The Chatbot Project (IVANA)

Abhishek, Nishit, Nikhil, Parth, Vinayak

19May 2015 - 24June 2015

Contents

1 Introduction **2**

2 Technology used **2**

3 How it works **2**

3.1 The structure of the Database 3
3.2 Finding N 3
3.3 Finding B 4
3.4 Finding W 4
3.5 Question Answering 5
3.6 Graphical User Interface (GUI) 6

4 Timeline **6**

5 Future Plans **8**

Ivana: Hello
You: Who are you?
Ivana: I am Ivana. I like talking to people.
You: Nice to meet you.
Ivana: Nice to meet you too.
You: bye

This is Ivana. She's a young girl, who absolutely loves talking to people. Won't you like to come along and have a chat?

1 Introduction

A chatterbot, or chatbot, is a computer program that chats with a human without the intervention of any other being. We came up with the idea of building a chatbot, taking inspiration from Cleverbot(cleverbot.com), which is widely successful. Our chatbot is called Ivana. It follows the one-sentence-each chatting convention, where both the user and the bot communicate with one statement at a time.

2 Technology used

Ivana is Python based, and the database is maintained through SQLite3. It makes use of many features of the Natural Language Toolkit (NLTK) available as a python library, to process the user input and assign a relevant response to it.

3 How it works

Ivana starts off with a 'hello', and then waits for the user to enter an input. Then, it processes the input and displays the best possible response that it could find. Then, it waits again for the user input, and the chat continues in this fashion, until the user types 'bye'.

To find the best possible response, Ivana looks up the input statement in its database, under the column 'Input' in the table 'sents' to find an exact match. If such a match is found, it displays one of the (possible) three responses stored against that input using a pseudo random number generator to select the response. If an exact match is not found, it tries to find the

most appropriate approximate match from its database of inputs using three algorithms one by one. The way this is done is described below:

3.1 The structure of the Database

The database has two important tables - 'sents' and 'words'. Both have a column known as 'id' (type int), which is the primary key. The 'sents' table has a column for possible Input, and three columns for storing the relevant responses for a particular input - Response1, Response2 and Response3. An Input in the table may have zero or more responses stored for it.

The 'words' table has 26 columns. The first is id, followed by four for nouns, three for adjectives, three for verbs, one for modals, two for adverbs and one for 'wh' type words like how, why, when and where. Other columns include sentkey, new, rowtot, denom, similar(1-4), W, B and total. The purpose that these columns has been explained later.

3.2 Finding N

There are three parameters determined by our program, which eventually decide, which of the Inputs from the existing database is the closest in meaning to the given input. Each of them is determined by an algorithm in a systematic way, and they serve different, but complementary purposes.

The first measure is N, which checks for a similar sentence structure to the one in the given input. It also tries to find common words, and assign them weights according to their popularity in the English language.

This works as follows: NLTK has a Part-of-Speech (POS) tagger, trained by the n-grams method over huge corpora of English language text. The program first tokenizes the input sentence S into words, then uses this POS tagger to tag these words. Then, these words are put into their respective categories (noun (including pronoun), verb, adjective, adverb, 'wh', and modal in the database. There is a room for a maximum of four nouns, three verbs and so on.

Now, the sentence is awarded a 'denom' - a denominator, which is like the total weight that the words in the sentence carry. For this, we took the most abundant and diverse corpora of NLTK (including gutenberg corpus, brown, inaugural, webtext etc.) and made a dictionary having keys as the words occurring in this volume of text, and the values being the number of times the word corresponding to their keys has been used. For example, worddict('a') is around 85700.

We decided the weight of a key to be the reciprocal of its value. Then, each part of speech is assigned a multiplier, which corresponds to its contribution

in the overall meaning of the sentence. For each word in the given input, the weight of the word is obtained, and multiplied by the multiplier of its part of speech tag. All these new weights are added, and form the denom of the input. This is stored against the denom column.

Now, each existing entry in the database is looked over, and for each sentence under the Input column, every tagged word is checked against the similarly tagged words of the new input, and if a match is found, its weight is added to the rowtot of the sentence, initially set to zero. For example, 'Where have you been?' will show a match with 'Where are you now?' for 'where' under the column 'wh'.

Finally, rowtot is divided by denom, and the ratio is stored under rowtot. This is N.

3.3 Finding B

In this part, we try to find similar patterns between input sentence and the database sentences using an algorithm somewhat similar to n-grams algorithm. Each database sentence is assigned weight(ie, value of B) depending upon how much sentence pattern is matched with the input sentence in this way- first of all if there are any words common between the input sentence and a database sentence, then, the position of the word in input sentence is mapped to the position of first occurrence of the word in that database sentence. Using this mapping, we identify and store the number of bi-grams, tri-grams, etc. into a list and use a mathematical formula to compute the weight of the particular database sentence for the given input sentence. The database sentence which gets the highest weight is taken as the one with the most-approximate match. We understand that this method is a bit flawed as it maps to the position of "first occurrence" of word in database sentence. Hence, as the database sentence length increases, the algorithm will have a higher probability of error. To tackle this, we reduce the contribution of the factor B towards gross weight using all the 3 algorithms as the database sentence length increases. We look forward to improving this algorithm further.

3.4 Finding W

This is the part where the meaning of the keywords of the sentence kicks in. Here, we are using the Wordnet provided by NLTK, which is like a graph connecting most of the words of a particular part of speech in the English language. Thus, the path joining any two words can be known, and thus the path length. We have used this feature to measure the path similarity between any two nouns and thus the likelihood of the nouns in a sentence in

the database to be related to the nouns in the new Input. This is an efficient method to match synonyms, hyponyms, hypernyms etc. correctly.

