# Chapter 17. The Pleasurable Difficulty of Programming

Benjamin Miller
University of Pittsburgh

**Land Acknowledgment.** I write from the city of Pittsburgh, in the ancestral territories of the Osage Nation and the Shawnee Tribe; much of my climb up the learning curve I will describe below took place in New York, home of the Wappinger and Munsee Lenape peoples ("NativeLand.Ca"). As I have striven in my research to situate graduate education in composition/rhetoric amid a flux of topics, methods, and mentors, so too do I situate that disciplinary flux itself within the long history of occupation, exclusion, preservation, and celebration of these peoples and their traditions. I honor and thank those whose sacrifices made possible my access to the resources I use every day, as well as those who continue to educate me about these histories and their ongoing effects.

## Why Programming? Why Now?

In *Coding Literacy*, Annette Vee traces the ways that computer programming has suffused modern life, such that even people who don't program themselves still need a "computational mentality": the ability to anticipate and respond to the ways computer programs shape our lives and interactions (196-97). Alongside everyday software applications like email and online shopping are a growing number of tools for academic work, from library search portals to multimedia composing platforms to suites for data analytics and visualization. In most cases, the tools are available to non-programmers and programmers alike, because the software provides user-friendly graphical interfaces: programming, that is, that seems to obviate the need to look too deeply into the programming. "At first glance," Vee writes, "thousands of apps, menus, and interfaces promise to deliver the power of programming to those who do not know how to write code" (22). Yet she cautions that the ability to read and write code, with its requisite habit of thinking "in hyper-explicit terms" (ibid), is no less important now. Increasingly, she writes, "to navigate many professions and the demands of life in the twenty-first century, we need to have computational skills, or at least know someone who does" (197).

The history of computers and writing (C&W) offers plenty of examples of "writing teachers writing software" (to borrow the title of Paul LeBlanc's 1993 book). Even by 1984, efforts in process-oriented computer assisted instruction could fill out a thirteen-chapter collection (Wresch), and a much larger bibliography by the time of Mike Palmquist's 2003 review. Since then, large-scale peer review platforms

DOI: https://doi.org/10.37514/PRA-B.2022.1664.2.17 

like Eli Review (Hart-Davidson et al.) and MyReviewers/USF Writes (Melançon), along with text-markup tools like <emma> (Desmet et al.) and Docuscope (Kaufer et al.), both support writing pedagogy and generate datasets for further analysis. More bespoke data visualization efforts in the field, such as those in a recent special issue of *Kairos*, use programming frameworks like d3.js (Lindgren and Ridolfo), amcharts (Turner and Gonzales), and R (Dighton), among others.

All the same, as Tim Lockridge recently noted, while "this type of work [i.e., building a digital tool to solve a problem] was once the norm in computers and writing [. . . it] seems less so today" ("The Problem"). There are several good reasons for this, as the rest of the present collection makes clear. First, many of the questions and problems facing C&W are better addressed by interpersonal means, rather than algorithmic ones; indeed, some problems are even *caused* by algorithms, which tend to embed cultural assumptions as biases or blindspots (Noble; Klein and D'Ignazio). If you're analyzing the uses of digital multimedia by musicians (Craig), or the impact of telepresence on a writing center conference (Feibush), then individual, embodied human perspectives are essential. Second, many of the digital tools by which we can preserve or present, say, interviews, already provide a great deal of flexibility and power to their users, even without having to touch a line of code. There's no need to reinvent the reel to take advantage of digital video editing software, for example, on top of which, the level of programming skill required to make such software lends itself to dedicated specialists: engineers, rather than writing scholars. Even questions and problems for which an algorithmic approach makes sense, such as statistical analyses of large bodies of text, are increasingly addressable without having to produce the code that powers the algorithms. Web-based corpus analytics suites such as Voyant Tools, or built-in search-and-filter functions in scholarly databases like COCA (the Corpus of Contemporary American English), have lowered the barriers to these kinds of computational research methods in ways that are surely worth celebrating.

All the same, I find myself drawn to the open-endedness of programming. As a discourse, it has much in common with writing more generally: rather than a proscribed set of options to select from in a menu, programming languages offer the materials by which to shape new approaches that fit the data and questions we bring to it. In that sense, codework is rhetorical, addressing the present situation by drawing flexibly on insights from the past. Like other forms of writing, writing with code can be both frustrating and tremendously rewarding, sometimes even in the same working session. And as with writing, the process of working through those frustrations is itself epistemically generative, forming a feedback loop that can shift one's sense of what's important and how the pieces fit together. So, despite the steep initial learning curve—and, yes, the ongoing challenges of maintaining the ever-expanding set of files and executable scripts that form my research codebase—I have continued to return to programming as a way of centering my attention on a research project, getting a handle on my data, and refining my understanding of what it must teach.

In this chapter, I write to explain the draw (as well as some drawbacks) of this kind of digital composing as a research method, and to demystify the process for those curious about but unfamiliar with code. Toward that end, the heart of the chapter is an extended example, or re-enactment, of a recent challenge in my research, and the programming workflow I used to solve it. Inspired in part by Dana Kantrowitz's "The Making of a Poem, Live and Uncensored" in *The Subject is Writing*, I trace a series of aims, misses, and rescues, presenting not only a reconstruction of what I was thinking, but also (some of) the code produced along the way. In doing so, I will highlight key affordances of functional programming that make it not only useful from the perspective of knowledge production, but also affectively rewarding.

## A Note About the Code in this Chapter

The code I share is not "live and uncensored"; it's only a small part of a much larger codebase, curated retrospectively. Even so, I realize it's still a lot of technical language and syntax to throw at you—and that, for some readers, *any* amount of code will feel alien, or alienating. What's more, my example comes from a statistically oriented programming language, R, which may not be the first language you want to try, even if you are convinced of the value of programming for writing research. Nevertheless, I believe it is important to show the code itself, for several reasons.

First, I want to make the sight of code *less* alien. If we allow graphic user interfaces (GUIs) to hide all the conditional statements and assumptions that the code makes explicit, we cede the ability to intercede in those operational decisions. Even if you have no interest in programming yourself, developing what Vee calls a "coding literacy" will help you communicate with those who do, with a better sense of what the code makes easy or hard. Increased circulation of coding literacy has ethical implications: "If we want a more inclusive and equal society," Vee argues, "the writing of code should not be left to a handful of elite or isolated groups" (224). Diversifying and expanding the group of people able to *read* code is an important first step toward that more inclusive society.

Second, setting several examples alongside each other can help make visible recurring patterns in the code, especially those that cross programming languages. One or two examples could show you what programming looks like in the abstract, but to really get a sense of how programming can work in the context of writing research and problem-solving, I need to show you more than that. One key concept I want to highlight is the high frequency of *control flows*: functions with inputs and outputs, iterative loops that run a series of inputs through the same chunk of code, if/then/else statements that divert the flow from one code chunk to another. Second, I also want to call attention to the recursive nature of programming: the ways *new code integrates and recontextualizes old code*, sometimes necessitating changes to the old code (a.k.a. refactoring) to address false or incomplete assumptions made apparent by the new use-case. A third key concept of programming as method

is *task decomposition*, which entails breaking large objectives into smaller pieces. Seeing the iterative, interactive process of identifying, constructing, and combining those smaller pieces is essential to understanding what's involved.

