INTENT RECOGNITION ORIENTED CHATBOT

NLP PROJECT

García de Viedma Pérez Lucas

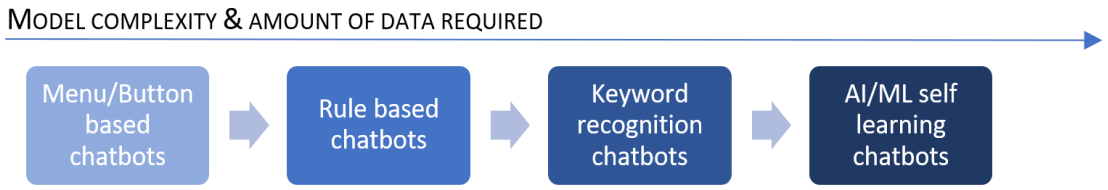
INDEX

The report is 5 pages long because it includes some images that occupy around 1 page of space in total.

LINK TO GITHUB: https://github.com/LucasGviedma/NLP-project

INTRODUCTION

The goal of this project was to build a **chatbot** using the **NLP techniques** we have learnt during the course. After a brief investigation on the field, I found that chatbots could be grouped in **4 iterative categories**:



- **Menu/Button based chatbots:** the most basic type of chatbots. Implemented with a structured decision tree that they use to show the users a set of buttons with different options. Depending on which button they click on, the chatbot will show a different set of options or show the required information. They can be used in FAQs scenarios.
- **Rule based chatbots:** also called linguistic chatbots. This group is characterized by the use of logic flows and conditions to determine the answer the bot will give to the user. However, since they don't have a set of forced inputs as the "menu based" ones, these bots won't be able to answer a query that is not included in their logic schema.
- **Keyword recognition chatbots:** unlike the previous types of chatbots, this category utilizes NLP (natural language processing) to analyze the user's input and answer appropriately. As other machine learning approaches, these bots require a previous training that will help them depict a series of keywords that they will use to discover the user's needs.
- **AI/ML self-learning chatbots:** similar to the previous category due to the use of AI/ML, these chatbots not only use keywords to determine what the user wants, but also utilize the contextual information to store/learn other elements of the query (f.e. the food that the user likes to order) and use it to give suggestions or to improve their "services".

If we consider the **input/output** of the different approaches they could also be divided in **text chatbots** or **voice chatbots**.

PROBLEM SELECTION

Since we haven't covered the most complex category of the chatbots, I decided to build one following the **keyword recognition approach**. To do so, I did a further investigation and finally determined to create a chatbot that simulated a **virtual assistant** which used **intent classification** to address the user's requests.

The chatbot has been programmed in R, and the main libraries that have been used are:

- **Keras and tensorflow:** for building the model.
- **Tm and texstem:** as providers of an English stop words corpus and the management of them.

DATA SELECTION

After analyzing different datasets, I decided to use the one provided by the SONOS company, which classifies a total of 14484 texts/queries in 7 different intents: AddToPlaylist, BookRestaurant, GetWeather, PlayMusic, RateBook, SearchCreativeWork and SearchScreeningEvent. The dataset was initially divided in 3 sub datasets – train.csv, valid.csv and test.csv – but I decided to merge them together to decide myself the percentage of data used in each step of the process.

