Recipes Alexa skill
Language Understanding Systems 2019-2020

Davide Zanella
University of Trento
davide.zanella-1@studenti.unitn.it

Abstract

This project aims at developing a conversational agent for the recipes' context using the rasa framework, testing and evaluating different configurations, and finally transforming the agent into an Alexa skill. In particular, the objective of the Alexa skill is to provide a conversational help while searching for a recipe or while following it step by step. It allows to search recipes filtering them by some constraints, like gluten-free food, vegetarian food, wanted and unwanted ingredients. After choosing a recipe, the user can ask for some information about it regarding nutrition, description, or necessary steps. Finally, the user can decide to start preparing the recipe, and the conversational agent will guide him step by step to complete the desired recipe.

All the code is available on GitHub[1].

1 The project structure

As previously introduced, the project is developed using the rasa framework, so it follows the rasa structure and file hierarchy.

1.1 The recipes database

The first thing to start the project is to discover a database containing some recipes with at least the necessary ingredients and steps. The Food.com recipes CSV file seems to be a good starting point to create a database upon it. In fact, extracting its data and web scraping the missing information from the food.com website allows me to create an SQLite3 database file. To access and manage the records of the database in an easy way, an useful ORM (Object-Relational Mapping) python library is used.

[1] `https://github.com/davidezanella/recipes-alexa-skill`

1.2 The rasa entities

In a rasa project, it is necessary to define some entities which should be extracted from the sentences by the NER (Named-entity recognition). The entities in this project are, of course, related to cooking and are the following:

- *avoid_ingredients*: ingredients that must not be present in the recipe;
- *search_ingredients*: ingredients that must be present in the recipe;
- *search_text*: name of the recipe to search for;
- *gluten_free*: states if the recipe should be gluten-free or not;
- *vegan*: states if the recipe should be for vegans or not;
- *vegetarian*: states if the recipe should be for vegetarians or not;
- *max_calories*: defines the maximum number of calories present in the recipe;
- *max_minutes*: defines the maximum amount of minutes necessary to complete the recipe;
- *ordinal*: entity used to extract ordinal numbers from the sentences (e.g.: first, second);

1.3 The rasa intents

In rasa, intents are used to understand what is the intention of the user. For this conversational agent, the intents defined are the following:

- *next_page*: the user wants to hear more results;
- *search_recipe*: the user wants to search for a recipe;
- *greet*: the user wants to greet;

- *goodbye*: the user is leaving the conversation;
- *choose_recipe*: the user has referred to a recipe (e.g.: Tell me more about the first recipe);
- *get_recipe_info*: the user is asking for some information regarding a recipe;
- *get_recipe_ingredients*: the user wants to know the necessary ingredients for a recipe;
- *get_recipe_nutrition*: the user wants to know the nutritional characteristic of the recipe;
- *get_recipe_steps* the user wants to know the necessary steps to prepare the recipe;
- *start_cooking*: the user wants to cook a recipe;
- *next_step*: the user has finished a recipe step and wants to know the next one;
- *affirm*: the user has provided an affirmative answer;
- *deny*: the user has provided a negative answer;
- *clear_search*: the user wants to clean all the previous search filters;

1.4 The rasa actions

The rasa actions are specific classes written in python invoked by rasa in some specific situations. For example, if the user writes a sentence recognized with the intent greet, rasa, if properly trained, will invoke the utter_greet action. In the actions code, it can be defined some custom code and, usually, is the part of the conversational agent that interacts with databases or external services.

In this project the following actions are defined:

- *action_choose_recipe*: saves in a slot the recipe index referred by the user;
- *action_clear_search*: clears the search filters;
- *action_get_recipe_info*: returns details about a recipe;
- *action_get_recipe_ingredients*: lists the ingredients needed for the selected recipe;
- *action_get_recipe_nutrition*: gives details about the nutritional values of the selected recipe;
- *action_get_recipe_steps*: lists the necessary steps for a recipe;
- *action_next_page*: increments the slot containing the page number;
- *action_start_cooking*: asks the user if he really wants to start doing the recipe;
- *action_next_step*: returns the successive step for a recipe;
- *utter_goodbye*: says goodbye to the user;
- *utter_greet*: answer to the greet of the user;
- *utter_rephrase*: asks the user to reformulate the phrase;

1.5 The rasa forms

The rasa forms are classes similar to the rasa actions that can be invoked in particular moments during a conversation. Forms are useful for executing actions where several slots are involved. In this project only one form is defined: the *action_form_search_recipe*, used when the user wants to find a recipe. Forms offer some useful functionalities, like slot mapping and validation. The slot mapping function defines how slots and entities are linked, so it allows us to link a slot with different entities. The slots validation functions become really useful in this project because thanks to them, the value of a slot could be cleaned or parsed, as later explained in Section 2.1.1.

