Design Lab Report

Human Chat Simulator

Submitted by:
Siddharth Rakesh
11CS30036

Supervisor:
Prof. Partha Bhowmick

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

November 4, 2015

Contents

1 Objective

The objective of this project is to develop an artificial human chat simulator in the English language, capable of carrying out intelligent conversations with human users via textual methods. It has been designed with the objective of engaging the user in small talk with the aim of passing the Turing test by fooling the conversational partner into thinking that the program is a human. Such chat applications also find use in various practical purposes including customer service or information acquisition.

The chat application has been designed to be able learn from the user over time. It can collect data and store it in a structured *JSON* format, as well as record responses to various queries given to it by the user. It can formulate responses based on its collected data and past conversation history. If multiple responses are known to the chat application with respect to some query, it is able to uniformly select a random response, allowing the conversation to be different each time.

2 Approach

The underlying approach for developing the chat application is to formulate the response mechanism as a finite automaton, which helps to significantly reduce the amount of space required to store possible responses to a particular user query. Common parts of two different queries are followed as the same sequence of states in the automaton, and branching happens on the first difference. The application has been developed in Python.

2.1 Pre-processing

Whenever the user gives a query as a sentence, the following actions are performed on it:

- **Tokenization**: The sentence is tokenized into individual words, splitting it on spaces, commas, and other punctuation characters.
- **Common words removal**: Words which occur very commonly, such as articles like *the, is, a, an*, etc., are removed from the sequence of tokenized words.

2.2 Query analysis

Each sequence of words in a query thus obtained, say $w_1, w_2, w_3, .., w_n$, is stored as a sequence of states in the automaton. More specifically, a word w_i corresponds to the state S_i. States are stored in a key-value storage database known as *LevelDB*. Corresponding to each state there exists an *id*, which is generated using the Python *UUID* generator class. Each query state of the automaton contains a map with the following fields:

- **Count**: The number of possible next words in this sequence.
- **Stop keyword**: This keyword, when it exists, indicates where the current state is the end state of the sequence.
- **Next word fields**: The various candidate next words are stored in the state, with an *id* stored corresponding to each of them. The *id*s are generated using the Python *UUID* generator class.
- **Response id**: If the current state is the end state of the query sequence, this field contains the *id* of the first response state in the response sequence corresponding to this query.

In the database, a special state is stored, known as the *start* state. The start state corresponds to the beginning of any user query. Upon tokenizing and preprocessing the user query, the start state is entered. The first word in the query words sequence is looked up in the map corresponding to the *start* state, and the corresponding *id* is recorded.

Using the recorded *id*, the next state is looked up from the database. This new state corresponds to the sequence with the first word the same as the first word of the query sequence. In a similar fashion, the next word in the sequence is looked up in this map, the id is recorded, and the next map is looked up from the database. This process is continued till the end of the query sequence. Once the end is reached, the response *id* is looked up from the corresponding end state.

While looking up words in either the query sequence, if we find that the next word does not exist in the corresponding map, it may be possible that some synonym of the word has been used instead of the word itself. We use an online *API*, *Big Huge Thesaurus*, for looking up the synonyms of the current word. Each of these synonyms is looked up in the current map, and if any synonym is found in the map, we proceed with this synonym in the sequence.

2.3 Response formulation

The response to a particular query is also modelled as a sequence of states. Unlike the query sequence, however, there could be multiple responses to particular query. Thus, a mechanism is required to store multiple responses to a particular query, as well as to be able to select a particular response sequence out of the possible response sequences.

A response state contains a map with the following fields:

- **Count**: The number of possible next words in this sequence.
- **Stop keyword**: This keyword, when it exists, indicates where the current state can possibly be an end state of the sequence.

- **Next word fields**: The various candidate next words are stored in the state. Each candidate next word is stored with the following fields:
  - An *id* corresponding to the next word. The *id*s are generated using the Python *UUID* generator class.
  - A *count* field, which stores the frequency of the candidate next word.

The response *id*, which is recorded at the end state of the query sequence, is used to enter the response sequence. Whenever a particular state is reached in the response state sequence, the candidate next words are considered in the corresponding map. The next word is chosen based on the frequency of its occurrence, which is stored along with the word. Naturally, a word which occurs with higher frequency is more likely to be chosen to as the next word. The probabilities are evaluated using the $Uniform$ probability distribution.

If no further next words are found to exists, or the *stop* keyword is selected earlier in the sequence, the response words collected so far are concatenated as a string, and produced as a response to the user.

