2 The Measured Words: How Computers Analyze Text

Words should be weighed not measured
fortune cookie

Richard Powers' novel *Galatea 2.2* (1995) is the story of a writer who agrees to teach a computer (an artificial intelligence) to pass an exam by interpreting English literary texts at the same level as a Master of Arts student. Helen (the system the writer develops with a computer scientist) commits suicide (terminates her own program) once she realizes she can never experience the world she is interpreting through literature. As the title of the novel suggests, *Galatea 2.2* is about the relationship of artists and engineers with their creations, which they can come to love into life.

In the original myth, Pygmalion falls in love with his statue, Galatea. Venus, taking pity on him, brings the statue to life. In *Galatea 2.2* it is the writer who lovingly tutors Helen into awareness. Powers depicts the believable possibility of an artificial intelligence that can interpret literature and make art. He asks us whether we would really want a work of *techne* that can do that most human of tasks, interpretation, for us. If it could, would an artificial intelligence want to? Helen's suicide suggests that interpretation without experience or agency is bereft:

> You are the ones who can hear airs. Who can be frightened or encouraged. You can hold things and break them and fix them. I never felt at home here. This is an awful place to be dropped down halfway.[1]

Of course there is no Helen. Computers cannot yet begin to approximate graduate level interpretive skills, but the questions Powers raises stand. What relationships do we want to have with hermeneutica and more generally interpretive machines?[2]

We need to understand what computers can actually do in order to answer questions about how digital analytics can enable new ways of

thinking through. This chapter looks at what questions we can ask of a text with a computer and the tools we build for it. By extension, it asks how a computer understands text. Much of this chapter will deal with the basics of how computers handle text files. Readers familiar with strings and pattern-matching algorithms may wish to move on to the next chapter. For humanists who use computers but don't program, this chapter will introduce data and programming concepts to illuminate the subtle differences between what we think text is and how a computer manipulates it. In the slippage between our literary notion of a text and the computer's literal processing lie the disappointment and the possibility of text analysis. Computers cannot understand a text for us. They can, however, do things that may surprise us.

Numbers and Codes

Is text analysis just quantification? Number crunching is often used as a way of explaining and denigrating what computers do. The philosopher Anthony Kenny, author of *The Computation of Style* (1982), argues in *Computers and the Humanities* (1992) that "in all humanities disciplines the computer is used in an endeavour to replace intuition with quantification."[3] Likewise, Lev Manovich, in *The Language of New Media* (2001), makes quantification one of the basic principles of new media.

Computers are not calculators. They are general-purpose symbol processors that can handle any formally definable symbol system. Numbers or quantities are just one thing they represent symbolically and process logically. All digital information is encoded in a common binary language and then manipulated in logical ways. Programming languages built on logic can do mathematical tasks, check spelling, or paint on a virtual canvas.[4] But computers don't "read" words any more than they "see" paint the way we do. When the letter 'a' is represented on a computer as the decimal number 97 or the binary number 1100001, that doesn't mean that it is a quantity. Instead it is the 98th position in a code lookup table that starts at zero. The 'a' could just as easily have been assigned position 65 instead. Computers are cryptographic machines that manipulate codes, which themselves symbolically represent phenomena (numbers, color, letters, and so on). What matters in cryptography is that everyone have the same code table so that a file encoded on one computer can be decrypted and used on another.

A computer can manipulate many symbolic systems for various purposes by encoding them into a binary format. Image files use color lookup tables that assign a binary code to each possible color; Musical Instrument Digital

Interface (MIDI) uses codes to represent musical events; text systems use character-encoding systems such as American Standard Code for Information Interchange (ASCII) and Unicode. If you have ever seen funny characters or boxes instead of legible characters on a webpage, chances are you are looking at an example of encoding failure. The failure happens because of inconsistencies in how symbol codes are used. The page was created using a different code table. If you would like to try your hand at fixing the problem, you can change the Character Encoding setting in your browser.

Strings: How a Computer Handles Text

Many of the techniques used by computing humanists are quantitative; analytical tools count and graph words. However, when we look at how a computer treats the symbols we think of as text, it appears that there is nothing essentially quantitative about text analysis.

