AIML CHATBOT INTEGRATED WITH WORDNET

Jacopo Dapueto

s4345255@studenti.unige.it

1: Introduction

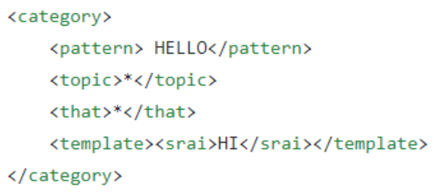
A chatbot is a conversational agent which allows a natural interaction between a machine and the user. Exploiting AIML[2] (Artificial Intelligence Modelling Language) it is possible to implement a chatbot by defining patters that the software agent is able to recognise and associate with them the answers, this kind of tool is suitable for defining a limited number of patterns and it makes very difficult to cover all possible variants of a sentence in a given natural language.
This project provides a chatbot that should work in the tourism domain as support to a website of a tour sharing community, where users can propose guided tours and participates to them to share the costs. The program is able to answer to common questions and to help the users to find the activity that suits them best to make the conversation more engaging.
To cope with the limitation of AIML the chatbot is integrated with the lexical database WordNet[3] so that the program can learn new patterns by recognising the synonyms, hyponyms and hypernyms of words that the chatbot is already able to recognize.

2: Tools and Usage

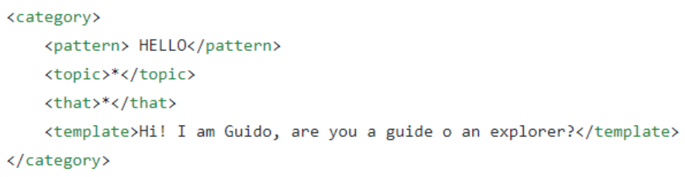
2.1: Tools

2.1.1: Program-Y

The entire software is written in Python 3.8.5 and the chatbot itself is implemented thanks to the Program-Y[4] library. Such library provides an interpreter fully compliant with AIML 2.1 plus some of the Pandorabot[5] tags. However the two tools behave differently with some nested combinations of tags, in fact Pandorabot results more flexible in the way the tags can be used to make working combinations: for example in Pandorabot the <*eval*> tag can be used nested in a <*srai*> one (to make the symbolic reduction explicitly used in learned categories), meanwhile Program-Y doesn’t support it and this affected the way the learned categories are produced. In the latter case the learned categories can only directly evaluate the reduction before the saving: the two following images shows the same learned category in Pandorabot and in Python of the word *HELLO* that should be reduced to the already existing category *HI,* the <*template*> tag on the left evaluates each time the reduction and so any modification of the pattern *HI* will be handled automatically, by contrast the <*template*> tag on the right evaluate the reduction at the time of the learning and making any future changes not available.



```
<category>
    <pattern> HELLO</pattern>
    <topic>*</topic>
    <that>*</that>
    <template><srai>HI</srai></template>
</category>
```



```
<category>
    <pattern> HELLO</pattern>
    <topic>*</topic>
    <that>*</that>
    <template>Hi! I am Guido, are you a guide o an explorer?</template>
</category>
```

Figure 1- Learned category in Pandorabot

Figure 2- Learned category in Program-Y

On the other hand Program-Y is pretty extensible and it allows to define your own AIML tags, dynamic sets and maps through Python classes but it would required a deep understanding of how the tool works, but the documentation doesn't provide enough informations to implement my own tags.

As for Pandorabot the tool normalizes the input of the user:

- It corrects some spelling errors;
- It substitutes some colloquialisms and abbreviations.

Normalization also splits sentences based on predefined punctuation characters such as full stop, colon, semi colon and brackets. Once a sentence is split, Program-y parses each sentence and then combines all the responses back into a single text output. Such normalization steps should be done before starting process the texts to properly learn new categories.