2 The model configuration

All the utterances needed to train the Nlu part of rasa, and the stories used to train the Core part were written including all the entities and intents. Now, it is necessary to find the best Nlu and Core configuration to create the most useful conversational agent possible.

2.1 Nlu components

The Nlu part of rasa has the role to extract entities and understand the intent of a sentence. There are a lot of components that can be used and put together to form the Nlu pipeline. Some of them tokenize the words, others extract features from text and the remaining classify intents or extract entities from text.

I decided to test the pipelines that are listed in the rasa documentation and some modified versions of them. Analyzing the results reported in Table 1, a custom pipeline seems to perform slightly better over the training utterances than the suggested pipeline for English agents. All the best

NLU	F1
conveRTT + SKlearn + CRF	**0.934**
default en	**0.932**
default conveRTT	**0.926**
mitie + mitieIC + DIETC	**0.878**
default spacy	**0.844**
en + LanguageModels	**0.840**
default supervised	**0.833**
default mitie	**0.812**

Table 1: Nlu test results performed over the training utterances.

three pipelines use *ConveRT* as word tokenizer and featurizer, but the default en, in addition, includes other components for the same scope. The three best Nlu pipelines use different intents and entities extractors.

2.1.1 Duckling

In all the pipelines tested, I inserted the Duckling entity extractor. It is a component that relies on an external service that is trained to extract some fixed entities. I used it to extract the ordinal ones: it can detect and extract the integer value from an ordinal string (e.g. ”second” $\rightarrow$ 2). This was useful and allows me to avoid static conversions, as done in a rasa tutorial project, where the mapping between words and numbers was written to a csv file (https://github.com/RasaHQ/tutorial-knowledge-base/blob/master/knowledge_base/data/mention_mapping.csv) and used in the actions.

2.2 Core policies

The Core part of rasa has the role to manage the flow of the conversation and decide the action to perform in every specific situation. I decided to test several Core policies with different configurations in order to understand which one performs better. I didn’t use the Memoization policy, since it simply memorizes the training stories and reuses them if encountered while testing. This would lead to useless results since, for now, we are testing the policies on the training stories. The policies including Keras uses an LSTM (Long short-term memory) to predict actions with two possible configurations: the ”LabelTokenizerSingleStateFeaturizer”, which creates a vector of features, and the ”BinarySingleStateFeaturizer”, which creates a one-hot encoding, as state featurizers. The TEDPolicy is a policy that uses a transformer model and a dense layer to get the next action. It has different parameters that were tested, like the number of epochs, the maximum number of history values to consider, and the number of hidden layers. The Fallback policy allows us to set some threshold values, under which we don’t want to follow the prediction and we prefer to ask the user to rephrase the sentence. In Table 2 the results over the 38 training stories are reported.

Core	Correct
Keras labelTokenizer	**35** / 38
Keras binary + Fallback	**35** / 38
TEDPolicy 3 hidden layers	**35** / 38
Keras labelTokenizer + Fallback	**34** / 38
Keras binary	**34** / 38
TEDPolicy	**34** / 38
TEDPolicy epochs 200	**34** / 38
TEDPolicy max history 3	**34** / 38
TEDPolicy 2 hidden layers	**32** / 38
TEDPolicy 4 hidden layers	**32** / 38
TEDPolicy max history 2	**29** / 38
TEDPolicy epochs 50	**23** / 38
TEDPolicy max history 1	**22** / 38
TEDPolicy epochs 1	**21** / 38

Table 2: Core test results performed over the 38 training stories.

2.3 End-to-end evaluation

In the previous two sections, we have focused on the best Nlu and Core configurations evaluated over the training data. In this section, we will put together the two rasa parts and test them over the testing data. For rasa, the testing data are end-to-end stories that include both the Nlu utterances and the Core stories. While writing the end-to-end stories, I tried to avoid using the exact terminology used in the training phrases, to make the testing phase more reliable. Table 3 reports the results of the evaluation and clearly shows that the default pipeline for English conversations performs better than the others. Since its F1 scores are really close to each other, it is reasonable to say that all the three best Core policies perform the same with the default en pipeline. In Table 4 are reported the detailed results of the end-to-end testing of the best Nlu pipeline and Core policy. Analyzing the results comes out that the model performs well, although some entities’ values are not extracted correctly. Manually testing the model can be noticed that it lacks generalization. For example, if the user asks to search recipes with a particular ingredient, depending on the ingredient, the model recognizes it as an ingredient to search for or to avoid. To partially mitigate this issue, could probably be enough to add more utterances and use a lot of different in-