In addition, I want to step through an extended sequence from problem to program in order to recognize that plans for coding, like plans for writing, must often shift as they are implemented. Learning to code can be difficult, and setbacks are assuredly frustrating. But they are also a persistent feature of composing, and I don't want to paper over that fact, or leave it for readers to discover and become discouraged by. I come from a privileged background, in which I was encouraged to believe that I could do anything if I put my mind to it, in which the language at my well-funded public schools never felt foreign (both my parents went to college, and my father has a Ph.D.), and thus early successes seemed in easy reach. I only took one computer science course, and not until college, but it was a lecture-and-recitation at Harvard that assumed most of us would explore documentation and find most examples on our own. This again reinforced the message that success was a given—but also implied that it was a matter of individual persistence. Even with my already-internalized sense that challenges are only temporary, only puzzles to play with, I found programming difficult; I can see how, without that sense of arbitrary self-efficacy, it would be easy to respond to such difficulty by saying, "programming just isn't for me."

So, it's important to me to counteract the idea that good programmers just do it right, the first time, on their own. On the contrary, when I really began learning how to program for analysis while writing my dissertation, I had the advantage of peer mentors from across the digital humanities whom I could turn to for example code and turn back to with questions. I had coursework and a fellowship in Interactive Technology and Pedagogy that helped me build that network of peer mentors. Because I recognize that not everyone reading this will have the same local support networks, or the same lifelong drumbeat of positive reinforcement, I hope the code I share here—including the code I wrote that failed, and the kinds of steps I took to try and fix it—will offer at least the starting place my friends[1] and *their* code were able to offer me, both as a graduate student at CUNY and as early faculty at University of Pittsburgh. At the end of this chapter, I will point to additional open resources for those interested in taking up programming as a digital research method.

## My Research Program

The context for the work I'm discussing in this chapter is a book-length study of doctoral dissertations in rhetoric, composition, and writing studies—several thousand of them, submitted over a fifteen year span—as a way of advancing what Derek Mueller calls a *network sense* of the field: "incomplete but nevertheless vital glimpses of an interconnected disciplinary domain focused on relationships that define

1. Thanks to Micki Kaufman, Evan Misshula, Matt Lavin, and Scott Weingart.

and cohere widespread scholarly activity" (3). Dissertations are well-suited to such questions of disciplinarity: they are widely distributed, sustained, and required to remain recognizable as work "within" the field, even as they advance new claims.

My book is primarily a descriptive study, aiming to intervene in scholarly debates about what the field *should* be doing by stepping back to first consider what we *have* been doing. One way to understand the goal of describing the field's practices (without predicting future behaviors or pre/proscribing what people ought to do) is that I'm engaged in a mapmaking project: I'm looking to chart the disciplinary landscape, to identify the existing balance of subject areas and methods, and thereby help newcomers navigate the potentially overwhelming range of possibilities.[2]

Digital tools offer two key advantages in this pursuit. First, they can read a lot faster than I can; and second, they make analysis more replicable. As Mueller points out, the size of our disciplinary domain has grown beyond what even a diligent reader could attend to, even reading all day, every day. Computers, though, can abstract data into metadata, consolidating great quantities of information into tables and graphs, which in turn amplify signals that human readers can interpret. Susan Lang and Craig Baehr clarify that such computer-assisted analysis doesn't change the responsibility or focus of human interpretation so much as the scale of what's being considered. Even so, "data and text mining extend [traditional humanities research activities] beyond what it is possible for us to do as individuals without the assistance of computer technology, as large amounts of numeric or textual data can be examined for various types of relationships, including classes, clusters, associations, and patterns" (Lang and Baehr 178). One of the traditional activities they specifically call out is reflection, reminding us that these patterns can't be taken as neutral or inevitable. Still, by externalizing part of the process of discovery into code, computers make it easier for subsequent researchers—or even ourselves—to repeat the process in a new context, to thoughtfully examine what changes, what's expected but missing, what other explanations might underlie the patterns we see.

## Programming My Research

As I suggested above, several software tools—including free tools—now exist to make it easier to classify, cluster, or otherwise detect associations and patterns in textual data. Laurence Anthony's concordance software, AntConc, can identify words that stand out more in one group of documents than another; it can also

2. I want to be clear, though, that I'm not trying to make a once-and-only map of the field. For one thing, maps always, of necessity, leave things out: a map that includes everything is just the territory itself, simultaneously perfect and pointless. I am, rather, trying to capture one set of phenomena; different methods, and different vantages, offer different senses of the network, and each will be useful in their own ways. For another thing, fields do and should change over time. But if we are interested in how things change over time, we need to take stock periodically, to establish points of comparison.

show keywords or phrases of interest surrounded by the preceding and following parts of each sentence where they occur. Voyant-Tools, a full-service text-analytics suite that runs in a web browser, offers these and many other operations besides: it can show the frequency of words rising and falling across the documents in a textual corpus; it can visualize the corpus as a word cloud, or as a set of two-word phrases that appear frequently together (bigram collocates), and much, much more. That these and similar tools are available, and free to use, marks an incredible advance in access to digital research methods, and I highly recommend them, especially in the early stages of becoming familiar with a body of texts.

At the same time, there are limits to what they provide. In particular, the outputs are essentially endpoints: the tool produces a graph or a table, and that is the graph or table it produces. Further transformations are not generally possible within the website or dialog box from which you operate the software. To be fair, there are usually options that you can vary, and you can sometimes filter an output table by one search term at a time, e.g., to narrow a list of all collocated terms to one particular term of your choice; but if you wanted to, say, identify a subset of documents based on the presence or absence of those collocates, and to proceed with a follow-up analysis of that subset, that's usually a move the tool won't support. You would want to program your own custom function.

Because I'm interested in what subject matter people are writing about, I've been drawn to *topic modeling*, a machine-learning method for identifying groups of words that tend to co-occur within sets of documents in a large corpus (Blei et al.; Weingart). Given several such groups to find, the algorithm calculates two sets of probabilities, one matching words to groups—the "topics"—and one matching topics to documents. The software generating the topics will generally also tell you how much of the corpus is associated with each topic. In the case that I describe below that software is MALLET,[3] an open-source command-line tool[4] most often used for topic modeling using the Latent Dirichlet Allocation method (but which is also generalizable to other applications, and other implementations, for those with a coding knowledge of the Java programming language).

By default, topics in MALLET (and, often, elsewhere) come labeled with the words associated with them at the highest probabilities, leading to labels such as these:

```
44. online web site media internet sites social users
```

3. MALLET is an acronym for MAchine Learning for LanguagE Toolkit; see http://mallet.cs.umass.edu/about.php. VoyantTools can also produce topic models, using a JavaScript implementation of the same algorithm (Latent Dirichlet Allocation); see https://voyant-tools.org/docs/#!/guide/topics.