2.1.2: WordNet

WordNet is lexical database where words are grouped together according to their meaning in sets called *synsets*. the words in a *synsets* are synonyms and a word can appear in different *synsets* because it can have different meaning in different context: the *synsets* represent synonymy relations among their members.
In addition, every *synset* has its own hypernym (which is a *synset* itself) and a list of hyponymy. These relations are exploited by the chatbot to look for possible substitutions in a sentence.
For example the word "*architecture*" belongs to a *synset* of "*the discipline dealing with the principles of design and construction and ornamentation of fine buildings*" which has as hypernym "*discipline*" and as hyponyms "*interior design, landscape architecture, urban planning*".

2.1.3: NLTK

NLTK[6] is a platform for building Python programs proving access to many corpora and databases and it includes many functionalities to process text data, in this project it's used to access the WordNet database and to process the sentences of the user.

2.1.4: Folder structure

In the main folder of the project there are the Python programs:

- *main.py*: the file that load the chatbot from the config file and that starts the program;

- *wn_integration.py*: the file containing the functions that access WordNet and to process the text;
- *utils.py*: contains the lists of terms that can be substituted and the patterns required to check the behaviour of the bot.

The folder *y-bot* contains the all the files necessary to Program-Y to load the chatbot with all the categories, sets etc. , through the *config.yaml* file. In particular in the folder *storage* are stored all the basic elements required, such as the categories and the sets. The learned categories are located in *y-bot/storage/categories/learnf*.

2.2: Usage

As specified before the application is implemented with Python 3.8.5 and the following libraries are required, with their associated versions:

- Program-Y: 4.3.
- NLTK: 3.6.2.

In the main folder of the project there is a file *requirements.txt*, which contains the list of required packages and can be used to prepare the environment to execute the program with the following command:

```
pip3 install -r requirements.txt
```

In addition to the libraries, the program requires the lexical database already downloaded, it should be downloaded with the command:

```
python3 -m nltk.downloader wordnet
```

Now the code can be executed through the terminal locate in the main folder with the command:

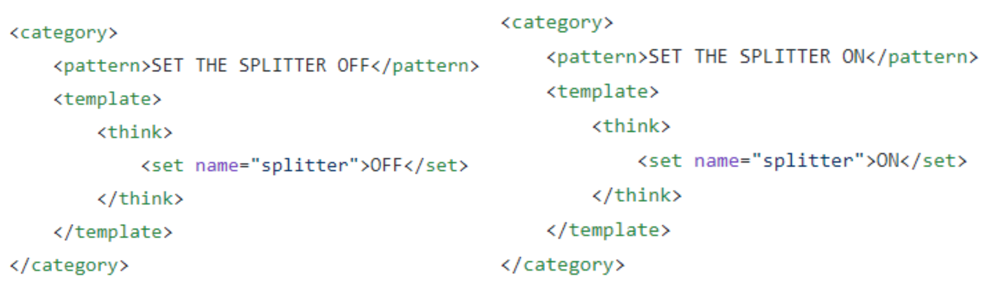
```
python3 main.py
```

3: Software explanation

3.1: Extensions of AIML

Before describing the code itself there is the need to explain some extensions used to define the categories that are not present in Pandorabot:

- **Dynamic variables**: are variables which, instead of loading static content from the bot, are a Python object and they are loaded through config file. To properly implement the stage of learning of the new terms it needs to temporarily disable the *splitter* that takes the sentence typed by the user and divide it into different sentences according to the punctuation marks.



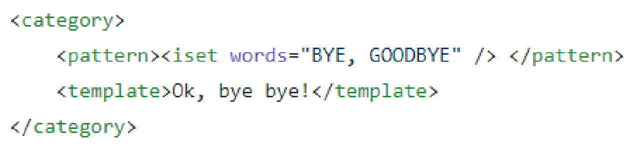
```
<category>
    <pattern>SET THE SPLITTER OFF</pattern>
    <template>
        <think>
            <set name="splitter">OFF</set>
        </think>
    </template>
</category>
```

```
<category>
    <pattern>SET THE SPLITTER ON</pattern>
    <template>
        <think>
            <set name="splitter">ON</set>
        </think>
    </template>
</category>
```

Figure 3-Disabling and Enablig the splitter functionality

The variable "*splitter*" is a dynamic variable associated with the customizable class that split the sentences, such behaviour can be enabled or disabled by setting the values *ON* and *OFF* respectively through the AIML tags.

