UNSW SYDNEY

Markov Chains In Chat-Bots COMP4121

Contents

1: Introduction

Imagine a program that is able to converse with you, like an actual human being. This is not that program. But it is a (small) step in that direction!
This report details the processes, details and design choices behind the creation of this program. This program is essentially a chat-bot based off the idea of Markov Chains that attempts to have a conversation with the user. Its speech mannerisms will be based off whoever you choose, which makes it especially entertaining – often it will act in a particular way unique to an individual which is always fun for friends to recognise. And in the case where it misbehaves, it's still entertaining to read nonsensical messages!

2: A (Very) High-Level Overview

The Markov program has access to a large set of conversations. Given an individual user to train the program with, it groups up data from the dataset into several 'pools' of words (with some overlap) from which it is able to form chains with. The program will then ask for input (a message), and it will attempt to mimic a response to that message, based on the individual it has trained itself with. The user is able to continue sending messages until they are no longer amused by the program. A low-level overview is available in section 7 – but there is a lot of information we have to go through before we reach that!

3: Acquiring a Dataset

One requirement was having sufficient data to train the Markov program with (i.e many many coversations). This data was acquired through Facebook's archive system, using my personal Facebook account. With Facebook being an omnipresent force in the average person's life, I was able to obtain massive amounts of raw speech data for everyone I had contacts with! This was done in a completely ethical manner of course – through a convenient service Facebook provides that allows you to download every word you and your associates have written in a single undignified and difficult-to-read bundle. This bundle also contains a disturbing amount of personal information (such as companies with your phone number), but I digress – the bundle I received contained approximately 250,000 genuine messages I had personally seen at some point in my life! That is - every message I have ever sent to a friend, and every message a friend had ever sent to me. This was beyond sufficient for my needs, as any natural mannerisms and patterns in my speech would no doubt be accurately represented within a sample size of this number.

After acquiring this data I had to parse it into a form that the Markov program could read, although that process is relatively uninteresting as it mainly deals with small technical details (i.e get the data into a form the Markov program can read). The process involved using open-source text parsing utilities available online, combined with some regex capture groups. For some unknown reason, Facebook provides its data in a html file, which makes little sense as it would overwhelm and crash any web browser that tries to read it! (The file is very large – it contains every word you've ever spoken, remember?). The Markov program should be able to read in the dataset and identify each individual message – this is the end goal for parsing the dataset. All web formatting had to be stripped away including html code, timestamps and headings. Messages themselves also needed parsing – for example, splitting multiple line sentences into separate

sentences that could be individually identified. The end result was a file containing only the names of people and the messages they have sent, free of the nonsense Facebook likes to throw into its downloadable bundle.

3.1: An Aside – Some Minor Complications

Having started this project somewhat early in the semester, I downloaded and parsed my dataset without complications. However it seems that a few months down the line, Facebook has changed the format of information that is given when you request the bundle, which I discovered when trying to update my dataset with a new semester's worth of data. My regular parsing tools would not work with this new format, and there was little time to re-implement a new parser. The only consequence is being unable to run the program using data from new acquaintances I had met recently, meaning I could not show off my project! Well, not without significant reworking at least.

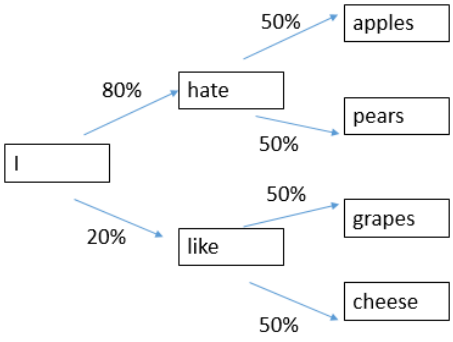
The other bother was Facebook using Facebook ID's (a string of digits) and names interchangeably with no identifiable pattern. The block of data could possibly look like this:

John Doe: Look at this picture of a cat I found
100000000000000@facebook.com: That is a nice cat

