Natural Language Processing Course Final Project: Chatbot with Information Retrival from Questions-Answers Knowledge Base
Prof. Roberto Navigli

Norman Di Palo, MSc. in Artificial Intelligence and Robotics

September 25, 2017

1 Introduction

In this work I'll show the design choices and implementation of the final projet for the Natural Language Processing course: a chatbot that answers questions by retriving information from a crowd-sourced database of question-answer pairs. The user can ask the chatbot a question in natural language, and the chatbot understands what is the relation/topic of the question, the subject and the object, and how to retrieve the information for the knowledge base. The chatbot is able to add information to the database aswell, by asking the user about the questions it couldn't answer to. The chatbot is based on Telegram, a popular messaging platform.

2 General architecture

The overall structure of the chatbot is the following: a Python program runs on a server, managing the input-output of messages and the processing of information. The Telegram APIs allow to receive the messages from the user, these are then first processed by a **machine learning algorithm** that predicts the **relation** between 16 possible ones. After that, another algorithm predicts **subject and object** of the question. In the next section, I will explain in detail the algorithms that I implemented and evaluated for this tasks, such as **recurrent neural networks, bidirectional LSTMs, convolutional neural networks and naive bayes classifiers** explaining my final choices based on numerous tests. With these informations, the program can now access the knowledge base to look for a matching pattern, thus the **domain** is not needed, making the chatbot more flexible and prone to natural language. Since some entries in the databases have subject and object written as only a **BabelNet ID**, the program finds the bnIDs of the detected subject and object.

The first step is trying to match the found relation with the relations of the entries. If a match is found, the algorithm proceeds to try to match the subject, or its bn id, with the one in the entry. If it succeds, it then retrieves the object as the answer, or, if the object is already given in input (so, the answer is "yes/no"), it analyzes the entry structure to give an answer. It must be noticed how some entries are **purposely negative answers**, such as *"Is Rome in Puglia?" "No"*, but having anyway *Rome - Puglia* as subject and object. Thus, a simple retrival of subject and object can often be wrong. The database has been found to be quite noisy, with many question/answer pair being grammatically or semantically wrong, and even the formal convention of the entries is often not respected, making it harder for the algorithm to retrieve information. Thus, I applied many tweaks to make it more robust to noise, with the drawback of sometimes needing more time to answer due to the need to make a longer search for numerous possible patterns.
The **enrichment phase** is activated when the chatbot cannot answer a question. In that case, it asks the user what was the answer to that particular question. Once the user replies, it performes a similar data processing as before, obtaining relation, subject and object of the question, based on the information provided, and creating a new entry for the database, than it's then sent to the server.
The flexibility of the algorithm allows the chatbot to answer to various question related to the same entry in the database. For example, if an entry specifies *"Where is Rome? Italy"*, the chatbot can answer not only questions like *"Where can Rome be found?" "Where is Rome located?"* by replying *"Italy"*, thus understanding that these refers to the same relation and subject, but even questions like *"Is Rome in Italy?"* with *"yes"*, thanks to an accurate pattern matching of the question and entries, based on the presence of objects or not, and so on.
This robustness is applied to the enriching phase aswell. The user can ask *"What is a surfboard made of?"*, and if the chatbot cannot find a match, it will ask for the answer. The user can reply *"Wood"*, and the chatbot will send the entry *""What is a surfboard made of?" "Wood" "Material" "surfboard" "wood"* to the server. But if the first questions was of the kind *"Is Naples in Italy?"* and the answer was *"Yes"*, the algorithm would understand the concept and then use *Naples - Italy* as subject and object, not, wrongly, the user's answer *"yes"*.
The implementation and all the algorithms can be found in the source code, I will gladly answer all the questions about those during the oral interview, or via e-mail if needed.

3 Dataset and learning algorithms

To accomplish relation understanding and subject and object detection, I designed and implemented different **machine learning algorithms**, evaluating their performances and testing them in different ways.
The first important step was **data preprocessing and cleaning**. I noticed how the crowd-sourced knowledge base is heavily unbalanced on the relation

part. Most of the relations are *place and generalization*, while other relations hardly appear, with a ratio of even 1000:1 for some. This makes the algorithms **heavily biased**, and affects even the validation phase, since the algorithm tends to train only on these relations, be validated on these too, resulting in apparently high accuracy, but ultimately failing in answer questions from the users that are spread among all the relations.
Thus, one first important step was to leverage the entries. On 600.000 entries, I set a maximum of 5000 entries per relation, but even in this case some relations were just a couple of hundreds. I used that size of database because, with a larger one, the performances weren't improving, while the training time became unfeasible.
The first algorithm that I tried was a **recurrent neural network** used for classification of the relation. The architecture is: a **LSTM layer** with the sentence as input, each word transormed in a 300 dimension embedding word2vec vector, and padded to be 10 vectors for each line. The output is a 100 dimensional vector. Then there is a **dropout layer** with 0.5 probability, a **fully connected layer** with ouput 50, another **dropout** and a final **fully connected layer** with 16 dimension as output. All the activations are ReLU except for a final softmax layer. This architecture achieved around **0.65 percent** accuracy on the leveraged dataset after 2-3 epochs. The training time is around 5 minutes per epoch on a Intel Core 2 Duo. I tried improving it with a **Bidirectional LSTM** with ta similar architecture but an additional recurrent layer, obtaining up to **0.79** percent, but with a training time of around one hour per epoch.
The last algorithm was a **Naive Bayes Classifier**, used with a bag-of-words approach. I extracted the words that appeard more than x times in the dataset, and used them as **features** to transform each question is a embedding sparse vector, suited for **NB**. I obtained the best results with 500-1000 times as hyperparameter for the frequency of words, obtaining **343 features** in the last case, and with a **0.83 percent** accuracy, but with a **instantaneous training**. Thus, the final used algorithm was the Naive Bayes classifier.
For the subject and object identification, I tried using similar approaches, with recurrent neural networks or using POS tags and semantic trees. A method that gave very good results despite being simple was a bag-of-words approach: I collected the frequent words in the questions and then eliminated them from each new query. This approach left only the "unusual" words in questions, that very often lead to the detection of subject and object. This method reached around**80 percent** of accuracy on a sample of 100 questions from the dataset. The performance are lowered mainly due to some noisy entries in the dataset or quite ill-posed questions.
On a sample of 300 questions from the leveraged dataset, the overall accuracy of the algorithm was **65 percent**. This can look like a under-achieving results, but it is mainly due to incorrect formatting in the entries, that didn't allow the algorithm to find a match, as discussed with some colleagues of mine working on the project as well.
The performance on natural language question asked by the user looks quite satisfying on the other hand, with a good robustness on both the understanding

