Hands-on with the Distant Reader: A Workbook

This workbook outlines sets of hands-on exercises surrounding a computer system called the Distant Reader -- https://distantreader.org.

By going through the workbook, you will become familiar with the problems the Distant Reader is designed to address, how to submit content to the Reader, how to download the results (affectionately called "study carrels"), and how to interpret them. The bulk of the workbook is about the later. Interpretation can be as simple as reading a narrative report in your Web browser, as complex as doing machine learning, and everything else in-between.

You will need to bring very little to the workbook in order to get very much out. At the very least, you will need a computer with a Web browser and an Internet connection. A text editor such as Notepad++ for Windows or BBEdit for Macintosh will come in very handy, but a word processor of any type will do in a pinch. You will want some sort of spreadsheet application for reading tabular data, and Microsoft Excel or Macintosh Numbers will both work quite well. All the other applications used in the workbook are freely available for downloading and cross-platform in nature. You may need to install a Java virtual machine in order to use some of them, but Java is probably already installed on your compu9er.

I hope you enjoy using the Distant Reader. It helps me use and understand large volumes of text quickly and easily.

Eric Lease Morgan emorgan@nd.edu
Notre Dame (Indiana)

February 22, 2020

Table of contents


```
 I. What is the Distant Reader, and why should I care?
     A. The Distant Reader is a tool for reading
     B. How it works
     C. What it does
 II. Five different types of input
     A. Introduction
     B. A file
     C. A URL
     D. A list of URLs
     E. A zip file
     F. A zip file with a companion CSV file
     G. Project Gutenberg and the Distant Reader
     G. Summary
III. Submitting "experiments" and downloading "study carrels"
 IV. An introduction to study carrels
  V. The structured data of study carrels; taking inventory through the manifest
 VI. Using combinations of desktop tools to analyze the data
     A. Introduction - The three essential types of desktop tools
     B. Text editors
     C. Spreadsheet/database applications - Excel and Excel recipes
     D. Analysis applications
         i. Wordle and Wordle recipes
```

ii. AntConc and AntConc recipes
iii. OpenRefine and OpenRefine recipes
iv. Topic Modeling Tool and Tool recipes
VII. Using command-line tools to "read" a study carrel
A. Building a library
i. Harvest
ii. Add scope notes
B. Feature extraction
i. N-grams
ii. Broader concepts
iii. Questions
iv. Sentences containing keyword
C. Classification and clustering
i. Cosine similarities
ii. Classifying/tagging documents
iii. Visualize clusters
iv. Visualize network diagram of nouns
D. Topic modeling
i. Visualize topic model
ii. Create Modeling Tool metadata file
iii. Visualize comparison of topics to metadata
E. Measuring big ideas, name dropping, and colorfulness
F. Searching
i. Concordancing
ii. Semantic indexing
a. Create semantic index
b. Search semantic index
iii. Free-text indexing with facets
a. Create free-text index
b. Search free-text index
VIII. Summary/conclusion
IX. About the author

What is the Distant Reader, and why should I care?

The Distant Reader is a tool for reading.

The Distant Reader takes an arbitrary amount of unstructured data (text) as input, and it outputs sets of structured data for analysis — reading. Given a corpus of any size, the Distant Reader will analyze the corpus, and it will output a myriad of reports enabling you to use & understand the corpus. The Distant Reader is intended to supplement the traditional reading process.

The Distant Reader empowers one to use & understand large amounts of textual information both quickly & easily. For example, the Distant Reader can consume the entire issue of a scholarly journal, the complete works of a given author, or the content found at the other end of an arbitrarily long list of URLs. Thus, the Distant Reader is akin to a book's table-of-contents or back-of-the-book index but at scale. It simplifies the process of identifying trends & anomalies in a corpus, and then it enables a person to further investigate those trends & anomalies.

The Distant Reader is designed to "read" everything from a single item to a corpus of thousand's of items. It is intended for the undergraduate student who wants to read the whole of their course work in a given class, the graduate student who needs to read hundreds (thousands) of items for their thesis or dissertation, the scientist who wants to review the literature, or the humanist who wants to characterize a genre.

How it works

The Distant Reader takes five different forms of input (described in the next section). Once the input is provided, the Distant Reader creates a cache — a collection of all the desired content. This is done via the input or by crawling the 'Net. Once the cache is collected, each & every document is transformed into plain text, and along the way basic bibliographic information is extracted. The next step is analysis against the plain text. This includes rudimentary counts & tabulations of ngrams, the computation of readability scores & keywords, basic topic modeling, parts-of-speech & named entity extraction, summarization, and the creation of a semantic index. All of these analyses are manifested as tab-delimited files and distilled into a single relational database file. After the analysis is complete, two reports are generated: 1) a simple plain text file which is very tabular, and 2) a set of HTML files which are more narrative and graphical. Finally, everything that has been accumulated & generated is compressed into a single zip file for downloading. This zip file is affectionately called a "study carrel". It is completely self-contained and includes all of the data necessary for more in-depth analysis.

What it does

The Distant Reader supplements the traditional reading process. It does this in the way of traditional reading apparatus (tables of content, back-of-book indexes, page numbers, etc), but it does it more specifically and at scale.

Put another way, the Distant Reader can answer a myriad of questions about individual items or the corpus as a whole. Such questions are not readily apparent through traditional reading. Examples include but are not limited to:

- How big is the corpus, and how does its size compare to other corpora?
- How difficult (scholarly) is the corpus?
- What words or phrases are used frequently and infrequently?
- What statistically significant words characterize the corpus?
- Are there latent themes in the corpus, and if so, then what are they and how do they change over both time and place?
- How do any latent themes compare to basic characteristics of each item in the corpus (author, genre, date, type, location, etc.)?
- What is discussed in the corpus (nouns)?
- What actions take place in the corpus (verbs)?
- How are those things and actions described (adjectives and adverbs)?
- What is the tone or "sentiment" of the corpus?
- How are the things represented by nouns, verbs, and adjective related?
- Who is mentioned in the corpus, how frequently, and where?
- What places are mentioned in the corpus, how frequently, and where?

People who use the Distant Reader look at the reports it generates, and they often say, "That's interesting!" This is because it highlights characteristics of the corpus which are not readily apparent. If you were asked what a particular corpus was about or what are the names of people mentioned in the corpus, then you might answer with a couple of sentences or a few names, but with the Distant Reader you would be able to be more thorough with your answer.

The questions outlined above are not necessarily apropos to every student, researcher, or scholar, but the answers to many of these questions will lead to other, more specific questions. Many of those questions can be answered directly or indirectly through further analysis of the structured data provided in the study carrel. For example, each & every feature of each & every sentence of each & every item in the corpus has been saved in a relational database file. By querying the database, the student can extract every sentence with a given word or matching a given grammer to answer a question such as "How was the king described before

& after the civil war?" or "How did this paper's influence change over time?"

A lot of natural language processing requires pre-processing, and the Distant Reader does this work automatically. For example, collections need to be created, and they need to be transformed into plain text. The text will then be evaluated in terms of parts-of-speech and named-entities. Analysis is then done on the results. This analysis may be as simple as the use of concordance or as complex as the application of machine learning. The Distant Reader "primes the pump" for this sort of work because all the raw data is already in the study carrel. The Distant Reader is not intended to be used alone. It is intended to be used in conjunction with other tools, everything from a plain text editor, to a spreadsheet, to database, to topic modelers, to classifiers, to visualization tools.

I don't know about you, but now-a-days I can find plenty of scholarly & authoritative content. My problem is not one of discovery but instead one of comprehension. How do I make sense of all the content I find? The Distant Reader is intended to address this question by making observations against a corpus and providing tools for interpreting the results.

Five different types of input

The Distant Reader can take five different types of input, and this section describes in detail what they are:

1. a file
2. a URL
3. a list of URLs
4. a zip file
5. a zip file with a companion CSV file

Each of these different types of input are elaborated upon below.

A file

The simplest form of input is a single file from your computer. This can be just about file available to you, but to make sense, the file needs to contain textual data. Thus, the file can be a Word document, a PDF file, an Excel spreadsheet, an HTML file, a plain text file, etc. A file in the form of an image will not work because it contains zero text. Also, not all PDF files are created equal. Some PDF files are only facsimiles of their originals. Such PDF files are merely sets of images concatenated together. In order for PDF files to be used as input, the PDF files need to have been "born digitally" or they need to have had optical character recognition previously applied against them. Most PDF files are born digitally nor do they suffer from being facsimiles.

A good set of use-cases for single file input is the whole of a book, a long report, or maybe a journal article. Submitting a single file to the Distant Reader is quick & easy, but the Reader is designed for analyzing larger rather than small corpora. Thus, supplying a single journal article to the Reader doesn't make much sense; the use of the traditional reading process probably makes more sense for a single journal article.

A URL

The Distant Reader can take a single URL as input. Given a URL, the Reader will turn into a rudimentary Internet spider and build a corpus. More specifically, given a URL, the Reader will:

1. retrieve & cache the content found at the other end of the URL
2. extract any URLs it finds in the content

3. retrieve & cache the content from these additional URLs
4. stop building the corpus but continue with its analysis

In short, given a URL, the Reader will cache the URL's content, crawl the URL one level deep, cache the result, and stop caching.

Like the single file approach, submitting a URL to the Distant Reader is quick & easy, but there are a number of caveats. First of all, the Reader does not come with very many permissions, and just because you are authorized to read the content at the other end of a URL does not mean the Reader has the same authorization. A lot of content on the Web resides behind paywalls and firewalls. The Reader can only cache 100% freely accessible content.

"Landing pages" and "splash pages" represent additional caveats. Many of the URLs passed around the 'Net do not point to the content itself, but instead they point to ill-structured pages describing the content — metadata pages. Such pages may include things like authors, titles, and dates, but these things are not presented in a consistent nor computer-readable fashion; they are laid out with aesthetics or graphic design in mind. These pages do contain pointers to the content you want to read, but the content may be two or three more clicks away. Be wary of URLs pointing to landing pages or splash pages.

Another caveat to this approach is the existence of extraneous input due to navigation. Many Web pages include links for navigating around the site. They also include links to things like "contact us" and "about this site". Again, the Reader is sort of stupid. If found, the Reader will crawl such links and include their content in the resulting corpus.

Despite these drawbacks there are number of excellent use-cases for single URL input. One of the best is Wikipedia articles. Feed the Reader a URL pointing to a Wikipedia article. The Reader will cache the article itself, and then extract all the URLs the article uses as citations. The Reader will then cache the content of the citations, and then stop caching.

Similarly, a URL pointing to an open access journal article will function just like the Wikipedia article, and this will be even more fruitful if the citations are in the form of freely accessible URLs. Better yet, consider pointing the Reader to the root of an open access journal issue. If the site is not overly full of navigation links, and if the URLs to the content itself are not buried, then the whole of the issue will be harvested and analyzed.

Another good use-case is the home page of some sort of institution or organization. Want to know about Apple Computer, the White House, a conference, or a particular department of a university? Feed the root URL of any of these things to the Reader, and you will learn something. At the very least, you will learn how the organization prioritizes its public face. If things are more transparent than not, then you might be able to glean the names and addresses of the people in the organization, the public policies of the organization, or the breadth & depth of the organization.

Yet another excellent use-case includes blogs. Blogs often contain content at their root. Navigations links abound, but more often than not the navigation links point to more content. If the blog is well-designed, then the Reader may be able to create a corpus from the whole thing, and you can "read" it in one go.

A list of URLs

The third type of input is a list of URLs. The list is expected to be manifested as a plain text file, and each line in the file is a URL. Use whatever application you desire to build the list, but save the result as a .txt file, and you will probably have a plain text file.

Caveats? Like the single URL approach, the list of URLs must point to freely available content, and pointing to landing pages or splash pages is probably to be avoided. Unlike the single URL approach, the URLs in the list will not be used as starting points for Web crawling. Thus, if the list contains ten items, then ten items will be cached for analysis.

Another caveat is the actual process of creating the list; I have learned that is actually quite difficult to create lists of URLs. Copying & pasting gets old quickly. Navigating a site and right-clicking on URLs is tedious. While search engines & indexes often provide some sort of output in list format, the lists are poorly structured and not readily amenable to URL extraction. On the other hand, there are more than a few URL extraction tools. I use a Google Chrome extension called Link Grabber. [1] Install Link Grabber. Use Chrome to visit a site. Click the Link Grabber button, and all the links in the document will be revealed. Copy the links and paste them into a document. Repeat until you get tired. Sort and peruse the list of links. Remove the ones you don't want. Save the result as a plain text file.‡ Feed the result to the Reader.

Despite these caveats, the list of URLs approach is enormously scalable; the list of URLs approach is the most scalable input option. Given a list of five or six items, the Reader will do quite well, but the Reader will operate just as well if the list contains dozens, hundreds, or even thousands of URLs. Imagine reading the complete works of your favorite author or the complete run of an electronic journal. Such is more than possible with the Distant Reader.

A zip file

The Distant Reader can take a zip file as input. Create a folder/directory on your computer. Copy just about any file into the folder/directory. Compress the file into a .zip file. Submit the result to the Reader.

Like the other approaches, there are a few caveats. First of all, the Reader is not able to accept .zip files whose size is greater than 64 megabytes. While we do it all the time, the World Wide Web was not really designed to push around files of any great size, and 64 megabytes is/was considered plenty. Besides, you will be surprised how many files can fit in a 64 megabyte file.

Second, the computer gods never intended file names to contain things other than simple Romanesque letters and a few rudimentary characters. Now-a-days our file names contain spaces, quote marks, apostrophes, question marks, back slashes, forward slashes, colons, commas, etc. Moreover, file names might be 64 characters long or longer! While every effort as been made to accomodate file names with such characters, your milage may vary. Instead, consider using file names which are shorter, simpler, and have some sort of structure. An example might be first word of author's last name, first meaningful word of title, year (optional), and extension. Herman Melville's Moby Dick might thus be named melville-moby.txt. In the end the Reader will be less confused, and you will be more able to find things on your computer.

There are a few advantages to the zip file approach. First, you can circumvent authorization restrictions; you can put licensed content into your zip files and it will be analyzed just like any other content. Second, the zip file approach affords you the opportunity to pre-process your data. For example, suppose you have downloaded a set of PDF files, and each page includes some sort of header or footer. You could transform each of these PDF files into plain text, use some sort of find/replace function to remove the headers & footers. Save the result, zip it up, and submit it to the Reader. The resulting analysis will be more accurate.