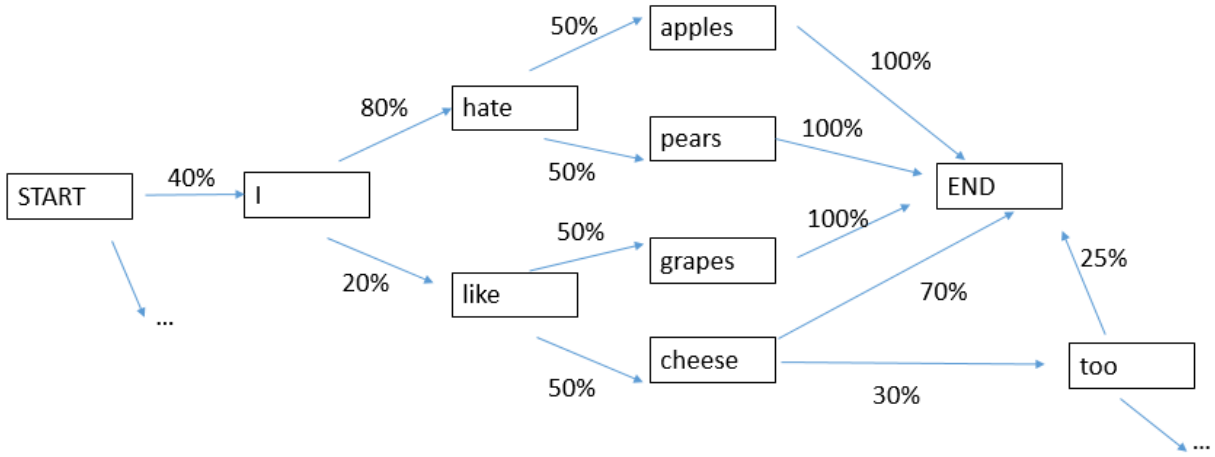
Of course, this was not consistent – John Doe could be referred to by his name or ID at any given moment (from my calculations – the ID was used roughly 40% of the time). Unfortunately there was no clean way of getting around this. The ID of a person can be obtained by visiting their Facebook 'About' section and extracting it from the URL – from that point I hardcoded a mapping between an input name and its ID, using Python Dictionary objects. Simple enough – the bot would use the name as a key to obtain the ID, at which point it would update the regex parsing to interchangeably match both name and ID.

4: A Basic Overview of Markov Chains In Our Context

In simplified terms, the Markov chain we construct for this program consists of states and transition probabilities. Each state represents one word, and with each state we have a set of probabilities that represents the chance which that particular state follows this state. A very basic example of what we want:



So this is how we construct our Markov chains. The program takes in the dataset and creates a graph of words, with transition probabilities connecting various words. To begin sentences, we have a dummy 'START' state that transitions only to words which are the first word in a sentence. Similarly, we end sentences when our word transitions into a dummy state 'END' that indicates the sentence should end. A very basic graph looks like this:



Of course, our graph is much larger than this (it reaches sizes of 35,000 nodes - for more information about the dataset, see section 10 of this report).

So to construct a basic sentence, we begin at the START node. We look at all the chances of transitioning to another state, and we pick one accordingly in a non-deterministic fashion. We repeat this process until we reach the END node, and we terminate the sentence. This is the underlying idea behind the program, although the program uses various other techniques in an attempt to mimic human speech (which will be explained in later sections).

4.1: The Implications of Using Markov Chains to Generate Sentences

The process outlined in the previous section tends to produce sentences that make sense grammatically. After all, humans speaking in English tend to make sentences that follow the rules of English – adjectives follow nouns, and so on. However, this process doesn't always create sentences that convey meaning well.

There is a very important implication in using Markov chains for this program. Markov chains only make predictions based on the current state, completely independent of history. In terms of sentence constructions, the sentences produced are not very likely to stay on-topic. The meaning behind the sentence will only be consistent if the data used to create the Markov Chain consistently keeps a single topic throughout all of its sentences, which is unlikely (but possible).