To the computer, a piece of text is a "string"—a sequence of characters from a defined list of possible characters called an "alphabet" or a "character set." If you are learning to program, "strings" are one of the fundamental data types that you learn about, along with integers (whole numbers), booleans (true or false), and arrays (ordered sets of other data types). To understand how a computer handles text, one needs to understand how programs handle strings. They are not exactly what we think of generally as text. They are a formal and limited subset:

- Strings are sequences of characters that have a beginning and end.[5] Each character is either at the beginning of the sequence or follows another character and only one other character. In a text, by contrast, characters don't necessarily follow each other in a discernible sequence. In a crossword puzzle, for example, every character is in two sequences (a horizontal one and a vertical one.) This doesn't mean that we cannot overcome the sequentially of strings, but to do that we typically build on them.
- The characters in strings come from predefined character sets. We cannot introduce a new character, such as some random squiggle, into a string. Instead we have to work with standardized character sets, such as Extended ASCII (which allows 256 characters) or Unicode. The UTF-8 Unicode transformation format[6] allows for more than a million different characters. This pre-defined set, as large as it is, is still finite. That can make representing the rich variation of something like Japanese calligraphy difficult.

- Strings are not formatted with all the rich typographic and layout choices that are available when one is writing a text or doing desktop publishing. In raw text there is no indication of higher-order features such as what is a word, italics, or paragraphs; one has to define a formal way of representing them on top of strings. One way is to have a dedicated character like the typewriter "return" in your alphabet that would start a new paragraph as early text systems had.[7] In ASCII there are a series of non-printing characters like "line feed" and "return" than could be used to control a printer to get the simple formatting effects needed.[8] HTML allows tags like *to indicate that the text between the tags should be emphasized* (though the choice as to what visual feature corresponds to emphasis is left up to the designer).

Most of these differences are really limitations to our broader and polysemous sense of text. Computers are not able to handle the ambiguity found within language without explicit instruction. Strings have to be defined formally and in such a way that it is always clear what the next character is, if any, when processing a file.

More important, computers do not read *meaning* in a string. They process a sequence of characters. Try to imagine that for a computer a string looks something like this:

★❄••❑✶❑❒•❄✁

What could you do with such strings? You might feel a bit lost, or you might want to see if you can decrypt this string. Some characters are repeated, but you don't know anything about that character other than that it differs from others. We have highly developed mental functions that begin reading without any conscious intervention when we see lines of character-like things. This makes it hard to appreciate the limitations of computers. It is difficult to not read text-like signs in your visual field unless you put your eyes out of focus or look at texts in foreign languages. Try looking at

HelloWorld!

and seeing it as a sequence of glyphs without reading it as meaningful text.[9]

Representing Texts

The first act of text analysis is to represent a text as a string so that the computer can manipulate it in ways that can be anticipated. This is an act of demarcation and then one of representation. Which features of the original are essential to the text? For example, encoders have to make decisions about whether the font sizes are important, or whether the pagination is

important, whether marginalia should be included, or which edition to use. All these decisions demarcate what *is* the text and how what you choose to be the text is represented.

Once you have demarcated *what* it is that you want to represent, you need to decide on the encoding you will use on the computer. A string is composed of characters from a limited alphabet. Some textual information may have to be translated or sacrificed to fit the available string representations. Most digital text scholars would recommend using a current standard like UTF-8 for the character encoding and an open markup language based on the TEI Guidelines[10] for the formatting and other information.[11]

The chosen character encoding and file format will make a difference to the sorts of questions you can ask of a digital text. The original seven-bit ASCII character set cannot represent, say, Arabic characters. Typographic formatting such as bold, italic, and 24-point need to be encoded if you want to represent that information in the electronic text. Pagination information likewise needs to be represented if searching by the original print page number is desirable. To add formatting and other non-linguistic information there are a number of file formats and markup languages, some open and some proprietary.

Most people first experimenting with text analysis will not, however, create an electronic text from scratch to experiment with. Rather, they will begin by finding an electronic text online, a text where others have made most of the circumscribing and representing decisions.[12] Webpages are difficult to analyze because they contain all sorts of navigational text, as are Microsoft Word files and other pre-formatted files.[13] Though Word has some useful analytical tools built in, proprietary formats like Word are not recommended for text analysis, as they can be difficult to process by other programs (in other words, analytic functionality is somewhat limited to the built-in features). Voyant Tools is able to process Word files natively, but it is preferable to export from Word to more generic formats (such as plain text or HTML) that can be used with a wider range of tools.

A number of scholarly projects, among them the Text Encoding Initiative (TEI), use open formats based on XML (eXtensible Markup Language). The TEI Guidelines present sensible choices for digital encoding, suggesting the best way to represent a speaker or a new chapter in XML. Because XML-encoded texts use open and documented formats, they are more likely to be usable with future tools than with proprietary formats.