There are many use-cases for the zip file approach. Masters and Ph.D students are expected to read large amounts of material. Save all those things into a folder, zip them up, and feed them to the Reader. You have been given a set of slide decks from a conference. Zip them up and feed them to the Reader. A student is expected to read many different things for History 101. Download them all, put them in a folder, zip them up, and submit them to the Distant Reader. You have written many things but they are not on the Web. Copy

them to a folder, zip them up, and “read” them with the… Reader.

A zip file with a companion CSV file

The final form of input is a zip file with a companion comma-separated value (CSV) file — a metadata file.

As the size of your corpus increases, so does the need for context. This context can often be manifested as metadata (authors, titles, dates, subject, genre, formats, etc.). For example, you might want to compare & contrast who wrote what. You will probably want to observe themes over space & time. You might want to see how things differ between different types of documents. To do this sort of analysis you will need to know metadata regarding your corpus.

As outlined above, the Distant Reader first creates a cache of content — a corpus. This is the raw data. In order to do any analysis against the corpus, the corpus must be transformed into plain text. A program called Tika is used to do this work. [2] Not only does Tika transform just about any file into plain text, but it also does its best to extract metadata. Depending on many factors, this metadata may include names of authors, titles of documents, dates of creation, number of pages, MIME-type, language, etc. Unfortunately, more often than not, this metadata extraction process fails and the metadata is inaccurate, incomplete, or simply non-existent.

This is where the CSV file comes in; by including a CSV file named “metadata.csv” in the .zip file, the Distant Reader will be able to provide meaningful context. In turn, you will be able to make more informed observations, and thus your analysis will be more thorough. Here’s how:

1. assemble a set of files for analysis
2. use your favorite spreadsheet or database application to create a list of the file names
3. assign a header to the list (column) and call it “file”
4. create one or more columns whose headers are “author” and/or “title” and/or “date”
5. to the best of your ability, update the list with author, title, or date values for each file
6. save the result as a CSV file named “metadata.csv” and put it in the folder/directory to be zipped
7. compress the folder/directory to create the zip file
8. submit the result to the Distant Reader for analysis

The zip file with a companion CSV file has all the strengths & weakness of the plain o’ zip file, but it adds some more. On the weakness side, creating a CSV file can be both tedious and daunting. On the other hand, many search engines & index export lists with author, title, and data metadata. One can use these lists as the starting point for the CSV file.✢ On the strength side, the addition of the CSV metadata file makes the Distant Reader’s output immeasurably more useful, and it leads the way to additional compare & contrast opportunities.

Project Gutenberg and the Distant Reader

The venerable Project Gutenberg is perfect fodder for the Distant Reader, and this section outlines how & why.

A long time ago, in a galaxy far far away, there was a man named Micheal Hart. Story has it he went to college at the University of Illinois, Urbana-Champagne. He was there during a summer, and the weather was seasonably warm. On the other hand, the computer lab was cool. After all, computers run hot, and air conditioning is a must. To cool off, Micheal went into the computer lab to be in a cool space.† While he was there he decided to transcribe the United States Declaration of Independence, ultimately, in the hopes of

enabling people to use a computers to "read" this and additional transcriptions. That was in 1971. One thing led to another, and Project Gutenberg was born. I learned this story while attending a presentation by the now late Mr. Hart on Saturday, February 27, 2010 in Roanoke (Indiana). As it happened it was also Mr. Hart's birthday. [1]

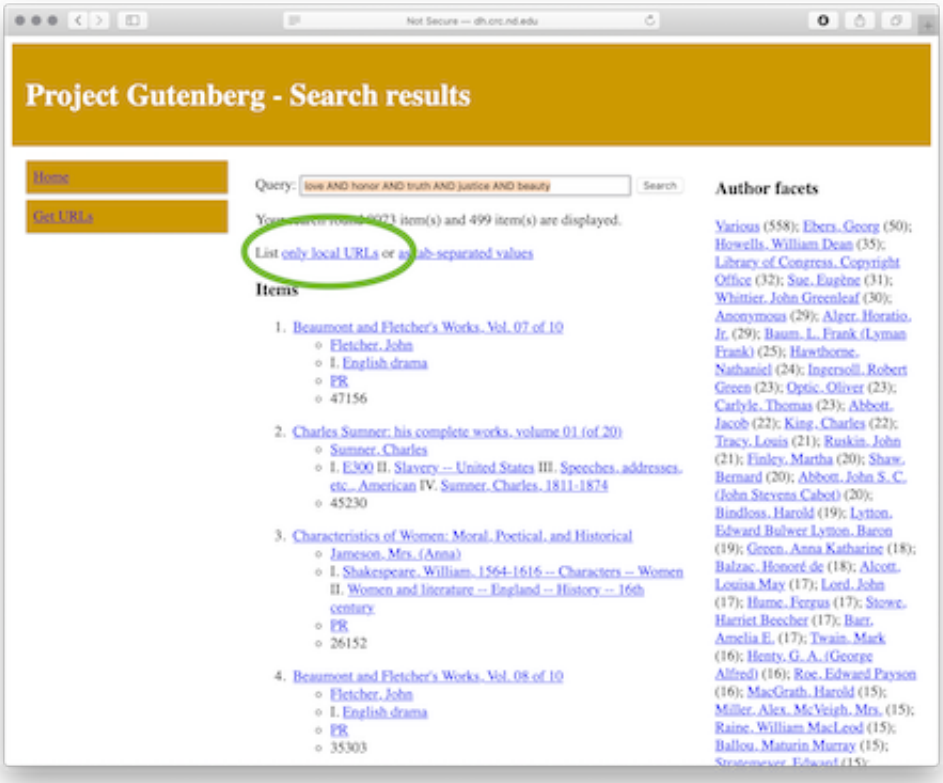
To date, Project Gutenberg is a corpus of more than 60,000 freely available transcribed ebooks. The texts are predominantly in English, but many languages are represented. Many academics look down on Project Gutenberg, probably because it is not as scholarly as they desire, or maybe because the provenance of the materials is in dispute. Despited these things, Project Gutenberg is a wonderful resource, especially for high school students, college students, or life-long learners. Moreover, their transcribed nature eliminates any problems of optical character recognition, such as one encounters with the HathiTrust. The content of Project Gutenberg is all but perfectly formatted for distant reading.

Unfortunately, the interface to Project Gutenberg is less than desirable; the index to Project Gutenberg is limited to author, title, and "category" values. The interface does not support free text searching, and there is limited support for fielded searching and Boolean logic. Similarly, the search results are not very interactive nor faceted. Nor is there any application programmer interface to the index. With so much "clean" data, so much more could be implemented. In order to de\monstrate the power of distant reading, I endeavored to create a mirror of Project Gutenberg while enhancing the user interface.

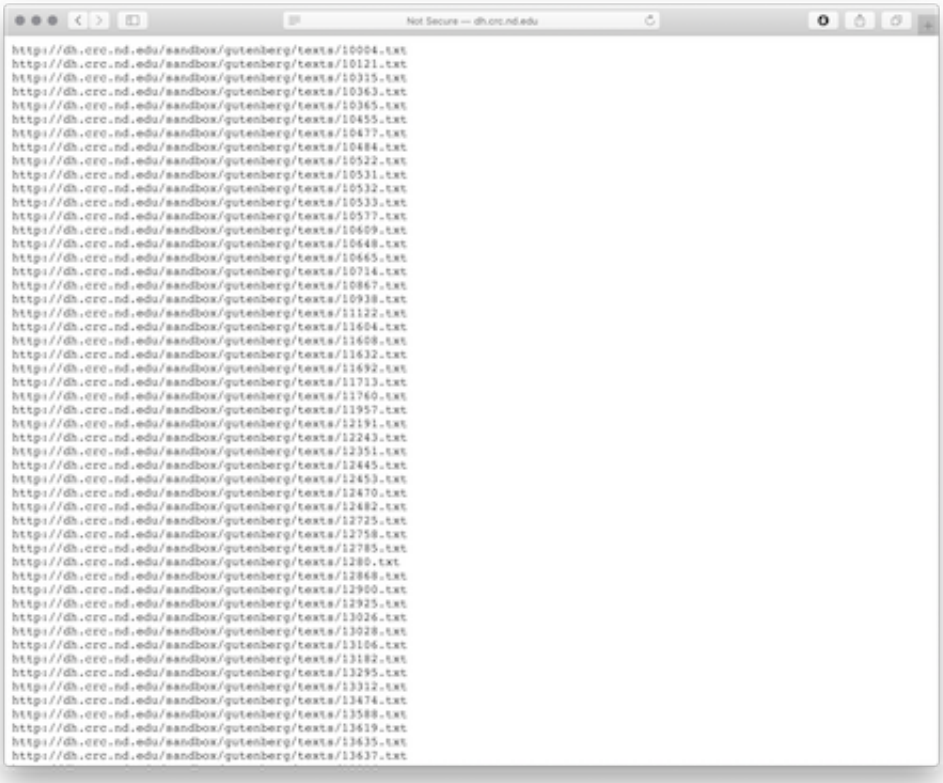
To create a mirror of Project Gutenberg, I first downloaded a set of RDF files describing the collection. [2] I then wrote a suite of software which parses the RDF, updates a database of desired content, loops through the database, caches the content locally, indexes it, and provides a search interface to the index. [3, 4] The resulting interface is ill-documented but 100% functional. It supports free text searching, phrase searching, fielded searching (author, title, subject, classification code, language) and Boolean logic (using AND, OR, or NOT). Search results are faceted enabling the reader to refine their query sans a complicated query syntax. Because the cached content includes only English language materials, the index is only 33,000 items in size.

The Distant Reader is a tool for reading. It takes an arbitrary amount of unstructured data (text) as input, and it outputs sets of structured data for analysis — reading. Given a corpus of any size, the Distant Reader will analyze the corpus, and it will output a myriad of reports enabling you to use & understand the corpus. The Distant Reader is intended to supplement the traditional reading process. Project Gutenberg and the Distant Reader can be used hand-in-hand.

As described in a previous posting, the Distant Reader can take five different types of input. [5] One of those inputs is a file where each line in the file is a URL. My locally implemented mirror of Project Gutenberg enables the reader to search & browse in a manner similar to the canonical version of Project Gutenberg, but with two exceptions. First & foremost, once a search has been gone against my mirror, one of the resulting links is "only local URLs". For example, below is an illustration of the query "love AND honor AND truth AND justice AND beauty", and the "only local URLs" link is highlighted:



By selecting the "only local URLs", a list of... URLs is returned, like this:



This list of URLs can then be saved as file, and any number of things can be done with the file. For example, there are Google Chrome extensions for the purposes of mass downloading. The file of URLs can be fed to

command-line utilities (ie. curl or wget) also for the purposes of mass downloading. In fact, assuming the file of URLs is named love.txt, the following command will download the files in parallel and really fast:

```
cat love.txt | parallel wget
```

This same file of URLs can be used as input against the Distant Reader, and the result will be a "study carrel" where the whole corpus could be analyzed -- read. For example, the Reader will extract all the nouns, verbs, and adjectives from the corpus. Thus you will be able to answer what and how questions. It will pull out named entities and enable you to answer who and where questions. The Reader will extract keywords and themes from the corpus, thus outlining the aboutness of your corpus. From the results of the Reader you will be set up for concordancing and machine learning (such as topic modeling or classification) thus enabling you to search for more narrow topics or "find more like this one". The search for love, etc returned more than 8000 items. Just less than 500 of them were returned in the search result, and the Reader empowers you to read all 500 of them at one go.

Project Gutenberg is very useful resource because the content is: 1) free, and 2) transcribed. Mirroring Project Gutenberg is not difficult, and by doing so an interface to it can be enhanced. Project Gutenberg items are perfect items for reading & analysis by the Distant Reader. Search Project Gutenberg, save the results as a file, feed the file to the Reader and... read the results at scale.

Notes and links

† All puns are intended.

[1] Michael Hart in Roanoke (Indiana) - video: https://youtu.be/eeoBbSN9Esg; blog posting: http://infomotions.com/blog/2010/03/michael-hart-in-roanoke-indiana/

[2] The various Project Gutenberg feeds, including the RDF is located at https://www.gutenberg.org/wiki/Gutenberg:Feeds

[3] The suite of software to cache and index Project Gutenberg is available on GitHub at https://github.com/ericleasemorgan/gutenberg-index

[4] My full text index to the English language texts in Project Gutenberg is available at http://dh.crc.nd.edu/sandbox/gutenberg/cgi-bin/search.cgi

[5] The Distant Reader and its five different types of input - http://sites.nd.edu/emorgan/2019/10/dr-inputs/

Summary

To date, the Distant Reader takes five different types of input. Each type has its own set of strengths & weaknesses:

- a file – good for a single large file; quick & easy; not scalable
- a URL – good for getting an overview of a single Web page and its immediate children; can include a lot of noise; has authorization limitations
- a list of URLs – can accomodate thousands of items; has authorization limitations; somewhat difficult to create list
- a zip file – easy to create; file names may get in the way; no authorization necessary; limited to 128 megabytes in size
- a zip file with CSV file – same as above; difficult to create metadata; results in much more meaningful

reports & opportunities

Submitting "experiments" and downloading "study carrels"

An introduction to study carrels

The structured data of study carrels; taking inventory through the manifest

The results of the Distant Reader process is the creation of a “study carrel” — a set of structured data files intended to help you to further “read” your corpus. Using a previously created study carrel as an example, this blog posting enumerates & outlines the contents of a typical carrel. A future blog posting will describe ways to use & understand the files outlined here. Therefore, the text below is merely a kind of manifest.

The Distant Reader takes an arbitrary amount of unstructured data (text) as input, and it outputs sets of structured data files for analysis — reading. Given a corpus of any size, the Distant Reader will analyze the corpus, and it will output a myriad of reports enabling you to use & understand the corpus. The Distant Reader is intended to supplement the traditional reading process. Given a question of a rather quantitative nature, a Distant Reader study carrel may very well contain a plausible answer.

