Justin Payan

Poetry-Writing AI Agent

In this semester project I attempted to construct a program that could write poetry. Although past poetry generators have created poems that accounted for structure, syntax, overall theme, and meter, I decided to first worry about semantics and secondly about syntax. I did not intend to factor in meter and structure, as I mostly enjoy reading poetry that has very free meter and structure.

Within the realm of computer-generated poetry there is a lot of variation on the amount of information provided by humans, both the software developer and the user. This variability leads to arguments over what really constitutes artful poetry generated by both humans and machines. When a computer generates poetry that relies too much on humans, people say the computer can not actually take creative ownership of the poetry. For example, RACTER was an early attempt at poetry generation. However, the poems generated were mostly based on templates provided by the programmers, and the user filled in blanks with words. This cannot be considered intelligent behavior; it is just an example of computers doing what we tell them to do.

On the other hand, when humans rely on technology too much, poets begin to complain that the poetry generated did not require any creative input on the part of the human. In both cases, the criticism comes from the poet (whether human or machine) not putting enough of his, her, or its own creative input into the final product. The main example of this is a genre of poetry called flarf. To compose a flarf poem, the poet enters a word or phrase as a query into a search engine. The poet then selects phrases from the search engine's results and combines them into a poem. There is also the idea of spoetry,

which is the genre of poem consisting of subject lines from spam emails. These genres are controversial postmodern genres. Actually, I believe that these genres of human-generated poetry could be good models for a computer algorithm. I will not describe using these models in this paper, but it provides an important idea of using paradigms of human poetry generation to create intelligent computer agents.

Some poetry-writing algorithms attempt to write a new poem from scratch. [Manurang 2003] describes a formulation of the problem as an optimization problem. He uses various implementations of a genetic algorithm to solve the search space optimization. He also discusses using hill-climbing, simulated annealing, and something else approach. He focuses on meter, semantics, and syntax, and he optimizes a multitude of parameters of each. This approach is important to consider as representative of poetry as an optimization problem. [Misztal and Indurkhya 2014] instead break the problem into sub-problems. They use a blackboard system in which there are separate programs to read the background text, generate words (in this case the authors aim to generate words based on the emotional content of the corpus), generate phrases, compose phrases into a poem, and evaluate poems. I will call this approach a segmented approach. [Toivanen et al. 2013] start by generating possible poems, and then use a constraint solver to optimize the output. The constraints are defined by preferred syntax, word relations, and number of lines and words. So it appears that many different basic approaches can be used to write poetry-writing algorithms. I decided that I would take a segmented approach.

My main inspiration came from [Toivanen et al. 2012]'s program that generates words from a corpus and borrows syntax and structure from an existing poem. They then try to revise the poem, changing enough so that the result can be considered a new poem.

With this method, they never have to generate a grammar for poetry. I was going to make a segmented type poetry generator, so I first focused on building word associations.

I chose the Open American National Corpus as my training data. It contains fiction, non-fiction, government, journal articles, letters, travel guides, and spoken word conversations both face-to-face and over the telephone. All of the documents are instances of American English that were generated later than 1990. This corpus was perfect because it contained a wide variety of topics, but it would still allow my program to generate poems in modern American English. Because I downloaded this corpus as a series of about 9,000 separate documents, I had to concatenate the documents from the corpus in a separate program.

I wrote the semantic web-learning program in Java, because I did not have as much time to familiarize myself with Prolog as I originally planned. I can query this semantic web with a word to obtain all associated words. This query is the "topic" of my poem. Of course, I needed to ensure that the web would not take too long to run, because I wanted to maximize the number of documents that the web could "read". This would make its word associations hopefully more similar to an actual English speaker.

I initially implemented this semantic web in a rather simplistic way. I decided to start small, only reading in a single document. This program first segmented the document into a list of paragraphs, and then it segmented each paragraph into a list of words. It created a list of distinct words found in the document, or a kind of base vocabulary. I also had the program remove special characters from the beginning and ends of words, as well as put all letters to lowercase.