Plain text files, such as those available from Project Gutenberg, are seldom considered scholarly editions; they are minimalist by design. For that reason, scholarly electronic editions are usually encoded in XML (and

sometimes rendered as webpages in HTML). A text editor will reveal the tags that demarcate and represent information. This markup can be used to show the text properly in a browser by adding the formatting information, but it can also be used for analysis. To analyze the discourse of a particular speaker, such as Shakespeare's Romeo, one must first identify how the character and his speeches are demarcated with encoding:

```
<sp who="Romeo"><p>Is the day so young?</p></sp>
```

How does one read encoding? The <p> tag indicates the start of a paragraph of text. The <sp> tags indicate that this paragraph is a speech by someone (indicated by the "who" attribute) called "Romeo." Angle brackets, < and >, are escape characters; the markup language uses them to indicate that what is between angle brackets is formatting and structural information, not content meant to be read. Tags are a layer of encoding built on simple strings; they are interpreted by the machine, but not displayed to the reader.[14]

What Then Can We Do with Strings?

Text-analysis algorithms work with strings in ways that are useful for interpreting the texts they represent. An algorithm is a procedure that involves a series of steps that can be reliably computed. When you describe the steps of a procedure in sufficient formal detail that they could be turned into instructions for a computing machine (e.g., when the instructions are formalized *and* computable or within the capabilities of the computer), then you have a text-analysis algorithm. It is a process that we can use to program a computer to reliably do something to text.

Algorithms are not necessarily quantitative methods. Counting and quantifying can be useful, but they are a means to an end; they are not essential to algorithms. Algorithms automate tasks through formal description of discrete steps. Thinking algorithmically is partly about reducing large and interesting tasks to smaller steps. So what interpretive tasks can we automate? What automated tasks can help with interpretation? If we want to know how a computer can help us question a text, we need to look at the tasks in questioning that can be effectively automated.

In the rest of this chapter, we will walk through the types of algorithms that are useful in textual interpretation. We will describe these intuitively, using what is often called "pseudocode," to help readers understand what the computer is programmed to do as if they were asked to do it manually. We will focus particularly on finding, concording, tokenization, word counts, lists, distributions, and comparisons.

Finding Patterns

Computers match patterns in strings to search text. One example of a simple search algorithm takes a short string, ❁■❁•I▲❊▲ and compares it against every possible substring of a larger text. Imagine comparing the word you want to search for against the first eight characters of the book you are searching in. Continue moving one character forward and repeat the process over and over until you come to the end. With each comparison, check if the search string and the subsection of the full string are the same. (This is a logical operation.) If they are, then you have a match. Google's search algorithm uses pattern matching (though for efficiency it builds an index of terms instead of searching through each document). It searches for short strings—say, the word "analysis"—and finds popular webpages that contain that string. Google doesn't have to know what ❁■❁•I▲❊▲ means; it merely has to find pages with that same pattern.

Computers can do more than just find exact matches for short strings. They can be programmed to look for patterns of characters. For example, software can find the singular "woman" or the plural "women," can find all the variants of "prince" (including "princes," "princess," and "princesses"), and can find word variants such as "sceptic" and "skeptic." The use of "regular expressions"[15] is a common way to enable flexible searching.

Here are some examples of regular expressions:

`wom[ae]n` Search for "woman" or "women."

`prince.*` Uses the metacharacters "." (any character) and "*" (any number of) to indicate that we want strings that start with "prince" and have any number of any characters after.

`(love|hate|whatever)` Looks for either "love" or "hate" or "whatever."

`s[ck]eptic.*` Search for different spellings and endings of "sceptic."

A regex (regular expression) is a small formal specification of a pattern to be matched. Formal languages have evolved for specifying these patterns; the exact syntax and functionality varies between programming languages. Regexes are sufficiently useful that they are built into Perl, or Ruby, and some other programming languages. They do not, however, "understand" text. You cannot simply search all the strings that are *about* love. You could, however, use a thesaurus to create a regex with all the relevant synonyms for love, such as

`(adulation|affection|allegiance|amity|amorousness|…).`[16]

Searching with a thesaurus for all known synonyms can provide volume, but not accuracy. Computers are not good at disambiguating polysemous words that have different meanings (for instance, "like" as verb, preposition, noun, conjunction, adverb, or adjective). You will have to read through the plethora of results carefully.

KWICs and Concording

Pattern matching may locate patterns, but this is not useful to the interpreter unless there is a way to display results. Word processors and browsers move to the first instance of the pattern you want matched and highlight it. Clicking "next" and "previous" buttons lets you navigate the full text, jumping from one hit (individual result) to another. The KWIC (Key Word In Context) display and the Concordance, two of the tools we will look at in depth in chapter 3, are the most common ways of displaying search results.

