Abstract

We created VitiBot, a multi-turn mixed-initiative dialog system to allow users to search for bottles of wine by either typing or saying out loud their various preferences. The system leverages the Wit.ai framework in order to extract keywords from and classify the intent of user utterances, and it queries the wine.com API as a knowledge base. We created a custom state management system to track the user’s progression through the dialog. In order to increase the autonomy of our users, we allow them to back out of or skip entirely lines of questioning which do not concern them. Our analysis shows that many of the system’s errors are due to failures in the ASR module used to transcribe spoken commands and preferences. We also show that our multi-turn, mixed-initiative system has higher levels of user satisfaction then a single-turn, user-initiative baseline model.

1 Introduction

According to the International Organization of Vine and Wine, each year around 24 billion liters of wine are consumed around the world (2012). However, despite the fact that the average adult in the US drinks more than 11 liters of wine per year, the average level of wine knowledge is relatively low. In a six question multiple choice test of wine facts, US consumers answered only 3 questions correctly on average, compared to an average score of 3.8 in Australia and 3.5 in the UK (Forbes et al., 2008).

We have therefore developed VitiBot, a multi-turn, mixed-initiative dialog system intended to help users find the best bottle of wine for them, regardless of their familiarity with wine in general. Using a combination of open-ended questions and direct prompts, VitiBot gathers as much information as possible from the user before searching the wine.com database for an appropriate bottle of wine. Because this system is intended to help average consumers, we do not assume that our users will have an answer for each of our questions and allow them to skip over questions if they choose.

The rest of this paper is organized as follows. We begin in Section 2 with a description of previous research into dialog systems. Section 3 then documents the architecture and training of VitiBot. In Section 4 we analyze our system to evaluate common sources of errors as well as user perceptions of our system. Finally, Section 5 draws some overarching conclusions about the project and presents directions for future work.

2 Background

Early attempts at dialog systems date back to the 1960s with the introduction of ELIZA from Weizenbaum (1966). The goal of ELIZA was to study natural language interactions between human and machine in a novel way at the time, which consisted of identifying keywords within a user’s statement and using those keywords to generate an appropriate response. Issues quickly become apparent, including the problem of what constitutes a keyword and how context can affect a keyword. Nevertheless, ELIZA was an early attempt to address these concerns and stands out as one of the first novel dialog system.

Mixed initiative dialog systems began emerging in the 1970s when Carbonell presented a dialog system for instruction that operated on an information network of facts, allowing it to both give and answer questions from students (1970). In conversation between two humans, each participant usu-
ally takes initiative from time to time, so we desire a similar kind of interaction between human and machine, i.e., mixed initiative. Some dialog systems that developed later on, such as the GUS system from Bobrow et al. in 1977, had factored in mixed initiative (1977). We too aimed to implement VitiBot as a mixed initiative dialog system where users may be able to provide information without an explicit prompt.

Dialog systems evolved over time, becoming more complex and complete through the years. By 2002, dialog systems such as the Why2-Atlas system by VanLehn et al. had emerged which strove to accomplish more complicated goals, i.e., physics tutoring (2002). This added complexity was driven by more complex and nuanced algorithms for natural language understanding, which allowed computers to capture the intent, certainty, and consistency of user responses. Such ideas were considered by Forbes-Riley and Litman when their system extracted both user certainty of their own response and meaning, allowing them to develop an artificial tutor that adapted better to a student’s needs (2011).

VitiBot’s primary aim is to learn the user preferences so as to make the best wine recommendation. To that end, we should ask the user follow-up questions, but as brought up in the work of Li et al. (2016), it is important to know when we should be prompting questions and when we should not. As mentioned previously, the average level of wine knowledge is relatively low, so we cannot expect the user to answer many of the questions that VitiBot will ask. To that end, when designing VitiBot we focused on questions that users may find the easiest and most important to answer. Of course, they may also still choose to skip over them.

3 Approach

The high level architecture of VitiBot can be seen in Figure 1. The details of these various elements will be described in the following subsections.

3.1 Wit.ai

First the user’s input is sent to our trained Wit.ai model. If the input is verbal, Wit first performs automatic speech recognition to convert the input to a textual format. Wit then extracts various custom entities from this transcription.

