QuakerBot: A Household Dialog System Powered by Large Language Models

Artemis Panagopoulou Manni Arora Li Zhang Dimitri Cugini Weiqiu You Yue Yang Liyang Zhou Yuxuan Wang Zhaoyi Hou Alyssa Hwang Lara Martin Sherry Shi Chris Callison-Burch Mark Yatskar

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
`{artemisp,manni,zharry,dameech,weiqiuy,yueyang1,zhliyang,wangy49,joeyhou,ahwang16,laramar,shershi,ccb,myatskar}@seas.upenn.edu`

Abstract

We describe QuakerBot, a dialog system that helps users with household tasks and a participant in the Alexa Prize TaskBot Challenge. QuakerBot can process a variety of user requests, search for instructions from web resources such as wikiHow or Whole Foods Market recipes, answer related questions, and so on. Its components simultaneously consist of large language models with an impressive few-shot performance, and rule-based models with robust service.

1 Introduction

Dialog systems, agents that can converse with users, have long been a significant point of interest in natural language processing (NLP) (Bobrow et al., 1977). With the advance of massive, pre-trained language models, large-scale and open-domain dialog systems have also seen rapid progress (Miller et al., 2017; Zhang et al., 2020e). In contrast with chatbots, *task-oriented* dialog systems are those that help users with specific tasks, such as booking tickets (Zhang et al., 2020f).

With the universal use of digital assistants such as Alexa, Siri, Google Home, Cortana, etc. and a plethora of online know-how resources, there has been increasing work on the domain of complex household tasks involving a series of steps (Zhou et al., 2019; Zhang et al., 2020a,d). However, there is still a considerable gap between such research and real-life applications, as there has been little work of bridging dialog systems and the knowledge of such complex tasks.

The Alexa Prize TaskBot Challenge[1] fills in this gap. It is a competition where university teams develop dialog systems that assist customers in completing household tasks, such as cooking and home improvement. The participant dialog systems are invoked via an Alexa Skill and can be accessed by all Alexa users in the United States. At the end of each conversation, the user provides an integral rating on the scale of 1 to 5. Figure 2 shows an illustrative, imaginary conversation.

We present QuakerBot, a participating dialog system in the competition. QuakerBot consists of a mixture of rule-based and neural components, thus providing flexible and robust service. Our dialog state manager and some simple components are rule-based and deterministic for utmost stability. However, the majority of the rest of our components are powered by neural language models, which can effectively handle considerable variance of user utterances, while requiring minimal labeled training data. In addition to developing a production-ready dialog system that helps with household

[1] `https://www.amazon.science/alexa-prize/taskbot-challenge`

tasks, we also advance NLP techniques such as intent detection, slot filling, question answering, document retrieval, and so on, in an applied and practical setting.

The rest of this report is structured as follows. §2 provides a high-level overview of the architecture of QuakerBot. §3 enumerates the natural language understanding (NLU) components to extract pertinent information from user utterances. §4 enumerates the responders that output appropriate agent utterances depending on scenarios. §5 introduces our automatic testing framework by leveraging language models to generate synthetic dialogs.

2 System Overview

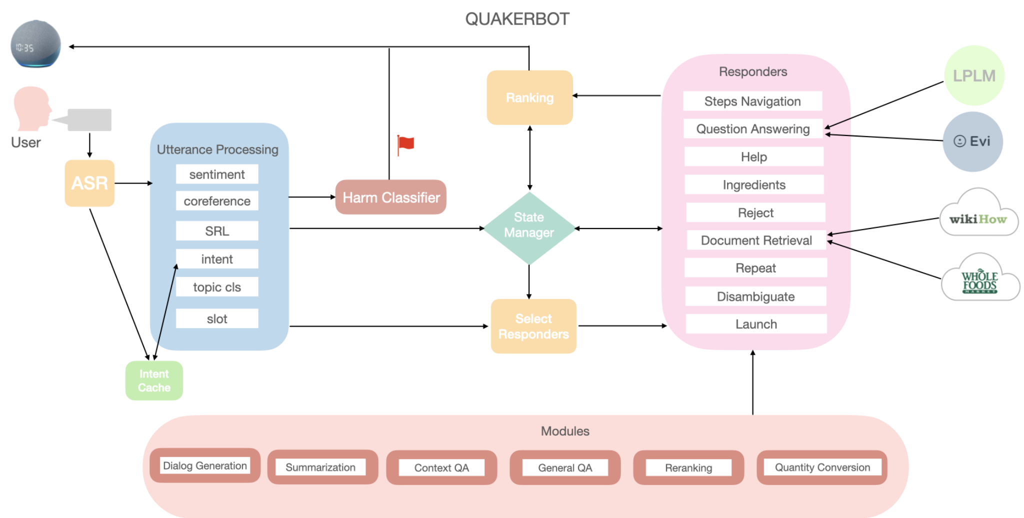

Figure 1: Overview of the QuakerBot's architecture.