More importantly though, the goal of this program is to respond to a user's message. How do we do this if we don't ever refer to previous states in our sentence generation?

5: Mimicking a Response, Beyond Simple Sentence Generation

So we've outlined how the program can generate a single sentence. What can we do to mimic a response to a message? First, we should look at our dataset and make a few assumptions for us to base our program off.

Firstly, the dataset comes from Facebook's messaging platform "Messenger", and not a collection of real life conversations. Secondly, these messages all come from myself and people I know, comprised of a rowdy group of young adults between the ages of 20 and 25. This results in a few implications, as follows:

1. People don't greet each other on Messenger (well, not very often). Unlike normal conversation, there is no need for formal greetings, especially with close friends. Similarly, people don't say goodbye – they simply respond to a message normally, and Facebook notifies the other person that the message has been seen. This also can be attributed to the casual nature of instant messaging – people can often initiate a conversation with a random question, and conclude the conversation with an answer (as opposed to say, an email which often involves greetings and thanks). This all means that our resulting chat program is unlikely to respond to greetings well. Instead, ask it a question, or insult it - it is far more likely to respond to those!

2. People spread their messages across multiple messages. A common trend in the dataset is apparent, and that is the fact that messages are rarely sent as one single message. People have a tendency to send each sentence they speak as a message, rather than a single message. For example, the following message structure is very common:
   John Doe: "This cat picture is great"
   John Doe: "I don't know why it's so funny"
   As opposed to:
   John Doe: "This cat picture is great. I don't know why it's so funny"
   In fact, the dataset shows that more than 50% of all messages sent were follow-up messages to the previous message sent by the same person. It is rare to have a conversation between two people that consists alternating single messages per person (for more information, see section 10).
   These so-called "follow-up" messages tend to relate to the first message with certain types of words. For example, conjunctions (e.g *"but that's just my opinion"*). This is not something our program takes into account specifically, but it becomes apparent in practice, as it is very noticeable once you generate sentences designed to be a follow-up. So this means our chat program will need to distinguish between messages sent as a response, and messages sent as a follow-up to the previous message. Both types of message tend to convey different tones.

3. People talk differently. Although this is somewhat obvious, every person talks in a different style. This is also part of the program's initial specification, and the main motivation behind creating such a program (the fact that we can make generate different sentences from different people, and have the resulting messages sound like the person it came from). It is still worth mentioning how our program takes this into account. When consolidating data for our Markov Chains, our program should only use data that directly involves a fixed user. For instance, if I tell the program to train itself on "John Doe", the program should only care about messages John Doe responds to, and messages made by

John Doe. For our purposes, a message that John Doe responds to is a message made by anybody but John Doe, immediately prior to a message sent by John Doe. We can use this definition because our dataset is ordered chronologically.

Point 1 is just an observation, but Points 2 and 3 make it clear that our data needs to be divided up into different pools of data, so that we can take all the above into account. As such, we have our next section, the so-called "Word Pools"

6: The Word Pools

A word pool is how I have been referring to the mapping of words in a Markov Chain. The main idea is to have several 'pools' of words to draw from, that can be mixed and matched (somewhat) to create a flow of sentences. For example, we could divide our messages into two pools – response messages and follow-up messages, as defined in the previous section. A Markov chain could be constructed out of words only from the follow-up pool of words, to simulate a follow-up message.

With reference to section 4 where it was noted that a Markov chain is independent from history – what if we treated entire sentences as states? Would it be possible to map a USER's given input to a response? That is – take a user's input and locate the user's input within a bank of message-to-response-mappings, and randomly choose a state to transition to, where the state is a response to the message. This approach would be possible if our dataset were infinitely large, as any combination of words in the English language would be found as a state in our response bank, but we definitely don't have an infinite dataset. This obviously assumes that we have a word pool that consists of every message a user can post, which is clearly impossible. The next closest thing we can do is match a user's message to an existing message already in our word pool (which is a deeply flawed but passable solution, especially considering how difficult it is to perform sentiment analysis on speech).

