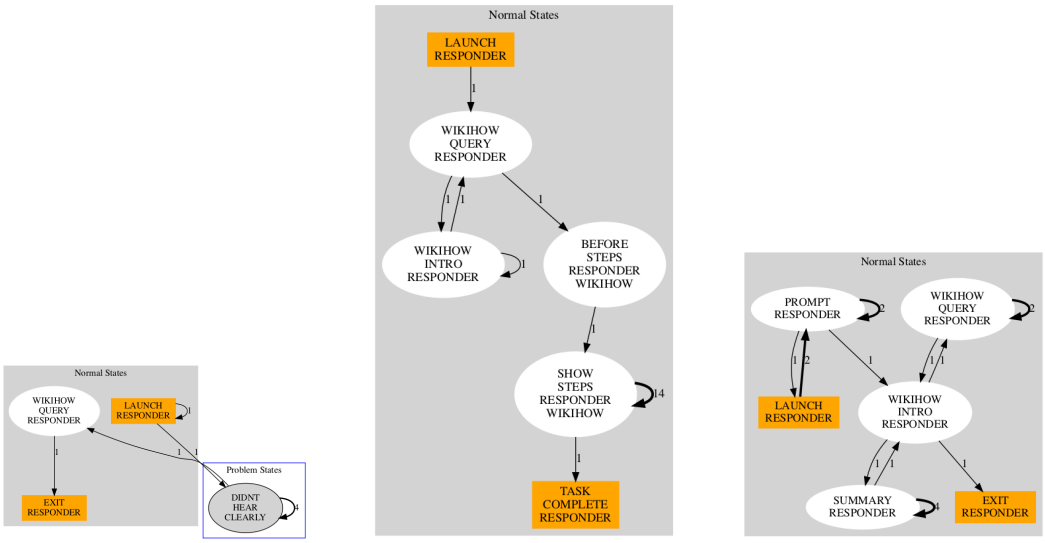
We examined a merged graph containing flows of sessions over different spans. We compared the graphs of 50 sessions starting from the first week of December, and the latest 50 sessions prior to 20th of March. We observed (7) far fewer "Problem States" in the newer sessions, due to improvements in remote modules, along with improvements in identifying user intents. Responders that were added after December 1st made the system more modular. These improved conversational flows, and show significant activity. As a result of this modularity, we are able to better understand which conversations are being properly exited, and how often a task is considered complete.

3 Remote Modules

Our remote modules provide services (hosted as lambda functions) that the core dialogue management module accesses through URLs. We delineate certain modules as remote modules in order to create a separation of responsibilities between core dialogue management and other components, enabling independent research and development of modules without affecting conversation flow. In this section, we describe remote modules.

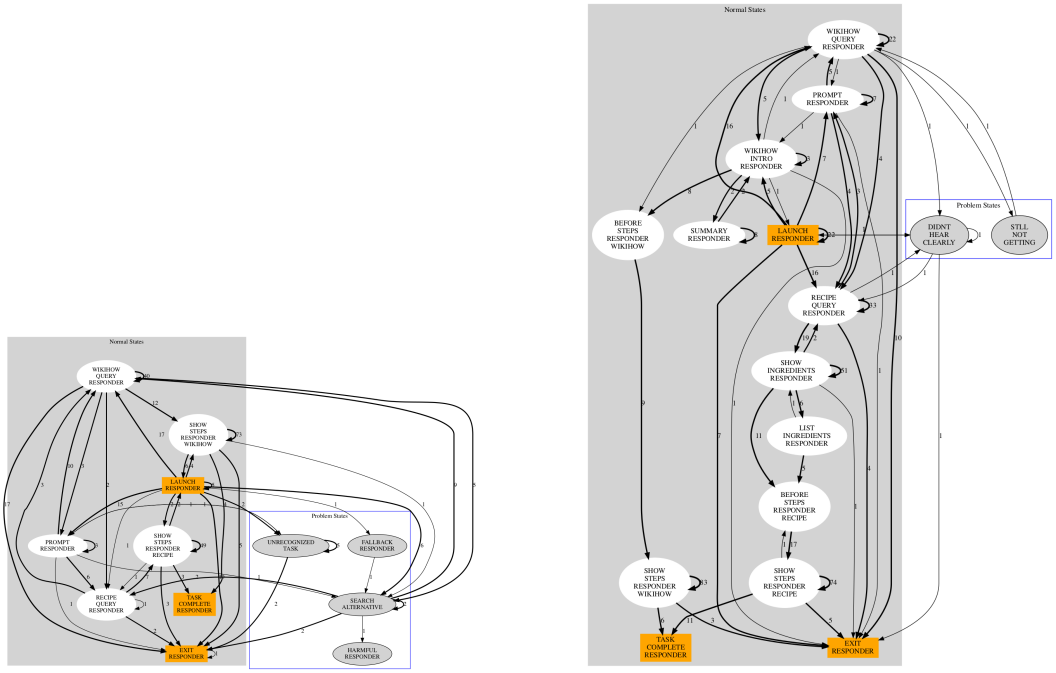
3.1 Embedding-similarity-based Classifier to Filter Offensive Input