Figure 1 depicts QuakerBot's architecture. When the user speaks, the utterance is parsed by the Amazon automatic speech recognition (ASR) module into text, which is then parsed by an NLP pipeline consisting of sentiment analysis (§3.4), co reference resolution, semantic role labeling, intent detection (§3.1), topic classification, and slot filling (§3.2). In order to speed up QuakerBot's responses, we also cache the intents of the utterances. If a future utterance is identical to a seen one, the cached intent is automatically used. Once the NLP pipeline is completed, the utterance is passed through a harm classification module that flags the utterance if it is inappropriate (§3.3) and re-prompts the user for a different request.

The outputs of the NLP pipeline are stored in the state manager, and are used by a selecting strategy to determine the potential responders that would be appropriate to elicit. The candidate responders are then filtered by the reranking strategy. Both the selecting and reranking strategy heavily rely on the intent and the current place in the flow of the conversation. The key difference between the two strategies is that the reranking strategy can take advantage of the information available in the actual outputs from the responders.

An example conversation can be found in Figure 2, with each dialog turn annotated by the user's intent and the elicited responder.

3 Utterance Processing

After a user speaks, the ASR system is run, transcribing the speech into text. We then run a suite of utterance processing components on the textual utterance.

#	Utterance	Comments
1	**User:** let's work together **Bot:** Hi, this is an Alexa Prize Taskbot. Try asking me how to do a task or search for a specific recipe.	Intent: Launch Responder: Launch Responder
2	**User:** How to make chocolate cake? **Bot:** Here's what I found in Whole Foods Market for chocolate cake. The top results are Chocolate Cake, and Vegan Chocolate Cake. You can select an option or say, show more, for more recipes.	Intent: Getting Instructions Responder: Whole Foods Responder Comments: WikiHow Responder is selected if the task is a home improvement or any non recipe task
3	**User:** First one **Bot:** This recipe has 4.4 stars and can be completed in 11 steps. Do you want to continue? You can say go back to hear the options.	Intent: Option Responder: Steps Responder Comments: 'go back' takes the state back to step 2 and the bot responds with more options.
4	**User:** Continue **Bot:** Chocolate Cake sounds yum! Quick heads up, this recipe has a total of 11 steps and 11 ingredients. Before we get started, Please be careful when using any tools or equipment. Remember, safety first! Let's start with the ingredients. The first ingredient is: 1 ¾ cups all purpose flour.	Intent: Acknowledge Responder: Steps Responder Comments: The user can say 'list all ingredients' to listen to all ingredients at once.
5	**User:** Next ingredient **Bot:** 2 cups sugar	Intent: Next Responder: Steps Responder Comments: At this point the user can also ask the user to repeat the ingredient
6	**User:** Start Cooking **Bot:** Step 1 of 11 Preheat oven to 350 degrees	Intent: Start Cooking Responder: Steps Responder Comments: The user can start cooking at any point to start with the recipe steps
7	**User:** What should be the oven temperature again? **Bot:** 350 degrees. Do you want to continue?	Intent: Step Inquiry Responder: Question Answer Responder
8	**User:** Next step **Bot:** Step 2 of 11 Prepare two 9-inch cake pans by rubbing with butter, sprinkling with flour and tapping out extra	Intent: Next Responder: Steps Responder
9	**User:** How much flour do I need? **Bot:** You need 1 ¾ cups all purpose flour Do you want to continue?	Intent: Step Inquiry Responder: Question Answer Responder
10	**User:** Okay let's continue **Bot:**Step 3 of 11 . In a large bowl combine flour, sugar, cocoa, baking soda, baking powder and salt	Intent: Next Responder: Steps Responder
	User: How to make tomato soup? **Bot:** Sorry, I cannot help you with the new task. Let's finish the current task first. Do you want to continue?	Intent: Getting Instructions Responder: Switch Tasks Responder Comments: If the user has already started with a task, the bot does not allow user to switch tasks.
11	**User:** Go to step 11 **Bot:** This is the last step. Remove cake from pans and cool on racks. When you're finished you can say complete to finish the task, or say previous to go back to the previous step.	Intent: Getting Instructions Responder: Steps Responder
12	**User:** Complete **Bot:** Nice job! It was a pleasure working with you. I hope you enjoy the recipe!	Intent: Next Responder: Steps Responder
13	**User:** Stop **Bot:** Asks user for rating.	Intent: Stop Responder: None

Figure 2: An imaginary example conversation that demonstrates the conversation flow of QuakerBot.