4. The "command line" refers to the text-only interface accessed via Terminal on Mac computers, or PowerShell on Windows, and as the primary mode of engaging with Linux systems. For more information on the command line, see https://learnpythonthehardway.org/book/appendix-a-cli/ex1.html.

```
information blog community people post virtual com-
munication blogs website content page websites

45. technology digital computer computers tech-
nologies online media university web writing elec-
tronic technological composition software design
access information internet hypertext multimodal
```

From just the words alone, we can infer that documents with high levels of these two topics discuss matters of likely interest to the computers and writing community: online communication and community, electronic writing, multimodal design, and so on. But suppose you wanted to go beyond the words themselves, to look at abstracts or the full text as a way of understanding the topics? Andrew Goldstone has built a beautiful interactive browser that allows navigation among terms, topics, and documents (Goldstone), but to make it work with your own data, you'll need to be able to download and run his custom scripts in the R programming language—and potentially to modify them, too.

To help explain what that would entail, in the pages that follow I present an example of my own workflow in R. It begins with a question about how best to interpret the topic model, and a challenge in the way MALLET represents the model data. I then write a multi-step plan, in English, for how to surmount that challenge. As I move to implement the plan in code, it reveals new problems, requiring a revision of the plan. The final working implementation strings together several smaller pieces of code into a composite script. Interspersed throughout the examples—which will appear in a fixed-width font when they are written for the computer to execute—I will point out important patterns that transcend programming languages, as well as define terms or explain bits of R-specific syntax that are essential to reading comprehension of the code.

I write, revise, and execute my code using a piece of software called RStudio, an integrated development environment (IDE). What it integrates, specifically, are several elements of a programming workspace that will be common across languages: in that sense, my descriptions of R would be equally applicable to working in Python (through an IDE like Spyder or IDLE) or Ruby (through an IDE like Aptana). These include:

- A *text editor*, where you can write and store programs that can be executed repeatedly at a later time. These files can then also refer to each other, e.g., to load or execute a function you have written previously. Ideally, the text editor includes features to improve the legibility of the code, such as syntax highlighting[5] or bracket folding.[6]

5. That is, formatting chunks of text in different colors or weights to signal the role each chunk plays in the program.

6. That is, allowing the user to collapse or expand discrete sections of code that act as a single unit, which are often demarcated with parentheses or brackets.

- A *console*, in which commands entered take effect immediately and any outputs or errors are displayed.
- A list of current *variables* and what they store, sometimes called the *environment*. In RStudio, this list also includes any user-defined *functions* that are currently loaded.
- A *package manager*, indexing the external libraries that are available on the system and enabling quick installation, loading, or unloading of those libraries.
- Searchable *documentation* for functions (both those included in the base distribution and in distributed packages), clarifying the expected inputs and outputs. These often provide examples, though these are not always as illuminating as one might hope; in such cases, external sources like Stack Overflow (https://stackoverflow.com) become an essential part of my workflow.

RStudio also includes a history of commands executed by the console, and a window for displaying plots and charts. These elements work together: in most IDEs, you should be able to write code in the text editor and execute it in the console, either all at once or only selected lines; the outputs can then be stored in variables, if your code says they should, so you can do what you'd like with them next. This is essential for finding and fixing bugs in complex programs because it allows you to check interim values and confirm that your code is doing what you think it's doing. (For me, at least, this is not always the case, especially at first.)

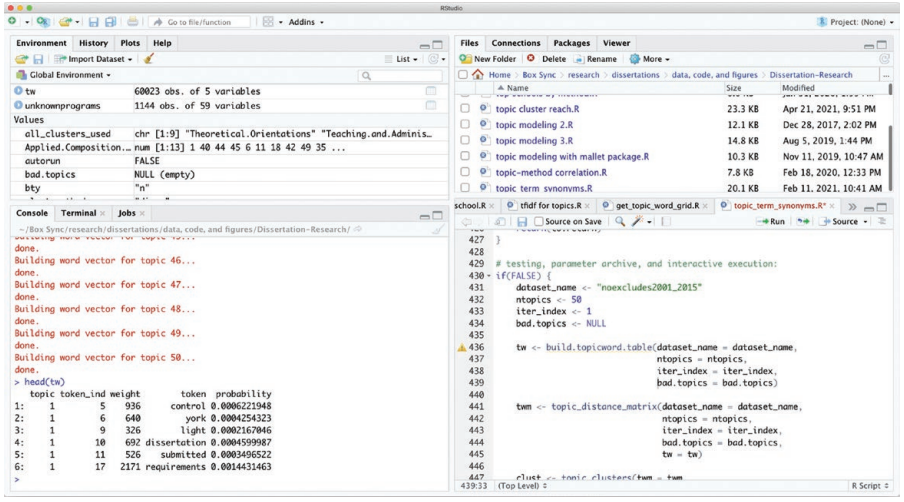


*Figure 17.1. The RStudio Integrated Development Environment (IDE). Clockwise from bottom right, the four quadrants show the text editor; the console; the environment of defined functions and variables; and the package manager. Documentation can be accessed through the "Help" tab in the upper left quadrant. (Note that these locations can also be customized.)*

## Setting the Stage: A Data Challenge and a Plan

For my research purposes, to better grasp the distribution of subject matter across RCWS dissertations, I was especially interested in finding clusters of related topics, like Topics 44 and 45 above. Following Rolf Fredheim and Ben Schmidt, I am convinced that trees showing such relations among topics help to mitigate the challenges in choosing the number of topics, as the LDA algorithm requires. Clustering topics also gives statistical heft to the intuition that areas of inquiry overlap and diverge in complex ways: when we say, for example, that computers and writing is like technical communication in some ways or like cultural studies in others, clustering can add to the evidence we have for such affinities and make them easier to compare.

To figure out what "relations among topics" means, though, takes a bit of digging around under the surface of topics' labels—and, in general, programming for data analysis often begins with determining the shape of the data, and whether reshaping it will make the data more amenable to computation. (It often does; see Lang et al. section 5.2.) Underlying MALLET's assessment of "top words" in a topic is a table of links between words and topics, stored as a plaintext (*.txt) file that looks like this:

```
0 em 11:13057 49:6232 33:736 15:654 4:190 36:123 40:119 37:103
44:67 48:44 42:40 26:38 18:33 16:25 13:24 3:4
1 vernacular 4:2080 19:1625 13:1611 20:1502 8:737 45:274 49:148
11:116 43:48 16:34 25:20 33:7 28:7 31:3
2 rhetorics 32:7604 1:3383 31:2085 8:1577 22:1480 4:1332 19:798
37:725 9:631 3:543 45:460 42:96 49:43 44:31
3 transgression 37:260 47:246 20:208 12:195 1:117 9:103 6:97
38:51 24:48 28:35 15:34 14:25 48:24 42:19
4 control 5:8000 20:6867 37:4924 45:4198 12:4089 17:3373
48:2853 30:2802 44:2679 41:2649 18:2496 1:2208 28:1720 3:1423
43:1028 0:936 27:871 47:824 26:805 15:638 6:572 31:542 11:449
22:438 46:398 16:378 21:345 42:342 19:258 23:250 7:245 49:211
29:125 38:117 24:110 39:99 4:96 32:41 13:41
```

Each line begins with an integer as an index, starting from zero and going up to the number of tokens in the corpus—in this case, over 1.6 million.[7] This is followed by a space and then the token itself, followed by a space-delimited list of key-value pairs; that is, each colon-linked pair includes a topic number and an observation count showing the model's current estimate of how many times that token appeared in the context of the given topic, i.e., accompanied by other tokens also associated with that topic.

