Introduction

For some years I have been disappointed by the lack of a complete, digitized version of Emily Dickinson's poems. Although some fraction of her poems are available, for instance on Project Gutenberg[1] or Wikipedia[2], these sources are explicitly not authoritative, include spurious titles, and are not aligned to either the Johnson, Franklin, or fascicle/set numbering.

There are numerous reasons why a digital variorum edition of Dickinson's poems is highly desirable. In the first place, such a file could be used to discover concordances between poems and periods that are difficult for the human eye to perceive; these relationships, in turn, could be used for more accurate dating and placement of the poems, and would of course support the interpretation of the works themselves. Given Dickinson's popularity among the public, I was surprised to learn that there is as yet no digital edition of either the Franklin or Johnson assemblages. At the same time, databases clearly exist: both the Emily Dickinson Lexicon and the Emily Dickinson Archive yield database records (texts, metadata, and/or images) that are clearly derived from Franklin or Johnson. Unfortunately, contact with the administrator of the EDL and Harvard University Press yielded no leads. Thus, I was forced to create my own.

Text Preparation

Digitization

I began with the 1955 Johnson edition for three reasons:

1. The reading edition had the fewest marginal notes of all editions currently in print, a factor which would simplify the digitization process.
2. The paperback edition was both low-cost and perfect-bound, desirable traits as my digitization process required defacement of the text.
3. I had first been exposed to Dickinson's poems through this edition, and I supposed that this familiarity would aid me in cleaning and constructing the dataset.

What follows is a description of the digitization workflow. Improvements could certainly be made. I tended to value economy of time over preservation of resources

1. A small handsaw was used to cut the spine and glued pages away from the book.
2. A stopped paper cutter was used to align the rough-sawn edges of the pages.
3. The pages were digitized using the top-feed feature of a <model> scanner, about 200 pages (100 sheets) at a time at 400 DPI. Higher resolution would likely aid in the OCR process described below.

[1] See for example https://www.gutenberg.org/ebooks/12242, which collects three series

[2] https://en.wikipedia.org/wiki/List_of_Emily_Dickinson_poems (note that while the poem transcripts are of doubtful accuracy, the large table on the linked page has proven useful for collating numbering between the poems' various organizational regimes).

OCR

OCR was first attempted using Adobe Acrobat; however, the results were unsatisfactory, so recourse was made instead to the ABBYY FineReader software. Specific steps for this portion of work were as follows (all commands refer to FineReader):

1. Open the scanned PDF(s).
2. Enter “Edit Image” mode.
    a. “Crop” margins, headers, and footers from all pages such that only the poem text and poem numbers remain.
    b. Apply “Recommended Preprocessing” to all pages.
3. Leave “Edit Image” mode.
    a. Draw a single “Text Area” around the whole of page 1.
    b. “Save Area Template…” with any name
    c. Select all pages
    d. “Load Area Template…” (check “Apply to All Pages”)
4. Click “Read”
5. Save as plaintext with line breaks.

Corrections

While FineReader produced a substantially more accurate output than Acrobat, there were still numerous transcription, OCR, and artifact errors in the text. While many of these errors were resolved by hand, a number of regular expressions streamlined the process substantially and provide a basis for a more programmatic approach to future digitization:

```
find  \r\n([0-9])
repl  \r\n\t\t\t\t\t\t\t\t\t\t\t\t$1

find  °
repl  0

find   -
repl  -

find   -
repl  -

find  -
repl  -

find  -
repl   -

find  \r\n(\d\d\d)\r\n
repl  \r\n0\1\r\n

find  \r\n(\d\d)\r\n
repl  \r\n00\1\r\n

find  \r\n[a-z]
repl  BY HAND

find  \r\n[^'"a-zA-Z\d\s:]
repl  BY HAND

find  \t[\w]
repl
```

```
find  “
repl  "

find  ”
repl  "

find  ’
repl  '

find   1
repl   I

find  ~
repl  -

find   \|
repl

find   [[DOT]]
repl

find   ■
repl

find   [^aI\W]\r\n
repl

find   ;
repl

find   s
repl  's

find  \*
repl  BY HAND

find  \^
repl  BY HAND

find  ([a-z])([A-Z])
repl  \1\r\n\2
```

Note that these regexes were written for Notepad++ and are not guaranteed to work for, e.g., Python or Perl implementations.

The function `linear_to_list()` is used to convert the OCRed output to a Python list (and, from there, to a CSV and/or Pandas DataFrame). This function accepts a utf-8 text file of the format:

```
OBSERVATION_A
OBSERVATION_N
SOT
TEXT_LINE_A
TEXT_LINE_N
EOT
```

And produces, for each "EOT" in the source file, a Python list of the format:

```
[OBSERVATION_A, OBSERVATION_N, 'TEXT_LINE_A\r\nTEXT_LINE_N']
```

At minimum the source file should include at least one observation for each text (likely a key number: in this case, the Johnson number of the poem), as well as the text itself.