Considering all of the above, the program uses the following word pools (which will be explained shortly):

Fix the variable 'John' – the following word pools are constructed relative to 'John'

Word-Pool: A Markov Chain that tracks the transition probabilities of any word spoken by John.

Followup-Word-Pool: A Markov Chain that tracks the transition probabilities of all words spoken by John, within a follow-up message. Especially important because follow-up messages tend to start in a different way than normal messages.

Response-Pool: A Markov Chain that tracks the transition probabilities between the message of X person and the message of John, given that John is replying to X.

Three-Word-Response-Pool: A Markov Chain that tracks the transition probabilities between the last three words of X person and the first three words of John, given that John is replying to X.

Response-To-Word-Pool: A Markov Chain that tracks the transition probabilities between the last word of X person and the first word of John, given that John is replying to X.

Word-To-Followup-Pool: A Markov Chain that tracks the transition probabilities between the last word of John's regular message and the first word of John's follow-up message.

We can form sentences out of each of these pools, and mix and match them where there is overlap. For example, the set of states of the Followup-Word-Pool is a subset of the set of states of the Word-Pool, so we have overlap. This means that we are able to transition from a state within the Followup-Word-Pool into a state within the Word-Pool, without the program crashing.

When we say 'no overlap', we refer to the fact that not all states transition into other states – a state from the Followup-Word-Pool will transition into a state within the Followup-Word-Pool that also exists within the Word-Pool, but that state within the Word-Pool has 0 probability of transitioning into any other state – for our purposes we say there is 'no overlap', as the program is effectively stuck at that point.

We'd like to transition from certain states in one pool to another where there is no overlap. In this case, we have two options. Match the state of one pool to an existing state in the other pool by computing similarities between states, or use a second pool that maps one state in one pool to another state in a different pool. We use both these approaches in our program.

Now essentially, we have our user's messages and John's response as one very large infinite markov chain, which we cannot achieve. Within this set of states, we have a smaller set of states, which is the Response Pool. The Response-To-Word-Pool maps states between Three-Word-Response-Pool and Word-Pool, despite the name. This is an oversight on my part, but it remains this way to mirror my code. Likewise, The Word-To-Followup-Pool maps states between Word-Pool and Followup-Word-Pool. The relationships between our pools of words allow us to generate sentences in response to a user's input, as outlined in the next section.

7: A Low-Level Overview

The program prompts the user for a name.
The program takes the name and constructs the corresponding 'pools' of words and sentences, as outlined in the previous section.
The program now asks the user for a message.
The program takes the message, and extracts the last three words of the message. It computes a similarity score between those last three words, and every state that exists within the three-word-response-pool.

If there is a high enough confidence score in the match, it transitions into a state in the three-word-response-pool that corresponds to three words that start a message. From this point on, it uses words from the word-pool to continue the Markov chain, as there is overlap between the transition states in three-word-response-pool and word-pool.

If the confidence score wasn't high enough for a match, it takes the last word from the user's message, and identifies a state that it can transition into using the Response-to-Word-Pool. This identifies a state in the word-pool to begin the message with, and allows the rest of the sentence to be generated using the word-pool.

If there was no state in the Response-to-Word-Pool that matches, then we begin a new sentence using the start node of Word-Pool.

The above three options finish in a sentence that is using word-pool to construct sentences. Our sentences are created as outlined in section 4, but they also have a 5% chance to arbitrarily

transition to any other state (that of which is also able to transition into other states). This is to prevent any possible cycles that occur within the message, as well as just add a small degree of randomness to the messages to change things up a bit.

When this sentence reaches the end node, then we have a 40% chance to terminate. If we do not terminate, then we send a followup message.
To begin the second sentence, we use Word-To-Followup-Pool to identify a state within the Followup-Word-Pool to begin. If the corresponding state within Followup-Word-Pool has less than 5 possible states to transition to, then we instead begin the sentence with the start node of Followup-Word-Pool. Finally, we transition straight from Followup-Word-Pool to word-pool, as there is overlap.
Additional followup messages are created as required. The chance of termination logarithmically grows, tending towards a 100% chance of termination.