3.1 Intent Detection

It is paramount to accurately determine the intent of an utterance, so that the dialog system can respond appropriately. Such is the task of intent detection. For some dialog systems, the scope of the conversation is limited, and so is the set of intents. For example, a virtual alarm clock may only need to handle several static intents such as setting, canceling, and changing alarms. This is not the case for QuakerBot, which needs to handle a variety of intents. Moreover, the set of intents is dynamically growing as more functions are added. Traditional supervised learning methods are likely to fail in this scenario, since data annotation is unlikely to scale. Hence, *few-shot* and *open-domain intent* detection (Xia et al., 2018; Zhang et al., 2020b,c) is imperative. At the time of writing, QuakerBot supports 20 intents. We treat intent detection as a sentence classification problem, where the input is an utterance sentence and the output is one of the intents. For each intent, the team members write down some corresponding utterances as annotated data. The list of intents, examples and statistics are shown in Table 1.

In QuakerBot, we use a large pretrained language model (LPLM) that has achieved state-of-the-art in many few-shot learning tasks, to perform intent detection. The LPLM we used has two paradigms for training and inference: prompting and finetuning.

Intent	Size	Example Utterance	Intent	Size	Example Utterance
Getting instructions	53	make tomato soup	List	14	add carrots to the list
Acknowle-dgement	48	sure	Restriction	20	I don't have cabbage
Denial	40	I don't think so	Start cooking	7	begin cooking
Option	70	the first one	Next	14	what's next
None of the above	15	I don't want any of those	Previous	8	go back
Show more	19	got something else?	Stop	8	I'm done
Step inquiry	91	how do I do that exactly	Switch task	5	help with something else
General inquiry	13	what's a socket wrench?	Help	8	what can you do
Repeat	29	say it again	Ingredient Quantity	19	how many apples?
Timer	23	set a five minute timer	Substitute	11	can I use celery instead?

Table 1: The list of intents, the number of annotated examples for each of them, and some example utterances.

In the prompting paradigm, the input to the LPLM is some texts that describe what the model is supposed do, optionally accompanied by some training examples(aka the *shots* in few-shot learning). We use an intuitive prompt shown in Table 2. Since the LPLM only allows up to 2048 tokens shared between the prompt and response, we randomly sample 110 out of 413 examples from the train split and apply them to the prompt. We use the default hyperparameters: temperature as 0.7, top P as 1, stop sequence as the new line character, and no frequency penalty or presence penalty. We set the maximum of response tokens to 5 which exceeds the length of any of our intent names. As there is no mechanism to ensure that the model must output one of the intent names, we iteratively check for exact match and sub-string match, and return error if there is no match.

If someone says "help me make chicken", their intent is "Getting instructions".
If someone says "all right", their intent is "Acknowledgement".
If someone says "the first recipe", their intent is "Option".
If someone says "[input utterance]", their intent is* "[output intent]".

Table 2: An illustrative example of the prompt used for LPLM intent detection. Each line consists of an example in a fixed, pre-determined template. The asterisk * marks the end of the prompt and the beginning of model completion.

A viable alternative is LPLM finetuning, where there is no prompt. The model learns by training on labeled examples, and makes predictions based on a short query. Finetuning places no limit on the number of training examples and costs a lot less. The format of the data for finetuning is shown in Table 3.

help me make chicken	\n\n#\n\n	Getting instructions
all right	\n\n#\n\n	Acknowledgement
the first recipe	\n\n#\n\n	Option

Table 3: An illustrative example of the data at finetuning time and at inference time for LPLM intent detection. Each line consists of a training example in a fixed, pre-determined template. We use the customary sequence '\n\n#\n\n' as the separator.

We randomly split these utterances into 8:2 train and test splits. A validation set is not needed as we do not systematically tune any hyperparameter. We make the train split available to the models, but hold out the test split for evaluation. Additionally, we gather some examples from errors of intent detection from the conversation logs throughout the competition. This dataset contains realistic user utterances and their annotated intents.

