PROJECT PLAN:
Exploring Ambiguity Detection and Generation for Poetry using Tree Adjoining Grammar

Poetry is open-ended – many writings could count as poetry. Yet, for a text, there is some sense of “successfulness” as a poem – by its popularity among academics or critics, or its appeal to you as an individual reader. And, while random strings of letters may appear in (or as) poems, we often look for some familiar structures, like repeated sounds or grammatical sentences or emotionally connected words. On the other hand, we also look for variation within and across poems.

For example, above, lines 1, 2, and 4 repeat an end rhyme (childs, while, files) and a stress pattern (with stresses on capitalized words), but line 3 – maybe pleasantly – breaks this pattern.

In contrast, consider:

Arguably, here, the repeated lines are more monotonic in rhythm, and the last two lines are ungrammatical and seem meaningless; this poem(?) appears altogether less pleasing.

These examples illustrate two motivations to study poetry in computer science: capturing human taste (and thought), and systematizing language. Or, as Manurung cites Binsted: with **Artificial Intelligence**, ever since Deep Blue beat chess masters, “more
artistic-based tasks are often proposed to be the defining benchmark” of intelligence; and with Natural Language Generation, there is a call for flexibility to generate “fluent and natural-sounding texts” or texts “where the communicative goal is vague.”

It is a sizable task to specify a poetry generation process and what (small) parts a computational system should automate. Using linguistic distinctions for language, the parts might be, for example, sound (phonemes, stresses...), word meaning (synonyms, connotations...), or grammar (order of clauses, verb tenses...).

Existing systems show a range of methods and focal points. Some use fill-in-the-blank templates to generate variation in meaning, for example “a ___ laptop was ___ without a trace.” Some use neural networks to induce poems from frequency distributions of words or characters. Some relevant systems do not generate text, but use statistics to analyze and reframe a poem visually, or provide dictionary-like tools for more involved searches like rhymes or stress weights or descriptive words. Manurung describes a system that uses evolutionary algorithms over a rich grammar, to generate well-scoring phrases measured by some evaluation functions. More specifically, Manurung interprets poem writing as a state-space search, where the states are some features of language, like sounds, words, or phrases. Since a poem calls for both structure and variation, a computational generator needs to recognize both incorrect states and desirable future states.

Studying the examples above, we find three broad characteristics we would like in a system: 1) the ability to generate large amounts of varying output, 2) the ability to detect “good quality”, and 3) the possibility to improve quality in specified features, like “be stricter with rhymes” or “add more ambiguity.”

To detect “quality” or to improve quality in specified features, our system should be aware of both feature specifications and quality measurements. For example, if we want to “be stricter with rhymes”, our system needs to know what are rhymes and how to compare their strictness. For such limited feature specifications like rhymes, a simple direction would be template systems. However, this would severely limit possibilities for other feature specifications and the ability to generate large amounts of varying output. Another direction might be neural networks, which might be able to detect “good quality” based on a sizable dataset, and could generate large amounts of output. But to improve quality in specified features, neural networks would also require large modifications in architecture and datasets.

1 Poetry Generator http://www.poemofquotes.com/tools/poetry-generator.php
As Manurung, we decide to begin with the linguistic formalism of Tree Adjoining Grammar (TAG) as a backdrop for studying more detailed features. This formalism allows more expressiveness than Context Free Grammar, and so should promise the chance for large amounts of varying output.

For more detailed features, we will begin by exploring ambiguity: We want to detect and generate sentences that have multiple plausible meanings. For example, “I saw Jack come up with a bag.” could both mean that Jack designed a bag or that Jack carried a bag up a hill. Or “Jack spoiled the lab with his cookeries.” could mean that Jack ruined the lab space or that Jack treated his lab team well. In the first case, our ambiguity could be specified by the distinction of “coheads” and “adjunctions” in TAG, with respect to the phrase “come up.“ In the second case, our ambiguity in the sense of “spoiled” would not be detectable in terms of TAG, but we may look into lexical resources like Wordnet.

For a representation of meaning, we will explore how to consider TAG as a semantic representation. For plausibility and meaning extraction, further extending into ambiguity detection, we will build upon the TAG parser by Jungo Kasai et. al. For a tool for exploration, and as a bridge to sentence generation using TAG, we will consider how to take ambiguous phrases and rephrase them to illustrate the distinct meanings. Finally, for sentence generation, we hope to find some principles to detect phrases that could be extended into something ambiguous, and complete this ambiguous sentence realization.

As time allows, we may consider phonetic constraints, partly relying on tools such as Prosody or Rhymezone.

---