7. I don't recommend opening a file of that size with Microsoft Word! To get the preview, essential for determining the file's structure–and thus for any follow-up analyses of that data–I used the Terminal command `head`, which displays only the first few lines of a given file.

MALLET's format here presents several obstacles for finding the proximity (or, equivalently, the distance) among topics. First, these topic-observation pairs are given not in order of topics, but in descending order of observation count for each token; there's no immediate way to read all the values for a topic of interest. Second, because many tokens are never observed in the context of many topics, the number of observations in each row varies significantly. So, we can't simply transpose the rows and columns of the table to get a readout of words observed for each topic—even though something similar must be happening somewhere under MALLET's hood to produce those labels of top words per topic.[8]

Just because the operation isn't a simple one, though, doesn't mean we can't perform it; we just have to do a little extra work. Now that we have a sense of what the data looks like, we can plan to transform the data into a shape that will make it easier to measure distances. To build a table indexed by topic, not token, we need to . . .

1. Read in the data, ideally in a format that's easy to index.
2. Find a given topic, if it exists, as the key in a key:value pair somewhere in the row for a token. Store the value (the observed count of that token) for that topic.
3. Repeat step 2 for each token, making a cumulative list of tokens and counts for our given topic.
4. Repeat step 3 for each topic.

Once we have that table of topics, existing functions should be able to measure the distances.

## Coding's Core: Defining Functions, Explaining Syntax

To begin step 1, reading the data file into R, I begin like this:

```
get.wordtopic.grid <- function(dataset_name = "noex-
cludes2001_2015",
ntopics = 50,
iter_index = 1
){
# Things will happen here
}
```

I'm going to pause there for a moment to explain the syntax, for any readers new to code or to R specifically. In R, most alphanumeric strings[9] are treated by

8. Perhaps if I were fluent in Java, I could locate and save such an internal MALLET state. Alas! But all the more reason to become proficient in more programming languages.

9. How a "string" is delineated can vary from language to language. In R, as shown here, only a space ends a variable name, allowing periods and underscores to be included; in JavaScript, by contrast, the period is reserved for another function, and would not be allowed.

default as variables—labels, essentially, for some other object in memory. Here, for example, `get.wordtopic.grid` is a variable, as are `dataset_name`, `ntopics`, and `iter_index`. A string that's meant to be a text value, rather than a variable, is framed in quotation marks, as in my name for the dataset I'm analyzing here, `"noexcludes2001_2015"`. To define the contents of a variable, you can *assign* something to it, either with a single equals sign (as on the right, where the number of topics, `ntopics`, is defined as 50) or with a kind of arrow made of a less-than sign and a hyphen (`<-`), which stores whatever comes to the right of it inside the variable to the left.

In this case, what's "to the right of" the arrow is much longer than a single value, because it begins with `function`. This signal word tells R to include, under the label `get.wordtopic.grid`, everything inside the parentheses and curly braces that follow. I'll have more to say about functions in a moment.

The final piece of syntax you'll need to read R code is the *comment*, shown here in gray. Essentially, any text from the `#` symbol through the end of the line where it appears is ignored by the computer. Instead, such "commented out" lines are aimed at a human audience, whether the programmer themselves or any other readers coming into the code, to try to make it easier to understand. (Note that other programming languages will use other symbols to signal the start of comments, and in some cases also to signal explicitly where they end.) Comments are most often used to label chunks of code, explaining what they're meant to do; they can also be written in advance of the code itself, as a way of making plans and marking placeholders.

Running the code snippet above will, as Auden said of poetry, make nothing happen—and not only because I've used a comment to simplify it. The code, as I mentioned above, defines a *function*, which means it can take a series of inputs, or *parameters* (the variables inside the parentheses), and act on them, eventually returning an output. Defining the function, however, does not execute the function; rather, it waits for something else—another script, or the user at the console—to pass it inputs. In that sense, as Auden also said of poetry, a function is "a way of happening, a mouth" through which information can later stream.

When I first started to use programming as a research method, while writing my own dissertation, I initially thought of scripts as files that would immediately execute a series of commands whenever I opened them, and that running entire files would be the primary way I would engage in analysis. If anyone reading this is sympathetic to that view, I hear you–and yet. I quickly came to realize that I needed, often, to batch those batches together, in various configurations. I came to realize that if I wanted to run the same analysis on two different subsets of data, or to try an analysis with two different assumptions about (say) how much of a document had to involve a particular topic for the document to count as being "about" that topic, I wanted to be able to save those different configurations without having to save multiple versions of essentially the same file, different only in one or two lines. I realized, in other words, that instead of files that al-

ways *run* functions, I wanted files that would *load* functions, so I could then call the functions interactively, at the console. Many of my files now consist solely of a set of custom function definitions, followed by a section which will never run on its own, where I save function calls in the various configurations I want to return to.[10]

## Loading Data and the Importance of Parameters

Armed with that reading framework, I can now share a fuller version of the code snippet above, incorporating some more content inside the curly braces.[11] Inline comments beginning with a single # are standard descriptive comments; to gloss specific R commands for this chapter, I'll use ##.

```
get.wordtopic.grid <-
  function(dataset_name = "noexcludes2001_2015",
           ntopics = 50,
           iter_index = 1
  ){
  # Build the filename to load, based on parameters of
  # the model (which dataset, the number of topics, the
  # particular iteration of the model-building
  # algorithm) and MALLET's naming convention.

  filename <- paste0(dataset_name, "k", ntopics,
              "_iter", iter_index, "_wordtopics.txt")

    ## NB: the paste0() function, built into R,
    ## combines its parameters into a single string,
    ## with no spaces.

  # Next, store and format the file – but only if it
  # exists. Otherwise, return an error message
  # clarifying what went wrong.

  if(file.exists(filename)) {
```

10. Careful functional programming also solves a problem of variable isolation or contamination, what is sometimes referred to as a "clean" or "unclean" workspace. In many languages, including R, values assigned to variables within the scope of a function do not propagate outside of that function: in fact, variables declared only within that function do not even exist except while the function is being executed. This is important because it prevents values from being overwritten when, as sometimes (okay, often) happens, the same variable name is used in more than one file. Should some, but not all, shared variables be changed in the global environment assumed by freestanding batch-style code execution, the scripts would run with a mixed set of assumptions, leading to errors or nonsensical results that may be difficult to retrace.