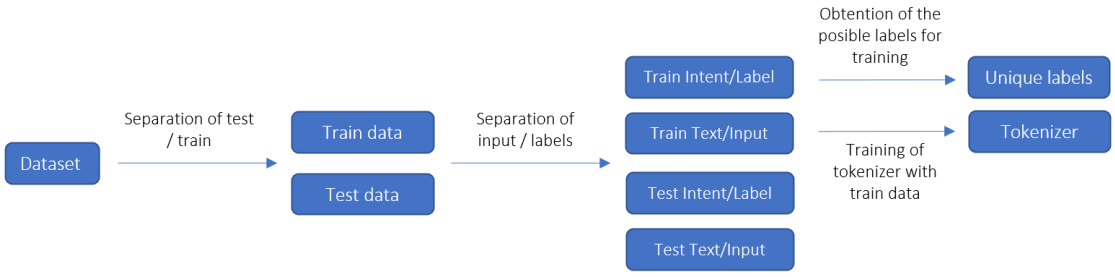
MODEL BUILDING PROCESS AND EXPERIMENTS

Data cleaning and normalization

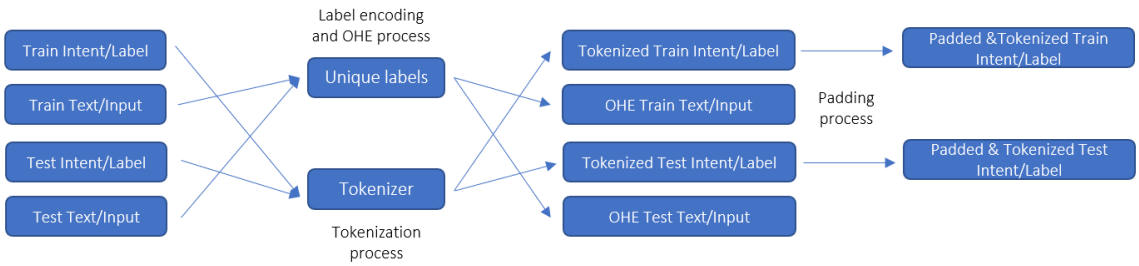
For the cleaning of the data I removed all the elements in the texts that weren't letters, numbers or the ' symbol, fixed some formatting errors in the data (words like "I'm" had been written as "I m"), removed the existent stop words and lemmatized the output of this process. Finally, I set all the resulting texts to lowercase.

Data preprocessing

Firstly, I divided the dataset in a 70% training set and 30% test set and separated the input from the label of the classification, which in this case are the inputted text and the determined intent. I used the training dataset to train the Tokenizer (I didn't use the test data to simulate a more realistic evaluation later on) and to obtain the possible labels our model might encounter in the process.



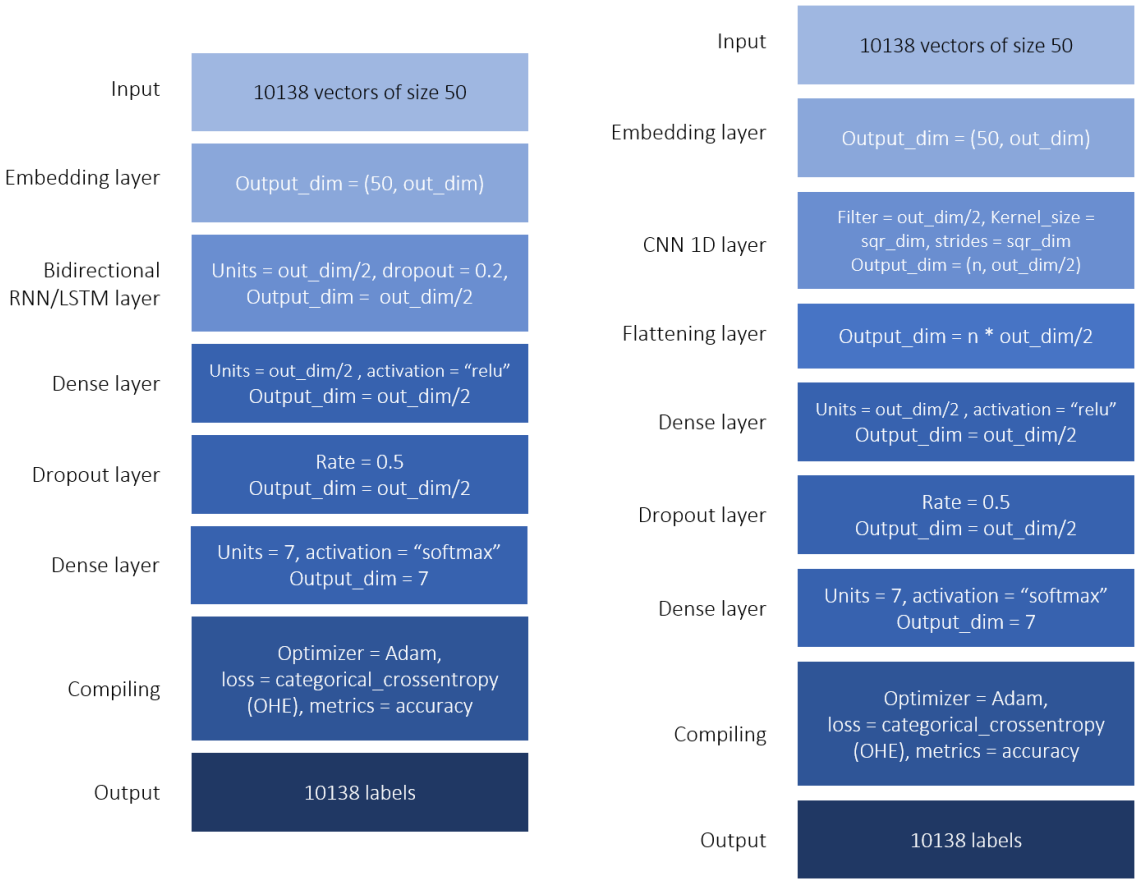
Once I had these two elements, I used the "unique labels" to one hot encode the labels of both the training and test datasets, and the tokenizer to tokenize the input of both models too. To unify the size of the input vectors resultants from the tokenization, I also applied a padding of zeros on all of them, setting their size to 50 tokens.



