Sport Assistant Bot with Rasa
Second Project Language Understanding System

Alberto Carbognin
Università degli Studi di Trento
mat. 211420
alberto.carbognin@studenti.unitn.it

Abstract

In this second project we built a conversational agent using the rasa open source framework. The domain is sport related; custom actions were created to answer questions based on a API provider. By using the rasa x and the interactive learning shell new annotated data were provided to the nlu module, and new stories to the core module. We finally evaluated it using the built-in tool.

1 Introduction

Conversational agents are becoming easier to build thanks to open source frameworks like RASA. This second project for the course of LUS aims to leverage this framework to develop a dialog system on a specific chosen domain. In this case the sport domain was chosen. The agent should be able to answer to users asking for information about sports players and sports teams; in addition features like *RASA Forms* should be explored and investigated. In this case a user can simulate to book a certain quantity of tickets for the next upcoming match of a provided sport team. We looked for the right API to use for the task from *Rapid API*[1]; we found that **TheSportsDB APIs** has the endpoints we needed. Subsequently we identify the main Stories that we want our system to be able to understand; we then explore the interactive learning shell and RASA X. Finally, we try different pipelines and policies and gather the evaluation results. The paper is structured in three main chapters; the first one is a Description of the system highlighting what is working and what is not. In the second chapter we show the gathered experimental results trying to explain what we observed. In the latter chapter we discuss what can be improved and what are the critical pieces of the work.

[1]An online service that collects multiple API service providers with a coherent documentation.

2 Description of the system

We used the RASA framework to build a dialog system on sport domain; finding the working configuration compatible with RASA x has not been trivial. Many hours has been spent trying to make it work; it has been tried Docker but with no result. The working development environment is an ubuntu machine with: rasa **2.8.14**, rasa sdk **2.8.9**, rasa x **0.42.6** and python **3.8.10**. In order to use rasa the most important yaml files that we configured were the following:

- **config.yml**: this file contains the *pipeline* and the *policies* used by the conversational agent;
- **domain.yml**: this contains configuration parameters and *intents*, *entities*, *actions*, *slots*, *responses* and *forms*;
- **nlu.yml**: this one contains the data used by the natural language understanding component to learn how to identify intents;
- **rules.yml**: in this file it has been defined the *Form* and other rules like greetings;
- **stories.yml**: in this file we defined the stories used by the rasa core.

In the initial pipeline, in the config.yml file, the default *WhitespaceTokenizer* has been replaced by the *SpacyTokenizer* importing a ready to use language model from the *Spacy* library. In particular, it has been explored the usage of the **en_core_web_lg** and the roberta-base transformer **en_core_web_trf**. It has also been added the **SpacyFeaturizer** and the **SpacyEntityExtractor**. The latter it has been used to extract entities like PERSON, ORDINAL and ORG. Using the roberta-base model shows a slightly better entity extraction during the early stage of the development. It has been also configured a DIETClassifier for both intent classification and entity extraction for the QUANTITY entities. The reason behind this lays in the fact that it can extract them better than *SpacyEntityExtractor*.

We wrote a total of 6 **custom actions**:

- **action_player_info**: utter a brief description of the requested *player*;
- **action_player_born_date**: utter the born date of the *player*;
- **action_player_sport**: utter the *player*'s sport;
- **action_player_height**: utter the *player*'s height;
- **action_player_plays_in_team**: checks if the given *player* has associated the a certain *team* and utter the result;
- **action_upcoming_events**: utter a list of the next upcoming events given a *team*.

and a custom *validation* form **validate_booking_ticket_form** to book tickets for the next upcoming event; still a very rough implementation. The custom *Actions* are reading and updating the slots based on what it is available; for instance if we ask to the dialog system for a player with an incomplete ambiguous *name*, there could be more than one match given by similar players' names. The system addressed the case showing the user a list of results from where choosing the correct one. We also defined the *intents* in the **domain.yml** file that are used in the *Stories* to trigger the custom actions:

- **player_info**
- **player_born_date**
- **player_sport**
- **player_height**
- **player_plays_in_team**
- **upcoming_events**

Moreover, we provide example data in the **nlu.yml** file to tell the *nlu* module how to classify them. In all the pipelines we keep the *DIETClassifier* to perform specifically this classification task.