Nlu	Core	Acc	Prec	F1
default en	Keras binary + Fallback	0.932	0.945	**0.936**
default en	TEDPolicy 3 hl	0.929	0.925	**0.924**
default en	Keras labelTokenizer	0.922	0.931	**0.923**
conveRTT + SKlearn + CRF	Keras labelTokenizer	0.904	0.902	**0.898**
conveRTT + SKlearn + CRF	TEDPolicy 3 hl	0.876	0.878	**0.868**
default conveRTT	TEDPolicy 3 hl	0.848	0.851	**0.847**
default conveRTT	Keras labelTokenizer	0.844	0.848	**0.844**
default conveRTT	Keras binary + Fallback	0.830	0.833	**0.829**
conveRTT + SKlearn + CRF	Keras binary + Fallback	0.826	0.836	**0.825**

Table 3: End-to-end test results performed over the testing stories.

gredients in order to force the model to generalize the ingredients.

3 Contributions to the rasa framework

During the development of this project, I encountered some bugs in the rasa framework and since it is an open-source project, I decided to contribute making some pull requests to the GitHub repository.

3.1 Wrong duckling testing

The first problem that was encountered regards the end-to-end evaluation of the duckling extracted entities. In particular, the ordinal entities were returned as integer values from duckling, but in the end-to-end utterances, all the values of the entities must be of type string. This leads to a lot of errors in the testing results. I forked the rasa project on GitHub and tried to fix the error. After that, I created a pull request [2] proposing my changes, but one of the maintainers of the project decided to fix the problem in a different way.

3.2 Wrong end-to-end evaluation

Doing end-to-end evaluations of different models, I noticed that the results changed a lot and that sometimes neither the duckling entities were counted as correct even if they were not reported in the error report. I decided to further investigate and I found a problem in the comparison between the predicted and the target entities. Just one prediction error could lead to other errors because of the

[2] https://github.com/RasaHQ/rasa/pull/5977

	f1-score	support
next_page	1.00	1
action_next_step	0.93	8
[third]{”entity”: ”ordinal”, ”value”: ”3”}	1.00	5
[without gluten]{”entity”: ”gluten_free”, ”value”: ”true”}	1.00	1
get_recipe_ingredients	0.80	2
[carrots](avoid_ingredients)	1.00	2
start_cooking	0.67	5
greet	1.00	18
action_start_cooking	0.67	5
[pastiera](search_text)	0.00	1
[first]{”entity”: ”ordinal”, ”value”: ”1”}	1.00	1
[carrots](search_ingredients)	1.00	1
[gluten free]{”entity”: ”gluten_free”, ”value”: ”true”}	1.00	1
utter_deny	1.00	1
get_recipe_nutrition	1.00	3
action_get_recipe_ingredients	0.50	2
[120](max_calories)	1.00	1
[sixth]{”entity”: ”ordinal”, ”value”: ”6”}	1.00	1
[pears](avoid_ingredients)	0.00	1
[second]{”entity”: ”ordinal”, ”value”: ”2”}	1.00	2
[pie](search_text)	1.00	1
search_recipe	1.00	18
affirm	0.67	4
action_get_recipe_nutrition	0.80	3
next_step	1.00	4
[meat](search_ingredients)	0.00	2
action_get_recipe_info	0.80	2
[rosemary](search_ingredients)	0.75	4
goodbye	1.00	18
action_form_search_recipe	1.00	19
get_recipe_info	1.00	2
[60](max_minutes)	1.00	1
[pineapple](avoid_ingredients)	0.67	3
action_listen	0.99	78
[pineapple](search_ingredients)	0.00	1
[53](max_minutes)	1.00	1
[80](max_minutes)	1.00	2
action_get_recipe_steps	0.67	2
utter_goodbye	1.00	18
[175](max_calories)	1.00	1
deny	1.00	1
action_choose_recipe	0.94	8
utter_greet	1.00	18
[vegetarians]{”entity”: ”vegetarian”, ”value”: ”true”}	1.00	2
action_next_page	1.00	1
[53](max_calories)	1.00	1
[rosemary](avoid_ingredients)	0.00	1
[vegans]{”entity”: ”vegan”, ”value”: ”true”}	1.00	2
get_recipe_steps	0.67	2
[70](max_minutes)	1.00	1
micro avg	0.95	283
macro avg	0.83	283
weighted avg	**0.94**	283

Table 4: End-to-end test detailed results of the best Nlu pipeline and Core policy.

way the two lists were aligned. I created another pull request [3] with the fix to the problem and, after some reviews, the maintainers decided to accept my changes.

[3] https://github.com/RasaHQ/rasa/pull/6419