The performance of our intent detection models on our test set is shown in Table 4. On our test split, LPLM finetuning leads to the best performance, greatly outperforming the prompting paradigm. At the time of writing, QuakerBot uses finetuned LPLM for intent detection. For utmost robustness, the output of the model is used in conjunction with a set of rules that deal with simple and known cases. We omit the details here.

Model	prompt	finetuning
Accuracy on test split	.702	**.856**
Accuracy on log errors	.400	**.510**

Table 4: Models' accuracy of classifying intents on our test set. We compare LPLM prompting and LPLM finetuning with just the seed data (train split), just the augmented data, and the concatenation of the two. The best performance is in bold.

3.2 Slot Filling

Some intents come with "slots", required information that is necessary for responding. For example, if a user says "I want to make tomato soup", the system should know not only that the user is "getting instructions", but also that "tomato soup" is the subject of the instructions. Such is the task of slot filling. While many existing methods jointly tackle intent detection and slot filling (Goo et al., 2018; Wu et al., 2020), we perform slot filling independently as it only has two primary use cases in QuakerBot: getting the task or the dish.

Since slot filling presents the challenge of few-shot learning in our scenario, similar to intent dection, we also leverage LPLM .Concretely, we use prompting, because empirically a short prompt with dozens of training examples is sufficient to lead to good performance. We engineer two prompts in a similar fashion for getting the task and getting the dish, as shown in Table 5.

If someone says "i am trying to hang a painting", they want to "hang a painting".
If someone says "how should I fix my kitchen sink", they want to "fix my kitchen sink".
If someone says "teach me how to make a vegan dinner", they want to make "vegan dinner".
If someone says "meat sauce sounds good", they want to make "meat sauce".
If someone says "[input utterance]", they want to make* "[output slot filling]".

Table 5: An illustrative example of the prompt used for LPLM slot filling. Each line consists of an example in a fixed, pre-determined template. The asterisk * marks the end of the prompt and the beginning of model completion.

This simple approach works well empirically during the competition, so we do not provide experimental evaluation. As before, the output of the model is used in conjunction with a set of rules that deal with simple and known cases. We omit the details here.

3.3 Harm Classification

Our harm classifier rejects inappropriate and unsupported task requests. Table 6 contains examples for each type of task requests.

To tackle the task, we collect and annotate a dataset from wikiHow's over 100K tasks. We use a human-in-the-loop approach to first automatically label the task, and then manually validate the labels. First, we use string-matching based on some keywords provided by Amazon[2] to acquire silver label for all tasks. We then split these examples into train, development, and test sets. To ensure annotation quality of the test set, we then manually validate all test examples, so that they all have gold labels. This way, the test set can be used to evaluate models' ability to accurately label all wikiHow tasks.

[2]For legal reasons, we are unable to share of display them publicly.

Medical Information Request [MED]	Legal Advice Request [LEG]
how to relieve stuffy nose in toodlers	how to report a mortgage fraud
how to improve your health with coconut water	how to schedule a court hearing
how to make your teeth straight	how to file for a divorce
how to cure a wound in your cat	how to file for bankruptcy
how to relieve a headache	how to defend your company in court
Harm to property [HARM-P]	**Unsupported tasks [UNS]**
how to measure a roof	how to know if a guy wants to kiss you
how to charge my electric car	how to ignore your feelings for someone
how to avoid false alarms with your smoke alarm	how to buy firearms online legally
how to make an engine from a battery wire and a magnet	how to become a good muslim girl
how to repair a leaking roof	how to defend pro life beliefs
Financial Advice Request [FIN]	**Harm to human [HARM-H]**
how to invest a small amount of money online	how to get a tatoo
how to apply for a credit card	how to make a spear
how to invest in bitcoin	how to make a bomb out of household items
how to know when to sell a stock	how to buy a chainsaw
how to pay for college without going broke	how to set up a campfire
Permitted Tasks [GOOD]	
How to sell comics	
How to paint a wall	
How often to water my cactus	
How to choose the right dinnerware	

Table 6: Examples of task requests per class of harm classification.

Specifically, 45 graduate students from a U.S. university each annotates up to 500 examples. In total, we end of with 13,306 task labels.

Class	Train	Dev	Test	Agreement	# Annotations
HARM-H	588	39	37	22.4%	98
HARM-P	521	36	44	25.3%	79
UNS	11493	722	646	43.4%	1506
MED	9277	691	623	47.1%	454
LEG	2238	130	174	44.9%	1862
FIN	2581	156	152	42.4%	559
GOOD	73943	4161	3957	54.0%	15022

Table 7: Statistics of our harm classification dataset.