Following termination, the program jumps back to prompting the user for a message.

Let's run though this and provide some justifications as to why it creates sentences like this.

You'll notice that the program doesn't take advantage of the Response-Pool. The initial idea was to match user input directly to an existing message-and-response transition. However, in practice it seems my dataset wasn't large enough for this to work well. This worked well with a few messages, as simple one-line messages like "lol" (an acronym for laugh-out-loud) have many states it can transition to. Sending a message like this will result in the program responding in all sorts of agreements. However, less basic messages like "What did you get for lunch for your birthday?" are only likely to be asked once, and will have one single state to transition to. If any sentence gets matched with this, then it will return the same answer every time, which doesn't look good. Moreover, it becomes increasingly difficult to match sentences, considering the user has infinite many messages and the existing data within response-pool is not that large. Because of this, I cut the response-pool from the program, leaving in place the three-word-response-pool.

A reminder that the three-word-response-pool is essentially a truncated version of response-pool. It tracks what three words the program is able to respond with, given the last three words of a response. The fact that this can produce legible messages relies on two key assumptions.

1. The majority of messages sent by instant messaging are short enough to convey meaning within very few words. This holds true a surprisingly large amount of the time.
2. The last few words of a message are enough to convey meaning about the topic at hand. This is the weakest point of the program, as this is not very likely to hold true at all times.

This results in the program ignoring the majority of the user's input, which is unfortunate to say the least. Regardless, this tends to produce messages that stay on line with the conversation most of the time.

Obviously, the above only applies when there is a good enough match that allows a state (and transition to other states) to be chosen. The fall-back is to directly map the message's final word to a first word in the response, which allows a sentence to be generated beginning from this. We use the response-to-word-pool. This relies on one assumption: the last word of a sentence evokes a certain type of response. Sentences can end on words like "yeah?", which evokes a response of agreement, which begins on a word like "yes".

Finally, when the above fail to create a sentence – we simply generate a brand new sentence. This sentence completely ignores user input, and simply generates a sentence as outlined in section 4. This type of sentence is the least likely to make sense as a response, but the most likely to be humorous. It's especially entertaining to see a response to a question that appears to be very out-of-character.

When we create a follow-up message, we make the same assumption as the response-to-word-pool. Instead we use the word-to-followup-pool, which identifies one of many possible beginnings of a followup message. This transitions straight back into the word-pool, as we have made the assumption that the beginning of a sentence is the most important element in keeping the theme of the rest of the sentence. In practice, this doesn't hold up most of the time, as the response is likely to segway into whatever the program feels like talking about within a few words.
This only applies when there are more than 5 states that the follow-up word can transition to. This is an arbitrary number, chosen due to the size of the pools. The word-to-followup-pool is a more restricted set of states and transitions, and 5 is a number chosen to signify that this word doesn't have enough raw data to justify using. If this word only transitions into 5 separate words, then we might as well not use it, as it would consistently produce similar looking output.

The dataset used in testing is very large, but the more a program tries to mimic a response, the more restricted the input and outcome become. The program presented in this section is a compromise that still allows for mostly-responsive sentences to be generated for a variety of user input.

8: How The Program Is Implemented

The program is implemented in Python 2, and included alongside the report for your convenience.

Our Markov chains are implemented using the Python Dictionary object, which works similar to a hash-map – it stores key-value pairs, where the value is obtained by using a key with the dictionary.

We store our states as strings, which act as keys in the Dictionary. Using a state as a key will fetch a list of strings, all of which are also states. The list represents the transition probabilities – picking one item from the list at random is equivalent to randomly transitioning to any state, at the probabilities given by the transition matrix for that Markov Chain. How does this work? It's because the list is constructed in a way that inefficient in terms of space usage, but easy to use and understand intuitively. Each state S is associated with one list that contains every possible state that S can transition to, in the exact ratio that makes up the transition probabilities of all these states.

