Second Project of the Natural Language Processing Course

Luca Di Liello
University of Trento
`luca.diliello@studenti.unitn.it`

Abstract

This document contains the instructions to prepare, implement and test a simple bot, which will allow a user to search for a restaurant and make a reservation using the voice. The program has been developed using the open-source python libraries RasaNLU, RasaCore, SpeechRecognition (with Google API's) and pyttsx3. RasaNLU is a natural language understanding framework to extract intents and entities from sentences while RasaCore uses this informations to choose the best next action to be executed. Both are written in Python given the giant number of machine learning tools available and both can be run in a "standalone" mode or in server mode. SpeechRecognition is a simple library that allows us to use the Google Speech Recognition service and to easily transform speech in text. Finally, we will use pyttsx3 to transform the speech in text (without using external services).

1 Credits

The software and the tecniques showed in this document were learned during the LUS course by professor Giuseppe Riccardi and professor Evgeny Stephanov. The core of the software is powered by the open-source libraries RasaNLU and RasaCore which permitted a much faster development of the project thanks to the powerful primitives to extract concepts from sentences and to take decision on what to do next. The TTS and STT tasks where allowed by the other two, previous mentioned, libraries pyttsx3 and SpeechRecognition.

2 Introduction

This document will touch four big tasks of the Human-Machine Spoken Dialog cycle, that are the speech recognition, the natural language understanding, the dialogue management and the speech generation. The language generation part will not be treated as it's pretty complicated and still not a completely solved problem. First we will briefly speak about the methods we used to transform speech in text and vice-versa, then there will a deeper treatment of the two Rasa frameworks.

2.1 The Human Machine Spoken Dialog

The Human Machine Spoken Dialog is the set of all components that, together, allow people to spoke with a machine and do some tasks. The components are executed in series as a cycle, each component taking as input the output of the previous one.

- Speaker
- Automatic Speech Recognition
- Natural Language Understanding
- Dialogue Manager
- Language Generation
- Text to Speech Synthesis
- Speaker

The *Automatic Speech Recognition* and the *Text to Speech Synthesis* are the components that has to interact directly with the user. The first has to take in input a discrete electric signal representing sound-waves and convert it to a sequence of words. There are many online services (Google first) that help doing this task easily. After having converted speech in text, the next step consists in the extraction of intents and entities from the input sentence. This task is called $Natural\ Language\ Understanding$. Intents are a finite set of what the user wants to do, for example a requests or a command. Better results can be obviously obtained trying to use one intent for each sentence. On the other side entities are the important informations that has to be extracted from the sentence and that has to be used by the machine. For example, the sentence "$please\ search\ for\ an\ Italian\ restaurant\ in\ London$" contains two entities: the type of cuisine and the location of the restaurant. In this case the intent was to do a research of restaurants. Once intents and entities has been correctly extracted, they are used by the $Dialogue\ Manager$ to take decisions on what the bot should do next. Dialogue Managers can be seen as routers: they receive as an input intents and entities and have to decide which action the bot should execute. For better performances they often receive in input a record of the past interactions. What are actions? Actions are functions that use all the previous informations to create a response for the user. They often can query a database or make online searches. The output of the action is a sequence of informations that has to be sent to the user, but are not yet sentences. The transduction of information in sentences for the user is done by the $Language\ Generation$ module. This task is one of the most difficult and this project will not cover how to create such a module, it will generate predefined sentences and use simple templates. Finally, once the sentence has been created, a $Text\ to\ Speech\ Synthesizer$ provides the functionalities to transform it in sound-waves that can be understood by a human, trying to emulate all the human behaviours while speaking, for example doing little pauses between different sentences, changes of tones, variations of the pitch and the correct use of accents. The cycle will then restart if the user will say something else.

2.2 RasaNLU and RasaCore

Rasa is an open-source project that develops two powerful frameworks to do $Natural\ Language\ Understanding$ and $DialogueManagement$.

2.2.1 RasaNLU

RasaNLU allows to train a model for the extraction of entities and the classification intents. It uses different existent ML and NLP libraries like Spacy and TensorFlow. Each model can be build as the combination of different layers that will be processed in pipeline. For example, the default $spacy_sklearn$ pipeline will use 7 different layers, each one with a different task like tokenisation and featurization. It reaches very high performances in entities extraction if a good training set is given. Broadly, it works better if sentences are not to long and there are no more than 3-4 entities per sentence. Intents are good classified too if they are not too many and if sentence are not too long.

2.2.2 RasaCore

RasaCore is a framework to decide the best action to execute given the story of the dialogue and the last intent/entities from the user. The core of the framework are the policies, which are trained to predict the next action to execute based on the history of the dialogue and the last user interaction. For each action a probability is computed and the one with the highest is launched. Policies can use neural networks, SVM and lots of other algorithms to improve their performances and should be trained through stories. A story is a sequence of user intents and bot actions describing a dialogue from beginning to end, often enriched with entities and other data. The additive data are contained in data structures called slots, that are filled and changed during the conversation.