- **Inline sets (iSets)**: are sets that rather than specify the collection of words in a file, they can be specified in line to the XML. They have a small number of member and the matching is the same as for sets in files, the word is checked for membership of the iset words listed. For example:



```
<category>
    <pattern><iset words="BYE, GOODBYE" /> </pattern>
    <template>Ok, bye bye!</template>
</category>
```

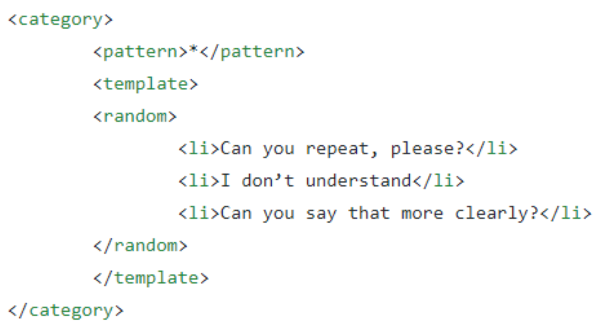
Figure 4-Example of iSet

The grammar will match either *BYE* or *GOODBYE*, providing the same answer for each word.

3.2: Integration with WordNet

3.2.1: Basic idea

The algorithm starts by normalizing and splitting the input into sentences and each of them is threatened as a single input.
Each substring has been checked if it matches one of the patterns present in the chatbot brain, actually the chatbot always answers the user thanks to the Ultimate Default Category.



```
<category>
        <pattern>*</pattern>
        <template>
        <random>
                <li>Can you repeat, please?</li>
                <li>I don't understand</li>
                <li>Can you say that more clearly?</li>
        </random>
        </template>
</category>
```

Figure 5-Ultimate Default Category of the chatbot

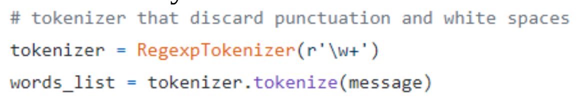
So the three possible answers (see Fig. 5) randomly selected are considered as the bot cannot properly respond to the user, and in this case the algorithm looks for substitutions of words that would match a category already defined, and if one is found it makes the chatbot learn the sentence as a new category reduced to the existing one. To perform the substitutions the algorithm looks for lexical relations for each (synonymy, hyponymy, hypernymy) words used the in input.

3.2.2: Words to substitute

Since AIML 2.0 allows to capture many inputs with one single category using the *wildcards*, it makes sense to look for lexical relations within a set of "ground" words that play a major role in the domain of interest and with words that are explicitly stated in the categories, also for an efficiency purpose.
The list of words is divided by the Part of Speech by hand, and it results in three list: one list containing the Nouns, one list containing the Verbs and one for the Adjectives. Having the words divided come handy to inspect the Wordnet database considering only the synsets with a specified grammar category.
Moreover the words that make up the user's input are lemmatized according to the POS to make easier the matching with the lemmas.
Such predefined lists are in the *TermsOfInterest* class, in *utils.py*.