The results of downloading and uncompressing the Distant Reader study carrel is a directory/folder containing a standard set of files and subdirectories. Each of these files and subdirectories are listed & described below:

- **A1426341535** – This, or a very similarly named file, is an administrative file, a unique identifier created by the system (Airivata, https://airavata.apache.org) which managed the creation of the study carrel. In the future, this file may not be included. On the other hand, since the file’s name is a unique identifier, then it could be exploited by a developer.
- adr – This subdirectory contains a set of tab-delimited files. Each file contains a set of email addresses extracted from the documents in your corpus. While the files’ names end in .adr, they are plain text files that can be imported into for favorite spreadsheet, database, or analysis application. The files have two columns: 1) id, and 2) address. The definitions of these columns and possible uses of these files are described elsewhere, but in short, these files can humorously answer the question “Who are you gonna call?”
- bib – This subdirectory contains a set of tab-delimited files. Each file contains a set of rudimentary bibliographic information from a given document in your corpus. While the files’ names end in .bib, they are plain text files that can be imported into for favorite spreadsheet, database, or analysis application. The files have thirteen columns: 1) id, 2) author, 3) title, 4) date, 5) page 6), extension, 7) mime, 8) words, 9) sentences, 10) flesch, 11) summary, 12) cache, and 13) txt. The definitions of these columns and possible uses of these files are described elsewhere, but in short, these files help answer the question “What items are in my corpus, and how can they be described?”
- cache – This subdirectory contains original copies of the files you intended for analysis. It is populated by harvesting content from URLs or were supplied in the zip file you uploaded to the Reader. Each file is named with a unique and somewhat meaningful name and an extension. These files are intended for reading on your computer, or better yet, printed and then read in the more traditional manner.
- css – This subdirectory contains a set of cascading stylesheets used by the HTML files in the carrel. If you really desired, one could edit these files in order to change the appearance of the carrel.
- **input.zip** – This file, or something named very similarly, is the file originally used to create your study

carrel. It has already served its intended purpose, but it is retained for reasons of provenance.

- ent – This subdirectory contains a set of tab-delimited files, and each file contains a set of named entities from a given document in your corpus. While the files' names end in .ent, they are plain text files that can be imported into for favorite spreadsheet, database, or analysis application. The files have five columns: 1) id, 2) sid, 3) eid, 4) entity, and 5) type. The definitions of these columns and possible uses of these files are described elsewhere, but in short, these files help answer questions regarding who, what, when, where, how, and how many.
- etc – This subdirectory contains a set of ancillary files, and each are described below
    - model-data.txt – the data file used by topic-model.htm, and it is essentially an enhanced version of reader.txt
    - queries.sql – a set of SQL queries used to generate report.txt, and this file is an excellent introduction to the use of reader.db
    - reader.db – an SQLite database file, and it is essentially the amalgamation of the contents of the adr, bib, ent, pos, urls, and wrd directories; the intelligent use of this file can be used to answer just about any question answerable by the carrel
    - reader.sql – a set SQL commands denoting the structure of reader.db
    - reader.txt – the concatenation of all files in the txt directory; a plain text version of the whole of the corpus is often used for other purposes and it is provided here as a convienence
    - report.txt – the result of applying queries.sql to reader.db; this file has the exact same content as standard-output.txt
    - stopwords.txt – a list of function words (i.e. "a", "an", "the", etc.) used through the creation of the study carrel
- figures – This subdirectory contains a set of image files used by the carrel's HTML files:
    - adjectives.png – a word cloud illustrating the most frequent adjectives in the corpus
    - adverbs.png – a word cloud illustrating the most frequent adverbs in the corpus
    - bigrams.png – a word cloud illustrating the most frequent bigrams (two-word phrases) in the corpus
    - flesch-boxplot.png – a box plot illustrating the average, quartile, and outlier readability scores of the items in the corpus
    - flesch-histogram.png – a histogram illustrating the distribution of readability scores of the items in the corpus
    - keywords.png – a word cloud illustrating the most frequent keywords (statistically significant unigrams) in the corpus
    - nouns.png – a word cloud illustrating the most frequent nouns in the corpus
    - pronouns.png – a word cloud illustrating the most frequent pronouns in the corpus
    - proper-nouns.png – a word cloud illustrating the most frequent proper nouns in the corpus
    - sizes-boxplot.png – a box plot illustrating the average, quartile, and outlier sizes of the items (measured in unigrams) in the corpus
    - sizes-histogram.png – a histogram illustrating the distribution of sizes of the items (measured in unigrams) in the corpus
    - topics.png – a pie chart illustrating how the corpus is subdivided if topic modeling were applied to the corpus, and the desired number of topics (latent themes) equals five
    - unigrams.png – a word cloud illustrating the most frequent unigrams (individual words) in the corpus
    - verbs.png – a word cloud illustrating the most frequent verbs in the corpus
- **htm** – This subdirectory contains a set of interactive HTML files linked from the file named index.htm. The functionality of each file is outlined below:
    - **adjective-noun.htm** – search, sort, and browse adjective/noun combinations by adjective, noun, or frequency
    - **adjectives.htm** – search, sort, and browse adjectives and/or their frequency

    - **adverbs.htm** – search, sort, and browse adverbs and/or their frequency
    - **bigrams.htm** – search, sort, and browse bigrams (two-word phrases) and/or their frequency
    - **entities.htm** – search, sort, and browse named-entities, their type, and/or their frequency
    - **keywords.htm** – search, sort, and browse keywords (statistically significant unigrams) and/or their frequency
    - **noun-verb.htm** – search, sort, and browse noun/verb combinations by noun, verb, or frequency
    - **nouns.htm** – search, sort, and browse nouns and/or their frequency
    - **pronouns.htm** – search, sort, and browse pronouns and/or their frequency
    - **proper-nouns.htm** – search, sort, and browse proper nouns and/or their frequency
    - **quadgrams.htm** – search, sort, and browse quadgrams (four-word phrases) and/or their frequency
    - **questions.htm** – search, sort, and browse questions (sentences ending with a question mark) and from which items they were extracted
    - **search.htm** – a free text query interface based on the narrative summaries of each item in the corpus
    - **topic-model.htm** – a topic modeler; a tool used to enumerate as well as compare & contrast latent themes in the corpus
    - **trigrams.htm** – search, sort, and browse trigrams (three-word phrases) and/or their frequency
    - **unigrams.htm** – search, sort, and browse unigrams (individual words) and/or their frequency
    - **verbs.htm** – search, sort, and browse verbs and/or their frequencies
- **index.htm** – This HTML file narratively reports on the content of your study carrel. It is the best place to begin once you have downloaded and unzipped the carrel.
- **MANIFEST.htm** – This file, and it is the third best place to begin once you have downloaded and unzipped a carrel.
- **job_1819387465.slurm** – This file, or a very similarly named file, is the batch file used to initially create your study carrel. In the future, this file may be removed from the study carrel all together because it serves only an administrative purpose.
- **js** – This subdirectory includes a set of Javascript libraries supporting the functionality of index.htm as well as the HTML files in the htm directory. Because these files are here your computer does not need to be connected to the Internet in order to effectively read your carrel. Study carrels are designed to be stand-alone file systems usable for years to come.
- **LICENSE** – This is the license file; each study carrel is distributed under a GNU Public License.
- **pos** – This subdirectory contains a set of tab-delimited files, and each file contains a set of part-of-speech files from a given document in your corpus. While the files' names end in .pos, they are plain text files that can be imported into for favorite spreadsheet, database, or analysis application. The files have six columns: 1) id, 2) sid, 3) tid, 4) token, 5) lemma, and 6) pos. The definitions of these columns are described in another blog posting. The definitions of these columns and possible uses of these files are described elsewhere, but in short, these files help answer question regarding who, what, how, how many, and actions as well as grammer and style.
- **README** – This file contains the very briefest of introductions to the carrel.
- **standard-error.txt** – As each study carrel is being created, error and status messages are output to this file. It is a log file. If the creation of your study carrel fails, then this is a good place to look for clues on what went wrong. Send me this file if you are stymied.
- **standard-output.txt** – After your study carrel as been created and distilled into a database, sets of queries are applied against the database. This file is the second best place to begin once you have downloaded and unzipped a carrel.
- **tsv** – Except for one (questions.tsv), this subdirectory contains a set of frequency tables in the form of tab-delimited text files. The exception is a tab-delimited text file too, but it is just not a frequency file. All of these files can be imported into for favorite spreadsheet, database, or analysis application. Possible uses for these files are destined to be outlined in future postings, but in short, perusal of these

files will help you answer questions regarding your corpus's "aboutness" as well as who, what, when, where, how, how many, and why questions. The structure of each file is listed below:
    - **adjective-noun.tsv** – three columns: 1) adjective, 2) noun, and 3) frequency where frequency denotes the number of times the given adjective appears immediately before the given noun in the corpus
    - **adjectives.tsv** – two columns: 1) adjective, and 2) frequency
    - **adverbs.tsv** – two columns: 1) adverb, and 2) frequency
    - **bigrams.tsv** – two columns: 1) bigram (two-word phrase), and 2) frequency
    - **entities.tsv** – three columns: 1) entity, 2) type, and 3) frequency
    - **keywords.tsv** – two columns: 1) keyword (statistically significant unigram), and 2) frequency
    - **noun-verb.tsv** – three columns: 1) noun, 2) verb, and 3) a frequency where frequency denotes the number of times the given noun appears immediately before the given verb in the entire corpus
    - **nouns.tsv** – two columns: 1) noun, and 2) frequency
    - **pronouns.tsv** – two columns: 1) pronoun, and 2) frequency
    - **proper-nouns.tsv** – two columns: 1) proper, and 2) frequency
    - **quadgrams.tsv** – two columns: 1) quadgram (four-word phrase), and 2) frequency
    - **questions.ts** – two columns: 1) identifier, and 2) question where each question is a "sentence" ending in a question mark
    - **trigrams.tsv** – two columns: 1) trigram (three-word phrase), and 2) frequency
    - **unigrams.tsv** – two columns: 1) unigram (individual word), and 2) frequency
    - **verbs.tsv** – two columns: 1) verb, and 2) frequency
- **txt** – This subdirectory contains plain text versions of the files stored in the cache directory. A plain text version of each & every item in the cache directory ought to exist in this directory. The contents of this directory is what was used to do the Reader's analysis. The contents of this directory are excellent candidates for further analysis with tools such as concordances, indexers, or topic modelers.
- **urls** – This subdirectory contains a set of tab-delimited files, and each file contains a set of URLs from a given document in your corpus. While the files' names end in .url, they are plain text files that can be imported into for favorite spreadsheet, database, or analysis application. The files have three columns: 1) id, 2) domain, and 3) url. The definitions of these columns and possible uses of these files are described elsewhere, but in short, these files help answer questions regarding document provenance and relationships as well as addressing the perenial issue of "finding more like this one".
- **wrd** – This subdirectory contains a set of tab-delimited files, and each file contains a set of computed keywords from a given document in your corpus. While the files' names end in .wrd, they are plain text files that can be imported into for favorite spreadsheet, database, or analysis application. The files have two columns: 1) id, and 2 keyword. The definitions of these columns and possible uses of these files are described elsewhere, but in short, these files help answer questions such as "What is this document about?"

Links

[1] Airivata – https://airavata.apache.org

Using combinations of desktop tools to analyze the data

This section first describes the types of desktop tools (computer programs) the student, researcher, or scholar will need in order to use a Distant Reader study carrel. This section then describes how some of the more specific tools can be used for the purpose of use & understanding.

Three essential types of desktop tools

There are three essential types of desktop tools you will need/want in order to use the content of a study carrel. These types include: text editors, spreadsheet/database applications, and analysis programs.

Text editors

Text editors read and write plain text files -- files with no formatting and no binary characters. Plain text files usually have a .txt extension. Every single file in a Distant Reader study carrel (except one) is a plain text file, and therefore, every single file (except one) is openable by any text editor.

There are a few essencial tools you will want in a text editor, and the most important is a find/replace function. The function ought to allow you to find any character and change it to something else. This is useful for removing stopwords from a file. It is useful for removing carriage returns or newline characters, thus unwrapping your text. The text editor gets bonus points if the find/replace function supports "regular expressions". The second most important function of a text editor is a sorting feature. Each line in many text files is really an item in a list, and you will invariably want to sort the list. Another very useful function of a text editor, especially used for the purposes of text mining and natural language processing, is the ability to change the case of all letters to either their upper or lower-case forms. Such is the most basic of text normalization/cleaning processes. Religious wars are fought over text editors, and for the purposes of this workbook, only a number are listed, and not all of them support all the functions outlined above: Notepad, Wordpad, Text Edit, Notepad++, Atom, and BBedit. The student, researcher, or scholar will want/need a plain text editor in order to truly exploit the use of the Distant Reader.

Spreadsheet/database applications

Spreadsheet/database applications are designed to read "delimited" files, plain text files where each line is a row in a matrix, and each item is punctuated by some special character such as a tab character or a comma. These items are the columns in the matrix. The whole file is a kin to a spreadsheet or a database. Like a text editor, you will want to use the spreadsheet/database application to support sort. The spreadsheet/database application will need to be able to do arithmetic against items in the file. The spreadsheet/database application ought to include charting features. And the spreadsheet/database application ought to be able to save/export its data to other types of delimited files: tab-delimited files, CSV (comma-separated value) files, Excel workbook files, HTML tables, etc.

The majority of the files in a study carrel are delimited files, and these delimited files are really annotated lists. Examples include lists of word and their parts-of-speech, lists of documents and the URLs they contain, or lists of sentences and the named-entities they include. Given these files the student, researcher, or scholar can compare & contrast the ratio of named entities across a corpus, or they could plot the ebb & flow of an idea over time.

Analysis programs

Analysis programs cover a wide spectrum of tools, and for the purposes of the workbook, these tools fall into a number of categories: counting & tabulating, concordancing, topic modeling, and visualizing. For the purpose of this workbook OpenRefine will be used to support counting & tabulating, as well as a few other things. Concordancing is really about find, and a program called AntConc is described. Topic modeling is the process of extracting latent themes from a body of text, and a GUI application called Topic Modeling Tool is demonstrated. Two visualization tools are useful: Wordle and Gephi. The former outputs tag clouds, and the

later outputs network diagrams. The student, researcher, or scholar is expected to supplement visualization with the charting functions of spreadsheet/database applications.

Again, every single file (except one) making up a study carrel is a plain text file, and all those files are readable by a text editor. The majority of the files in a study carrel are delimited files, specifically, tab-delimited files, and they can be opened with a spreadsheet/database application. The student, researcher, or scholar will need to have these sort of programs at their disposal. The other, more specific application used in this workbook, are freely available for downloading as well as cross-platform in nature.