Model selection

Analyzing the different approaches that are followed in the ML domain for summarizing the main features of the inputs (in this case, finding and extracting the **keywords**) I found two different options: RNNs, which we have seen during the course, and CNNs, which are commonly used to summarize features in domains such as image labeling. I was curious on comparing their

performance in the task of intent classification, so I decided to build three models: one with a generic **RNN**, another one with a **LSTM** and a final one with a **CNN**. Their structure can be seen below:



Logic behind the models

Common layers

- **Embedding:** the embedding layer is responsible for generating high dimensional weighed vectors that will represent the tokenized vectors we introduce in the model. They are the basis for comparing different vectors and learning the similarities and key words that they share. The length of the embedded vectors is determined by the value: **out_dim**.
- **Dense layers:** we can see two dense layers in both models, with the same dropout layer in between. However, their goal is quite different. The **first layer** uses a **RELU** as activation function since it allows for a faster training than others, like the sigmoid, and has half the neurons of the previous layers (**out_dim/2**) to try reducing a bit the cost of the training. After it we have the **dropout layer**, which will **set to 0 half of the previous layer's neurons** to avoid overfitting. And finally, we have the **last dense layer**, with **7 neurons**, that has the objective of calculating how probable is that the input belongs to one of our 7 labels (that's why we use **softmax**).
- **Compiling:** For the compiling, I picked the **Adam** optimizer (sparse gradient descent) because I saw that, together with the general SGD, is the one that obtains better results. The **categorical_crossentropy** election for the loss was due to the One Hot Encoding of the labels and the selection of **accuracy** was due to the nature of this problem, which is a classification.

RNN model layers

- **Bidirectional RNN layer:** as we have seen in the lectures, RNNs are used in NLP because they have a short time memory that helps in analyzing the relation between the tokens that are inputted in the model. Included in a bidirectional layer we consider both towards and backwards short time relations. Since in this layer we actually use two RNNs, I decided to use half the number of neurons of the previous layer for each of them: **out_dim/2**. This layer also has a small dropout rate to avoid overfitting.

LSTM model layers

- **Bidirectional LSTM layer:** although similar to their generalized version, the RNNs, these networks' structure allows them to take into account more relations between the tokens of the vectors that we input (and therefore more context). Included in a bidirectional layer we consider both towards and backwards relations. Since in this layer we actually use two LSTMs, I decided to use half the number of neurons of the previous layer for each of them: **out_dim/2**. This layer also has a small dropout rate to avoid overfitting.