To create this capability, we curated a database of dangerous tasks and offensive sentences internally. A state-of-the-art sentence embedding model (SimCSE) [3] was used to compute a sentence vector for each sentence in our database. Whenever we receive a new utterance its sentence vector is likewise



(a) Example 1 (b) Example 2 (c) Example 3

Figure 6: Three WikiHow flow graphs, showing different styles of conversational interaction. Nodes are responder/states invoked by the bot. Each edge label describes frequency of transition.



(a) First 50 sessions from Dec 1st (b) Latest 50 sessions before March 20th

Figure 7: Composite conversational flow graphs using 50 sessions from the first and latest periods. The first set starts Dec 1st, the last ends on Mar 20th. Note the increase in session complexity.

computed, followed by a maximum embedding similarity search. If the highest similarity score exceeds a preset threshold, we treat the utterance as a harmful one.

3.2 Domain Classifier

At every point of the conversation, the taskbot needs to identify the topic being referred to by the user. We used a three-way classifier, where we focus on *recipe*, *DIY* and *others* as target classes. We used 1500 utterances from the user logs, performed deduplication and applied filtering. This gave a set of 500 utterances. Two annotators annotated athe data. An adjudication was used, with a third team member resolving conflicts. Finally, we use a lightweight and robust DIET classifier[2] to perform the three-way intent detection task.

3.3 Food Name Extractor

Users express recipe entities in many different ways; the taskbot needs to determine the target food. Existing toolkits like SpaCy[4] are not sufficiently accurate for extracting food names. We used a comprehensive database from https://fdc.nal.usda.gov/download-datasets.html. We preprocessed the data to clean up the database. From user logs, we created a set of templates to represent the user utterances asking for recipes. We constructed a training dataset using the downloaded database along with the templates. A new SpaCy NER model was trained on the training dataset. On our manually-annotated test set, our NER model retrieves 91% of food names correctly.

3.4 In-conversation QA

Users will ask factual questions about the recipe/DIY tasks during the conversation. For example, a user might ask *"What should I put the tomatoes in?"* to determine which container to use. A QA model can answer questions like this one.

Training QA models on the recipe/DIY tasks is not a scalable solution since we would need to fine tune the model for every new topic that the taskbot might be used to support in the future. We therefore prefer to train and deploy a few-shot or a zero-shot QA model.

We generated synthetic data from logs to have self-supervised training, using the following:
(i). Mask part of a recipe/DIY task as the answer and use whole procedure content as context.
(ii). Generate questions using a T5 model tuned on SQUAD and COCO by prompting the model
We fine tuned and deployed a T5 model[5] that was pretrained on the SQuAD dataset[6].

We also found that common sense reasoning is required for a QA capability. Users will ask questions for without answers in the recipe/task itself. For example, a user might ask *"Is this recipe a vegan recipe"*. If this information is not attached to the model, some kind of reasoning based on world knowledge is needed.