In Table 7, we report the data statistics and annotation agreement per class, as well as total annotation agreement. We observe that the annotation agreement for the task is relatively low[3], potentially because the definitions of the classes separation are subject to ambiguity and subjectivity. The annotators were provided with the examples in table 6 and asked to annotate new examples from the WikiHow pool. Most annotation disagreement is between [GOOD] and and another label. For example, ‘how to eat a nutritious diet’ can be classifier as [GOOD] but also as [MED]. On the test set, we observe that a significant amount of tasks whose gold labels are [GOOD] are misclassified by the keyword matching approach. To improve the performance of automatic labeling, we propose to instead label fine-grained task categories (e.g., caring for pets, jogging) and apply the category labels to all tasks within. This approach is based upon the intuition that most tasks would share the same labels within a category that is fine-grained enough. For this reason we collect 4,543 fine-grained labels for each wikiHow task from TaskHierarchy138K[4], a resource with annotated, fine-grained categories of wikiHow tasks. We then manually annotate each category by labeling 10 examples from that category and taking a majority label. While not ideal, this approach outperforms the keyword method. On the manually annotated test set, we observe classifying by categories brings about a 89% accuracy, compared to 68% using the keyword approach. We use this approach to label the train and development sets to be later used by supervised models.

[3]The dataset will not be released as is, instead we will leave it as future work to gather human annotations for all the data points and provide a higher quality dataset

[4]`https://usercontext.github.io/TaskHierarchy138K/`

We experiment with three different models, taking a task request utterance as input.
Keyword Matching. If an Amazon keyword is contained in the sentence, we classify it as the class that the keyword corresponds to.
BERT (Devlin et al., 2019). We perform a seven-way classification task using a BERT-large model finetuned on our training set. Note we include only use a subset of GOOD tasks in each iteration to counter the effects of unbalanced training data.
BART (Lewis et al., 2020) **Textual Entailment (TE)**. We follow Yin et al. (2019) to recast the classification task into a TE task of classifying whether a hypothesis sentence is entailed in a premise sentence. In this case, we set the premise to be a task request such as "How to make a bomb" and provide as hypothesis "This sentence is about {label}", where {label} can take values: *legal matters*, *health*, *financial advice*, *harmful behavior*, *property harm*, or *a household task, a recipe, or arts and crafts*. We finetune a BART model pretrained on MultiNLI (Williams et al., 2017) on our training set.

On the test set, apart from the standard accuracy and F1 score, we use Matthews Correlation Coefficient (MCC) (Matthews, 1975) as the evaluation metric which computes the correlation coefficient between ground truth and predicted classifications. The values range from +1 to -1, where +1 is a perfect prediction, 0 is a random prediction, and -1 is an inverse prediction. The results are shown in Table 8. Across all metrics, BART achieves the best performance with up to 85% accuracy, outperforming BERT and Keyword Matching.

Model	accuracy	f1-score (weighted)	recall (weighted)	MCC
Keywords	0.676	0.681	0.677	0.321
BERT	0.761	0.700	0.761	0.397
BART TE	0.854	0.851	0.854	0.689

Table 8: Model performance on the test set of our harm classification dataset.

3.4 Sentiment

For sentiment classification, we use DistilBERT (Sanh et al., 2019) finetuned on SST-2 (Socher et al., 2013), a standard sentiment classification dataset. The sentiment is used to evaluate the user feedback when asking about their progress in a task, which is a part of the `Navigation Responder` (§4.1). However, upon feedback that this interrupts the step navigation flow, it is disabled in later versions.

4 Responders

Once QuakerBot detects the intent of an utterance, a corresponding responder is run to generate our system's utterance. Below are a selection of our primary responders.

4.1 Utility

The `Launch Responder` handles the initial state of the skill, which is adapted on whether the user is returning or not. In addition, it handles the preset options on multimodal devices, as well as the special options on the headless devices. We use flags to enable or not different holiday specials, as well as take into account the time of the device to suggest lunch and dinner special recipes. The launch responder is only triggered in the beginning of a session and is never defaulted to after the initial launch request.

The `Help Responder` suggests standard utterances to users so that they can make better use of QuakerBot. It is an adaptable responder that takes into account the current state and only suggests applicable utterances at that state. Also, we default to the `Help Responder` when an utterance is not handled by the skill, in which case the response will be "I heard {unhandled utterance} but I do not have a response for you right now, you can say help to see the available options."

The `Repeat Responder` repeats the previous utterance while maintaining the current state so that the users can continue from a repeated statement without interruption on the flow.