3 Datasets Overview

There are 2 main dataset, one to train the Natural Language Understanding model and one for the Dialogue Manager.

3.1 Franken Data

The dataset for the NLU task is called Franken Data and contains 1977 sentences. For each sentence, the corresponding intent and entities are given. There are only six possible intents, that are:

- inform, that appears 1014 times
- thankyou, that appears 589 times
- affirm, that appears 260 times
- deny, that appears 85 times
- greet, that appears 13 times
- request_info, that appears 16 times

The sentences has a very low average length of 4.4 words, and this will leads to very satisfying results in the accuracy of the RasaNLU model. The entities are not so many, with an average of only 0.65 entities extracted from each sentence. There are lots of very simple sentence like $ok\ great$ or $perfect\ thank\ you$ that does not contain entities at all while some others have up to 4 entities and are longer (up to 22 words), especially those whose intent is to inform or request for informations. For each entity there are information about the part of the sentence it was found, to help the algorithm perform better. There are 5 different types of entities:

- cuisine, with 70 different specialities (573 total instances)
- price, with 6 different ranges (354 total instances)
- location, with 9 different places (338 total instances)
- people, meaning the number of sits to book (12 total instances)
- info, address or phone number (16 total instances)

As for the intents, the distribution of different entities is not uniform, and this will lead to situations in which some not popular entities will not be correctly extracted.

3.2 bAbi stories

The bAbi story dataset is a collection of stories developed by Facebook. Our version contains 1000 different stories about the process of booking a table in a restaurant. There is a common pattern that can be identified easily looking at the dataset, that is:

- The user greets
- Bot ask how can help
- User ask for a restaurant with some requirements
- Bot keeps asking other missing info
- User gives info requested by the bot
- Bot shows some results and ask the user if they are ok
- User says if they are fine, otherwise Bot propose other solutions
- User thanks
- Regards

The average length of the conversation, including both user and bot is of 25 actions and intents.

4 The main idea

The core of our algorithm is based on the composition of two FSTs encoding two different probabilities.
The first FST will be equivalent to the formula $P(token|concept\_tag)$, and will be able to transduce a sequence of tokens (a sentence) in the corresponding IOB notation. This probability will be computed using the number of times a concept tag appears with a given token divided by the frequency of the single token, as showed in Formula 2. This FST will have only one state with lots of self-transitions, each of them encoding the translation of a token to a concept tag and the respective cost.

$$P(token|concept\_tag) = C(token, concept\_tag)/C(concept\_tag) \quad (1)$$

Given that an FST encodes the *cost* of going through a specific path, the previous probability is converted in cost with the formula 3.

$$Cost(token|concept\_tag) = -\log\left(P(token|concept\_tag)\right) \quad (2)$$

The second FST will encode the probability of a specific concept given the previous n ones.

$$P(concept\_tag_n| concept\_tag_{n-1}, concept\_tag_{n-2}, ...) \quad (3)$$

This FST, that indeed is an FSA, will be created with the help of the OpenGRM library. As described in the their documentation, the available probability distributions to build the n-gram model are: witten-bell, absolute, katz, kneser-ney, presmoothed, unsmoothed. In the test section there will be results of different tests with each of this distributions. Moreover, there is the possibility to choose the deepness of the dependency of a concept tag with respect to the ones that precede. The more the number of previous concept tags on which compute the probability of the current tag, the higher the size of the equivalent FST.

The next step will consist in the composition of the previously created FSTs, being aware of the order, that is, the first left-composed to the second. As we mentioned in the introduction, the composition is a very powerful operation that should be followed by a determinisation and a minimisation to reduce the size of the result and to improve performances of next usages. What compose basically does is to merge two FSTs in such a way that if the first FST transduces token a to b with cost $cost_1$ and the second transduces b to c with cost $cost_2$, the result of the composition should transduce a to c with cost $cost_1 \circledast cost_2$.

4.1 Improvements

There is a simple way to improve our algorithm to perform better, that is to do some considerations on the numbers of O tags in the IOB notation of the sentences, because they are about the 75% of the total. Given that a B tag depends on the previous O tag, if it exists, and that eventually I tags can depend on previous O tags in n-gram model with a sufficiently big n, one can comprehend that in this case the probabilities of B and I tags does not discriminate over the previous symbol because often it's a simple O. The idea is to substitute the O tags in the language model with the token they are representing. There is a simple way to that without applying lots of edits to the previous algorithm, that is to pre-process the input file and then to post-process the results. The preprocess will simply substitute all the O tags of the training dataset with the corresponding token. Note that the input sentences that has to be tested will not need a preprocessing phase. Finally, given the modifications we applied, the results will have B tags, I tags and tokens. The post-process will consist only in the substitution of all tokens with O tags.

