Natural Language Computer Commands

A study examining the possibility of conversing with our computer

CSE 40657

Jack Klamer

Intro:

The evolution of the computer has always been marked with increases in computer accessibility. Computers have moved from room sized monstrosities to desk-top machines with usual friendly operating systems; rushing in the age of the personal computer. Fast forward some years more, computers more powerful than anything created at the dawn of computing fits into a pocket. With just two buttons, these computers are simple enough that a toddler can use them. The next step in this evolution is voice command.

Goal:

Voice command has come a long way from simple speech to text. AI's like Alexa, Siri, and Google assistant are all inching ever closer to becoming ideal digital personal assistants. Part of that is, in the future, all speech commands and interactions are as natural as possible, and do not require uniform phrasing or a singular speaker. Currently, most hands free systems use a "catch phrase" of sorts to get the computer's attention so that it can listen for queries or commands. While this is simple from a signal processing perspective, this protocol stops natural flow of command sequences that go together to help a user. For some singular tasks, using a computer assistant to complete is faster; however, for sequences of tasks, the overhead of referencing the assistant all over again causes it to work slower than manual button pressing. The latter situation is a much more common way we interact with computers, so if the computer knows how we talk and can recognize when it's being talked to, we have a whole new way to operate a computer. That is the goal for this project: teach the computer to detect when it is being talked to, using a "catch phrase" **or** in a series of commands.

Data:

The data for this project comes none other than the computer nerd fan favorite: Deep Space Nine[1]. The scripts, pulled from html, were cleared of any stage directions, titles, star dates etc. The only thing that was left was the dialogue and the speakers. The data was then prepped using a number of different rules to clean it until there was one line of dialogue with a speaker on each line. The next part was the manual addition of what are called reference tags. A single tag marks the beginning at which point a character is talking to the computer, and the next tag is where referencing ends. These tags are used in both training and testing, to see if the computer can accurately tell when it is between tags.

While this data exemplifies ideal interactions with a computer, it exists in a world where the computer is responsible for many different tasks than current systems. Because this is a show, the computer becomes more of a character than tool, and many interactions are episode specific. Mundane processes are not repeated enough or reliably to train on well, because the writers are more interested in

[1] http://www.chakoteya.net/DS9/episodes.htm

writing good dialogue than good training data. All of which needs to be considered.

Method:

One of the challenges in developing a system is speed. CKY and other parsing algorithms, that would help the computer figure out what the user is saying, are too slow to be asked to run live. This takes away the way the possibility of cross checking what the meaning of the sentence was with all possible computer command meetings. Any system that I built had to be able to detect every single word and guess whether it was being direct at it or not.

I also aimed to improve recall. The idea being that it is more important that the computer know when it's being talked to every time even if it ends of thinking a lot of false instance are directed at it. The idea being that farther down the pipeline, when meaning is accessed, that the false positives let by would be filtered out. However, a balance must be found so that the meaning parsing will not be overloaded with bad instances.

There are no existing approaches in text only to the problem I am trying to solve. While there may exist certain solutions being developed relating to hardware, I do not have access to those resources and will focus on text based only.

The first solution worked on involved keeping a buffer of the last *n* words and then keeping records based off of those. For example, if the last 5 words were "the dog barked at the", then I would train by recording whether the computer was being talked to or not during this point, and I would also record for the last four words ("dog barked at the"), and three, and two, and one. While testing, if the same 5 words would come up again, then I would use 5 times the percentage chance that if was referencing the computer(during those 5 words) plus 4 times the percentage of the last 4 words were referencing the computer, going down all the way down to unigram. The idea here being if the computer has seen the 5 words before then those would be weighted more because the 5 word combinations are more unique. The same process would be done, with percentage of it not referencing the computer and the higher score would win. Ties originally going to guessing positive.

In the addition to the words, I also used meta characters to signify different things that computer hardware would be able to tell the machine. Each time the character talking switched I would pass the new speaker('<ns>') meta character to the model, and it would be placed into the buffer like a word. Another meta character used was the computer talk ('<ct>') meta character. This meta character signified when the computer had a line. While I did not train or test on the computer dialogue, it allowed the computer to maybe tell when it was being responded to.

This is the base point from which I made many tweaks and improvements.

Progress and Experiments

This beginning yielded a recall of .27 and a precision of .02. For every word of dialogue I would guess if it was referencing the computer by reading it into the buffer and using the process previously described. Immediately after inspecting this I determined that it made no sense to validate word by word, because human natural language is understood best at the sentence level.