![Figure 1: The high level overview of VitiBot’s pipeline showing the various stages a user’s utterance goes through as it gets processed by the system.](image)

3.1.1 Wit Entities Used

An example of each of the three different types of entities extracted by Wit are shown in Figure 2. The simplest entities are those, like a wine’s color, which take their value from a predetermined set. In this situation, Wit can simply scan the input utterance and extract any of these keywords. We exploited this type of entity to implement our wine color selections as well as the ability to filter wines based on certain predetermined food pairing categories.

A slightly more complex entity type is the free-text/keyword entity type, such as wineLoc in Figure 2. These are entities in which Wit is given explicit knowledge about certain values the entity may take and must try to match new values to this entity based on word usage patterns. This is particularly useful for entities which could take on a large number of values or whose set of values might change over time. As mentioned, we used this type of entity both to detect the desired region that a wine should come from, as well as the wine’s varietal.

Finally, the “trait” entity class allows us to use Wit to classify utterances as a whole. These entities are particularly useful for cases in which no single word or phrase within the utterance directly represents the desired entity value. As seen in Figure 2, we used this entity type to determine the user’s intent within each dialog turn and then pro-
3.1.2 Training the Wit.ai Model

Wit.ai trains its natural language understanding models using three different data sources. First, we provided a small number of labeled examples when defining which entities it ought to be looking for. For instance, we might give Wit the input utterance “I’m looking for a red wine from France,” and tell it that for this utterance, the set of entities it should extract are intent = setQueryParam, color = red, and wineLoc = France.

Next, Wit allows you to explicitly validate examples which it has encountered during the execution of your app. This form of data collection was extremely important as the examples we collected in this way represented how users truly interacted with the system.

Finally, and perhaps most importantly, Wit leverages the labeled data from other similar apps in order to augment the training dataset. After receiving a couple labeled utterances, Wit retrieves other examples it has been trained on which are similar and finds connections between those extracted entities and the ones that you are looking for. This allows Wit to make accurate NLU predictions given only small amounts of training data per app.

3.2 Custom State Management

Once we have extracted any relevant entities from the utterance using the Wit.ai model, we then have to actually incorporate those entities into a persistent state so that they can eventually be incorporated into the wine query. We therefore created a frame with various slots which can be filled with the entity values detected from Wit. Currently, our system has slots for (and can therefore filter a user’s wine preference by) wine color, varietal, minimum price, maximum price, vintage, vineyard location, and food pairing.

3.2.1 Non-Blocking Question States

When a user first approaches VitiBot, the system is in a user-initiative state. That is, we first allow the user an opportunity to tell the system what he or she is looking for. However, as mentioned in the introduction, the typical wine consumer knows relatively little about wine and is likely to not have strong opinions about many of the slots. Based on discussions with wine consumers, we settled on “color,” “price,” and “food pairing” as the three slots that users found most important when selecting a wine. Therefore, once the user has given us their input, we transition to system-initiative and begin prompting the user if any of these three slots are unfilled.

Of course, it is entirely possible that the user actually has no preference for these slots and meant to leave them unfilled. In order to make the switch from user- to system-initiative less jarring for the user, we provide them with a means of skipping over questions about slots which they do not care about. We modeled this form of non-blocking question as shown in Figure 3. The question can either be answered with a simple affirmative answer (“Yes, I have a color preference”), a simple negative answer (“No, I don’t care about the color”), or by giving the specific answer (“I enjoy red wine”). However, only a negative answer or an answer with the color actually allows the dialog to progress past this question. A positive answer results in a more direct phrasing of the question (“Ok, which color do you prefer?”), but the dialog remains in this questioning state, as it does if the user answers with something else unexpected.

3.2.2 Food Pairing Slot

One slot we were particularly interested in incorporating into our design was the ability to filter wines based on what food they pair well with. The knowledge base backing VitiBot does contain this information, with the food pairing categories it recognizes forming a tree-structure of variable depth. A pairing is found by first choosing a general pairing type (e.g. “meat” or “seafood”), then a main ingredient (e.g. “oysters” or “tuna” within the “seafood” type), and then potentially a style for that ingredient (e.g. “tuna” has styles “seared” and “spicy,” while “oysters” has no defined styles).

In order to find the pairing for the user, our VitiBot system simply walks this pairing tree, asking an appropriate non-blocking question at each branching node of the tree. Once a leaf node is reached, the food value associated with that node is used to fill the slot. In addition, the user can end the line of questioning at any point, allowing for them to specify that they plan to pair the wine with “tuna” in general but not “seared tuna” or “spicy tuna,” for example.