2.1 pipelines

The base pipeline (*config.base.yml*) and policies was the following:

- **pipeline**

  defined SpacyNLP: *en_core_web_trf*

  SpacyTokenizer

  SpacyFeaturizer

  SpacyEntityExtractor: *extracting PERSON, ORDINAL, ORG, QUANTITY entities.*

  RegexEntityExtractor

  CountVectorsFeaturizer

  CountVectorsFeaturizer

  DIETClassifier: disable entity recognition

  EntitySynonymMapper

  ResponseSelector

- **policies**

  MemoizationPolicy

  RulePolicy

  TEDPolicy

the configuration *config.variant_1.yml* changed the Spacy model to use and added a *LexicalSyntacticFeaturizer*:

- **pipeline**

  defined SpacyNLP: *en_core_web_md*

  [...]

  RegexEntityExtractor

  LexicalSyntacticFeaturizer

  CountVectorsFeaturizer

  [...]

- **policies**

  [...]

the configuration *config.variant_2.yml* is similar to the base one with the *DIETClassifier* performing entity extraction for the **QUANTITY**, but it adds a *FallbackClassifier* in the pipeline and the according policy to the policies section:

- **pipeline**

  [...] DIETClassifier with entity extraction enabled

  ResponseSelector

  FallbackClassifier with *threshold* 0.7 and *ambiguity_threshold* 0.1

- **policies**

  [...]

  UnexpecTEDIntentPolicy with *max_history* 5 and 100 epochs

  TEDPolicy with *max_history* 2

the configuration *config.variant_3.yml* is similar to the *config.variant_1.yml* one with the *DIETClassifier* performing entity extraction for the **QUANTITY**, but it adds a *FallbackClassifier* in the pipeline and the according policy to the policies section:

- **pipeline**

  [...] DIETClassifier with entity extraction enabled

  ResponseSelector

  FallbackClassifier with *threshold* 0.7 and *ambiguity_threshold* 0.1

- **policies**

  [...]

  UnexpecTEDIntentPolicy with *max_history* 5 and 100 epochs

  TEDPolicy with *max_history* 2

The pipelines are very similar to each other, and it is done with the idea to highlight if there is a significant improvement by changing only a small piece. We finally connect via *ngrok* using the proper connector the rasa server to an Alexa Skill.

2.2 training & issues

The system has been trained by initially exploring the *interactive learning* shell and after the *RASA X* web application to save new stories and annotate from new example data that were added to the nlu file automatically. In the used rasa x configuration it was not possible to run the *interactive learning* by GUI; this was a major breakdown for the improvements. Some issues that we found were for instance the fact that we didn't manage to exit from the middle of the Form's asking for empty slots even defining the proper *stop it* rule.

3 Comparative Evaluation

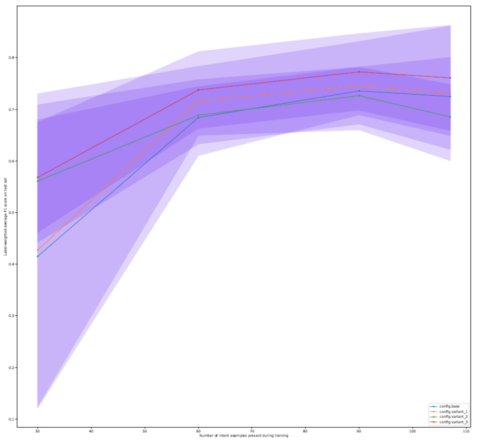

Figure 1: F1 score before interactive learning and annotating with rasa x

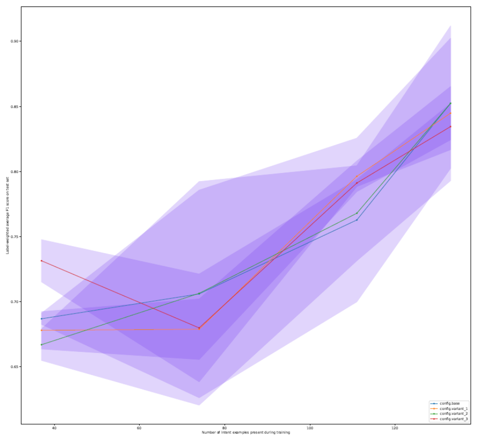

Figure 2: F2 score after interactive learning and annotating with rasa x

We run the evaluation across the pipelines two times; the first one without having an expanded dataset, and the latter after some *rasa x* interactions and *nlu entity annotation*.
We launch the following command:

```
rasa test nlu --nlu data/nlu.yml
--cross-validation
--config
pipelines/config.base.yml
pipelines/config.variant_1.yml
pipelines/config.variant_2.yml
pipelines/config.variant_3.yml
```