11. Note that I'm ignoring, for the sake of streamlining the code, considerations like data files located elsewhere on the disk, but those locations could be included as well, if needed.

```
      wt <- read.table(filename, header=FALSE,
            fill=TRUE, col.names=c("index", "token",
            paste0("TopicRanked", 1:ntopics))

    ## NB: read.table() is also a built in R function,
    ## with parameters for the source file, how to name
    ## columns, etc. Here, paste0() is used to generate
    ## column names in a sequence from 1 to the value
    ## of the ntopics variable.

      require(data.table)
      wt <- data.table(wt)

    ## require() loads a library of external functions;
    ## here, I use the data.table() function in that
    ## library to improve the formatting and indexing
    ## of the word-topic table.

  } else {

    stop("'get.wordtopic.grid R': could not load
       word-topic pairs from file ", filename)

    ## stop() is a built-in R function to exit early
    ## and return an error message.
  }

  return(wt)

    ## Exit the function, with the value of the wt
    ## variable as the output.
}
```

More than the specific tasks performed by this short function, I want to draw your attention to the ways the function's parameters—the list of inputs—are incorporated into the function body. Almost immediately, they are combined into a new variable, `filename`, which in turn is used to check conditions (`if(file.exists(filename))`) and, depending on the outcome of that check, to generate either a new data object or an error message. There is little in the function that can work without drawing on one or more of the parameters named. Significantly, parameters are *inherently changeable*: the values after the equals signs at the start of the function will be used by default, if nothing else is specified in the function call, but they can be overridden easily at call time. Depending on how the function is written, no defaults need to be provided at all—in which case, some value for that parameter must be named outright every time the function is called. In this way, parameters encourage researchers to be explicit about what conditions they're assuming; they also encourage systematic variation of those inputs, which helps me interrogate my assumptions, and what they reveal or mask.

In the present example, by splitting out the `dataset_name` as a changeable parameter, I signal that I expect to use different corpora and subcorpora as I continue with this project. By allowing the number of topics to vary, in `ntopics`, I remind myself that the topic model will look different if I split the corpus into 100, or 150, or 10 topics, instead of the 50 I'm working with primarily. And the final parameter, `iter_index`, reminds me that the LDA algorithm is non-deterministic, and that even repeated runs of that algorithm with the same dataset and number of topics will vary, at least a little, in its assignments of tokens to topics. (We can expect that the major divisions of the corpus will remain—one reason I want to study topic *clusters*—but this expectation needs to be interrogated.)

## Task Decomposition: Lines, Loops, and Mid-Process Feedback

The `get.wordtopic.grid()` function above, when called, will do more than load the plaintext output by MALLET into R; it will also convert the space-delimited values into an actual table, aligning and adding column labels as it goes, making the data much easier to read and index. The first few lines now look like Table 17.1, with the columns continuing off to the right up through TopicRanked50:[12]

**Table 17.1. Data loaded into R and reformatted**

| index | token | TopicRanked1 | TopicRanked2 | TopicRanked3 | TopicRanked4 | … |
|---|---|---|---|---|---|---|
| 0 | em | 11:13057 | 49:6232 | 33:736 | 15:654 | … |
| 1 | vernacular | 4:2080 | 19:1625 | 13:1611 | 20:1502 | … |
| 2 | rhetorics | 32:7604 | 1:3383 | 31:2085 | 8:1577 | … |
| 3 | transgression | 37:260 | 47:246 | 20:208 | 12:195 | … |
| 4 | control | 5:8000 | 20:6867 | 37:4924 | 45:4198 | … |
| 5 | york | 12:20054 | 18:12124 | 13:9297 | 1:6256 | … |

RStudio makes it possible to view the results in the console or in a more tabular data inspector. But if we want to do more than look at the results—which was the whole point of programming, rather than using pre-built tools—then we need also to bind the results to a variable that we can then pass along to other functions. Thus, when calling the function, we instruct R to hold onto the output: `wt <- get.wordtopic.grid()`. (Leaving the parentheses empty uses the default values for dataset, number of topics, and model iteration.) After entering this call in the console, the variable `wt` will now hold all million-plus lines and 52 columns.

12. In truth, for this particular iteration of the model, the columns TopicRanked50 and TopicRanked49 are blank–but most have a value even as far out as TopicRanked48.

In the plan outlined above, we have now reached step 2, in which we'll build a function to find and store values from one small unit, so we can then call that function repeatedly for the whole big list of units:

1. Read in the data, ideally in a format that's easy to index. [Done!]
2. Find a given topic, if it exists, as the key in a key:value pair somewhere in the row for a token. Store the value (the observed count of that token) for that topic.
3. Repeat step 2 for each token, making a cumulative list of tokens and counts for our given topic.
4. Repeat step 3 for each topic.

Because we'll need to search every topic and row, I don't want default values for these parameters. And by requiring that a word-topic table be passed in as a parameter, I insist that it already exist before this new function is called: otherwise, we would have to build it anew each time, at a tremendous cost of time.

```
find.topic.in.one.row <-
  function(topic,            # what we're looking for
           rowindex,         # which row to look in
           wt                # a word-topic table
  ){
   # Build the search string from a topic number,
   # converting from 0-indexed to 1-indexed

   my_expr <- paste0("^", topic-1, ":")

   # Use it to find a column. It should match exactly
   # one, or none.

   colindex <- grep(my_expr, wt[index == rowindex])

      ## grep() is a built-in search function.

   # If nothing's found, the length of colindex will
   # be 0, which an "if" statement will interpret as
   # False; if the length is greater than 0, "if" will
   # interpret it as True.

   if(length(colindex)) {
      value <- just.value(wt[rowindex, ..colindex])

      ## just.value() is a function I've defined
      ## elsewhere to extract the second half of a
      ## key:value pair, a task that recurs fairly
      ## often.

   } else {
     value <- NULL
```

```
    }

    return(value)
}
```

One key advantage of the process of *decomposing* the overall data-transformation task into a small chunk we can repeat—into a series of functions, rather than one big function—is that it allows us to confirm each small chunk works as expected . . . or doesn't. When I run find.topic.in.one.row() on a known token ("vernacular," with row index 1 in the word-topic table shown earlier), and I search for its top-shown topic (topic 4), the value returned is not 2080, as expected, but NULL. A little digging in the help page for the search function, grep(), reveals why: it assumes that the search space will have a particular, consistent format, and it turns out that R tables use that format only for columns, not rows. The search is quietly failing. Had we not put specific known values in to test with, we could have looped through tens of thousands of rows and gotten no results—or worse, misleading results–before realizing something was wrong.

I have argued before (Miller) that writing is like finding one's way through a maze,[13] and in my experience writing code is similar. The setback of an unexpected data-type mismatch is a frequently recurring obstruction in the maze. One solution is to try to find another tool, i.e., another function, that *does* work with the datatype we're looking at; in this case we'll need to delve into a side labyrinth of linked help pages, maybe even searching the web to discover a whole new library that deals with tables in a new way. Another solution is to re-examine our initial plan and see if there's a way to just get *around* the barrier without too much cost.