3.2.3: Preprocessing

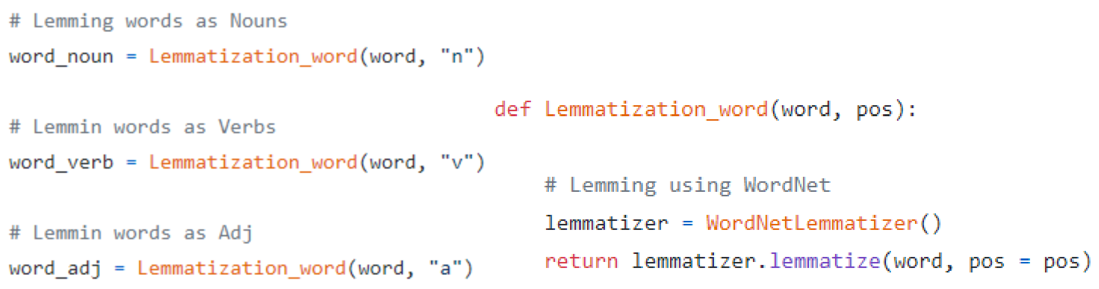
After the sentence has been normalized, the words that compose it are extracted with a tokenizer provided by NLTK library.



```
# tokenizer that discard punctuation and white spaces
tokenizer = RegexpTokenizer(r'\w+')
words_list = tokenizer.tokenize(message)
```

Figure 6-Tokenizer

The list contains only words without white spaces and the punctuations.
WordNet is not only a lexical database but it offers lemmatization capabilities as well and the procedure exploits them to normalize the words extracted to make easier the inspection of the Database: each word is lemmatized thanks to WordNet lemmatizer whose interface is provided by NLTK, and this will make sure to find always a synset.
To properly preprocess the words, the lemmatization is performed according to the grammar categories: *noun*, *verb* and *adjective*.



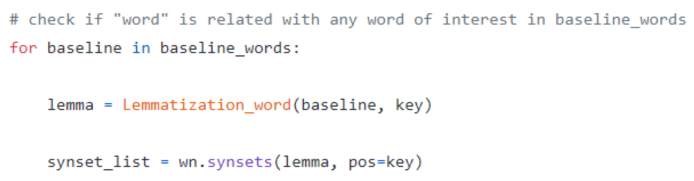
```
# Lemming words as Nouns
word_noun = Lemmatization_word(word, "n")

# Lemmin words as Verbs
word_verb = Lemmatization_word(word, "v")

# Lemmin words as Adj
word_adj = Lemmatization_word(word, "a")
```

```
def Lemmatization_word(word, pos):

    # Lemming using WordNet
    lemmatizer = WordNetLemmatizer()
    return lemmatizer.lemmatize(word, pos = pos)
```

Figure 7-Lemming according to the POS

3.2.4: Synonymy relation

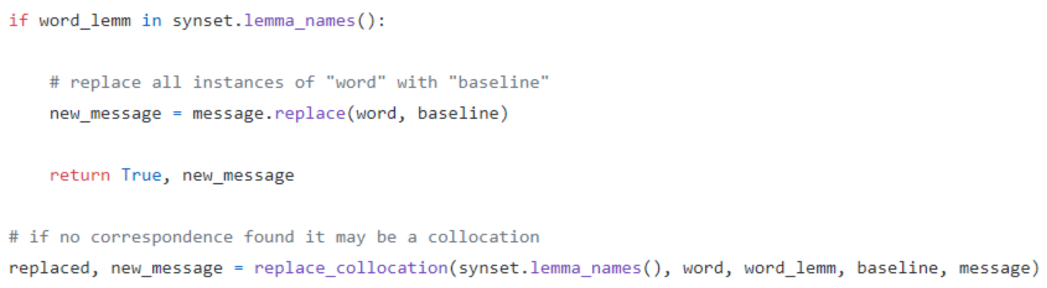
After the preprocessing, all the synsets of “ground” terms (also them are lemmatized to be sure to find them in the database) in the predefined lists are obtained.



```
# check if "word" is related with any word of interest in baseline_words
for baseline in baseline_words:

    lemma = Lemmatization_word(baseline, key)

    synset_list = wn.synsets(lemma, pos=key)
```