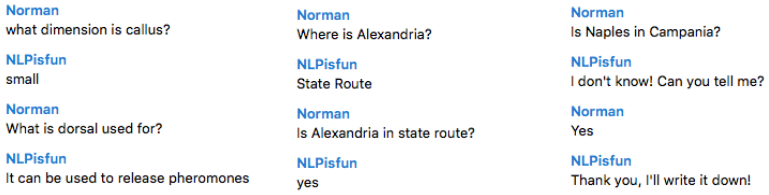

Figure 1: (left) Example of questions and answers. (center) Example of different answers retrieved by the same entry. (right) Example of enrichment phase for the chatbot.

```
Is Naples in Campania?  Yes     PLACE   Naples  Campania
<Response [200]>
```

Figure 2: Terminal output for the enrichment phase, with entry explicity shown.

part and the retrival part, as well as the enriching phase of the database.
You can see some examples in the figures I inserted here.

4 Extra work: virtual assistant and conversational model

In this section I will explain the two main parts that I developed as extra work, as suggested in the project slides. I decided to try to develop the tasks described for who didn't submitted the homeworks, because I was interested in trying to figure out how to accomplish those goals.
The first part is the **virtual assistant**: I developed an additional part to the chatbot that allows it to take note of upcoming events of the user, in order to notify him in the exact date and time. To do so, I created a data structure of the year, composed by months and days, all of which are dictionaries. The chatbot can understand the time and the subject of the event using *api.ai*, a natural language api. Those informations are then sent to the information extraction algorithm and sent to the data structure. Each second, the program checks if in that particular year-month-day the user has an event corresponding to the current time. The chatbot can handle different events for the same day. Some examples are show in the figures.

Additionally, I wanted to implement a conversational model to the chatbot, so that it would have been able to discuss randomly with the user if desired. I decided to implement a **seq2seq** model, which is widely used in language translation and conversational models. As a training set, I used the *Cornell Subtitles Corpus*, along with **SMS messages corpus from Seoul University**. As input and outputs, I used one-hot vectors of the dimension of the vocabulary, around 20.000 words, but in many cases the training was hard to tune and didn't

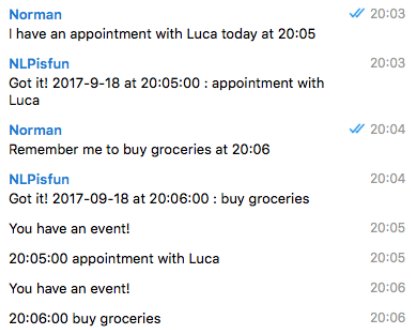

Figure 3: Example of virtual assistant task for the chatbot. Notice how it handles different kind of requests, and can manage multiple events.

converge to anything significant. I used then a word-embedding as input of the model, thus reducing the one-hot vector into a 300 dimensional dense vector, while keeping the output as before. In this case too, the model didn't converge to good results, and the training process was very slow, having around 20 million parameters. The great amount of time and computational power needed to train a sufficiently large model on a large enough corpus made it difficult to run many experiments and try different architectures and hyperparameters.
I implemented another type of model, a **language modelling LSTM**, that trained on the same corpus as before learned to generate, at a character level, the replies given the user's messages. In this case, too, it was quite hard to make the model converge to interesting results.
Finally, I decided to implement a **retrieval based model**. I created a small dataset of different messages and questions, with the relative answer, inspired from casual conversations. In this kind of model, the algorithm retrieves a pre-made answer from the dataset based on the context of the user message. To encode the messages, I transformed each word in the user message into **word2vec vectors**. Then, I added up all the sentence vectors and divided by the number of words, thus making an average of vectors that could encode the meaning of the sentence. Other techniques used for this kind of models are **LSTM encoders** that reads the word vectors sequentially and outputs a vector that represents the embedding of the sentence, using it to retrieve an answer. In my model, I trained a fully connected neural network to relate the embedded sentences with an answer from the dataset, thus making it a classification problem. Even if the training set was very small, the results were quite good. The bot can answer questions that are similar to the ones it was trained to, and since the answers are retrieved from the dataset there are no syntax or semantic mistakes. I believe that with a larger dataset the behaviour could me much better. The conversation model is not included in the normal chatbot routine because it requires loading word2vec, which needs a lot of time, thus I prefered to use it as a separate Jupyter Notebook.