Wordle

Visualized word frequencies, while often considered sophomoric, can be quite useful when it comes to understanding a text, especially when the frequencies are focused on things like parts-of-speech, named entities, or co-occurrences. Wordle visualizes such frequencies very well. For example, the 100 most frequent words in the Iliad and the Odyssey, the 100 most frequent nouns in the Iliad and the Odyssey, or the statistically significant words associated with the word ship from the Iliad and the Odyssey.

Wordle recipes

Here is a generic Wordle recipe where World will calculate frequencies:

1. Download and install Wordle. It is a Java application, so you may need to download and install Java along the way, but Java is probably already installed on your computer.
2. Use your text editor to open reader.txt which is located in the etc directory/folder. Once opened, copy all of the text.
3. Open Wordle, select the "Your Text" tab, and paste the whole of the text file into the window.
4. Click the "Wordle" tab and your word cloud will be generated. Use the Wordle's menu options to customize the output.

Congratulations, you have just visualized the whole of your study carrel.

Here is another recipe, a recipe where you supply the frequencies (or any other score):

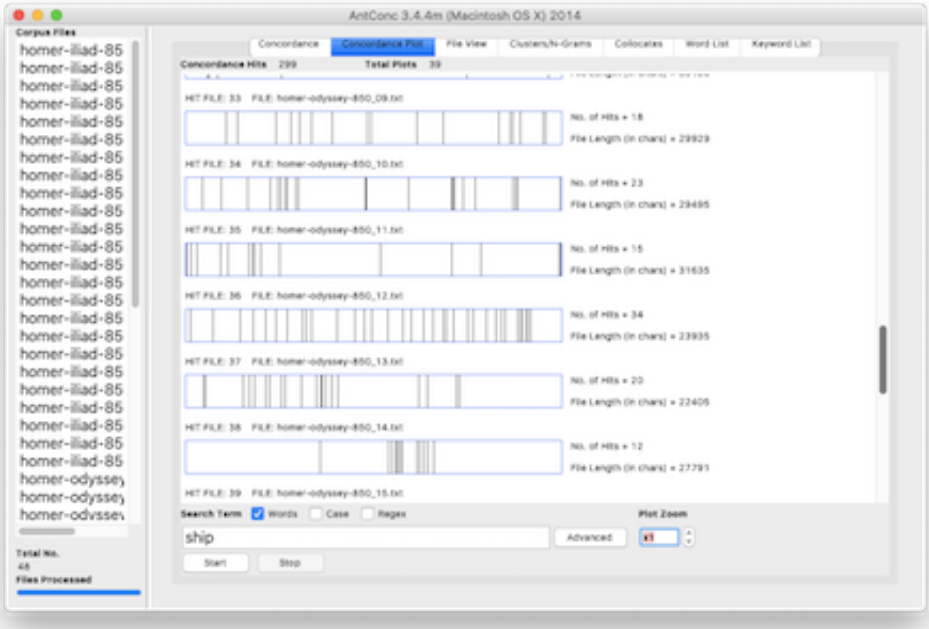
1. Download and install AntConc.
2. Use the "Open Files(s)..." menu option to open any file in the txt directory.
3. Click the "Word list" tab, and then click the "Start" button. The result will be a list of words and their frequencies.
4. Use the "Save Output to Text File..." menu option, and save the frequencies accordingly.
5. Open the resulting file in your spreadsheet.
6. Remove any blank rows, and remove the columns that are not the words and their frequencies
7. Invert the order of the remaining two columns; make the words the first column and the frequencies the second column.
8. Copy the whole of the spreadsheet and paste it into your text editor.
9. Use the text editor's find/replace function to find all occurrences of the tab character and replace them with the colon (:) character. Copy the whole of the text editor's contents.
10. Open Wordle, click the "Your text" tab, paste the frequencies into the resulting window.
11. Finally, click the "Wordle" tab to generate the word cloud.

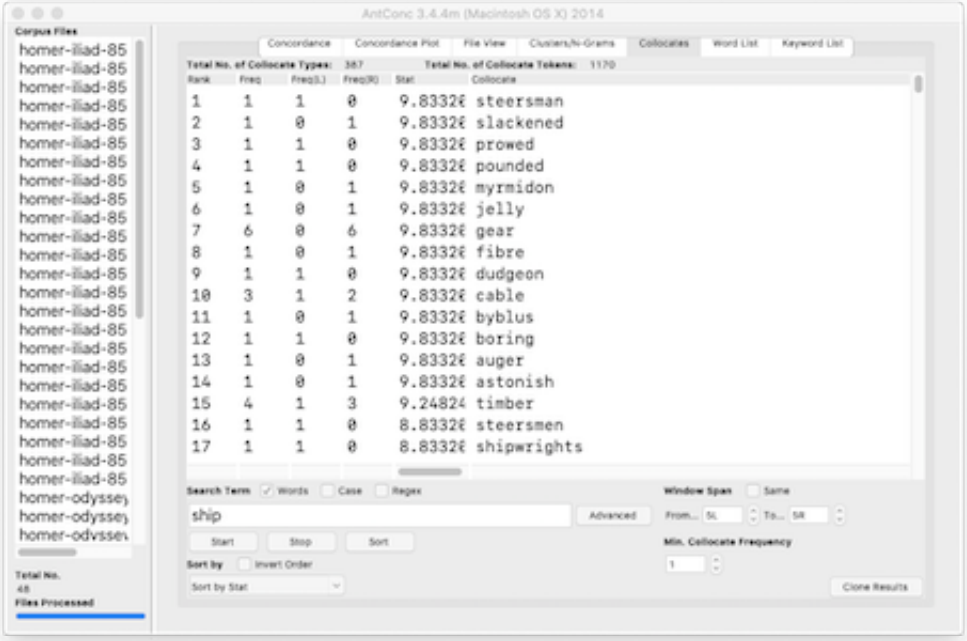
Notice how you used a variety of generic applications to achieve the desired result. The word/value pairs given to Wordle do not have be frequencies. Instead they can be any number of different scores or weights. Keep your eyes open for word/value combinations. They are everywhere. Word clouds have been given a bad rap. Wordle is a very useful tool.

Concordancing with AntConc

Concordancing is really a process about find, and AntConc is a very useful program for this purpose. Given one or more plain text files, AntConc will enable the student, researcher, or scholar to: find all the occurrences of a word, illustrate where the word is located, navigate through document(s) where the word occurs, list word collocations, and calculate quite a number of useful statistics regarding a word. Concordancing, dating from the 13th Century, is the oldest form of text mining. Think of it as control-F (^f) on steroids. AntConc does all this and more. For example, one can load all of the Iliad and the Odyssey into AntConc. Find all the occurrences of the word ship, visualize where ship appears in each chapter, and list the most significant words associated with the word ship.

AntConc 3.4.4m (Macintosh OS X) 2014
Corpus Files
homer-iliad-85
homer-odyssey
homer-odyssey
homer-odyssey
Total No.
48
Files Processed
Concordance
Concordance Plot
File View
Clusters/N-Grams
Collocates
Word List
Keyword List
Concordance Hits 299
Hit
KWIC
File
1 r the present, let us draw a ship into the sea, and find a homer-il
2 me, I shall send her with my ship and my followers, but I homer-il
3 ut of all else that is at my ship you shall carry away not homer-il
4 are now taking the girl in a ship to Chryse, and sending g homer-il
5 e sails and laid them in the ship 's hold; they slackened homer-il
6 nto its place, and rowed the ship to the place where they homer-il
7 ollo; Chryseis also left the ship, and Ulysses led her to homer-il
8 p by the stern cables of the ship, and when the child of m homer-il
9 il bellied with the wind the ship flew through the deep bl homer-il
10 meeting of the elders at the ship of Nestor king of Pylos, homer-il
11 as yet laid a hand upon his ship, for he was grieved and homer-il
12 let him lay his hand to his ship that he may meet his doo homer-il
13 ay hereafter as he sails his ship over the sea,' This is t homer-il
14 at the stern of Agamemnon 's ship, and took his place in t homer-il
15 huge black hull of Ulysses ' ship, which was middlemost of homer-il
16 oming hither, I never let my ship pass your altars without homer-il
17 s he is leaping on board his ship, that others may fear to homer-il
Search Term
Words
Case
Regex
Search Window Size
ship
Advanced
50
Start
Stop
Sort
Kwic Sort
Level 1
Level 2
Level 3
Clone Results


AntConc 3.4.4m (Macintosh OS X) 2014
Corpus Files
homer-iliad-85
homer-odyssey
homer-odyssey
homer-odyssey
Total No.
48
Files Processed
Concordance
Concordance Plot
File View
Clusters/N-Grams
Collocates
Word List
Keyword List
Concordance Hits 299
Total Plots 39
HIT FILE: 33 FILE: homer-odyssey-850_09.txt
No. of Hits = 18
File Length (in chars) = 29929
HIT FILE: 34 FILE: homer-odyssey-850_10.txt
No. of Hits = 23
File Length (in chars) = 29495
HIT FILE: 35 FILE: homer-odyssey-850_11.txt
No. of Hits = 15
File Length (in chars) = 31635
HIT FILE: 36 FILE: homer-odyssey-850_12.txt
No. of Hits = 34
File Length (in chars) = 23935
HIT FILE: 37 FILE: homer-odyssey-850_13.txt
No. of Hits = 20
File Length (in chars) = 22405
HIT FILE: 38 FILE: homer-odyssey-850_14.txt
No. of Hits = 12
File Length (in chars) = 27791
HIT FILE: 39 FILE: homer-odyssey-850_15.txt
Search Term
Words
Case
Regex
Plot Zoom
ship
Advanced
x1
Start
Stop



AntConc recipes

This recipe simply implements search:

1. Download and install AntConc
2. Use the "Open Files(s)..." menu option to open all files in the txt directory
3. Select the Concordance tab
4. Enter a word of interest into the search box
5. Click the Start button

The result ought to be a list of phrases where the word of interest is displayed in the middle of the screen. In modern-day terms, such a list is called a "key word in context" (KWIC) index.

This recipe combines search with "control-F":

1. Select the Concordance tab
2. Enter a word of interest into the search box
3. Click the Start button
4. Peruse the resulting phrases and click on one of interest; the result ought to a display of a text and the search term(s) is highlighted in the larger context
5. Go to Step #1 until tired

This recipe produces a dispersion plot, an illustration of where a search term appears in a document:

1. Select the Concordance tab
2. Enter a word of interest into the search box
3. Select the "Concordance Plot" tab

The result will be a list of illustrations. Each illustration will include zero or more vertical lines denoting the location of your search term in a given file. The more lines in each illustrations, the more times the search terms appear in the document.

This recipe counts & tabulates the frequency of words:

1. Select the "Word List" tab
2. Click the Start button; the result will be a list of all the words and their frequencies
3. Scroll up and down the list to get a feel for what is common
4. Select a word of interest; the result will be the same as if you entered the word in Recipe #1

It is quite probable the most frequent words will be "stop words" like the, a, an, etc. AntConc supports the elimination of stop words, and the Reader supplies a stop word list. Describing how to implement this functionality is too difficult to put into words. (No puns intended.) But here is an outline:

5. Select the "Tool Preferences" menu option
6. Select the "Word List" category
7. Use the resulting dialog box to select a stop words list, and such a list is called stopwords.txt found in the etc directory
8. Click the Apply button
9. Go to Step #1; and the result will be a frequency list sans any stop words, and the result will be much more meaningful

Ideas are rarely articulated through the use of individual words; ideas are usually articulated through the use of sets of words (ngrams, sentences, paragraphs, etc.). Thus, as John Rupert Firth once said, "You shall know a word by the company it keeps." This recipe outlines how to list word co-occurrences and collocations:

1. Select the "Cluster/N-grams" tab
2. Enter a word of interest in the search box
3. Click the Start button; the result ought to be a list of two-word phrases (bigrams) sort in frequency order
4. Select a phrase of interest, and the result will just as if you had search for the phrase in Recipe #1
5. Go to Step #1 until tired
6. Select the Collocates tab
7. Enter a word of interest in the search box
8. Click the Start button; the result ought to be a list of words and associated scores, and the scores compare the frequencies of the search word and the given word; words with higher scores can be considered "more interesting"
9. Select "Sort by Freq" from the "Sort by" pop-up menu
10. Click the Sort button; the result will be the same list of words and associated scores, but this time the list will be sorted by the frequency of the search term/given word combination

Again, a word is known by the company it keeps. Use the co-occurrences and collocations features to learn how a given word (or phrase) is associated with other words.

There is much more to AntConc than outlined in the recipes outlined above. Learning more is left up to you, the studen, research, and scholar.

