Remembrall

MEng Project by Jay Tappen, Fall 2018
With Professor Claire Cardie

Introduction

As discussed in the project proposal, the motivation for this project was to create a small application that can remember statements that it is told and can recall those statements when asked about them.

This was designed as a take on the classic problem of question answering, with a simple but severe constraint to make the problem tractable for a semester of solo projecting: the system only has to answer questions it has explicitly been told the answer to.

After building a simple system, I hoped I would be able to either improve the quality of the question answering or turn the system into a speech application so that I could interact with it by voice.

Initial System

Design

The problem constraint meant that every possible question could be answered by a verbatim statement previously told to the system. This means the problem is really a matter of finding the most relevant statement to answer a question. The obvious solution is a simple bag-of-words approach, in which the system returns the statement with the most words in common with the question. This system is very simple to build, can be expanded upon easily, and does not require training data. I don't have training data because most question answering dataset involves answering a question from a passage of text, and most of the answers are short phrases rather than conversational sentences.

Implementation

I built a prototype in Python. It had a simple text interface based on regular expression matching for statements versus questions. The first implementation that occurred to me was to store the statements in a list of sets (each set containing the unique tokens in a statement). Then, to lookup the answer to a question, I would return the statement that had the largest set intersection with the set of tokens in the question. However, this implementation worried me because the recall time scales with the number of statements (the recall time is *O(nk)* if *n* is the number of statements and *k* is the average number of tokens in a statement, best and worst case). Having a low recall time is important for conversationality. Therefore, instead of just storing the set of tokens from a statement, we are going to store an index that maps from tokens to the list of statements they appear in. Remembering a statement will involve adding the statement to a list, which will give it a unique index as an ID, and then adding this ID to a custom mapping. This mapping maps each token to a list of statement IDs for all the statements in which that token appears. For example, the statements ["I like pie", "You like pie"] would create the mapping {"I": [0], "like": [0, 1], "pie": [0, 1], "You": [1]}.

To answer a question in this system, you look up each token from the question in the mapping and record which statement appears in the most tokens' lists. This means the recall time is now *O(pk)*, where *p* is the average number of statements in which each token appears. In the worst case, *p=n* and this is not an improvement, but *p* should grow slower than *n* for common usages of the system.

Final System

Voice Interface

One of my personal goals for this project was to learn how to create custom agents for Google Assistant. These agents are called Actions, and I decided to create an Actions project for my system. This also meant that I could add a voice interface for my system without having to deal with actual voice recognition.

Getting Actions to work took significantly longer than anticipated. There are two primary pieces. The first is a system for distinguishing “intents,” or which function of the application a user is trying to use. For my system, the two primary intents were “remember” and “recall” (for storing and retrieving a memory, respectively). I had intended to do this myself, as in the Python prototype, but Actions is heavily integrated with a system called Dialogflow, which is designed for this purpose, so I used that. It gave me less control, but it is an ML system designed expressly for this purpose, and it honestly would have been a pain to work around it. The second primary piece of the Actions framework is the handler for each intent. These handlers must be made available as webhooks, so I have a handler running as a free-tier “Function” in Firebase. This required the code to be rewritten in Node.js.

The implementation of the logic was functionally the same as the Python prototype, except with a shift to JavaScript data structures. Additionally, I had to make the Actions agent link to each user’s Google account, so that I could save a user’s memories between conversations.

Improvements to Question Answering

Responding in the second person

The first improvement I made to question answering was to convert any first-person statements into second-person before repeating them. This significantly improved the conversationality of the system. This was mostly a simple find and replace of “me” with “you” and “mine” with “yours,” and so on. I also took care with some contractions and some conjugations of the “to be” verb.

Tokenization

In my prototype, I was simply splitting on spaces, but this has the obvious issue that the last word in any question, if it is attached to a question mark, will not match any tokens from any statements (in addition to other punctuation attachments and various downsides). To fix this, I switched to using an npm package called “natural,” which has a tokenizer that pulls out only words. Many tokenizers will include punctuation as tokens, but my system would not benefit from that since statement relevance is much more affected by the words used than by the punctuation used. This has the side effect of separating possessives and contractions, but I believe this is an advantage because it means that names will match in statements and questions even if one of them was possessive.

Stemming

I also tried stemming the words. I thought that this would improve the ability of the system to generalize somewhat. However, I could not find any good stemmers in JavaScript. The “natural” package seems like the primary NLP package for Node, but I tried both of its available English stemmers and they both failed after just a couple simple test words. I decided that raw words were better than poorly stemmed words, so the system does not incorporate stemming.

Breaking Ties

A tie occurs when multiple statements have the same number of tokens in common with a question. Originally, ties defaulted to being broken by whichever statement had been remembered first. I changed this so that the system will return the shortest of the tied statements, the logic being that it is likely to be more relevant if it has the same number of tokens in common with the question while being shorter.

This is important in the following example:

Statement 1: “Alice is 12 years old.”
Statement 2: “Bob is 11.”
Question: “How old is Bob?”

Because the first statement is more verbose, both statements end up have two tokens in common with the question. Therefore, the original system would have defaulted to the first statement. The improved system balances for verbosity and will return the second, shorter statement, which is correct.

Of course, the system will still respond with the wrong statement if the question were “How many years old is Bob?” but it is still a clear improvement.

Shortcomings

The system’s failures largely come down to places where a person asks a question using different words than they originally used to remember something. For example, using “years old” inconsistently in the example above, or using synonyms in place of some words.

The second type of failure results from changes in sentence structure between statements and questions. This is a smaller problem because it affects all statements and questions similarly, so the relative similarity across statements remains mostly accurate.

Possible Remedies

The changes in sentence structure between statements and questions should be fixable with a good stemming approach and better treatment of contractions, tenses, and possessives.

The use of different language to express the same memory in a statement versus a later question is more difficult to fix because it requires semantic reasoning. A simple approach might be making a word’s synonym count for half the word itself, or a more complex approach using cosine similarity of word embeddings might be necessary. The real, but much more complex, solution is a knowledge graph.

Conclusion

Overall, the project went well. As long as the memories are distinct enough and expressed with sufficient verbosity, the system works as intended. I’m excited that I was able to create a voice interface for the system that works on my phone. I am a little disappointed that the system doesn’t work more reliably, but this is mostly due to my underestimation of the difficulty of the problem. The system I built has the level of complexity that I envisioned for this project, but it seems that the constraints I placed on the problem were not enough to make it completely solvable with this simple model.