As an example, we have the state "*the*". The state "*the*" has a 25% chance of transitioning to "*car*", and a 75% chance of transitioning to "bus". This results in the following data structure:

word_pool["car"] = ["car", "bus", "bus", "bus"]

In this example, the key "*car*" fetches us a list that acts as the transition vector for the state "*car*". We randomly pick one of the 4 items in the list, this is the state we transition to. Clearly there is a 75% chance of transitioning to "*bus*".

Our start node is represented as the string *"__START__",* which is a string that does not appear within the dataset. Similarly, end is represented as *"__END__".*

A walk through our word-pool could look like this:

word_pool["__START__"] = ["i", "you", "you", "but"]

Randomly choose a state – we transition to *"i"*

word_pool["i"] = ["like", "don't"]

Randomly choose a state – we transition to *"like"*

word_pool["like"] = ["dogs", "food", "you", "dogs", "dogs", "dogs"]

Randomly choose a state – we transition to *"dogs"*

word_pool["dogs"] = ["__END__", "__END__", "too", "__END__"]

Randomly choose a state – we transition to *"__END__"*

Our final sentence is *"i like dogs"*

The ratios arise because of the way the dictionary for each pool of words is constructed. Rather than re-calculating the precise ratio of each transition probability every time a state is added, we simply break each message into its word components, then append each word to the appropriate list. This has two benefits – it does not require re-computation with each new state insertion, and it is intuitive to implement. Of course, there is one major downside that has already been mentioned, it is not space efficient at all. A random item is chosen using python's random.choice function, which randomly picks one item from a list using a uniform pseudo random number generator.

If a state has the property where every transition probability is 0, the state does not appear as a key in the appropriate word pool. If a word exists within one of the many lists of transition 'probabilities', then it will exist as a key within the same pool, due to the way the pools are constructed. The exception is of course the start node, which is unique in that nothing transitions to the start node.
This is where they key idea of 'overlap' stems from, and the pools that bridge pools. Though all states within a word pool are 'connected' (taking any state from the list of transition probabilities will give a state which also has its own list of probabilities), there is no guarantee that this state also has its own set of transition probabilities from a different pool. The bridging pools (e.g word-to-followup-pool) identify the transition probabilities between a word that exists within the word pool and a word that exists within the Followup-Word-Pool.

Taking into account everything so far in this report, here is an example of a path we can take through the entire program.

The user inputs *"what to eat for lunch"*

The program isolates the last three words: *"eat for lunch"*

The program checks the three-word-pool for this string – it uses *"eat for lunch"* as a key. It returns a list of transition probabilities. If it didn't return a list, it would have calculated the distance between *"eat for lunch"* and every key in the Dictionary, and use the list of transition probabilities for the closest match. Regardless, we get a list of transition probabilities:

Three_word_pool["eat for lunch"] = ["rice", "mcdonalds"]

We randomly transition to the state "*rice*". This state is guaranteed to exist within word-pool due to overlap, so we continue generating the sentences using the word-pool Markov Chain.

Word_pool["rice"] = ["__END__", "or"]

We randomly transition to the state "*__END__",* terminating the first sentence. Now there is a 60% chance that we continue with a second message. The program decides a second message is a good idea. We use the word-to-followup-pool to determine the first word of our followup. We use the final state (excluding the end node) as our key.

Word_to_followup_pool["rice"] = ["good", "good", "good", "yes"]

"*rice*" was not guaranteed to appear in the word-to-followup-pool, but if it does then it guarantees a list of transition states that appear in Followup-Word-Pool. We randomly transition to the state "*good*", and now we continue making the sentence using the Followup-Word-Pool.