A KWIC works with our ability to skim. Pattern matching on a large text might throw up a hundreds of results. Having a line of context and the key word highlighted enables you to skim to quickly locate relevant matches. Voyant, for that matter, provides the ability to expand a hit to show more context if you are interested in doing so. You can also use the KWIC to get back to the full text in Voyant. Pattern matching analyzes the string to find

Left	Keyword	Right	Document
) will coordinatethe various elements	text, analysis	, variants, translation alignment)in such	1) 1987-88
this meansword-processing and some	text analysis	programs. My academictraining was	2) 1988-89
...on text documentation, textrepresent	text analysis	and interpretation, and syntax and	3) 1989-90
...recommend HUM-- A CONCORDA	TEXT ANA...	PROGRAM. It isavailable from	4) 1990-91
...develop electronic materials in ortho	text analysis	, databasecompilation and linguistic g...	5) 1991-92
...processing program, SGMLparser/e	text analysis	program, collation program, stemma ...	6) 1992-93
text software. Commercial softwarefor	text analysis	and manipulation covers only a	7) 1993-94
...include: Internet Resources; Electro	TextAnalysis	Tools; Text Corpora in Humanities	8) 1994-95
and to provide software for	text analysis	. Sorry, don't have an address	9) 1995-96
:drain, and coolielabour. June 24-29	Text Analysis	I: TACT (Doorn/Leenarts) The	10) 1996-97
,practical side of producing HTML	textanalysis	and hypertext design. Students would	11) 1997-98
it is entirely probable that	text analysis	skills have been developed outside	12) 1998-99
qualitativeand quantiative software for	text analysis	:Alexa Melina & Cornelia Zuell: A	13) 1999-00
,imaging work(e.g., scanning	text analysis	software). Must be able to	14) 2000-01

Figure 2.1
KWIC in Voyant (from the Humanist Listserv archive).

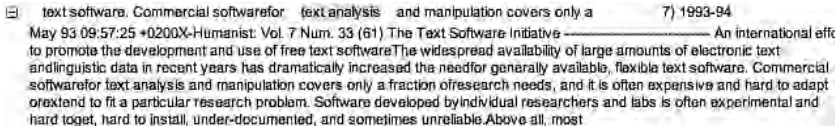

Figure 2.2
A KWIC line in Voyant with more context to preview.

matches; a concordance synthesizes resulting matches for review and further interpretation. They are hermeneutica.

Tokenization

Pattern matching may be good for finding strings in a text, but it will not help to summarize a text. One approach for summarizing a text is to identify the words that best characterize it. Either these words appear frequently within the text or they are the most distinctive words within the text. But before summarizing texts, computers need a way of recognizing individual words.[17] This is where tokenization comes in.

Tokenization is the breaking apart of a text into smaller units that can be manipulated and counted. It is the analysis (etymologically a breaking down into parts) in text analysis, in the sense that a preliminary step of many text-analysis processes is breaking a long string into simpler parts for manipulation and recombination. The generation of a concordance is a synthesis of a new text from processed parts.

Although phrases, sentences, paragraphs, and speeches can be tokenized, it is most often done on individual words. The human brain is so adept at recognizing words in text that one forgets how difficult that same recognition is for a computer. Attempting to define what a word is only complicates the matter.[18] Defining a word as the smallest unit of meaning is not useful. Computers cannot yet understand meaning, per se. Instead, tokenization uses the orthographic word—units that are written like words with definable boundaries such as a space on either side—for the purpose of digital text analysis.

Defining a word as a sequence of characters with a space at either end gets us closer to something a computer could be programmed to recognize. A computer can search for the space character, and every time it encounters a white space (character) it can break off what came before and call it a word. Breaking a string on white space doesn't quite give us orthographic words, however. In some cases punctuation appears before or after a word—for

example, "word." Thus we need to break not only on spaces, but also on punctuation marks—brackets, parentheses periods, commas, question marks, semicolons, colons, and quotation marks. However, without care, the contraction "don't," for instance, would be resolved into "don" and "t")= rather than into "do" and "not." Similarly, there is no absolute way to handle hyphenated words such as "e-text" and "client-side," especially since the orthography can shift over time and with variations in usage.

There are numerous writing patterns that make it difficult to tokenize cleanly. For instance, decimal numbers that look as if they contain sentence-ending periods (e.g., 100.12) and URLs that contain punctuation marks (http://hermeneuti.ca) complicate matters. Nonetheless, tokenization algorithms can be tuned for most languages to work well enough to generate a list of all the words. Insofar as orthographic words correlate with words as units of meaning, we can then begin to try to count meaning.[19] Tokenization takes us from a string (such as "Hello world! I'm now alive to the world again") to a sequence of things we recognize as words:

hello
world
i
am
now
alive
to
the
world
again

Word Counts, Lists, and Distributions

Text-analysis tools that simply sort words by frequency are very useful. The computer processes the list of tokenized words to build a new list of unique word forms. It then counts the number of occurrences for each unique word type. We call this a list of word "types." Each unique type of word has one entry with the count of occurrences. By contrast, a word token is a particular instance of the general type. All the tokens are counted for each type.[20] Table 2.1 shows "Hello world! I'm now alive to the world again" as an alphabetically sorted word-type list.

