Enterprise-scale conversational agents with knowledge graphs, entity refinement and dynamic conversation flows

Devin Conathan, Joseph Bockhorst, Glenn Fung
Machine Learning Research and Innovation
American Family Insurance
Madison, WI
{dconatha,jbockhor,gfung}@amfam.com

Abstract

A key barrier to the enterprise wide scaling of conversational virtual agents, beyond the significant challenges posed by natural language understanding, are that the marginal development and maintenance costs of additional content areas is high. Here we describe an approach for development of virtual agents that scales. Our approach dynamically generates conversational flows from a knowledge graph in a way such that our conversational model can be interpreted as an instance of a (dynamic) finite state machine. Importantly, our design supports a synergistic collaboration between domain experts and IT professionals using each group's core competencies. IT is responsible for designing the schema of the knowledge graph and implementation of certain key routines while the domain experts use their expertise to populate the knowledge graph. We have implemented our approach at an American insurance company using it to deploy an internally-facing question-answering virtual agent. A short video overview of our system is available at https://youtu.be/12CsG_HAMhM

1 Introduction

Enterprises are increasingly looking to use virtual agents (a.k.a. chatbots) to better serve customers and to reduce costs. Within customer support centers, for example, there is optimism virtual agents will increase efficiency and decrease costs by automating the fulfillment of simple tasks and providing answers to common questions while providing customers clear and consistent answers and around-the-clock, instantaneous availability.

Chatbots present many interesting academic challenges because they combine aspects of natural language understanding, machine learning, linguistics, communication and information science. Despite these challenges, it is surprisingly easy today to build a chatbot that can perform a few simple tasks using off-the-shelf tools. However, scaling such approaches has proven to incredibly challenging if not next to impossible.

The difficulty of enterprise wide scaling of virtual agents is not surprising when one considers that in essence any chatbot operates by translating natural language utterances into mini-programs in some formal language. Thus, even if the translation problem were solved (the focus of a large portion of the field of natural language understanding), scaling virtual agents would need thoughtful software design, development and maintenance in order to succeed. Moreover, considering that i) the system language is frequently an implicitly defined domain specific language and that ii) non-technical subject matter experts are often expected to be the primary authors of chatbot functionality, it is no surprise that the challenges of scaling virtual agents are vast.

Our main contribution in this paper is an approach to virtual agents that scales. We design our approach with two primary objectives: i) that common conversational actions, for example obtaining parameters needed to carry out user intents, be performed for new or updated virtual agents by automatically generating conversational flows from an underlying knowledge graph (KG) and ii) that non-technical subject matter experts be empowered to create and update virtual agents by simply *describing* data within their domain of expertise in the KG.

Throughout we will refer to a hypothetical *pizza ordering bot* to demonstrate concepts with simple examples. At our company, we have applied this approach to a *question answering bot*, which is introduced in more detail in Section 2.4.

2 Background

2.1 Knowledge Graphs, Entities and Types

A knowledge graph is a way of storing information that is particularly useful for virtual agents. Let $\mathcal{E}$ be a set of **entities** and $\mathcal{P}$ be a set of **properties**. A **knowledge graph** $\mathcal{K} \subset \mathcal{E} \times \mathcal{P} \times \mathcal{E}$ represents a set of facts; if $(e_1, p, e_2) \in \mathcal{K}$, then entity e_1 has property p equal to e_2 (e.g. (`Barack Obama`, `born in`, `Honolulu`)).

$\mathcal{E}$ contains both entity instances and entity types. An entity's type is expressed in the graph with a special property `type`: (`10`, `type`, `Number`), which says there's an entity `10` that's an instance of the type `Number`.

$\mathcal{E}$ usually contains basic types like `String`, `Number` and `Boolean` and custom types specific to a domain (e.g. `Pizza`). $\mathcal{K}$ includes schema information about types; for instance, (`Pizza`, `diameter`, `Number`) indicates that entities with type `Pizza` have a property `diameter` that is a `Number`. $\mathcal{K}$ also includes facts about specific entities, such as (`pizza #123`, `diameter`, `10`), which says that there exists some entity called `pizza #123` that has a `diameter` of `10`.

2.2 Intents and Entities

An **intent** is a standard term in the chatbot domain meaning *what the user is trying to accomplish*. For our approach we formalize an intent as a function that fulfills the user's need by returning an answer and/or executing some side effect. For example `GET_MENU` might be an intent that queries a pizzeria's website and returns the menu, which would be invoked if the user asked "Can I see the menu?"

Most intents need parameters in order to be fulfilled. Under our approach intents are designed such that the types of their parameters are entity types in the KG. For example the `ORDER_PIZZA` intent requires the instance of `Pizza` (with its size and toppings specified) that the user would like to order.

The task of determining the parameters of the intent, similar to the dialog-state-tracking problem (Williams et al., 2016), is a central task of virtual agents and is the focus of this paper. Since we assume intent parameters are entities, we call this process **entity refinement**.

2.3 Entity Refinement

Natural language is inherently ambiguous and especially problematic for virtual agents in domains with complex content. Often this ambiguity derives from incomplete requests (*"I'd like to order a pizza."*). In this case, an instance of the type `Pizza` is necessary to fulfill the intent, but the user has given no information about it.