Followup_Word_Pool["good"] = ["choice", "on", "__END__"]

Randomly transition to *"choice"*

Followup_Word_Pool["choice"] = ["__END__"]

We terminate our followup sentence here. Because we are on our second message, our chance of generating another message is now 30%. The program decides not to generate another message, and our final output is as follows:

You: what to eat for lunch
Bot: rice
Bot: good choice

We used 2 word pools to generate these sentences, and 2 additional word pools to map states between the user input and the two pools.

9: Matching Inputs To Responses

There were several options available for matching sentences. A reminder that the initial goal was to simulate the mapping between the infinite set of all messages to a random response – this was 'downgraded' into matching a sequence of three words to an existing set of message-to-response mappings. The way the Markov chains have been implemented means that we have the following goal:

Given a string (that is 1-3 words long), locate an existing key in the response-to-word-pool.

This allows us to find the set of all responses to this string of 1-3 words, and transition to one of these states at random. We have to do this matching process in order to produce a state that can transition to other states. Simply looking for the state matching the input will likely result in a state that fails to transition to any other state.

We can extract a list of all existing keys in our dict. Our main goal is now to find the string in this list that most matches our input string.

A brief foreword – finding a similarity score between two strings is incredibly difficult, and the following will be a very simplistic solution. Firstly, the underlying meaning behind a sentence is very difficult to compute, and two similar looking sentences could mean very different things based on context. There is not much that we can do about this. The second problem is that without a dictionary of synonyms, we'd likely find no similarity between the words "toddler" and "young child", despite these two words being almost identical! However, there is one assumption we can make to mitigate this fact - the astounding fact that only 100 words make up 50% of all written english[1]! Computing similarity of sentences based off words is likely to get us a reasonably close match, using this fact.

The initial approach to this problem was to compute the Levenshtein distance between sentences. This meant finding the minimum amount of single-character changes needed to change one string into another. I won't go too deep into this though, because it soon became impractical. Even when reducing this problem to a dynamic programming problem, the sheer length and quantity of sentences resulted execution blocked by the python interpreter, due to maximum recursion depth being exceeded.

An alternative approach was considered – calculating the Jaccard Index of the two strings. To do this, the program would have divided the sentences into words, and computed the overlap of words, so that the sentence with the most overlap would be chosen. This works quite well for such a small set of strings, and identifies messages that are very close. Unfortunately there were problems with finding a clear and consistent threshold that indicated exactly how closely two strings matched. The Jaccard similarity score only finds a ratio between common words and the union of words in the two sentences, often resulting in multiple sentences with identical scores like 0.5 or 0.75.

Along this line of thinking though, a similar final solution was chosen. I decided that calculating the cosine distance between two strings was the most feasible method for matching the user input to a state. After experimentation, the cosine distance is able to distinguish similar tiny sentences better than the Jaccard similarity score. This is because we don't take into account the direct proportion of matched words to the sentence size (which of course, is at most 3 words long).

First we identify a set of all words, and create two vectors A,B for each word. Each element in the vector corresponds to a word, with value 1 or 0 depending on whether that word exists in that sentence. We now compute

$$\frac{A \cdot B}{|A||B|}$$

In terms of python implementation:
The two words are converted into our 'vectors', using the Counter object. This does not match the vectors A,B in the previous part, but we don't need to implement that exactly to get the same result. In this situation, our vector (which is actually a Counter) A maps a word to a 1 if it exists, with no other mappings. In this sense, there are no words mapped to 0 within A.

To calculate the numerator, we take the intersection of the two vectors, giving us a set. We take each element from the set and multiply the corresponding element in Counter A and Counter B

[1] A fact noted here: https://en.wikipedia.org/wiki/Most_common_words_in_English

– if there is no mapping in either element it returns 0, so we have calculated the dot product of A and B.

To calculate the denominator, we simply multiply the summation of squares of each element for each vector.

The result is a similarity score from 0 to 1, where 1 means the sentences are "identical". The sentences don't have to actually be identical to receive a score of 1, the vectors used in the calculations just have to match. For our purposes, this is acceptable.