3.5 Automated Testing of Dialogues

In addition to common software unit tests, the integration testing framework for the taskbot had to address the following: maintaining context across dialogue turns and supporting different responses for the same dialogue acts.

We developed an automated testing framework using the existing `\stylesc{cobot} local-test` to allow us to test processing of dialogue acts, intents, entities and dialogue states needed for successful dialogues. This framework also allows us to perform A/B tests for the taskbot's responses to same dialogue acts.

4 Dialogue Management

Our dialogue management system is composed of two essential parts: a selection strategy and response generators. Two factors are considered: 1) the user's intent 2) the taskbot's current context. The selecting strategy relies more on the context of the taskbot, while the response generators make more use of the user's intent. For instance, if the internal state shows that the user is in a sequence of recipe instructions, the selecting strategy will choose the corresponding response generator for reading instructions. And if we further classify the user intent to be "show next instruction," the logic inside the response generator produces the appropriate instruction.

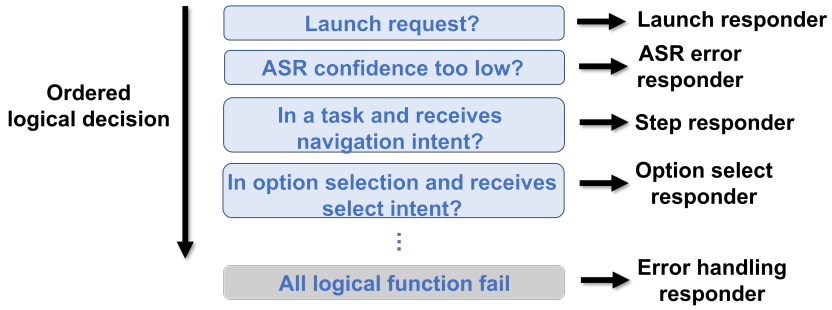


Figure 8: Overview of our implementation of selecting strategy, which is one of the core components of our dialogue management logic.

As shown in Figure 8, we used a sequence of ordered logical functions as the selecting strategy. A function ca return a response generator; if it doesn't the operation continue to the next logical function. This design takes care of priorities per context. For instance, a launch request should always be directed to the launch response generator regardless of the input (so we placed a function returning the launch response generator at the beginning of the sequence).

A dialogue manager must handle unexpected inputs that do not trigger an existing logical function. A response to unexpected inputs should provide: 1. help messages to navigate through the current context, or 2. an indication that the user's intent is only partially understood. The latter is more difficult but it is a crucial component for building trust (when users feel like the taskbot understands their intents). The former one, while easier to implement, still relies on the ability of our bot to monitor the current context. An unexpected input during the option selection phase means that the bot can ask whether the user is selecting options or asking for a new recipe. The bot needs awareness of the context to generate subsequent turns. For example, when the user says "repeat", the user likely does not want the taskbot to repeat the help message, but actually the prior instruction. State transparency (Section 2 is a key attribute of dialogue.

Based on our log analyses (Section 2.1), we realized the need for a task overview, as most users tend to be browsers. We added an option to view the WikiHow DIY task summary. Some dialogue management was subtle and non-technical, being at the level of raw taskbot utterances. For example, we went from "say which recipe/task you want" to "say which NUMBER recipe/task you want." The first utterance encouraged users to select a recipe or a DIY project by its name, as listed in the search results. But this made way for pernicious ASR errors, where Alexa failed to pick up on various pronunciations of recipe names or DIY titles. The second utterance encouraged users to say a number between one and ten, which Alexa's ASR could easily recognize.

5 Additional Capabilities

Although the core functionality of a taskbot is its ability to understand the user's intent and select an appropriate action, a number of distinct sub-systems are needed for this functionality to work. In this section, we discuss several such capabilities. These functions are not parts of the main dialogue flow; instead, they are extensions of the taskbot's ability to deal with a wider variety of user inputs.