We propose an approach for determining an instance of a specific type; that is, refining from the entity type (`Pizza`) to a fully-qualified instance of that entity (*a large thin-crust pizza with pepperoni and extra cheese*).

We consider this problem orthogonal to other ambiguity problems that arise from conversational agents, such as intent recognition. Determining the user's intent is usually the first step for any conversational engine, and for this paper we assume that the correct intent has been identified.

2.4 Applications to Many Domains

This entity refinement approach generalizes to many domains. At our company, we have developed an insurance agent-facing chatbot for answering common questions about our products. However, questions like *"How do I remove someone from a policy?"* have a number of possible answers. The answer to this particular question is different if the policy is an auto or property policy, or if the person being removed is the policy holder or not.

Rather than having different intents for each possible answer (`REMOVE_HOLDER_AUTO`, `REMOVE_NONHOLDER_AUTO`, ...), it is more efficient and scalable to define one intent (`REMOVE_PERSON`) and use a general entity refinement procedure to determine the needed entities: what kind of policy (auto or property) and what kind of person (policy holder or not).

2.5 Current Approaches

The standard approach to entity refinement is programming some kind of workflow that is triggered when the intent is recognized. This approach is problematic because the control flow and logic of your chatbot is now intricately linked with the content. This means that either your subject matter experts (SMEs) learn and apply software engineering principles, or your software engineers need to become subject matter experts.

Indeed, many chatbot vendors develop custom

visual programming interfaces in an attempt to give SMEs the ability to develop such workflows on their own. While visual programming offers some advantages, decreasing the learning curve for non-programmers by hiding some elements of syntax most notably, they come with a number of shortcomings, especially associated with maintenance, that may render their apparent advantages illusory. This is especially true at enterprise scale. Standard software engineering practices, for example, like version control, debugging, and unit testing are difficult if not impossible with these interfaces, and even minor software changes becomes a costly burden.

3 Our Approach

Programming languages offers a helpful prism through which to distinguish our approach to entity refinement from those described in Section 2.5. The visual programming environments for creating intent workflows that we mentioned above provide an *imperative* programming model for creating workflows. SMEs give instructions to the chatbot on *how* to respond to different inputs and change state. On the other hand, our approach uses a *declarative* paradigm where SMEs describe *what* the chatbot is trying to accomplish. This establishes a convenient separation of duties; the SMEs are responsible only for describing their data and the software engineers are responsible for implementing algorithms that operate on that data.

3.1 A Conversational Data Model

Our conversational model depends on the following components:

- A set of entities and entity types $\mathcal{E}$ and properties $\mathcal{P}$
- A knowledge graph $\mathcal{K} \subset \mathcal{E} \times \mathcal{P} \times \mathcal{E}$
- A set of n intents $\mathcal{I} = \{I_1, I_2, ..., I_n\}$
- For each intent I_i a vector E_i of entity types in $\mathcal{E}$ that defines the signature of I_i.
- A set of **refining questions** $q_t \in \mathcal{Q}$, one for each type $t \in \mathcal{E}$. Each q_t is a function, that in the common case returns a single question.

Example 3.1. Say a `Pizza`'s schema specifies a size, one topping and a kind of crust.[1] Then a conversational model for this intent would require the following refining questions:

- $q_{\texttt{Size}}$ = "What size pizza would you like?"
- $q_{\texttt{Topping}}$ = "Which topping would you like on your pizza?"
- $q_{\texttt{Crust}}$ = "Would you like thin or thick crust?"

Our conversational model has three main steps:

1. Identify the user's intent $I_k \in \mathcal{I}$.
2. Identify the parameters $\vec{e}$ of I_k. The types of the elements of $\vec{e}$ must match E_k.
3. Fulfill the intent by invoking $I_k(\vec{e})$.

This paper focuses on the second step, which we call entity refinement.

3.2 Entity Refinement Algorithm

Let I_k be the recognized intent. The goal of entity refinement is to acquire parameters for I_k by identifying a single instance of each entity type in E_k. With each intent we have a counting function $O_{\mathcal{I}_k} : \tilde{E}_k \to \mathbb{N} \cup \{\infty\}$, where $\tilde{E}_k$ is the set of all valid *partial* configurations of E_k. A partial configuration is an entity with potentially unspecified properties (e.g. a *small pizza* with its size specified but not its toppings or crust is a partial configuration of `Pizza`).

An intent's counting function tells us how many candidate parameter vectors are consistent with a given partial configuration. Each intent may have it's own bespoke counting function, but the default implementation that uses combinatorics and querying $\mathcal{K}$ suffices in most cases. The output of O can be large or even infinite (such as when an entity has an unbounded `Number` property). Our approach to refinement is to iteratively prompt the user for the parameter values until only one candidate parameter vector remains, at which point we fulfill the intent.