At this next step I had to decide what heuristic would determine whether or not a word was related to another word. I had a number of choices of heuristics to use, but due to my lack of experience in natural language processing or computational corpus linguistics, I decided to choose the easiest option. I got the basic idea from [Toivanen et al. 2012], and I initially tweaked it to be slightly easier for me to code. They use the log-likelihood ratio (LLR) test to determine relatedness between two words. To compute this value, one calculates the probability of a certain word co-occurring in a single sentence with another word by multiplying the individually determined frequencies of the two words, and then one also calculates the observed frequency of co-occurrence of the two words. The first method computes the probability of two words co-occurring if the words are independent of each other (p-null). The second method computes the probability of the words if they are dependent on each other (p). One computes the sum of the number of total occurrences * log(p-null/p) for all 4 combinations of word x being present or not and word y being present or not. The final sum is multiplied by -2. If a set of two words have above a certain threshold for the LLR statistic, [Toivanen et al. 2012] draw an edge (with unit length) between the two words on the graph. Later, when they query the graph with a word, they first look at nodes with distance 1 from the queried word. If that does not provide enough words to make a poem, they also view nodes with distance 2. They also suggest using (probabilistic) latent semantic analysis, and latent Dirichlet allocation as more complex parameters for semantic web learning.

I borrowed my basic idea from Toivanen et al., but I utilized co-occurrence within paragraph rather than within sentence. I also started by calling two words related if they simply co-occurred a certain number of times. I could adjust the threshold easily based on

the size of the document/corpus. This heuristic was easier to implement than the LLR heuristic. I figured this would allow me to use a large corpus (with plenty of variation, therefore including lots of words). However, it would not make the computations too complex for the computer to handle.

To count word co-occurrences, my first solution was to make a matrix. I wanted to go through each word in the word list, and for each word make a row of the matrix hold in each element the number of co-occurrences with every other word. This worked for the small document (93 KB of plain text) on which I initially tested my semantic web, but it already started to take a long time when I scaled up to a 401 KB text file. I knew I would eventually have to reject this matrix solution, since the total corpus is 96.8 MB. In addition, most of the thousands of words in the list would never be queried. Therefore it is a waste of time and computer resources to compute data for all of them. Instead, single "rows" of the matrix can just be calculated when needed by the query engine. That is, when a person queries the program with a word, the program searches for paragraphs containing the word and increments the co-occurrence values based on other words in those paragraphs.

At this point, I decided it was time to use the LLR rather than simple co-occurrence. The matrix was not working very well, and I did not expect it to ever work well. I also decided to change how the program determined when words co-occurred or did not co-occur, as I would need frequency of occurrence of the word as well as frequency of lack of the word for both the queried word and every word that was compared to the queried word. For every word, I made a list of what paragraphs that word occurred in. Then I was able to calculate independent probabilities of two words (p-

null) simply by dividing the size of their lists by the total number of paragraphs. I could also calculate dependent probabilities using the intersection and union of two words' lists.

I tried training the program on the entire 96.8 MB. The program ran out of heap memory in the JVM when I first ran it, so I gave it 20 GB of maximum memory to run with. The program took maybe an hour and a half to run, and an error occurred in writing the word list to a file. I noticed that many of the later paragraphs in the corpus took much less time than the initial paragraphs in the corpus. I believe that had to do with some annotations between the paragraphs in the beginning of the corpus that confused my document scanner. The program also took a short time to find what paragraphs contained what words. I think the program essentially had a word list of size 0, but I am still unsure.

After that failure, I decided to try a sub-corpus. I tried just using the letters (about 20 MB), but that sub-corpus was too small and contained too many instances of words like "dear" and "sincerely". I then decided to try just the non-fiction articles, which contain about 73 MB of information. This sub-corpus performed quite well for simple example purposes. I also tried training the web on a sub-corpus of about 1.48 GB that included non-fiction, travel guides, and technical writing. Even when I increased the cutoff on the LLR to over 500, the list of related words for "love" was 3, 078 words long. A random subset of the words returned when I queried "love": leading, s, research, utility, miles, continues, so, coast, power. The web learner thus improves variability when the corpus gets larger, but I still need to whittle the list down to more closely related words.

My next step will be to have the program only put words in the word list that occur a minimum number of times. After that I could implement a limit on the frequency of the word relative to the number of words in the document. In addition to not adding such words to the word list, I would also want to delete the words from the data structure representing the corpus in my program. These rarely used words would essentially never have existed. This would greatly reduce the amount of words in the corpus, which would seriously improve performance time for determining how many paragraphs contain each word.

I think another significant improvement could be made to the program by counting based on sentences rather than paragraphs. Currently the program returns a lot of the same words no matter what the user queries. Words such as “in” and “between” (common prepositions, determiners, conjunctions) come up often in both the non-fiction trained program and the non-fiction, technical article, and travel guide trained program. One way to fix this particular problem would be to explicitly exclude such words from the output. This solution would not be so difficult, because English has a fixed set of determiners, prepositions, and conjunctions. I plan to exclude this list from the output, but also add this list to the base structure of my Prolog grammar so that they can always be used in a poem. Words like “parents” also show up a lot. These words do have a bit more relevance, but I doubt their actual relation to the query word because these words are output for many if not all of the queries. I am not sure what the reason for this repeated output is, but I believe it has something to do with using paragraphs rather than sentences.