5.1 Re-ranking Search Results

User queries need to get the most relevant results; users leave low ratings when they cannot find what they want. Worse, they may lose trust in the taskbot if they get irrelevant results. While the WikiHow API can be tuned with proper elastic search parameters, the WholeFoods API does not. To further tune the retrieved results, we adopted additional stages of filtering and re-ranking. First, we compute the fuzzy match of the search results and the user query. We remove stop words and prefix phrases (e.g. "how to" or "teach me") and set a threshold (based on empirical observations) to filter out low scores. Second, we re-rank the result based on the scores and on recipe popularity.

The aforementioned approach worked well enough. Though not deployed, we also experimented with a search result re-ranking system that turned queries and results into BERT embeddings, which

approximated the semantics of queries and results. In this approach, results were re-ranked such that the highest rated results were those with the greatest vector similarities to the search query. This approach was 40% more accurate than the base WikiHow API, determined by how close it placed the best search result (subjective) at first place in the list (evaluated over 100 unique WikiHow queries that past users had tried). The disadvantage of this (non-deployed) approach was greater computational overhead. Sometimes the delay was noticeable, during testing.

5.2 Multi-modal Design

Unique to this year's competition was the need to accommodate screen devices. While the user logs show that only 20% of the users accessed the taskbot via a screen device, we believe these need a separate dialogue flow. Some information is better presented visually, which has a much higher information flux then that of the audio modality. For exaple, it allows users to see all ingredients at once instead of one-by-one. For screen devices, we displayed steps and ingredients on the screen and only provided minimal instructions for the audio output. The user can still say "start reading" and "stop reading" to switch between verbal mode and screen mode. The user interface made navigation easy; almost all voice navigational commands could be achieved using a touchscreen.

5.3 Advanced Search and Recommendation

In the simplest types of recipe queries, users typically name a recipe and the system returns the results. For example: "Show me recipes for chicken salad." Accordingly, our initial approach involved setting the "dishname" attribute in our search. However, this approach has several problems: a) If the recipe name is unusual, either be no matches are returned or too few results; b) If many results are returned, the user will need to browse a long list; c) Users are assumed to have a specific recipe in mind; if they don't, they may still get results, but not what they intended.

In the first case, the main issue was having users speak the complete recipe name. This can create at least two scenarios: 1) A sufficient number of relevant recipes were found, or 2) Too few or no results were found. In the first case, everything is fine. In the second problematic case, one approach is to allow users to look up recipes not just by name, but by other attributes as well. For example, the user may query recipes that contain certain ingredients ("recipes with chocolate"), or ask for recipes of a certain difficulty ("easy recipes"). Such an approach allows the user to be more vague in their requests but still get a sufficient number of results, which can then be further narrowed down.

However, too many recipes can be returned for extremely common recipes like "tuna salad". Additionally, many of these may have extremely similar names ("Easy Tuna Salad", "Tuna Salad - Easy", etc.). In this case, it may be difficult for the user to compare these results and pick the one best suited to their needs. We allow users to filter and sort the returned recipes based on rating, cooking time, number of servings, number and name of ingredients, number of steps, and calories. This gives the user according to they intent.

Lastly, it is possible that users may not always have a specific recipe in mind when they use the system. This makes it necessary to have a form of recommendation system that will provide users a few options to start with, and then modify their search accordingly. We now briefly describe the design of this "Advanced Search and Recommendation" feature.

Advanced Search Design: Our workflow involves four steps: a) Parse the query for ingredients and filters (time, servings, etc.); b) Query the WholeFoods API using a subset of the keywords; c) Use the remaining keywords to further filter the results; d) Sort and display the results (by the attribute specified by the user, or rating by default). The user can make follow-up queries to fine-tune by adding more filters, or sorting the results differently.

Recommendation Design: After the parsing step of the previous workflow, if no keywords were obtained, we use two types of recommendation: a) across-session recommendation: used for providing an initial set of recommendations based on all past sessions; b) within-session recommendation: used to update the list of recommendations based on user activity in the current session.