Word types can be sorted by frequency. Insofar as occurrence indicates importance, frequency counts can provide a sense of what a text is about.

Table 2.1
Alphabetized types with counts.

again	1
alive	1
am	1
hello	1
i	1
now	1
the	1
to	1
world	2

For instance, in our analysis of David Hume's *Dialogues* in chapter 10, we used the frequency of "sceptical" and related words to show that scepticism is a important philosophical issue in the dialogue—an issue that Cleanthes mentions and Philo models. However, function words, such as "the," "a," "of," "to," "in," and "that," appear more frequently than other words in most English texts. Although they serve grammatical purposes, function words do not bear analytical content. For this reason we often use a Stop-word List of the common function words to stop those words from appearing in a Word Type List. Applying a Word Frequency List to Mary Shelley's *Frankenstein* (1818) with stop-words eliminated generates a list like that in figure 2.3. Such a word list is a good place to start a computer-assisted analysis of a text. This list provides a sample of potentially useful search terms to illuminate how "life," for instance, is referenced in *Frankenstein*.

Word Clouds provide a visually stimulating way of looking at a list of high-frequency words. The cloud arranges and sizes words by their rate of appearance. High-frequency words are made large and central in the cloud. A word-cloud tool such as Voyant Cirrus (see figure 2.4) can also rotate and color words so that they to fit more aesthetically into the cloud and make the cloud as a whole easier to explore visually. One gets the impression of a birds-eye view of all the important words. Words appear next to other words serendipitously, which can rightly or wrongly suggest combinations to explore. The word cloud provides a different visual synthesis of the information. It has different affordances for interpretation.

Refreshing the Voyant Cirrus screen regenerates it, providing a different arrangement of the words suggesting other combinations. Randomness can provide interpretive hints, stimulating our imagination, though we should beware of falling in love with these artful arrangements.[21]

Words in the Entire Corpus

Frequencies	Count
man	131
life	114
father	113
shall	107
eyes	104
said	102
time	98
saw	94
night	87

Figure 2.3
A frequency-sorted word list generated by applying a Word Frequency List to *Frankenstein* with stop-words eliminated.

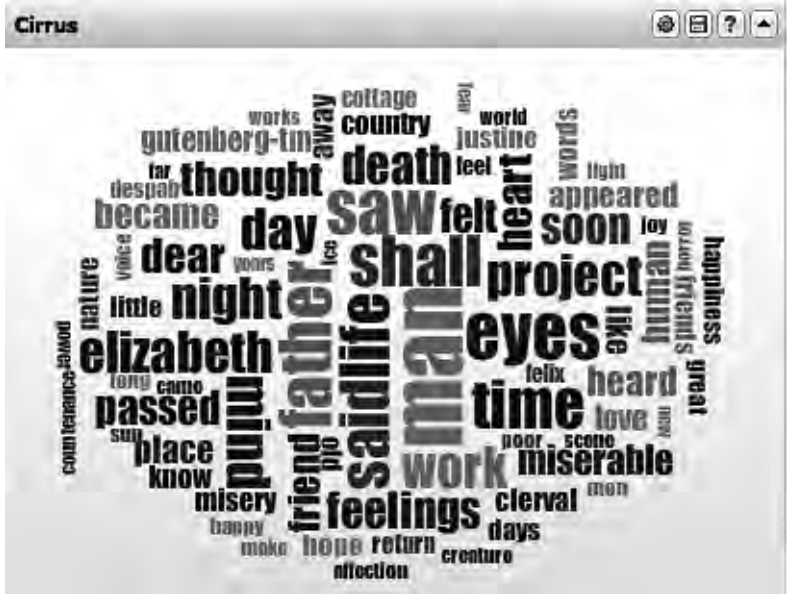

Figure 2.4
A word cloud of *Frankenstein* with stop-words removed.

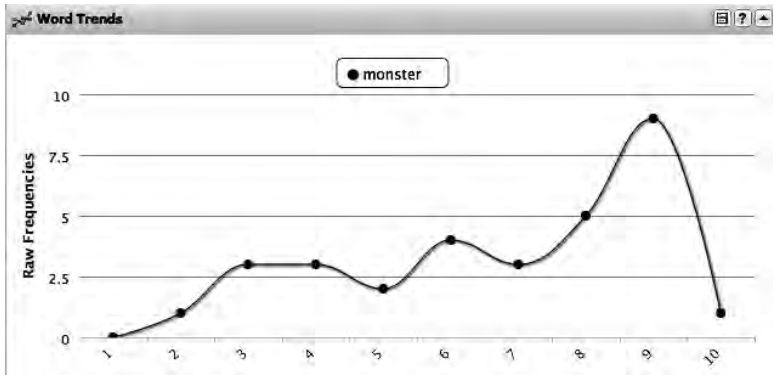

Figure 2.5
Distribution of "monster" over *Frankenstein*.

