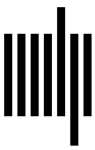
The Open Handbook of Linguistic Data Management

Edited By: Andrea L. Berez-Kroeker, Bradley McDonnell, Eve Koller, Lauren B. Collister

Citation:

The Open Handbook of Linguistic Data Management
Edited By: Andrea L. Berez-Kroeker, Bradley McDonnell, Eve Koller, Lauren B. Collister
DOI: 10.7551/mitpress/12200.001.0001
ISBN (electronic): 9780262366076
Publisher: The MIT Press
Published: 2022



The MIT Press

6 Transforming Data

Na-Rae Han

1 Introduction

Data rarely come in a readily usable form. From the moment they are sourced to the end point when they are published or archived for long-term storage, they typically undergo many stages of transformation. Throughout the life cycle, it is up to the data practitioners—creators, end users, and anyone who processes data in any manner—to ensure its integrity by tightly controlling the four main outcomes of a transformative operation—*transliteration*, *loss*, *augmentation*, and *corruption* of information—all the while maintaining reversibility through judicious employment of version control. This chapter presents an overview of considerations going into planning and managing data transformation processes as well as recommended tools and best practices. Of the two main forms of linguistic data—textual and audiovisual—the chapter will largely focus on the former.

2 Why transform data

There are several use cases in which the need for data transformation may arise. A research project may begin with a targeted data collection effort—fieldwork, surveys, elicitation, and more; the resultant data will then go through the usual process of organization, clean-up, and transformation in the course of the intended research activity. Further polishing and documentation may also be applied in preparation for its ultimate publication to accompany the research output.[1]

Another frequent scenario starts with a published data set. Adopting such data to use in one's research project will often require conversion or reformatting of data files as the very first step, as each software platform is typically designed to work with certain types of input. Data transformation in this case is conducted primarily with the purpose of producing intermediate forms that can then be passed to a data analysis platform of choice. In this scenario, data transformation is performed for personal use on an as-needed basis.

In this last and increasingly common scenario, proliferation of digital text has enabled linguists to adopt a *data science*[2] approach through which spontaneously occurring instances of language are harvested and repurposed for linguistic research. These data sources' native formats are often ill-suited for the purpose of data exchange and analysis and hence will require conversion. For example, web contents are usually encoded as HTML,[3] and text documents typically exist as word-processed formats such as Microsoft Word or PDF: their proprietary nature and/or heavy presence of stylistic elements mean the linguistic content contained within are not readily processable as *data*. In addition, these types of "data in the wild" are notoriously noisy and will necessitate multiple steps of cleaning and transformation before they can be utilized in scholarly research. With appropriate copyright clearance and redaction of personal information, it may then be published and shared with scholarly communities at large, the original language community, and a wider public (see Collister, chapter 9, this volume).

3 What's transformed: Form vs. information content

Data as stored in their electronic form are binary sequences of 0s and 1s, where there is no distinction between form and content. On a higher level, however, it makes a conceptual sense to draw a principled distinction between transformation's two main targets—*form* and *information content* of data.

The issues surrounding file storage mostly concern the *form* of data. Examples include conversion between DOS-format and Unix-format text files, and character-encoding

conversion between, say, ASCII and eight-bit Unicode transformation format (UTF-8). To a limited extent, conversion from a proprietary format such as Microsoft Excel workbook (.xlsx file extension) or Microsoft Word (.docx) into a portable, open standards such as comma-separated values (CSV, .csv file extension) or a plain text file (.txt) can be thought of as a matter of form only. With such form-focused transformation, the intent on information content is, invariably, *preservation*: relevant operations are done in a way that no information content is lost during the process, and any loss of information, if unavoidable, must be minimized, controlled, and remedied. Note that conversion from proprietary formats such as Microsoft Word or Excel is fundamentally not lossless and will incur removal of typesetting references such as paragraph formatting and font sizes. Loss of these data bits, unless they encode linguistically relevant information,[4] is something that is desired and intended as part of a transformation process.

Beyond individual files, a data set involving multiple files and directories may need reorganization, especially during a development cycle. Renaming of files and directories, creation or removal of directory structures, and moving of files are common tasks; furthermore, multiple files may need to be joined into one, or a single file may need splitting. At first these operations may seem to only concern the form of the data, but they in fact tend to be prompted by a need to address deeper, often implicit, central parameters of a given data set, which is to say they hinge crucially on the *metadata* aspect of data content. Consider, for example, how positive and negative movie reviews may be segregated into two folders named "neg" and "pos" and how file names often bear extralinguistic information such as a language user's gender and types of elicitation tasks. File and directory structures are often co-opted this way as the principal means to encode the key dimensions of a data set. When files and directories undergo reorganization, these crucial bits of information will need to be preserved and encoded in some other, improved, way. Undertaking such an overhaul, then, must be preceded by a careful reflection on the purpose and design of the data set at hand. Also important is compliance with best practices and data management principles shared by the research community at large; Corti et al. (2019) and Mattern (chapter 5, this volume) provide in-depth recommendations on this very topic.

Turning now to the effect of data transformation on information content, let us distinguish its four main outcomes: *transliteration*, *loss*, *augmentation*, and *corruption* of information. *Information* in this regard does not narrowly refer to literal symbols, tokens, or values but should be understood broadly as *meaningful distinction* as present in the data.