CNN model layers

- **CNN 1D layer:** convolutional networks are an alternative for feature extraction. However, the process is quite different from the one in RNNs. Since the performance of this network depends a lot on the parameters established on the **filters, kernel size and strides**, I included them in the hyperparameters selection process together with the **number of neurons**.
- **Flattening:** one of the characteristics of CNNs is that they alter the dimensionality, reason why a flattening layer is required to return the data to its previous dimension.

Hyperparameter selection and analysis of results

As it was shown in the previous section, I've represented the number of neurons with a variable called **out_dim** (output dimension basically), which is used to calculate the number of neurons of each layer (and thus the **filter** in the CNN). Also, for the CNN we will represent the **kernel_size** and **strides** with the variable **sqr_dim**, since we will use the same value for both of them.

I performed a total of **25 experiments** with **each** of the following values for the hyperparameters and calculated the mean loss and accuracy obtained in the evaluation of the resulting models (we will only consider the **accuracy**):

RNN parameters

out_dim	avg_loss	avg_acc
128	0.1513414	0.9506903
256	0.1619580	0.9488495
384	0.1635982	0.9510584
512	0.1733697	0.9483893

LSTM parameters

out_dim	avg_loss	avg_acc
128	0.1383707	0.9567878
256	0.1350218	0.9582145
384	0.1398952	0.9584445
512	0.1434449	0.9570180

We can see that, in average, the **LSTM version performs slightly better** than the simple RNN model, obtaining the **best performance with 384 neurons**.

CNN parameters

out_dim	sqr_dim	avg_loss	avg_acc
128	1	0.1456713	0.9578233
128	3	0.1521093	0.9546249
128	5	0.1522206	0.9533364
128	7	0.1546812	0.9537045
256	1	0.1500069	0.9573631
256	3	0.1594114	0.9544639
256	5	0.1595953	0.9549931
256	7	0.1701113	0.9503221

out_dim	sqr_dim	avg_loss	avg_acc
384	1	0.1587491	0.9549241
384	3	0.1660594	0.9523930
384	5	0.1696167	0.9511505
384	7	0.1804352	0.9493787
512	1	0.1706811	0.9554993
512	3	0.1791214	0.9525081
512	5	0.1922997	0.9476070
512	7	0.1879416	0.9489185

I was surprised to see that, even though the **CNN average and best performances** are **worse than the ones obtained by the LSTM**, there were multiple occasions where they **outperformed the RNNs**.

Nevertheless, the model that obtained the **best overall performance** was the **LSTM** based model with **384 neurons**.

I didn't consider using BERT because I thought that using it would not have a lot of merit since it's already trained, and the combinations of the best parameters have been already determined.

CONCLUSION

While it's true that we have found a model that performed better than the others that were tried, the difference in between them is very small, and might have to do with the reduced size of the dataset (around 14k values), the small number of labels to be predicted, 7 labels, and the similarity between the training and testing inputs.

However, the goal of this project, which was to build a chatbot with R though intent recognition has been achieved with a 95.8% of accuracy.

CONVERSATION

The conversation of the chatbot is quite plain. I've just created a default response that basically states the predicted intent of the user. Also, when the chatbot detects words that are not included in the tokenizer, he will predict the intent of the user and then ask whether he succeeded or not. In the case he did, he will include the new input in the dataset together with the detected intent. If he fails, he will ask the user which intent was being assessed and will also include it in the dataset with the correct value. Since the bot will include these elements in the dataset, that might be used for a future retraining for a better model, try to write the queries as properly as possible. Saving the values can also be avoided saying that the prediction was incorrect and selecting the IGNORE option when the chatbot asks for the right intent.

Remember that the chatbot will only give a "logic" answer to queries around the topics: AddToPlaylist, BookRestaurant, GetWeather, PlayMusic, RateBook, SearchCreativeWork and SearchScreeningEvent.

For example: I want to eat something, let's order some food, find the film Ratatouille, punctuate this book with a seven out of ten, let's put some music for the party, etc.