5 Test and Results

5.1 Base Algorithm

The graph in Fig. 3 shows the results of the base algorithm with all the different probability distributions and with and n-gram length going from 2 to 5. First some comments about the F1-score: the better tuples $(probability\ distribution, n-gram\ order)$ are the $(witten\_bell, 2)$ and $(absolute, 2)$. They reached an maximum F1-score of 76.37% in our test. Notice that all different combinations, apart from some particular cases, performs very similar and the variance of the results is very low. The average F1-score is of 74.25%. In this tests, the best algorithms are those with a small n-gram order and that requires less time to execute. This is probably due to the fact that most of the concepts have a length that is less or equal than 2 tokens and that applying an n-gram model with a high order on a dataset with a strong dominant tag as O is not useful. The time grows pretty linearly on the size of the order of the n-gram model, with all the test performed on the same machine with the same system load. The time is expressed in seconds and as you can see, our machine is able to process the test dataset in few minutes in all cases (the test dataset contains about a thousand sentences). The conclusions about this first test are that it's completely useless to use n-gram orders greater than 2 without applying the improvements we wrote about.

5.2 Improved Algorithm

The results of the improved model, reported in Figure 4, shows an average increase of the F1-score by 6.29 percentage points, but there are combinations, like the $(unsmoothed, 2)$, that do not really perform better, with a difference in the F1-score of only 0.6. On the other side, the presmoothed distribution with a n-gram order of 5, earn more than 14 percentage points w.r.t. the base case. In this test, the standard deviation of the F1-score results w.r.t. the base case is even lower, having all scores between 76.81% and 82.43%. We will now analyse some interesting cases. First of all take a look at the time: the $witten\_bell$ distribution with an n-gram order between 2 and 3 is about the 30% faster than all the other probability distributions. This should be taken into account in eventual large scale application, where the efficiency is a focal point. If you are more interested in maximising only the F1-score performances, probably you will note that the kneser_ney distribution with an n-gram order of 4 has the higher F1-score of 82.43%. Is it worth to use the kneser_ney distribution only for this very small improvement of the performances with respect to the others? This should be decided by the reader. Finally, some considerations about the time needed after having applied the improvement to the basic case. The language model built on a dataset with all the test_base.png

Figure 1: F1-score and Time of test on base case

test_impr.png

Figure 2: F1-score and Time of test on improved case

'O's replaced by tokens is really bigger than the previous, and our tests confirmed the assumptions showing an increment of the number of edges of 17 times. This leads to test timings that are more than 2 times longer w.r.t. the basic case, especially with n-gram orders greater than 3.

6 Error Analysis

There are few different type of errors, each of them more or less frequent as the others:

- The concept is completely missed by the algorithm.
- A concept is found but it do not really exists.
- A concept is matched partially or it's longer than the original.
- The type of the concept is wrong, but B and I tags were right.

Each of this error types appears many times, they are not rare cases. One of the most interesting type of error is the one that starts with an I tag even if, by definition, concept must start with a B tag. With respect to the other errors, this can be easily avoided by adding some constraints on the solutions proposed by the algorithm. Given the shortest path found, check if the solution contains concepts that starts with an I tag and, if present, remove that path by applying the difference between the FST and a new linear FST containing only the solution that has to be avoided. The other errors are more difficult to be corrected because they are the consequences of the design of our algorithm and of the small size of the training set. There are words in the test sentences that were never seen during the training phase and were treated as unknown symbols.

7 Conclusions

The algorithm presented in this paper has some particular advantages that has to be taken into account in eventual comparison: it is easy to be implemented and based on completely open-source libraries, it's fast and does not need a particular computing power. The accuracy and the F1-score are satisfying but not the state of the art. In particular, given the presence of a dominant tag, the F1-score is a better measure for the evaluation of the performances of the system, being about 82% while the accuracy is around 93%.

An complete example of the code is available here: https://github.com/Luca1995it/LUS-project

References

[1] OpenFST - Open-source library for constructing, combining, optimizing, and searching weighted finite-state transducers (FSTs). http://www.openfst.org/twiki/bin/view/FST/WebHome

[2] OpenGRM - Collection of open-source libraries for constructing, combining, applying and searching formal grammars. http://www.opengrm.org/twiki/bin/view/GRM/WebHome

[3] Wikipedia - Inside-outside-beginning tagging https://en.wikipedia.org/wiki/Inside?outside?beginning_(tagging)

[4] Wikipedia - Language Model https://en.wikipedia.org/wiki/Language_model

[5] NYU Courant - Weighted Finite-State Transducer Definitions (Chapter 2) https://cs.nyu.edu/~mohri/pub/csl01.pdf