Figure 8-Get the synsets of a ground term

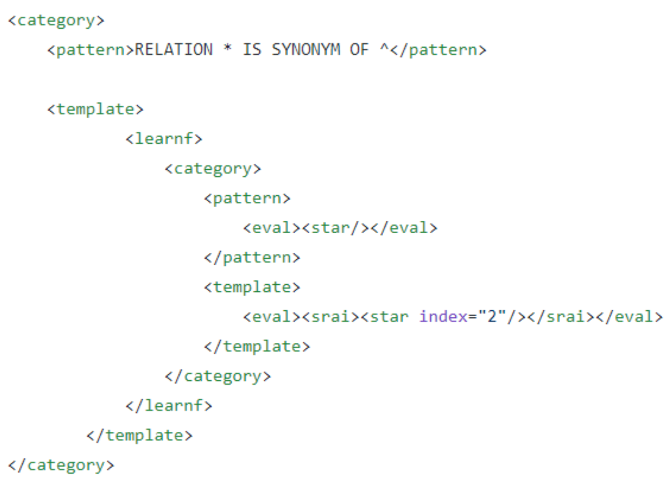
At this point each word in the input is checked if it belongs to one of the synsets found previously, and if a word is a member of a synsets then it is substituted with the ground term that “generate” the corresponding synset.



```
if word_lemm in synset.lemma_names():

    # replace all instances of "word" with "baseline"
    new_message = message.replace(word, baseline)

    return True, new_message

# if no correspondence found it may be a collocation
replaced, new_message = replace_collocation(synset.lemma_names(), word, word_lemm, baseline, message)
```

Figure 9-Substitution is the word is a lemma in the synset

First such stages are performed to find a suitable substitution with the terms in the list of nouns, if no one is found then the program execute the very same steps but with the verbs and finally it tries with the adjectives.
Once composed the “basic” sentence with the “ground” word, the algorithm check whether this matches one of the meaningful categories and in case of positive result the relation is permanently saved in the brain of the chatbot.



```
<category>
    <pattern>RELATION * IS SYNONYM OF ^</pattern>

    <template>
            <learnf>
                <category>
                    <pattern>
                        <eval><star/></eval>
                    </pattern>
                    <template>
                        <eval><srai><star index="2"/></srai></eval>
                    </template>
                </category>
            </learnf>
        </template>
</category>
```

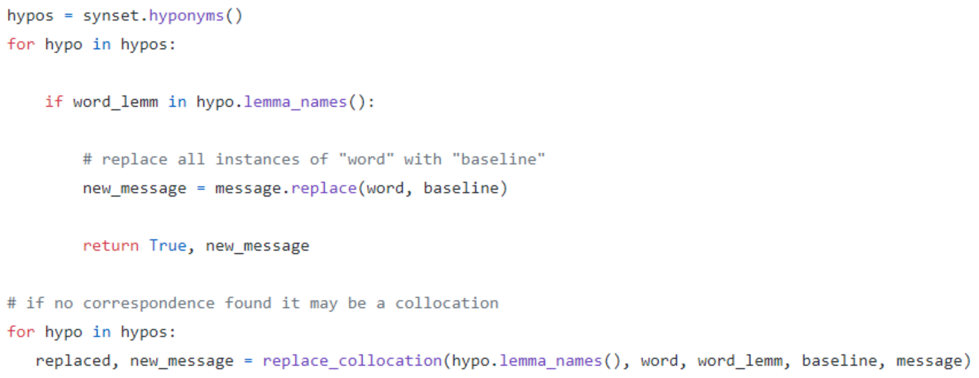
Figure 10-AIML category used to save the new relation

The *category* above is used to permanentely save the relation.

The first *star* wildcard contains the sentence typed by the user and the second one is the sentence that the chatbot is already able to recognize. Thanks to the symbolic reduction the new sentence will be reduced to the already existing category.
In case a substitution is found but it leads to a pattern that is not already in the brain, the algorithm looks for synonymy relations with the remaining ground terms.

3.2.5: Hyponymy and Hypernymy relations

If no synonyms are found for any grammar category (noun, verb, adjetive) the procedure tries to look for hyponyms and hypernyms in a similar way.
For each synset, the database is queried for the hyponyms of that synset and if a word of the sentence is in one of the hyponyms it is substituted with the terms that generate the synset.