We use counting functions instead of simply requiring values for all properties of each parameter's entity type so that we do not unnecessarily prompt the user. A parameter may be irrelevant to the intent if some configurations are invalid. Perhaps the *thick crust* option is not available for *small* `Pizzas`, and this constraint is encoded in our graph. Then $(\texttt{small}, \texttt{pepperoni}, \texttt{NULL})$ would only match one valid pizza configuration

[1] To simplify this example, we restrict ourselves to one-topping pizzas, but this approach does generalize to many-topping pizzas by introducing the `Set` type.

and we would have no need to refine the `crust` property. Such constraints may be encoded in the KG.

We show the pseudocode of entity refinement in Algorithm 1. In our model, an entity can always be uniquely identified by specifying all of its properties (i.e. a `Pizza` entity is equivalent to a (`Size`, `Topping`, `Crust`) triple). Thus, refining its properties is equivalent to refining the `Pizza` entity itself.

The `CHOOSE` function is important to optimize because refining entities in different orders can result in a different number of total questions asked or a more natural conversation flow. The default implementation of `CHOOSE` arbitrarily selects an as yet unknown entity, but more complex heuristics, for example, using information theory to guide the selection (Bockhorst et al., 2019), may be preferred.

Algorithm 1 Pseudocode of our approach

function REFINE($\mathcal{I}, \tilde{e}$)
  $\mathcal{I}$: the intent
  $\tilde{e}$: a partial configuration of entities
  if $O_{\mathcal{I}}(\tilde{e}) = 1$ **then**
    return $\tilde{e}$
  else
    $q_t \leftarrow$ CHOOSE($\mathcal{I}, \tilde{e}$) ▷ choose which refining question to invoke
    $e_t \leftarrow$ ASKUSER(q_t)
    $\tilde{e} \leftarrow$ UPDATE($\tilde{e}, e_t$)
    return REFINE($\mathcal{I}, \tilde{e}$)
  end if
end function

Example 3.2. Here is a simple example that demonstrates our algorithm:

User: "I'd like to order a pizza."

The `ORDER_PIZZA` *intent is triggered*

$E_{\texttt{ORDER\_PIZZA}} =<$ `Pizza` $>$, *so an instance of* `Pizza` *is needed* $O_{\texttt{ORDER\_PIZZA}}(\texttt{NULL}) > 1$ *so we need to refine.*

We choose to refine the property `size`

Bot: "What size pizza would you like?"

User: "Large"

We continue to refine until $O_{\texttt{ORDER\_PIZZA}}(e) = 1$

Bot: "Which topping would you like on your pizza?"

User: "Pepperoni"

Bot: "Would you like thin or thick crust?"

User: "Thin"

$O_{\texttt{ORDER\_PIZZA}}(e) = 1$ *and we have a fully specified instance of a* `Pizza`.

We invoke the side effect associated with `ORDER_PIZZA` *and respond to the user.*

Bot: "Got it. I have ordered your large thin crust pizza with pepperoni."

4 In Practice

At our company, we have deployed a insurance agent-facing chatbot instance using this approach to answer questions about our products as is introduced in Section 2.4. The chatbot has one intent: `ANSWER_QUESTION` which takes an instance of a `Question` as a parameter. Instances of `Questions` fall into some subtype, like `CoverageQuestion` or `RentalQuestion`, each with their own schema. When a user asks a question, we apply a convolutional neural network-based sentence classifier (Kim, 2014) to classify which subtype of `Question` is being asked, and then begin the refinement process described in Algorithm 1. Our latest release contains about 370 questions and 100 question subtypes.

5 Conclusions

This paper describes a novel design for conversational virtual agents that scale. Our design emphasizes a separation of duties that empowers both domain experts and IT professionals to meaningfully contribute to development using knowledge and skills appropriate to their training. Our approach does not ask subject matter experts to apply principles from software engineering nor for programmers to become experts in problem domains.

Our approach includes a conversational model that dynamically queries a knowledge graph to generate conversation flows. Like other recent approaches to virtual agents (Larionov et al., 2018; Pichl et al., 2018), our approach can be interpreted as instance of a finite state machine, but whose states are not explicitly given, but rather are dynamically generated from the current state and the knowledge graph.

References

Joseph Bockhorst, Devin Conathan, and Glenn M. Fung. 2019. Probabilistic-logic bots for efficient evaluation of business rules using conversational interfaces. In *Proceedings of the Thirty-third Conference on Innovative Applications of Artificial Intelligence, Jan 27-Feb 1 2019, Honolulu, Hawaii*.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882.

George Larionov, Zachary Kaden, Hima Varsha Dureddy, Gabriel Bayomi Tinoco Kalejaiye, Mihir Kale, Srividya Pranavi Potharaju, Ankit Parag Shah, and Alexander I Rudnicky. 2018. Tartan: A retrieval-based socialbot powered by a dynamic finite-state machine architecture. *CoRR*, abs/1812.01260.

Jan Pichl, Petr Marek, Jakub Konrád, Martin Matulík, Hoang Long Nguyen, and Jan Sedivý. 2018. Alquist: The alexa prize socialbot. *CoRR*, abs/1804.06705.

Jason Williams, Antoine Raux, and Matthew Henderson. 2016. The dialog state tracking challenge series: A review. *Dialogue Discourse*.