Another simple improvement will be to normalize the LLR values, so I can keep the cutoff level constant rather than having to guess what the cutoff should be based on the size of the corpus. I want the values to be doubles between 0 and 1, but I am not currently sure how to normalize them.

It would also be very helpful to train the semantic web on a corpus containing a variety of genres rather than a single genre. Ideally I would use the entire Open American National Corpus, but this computation takes too long and too much memory. Perhaps once I employ the first two fixes listed above I will have much better performance and will be able to utilize corpuses big enough to include a wide array of genres.

There is also another problem in my method for generating semantically related words. Although probabilistic corpus-based models are great for determining actual relations between words, poetry is all about constructing unexpected juxtapositions. If we ask the poetry writer to only use words that often occur together, it may generate some poems, but they will be boring. One possible solution would be to use a different corpus that keeps track of multiple word meanings and metaphorical uses of words. Then I could generate multiple poems with the same word list, but poems that maximize usage of multiple meanings and metaphor would be considered more optimal. For optimization of this nature I could use a genetic algorithm, similar to [Manurang 2003].

I have also made a brief foray into creating a syntactic structure for poems, though I have not spent very much time on this part of the problem. Once you have a list of semantically related words, the first structural or syntactic question is how you can make a program learn the parts of speech of the words in your list. Deep learning has been used to solve this problem, because most of the large databases of text that the

program must learn from are unlabelled. However, once again I went with a simpler structure to implement, at least while I still do not know much about Prolog. That structure is a definite clause grammar.

This grammar allowed me to type in the words given by the program above, along with their parts of speech. Ideally the program should determine parts of speech on its own, but I was not at that stage yet. My word lists coming out of my semantic association program made sense at this time, but they were not incredibly beautiful or meaningful. Therefore I took some liberties. I personally removed some boring words, and I left in more interesting words. I also had to manually define the parts of speech that the words were. I used the categories noun, verb, preposition, adjective, conjunction, pronoun (as an object, subject, or adjective), and determiner (such as a, an, the). It wasn't a very sophisticated grammar, but a useful prototype for the time being.

I currently have to tell the grammar what a poem looks like syntactically, and I then have to sort through all of the different poems that the grammar can possibly generate. When I used "love" as the initial query word (for the non-fiction trained program alone), I first get poems such as:

A love was a love
A love for a love
A love was a love, but a love was a love.

I can then continue to receive poems that have identical first two lines, but end with more creative sentences like "A love was a love, but a love was a year of a play". I also tried generating some simple sentences, such as "A children make a year of a life", or "Love was a year of a life". I generated these sentences by specifying that I wanted the

sentence to end with “a year of a life”. These sentences are not terribly creative (or 100% grammatical), but they are a step in the right direction.

There are two solutions to making my generative grammar more creative. One idea is to simply add more grammar rules. I do not think this solution is the best, because it will mean sifting through more useless material. Rather, I think the solution will be to use a grammar that associates probabilities with words and syntactic structures, so that the most likely output will appear first. That way I can see the poems that have a more interesting mix of words first, and I do not have to progress to view the redundant poems towards the bottom of the list. This kind of grammar is called a probabilistic context-free grammar.

Through doing this project, I realized that poetry writing for AI involves lots of different subcategories of AI and natural language processing. If I hope to make progress on this problem in the future, I will have to learn more about semantic webs, machine learning, and grammar learning. However, the problem is really fascinating, and it opens all kinds of inquiries into the nature of artificial creativity.

Sources:

Gervas, P. 2001. An expert system for the composition of formal spanish poetry. *Journal of Knowledge-Based Systems 14(3–4):181–188.*

Ide, N. and Suderman, K. 2012. *Open American National Corpus.* Available online at: http://www.americannationalcorpus.org/oanc/index.html#.

Manurung, H. 2003. An evolutionary algorithm approach
to poetry generation. Ph.D. Dissertation, University of Edinburgh, Edinburgh, United Kingdom.

Misztal, J.; Indurkhya, B.: Poetry generation system with an emotional personality. In *Proceedings of 5th International Conference on Computational Creativity, ICCC 2014, Ljubljana, Slovenia, June 2014.*

Toivanen, J.; Jarvisalo, M.; and Toivonen, H. 2013. Harnessing constraint programming for poetry composition. In *Proceedings of the International Conference on Computational Creativity*.

Toivanen, J.; Toivonen, H.; Valitutti, A.; and Gross, G. 2012. Corpus-based generation of content and form in poetry. In *International Conference on Computational Creativity*, 175-179.

.