Distribution Graphs are another common form of visualization. Easily generated through tokenizing and counting, they display locations of occurrences (or hits) in a text or corpus (as we call a collection or body of texts).[22] Dividing a text into parts leads to generating distribution graphs by those parts. (Figure 2.5 shows *Frankenstein* divided into ten equal parts, or "bins," each with the same number of words.) The computer then counts and plots the number of tokens or instances of a word in each part. Connecting the points with a smooth line (a line graph) can illustrate the rise and fall of a theme. However, in cases where search terms appear as discrete references, the line itself might suggest connections where there are none. In this case, a bar graph might be more appropriate, although such graphs tend to be visually more cluttered than line graphs.

Distribution graphs suggest that word frequency is a reliable indication of a theme's significance, which is not necessarily true. Word frequency tells only how many tokens (words) there are in the string or part of a string (text segment). However, frequency can change for many reasons. For example, the frequency of a word that has multiple meanings might increase because another sense of the word is being used that has nothing to do with the theme under consideration. It is a good idea to always check results of text analysis for false positives.

Words in larger strings, each consisting of thousands of words (or tens of pages), might have bursts of usage with long empty stretches in between. To plot the graph shown in figure 2.6, we regenerated the graph shown in figure 2.5 using forty parts rather than ten. This changed the look of the graph greatly.

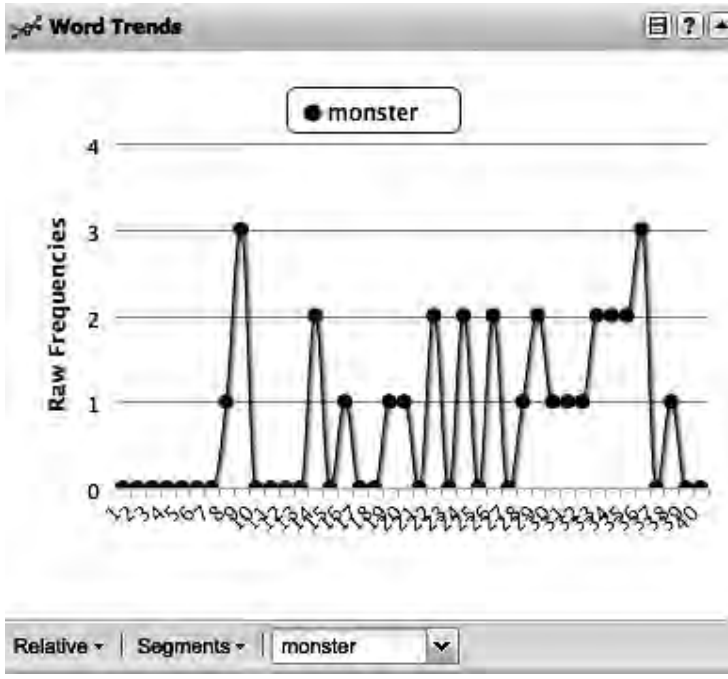

Figure 2.6
Distribution of "monster" over forty segments of *Frankenstein*.

One should also be careful about the assumption of flow in the *x* axis of a distribution graph—that is, the assumption that a text should be read from start to finish, from left to right. Encyclopedias and some other texts are collections of articles arranged in alphabetical order. The distribution of a string over an alphabetized collection would not be particularly indicative of how a theme might appear to a reader hopping from article to article. Reading direction should not be taken for granted, and analysts should be careful of the assumptions tacit in a distribution graph. The string has dimension as a sequence of characters, but that is not the same as reading.

Despite all these qualifications, distribution graphs can still illustrate something about how a theme might move through a text. First it is necessary to find a word (or a group of words) that is (or are) indicative of a theme. The trend line of the pattern can be used to help form hypotheses that can be checked by other means. A pattern that occurs more at the beginning and then slopes down may show an introductory theme; a pattern sloping up to the end might signify a gradual build-up that culminates

in something noteworthy. One theme may fall when another rises; or perhaps themes rise and fall together, suggesting an interesting correlation.

Comparisons

Once you have generated a Word List, you can compare it against Word Lists generated from other texts to better see what makes your text distinctive. How do you get such a comparison? First, you need a control corpus: texts published around the same time period, or other texts by the same author, for instance. The computer then gets the relative frequency of word types for the study text and for the control corpus and calculates the difference between the two. To compute relative frequency, it divides the count for a word type like "mother" by the total number of words in the text being studied, and then does the same for texts in the control corpus. Relative frequency provides the ratio of appearance whatever the size of text. Finally, the computer must compare the relative frequency of words that appear in both texts so that you can see the words that appear more or less often in the study text than in the control text. That provides a list such as the one shown in figure 2.7.