The gathered results are reported in the tables 1 and 2. As we might observe the tool that runs the evaluation was removing a certain portion from

pipeline	25%	50%	75%	100%
config.base	0.42	0.68	0.74	0.72
config.variant_1	0.43	0.72	0.75	0.73
config.variant_2	0.56	0.69	0.73	0.69
config.variant_3	0.57	0.74	**0.77**	0.76

Table 1: F1 Score of Pipelines compared with percentage of intent examples during training, before using rasa x to add new nlu data and stories.

pipeline	25%	50%	75%	100%
config.base	0.69	0.71	0.76	**0.85**
config.variant_1	0.68	0.68	0.80	0.84
config.variant_2	0.67	0.71	0.77	**0.85**
config.variant_3	0.73	0.68	0.79	0.83

Table 2: F1 Score of Pipelines compared with percentage of intent examples during training, after using rasa x to add new nlu data and stories.

the dataset before running the evaluation. Moreover, three runs were performed and the averaged together to get the obtained result. In the generated plots depicted in figure 1 and figure 2[2] we can observe that the calculated confidence interval is getting smaller. The increasing score trend of the pipelines in the *before rasa x* is suggesting us that around 75% we reached the maximum performance. After using rasa x to expand the dataset the conclusion was different. Probably we get stuck in a local minima, because, as we can see from table 2 for variant 2 and 3 with half dataset for training we get a lower performance than only train with 25% of the total size. The conclusion was different because the trend still increasing until the best performance observed by the base configuration and the variation 2. The confusion matrices reported in figure 3 and figure 4 are reporting some insights about the the correct intent prediction. Both cases are showing that of the intents were correctly predicted. There are some missing blocks along the diagonal and the reason is that there are missing examples in the stories for intent such as **out_of_scope**.

4 Discussion

The obtained result after the interactive training are showing that there are improvements that can be done. We consider the obtained result not good enough; the F1 score still very low considering the simple domain and task that the dialog system should perform. Other components like the intent classifier or the entity extractor can be improved. The latter could be changed with a custom component based on some other transformer, better if already finetuned on the considered domain. The main problems that the current system agent has is that sometimes is not able to predict the intention correctly when using Alexa. For instance, it is not able to extract the *ORDINAL* entity within the sentence ”the last one”. In the tests that were run with rasa X in the other hand, the entity *ORDINAL* or *choice* where extracted. No solution were found for this problem. There was also the need to manually setup lookup tables.

5 Conclusion

The projects shows how difficult can be to create a smart dialog system even using a very complete

[2]look at the github repository for a high resolution of the plots.

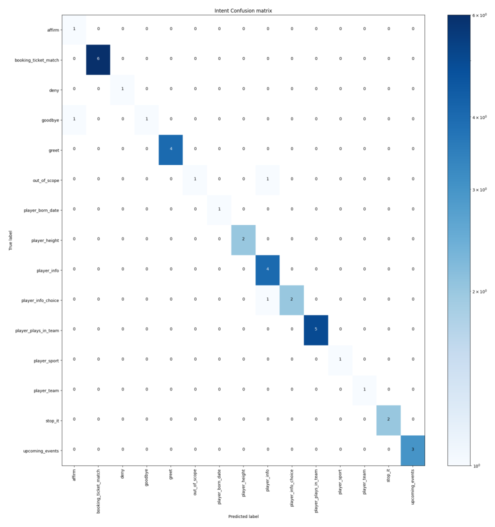

Figure 3: Intent confusion matrix before using rasa x

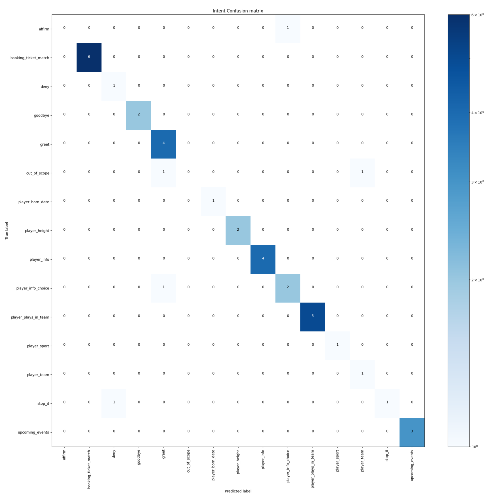

Figure 4: Intent confusion matrix after using rasa x

open source framework. Once you get used to the yaml files and how to use slots it can be really quick to setup. In the other hand between same rasa versions there may be a lot of difference with not clear documentation. I also lost many hours only on the setup of rasa x with the proper version compatibility. In conclusion future improvements could be done by trying to use different intent classifiers together with a better nlu annotated examples.