However, when we started running preliminary tests on users, we found that our original implementation of the pairing slot left something to be desired. The issue was that each of the base-level
pairing types (“meat,” “poultry,” “seafood,” “pasta / grain,” “cheese,” and “dessert”) is very general, making it unlikely that a typical user would naturally start their search by specifying which of the types they were interested in. Instead, we found that users were much more likely to specify a second-level ingredient (e.g. “beef” or “pizza”). This led to frustrating interactions like the following, in which the user states an ingredient, VitiBot prompts them with the list of pairing types, they choose the appropriate one and then are immediately asked about the original ingredient they mentioned:

VitiBot: Do you plan on having a meal with the wine?
User: Yes, I’m going to have it with chicken.
VitiBot: If you are planning to have a meal with the wine, let me know which of these categories it is closest to: poultry, cheese, meat, dessert, seafood, pasta and grains.
User: Poultry.
VitiBot: Will it be chicken, duck, turkey, or none of these?

To get around this challenge, we simply changed from doing a depth one to a depth two search of the pairing tree structure. This small change greatly increased user satisfaction with VitiBot’s pairing functionality and created significantly more natural conversations with the bot.

3.3 Wine.com Knowledge Base

Once we have gathered as much information and filled as many slots as possible in the query frame, VitiBot then queries the wine.com database in order to find the bottles of wine that are relevant to the user. Wine.com maintains an ID number for each possible search criteria, and when running a search query one simply includes a list of the IDs that should be used to filter the results.

Unfortunately, wine.com does not provide a convenient or robust method of translating from human readable criteria to the corresponding IDs (i.e. there is no particularly nice way of going from “Italian red wine” to the IDs 105 and 124). We therefore had to maintain our own offline lookup table mapping criteria keywords to their ID numbers. This was not too inconvenient given our current limited scope, but it would likely become unwieldy if VitiBot’s feature set grew or if our domain involved collections of keywords which changed much more often than those in the wine domain.

4 Experiments

To evaluate the success of VitiBot, we test our trained Wit.ai component individually and con-
duct user studies comparing a baseline implementation against the full VitiBot dialogue system.

4.1 Wit.ai Entity Extraction

Because a large part of VitiBot's successful state management relies on correct entity extraction, we evaluated our trained Wit.ai component individually on a test set of user-generated utterances.

4.1.1 Procedure

The test set consists of $N = 58$ utterances generated from users. Each utterance was transcribed to text and had its entities of interest extracted by a human. These human transcriptions and human-extracted entities were used as the gold labels.

Each utterance sound recording is sent to the Wit.ai app, which returns a JSON of extracted entities. An entity is considered detected if the gold entity key is present. An entity is considered correctly extracted if it is detected and its value is also correct. An utterance’s extraction is considered correct if every entity is correct. We calculate four measures of success:

- Utterance Accuracy: proportion of correctly extracted utterances
- Entity Extraction Accuracy: proportion of entities correctly extracted
- Entity Extraction Error: proportion of entities correctly detected but incorrectly extracted
- Entities Detection Error: proportion of entities not detected

Utterance accuracy is normalized by the total number of utterances. Entity detection and extraction measures are normalized by the total number of expected entities ($E = 87$ entities).

Additionally, to evaluate the entity extraction itself (i.e. if the transcription of utterances were perfect), we repeat the above analysis, but instead, send in each gold text transcription.

4.1.2 Results

From Table 1, we see that the utterance accuracy is greatly improved when extracting entities from the gold text transcriptions, indicating that Wit.ai’s ASR mis-transcribes many of the utterances. We can infer that this mis-transcription contributes greatly to entity detection error. When looking at the individual entities that Wit.ai fails to extract, we noticed that many of them are keywords that are related to multiple entities or are mis-transcribed foreign French words that are common within the viticulture domain. However, if an entity is detected, in both raw speech and gold text scenarios the extracted entity usually has the correct value.

<table>
<thead>
<tr>
<th></th>
<th>Raw Speech</th>
<th>Gold Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utterance Acc.</td>
<td>0.482759</td>
<td>0.810345</td>
</tr>
<tr>
<td>Entity Extraction Acc.</td>
<td>0.620690</td>
<td>0.862069</td>
</tr>
<tr>
<td>Entity Extraction Err.</td>
<td>0.011494</td>
<td>0.022989</td>
</tr>
<tr>
<td>Entity Detection Err.</td>
<td>0.367816</td>
<td>0.114943</td>
</tr>
</tbody>
</table>

Table 1: The resulting success measures of the trained Wit.ai component on raw speech vs. gold text transcriptions.