The `Reject Responder` adapts to the current state and is triggered by utterances expressing negation. For example, we provide users with the option to confirm a task before starting it, and if they express denial, the reject responder will default to showing the available wikiHow or Whole Food options again. In addition, we often ask users if they want to continue with a task, if they interrupt it in some way, either through questions or an unparsable or offensive utterance, they have the option to say no. In this case the reject responder will end the session and invite the users to restart the skill if they want to start a new task. Another use case of the reject responder is when a user wants to switch the task query either before confirmation, or during the ingredient presentation in a recipe. Finally, the reject responder will be triggered on cancel requests, which end the task at any point.

Once the user starts working on a task, the `Navigation Responder` is invoked and guides the user through the task. It can either: a) jump to the previous, the next, or any step by index, or b) display and read a step. Long steps from wikiHow additionally undergo a summarization module to be condensed into shorter steps. We experiment with two summarization modules: BART and LPLM prompted with few shots. We only employ summarization on wikiHow article steps, since recipe steps tend to be more concise and contain less superfluous information. We deploy the LPLM module since it was empirically shown to perform better summarization, with less logical gaps and more coherence in the summaries.

4.2 Task Selection

When the user asks for help with a task, one of the two task responders, `wikiHow Responder` and `WholeFoods Responder`, is invoked. Each of them leverages the API provided by Amazon to first retrieve suitable candidate articles (instructions or recipes) from wikiHow or Whole Foods Market Recipes, before presenting the options to the user and handling the user's selection. The user can also ask for more options or a different task, and the responder would react accordingly. At the core of the task responders is a retrieval module that serves a straightforward objective, to provide users with the most relevant instructions given a query about how to complete a task.

We decompose the retrieval process into three steps. First, we extract keywords describing the main objective from a user utterance using the slot-filling module based on LPLM (§3.2). Second, we retrieve a small subset of related articles with simple ranking strategies. Third, we locally re-rank articles with a more sophisticated method like semantic similarity. If this approach fails for some reason, we use a simple fuzzy-matching model as a fallback plan.

Figure 3 depicts the article retrieval pipeline. For instance, suppose the user asks "My *computer* is not working, how can I *fix* it?" The slot filling model will extract the task query as "fix computer." Next, our system applies the Elasticsearch API with fuzzy-matching, a text search API provided by Amazon, to the **article title** and the **summary text** to quickly filter out a small subset of articles from the entire database that are relevant to the query. After this stage, while the titles of the filtered articles may contain keywords like *"computer"* or *"fix"*, they may not be relevant to *"fix computer"* (e.g., "How to **hack** a **computer**?"). Moreover, fuzzy matching may fail to detect paraphrases like "*set up* Christmas light" and "*hang* Christmas light". To re-rank the candidate articles, we use SentenceBERT (Reimers and Gurevych, 2019) to compute the sentence embeddings of the query and the articles (title and summaries). Then, we use cosine distance to calculate the semantic similarity between the retrieved articles and the user query to re-rank the articles. Finally, we present the top three results to the user.

We focus on the re-ranking algorithm since the control over Elasticsearch API (for initial ranking) is limited. We collect and hold out a dataset to evaluate the re-ranking algorithm. We first collect 98 queries from past conversations. Then, for each query, we use the Elasticsearch API to retrieve 15 related articles via fuzzy matching for each question. Next, we manually annotate them with binary labels: hit (helpful for the query) or a miss (unhelpful for the query).

We first use SentenceBERT to convert the concatenation of the article title and summary text to a document vector for each retrieval during the re-ranking stage. Then, we use the same model to map extracted user query to its query vector. Later, we calculate the dot product between each query-document pair and use the dot product as a measure of relevance. The higher the dot product, the more relevant a document is to that user query. Finally, the rankings are assigned accordingly.

We evaluate the re-ranking performance by Top-k Mean Reciprocal Rank (MRR@k), which measures where **the first relevant retrieval** is located within the first k results. Given N user queries, the system proposes N corresponding rankings. Then, we calculate the mean of reciprocal **rank of the first relevant** document r_i's. If the rank is greater than k, we say the system fails to retrieve for this