The path similarities of all the nouns in the new input with all the nouns in a database sentence are computed, minimum is selected for each noun and then these are added to get W - our third measure of similarity.

3.5 Question Answering

A separate module has been implemented for answering questions with non-descriptive answers.Keeping in mind the limitations of the database to provide answer to every question, this module generates an appropriate , though not necessarily factually correct, output. The types of questions currently processed are "when.....?" ,"can.....?","is.....?","are....?", "why did you .....?"

Here is a sample interaction illustrating the module.
(code)
Ivana: Hello
You: Can you do me a favour?
Ivana: No ,I can't do you a favour
You:Are you crazy?
Ivana: Yes ,definitely.
You:When is your birthday?
Ivana: My birthday is on 11 May
You:is the sky blue?
Ivana: Yes ,I am sure.
You:Why did you say that?
Ivana: It's a free country.
(/code)

For "can?" , "is?" and "are?" types of questions ,the answering module pseudo randomly chooses to answer negatively ," No, ¡complete sentence with context from the question¿" or positively "Yes, ¡complete sentence with context from the question¿" .

For "when" type questions, the module pseudo randomly chooses a date or a time and answers the question with that date/time. As the date/time is random, responses may go wildly against common knowledge and common sense.

The "why did you.. " answer template is just makeshift provision to catch and reply to frequent queries questioning the coherence of the chatter bot's responses.

3.6 Graphical User Interface (GUI)

We made the GUI using the tkinter module of python. It is a colourful GUI having an image of the person who is to be assumed as the person you are chatting with, a textbox and an input box. Recent messages are displayed in a visually attractive textbox. The user can input his response in an input box and can send the message using the return(enter) key. The bot replies and the output is displayed on the text box.

4 Timeline

19th May : first meeting

20th - 24th May : All of us did a short course on Data Structures and Algorithms offered by MIT Open Course Ware.

25th - 30th May : We did an online course on Machine Learning offered by Andrew Ng on Coursera.

30th May : First evaluation. We were asked to start working on the code.

31st May : Searched through the net for the best options possible to build a chatbot. Using Prolog, or Octave or Python was on our list of choices.

1st June : Finalized on making the bot using Python and Natural Language Toolkit. Started reading about NLTK.

5th June : A very basic version of Chatbot was built, that only relied on the existing database and exact input matching.

9th June : Second version of Chatbot was made. This made use of Word tokenization, part of speech tagging, and manipulated two SQL tables for its functioning. Wasn't functioning properly.

11th June : We worked out a way to find the best match among inputs already encountered for a new input. Started working on it.

12th June : The above said functionality was added but the bot wasn't going beyond two dialogues with it. Also, we came up with another idea that

might improve performance.

Had a meeting with our mentor, in which we decided to start focussing on the output synthesis part using machine learning. We decided to meet the next day with our findings.

13th June : After hours of searching on the internet, we found that the kind of algorithm that we wanted to implement hasn't been used (apparently). We decided to improve upon the current bot by introducing a few new language processing features, like synonym finding, keyword searching etc.

14th June : Second review. We explained our progress and our plans and started working on the implementation of Wordnet based features.

15th - 17th June : Implemented the Wordnet based matching algorithm. We hadn't reached a conclusion as to how this has to be represented numerically, but the path similairty finding was working properly. We also hadn't figured out the way to deal with multiple nouns in a single sentence.

18th - 20th June : Now, we had a numerical measure of the similarity found by the Wordnet. we came up with another idea to improve our answers - a piece of code that can detect certain types of questions and answer them accordingly, putting the proper parts of speech in the proper places with randomly set dates and places. We started working on it.

21st - 22nd June : We had gone a little far with the above mentioned answering code, and had also developed a pattern matching mechanism, similar to the n-grams technique used by POS tagger of NLTK. This was developed, and a measure to determine similarity of statements was decided upon.

We also stumbled upon the AIML files of ALICE. They looked like a great source for sample inputs for building our database. We wrote a Python script that could extract just the inputs from the files, removing all the tags and the outputs, and prepared ourselves for building the database.

23rd - 24th June : All three algorithms came together and yielded decent results. We decided to start increasing the database. As we did so, we ran into a huge problem - too much time was being taken for replies (around 50 seconds to a minute).

We downloaded the SnakeViz tool and used it to help visualize the results found by python profiler on our code, and found out that database commits

were consuming more than 99% of the time. With a few tweaks in the code, these commits were greatly reduced and the time taken per reply came down to about a second.

We also started working on a GUI and tried developing a website with an SQL table on the server, but failed on the latter attempt. The question answering code was being developed further.

25th June : Developed the GUI and integrated the code with it. Completed the question answering mechanism. Expanded the database manifold. Wrote documentation.

5 Future Plans

1. Make the program more of a synthesizer than a searcher. We will try to make the program make responses by better understanding the meaning of the input and thereby generating responses. Will figure out some way to make it learn to generate outputs with the help of a bigger and more diverse corpora.

2. Make the Wordnet similarity algorithm cover verbs, adjectives and their various forms too, to improve performance.

3. Take our database to an all new level, by increasing it to tens of thousands over a period of time. This may require opening the bot to the public and keeping the database online, so that it can expand ever more rapidly.

Special Thanks to:
Bhuvesh, Aadil, Arunothia and Siddharth
www.cleverbot.org
MIT Opencourseware