Words in the Entire Corpus

Frequenc...	Count	Difference
was	1,022	5.4
had	686	3.8
towards	94	0.8
feelings	76	0.8
felt	79	0.7
passed	67	0.7
miserable	65	0.6
elizabeth	86	0.6
gutenber...	56	0.6

Figure 2.7
A word list sorted by difference.

The advantage of such a word list is that you don't have to use a Stop-word List. Using a stop-word list can sometimes hide significant uses of words, such as the use of "was" and "had" in *Frankenstein*—words that could indicate something about, say, the tense of the gothic novel. Sometimes an unusual use of a function word can indicate something worth following. It is also worth noting that function words are the most powerful means of identifying the style of an author of a disputed text. Ultimately, word lists, like distribution graphs, do not prove anything about the text. However, they can encourage the formation and exploration of new hypotheses, which one can then double check with a KWIC and, ultimately, check against the full text. Voyant provides "skins" that combine tool panels into an interpretive environment to encourage just such exploration. These skins express one of our fundamental beliefs about text analysis: that it is not about replacing interpretation, but about enhanced reading.[23] Voyant is meant to be ready at hand if you want to think through texts. As such, it is just another machine for interpretation, like the codex or the concordance. It re-presents the text through a mixed skin of panels, like those in a comic book. But these panels are interactive and movable and can be played with.

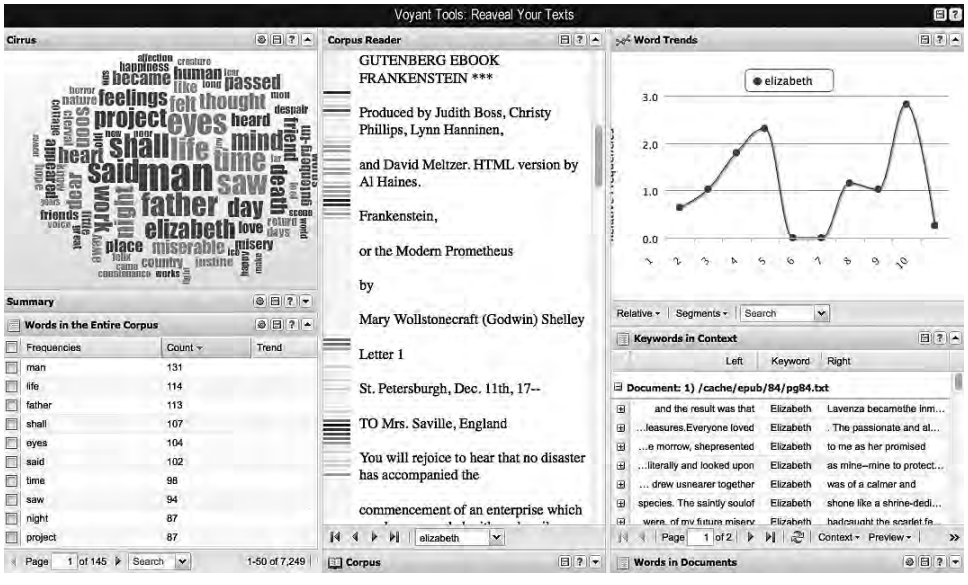

Figure 2.8
A normal Voyant 1.0 skin showing *Frankenstein* with "Elizabeth" selected.

Interpretation and Quantification

Interpretation can be informed by quantification, as we discussed above. A summary of a book is useless if it doesn't distinguish one book from another, and relative frequencies are one way to distinguish works. The assertion that *Frankenstein* is about man's relationship with his technological creations is a useful one only if it is understood that *Frankenstein* is markedly more about technology than other nineteenth-century novels. In talking about such difference we often use semi-quantitative words such as "more" and "less." We might say things like "There is a lot more discussion in *Frankenstein* about technology than in other novels." Whether or not we are right, we are making a claim that can be investigated by using quantitative tools to count words. That is why we should beware of hard distinctions such as that between hermeneutical and quantitative methods.

Computers can do much more than "crunch numbers." They can help us try to formalize claims and to test them. This gives us an alternative to the artificial intelligence approach fictionalized by Richard Powers for using computing in interpretation. Following Tito Orlandi, we can say that computers force us to formalize what we know about texts and what we want to know.[24] We have to formally represent a text—something which may seem easy, but which raises questions. (What is important to the digital edition? How should it be encoded?) Computing also forces us to write programs that formalize forms of analysis and ways of asking questions of a text. Finally, computing forces us to formalize how we want answers to our questions displayed for further reading and exploration.[25] Formalization, not quantification, is the foundation of computer-assisted interpretation.