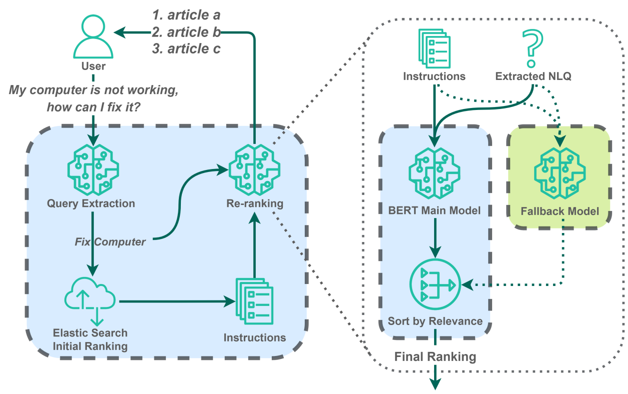

Figure 3: Our pipeline for article retrieval.

query and give it a score of zero. This truncation is controlled by the indicator function $\mathbb{1}[r_i \leq k]$. In a word, higher MRR with smaller k is better, with the maximum being one. The performance is summarized in Table 9.

Model	$k = 3$	$k = 6$	$k = 15$
No Reranking	.69	.70	.71
BM25	.65	.67	.67
SBERT	**.80**	**.80**	**.81**
Perfect	**.83**	.83	.83

Table 9: Re-rank Performance by **MRR@**k

Reranking algorithms take both article title and summary text as inputs. BM25 is another efficient keywords matching based measure of document relevance. However, due to its exact-matching nature, the reranking performance actually lowered comparing to Elasticsearch initial fuzzy matching ranking. On the other hand, we can see that the SentenceBERT-based model is very close to the perfect ranking result where relevant document is always ranked the first.

4.3 Question Answering

If the user asks a question during the conversation, the `Question Answering Responder` provides an answer. It handles five types of questions: Ingredient Questions (e.g., "How much butter do I need?"), Context Dependent Questions (e.g., "What temperature should I set the oven to?"), General Questions (e.g., "How many tablespoons in half a cup of sugar?"), Step Related Questions (e.g., "How many steps are left?"), and Article Related Questions (e.g., "How long does this recipe take?").

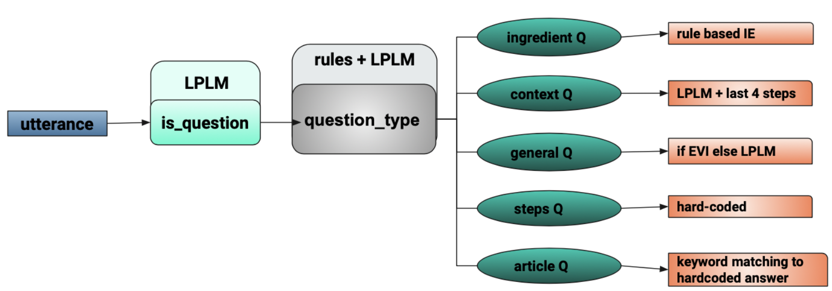

Figure 4: Our question answering pipeline.

Figure 4 depicts the question answering pipeline. If the intent of an utterance is classified as a question, the pipeline proceeds. We use a combination of rules and LPLM intent detection (§3.1) to classify the question type. In particular, we use keywords to identify if the question is an ingredient, steps, or article question, and LPLM intent detection to distinguish between general and context dependent questions. Then, we parse the question based on its type. For ingredient, steps, and article related questions, we extract the relevant information from the recipe or instructions and present it to the user. For general questions, we rely on the EVI API (a question-answering service provided by Amazon) if it has high confidence, and revert to LPLM question answering (see below for the prompt). Finally for context dependent questions we use LPLM prompting with the prompt template shown in Table 10.

goal: how to remove the lightbulb? # step: make sure the power is turned off. # context: last four steps Use the context to answer the question: #Q: how can i turn off the power? #A: *
Example Answer (without using context): #A: You can turn off the power by flipping the switch on the wall Example Answer (with context): #A: Switch the red power button to "off" in your fuse box

Table 10: An illustrative example of the prompt used for LPLM question answering.

5 Synthetic Dialogues

During the competition, our main source of error analysis is the historic conversations. However, these conversations might be too few for troubleshooting and improving QuakerBot. Hence, we pilot an experiment with automatically generating synthetic dialogues, allowing us to obtain more conversation data. To create realistic synthetic dialogues, we model single-turn interactions and utilize the LPLM to generate "user" responses to different prompt types, by finetuning the LPLM on the dialog turns in our past conversations. We thus create complete synthetic conversations by combining these single-turn dialogs.

QuakerBot: Hi, this is an Alexa Prize TaskBot. Try asking me how to do a task or search for a recipe. **LPLM** : how do i make mole sauce
QuakerBot: Sure, I can help you with Turkey Mole sauce... (instructions start) **LPLM** : next
QuakerBot: 1 tablespoon ground cinnamon. **LPLM** : can i use ground nutmeg instead?