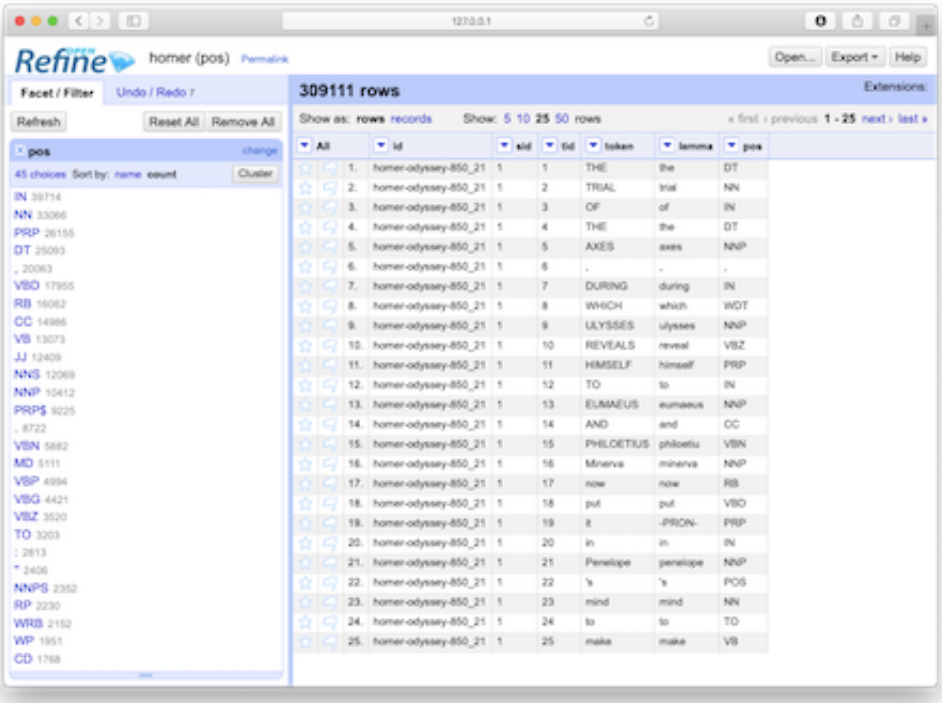
Excel - charting tabular data

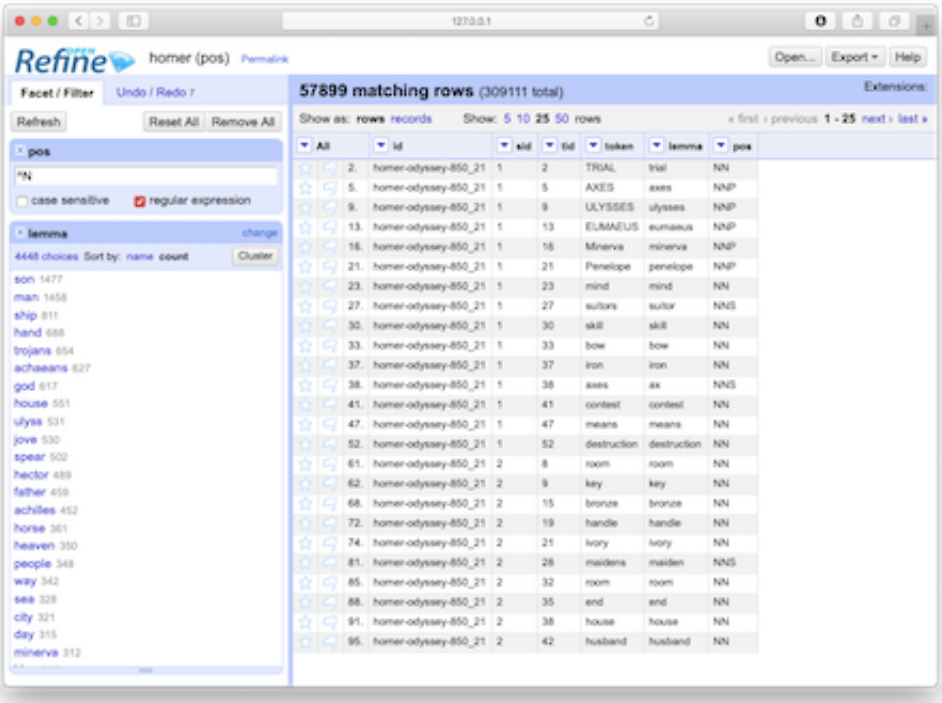
OpenRefine - fielded searching and grouping

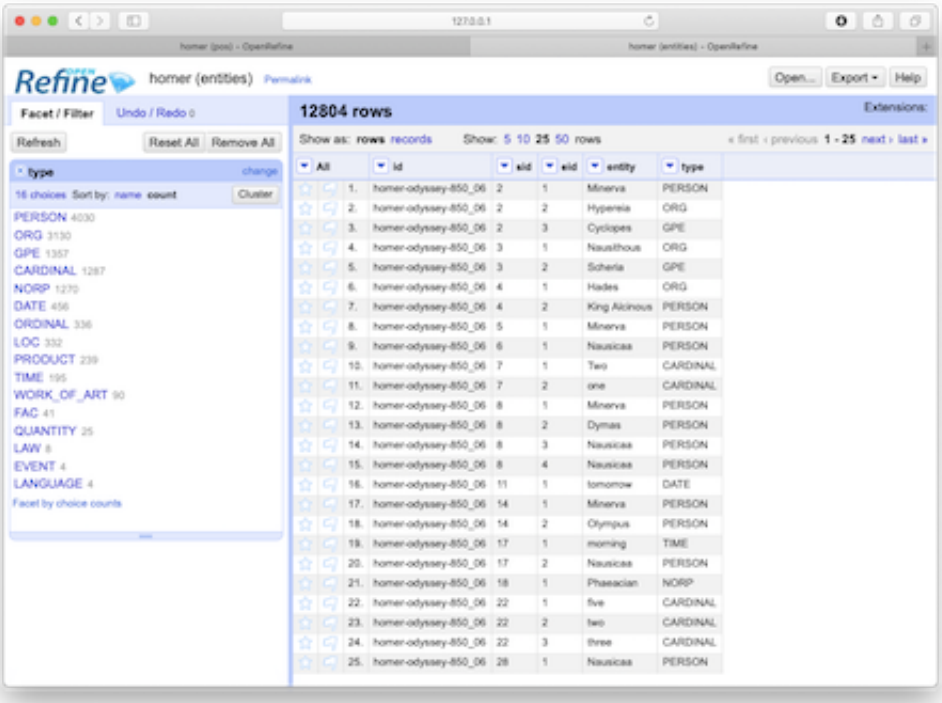
The student, researcher, or scholar can use OpenRefine to open one or more different types of delimited files. OpenRefine will then parse the file(s) into fields. It can makes many things easy such as finding/replacing, faceting (think "grouping"), filtering (think "searching"), sorting, clustering (think "normalizing/cleannig"),

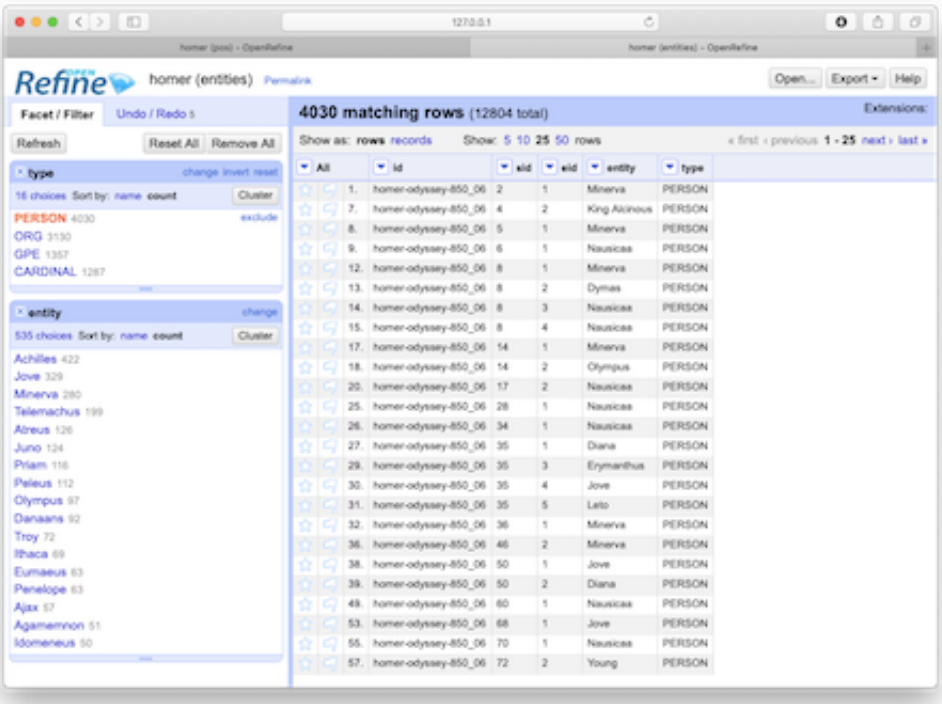
counting & tabulating, and finally, exporting data. OpenRefine is an excellent go-between when spreadsheets fail and full-blown databases are too hard to use. OpenRefine eats delimited files for lunch.

Many (actually, most) of the files in a study carrel are tab-delimited files, and they will import into OpenRefine with ease. For example, after all a carrel's part-of-speech (pos) files are imported into OpenRefine, the student, researcher, or scholar can very easily count, tabulate, search (filter), and facet on nouns, verbs, adjectives, etc. If the named entities files (ent) are imported, then it is easy to see what types of entities exist and who might be the people mentioned in the carrel:









OpenRefine recipes

Like everything else, using OpenRefine requires practice. The problem to solve is not so much learning how to use OpenRefine. Instead, the problem to solve is to ask and answer interesting questions. That said, the student, researcher, or scholar will want to sort the data, search/filter the data, and compare pieces of the data to other pieces to articulate possible relationships. The following recipes endeavor to demonstrate some such tasks. The first is to simply facet (count & tabulate) on parts-of-speech files:

1. Download, install, and run OpenRefine
2. Create a new project and as input, randomly chose any file from a study carrel's part-of-speech (pos) directory
3. Continue to accept the defaults, and continue with "Create Project »"; the result ought to be a spreadsheet-like interface
4. Click the arrow next to the POS column and select Facet/Text facet from the resulting menu; the result ought to be a new window containing a column of words and a column of frequencies -- counts & tabulations of each type of part-of-speech in the file
5. Go to Step #4, until you get tired, but this time facet by other values

Faceting is a whole like like "grouping" in the world of relational databases. Faceting alphabetically sorts a list and then counts the number of times each item appears in the list. Different types of works have different parts-of-speech ratios. For example, it is not uncommon for there to be a preponderance of past-tense verbs stories. Counts & tabulations of personal pronouns as well as proper nouns give senses of genders. A more in-depth faceting against adjectives allude to sentiment.

This recipe outlines how to filter ("search"):

1. Click the "Remove All" button, if it exists; this ought to reset your view of the data
2. Click the arrow next to the "token" column and select "Text filter" from the resulting menu
3. In your mind, think of a word of interest, and enter it into the resulting search box
4. Take notice of how the content in the spreadsheet view changes
5. Go to Step #3 until you get tired
6. Click the "Remove All" button to reset the view
7. Text filter on the "token" column but search for "^N" (which is code for any noun) and make sure the "regular expression" check box is... checked
8. Text facet on the "lemma" column; the result ought to be a count & tabulation of all the nouns
9. Go to Step #6, but this time search for "^V" or "^J", which are the codes for any verb or any adjective, respectively

By combining the functionalities of faceting and filtering the student, researcher, or scholar can investigate the original content more deeply or at least in different ways. The use of OpenRefine in this way is akin to leafing through book or a back-of-the-book index. As patterns & anomalies present themselves, they can be followed up more thoroughly through the use of a concordance and literally see the patterns & anomalies in context.

This recipe answers the question, "Who is mentioned in a corpus, and how often?":

1. Download, install, and run OpenRefine
2. Create a new project and as input, select all of the files in the named-entity (ent) directory
3. Continue to accept the defaults, but remember, all the almost all of the files in a study carrel are tab-delimited files, so remember to import them as "CSV / TSV / separator-based files", not Excel files
4. Continue to accept the defaults, and continue with "Create Project »"; the result ought to be a spreadsheet-like interface
5. Click the arrow next to "type" column and select Facet/Text facet from the resulting menu; the result ought to be a new window containing a column of words and a column of frequencies -- counts & tabulations of each type of named-entity in the whole of the study carrel
6. Select "PERSON" from the list of named entities; the result ought to be a count & tabulation of the names of the people mentioned in the whole of the study carrel
7. Go to Step #5 until tired, but each time select a different named-entity value

This final recipe is a visualization:

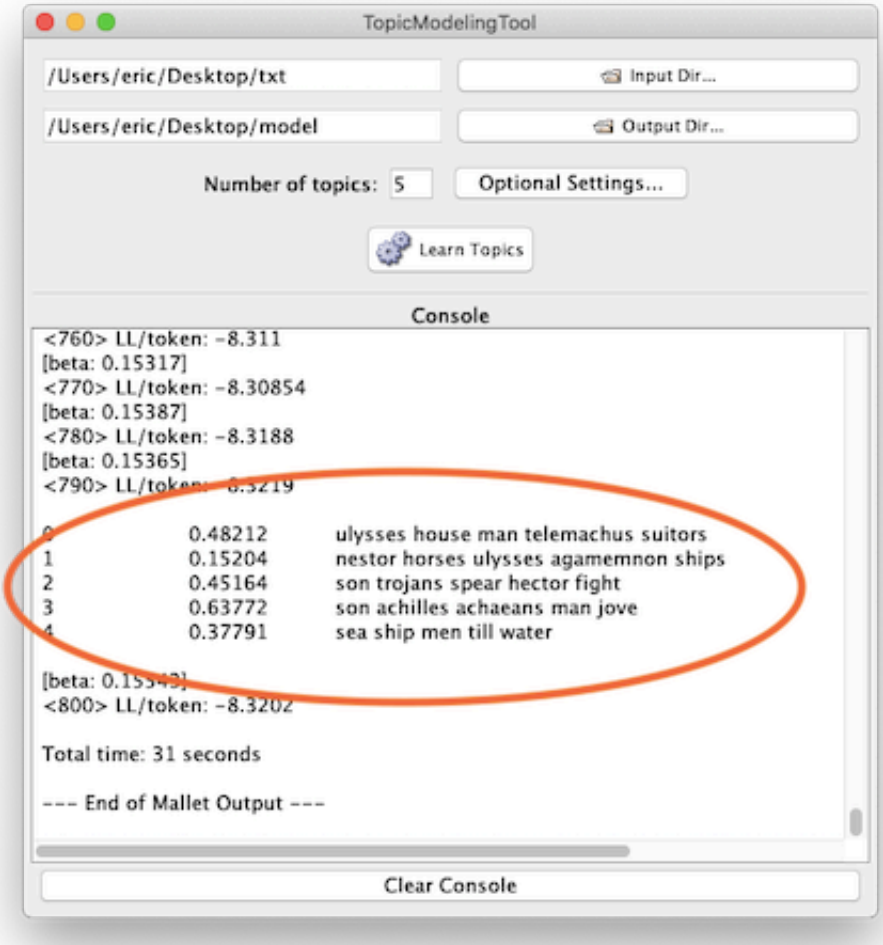
1. Create a new parts-of-speech or named-entity project
2. Create any sort of meaningful set of faceted results
3. Select the "choices" link; the result ought to be a text area containing the counts & tabulation
4. Copy the whole of the resulting text area
5. Paste the result into your text editor, find all tab characters and change them to colons (:), copy the whole of the resulting text
6. Open Wordle and create a word cloud with the contents of your clipboard; word counts may only illustrate frequencies, but sometimes the frequencies are preponderance.