We use the computer to model a text in both the sense of creating a representation and in the sense of manipulating that representation by creating interpretive tools that allow us to do both. Orlandi, and later Willard McCarty in *Humanities Computing*, proposed modeling as a paradigm for what we do in the Digital Humanities. Modeling is much more true to the practice of using computers to interpret texts than quantification. In text analysis you make models, manipulate them, break them, and then talk about them. Counting things can be part of modeling, but is not an essential model of text analysis. Modeling is also part of the hermeneutical circle; there are formal models in the loop.

As McCarty points out, thinking through modeling and formalization is itself a useful discipline that pushes you to understand your evidence differently, in greater depth, while challenging assumptions. We might learn the most when the computer model fails to answer our questions.

The act of modeling becomes a path disciplined by formalization, which frustrates notions of textual knowledge. When you fail at formalizing a claim, or when your model fails to answer questions, you learn something about what is demonstrably and quantifiably there. Formalizing enables interrogation. Others can engage with and interrogate your insights. Much humanities prose supports claims with quotations, providing an argument by association or with general statements about what is in the text—vagaries that cannot be tested by others except with more assertions and quotations. Formalization and modeling, by contrast, can be exposed openly in ways that provide new affordances for interaction between interpretations.

We are not claiming that all arguments need to be formalized. Rather, we are suggesting that formalizing processes can help in modeling our understanding of a text and exploring it in ways that can produce insights and interpretations that don't necessarily have to be formalized. With Voyant and similar tools, you can share hermeneutical panels (hermeneutica) that package text, tool, and parameters in such a way that a sceptical reader can play with your model. Such embeddable analytic toys don't try to replace the human interpreter with an artificial interpreter, such as Galatea or Helen, but rather give you ways of thinking through text.

Conclusion

John Searle (1980) introduced the Chinese Room thought experiment as a way of showing how machines do not think the way we do, even if they can mimic thinking behavior. That thought experiment has us imagine Searle, who doesn't know Chinese, confined in a room. Elaborate instructions in English for responding to sequences of Chinese characters are slipped under the door. He gets a slip with some instructions in Chinese on it, slowly follows the instructions (the program), and writes output on another slip, which he then slips back under the door. From the outside it appears that there is something in the room that understands the Chinese well enough to interpret the texts slipped in, and yet Searle doesn't really understand Chinese at all: he is just following a program of instructions. Searle uses this thought experiment to critique "Strong AI" claims about computers' ability to think and understand as humans do. The experiment helps us to imagine how computers can do things that appear to be based on understanding but are not understanding as we experience it. We can see how Searle, hidden in a room with good instructions, could appear literate in Chinese while not understanding a word of it.

Powers, in *Galatea 2.2,* takes the story further and imagines what it would be like if an interpretive intelligence such as Helen recognized that she was trapped in a room with only the art of others as input. To be able to interpret the literary texts assigned, she needs to imagine the real world experienced through them, and that includes trying to understand relationships of love such as that of a creator and a trainer and that of one created or trained. How claustrophobic it would be to recognize you have no body with which to interpret! Hubert Dreyfus (1979) argues that cognition is embodied. If he is right, it is a wonder that Helen can respond as well as she does, and it is no wonder she terminates her disembodied life.

Searle's imaginative experiment, like Powers' novel, is relevant to understanding computer-assisted text analysis in a number of ways. It is a mirror image of artificial intelligence—human practices that artificially mimic computing. Imagining ourselves as the processor inside Searle's translating machine, we can see how rote processing is not understanding. Searle's point is not that computers may never behave intelligently, but that they cannot process information as embodied minds do. This chapter has tried to likewise carefully disembody that difference so that readers can better appreciate how stupid or surprising a computer can be. Though not intended for that purpose, Searle's thought experiment also shows what computers are good for and why we would use them. They are good at following long, tedious, and complex instructions carefully; they can manipulate symbols without over-interpreting them. The instructions in English can change how Searle acts, but the Chinese script cannot. That is an important difference between computers and us. We can be influenced by what we analyze as we are analyzing. Current computers aren't as plastic as we are.

If a computer could be influenced, like Helen in *Galatea 2.2*, we would have to worry about what its agenda might be and how it might change. The computer would no longer be an innocuous aid for interpretation; it would become another problem of interpretation. We would still want to reliably implement useful algorithms on text strings. We would still want tools with which to break a text into units that can be searched, counted, graphed, and recombined for further interpretation; tools to troll through millions of books matching patterns; tools to compare one text against others; tools that counted things. Should artificial intelligence arise, we will still invent (or re-invent) tools that allow us to use *our own* intelligence. We will still develop interpretive tools—hermeneutica—that can augment and extend our reading, not replace us.