In this case, because the search tool already works with columns, we can update our plan to search by columns instead of by rows:

1. Read in the data, ideally in a format that's easy to index [still done!]
2. Find a given topic, if it exists, as the key in a key:value pair everywhere it appears in the column for a particular topic-rank
3. For each match in step 2, note the token, and store the observed count of that token for that topic
4. Repeat steps 2 and 3 for each column / topic-rank, making a cumulative list of tokens and counts for our given topic
5. Repeat steps 2, 3, and 4 for each topic

The row vs. column question is a recurrent problem-class for data analysis: the sooner we learn to recognize it as a pattern, the sooner we'll notice when it happens, and the more confident we can be that we have a solution . . . and what

13. More specifically, a Zelda-like dungeon filled with traps, puzzles, and enemies–but with treasures and increased life-force as a reward for making it through.

kinds of new wrinkles those solutions introduce. In this case, one wrinkle is that we've gone from a unique result for each step in the loop to a list of matches, because the same topic can be top-ranked for more than one word. I therefore need to return a table at each point, which will need to be merged later.

```
find.topic.in.one.col <-
  function(topic,            # what we're looking for
           rank.col,         # which column to look in
           wt                # a word-topic table
  ){

  # load required library
  require(data.table)

  # Build the column name from a topic rank
  colname <- paste0("TopicRanked", rank.col)

  # Build the search string from a topic number,
  # converting 0-index to 1-index)
  my_expr <- paste0("^", topic-1, ":")

  # Search the column, allowing for more than one
  # possible result
  index <- grep(my_expr, wt[[colname]])

  # Use the search results to extract tokens
  tokens <- wt[index, token]

  # ... and key:value pairs
  key.value.pairs <- wt[index, ..colname]

  # Extract values from the key:value pairs
  values <- sapply(key.value.pairs[[1]], just.value)

    ## The need to apply a function across all members
    ## of a list is common enough that R has a set of
    ## built-in functions, including sapply(), to make
    ## it easy.

  # and return as a table
  result <- data.table(token_ind = as.integer(index),
                       token = as.character(tokens),
                       weight = as.integer(values))
  return(result)
}
```

Again, we'd better test to make sure that works! Trying as above to find "vernacular," we can run `find.topic.in.one.col(topic = 5, rank.col = 1, wt = wt)`, which returns this table (of which only the first 5 and last 5 rows are shown):

```
token_ind token weight
1: 2 vernacular 2080
2: 267 chinatown 35
3: 462 american 33101
4: 474 americans 11094
5: 568 races 1494
---
24871: 1615776 asunky 1
24872: 1615781 lesssss 1
24873: 1615782 unprobable 1
24874: 1615783 sollubles 1
24875: 1615784 hypnotical 1
```

There's our expected result at the top of the list. We're safe, then, to move on to step 4: repeat the process across all columns, building up a combined list of tokens and weights for a single topic.

## Insights and Upgrades Along the Way

Eagle-eyed readers will have noticed that the top topic for "vernacular' had been listed as 4, but our search was for topic 5. This is because the fifty topics in MALLET's output are numbered 0-49 but trying later to write a query with a topic of 0 would return an error, so we need to increment them all by one. Care to guess whether I anticipated that error in advance, or had to stub my toe on the error to discover it? There's a reason I try to move slowly, confirming my footing at each step.

Other things we can discover by moving slowly and inspecting our interim results include those strange words down at the bottom of the frequency list: "asunky," "lesssss," "unprobable." Each of these tokens was observed only once, even though we're limiting our search here to the topic with the *highest* value in each row. With the table at well over a million rows, we can obtain a significant speed boost by setting a lower bound on how many observations we want to consider: another function, taking as parameters the word-topic table and a threshold frequency for each word. (After some experimentation, I found that a threshold of 2 reduces the table from 1,616,842 words to 544,036; a threshold of 5 drops it down to a still sizable 254,092. And five mentions of a term in a few thousand documents still signals a very low-frequency word.)

## Now, Where Were We?

I jest, but for a serious reason. Programming—and especially debugging—frequently asks me to scale-shift, sometimes zooming way in to the level of single punctuation marks for debugging, sometimes zooming out to remember why I wanted this function in the first place; a lot of the time is spent in between. This is also true of other forms of writing, to be fair, but it sometimes seems an especially

prominent feature of programming, which rewards coders for breaking down large-scale challenges—like measuring the similarity of topics—into ever smaller, modular pieces that can be carefully inspected and quality-controlled before being assembled into ever larger, more complex machines. I forgive you, and hope you'll forgive me, if you'd momentarily forgotten that we're building a table directly relating topics to the observed frequencies of tokens in the context of those topics. Or that the purpose of building such a table was so that we can measure the distance from one topic-word vector to another, and thus to identify clusters of topics. The topic-clusters themselves are in service of the larger goal of mapping the range of research activity in graduate dissertations, as representative of the field. But, to scale back down again, the example of the dissertation topic-clusters is, in this chapter, just one example of the kind of custom analytical work made possible by programming, work not supported by more plug-and-play digital research tools.

## Wrapping Up by Wrapping Functions in Other Functions

From above, we have a function that finds the tokens associated with a topic when it appears at a particular rank. Two more functions will suffice to get us from there to the fully searchable table: one, keeping the same topic, that iterates across all ranks; then another to iterate *that* function across all topics. I'll simplify a little to show the essentials:

```
find.topic.in.all.cols <-
  function(topic,            # what we're looking for
           ntopics,          # how many loops to make
           wt,               # a word-topic table
           threshold = 5     # minimum weight per token
  ){

  require(data.table)

  # start with an empty container,
  # with specified data types

  topic_word_vec <- data.table(token_ind = numeric(),
                               token = character(),
                               weight = numeric())

  # then, to fill it, loop through the columns,
  # from 1 through the total number of topics,
  # because that will also be the largest possible
  # topic rank.

  for (column in seq_len(ntopics)) {

    # each time through the loop, attach one more row
    # to the existing list...
```

```
    topic_word_vec <-
        rbindlist(list(topic_word_vec,

        # ... by calling the function defined earlier.
        find.topic.in.one.col(topic, column, wt)
    ))
  }

  # (here we could trim, sort, label, do some norming,
  # etc)

  return(topic_word_vec)
}
```

The final function to build the topic-word table (which I creatively call `build.topicword.table()`), works very similarly to the one above: start with an empty table with labeled columns, loop through all the topics, and at each step call the previous function. But this time, the previous function is `find.topic.in.all.cols()`, which in turn calls `find.topic.in.one.col()`, which in turn relies on `get.wordtopic.grid()`, so that by the time we've assembled the whole thing, we no longer need to explicitly run those earlier scripts. Instead, they will be called only from `build.topicword.table()`, which therefore consolidates all the parameters for the nested function calls, and returns a single clean output: a topic-word table I call `tw`.