Initially, we provide the user with a randomized set of 10 results taken from a collection of the most popular recipes (this list is updated after every user session). The recipes in this database are stored according to their recency. A higher preference is given to more recent recipes while

adding some random recipes. Once the users look through these 10 recipes, they can ask for more recommendations, or ask about a specific one. In the latter case, they may select the recipe to learn more about it. If they decide against making the dish, and ask for more recommendations, we can use the information regarding the viewed recipes (ingredients, keywords, cuisines, etc.) to tailor the next batch of recommendations.

Application to the WhikiHow Domain: The attributes of the WholeFoods dataset are much richer (for search) than that of the WikiHow. Accordingly, we only match on task names. While we still use filtering and sorting capabilities for the DIY results, there are fewer options than the recipe counterpart. We also don't have within-session recommendations for DIY since tasks can have varied names and, unlike recipes, tasks with similar names may not necessarily be similar. Thus we always get a new randomized batch of recommendations when the user asks for DIY recommendations.

6 Learnings and Insights

Exploration vs Task Completion Initially, we had the goal of high task-completion rate (i.e highest number of tasks that reach the final stage), we observed that most real world users interacted with the bot in an exploratory way. Users spent more time exploring what recipes they could cook and what DIY tasks they can take up. Our initial assumption that incoming users would already know what they wanted. We may have provided a better user experience if we had focused sooner on browsing and good search - recommendations and ranking systems.

Intent Recognition While it might seem like a simple classification task, real-world users tend to interact with the bot in many unexpected ways. It's a non-trivial to identify in-domain requests while filtering out-of-domain utterances. Our Knowledge Graph-based out-of-domain classifier required constant updates as we encountered novel user inputs. Even after identifying an utterance as in-domain, disambiguating intents was difficult. For example, *How do I stop my headache?* is both a WikiHow task as well as a sensitive medical topic. Or, *How do I make a soup to relieve high blood pressure?* triggers all intents (Medical, Recipe as well as DIY). We noticed that implementing hard-coded decision rules were the only viable approach.

Conversational UX A great user experience was a key goal: a taskbot that enables the user to quickly complete the task using the clearest instructions. For example, users needed to know ingredients/tools required for a task. Voicing each one at the start of a task becomes tedious. In contrast the (less-used) Show version could display all ingredients on the screen.

Dialogue State Traversal We under-estimated the complexity of traversals that a real-world user expects even out of a simple linear task. Real world users tend to traverse the dialogue space in multiple non-linear ways such as looking ahead by several steps without actually following any of it and then coming back to the current step, replacing certain steps with alternatives, in-recipe QA, etc. Using zero-shot in-recipe QA neural models (see above) to answer arbitrary questions.

7 Conclusion

The primary challenge in designing Tartan was the elusive conversation structure which did not follow a linear form (recipe/wikihow steps). We approached conversation structure by careful log analysis and conversation graphs, which in turn provide a critical understanding of non-linear dialogue structure. Several service modules (domain, food name, QA, and offensive classifiers) support dialogue structure parsing. We combine a traditional task-oriented framework with modern NN-based models to tackle non-linear dialogue. We believe that our technical approach is a first step towards building more intelligent, dynamic taskbots.

Acknowledgement

We would like to acknowledge the help from Amazon Alexa Prize team for their financial and technical support.

References

[1] Dan Bohus and Alexander I. Rudnicky. The ravenclaw dialog management framework: Architecture and systems. *Computer Speech & Language*, 23(3):332–361, 2009.

[2] Tanja Bunk, Daksh Varshneya, Vladimir Vlasov, and Alan Nichol. DIET: lightweight language understanding for dialogue systems. *CoRR*, abs/2004.09936, 2020.

[3] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[4] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. 2017.

[5] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[6] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.

[7] Yichi Zhang, Zhijian Ou, Huixin Wang, and Junlan Feng. A probabilistic end-to-end task-oriented dialog model with latent belief states towards semi-supervised learning, 2020.

[8] Jeffrey Zhao, Raghav Gupta, Yuan Cao, Dian Yu, Mingqiu Wang, Harrison Lee, Abhinav Rastogi, Izhak Shafran, and Yonghui Wu. Description-driven task-oriented dialog modeling, 2022.