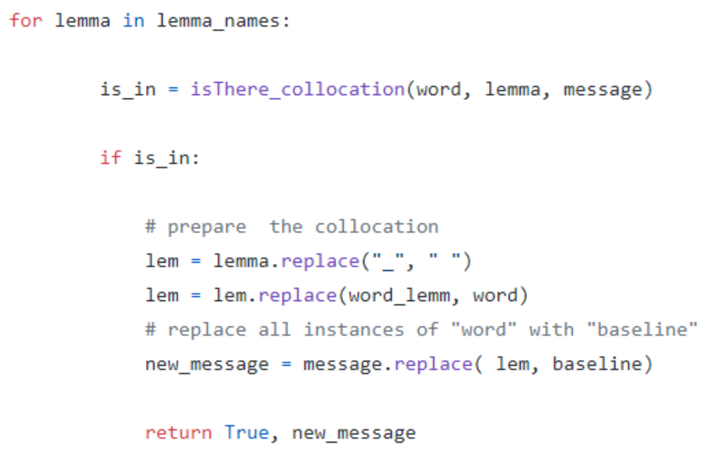
```
hypos = synset.hyponyms()
for hypo in hypos:

    if word_lemm in hypo.lemma_names():

        # replace all instances of "word" with "baseline"
        new_message = message.replace(word, baseline)

        return True, new_message

# if no correspondence found it may be a collocation
for hypo in hypos:
   replaced, new_message = replace_collocation(hypo.lemma_names(), word, word_lemm, baseline, message)
```

Figure 11-Substitution if the word is a hyponym

If no hyponymy relations are found, then the program looks for hypernym relations following the same steps. They differ only in that a synset has only one hypernym and but can have more than one hyponym and so the hypernym is not a list.
If there are no hypernymy, then the input cannot be learned and the chatbot respond properly saying that it doesn’t understand.

3.2.6: Collocations

A collocation is a sequence of words or terms that co-occur more often than would be expected by chance.
Since it’s a sequence of words they must be treated differently: they appear in WordNet as a single string where the words are separated by an underscore, hence it should be removed.
If a collocation, that is a lemma in a synset, is found in the input’s user, it is substituted with the ground term to make the basic message the chatbot is able to recognize.



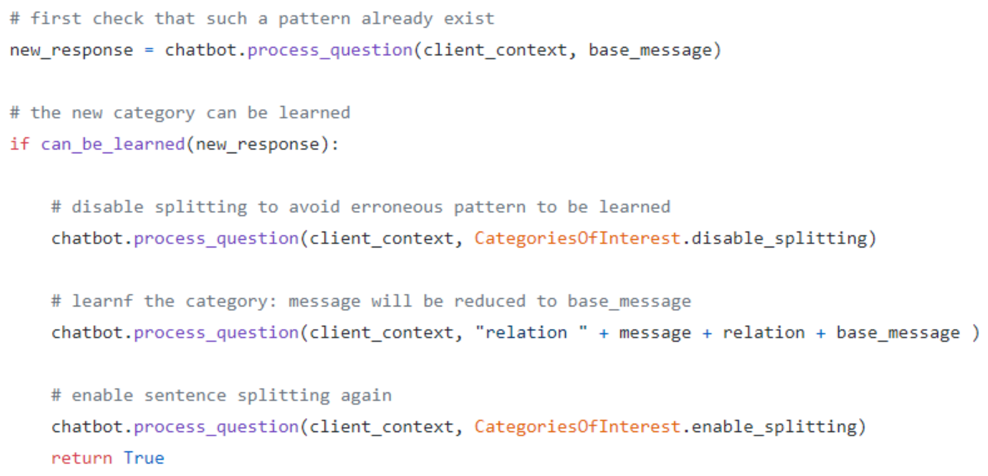
```
for lemma in lemma_names:

        is_in = isThere_collocation(word, lemma, message)

        if is_in:

            # prepare  the collocation
            lem = lemma.replace("_", " ")
            lem = lem.replace(word_lemm, word)
            # replace all instances of "word" with "baseline"
            new_message = message.replace( lem, baseline)

            return True, new_message
```

Figure 12-Substitution of a collocation

Such procedure is performed for every lexical relation in the very same way, except that the collocations are taken from the hyponyms and the hypernyms and not directly from the basic synsets.