2.4 Learning from the user

Up till this point, we have considered the working assuming that some response sequence exists corresponding to the user query. If, however, it is found that no response sequence exists, the chat application prompts the user to specify what he/she would consider as an appropriate response to his/her query. Once such a response is received from the user, both the query and the response are processed as mentioned earlier, and stored as sequences of states, with the query being saved as a query state sequence and the response as a response state sequence.

Apart from very general small talk performed by the user, user queries can be of the format of standard questions, expressing the intent of the user to question the chat application for information. For example, standard queries can be of the format:

- ”Who is Tom Cruise?”
- ”What is the Capital of India?”
- ”When is Christmas?”
- ”Who is the spouse of Brad Pitt?”
- ”Where is Kolkata?”
- ”How old is Narendra Modi?”

For answering queries of these formats, a set of query formats has been manually written. These query format include many well known question formats. A typical question format is written in the form:

What is the \$1 of \$2?

Here, \$1 and \$2 are variable place-holders, whose actual values are replaced with the corresponding words in the user query. A map is maintained corresponding to these variables, storing their values. This map is used to formulate queries to the knowledge database, for extracting the stored data, which is discussed subsequently.

Apart from storing the standard question formats, we also store the standard answer formats to each such question format. The answer format is typically of the form:

The \$1 of \$2 is [\$1 : \$2].

Using the variables map created while analysing the user query, the variables are substituted in the expressions of the form [\$1 : \$2]. This expression represents a query to the knowledge database of the chat application. The knowledge is stored in a *LevelDB* database, as are the states of the finite automaton.

Each $\$i$ in the sequence \$1 : \$2 : \$3 : ... represents the i^{th} key to be searched in the database. More specifically, data is stored in structured formats, known as *JSON* objects. *JSON* objects are essentially structured maps which store values to keys. Each value in itself could be another map, and this structure continues recursively. The key are looked up in the nested maps stored in the *JSON* object, and the value which is stored corresponding to the last variable in the sequence is returned as the required value. This data is then inserted in the position specified by the the answer format.

Apart from the answer format corresponding to each question format, we also store the corresponding inquiry format. In the event that data is not found corresponding to any of the keys in the key sequence while querying the database, we follow the inquiry format to ask the user the answer to his query. This allows the chat application to learn and store data from the user, which can be used in future chats with the same or other users.

An inquiry format is typically of the form:

Can you tell what is the \$1 of \$2?

The variables in the inquiry format are substituted using the values in the variable map created earlier. The user response in then collected following the inquiry.

When analysing the user's response to his/her own question, we parse the response based on the answer format, since his/her response should ideally match the answer format corresponding to the query. The answer portion is then extracted from the response, for example, in the response "Kolkata is in West Bengal", corresponding to the query "Where is Kolkata", the answer format is $\$1 is in [\$1 : location]$. Thus, "West Bengal" matches the position of $[\$1 : location]$ in the answer format, and is determined to be the data to be stored to the database.

From the answer format, we also obtain the sequence of keys to which the data is to be stored, which in this case, are $\$1, location$. Here, \$1 corresponds to "Kolkata". The data is then stored to the database following the keys in a recursive manner. For example, the map corresponding to "Kolkata" is looked up, and if it does not exist, it is created. Next, the map corresponding to "location" is looked up in this map, and if it does not exist, it is created. Finally, the data, "West Bengal", is stored in the map corresponding to "location", thus storing the data to the database.

3 Data collected

For implementing the approach described above, data was collected to account for the commonly occurring questions and small-talk sentences in English.

Currently, the database includes 43 question formats, with answer as well as inquiry formats additionally stored for each of them. Apart from these, 73 commonly occurring English expressions and sentences, such as "How are you?" are also stored, with their corresponding answers.

4 Results

Using the approach described above, the system has been shown to be robust in terms of learning and answering known queries. As the project is still in its infancy, the data yet collected is relatively small. It is still primarily learning from the users via textual interactions, and is in the process of acquiring data as well as responses to general queries.

The implementation for the project is available at: https://github.com/sidrakesh93/Donna.

5 Future work

Improvements which can be made to the existing model include:

- Spelling errors on the part of the user are not yet accounted for. Algorithms for determining nearest possible words can be used for determining the actual word meant by the user.

- In the word sequence, each word is considered as if it were independent. However, words in the English language do exhibit some correlation among themselves, and this correlation can be studied to build even more robust models.
- More query formats can be added to the existing model.

6 References

2012 Bamane, A., Bhoyar, P., Dugar, A., Antony, L., ”Enhanced Chat Application” (GJCST Volume XII Issue XI Version I).

2011 Dean, J., Ghemawat, S., LevelDB: a fast key-value storage library written at Google [Computer Software].

2015 Thesaurus service provided by words.bighugelabs.com.