A study carrel's parts-of-speech (pos) and named-entities (ent) files enumerate each and every word or named-entity in each and every sentence of each and every item in the study carrel. Given a question relatively quantitative in nature and pertaining to parts-of-speech or named-entities, the pos and ent files are likely to be able to address the question. The pos and ent files are tab-delimited files, and OpenRefine is a very good tool for reading and analyzing such files. It does much more than was outlined here, but enumerating them here is beyond scope. Such is left up to the... reader.

Topic Modeling Tool - Enumerating and visualizing latent themes

Technically speaking, topic modeling is an unsupervised machine learning process used to extract latent themes from a text. Given a text and an integer, a topic modeler will count & tabulate the frequency of words and compare those frequencies with the distances between the words. The words form "clusters" when they are both frequent and near each other, and these clusters can sometimes represent themes, topics, or subjects. Topic modeling is often used to denote the "aboutness" of a text or compare themes between authors, dates, genres, demographics, other topics, or other metadata items.

Topic Modeling Tool is a GUI/desktop topic modeler based on the venerable MALLET suite of software. It can be used in a number of ways, and it is relatively easy to use it to: list five distinct themes from the Iliad and the Odyssey, compare those themes between books, and, assuming each chapter occurs chronologically, compare the themes over time.



Topic Modeling Tool Recipes

These few recipes are intended to get you up and running when it comes to Topic Modeling Tool. They are not intended to be a full-blown tutorial. This first recipe merely divides a corpus into the default number of topics and dimensions:

1. Download and install Topic Modeling Tool
2. Copy (not move) the whole of the txt directory to your computer's desktop
3. Create a folder/directory named "model" on your computer's desktop
4. Open Topic Modeling Tool
5. Specify the "Input Dir..." to be the txt folder/directory on your desktop
6. Specify the "Output Dir..." to be the folder/directory named "model" on your desktop
7. Click "Learn Topics"; the result ought to be a a list of ten topics (numbered 0 to 9), and each topic is denoted with a set of scores and twenty words ("dimensions"), and while functional, such a result is often confusing

This recipe will make things less confusing:

1. Change the number of topics from the default (10) to five (5)
2. Click the "Optional Settings..." button
3. Change the "The number of topic words to print" to something smaller, say five (5)

4. Click the "Ok" button
5. Click "Learn Topics"; the result will include fewer topics and fewer dimensions, and the result will probably be more meaningful, if not less confusing

There is no correct number of topics to extract with the process of topic modeling. "When considering the whole of Shakespeare's writings, what is the number of topics it is about?" This being the case, repeat and re-repeat the previous recipe until you: 1) get tired, or 2) feel like the results are at least somewhat meaningful.

This recipe will help you make the results even cleaner by removing nonsense from the output:

1. Copy the file named "stopwords.txt" from the etc directory to your desktop
2. Click "Optional Settings..."; specify "Stopword File..." to be stopwords.txt; click "Ok"
3. Click "Learn Topics"
4. If the results contain nonsense words of any kind (or words that you just don't care about), edit stopwords.txt to specify additional words to remove from the analysis
5. Go to Step #3 until you get tired; the result ought to be topics with more meaningful words

Adding individual words to the stopword list can be tedious, and consequently, here is a power-user's recipe to accomplish the same goal:

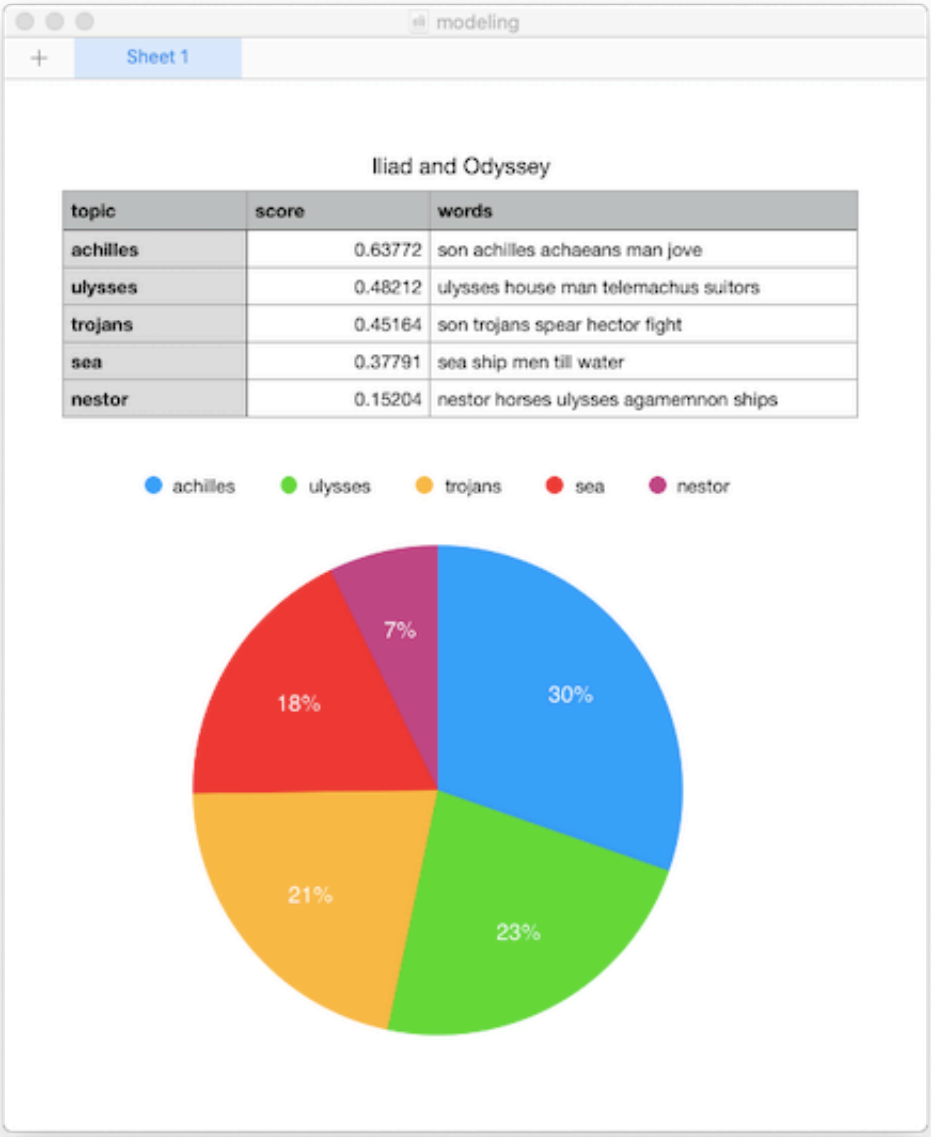
1. Identify words or regular expressions to be excluded from analysis, and good examples include all numbers (\d+), all single-letter words (\b\w\b), or all two-letter words (\b\w\w\b)
2. Use your text editor's find/replace function to remove all occurrences of the identified words/patterns from the files in the txt folder/directory; remember, you were asked to copy (not move) the whole of the txt directory, so editing the files in the txt directory will not effect your study carrel
3. Run the topic modeling process
4. Go to Step #1 until you: 1) get tired, or 2) are satisfied with the results

Visualize topic model (./bin/topic-model.py)

Now that you have somewhat meaningful topics, you will probably want to visualize the results, and one way to do that is to illustrate how the topics are dispersed over the whole of the corpus. Luckily, the list of topics displayed in the Tool's console is tab-delimited, making it easy to visualize. Here's how:

1. Topic model until you get a set of topics which you think is meaningful
2. Copy the resulting topics, and this will include the labels (numbers 0 through n), the scores, and the topic words
3. Open your spreadsheet application, and paste the topics into a new sheet; the result ought to be three columns of information (labels, scores, and words)
4. Sort the whole sheet by the second column (scores) in descending numeric order
5. Optionally replace the generic labels (numbers 0 through n) with a single meaningful word, thus denoting a topic
6. Create a pie chart based on the contents of the first two columns (labels and scores); the result will appear similar to an illustration above and it will give you an idea of how large each topic is in relation to the others

Topic modeling is an effective process for "reading" a corpus "from a distance". Topic Modeling Tool makes the process easier, but the process requires practice. Next steps are for the student to play with the additional options behind the "Optional Settings..." dialog box, read the Tool's documentation, take a look at the structure of the CSV/metadata file, and take a look under the hood at pivot.py.



topic	score	words
achilles	0.63772	son achilles achaeans man jove
ulysses	0.48212	ulysses house man telemachus suitors
trojans	0.45164	son trojans spear hector fight
sea	0.37791	sea ship men till water
nestor	0.15204	nestor horses ulysses agamemnon ships

Using command-line tools to read a study carrel

Building a library

This set of recipes outline how to create a collection of study carrels, and then how to describe them. The first one is simple:

1. create a study carrel and download it to your computer
2. give your carrel a meaningful, one-word name
3. copy it to the directory named "library", inside the workbook directory

Congratulations, your Distant Reader library now contains two items: homer and your newly created study carrel; your library has doubled in size.

This next recipe, which only requires vanilla Perl, gives a study carrel a more meaningful name and scope:

1. Open your command-line interface, and navigate to the workbook's root directory

2. Run `./bin/add-metadata.pl` sans any input to get an idea of the input required
3. Run `./bin/add-metadata.pl homer`; the result ought to be a stream of HTML
4. Scroll backwards in your terminal, and you may notice how the HTML is a study carrel's home page
5. Run `./bin/add-metadata.pl homer > ./library/homer/index.html`; the result will be the creation of a new file -- index.html
6. Use your Web browser to open index.html; remember, as per Step #5, it ought to have been saved in the study carrel named homer
7. Use your text editor to open ./etc/homer.txt
8. Change the existing email address to your email address, and save the file
9. Go to Step #3 two or three times, but change something different each time

By editing ./etc/homer.txt, you were able to give the study carrel a title, scope, and provenance. Here's how to do the same for your study carrel:

1. Duplicate ./etc/homer.txt, and rename it to the one-word name of your study carrel
2. open the file from Step #1, and edit as per the previous recipe
3. Run `./bin/add-metadata.pl NAME` where NAME is the name of your carrel
4. Scroll backwards in your terminal, and you ought to see your edits
5. Run `./bin/add-metadata.pl NAME > ./library/NAME/index.html` where NAME is the name of your carrel
6. Use your Web browser to open index.html; remember, as per Step #5, it ought to have been saved in your study carrel
7. Go to Step #2 until satisfied

If a student, researcher, or scholar finds Distant Reader study carrels both interesting and useful, then this librarian bets subsequently created study carrels will be of a similar theme or ilk. Each carrel won't be exactly the same but similar. Moreover, this librarian bets the student, researcher, or scholar will eventually want to compare & contrast the study carrels. By giving study carrels meaningful titles, scope notes, and provenance the collection of carrels becomes even more useful. The carrels also become more shareable. ("Hint, hint.")

A library of previously created study carrels is available from the Distant Reader website. This recipe outlines how to copy any number of those study carrels to your computer. First you will need a few ingredient: Bash, wget, and a program called untar, but that is probably already a part of the Bash environment. Next, you:

1. peruse the previously created study carrels at https://carrels.distantreader.org
2. identify a carrel, any carrel, of interest, and note its short, one-word name
3. open your command-line interface, and navigate to the root of the workbook directory
4. run `./bin/harvest.sh` sans any input to get an idea of what type of input it expected
5. run `./bin/harvest.sh NAME`, where NAME is the short name of the study carrel identified in Step #2
6. go to Step #1 until you get tired

In the end, you ought to have a collection of at least two or three study carrels. Consider the repeated use of ./bin/add-metadata.pl to give each item more context.

Feature extraction

Text mining and natural language processing often requires the enumeration of "features" -- rather numeric characteristics of a text. These characteristics are also called "features". These recipes outline how to text a few sets of features. For example, everybody want to count & tabulate the frequency of ngrams. This first recipe requires Perl and a module called Lingua::EN::Ngram:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/ngrams.pl` sans any input to learn what sort of input it expects
3. run `./bin/ngrams.pl ./library/homer/etc/reader.txt 1` to output all the words and their frequency in Homer
4. go to Step #2 a couple more times but change the value of the number of ngram

By counting & tabulating the frequent ngrams the student, researcher, or scholar can begin to get an idea regarding the "aboutness" of their corpus. The output of ./bin/ngrams.pl is tab-delimited. Thus the student, researcher, or scholar could redirect the output to a file and subsequently open it in their favoriate spreadsheet application for further processing -- "reading".

Broader concepts

The Distant Reader uses an algorithm called XYZZY to generate lists of keywords for every document in a study carrel. These keywords allude to the "aboutness" of a document. Many words in the English language have broader words as their parents. For example, the word "feeling" may be a broader word for the words "love", "sadness", or "elation". Sets of words characterizing this broadness are called "hypernyms". Given a word, the venerable WordNet thesaurus will return a word's broader term.

This recipe takes a type of word found in a study carrel and returns a list of broader terms as well as their frequency, thus giving the reader an additional sense of aboutness. The ingredients for this recipe include Python and a couple of modules: 1) the whole of the nltk.corpus module, and 2) sqlite3. The first comes along for the ride when you install the whole of the ntlk, and the later almost surely comes along for the ride with any Python distribution. Here is how to output broader terms from many different types of words (nouns, verbs, adjectives, lemmas, and keywords) found in a study carrel:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/word2hypernym.py` sans any input to get an idea of what the script requires
3. run `./bin/word2hypernym.py homer noun`; the result ought to be a list of broader concepts of all nouns in the carrel
4. go to Step #3 but change the type of frequency desired

These sorts of frequencies are rather meaningless in-and-of-themselves, but they begin to take on additional meaning when they are compared with other collections (study carrels). For extra credit, add a different study carrel to your library, and then repeat this exercise against it. Then ask yourself, "How are the carrels similar or different?"

Questions

In the English language, sentences ending in question marks are... questions, more or less. The Reader enumerates each and every punctuation mark in each and every sentence of each and every document in a study carrel. Thus, it possible to find each question mark which is at the end of every sentence, rebuild the sentences, and output the result. Why would you want to output every question in a study carrel? Well, what do yo think you will find near questions in a document? Answers. And everybody seems to be looking for answers.