To finally measure distances, a fourth function cleans up `tw` by ensuring all the words are in the same order for each topic, even if some words have zero observations in the context of some (many) topics; then it isolates just the quantitative values (i.e., it strips out the tokens themselves and their indices within the original MALLET table), and norms them by the number of observations for each topic. This allows the resulting numerical matrix to be passed along to a fifth function that calculates distances between its constituent vectors—which can, finally, produce the topic clusters we were initially after. But only through still another function, with its own parameters and choices to make.

## The Takeaway for Digital Research in Writing

As I said earlier, I realize that was a lot of code to throw at you. But my point isn't to show off my sweet, sweet programming skills—in fact, I'm sure I'm exposing some major inefficiencies or infelicities that a professional software engineer would be able to diagnose and fix immediately. Nor are the details of my control flows (if/else statements, for loops, and the like) or the specific R syntax, or calls to pre-made functions from base or imported packages, something I'm trying to teach. However, in the service of explaining my digital research methods, I do think it's important to illustrate 1) the frequent *presence* of control flows and 2) the significance of *each new function calling previous ones.*

In other words, 1) an essential aspect of programming as methodology is preparing to handle and respond to changing contingencies: not to assume that the data you're looking at now, and the circumstances in which you're looking at it, are the only situation for which the code should work. For instance, beginning programmers may be tempted to reference columns of a data table by column number, rather than by a column name–or by assuming the columns will always be named a certain way, rather than looping through an inferred set of names. (I know I was.) However, these assumptions are prone to breakage, e.g., if something inserts or removes a column, such as a row index that may become shifted by saving to Excel. Depending on the outcome, this may result in particularly insidious silent changes, where the function compiles and runs with no explicit errors–but with all the conclusions you would draw from its output based on a mismatch compared to its inputs. True, there's a risk of over-preparing for conditions that never actually arise, but learning to make my code less brittle has been key to my growth as a researcher-who-programs, from when I first started in graduate school and continuing today.

I also wanted these examples to highlight 2) the iterative and cross-referential nature of the process. In contrast to the output-as-endpoint default behavior of most ready-to-use digital analytical tools, functional programming suggests that any output can become an input. The diction surrounding functions emphasizes their relationality: you *call* functions, *pass* them information, and they *return* something back to you. At the same time, the sense of completion implied by that return is only ever temporary: what is returned can be passed again. While the examples above were constructed to show a nested set of calls—outputs passed, as it were, straight up the ladder of scale, such that we could eventually invoke all of them with a single call—the modular nature of the functions also means they can stand in relation to more than one partner.

Take, for example, the table `tw` returned by `build.topicword.table()`. While I built it initially to find that distance matrix, and thus clusters of dis/similar topics, I was also able to repurpose it for the sake of changing how topics are labeled. As I noted earlier, by default MALLET labels topics with the twenty words most frequently found in the context of those topics. But because some topics are related, some words will show up high on the list of several topics. To better differentiate the topics from one another, I adapted a technique more often used in differentiating subcorpora of documents: Term Frequency * Inverse Document Frequency, or TFIDF. This technique reduces the observed occurrence of terms within each single document in proportion to the observed occurrence of those terms across the whole set of documents, essentially muting the terms that have high frequencies purely by virtue of being common words in general. My adaptation, TFITF, substitutes Topic for Document, and for the same reason: to highlight the terms that are more specific to individual topics than to the corpus. As it turns out, the token frequencies needed for the calculation, both within individual topics and across the dataset, are trivial to

derive from the topic-word table `tw`. Analytical scripts, once programmed, beget further analytical scripts.

## Conclusions: Learning to Program, and Learning to Like It

Like all literacies, coding is social. As Vee argues, "Ultimately, all programming is collaborative—although it is often asynchronously so. Even if they aren't working alongside other programmers physically or online, programmers work with languages, machines, and programming environments designed by others. They work with libraries of procedures or codebases or frameworks programmed by teams of other programmers" (128). This should, I hope, read as inspiring for any of you who may be new to programming and intimidated by the idea of picking it up: not only are you not expected to go it alone, the very software you'll be using to practice is infused with the traces of prior learners, not only in the languages and the libraries but also in the copious documentation left by their designers. Digital humanists share and discuss their code online, including through sites like The Programming Historian (https://programminghistorian.org), which hosts "novice-friendly, peer-reviewed tutorials that help humanists learn a wide range of digital tools, techniques, and workflows to facilitate research and teaching." Popular languages like R, Python, and JavaScript have large and active online communities, with forums like Stack Overflow (https://stackoverflow.com), where questioners can find answers and explanations—usually without a delay, because chances are good that someone else has had that question before, and the answers are already online – and even vague or halting questions are often met not with dismissal, but with probing queries that help to clarify the nature of the problem. The barriers for entry into digital research methods are lowered not only by software to use as-is, but also by resources (including *human* resources) to help you customize how one bit of software feeds into another.

Even if you find that it's easier to collaborate with a programmer than to write code yourself, trying your hand at programming can facilitate that collaboration. Knowing the kinds of data structures out there, or the way the computer already structures the data you want to analyze or circulate, will help you describe your project and ask your own probing queries about what the programmer can do for you; and knowing something about functions, loops, and changeable parameters vs. fixed values will help you parse or push back on the jargon you may get back in response. Programming ourselves can also help us appreciate the limitations of what code can do, and what modifications are likely to be easy or heavy lifts.

I want to be clear that I'm not advocating we ignore all the excellent digital tools already out there, from some false sense that only code we ourselves have written can be trusted. (On the contrary, some things, like security protocols, are probably better left to the experts.) The choice between existing tools and custom scripts is a false binary: as I hope I demonstrated above with my transformation of output from MALLET, itself one of those ready-to-use tools, they often work especially

well in concert. And for many research questions, an existing analysis may be just what you need. What I am saying, though, is that I have found real value in being able to think through my inquiry by thinking through code, in at least two ways.

*Programming is epistemic.* Just as organizing thoughts to teach someone (whether in person or in writing) can help us bring to consciousness and further develop what we think, so too can organizing our questions to explain them to a computer. Except now we also have to (get to) explain the structure of our data, too—and that helps us better grasp what we have, which, in turn, helps us formulate both new questions and new conclusions. Writing a custom function means being aware enough of my specific research goals and the shape of my data to speed re-entry and replication, yet flexible enough in my assumptions to respond to changes in the data source or the possibility of the function's reuse in another context.

*Programming is rewarding.* More affectively, I find that programming as a research method affords more frequent opportunities for positive reinforcement than most other forms of writing I do. It's still challenging, with a lot of wrong turns, searching for help, and plenty of debugging—but it's "pleasantly frustrating" (36), to borrow James Gee's description of the appeal of challenging video games: it keeps me coming back for more, with challenges that "feel hard, but doable" (ibid). When an incremental piece of a larger analysis returns the expected result, or when a series of functions finally hand off to one another without error after a while in the weeds of debugging, the feeling of getting it right is just incredibly satisfying. Composing in prose can offer a similar feeling of rightness, in the pleasure of a balanced phrase, say, or a final paragraph that satisfies, finally, the itch of the opening. But the all-at-once-ness of writing, as Ann Berthoff put it (86), means that so much remains in flux throughout the composing process that it can be hard, before the ending, to know what's really worked. Programming is more modular, with more frequent feedback already built in.