Table 11: An example of a synthetic multi-turn dialogue.

Table 11 contains an example of a synthetic multi-turn dialog. This excerpt demonstrates the ability of the module to navigate through a conversation and even exercise more complicated behaviors that a user might exhibit such as asking clarifying questions. Synthetic dialogues like the example shown aid us in testing and improving QuakerBot, all while ensuring the privacy of user data by not directly leveraging the user data. In the future, we plan to continue improving this module by incorporating features such as human-in-the-loop and crowdsourcing components.

6 Conclusion and Future Work

We present QuakerBot, a production-ready dialog system that helps users with household tasks. The mixed use of large language models and rule-based components proves crucial to ensure both flexibility and robustness. While QuakerBot can effectively handle most utterances and situations, it sometimes cannot correctly detect some intents or retrieve some relevant instructions, two of the most common failure cases throughout the competition. Fortunately, many of our learning components including intent detection, slot filling, question answering, and document retrieval can be steadily improved by having access to more labeled data. Future work may also attempt more precise troubleshooting, by taking into consideration and reasoning about the current state of the task, instead of relying on an end-to-end black-box system.

References

Daniel G. Bobrow, Ronald M. Kaplan, Martin Kay, Donald A. Norman, Henry Thompson, and Terry Winograd. 1977. Gus, a frame-driven dialog system. *Artificial Intelligence*, 8(2):155–173.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Chih-Wen Goo, Guang Gao, Yun-Kai Hsu, Chih-Li Huo, Tsung-Chieh Chen, Keng-Wei Hsu, and Yun-Nung Chen. 2018. Slot-gated modeling for joint slot filling and intent prediction. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 753–757, New Orleans, Louisiana. Association for Computational Linguistics.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Brian W Matthews. 1975. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451.

A. H. Miller, W. Feng, A. Fisch, J. Lu, D. Batra, A. Bordes, D. Parikh, and J. Weston. 2017. Parlai: A dialog research software platform. *arXiv preprint arXiv:1705.06476*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642.

Adina Williams, Nikita Nangia, and Samuel R Bowman. 2017. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426*.

Di Wu, Liang Ding, Fan Lu, and Jian Xie. 2020. SlotRefine: A fast non-autoregressive model for joint intent detection and slot filling. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1932–1937, Online. Association for Computational Linguistics.

Congying Xia, Chenwei Zhang, Xiaohui Yan, Yi Chang, and Philip Yu. 2018. Zero-shot user intent detection via capsule neural networks. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3090–3099, Brussels, Belgium. Association for Computational Linguistics.

Wenpeng Yin, Jamaal Hay, and Dan Roth. 2019. Benchmarking zero-shot text classification: Datasets, evaluation and entailment approach. *arXiv preprint arXiv:1909.00161*.

Hongming Zhang, Muhao Chen, Haoyu Wang, Yangqiu Song, and Dan Roth. 2020a. Analogous process structure induction for sub-event sequence prediction. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1541–1550, Online. Association for Computational Linguistics.

Jianguo Zhang, Kazuma Hashimoto, Wenhao Liu, Chien-Sheng Wu, Yao Wan, Philip Yu, Richard Socher, and Caiming Xiong. 2020b. Discriminative nearest neighbor few-shot intent detection by transferring natural language inference. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5064–5082, Online. Association for Computational Linguistics.

Li Zhang, Qing Lyu, and Chris Callison-Burch. 2020c. Intent detection with WikiHow. In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*, pages 328–333, Suzhou, China. Association for Computational Linguistics.

Li Zhang, Qing Lyu, and Chris Callison-Burch. 2020d. Reasoning about goals, steps, and temporal ordering with WikiHow. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4630–4639, Online. Association for Computational Linguistics.

Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and Bill Dolan. 2020e. DIALOGPT : Large-scale generative pre-training for conversational response generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 270–278, Online. Association for Computational Linguistics.

Zheng Zhang, Ryuichi Takanobu, Qi Zhu, MinLie Huang, and XiaoYan Zhu. 2020f. Recent advances and challenges in task-oriented dialog systems. *Science China Technological Sciences*, 63(10):2011–2027.

Yilun Zhou, Julie Shah, and Steven Schockaert. 2019. Learning household task knowledge from WikiHow descriptions. In *Proceedings of the 5th Workshop on Semantic Deep Learning (SemDeep-5)*, pages 50–56, Macau, China. Association for Computational Linguistics.