3.2.7: Learning

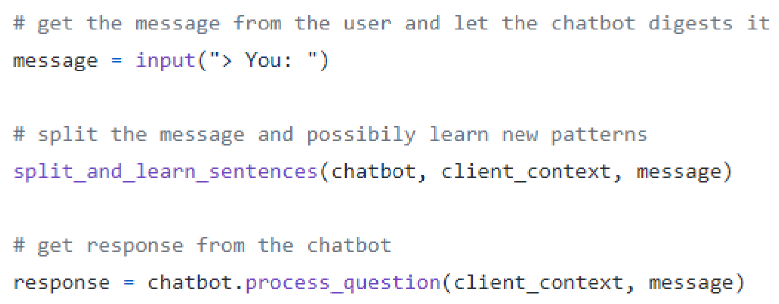
Once the algorithm found the relation and generate the sentence with the substitution, the latter is tested and if it triggers a meaningful category, the sentence can be learned. To do first the splitter of the chatbot disable to prevent the it from digesting a substring indipendently and trigger a pattern not requested.



```
# first check that such a pattern already exist
new_response = chatbot.process_question(client_context, base_message)

# the new category can be learned
if can_be_learned(new_response):

    # disable splitting to avoid erroneous pattern to be learned
    chatbot.process_question(client_context, CategoriesOfInterest.disable_splitting)

    # learnf the category: message will be reduced to base_message
    chatbot.process_question(client_context, "relation " + message + relation + base_message )

    # enable sentence splitting again
    chatbot.process_question(client_context, CategoriesOfInterest.enable_splitting)
    return True
```

Figure 13-Learning process

At the end the original message is passed to the engine of the chatbot since all the new patterns should be recognized immediately.



```
# get the message from the user and let the chatbot digests it
message = input("> You: ")

# split the message and possibily learn new patterns
split_and_learn_sentences(chatbot, client_context, message)

# get response from the chatbot
response = chatbot.process_question(client_context, message)
```

Figure 14-The main code that ask the user a message, process and possibly learn it and then retrieve the response of the chatbot

4: Conclusion

The chatbot started with about 50 basic categories and it generates 132 new categories by using it, at the end it is made up of 220 patterns.
The AIML categories are added gradually as the users ask questions to it but the learning is limited by the synsets of Wordnet that usually don't take into account the tiny differences between lemmas. Furthermore the code looks for synonyms in all the synsets and so there is the possibility that a terms used it might not be a synonymy (or hyponymy, hypernymy) in the context where it is used, but this problem is mitigated by the fact that a term is learned if and only if a user use it and it can be assumed that the usage of a term implies a sort of correctness because of the variability and the evolution of the language overtime, that cannot be taken into account in grammars defined with AIML.
A weakness of the project is that if the set of "basic" categories is extended, then the list of terms that can be substituted must be updated.

4.1: Possible improvements

The software can improved by adding the possibility to access a Thesauri database, not only to exploit the lexical relation between terms but also filter words and synsets that not belongs to the domain of interest.
Moreover it might be useful understand how the chatbot can extended with new tags to make the learning similar to what PandoraBot does and also supporting the <*that*> and <*topic*> tags that cannot be easily integrated in the learned category.

References

[1] GitHub project:https://github.com/LazyRacc00n/ChatBot .

[2] AIML: http://www.aiml.foundation/.

[3] WordNet: https://wordnet.princeton.edu/.

[4] Program-Y: https://github.com/keiffster/program-y.

[5] Pandorabot: https://home.pandorabots.com/home.html.

[6] NLTK: https://www.nltk.org/.