4.2 Example Outputs

Below, we’ve included some examples of full successful conversations and common errors. You can also see a video demonstration of VitiBot here: https://www.youtube.com/watch?v=VPvt_WxOMDw

4.2.1 Successful Conversations

In the first conversation below, because the user did not specify any wine qualities other than red, VitiBot asks basic follow-up questions. The food pairing detection also successfully recognizes “pizza” as a subcategory of pasta and grains and jumps straight into the specific details regarding “pizza”.

**VitiBot:** Hello there! I am VitiBot, your personal wine expert. Let me know if there is anything I can search for you.

**User:** I want a red wine.

**VitiBot:** Ok, I’ll take that into consideration. You’re looking for a red wine. Are you on a budget? If so, what’s your spending range?

**User:** Yes.

**VitiBot:** What is your price range?

**User:** Under 50 dollars.

**VitiBot:** Ok, I’ll take that into consideration. You’re looking for a red wine costing no more than 50 dollars. Do you plan on having a meal with the wine? For example, will you have meat, cheese, dessert, pasta and grains, poultry, seafood, or none of these?

**User:** I’ll be eating pizza.
VitiBot: Will the pizza’s main ingredient be meat, vegetable, or none of these?
User: None of these.
VitiBot: Ok, I’ll make my wine recommendation so that it pairs well with pizza in general. Have you told me everything that you’re looking for?
User: Yes.
VitiBot: Here is the information about the chosen wine:

Name: La Colombina Brunello di Montalcino 2012
Varietal: Sangiovese
Vineyard: La Colombina
Vintage: 2012
Price: 39.99
Location: Tuscany — Italy

VitiBot: I hope you enjoy the recommendation! This is wine number 1 of 5. Let me know if you want to see any of the other wines or if you want to start a new search.

The next conversation takes in a complex user query and extracts the relevant entities from the statement and also asks follow up questions for the food pairing.

VitiBot: Hello there! I am VitiBot, your personal wine expert. Let me know if there is anything I can search for you.
User: We had duck confit last time we had a dinner party. I want a wine that will pair well with the food from this time though. I think we’re having chicken.
VitiBot: Ok, I’ll take that into consideration. Will the duck be seared, confit, fois gras, with fruit, or none of the above?
User: I’m not having duck. I’m having chicken.
VitiBot: Ok, I’ll make my wine recommendation so that it pairs well with poultry: duck in general. Pinot Noir is a classic match with a simple duck dish. Are you on a budget? If so, what’s your spending range?

Another common error is VitiBot’s inability to detect contradictory information. In the below example, the user requests a white wine but with a
food pairing that goes well with a red wine. As a result, VitiBot returns no results.

User: I’m looking for a white wine that pairs well with chicken.
VitiBot: You’re looking for a white wine. There are many ways of preparing chicken, several which includes with herbs, lemon/citrus, mushroom, mustard, spicy, as a barbecue, cream-based, and of course, fried. Which one is it?
User: With mushrooms.
VitiBot: Sounds delicious! I will definitely take these foods into consideration! Earth mushrooms and mild chicken go well with a Pinot Noir, particularly from New Zealand, Oregon or France.

4.3 User Studies

To evaluate the dialogue system as a whole and see what features of VitiBot contributed most to a successful conversation, we asked users to interact with a baseline implementation and the full-featured VitiBot. The baseline implementation has the ability to extract relevant entities such as wine type/varietal and food pairings, but does not ask any follow up questions. Given an input, the baseline implementation will immediately return a result, whereas the full-featured VitiBot will ask follow up questions for any incomplete features and add some information about the user’s selection.

4.3.1 Procedure

We recruited $U = 8$ users to interact with VitiBot. Each user is asked to interact with both the baseline implementation and VitiBot (order of interaction randomly assigned). For each dialog system, the user is asked to make a few queries and interact with the bot normally. Then, users filled out a survey for each system.

4.3.2 Evaluation Questions

For each system, the user rates the following statements with various adjectives on a scale of 1 (strongly disagree) to 10 (strongly agree):

- The conversation I had was [one-sided, natural, insightful].
- The responses I received were [expected, relevant, useful].
- It was easy to specify a [type of wine, wine pairing, price range].
- I could express many preferences.
- The bot understands how to recommend a wine.