This recipe's ingredience include Bash, a program called "parallel", Perl, and the Perl module named "DBI" is also required. Here's how to list the questions in a study carrel:

1. open your terminal application and navigate to the root of the workshop directory

2. run `./bin/list-questions.sh` sans any input to get an idea of what the input is
3. run `./bin/list-questions.sh homer`; the result ought to be a list of questions homer study carrel
4. identify a question of interest, and use a concordance to find the question in the study carrel
5. read the text around the location of the question; is there an answer nearby?
6. go to Step #3 until you get tired
7. go to Step #3 but this time output the questions from a different study carrel in your library

Sentences with given keywords

If a word has been denoted as a keyword, then the student, researcher, or scholar will want to read the sentences with the keyword. Such sentences and their surroundings will usually allude to the aboutness of a document. The ingredients for this recipe include Perl and the Perl module named "DBI". This recipe is simple:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/keyword2sentences.pl` sans any input to get an idea of what the input is
3. run `./bin/keyword2sentences.pl homer`; the result ought to a stream of Sentence
4. identify a sentence of interest, and use a concordance to find the sentence in the study carrel
5. read the text around the location of the sentence; does it give you further insite regarding the document?
6. go to Step #3 until you get tired
7. go to Step #3 but this time output the sentences from a different study carrel

Classification and clustering

Classification and clustering are complementary tasks used to subdivide a corpus into smaller corpora whose documents center on a set of themes, subjects, or topics. The use of classification and clustering is a good way to compare and contrast items in a study carrel. Classification and clustering are akin to the process of "divide and conquer".

Cosine similarities

Given a directory of plain text files and a lexicon articulated by the student, researcher, or scholar, this first recipe employs an algorithm called "cosine similarity" to calculate which documents are most alike.

Computing the cosine similarity between sets of documents can be quite computationally heavy. It usually involves the creation of a set of vectors from an entire corpus, and then comparing those vectors to individual items in the corpus. Study carrels can easily include millions of words and the size of resulting matrix can be quite high -- beyond reasonable feasibility for personal computers. Thus, instead of using a whole corpus as the yard stick for comparison, this recipe allows the student, researcher, or scholar to employ a lexicon of their own design as the standard for comparison. Three sample lexicons are provided here: a list of colors, a list of names, and a list of "big ideas". The ingredients for this recipe merely requires Perl and an already-included library called "tfidf-toolbox.pl"; this is a simple recipe:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/compare.pl` sans any input to get an idea of what input is required
3. run `./bin/compare.pl ./library/homer/txt ./etc/ideas.txt`; after a minute or so the result will include a matrix of cosine similarity scores, a list of document pairs ordered by similarity score, and a key denoting which identifiers in the matrix are associated with which documents in the corpus
4. go to Step #3 but this time use ./etc/names.txt and ./etc/colors.txt as the given lexicons

Using the homer study carrel and the ideas lexicon, we can see that the files homer-iliad-850_17.txt and homer-iliad-850_24.txt have a similarity score of 980. Since scores of 1000 denote exact similarity, we can state that when it comes to the set of big ideas, the files homer-iliad-850_17.txt and homer-iliad-850_24.txt are almost identical. In other words, these two documents use the given lexicon in almost the exact same ratio. They are not the documents which use the lexicon the most; they are the documents which use the lexicon similarly.

Here are two exercises left up to the reader. First, identify a document of particular interest. Then, duplicate the document, lower-case all of its words, remove any punctation, find & replace white space with carriage returns, sort the resulting list, remove duplicates, and save the result in a location where you can find it again. Re-run the foregoing recipe but this time use the file you just created as the lexicon. The result will be an ordered list of documents which are most similar to the document of interest. The document of interest ought to be at the top of the list. This exercise is one way to address a perennial problem, "Find more like this one."

Second, identify a document of particular interest. Articulate the qualities that interest you. Do these qualities center around what is discussed (the nouns), what is done (the verbs), how things are described (the adjectives), or maybe who or where is discussed (the named-entities)? Use OpenRefine to count & tabulate these qualities in your document of interest. Save the resulting words in a file, and use them as the lexicon in the foregoing recipe. Again, you will be addressing the perennial problem, "Find more like this one."

Classifying/tagging documents

Students, researchers, and scholars often desire to classify their documents by assigning them with one or more keywords. The student, researcher, and scholar then want to observe the overall "aboutness" of their corpus. This recipe uses the venerable term-frequency/inverse-document frequency (TF/IDF) algorithm to accomplish this task.

TF/IDF is one of the simplest and most well-understood algorithms for calculating the relevancy ranking of search results. Simply stated, it first calculates the relative weight of a given word in a document, and it then compares the weight of the word across the whole corpus. Documents which contain the given word relatively a lot compared to all the other documents in the corpus are considered more relevant. This same algorithm can be turned on its head to determine what words are significant in a corpus as well as significant for a document. In the end, a sort of aboutness of each document can be articulated -- classification/tagging.

The ingredients for this recipe include Perl and two libraries. The first, Lingua::StopWords, is used to import a list of stopwords. The second, which is already included here, is called "tfidf-toolbox.pl":

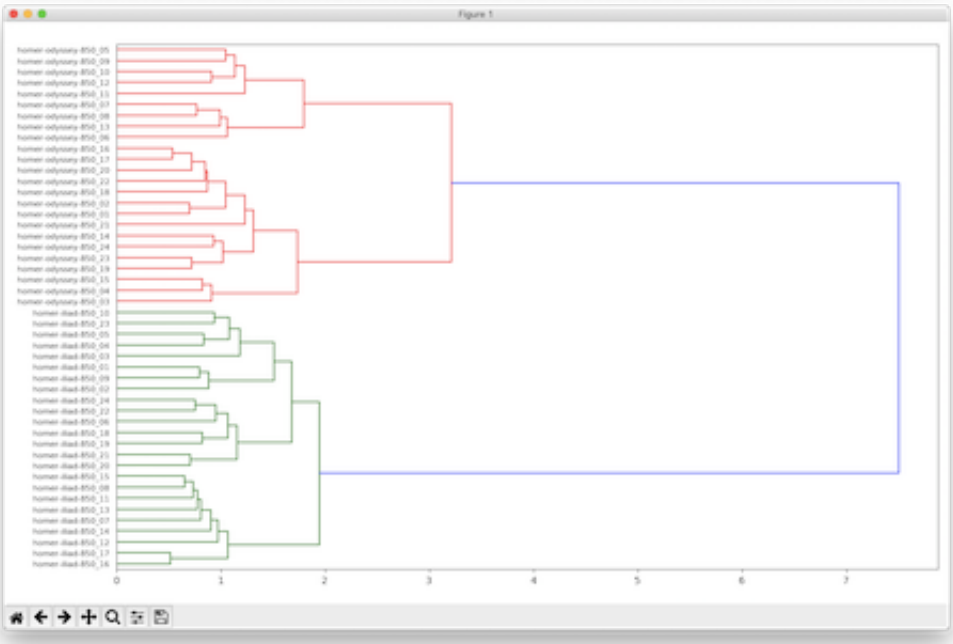
1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/classify.pl` sans any input to get an idea of what input is required
3. run `./bin/classify.pl ./library/homer/txt .005`; the result will be two-fold: a list of keywords (tags) for each document as well as a count & tabulation of the keywords for the corpus as a whole
4. run `./bin/classify.pl ./library/homer/txt .009`; the result, because of a higher threshold value, will be similar to the results of Step #3 but with fewer keywords
5. run `./bin/classify.pl ./library/homer/txt .001`; the result, because of a lower threshold value, will be similar to the results of Step #3 but with a greater number of keywords
6. go to Step #5 until you get tired, but each time change the value of the threshold value until the output outputs values for most (all) documents makes the most sense to you

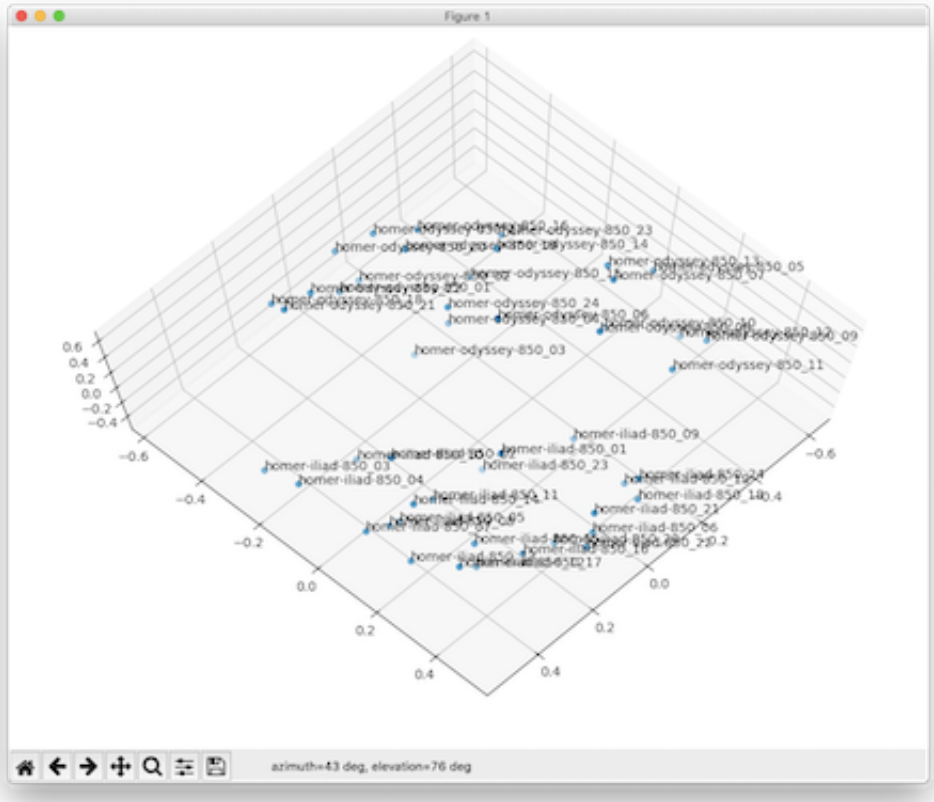
There is no correct value for the threshold, and it needs to be adjusted up or down depending on: 1) the size of your corpus, and 2) the level of detail the student, researcher, or scholar desires. The application of TF/IDF is a quick & easy way to read a corpus.

Visualizing clusters

The following recipe literally illustrates how sets of documents can be grouped into clusters, but the author is remiss because he does not really know how nor why they work. Such is left for the second edition of this workbook. Despite this omission, this recipe is both fun and demonstrative. The ingredients include Python and a whole slew of modules. Please see the source code for details, and upon examination the student, researcher, and scholar will observe the modules rely on themes from the previous recipes: cosine similarity and TF/IDF:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/cluster.py` sans any input to get an idea of what input is required
3. run `./bin/cluster.py homer dendrogram`; the result will be the appearance of a dendrogram illustrating the similarity and differences between the documents in the given study carrel
4. run `./bin/cluster.py homer cube`; the result will be the appearance of a movable cube containing points plotted in a 3-dimensional space, and if you move the cube correctly, then you will literally see how the documents fall into two clusters





Visualizing a network diagram of nouns

Mathematical graph theory (think "nodes and edges") can be applied to texts for the purposes of illustrating relationships between words -- network diagrams. The following recipe applies this concept to co-occurring nouns -- pairs of nouns found in the same sentence. The heart of this recipe is the good work of Team JAMS who submitted it to the PEARC '19 Hack-a-thon contest and subsequently won first prize. *Thank you, and congratulations Team JAMS!*

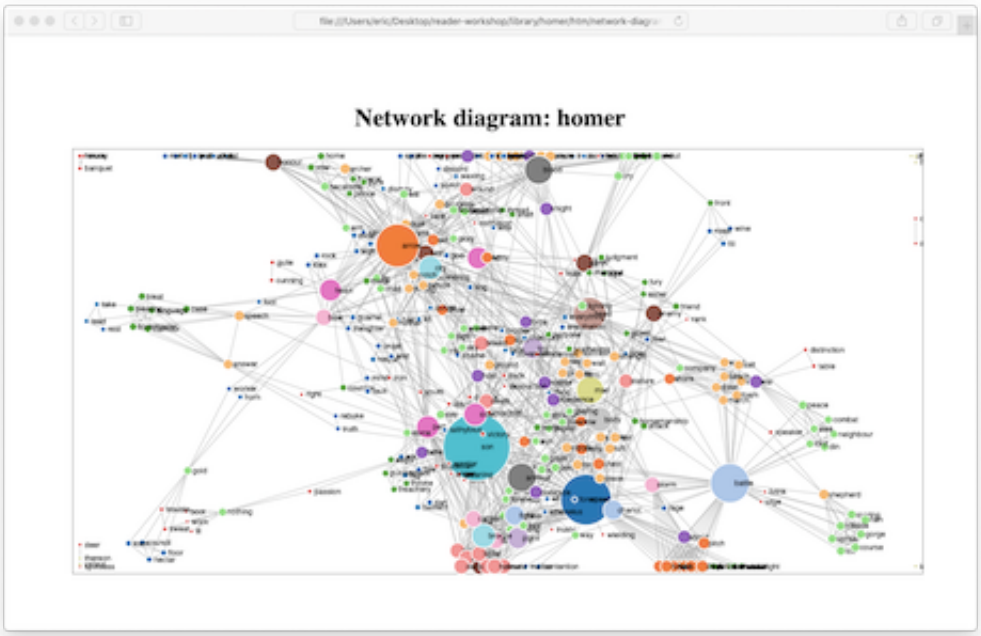
The ingredients for the recipe include Bash, Python, a number of pretty much standard Python modules (which you probably already have installed), and a Javascript library called "D3" (which is included in all Distant Reader study carrels). Let's go:

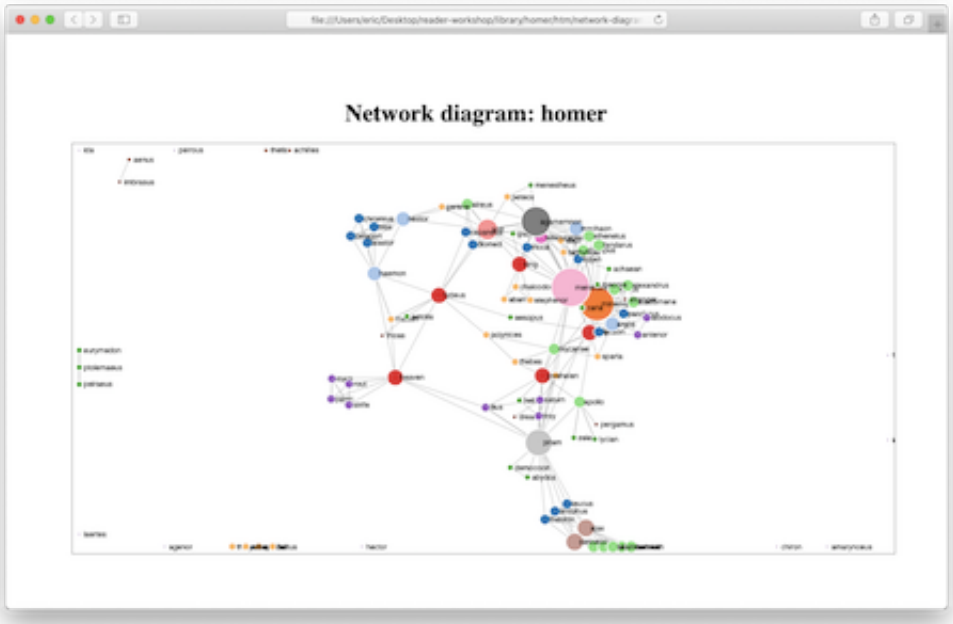
1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/carrel2diagram.sh` sans any input to get an idea of what input is required
3. run `./bin/carrel2diagram.sh homer NN`; after a bit of computation, the result will be the creation of two new study carrel files: 1) a data file (./library/homer/etc/homer.json), and 2) an HTML file (./library/homer/htm/network-diagram.htm)
4. use your Web browser to open the newly created HTML file; the result will look similar to the images below
5. go to Step #3 for each type of noun (NN, NNS, NNP, NNPS)

Each visualization illustrates the number of times a given noun type (nouns, plural nouns, proper nouns, and plural proper nouns) occurs as well as the number of times they co-occur in the same sentence. Put another way, "What are the nouns mentioned 'in the same breath' for a given noun and how often?"