~~The cleaned OCR output included Johnson numbers and the text of the poem (with line breaks intact though often erroneous; see below). Several other data fields were added via a regex (\r\n([0-9]+)\r\n, EOT\r\n\1\r\n0000\r\n0\r\n0\r\nA\r\n0000\r\n0000\r\n0000\r\n0000\r\nSOT):~~

~~0~~	~~Johnson Number~~
~~1~~	~~Franklin Number~~
~~2~~	~~Fascicle (0=None)~~
~~3~~	~~Stanza Structure (0x0, 0=No Stanzas)~~
~~4~~	~~Rhyme Scheme (TBD programmatically)~~
~~5~~	~~Johnson Year Written~~
~~6~~	~~Johnson Year Published~~
~~7~~	~~Franklin Year Written~~
~~8~~	~~Franklin Year Published~~
~~9~~	~~Johnson Editorial Note~~
~~10~~	~~Franklin Editorial Note~~
~~11~~	~~Title~~
~~12~~	~~\SOT~~
~~13:~~	~~\POEM TEXT~~
	~~\EOT~~

~~Where "SOT" means "Start of Text" and "EOT" means "End of Text." This resulted in a final plaintext file consisting of records, each of which was formatted as follows:~~

~~0000~~

~~0000~~

~~0~~

~~0000~~

~~A~~

~~0000~~

~~0000~~

~~0000~~

~~0000~~

~~J_note~~

~~F_note~~

~~SOT~~

~~POEM TEXT~~

~~EOT~~

~~These data fields have been changed in the final version and, in retrospect, constituted unnecessary inclusions: it proved simpler instead to construct a separate list of data fields and join them, programmatically, to the plaintext output.~~

Cleaning

Additional cleaning was undertaken by hand. Common errors included "2" for "?", dashes (hyphens) without spacing (despite the regex above), and "111" for "I'll"; note, however, that these precise transcription errors are likely to vary significantly depending on the source images.

Data Shaping

With the texts successfully and (largely) accurately serialized in a Python list, it was necessary both to create the data structure in which the texts would reside, and to reshape the texts to fit this structure.

Data Structure

A Pandas DataFrame was chosen as the data structure for the texts for several reasons:

- **Extensibility.** Python's built in `dict` type allows easy key:value assignment, but does not preserve order and becomes onerous for data more than 2 levels deep; similarly, Python's built-in `list` type preserves order, but lacks extensive assignment ability. The DataFrame (which operates much like an R data.frame) allows simple assignment of new observations while preserving order.
- **Functionality.** As built-ins, `dict` and `list` feature limited overall functionality. DataFrames, however, have full slicing and subsetting operations, and also support NumPy's fast, vectorized mathematical operations.
- **Portability.** Rather than developing a "bespoke" data structure that would increase the number of program dependencies (and the possibility for error), use of DataFrames ensures that the Glass program will run in a wider variety of environments; DataFrames also features built-in methods for exporting objects to CSV files for use in other applications.

Finally, the similarity between Pandas DataFrames and R data frames should somewhat facilitate migration to R in case more advanced statistical functions are required (or if R's other visualization and data shaping libraries are required).

Stanzas

Line breaks were inconsistently detected in the scanning and digitization process (see Appendix). Therefore, it was necessary to manually review each poem and note down its stanza structure, a task facilitated somewhat by Dickinson’s highly regular style. For the curious, 568 of her poems include no stanzas/a single stanza, 998 of her (multistanza) poems are in quatrains, and the remaining ~200 are in various formats, most frequently sets of triplets and sextets.

Splitting

The smallest formal (not grammatical or lexical) unit of a poem is a line. For the purposes of classification, all lines belong to a stanza, with each stanza containing at least one line and each poem containing at least one stanza. Under this structure, a collection of poems can be split into individual lines, with line number, stanza number, and poem number appended to each. This allows the possibility of direct, vectorized, line-by-line operations (the most frequent) as well as semi-vectorized operations by stanza and by poem via aggregation.

An additional benefit of a data structure wherein each line has its own entry is that it more easily allows for the inclusion of variant readings. E.g., if a certain line is translated from stanza 2 to stanza 1 in Franklin versus Johnson, this change is easily indicated by the addition of new "Franklin Stanza" and "Franklin Line" enumerations. It is then possible to index either according to Johnson, Franklin, or, more generally, any arbitrary manuscript version of a corpus for which numbering is available.

Numbering

For completeness, and following the Johnson numbering and formatting of Dickinson's poems, the following numbering fields were appended to the DataFrame:

• j_num	The Johnson number	(1 – 1775)
• j_lin_abs	The absolute line number in the Johnson scheme	(0 – 19260)
• j_stz_abs	The absolute stanza number in the Johnson scheme	(0 – 4480)
• j_lin_poem	The line number relative to the current poem	(0 – ~40)
• j_stz_poem	The stanza number relative to the current poem	(0 – ~10)
• j_lin_stz	The line number relative to the current stanza	(0 – ~40)

Data Transformations

The bulk of the Glass code and execution time are dedicated to a set of functions developed to extract poetic, phonetic, grammatical, and lexical characteristics from the poems themselves.