Additionally, we asked some qualitative questions:

- What preferences did you want to specify that you could not?
- Where did VitiBot fail? Where did VitiBot do well?
- Did you have any circular conversations?
- Compared to the baseline, how helpful/useful was VitiBot? Compared to Siri/Google, how helpful/useful was VitiBot?

4.3.3 Results

We see in Table 2 that adding multi-part dialogue naturally made the conversations less one-sided, somewhat more natural, and somewhat more insightful. Additionally, the wine pairing feature along with its follow up questions was the most useful and easy to use feature for users.

From the responses to the qualitative questions, most users agreed that while multi-part dialogue helped narrow down preferences, it also led to more circular conversations when user input was incorrectly recognized, and thus was even more confusing. This likely explains the slight decrease in VitiBot’s usefulness and relevance. Also, because VitiBot generates its next response based on its current state and its conversation history, after using the bot once or twice, the bot sounds very scripted as the questions were not changing. Several responses indicated that VitiBot is more useful than other virtual assistants simple because VitiBot does return an exact bottle of wine, whereas Siri or Google either makes vague suggestions of wine types or directs the user to a restaurant.

It should be noted that the majority of users describe themselves as having limited wine knowledge and do not know what they are looking for. These users agreed that follow-up questions for food pairings was the most useful feature.

5 Conclusion

In conclusion, we have developed a multi-turn, mixed-initiative dialog system to assist users in
<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>VitiBot</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-sidedness</td>
<td>7.750</td>
<td>3.750</td>
</tr>
<tr>
<td>naturalness</td>
<td>3.125</td>
<td>5.500</td>
</tr>
<tr>
<td>insightful</td>
<td>1.375</td>
<td>2.750</td>
</tr>
<tr>
<td>expected</td>
<td>6.250</td>
<td>5.625</td>
</tr>
<tr>
<td>relevance</td>
<td>6.500</td>
<td>6.375</td>
</tr>
<tr>
<td>useful</td>
<td>6.625</td>
<td>6.375</td>
</tr>
<tr>
<td>type/varietal of wine</td>
<td>6.250</td>
<td>6.250</td>
</tr>
<tr>
<td>wine pairing</td>
<td>2.625</td>
<td>6.750</td>
</tr>
<tr>
<td>price range</td>
<td>5.625</td>
<td>5.500</td>
</tr>
<tr>
<td>expressiveness</td>
<td>4.125</td>
<td>5.250</td>
</tr>
<tr>
<td>understanding of wine</td>
<td>5.000</td>
<td>6.375</td>
</tr>
</tbody>
</table>

Table 2: Average user ratings.

finding a bottle of wine consistent with their preferences. The inclusion of both user-initiative and system-initiative in the dialog allows the user to maintain a sense of autonomy while also enabling VitiBot to narrow down their priorities as much as possible. Because of the disparity between the number of possible search criteria and the assumed level of wine knowledge for the average user, we incorporated non-blocking question states which extract information from the user when they have it and are ignored when they do not.

As human-human dialogues typically involve both many turns and each participant assuming initiative periodically, it is understandable that the final system had higher satisfaction levels than the single-turn, pure user-initiative baseline model.

There are a few remaining technical hurdles with the existing system. Primary among these is the ASR module, which we have shown is responsible for a significant portion of the system’s mistakes. Furthermore, the inability of the ASR system to correctly transcribe French varietal names is a severe limitation as well. From preliminary testing, it seems like the Google Speech API might be a good choice for a replacement ASR system, as it would allow us to pass in “likely phrases” which could help account for the unexpected French phrases in our English dialogues.

There are also several directions in which this system could be extended, the most of obvious of which is simply extending the bot to allow it to filter wines using more criteria. We could also extend it to remember the preferences of and suggestions made to different users, preventing users from having to repeat the same information each time they use the system. Another intriguing possibility would be the incorporation of reinforcement learning techniques. It is likely that with enough people using and evaluating the results of our system, VitiBot would eventually be able to learn which line of questioning leads to the highest levels of user satisfaction with the recommended wine. We would be able to dynamically tailor the bot to both experienced wine drinkers looking for a specific bottle and novice wine drinkers looking to develop their familiarity with different types of wine.

A video demonstration of VitiBot can be found here: https://www.youtube.com/watch?v=VPvt_WxOMDw

References


