Homework 3 & 4 Report

In order to tackle the problem of making a text generator via a Markov chain, I had to first consider which data structures could be used in conjunction with one another for an efficient implementation of the class. Additionally, I considered whether inheritance from another class might provide additional useful methods (ended up only using enumberable).

I decided to use hash tables as the backbone for the required data structure. Arrays would then be stored as the values of the hash table elements. Each array would contain the possible words to which an n-gram may have pointed in the sample text, with multiple occurrences of a string increasing the likelihood that it would be selected. Further, it made sense to use symbols (a feature of ruby) as the key components of my hash table, as symbols are unique and immutable. I had a fear that strings, if used in as keys in the hash table, may not reliably point to the same value as otherwise identical strings could have different object ids. Additionally, I created a parser class to assist with the conversion of my input file into an array. As an array, the contents of the file could easily be iterated through to construct the Markov chain during the initialization of the class.

Once the Markov chain was constructed, the implementation for the remaining class methods was fairly straightforward. The only methods which required intricate solutions were `generate_sentence` and `generate_text`. Their edge cases had to be hardcoded and I also had to develop a method for allowing random sentence generation. This was achieved by forming another array, one that tracked the words that began each sentence. The `sample` method (for arrays) could then be used not only to randomly select the word that followed an n-gram but to generate a random beginning for my generated sentence.

After having completed my text generator class, I next tackled the creation of a chatbot program. I wanted for it to be able to engage in conversation using proper sentence structure and relevant responses. This would require a large data set (text file) with which to train my Markov chain text generator. The bot would then select a word from the user's response at random and construct a reply incorporating the aforementioned word via the `generate_sentence` method.

This would require the program to find a word to precede the selected word to form a valid trigram with which it could start the iterative sentence construction process. The most efficient solution I thought of for problem was to create a second Markov chain, within the text generator class, to record the two words that preceded every word.

Further by implementing an additional method`, add_to_chain`, in the text generator class, my program would also learn from a user's responses to make new connections in the forward and backward pointing Markov chain structures. This would allow the chatbot to learn new words and their sequences. However, of the words given by the user, the name of the chatbot would not be added to Markov chain structures. The repeated use of its name would skew the probability of

its occurrence in sentences. Further, not only would it be odd for the bot to refer to itself in the third person but its removal rarely affected the grammatical correctness of a sentence.

To prevent stylistic blunders, I chose to use two separate methods for the purposes of this program in order to reduce the magnitude of nested statements in my code, to improve its readability. As ruby lacks the ability to pass a parameter by reference, it made the use of additional methods unattractive.

The final problem I faced with my program was deciding the approach my program would take on letter capitalization (Capital vs. capital) in order to optimize learning and streamline its interactions with the user. By setting the keys (n-grams) of my hash tables as not being case sensitive (only using lower case n-grams) and the values (string within the array) as case-sensitive; I was able to improve the hit rate (program recognizing a word from the users input) while retaining the semantic importance of a capitalized word (Turkey vs. turkey).

The readability and simplicity of Ruby's syntax made this homework considerably easier. The use of mix-ins substantially reduced the amount of code needed and added considerably to its functionality through inheritance. Other features of ruby such as code blocks and easily nested (compounded) methods (i.e. object.sample.to_i.size) only served to improve my working efficiency.

Note: I decided to let the chatbot remain sensitive to punctuation. For example, 'end ' is not the same as 'end.' in the eyes of the program. Further it is not advisable to enter phrases such as 'Chatbot sampleword.' If a word that ends in a period is chosen, the sentence returned will be 1 word long (as sentences end with a period). This was a design choice I decided for my program as sentence grammar and structure would deteriorate without it.