Lexical

Some operations are accomplished more accurately with the original text—all words, capitalization, and punctuation intact. Others require or function better with a text consisting only of lowercase words, only of alphanumeric characters, while still others require a text to be split into discrete word tokens. These transformations are easily accomplished with Python's built-in string functions, with support from NLTK for more sophisticated tokenization.

Phonetic

Phonemes are discrete sounds that are combined to form syllables and therefore words. Decomposing a word into its syllables and phonemes is an essential operation for poetic analysis, as a great number of poetic techniques are visible only at the level of phoneme, stress, and syllable. The Carnegie Mellon University Pronouncing Dictionary (cmudict) is an excellent resource for phonetic extraction. Imported via Python's Natural Language Toolkit (NLTK) (Bird), cmudict is a dict with records in the following format:

```
{ 'the' : [['TH', 'UH1'], ['TH', 'IY1']] }
```

Where all variant pronunciations are included as sublists. Because Dickinson uses many archaic words as well as words with non-conventional spellings and morphologies, about 1,500 words out of her 17,000 word vocabulary were not included in cmudict. Luckily CMU also provides the Logios Lexicon Tool, which programmatically determines the likely pronunciation of an unknown word.

While it is possible to use Logios within the Python script, it was mechanically simpler to extract a list of missing words and use the web-based Logios interface to generate likely pronunciations. These pronunciations are then appended to the cmudict at runtime.

Note that this lookup process must be repeated with any new corpora in order for the phoneme-based analysis to succeed: missing words will raise an exception.

Grammatical

While not a focus of this analysis, the grammar and syntax of a text may yield useful insight into its meaning. Tagging each word with its part-of-speech (POS tagging) is the normal first step in this process. This was accomplished using NLTK's Perceptron tagger. Because Python has high function call overhead, the corpus was condensed into a single list of word tokens—capitalized, with punctuation intact. The long list was passed into the tagger and the result re-split and appended to the DataFrame.

Rhyme and Meter

Dickinson's novel approach to rhyme is one of the most immediately noticeable aspects of her poems. While rhyme of any kind poses a challenging problem for automatic detection[3], Dickinson's extensive employment of slant rhyme amplifies these difficulties. For this reason, Glass employs a "Swiss cheese" approach to rhyme detection. Each poem is tested for twelve different types of rhymes:

Function	Description
`perfect()`	Groups words sharing a common final syllable
`augmented()`	Group words wherein a word "base," which ends in a vowel, is augmented by the addition of >= 1 final consonant
`diminished()`	Group words wherein a word "base", which ends in a consonant, is diminished by the subtraction of all final consonants
`unstressed()`	Group words sharing a common final syllable and whose final vowel is unstressed
`eye()`	Group words sharing a common terminal spelling (default: same final 4 characters)
`identical()`	Group words sharing a completely identical spelling
`rich()`	Group words sharing a completely identical pronunciation
`assonant()`	Group words sharing the same final vowel sounds
`consonant()`	Group words sharing the same final consonant sounds
`trailing()`	
`imputed()`	Group words according the rhyme scheme dictated by the form of the poem (e.g., "Common Meter," "Petrarchan Sonnet," "Ballad Meter")

[3] http://ischoolreview.com/iSR_Grav/entries/entry-6 provides a useful overview of an approach to rhyme detection, one I employ here in a novel way. See also Sravana, R., Knight, K. 2011. Unsupervised Discovery of Rhyme Schemes. Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: shortpapers.

Slant rhyme (slant()) is something of a special case: it uses a series of formant-based transformations to detect words that possess similar vowel sounds. In this it uses a "base" "augment" system similar to that deployed in augmented() and diminished(): the final vowel from each word is extracted and compared against all other words; this results in multiple potential rhyme schemes, as each word may be both its own base and an adjacent word for >=1 other scheme.~~For each poem, map ARPAbet phonemes and standard (non-poetic) stress values to each word in each line via the CMU Pronunciation Dictionary.~~

   a. ~~### unidentifiable words were manually tagged.~~

2. ~~For each poem:~~
   a. ~~Instantiate a rhyme array filled with the value "0"~~
   b. ~~Extract the final vowel phonemes from each line~~
   c. ~~For each phoneme, if the phoneme occurs >=1 time in any other line, assign a letter ("A") to the rhyme array for that line and all matching lines~~
   d. ~~that phoneme occurs 0 times in any other line, assign the line *None*.~~
3. ~~Compare rhyme arrays against a selection of Dickinson's known, hand-keyed, "standard" rhyme schemes (e.g., "ABCB")~~

~~Return a list of poems that do not match a standard scheme~~