It is not uncommon for mathematical graphs to include too many nodes for effective visualization; the canvas for a large graph often needs be much larger than the typical computer screen. Consequently the results of

this recipe are sometimes difficult to interpret. Solutions are three-fold: 1) output a data file with fewer nodes, 2) apply the recipe to a smaller rather than larger study carrel, or 3) get a bigger computer screen.





Topic modeling

Create Modeling Tool metadata file (./bin/db2malletcsv.sh)

Visualize comparison of topics to metadata

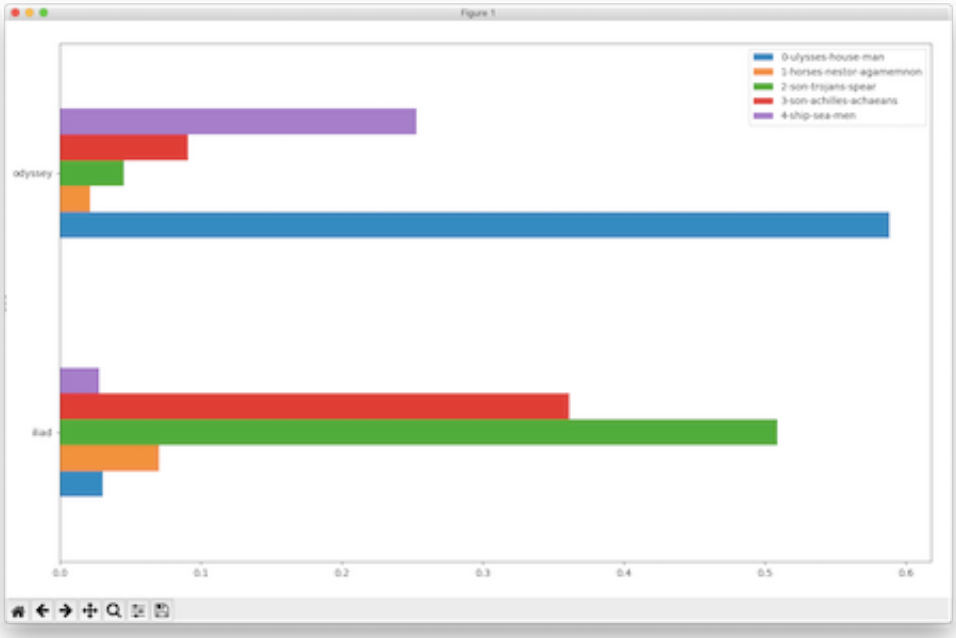
Because of a great feature in Topic Modeling Tool it is relatively easy to compare topics against metadata values such as authors, dates, formats, genres, etc. To accomplish this goal the raw numeric information output by the Tool (the actual model) needs to be supplemented with metadata, the data then needs to be

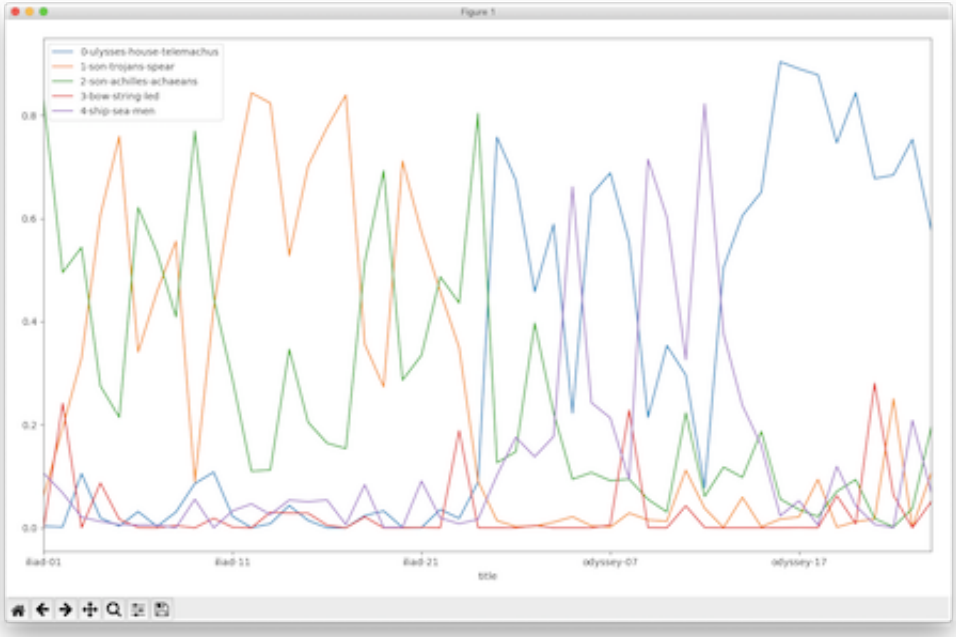
pivoted, and subsequently visualized. This is a power-user's recipe because it requires: 1) a specifically shaped comma-separated values (CSV) file, 2) Python and a few accompanying modules, and 3) the ability to work from the command line. That said, here's a recipe to compare & contrast the two books of Homer:

1. Copy the file named homer-books.csv to your computer's desktop
2. Click "Optional Settings..."; specify "Metadata File..." to be homer-books.csv; click "Ok"
3. Click "Learn Topics"; the result ought to pretty much like your previous results, but the underlying model has been enhanced
4. Copy the file named pivot.py to your computer's desktop
5. When the modeling is complete, open up a terminal application and navigate to your computer's desktop
6. Run the pivot program (python pivot.py); the result ought to an error message outlining the input pivot.py expects
7. Run the pivot program again, but this time give it input; more specifically, specify "./model/output_csv/topics-metadata.csv" as the first argument (Windows users will specify .\model\output_csv\topics-metadata.csv), specify "barh" for the second argument, and "title" as the third argument; the result ought to be a horizontal bar chart illustrating the differences in topics across the Iliad and the Odyssey, and ask yourself, "To what degree are the books similar?"

The following recipe is very similar to the previous recipe, but it illustrates the ebb & flow of topics throughout the whole of the two books:

1. Copy the file named homer-chapters.csv to your computer's desktop
2. Click "Optional Settings..."; specify "Metadata File..." to be homer-chapters.csv; click "Ok"
3. Click "Learn Topics"
4. When the modeling is complete, open up a terminal application and navigate to your computer's desktop
5. Run the pivot program and specify "./model/output_csv/topics-metadata.csv" as the first argument (Windows users will specify .\model\output_csv\topics-metadata.csv), specify "line" for the second argument, and "title" as the third argument; the result ought to be a line chart illustrating the increase & decrease of topics from the beginning of the saga to the end, and ask yourself "What topics are discussed concurrently, and what topics are discussed when others are not?"





Measuring big ideas, name dropping, and colorfulness (./bin/measure-ideas.pl)

Searching

Everybody likes to search.

Now-a-days concordancing goes by the name of keyword-in-context indexing. This recipe's ingredients include Perl and two of its library modules: 1) Lingua::Concordance which does the actual work, and 2) Text::BarGraph used to create a sort of dispersion chart/plot. Here goes:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/concordance.pl` sans any input to get an idea of what input is expected
3. run `./bin/concordance.pl ./library/homer/etc/reader.txt peace`; the result will be two fold: 1) a list of phrases centered on the given word (or regular expression), and a bar graph illustrating where the given word occurs in the text
4. go to Step #3 until you get tired, and if you do go to Step #3, then change your query

The concordance script demonstrated here is no match for the functionality of AntConc, but the script is relatively quick and easy.

Semantic indexing

A semantic index (often times called "word embedding") is essentially a matrix of vectors where each vector denotes a word in a corpus. Query words are first looked up in the matrix and then linear algebra is used to compare the associated vector with the other vectors in the matrix. When vectors "point" in the same direction, then they are considered similar. When vectors point in opposite directions, then they may be akin to antonyms. Searches against semantic indexes return relationships. Given three words of input, some semantic queries may solve an analogy. [Eric waves his arms around in an attempt to point to a few places in a three-dimensional space in order to illustrate the definition of an analogy in a semantic index.]

Semantic indexes require "a lot" of data in order to truly be effective; the following recipes do not do

semantic indexing justice because of the size of the corpus is too small. That said, the first recipe creates a semantic index, and the process is heavy. The ingredients include: Bash, Python, and a few few modules. One Python module is called "gensim" and it supports many natural languaging functions. Gensim does all the hard work of this recipe. The NLTK is needed, merely for its stop words. Lastly, a large module called "spacy" is required. Like Gensim, it supports a wide variety of natural language processing functions, but it used here mainly to parse the corpus into sentences. After installing the necessary modules:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/carrel2vec.sh` to get an idea of what is required
3. run `./bin/carrel2vec.sh homer`; the result will be a set of diagnostic messages outlining the indexing process, and after at least a couple of minutes the process ought to complete

In the end a new file will have been created -- ./library/homer/etc/reader.vec. The resulting file is an index, and the index is not really readable with your text editor.

Once the index is created, you will want to search it. This recipe only requires Python, and more specifically the Gensim module. While the index supports many functions, the following recipe only returns words which have similar vectors to the search word:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/search-vec.py` san any input... to get an idea of what is required
3. run `./bin/search-vec.py homer war`; the result ought to be a list of "similar" words and a similarity score where scores closer to 100 are exact matches
4. identiy a word of interest from the result
5. got to Step #3 until you get tired but this time use the word of interest

Your search results will most likely be disappointing, especially since the similarity scores will almost always be near 100. This is because the corpus is not large enough to be effective. For extra credit, harvest an additional study carrel or two and repeat the previous two recipes accordingly, but keep in mind, the indexing process is not fast. The size and scope of the study carrel named "knowledge" begins to demonstrate the ideas behind semantic indexing.

Free-text indexing with facets

Free-text indexing with faceted results is the type of searching we have all come to love. Enter a word, get back (faceted) results, select an item, and get the associated document. Unfortunately, preparing this recipe is by far the most complicated in the workbook, thus, only an outline will be presented here.

The ingredients are many. First you need a free-text indexer and search engine called "Solr", which is pretty much the gold-standard these days. Installing Solr is as easy a downloading the distribution, uncompressing it, and saving it in a location where you can find it again. Solr requires Java, and you probably already have Java installed. Second, you need Perl, and more specifically, you need two Perl modules: 1) DBI which will interact with the Distant Reader's underlying SQLite database, and 2) WebService::Solr which takes the database output and feeds it to Solr. WebService::Solr is not a small installation.

The first few steps of the following recipe are the most complicated:

1. create a Solr instance/core and the core must be named "carrels-reader" (see the Solr documentation)
2. copy ./library/homer/etc/schema.xml to the newly created core's conf directory; each core is required to include a file denoting the shape of the index, and ./library/homer/etc/schema.xml is just such a file
3. open your terminal application and navigate to the root of the solr directory

4. run `./bin/solr start`
5. examine the output
6. if the output looks like a failure, then ask a friend to help you, and go to Step #1 until you get tired
7. open your Web browser and go to http://localhost:8983/solr/; the result ought to be graphical interface to your Solr index(es)
8. if the result looks like a failure, then ask a friend to help you, and go to Step #1 until you get tired

Whew, if you got all the way through, then the hardest part is over. You now need to index a carrel. Here's how:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/db2solr.pl` sans any input... to get an idea of what is required
3. run `./bin/db2solr.pl homer`
4. examine the output
5. if the output looks like garbage, then find a friend and repeat the first recipe

If you have gotten this far, then the output probably looked like the contents of your study carrel, and it is all down hill from here.

Solr is now running. An index has now been created. It is time to search. The following recipe supports free-text and fielded queries with Boolean logic. The results are faceted on the names of people and keywords. The search results are relevancy ranked, and each item in the list includes basic bibliographics as well as a pointer to the associated document:

1. open your terminal application and navigate to the root of the workshop directory
2. run `./bin/search-solr.pl` sans any input... to get an idea of what is required
3. run `./bin/search-solr.pl homer war`; the result ought to be a narrative text
4. scroll up and down the text and identify an item of interest
5. use your text editor to open the item of interest
6. use your Web browser to open to the same item of interest but located in the study carrel's directory named "cache"; the item in the cache may be more amenable to traditional reading
7. go to Step #3 until you get tired, but each time enter different queries with an understanding that each field is searchable

Everybody likes to search, but even more, everybody loves to get. Remember, the Distant Reader caches the content it reads, and in this way the study carrels are independent of the Web. Search for items in your index and open them from the cache.

Summary/conclusion

About the author

Eric Lease Morgan has been practicing librarianship since 1984, but he has been consistently writing software since 1976. He is currently employed at the University of Notre Dame where he works in the Navari Family Center for Digital Scholarship. In the Center he provides text mining and natural language processing services to the University community.