In fact, in some ways this pleasurable difficulty is also a liability: it's easy to feel that there's more tinkering to do, a more efficient approach, a follow-up question to ask, one more level to play, and each new attempt introduces new problems to solve. It's important to keep aware of time. But as Gee also argues, this sensation of earned reward is an excellent motivator for learning, and with programming, as with any literate skill, there is always more to learn.

## Works Cited

Auden, W. H. "In Memory of W.B. Yeats." *Another Time*, W. H. Auden, Random House, 1940, https://poets.org/poem/memory-w-b-yeats/.

Berthoff, Ann E. *The Sense of Learning*. Boynton/Cook, 1990.

Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent Dirichlet Allocation." *The Journal of Machine Learning Research*, vol. 3, 2003, pp. 993-1022.

*Corpus of Contemporary American English (COCA),* https://www.english-corpora.org/coca/. Accessed 18 July 2020.

Craig, Todd. "'Makin' Somethin' Outta Little-to-Nufin': Racism, Revision and Rotating Records–The Hip-Hop DJ in Composition Praxis." *Changing English*, vol. 22, no. 4, 2015, pp. 349-64.

Desmet, Christy, Ron Balthazor, Robert Cummings, Nelson Hilton, Angela Mitchell, and Alexis Hart. "<emma>: Re-Forming Composition with XML." *Literary and Linguistic Computing*, vol. 20, no. 1, 2005, pp. 25-46, https://doi.org/10.1093/llc/fqi023.

Dighton, Desiree. "Arranging a Rhetorical Feminist Methodology: The Visualization of Anti-Gentrification Rhetoric on Twitter." *Kairos: A Journal of Rhetoric, Technology, and Pedagogy*, vol. 25, no. 1, 2020, http://kairos.technorhetoric.net/25.1/topoi/dighton/attuning-to-silences.html.

Feibush, Laura. "Gestural Listening and the Writing Center's Virtual Boundaries." *Praxis*, vol. 15, no. 2, 2018, http://www.praxisuwc.com/feibush-152.

Fredheim, Rolf. "Visualising Structure in Topic Models." *Quantifying Memory*, 2013, http://quantifyingmemory.blogspot.com/2013/11/visualising-structure-in-topic-models.html.

Gee, James Paul. "Good Video Games and Good Learning." *Phi Kappa Phi Forum*, vol. 85, no. 2, 2005, pp. 33-37.

Goldstone, Andrew. *dfr-browser: Take a MALLET to Disciplinary History*, http://agoldst.github.io/dfr-browser/. Accessed 26 June 2014.

Hart-Davidson, Bill, Jeff Grabill, Mike McLeod, and Melissa Graham Meeks . "About Eli Review." *Eli Review*, https://elireview.com/about/. Accessed 29 Jan. 2021.

Kantrowitz, Dana. "The Making of a Poem, Live and Uncensored." *Acts of Revision: A Guide for Writers*, edited by Wendy Bishop, Boynton/Cook Heinemann, 2004, pp. 134-43.

Kaufer, David, et al. *DocuScope Project*, http://www.cmu.edu/dietrich/english/research-and-publications/docuscope.html. Accessed 29 Jan. 2021.

Klein, Lauren F., and Catherine D'Ignazio. *Data Feminism*. e-book ed., MIT Press, 2020, http://ebookcentral.proquest.com/lib/pitt-ebooks/detail.action?docID=6120950.

Lang, Susan, Laura Aull, and William Marcellino. *A Taxonomy for Writing Analytics*. 2019, pp. 13-37.

Lang, Susan, and Craig Baehr. "Data Mining: A Hybrid Methodology for Complex and Dynamic Research." *College Composition and Communication*, vol. 64, no. 1, 2012, pp. 172–94.

LeBlanc, Paul J. *Writing Teachers Writing Software: Creating Our Place in the Electronic Age. Advances in Computers and Composition on Studies Series*. e-book ed., National Council of Teachers of English, 1993, https://eric.ed.gov-/?id=ED357369.

Lindgren, Chris, and Jim Ridolfo. "Rhetmap.Org: Composing Data for Future Re-Use and Visualization." *PraxisWiki*, 25 June 2020, http://praxis.technorhetoric.net/tiki-index.php?page=PraxisWiki%3A_%3Arhetmap.

Lockridge, Tim. "Building Rhetorlist: A Call for Small, Meaningful Projects in Rhetoric and Composition." *Kairos: A Journal of Rhetoric, Technology, and Pedagogy*, vol. 24, no. 2, 2020, http://kairos.technorhetoric.net/24.2/disputatio/lockridge/index.html.

Melançon, Lisa. *USF Writes*. 2020, http://myreviewers.usf.edu/research.
Mueller, Derek. *Network Sense: Methods for Visualizing a Discipline*. The WAC Clearinghouse/UP of Colorado, 2017, https://doi.org/10.37514/WRI-B.2017.0124.
Miller, Benjamin. "Metaphor, writer's block, and the legend of zelda: A link to the writing process." *Rhetoric/Composition/Play through Video Games*, edited by Richard Colby, Mathew S. S. Johnson, and Rebekah Shulz Colby, Palgrave Macmillan, 2013. pp. 99-111.
"NativeLand.Ca." *Native-Land.ca - Our Home on Native Land*, https://native-land.ca. Accessed 25 Jan. 2021.
Noble, Safiya Umoja. *Algorithms of Oppression: Race, Gender and Power in the Digital Age*. New York UP, 2018.
Palmquist, Mike. "A Brief History of Computer Support for Writing Centers and Writing-Across-the-Curriculum Programs." *Computers and Composition*, vol. 20, no. 4, 2003, pp. 395-413.
RStudio Team. *RStudio: Integrated Development Environment for R*. RStudio Inc., 2019, http://www.rstudio.com.
Schmidt, Benjamin M. "Words Alone: Dismantling Topic Models in the Humanities." *Journal of Digital Humanities*, vol. 2, no. 1, 2012, pp. 49-65.
Turner, Heather Noel, and Laura Gonzales. "Visualizing Translation." *Kairos: A Journal of Rhetoric, Technology, and Pedagogy*, vol. 25, no. 1, 2020, http://kairos.technorhetoric.net/25.1/topoi/turner-gonzales/casestudy.html.
Vee, Annette. *Coding Literacy: How Computer Programming Is Changing Writing*. The MIT Press, 2017.
*Voyant Tools*. http://voyant-tools.org. Accessed 6 June 2017.
Weingart, Scott B. "Topic Modeling for Humanists: A Guided Tour." *The Scottbot Irregular*, 25 July 2012, http://www.scottbot.net/HIAL/?p=19113.
Wresch, William. *The Computer in Composition Instruction: A Writer's Tool*. National Council of Teachers of English, 1984.