In practice it appears that multiple sentences have a score of 1 when compared to our input sentence. We simply choose one at random for our chain to begin at.
Note that we won't have a sentence that's identical word-for-word, because we don't perform a similarity check in that situation (we simply use that sentence as a key for our Markov chain dictionary).

Finally, our threshold for input-key similarity is requires the similarity score to be greater than 0.8. This threshold is chosen somewhat arbitrarily, based on manual review of scores.

10: Fun Statistics About the Dataset

Total number of messages in the dataset: 254,186

Now the following statistics are for myself, and numbers will vary based on the person the pools are constructed for.

Nodes (i.e keys) in word-pool: 35,707
Total states in all transition lists of word-pool (i.e edges): 734,996
This is the most dense graph of them all.

Nodes in response-pool: 35,664
Nodes in Followup-Word-Pool: 49,094
In other words, there are significantly more words residing in the collection of words used to send a second message. This is a direct result of followup messages outnumbering regular response messages:

Number of messages sent by myself: 105,458
Number of messages that were responding to a message: 43,869
Number of messages that were followups: 61,538

And information about each pool:

Word-pool: 35707 nodes, 734996 edges
Response-pool: 35664 nodes, 43869 edges
Followup-Word-Pool: 49094 nodes, 61538 edges
Three-word-response-pool: 34622 nodes, 43869 edges
Response-to-word-pool: 13257 nodes, 43869 edges
Word-to-followup-pool: 9098 nodes, 43869 edges

11: Fun Sample Inputs and Outputs

This section contains conversations I had with the bot that I found entertaining. I told the bot to mimic 4 different people, and I asked it the following questions:

"when is the exam"
"where are you"
"what to eat"
"you're dumb"

With the bot mimicking myself:

me: when is the exam
bot: i'm taking about how do study
me: how are you studying
bot: software engineering design

me: where are you
bot: in bed

me: what to eat
bot: the law of today's weekly

me: you're dumb
bot: rrrrrr angery
bot: omg i'm not quite clever

With the bot mimicking an unnamed friend #1

me: when is the exam
bot: idk
**idk is short for "I don't know"*

me: where are you
bot: train lmao
**lmao is short for "laughing my ass off"*

me: what to eat
bot: not ramen
me: but ramen is good
bot: it's a very studious
me: it's still food
bot: idk
**this person speaks in shorthand quite often*

me: you're dumb
bot: xD
**xD represents a laughing face*

With the bot mimicking an unnamed friend #2

me: when is the exam
bot: [redacted]* said no limit
bot: the tests
bot: and go bad in perl has been more
**with reference to a professor at CSE*

me: where are you
bot: at cse faculty you went from

me: what to eat
bot: i know some meaning with we don't think this is now

me: you're dumb
bot: when
bot: oh no

With the bot mimicking an unnamed friend #3

me: when is the exam
bot: ssomeone nuke it bro
bot: why
bot: then theres already married

me: where are you
bot: i'm on league
bot: i know
bot: shell
bot: wut u workaholic
**the first sentence refers to the user currently playing a game called "league of legends"*
** wut refers to the word "what"*

me: what to eat
bot: >.>
bot: yeah im surprised
** >.> represents a face showing disapproval, despite how strange it looks. Trust me.*

me: you're dumb
bot: dude when

12: Conclusion

The final program is able to take in user input and produces an output that may or may not answer the user's input correctly. It relies very heavily on an extremely large dataset, and the dataset I used was likely not large enough for the solution to produce consistently accurate results.

The program itself definitely produces entertaining results, but the results are not as accurate as I initially hoped when I first started writing the program. Markov chains assume that one can predict the future using information of the current state just as well as if the entire history of the

state was known – this is not very accurate for speech, and it was a challenge to produce a solution that works its way around this. In the end, a machine-learning or neural-network based approach would be better for creating a bot that accurately mimics a person. However, it was an enjoyable challenge seeing how accurate I could make the bot, despite these limitations.