The first method developed for guessing if a sentence was being said to the computer was if it had just one word in it that was guessed to being directed at the computer. Any sentence that had some part of it tagged in between <c> characters is considered to be directed toward the compute. This yielded a .75 recall and .0113 precision.

This prompted the question of just how many sentences were being guessed as directed toward the computer. Turns out with this system that .3498 of all sentences were being classified as directed to the computer. Which, when only .005 of sentences were actually directed to the computer is too much.

This then prompted the question on how to change the guessing method. The two options were: change how to classify sentences based on the classification of each word or change how each buffer (word) was classified.

From this point forward I decided to use 5 fold validation. I would train on every combination of 4/5th of the data and test on the last 1/5th of the data. I would then average the performance metrics for every run into a cumulative measure. From the original system the recall was .777, the precision .025, and the percentage sentences guessed was .19.

The first experiment was to increase the number of sentences buffered. I increased from 5 to 10, which caused no change. I believe this is because there are no 10 word buffers that show up more than once within the data, and it takes down to 5 in the buffer to get meaningful change, which ends up being incorporated by the recursive nature of the classification metric. I did not explore this farther.

The next experiment dealt with creating a threshold for the number of words in a sentence that had to be guessed as directed toward the computer before the whole sentence would be classified as directed toward the computer. The data for this experiment is shown in Figure 1.

What I learned was that the threshold improved precision, usually at the cost of recall with some exceptions. I began using the F1 and F2 metrics here to determine one number by which to gauge performance. I aired on the side of higher recall using F2 more, noting that a .3 threshold gave me the highest F2 score.

I then discovered a major improvement to how I guessed based on the state I was in. Originally, all tied decision went to classifying as directed toward the computer. I then revered that and got dramatically better results. This is because most sentence instances are not directed toward the computer, and there has to be serious percentage evidence to back it up.

I found that reverting to the original way to classify sentences created the highest F2 score at 0.5635. Even though the .3 threshold classifier had a .745 precision it only had .39 recall. The .2 threshold classifier had slightly better recall and worse precision, but its F2 score was .53. So I moved forward with the original sentence classifier.

Threshold	Recall	Precision	F1	F2	Amount Sentences Guessed
1/sentence length (Original)	.777	.025	-	.1139	.19
.7	.12	.41	.186	.140	.00177
.5	.272	.0970	.143	.199	.0162
.4	.35	.116	.175	.252	.018
.3	.46	.10	.167	.272	.029
.2	.64	.070	.127	.245	.058
.1	.763	.037	.0707	.155	.1317

Figure 1: Data for the first experiment changing Sentence Classification

I then implemented a HMM type system where I buffered word tags and meta characters. The two tags a word could be were either referencing a computer (directed at) or not referencing a computer (not directed at). And based on this tag buffer, and the current word, I would generate a probability that it was being directed at the computer. I then multiplied this probability by the previous decision metric and compared the directed vs not directed metrics again.

This caused sizeable performance improvement. With a tag buffer size of 3 and a threshold sentence classifier of .2 I was able to achieve .82 precision, as the cost of .48 recall. With this new word classifying system with the original sentence classifier still yielded the highest F2 score. I then adjusted the size of the tag buffer which reached its peak performance at 4. This gave me the best results I was able to get.

Recall: .6526

Precision: .6755

F1: .6639

F2: .6571

Ratio of sentences guessed: .00581

Ratio of sentences actual: .00684

This is significantly better than my first system, but has a long way to go before real implementation. It is also important to note when testing I used my own word classifications to fill the tag buffer, not the correct ones from the training data.

I tried another experiment in changing the word classifier to include tag to tag probabilities which caused minimal but existent drop in performance. I then tried adding the probability of classification based on tag buffer and word to the existing classification metric. This caused significant decrease so I changed it back.

Conclusion:

I learned many things during these experiments. Precision often comes at the cost of recall, and visa versa. That a system (the original sentence classifier) that was once worse can become better when the input to it changes. It is import to check all dimensions when creating a system like this.

Also, I was successful in classifying sentences that were not directly stated as “Computer, <insert command>”. During testing all correct classifications print to the screen. A couple such being:

“Display it on screen .”

“In that case , launch a communications probe and instruct it to begin a continuous broadcast of our whereabouts as soon as it clears the atmosphere .”

And while most do start with “Computer ,” I accomplished my goal of generalizing beyond that formula.

All code is available at https://github.com/jklamer/NLP-Project

and all questions can be answered at jklamer@nd.edu .