1. *Transliteration* occurs when the information contained within data is converted to its isomorphic counterpart. The source and the target will have one-to-one mapping, and as a result no information is gained or lost through the process. An important corollary is that the output of transliteration can always be transformed back to its original form. At the file level, certain format conversions can be conducted in this fashion, such as one between CSV and tab-separated values (TSV). Closer to the actual content of data, relabeling a closed set of attributes can be done in an isomorphic manner: converting integer labels "1-2-3" to string-based and more descriptive "low-medium-high" is a simple example. A more complex case might involve, say, converting raw frequency counts of words into a more relative measure such as the per-million frequency count: this process is considered transliteration as long as the total word count that forms the basis of the latter figure stays known, thereby ensuring recoverability of the former, original raw counts.
2. *Loss* refers to loss of immaterial information that was managed in a purposeful manner; it can therefore be thought of as more of "trimming" or "simplification." There are a few reasons why one would implement loss of information. The first is to increase internal *consistency* of information: mapping an open set of attribute values into a closed set will necessarily incur loss of information but hopefully of the spurious kind only, which leads to increased data-internal consistency. The second reason is to increase information *density*: language data pulled directly from authentic communication streams will be saddled with vast amounts of noise and clutter, which then will need to be pruned to bring the main purpose of the data set into a sharper relief. Another is *redaction*: naturally occurring data may contain sensitive and private information that should be purged before publication.

3. *Augmentation* occurs when new information is added into a data set from a secondary data source. The new knowledge can consist of a layer of linguistic interpretation, commonly referred to as annotation: it can be added via manual annotation done by humans, or through the use of automated natural language processing tools. Producing and then incorporating various types of standard measurement is another way; merging in information culled from a static, published data source is yet another. When managed right, augmentation can be an excellent way to increase utility of a data set.
4. *Corruption* occurs with *unintended loss* of *meaningful* information that was present in the original data. A failed attempt at encoding conversion may leave an entire data file unusable; a poorly managed transformation operation may accidentally erase a meaningful distinction contained in the source data. Corruption may also occur in the course of adding *new* information, when such a process leads to introduction of unreliable or even false distinctions that are taken as legitimate or slip in unnoticed as a by-product. In other words, newly added knowledge that is low in fidelity is more of a detriment, even an element of corruption, than an improvement to a data set.

When managing a data transformation project, it is imperative to vigilantly ward off corruption as an unintended consequence while maintaining a clear-eyed approach to the desired outcomes of transliteration, loss, and augmentation. These three modes must also be isolated from each other: a single processing step should identify one of the three as its target outcome and execute it with precision. Versioning likewise should be conducted in-step; data versions that are associated with a clearly delineated set of changes that neatly align with the three modes of transformation will facilitate their management. One key consideration is that these operations must be kept reversible through stages of transformation, for which employment of popular version control systems such as Git becomes paramount.

4 Transformation operations in depth

This section presents an in-depth look at many commonly used transformation operations.

4.1 File format conversion

Electronic documents come in many different formats, which are signaled by file name extensions such as .docx, .xlsx, .pdf, .html, .xml, .json, and so forth (see table 6.1 for detailed specifications). These formats reflect their origin as digital content: for instance, student writings typically come in the Microsoft Word format, and scraped web documents will likely be in HTML. These formats will then need to be converted into those that are more suited to the purpose of data analysis and, further, data exchange (Austin 2006; Bird & Simmons 2003). The first essential consideration in this process is that of proprietary versus open-standard formats. Proprietary formats such as Microsoft Word and Excel are not accessible without a paid software license, and they are not machine-readable through industry-standard computational platforms, which make them particularly ill-suited in a data processing context. Their ownership by private companies also means long-term accessibility is far from guaranteed. It is therefore imperative that such files be converted and saved as formats with wider utility, such as XML, plain text files, or CSV files. Table 6.1 summarizes popular file formats for textual data.

The ultimate choice of a data file's format hinges on the natural organization of the information it carries. In general, CSV is well established as the standard format of choice for tabular data consisting of columns and rows of records; the plain text format with the .txt extension is best for a large number of flat-structured texts that are individually too long to store as a single column value in a tabular structure; hierarchically structured textual data, especially those with annotation, will find a best fit in XML.

Whatever the end choice, it is important to understand that these optimal data-exchange file formats are all essentially plain text files. While truly plain text files, by convention given the .txt extension, are understood to be a piece of continuous text, other formats such as CSV, TSV, XML, and JavaScript Object Notation (JSON) are merely an extension where a predefined set of reserved characters (e.g., "," in CSV and "\t" in TSV acting as column separators) or a sequence of characters (e.g., < . . . > in XML that acts as markup tags) are designated as formatting constructs and keywords. Applications specializing in these formats are able to open the files, parse the formatting constructs and contents into appropriate data structures, and present a rendered view, along with functions through which an end user can manipulate the data

Table 6.1
Common file formats and recommended conversions

Extension	File format description	Plain-text format?	Suitable for data exchange?	Recommendation
.doc, .docx	Microsoft Word files	No	No; proprietary file format	Convert to XML or plain text
.xls, .xlsx	Microsoft Excel spreadsheet files	No	No; proprietary file format	Convert to CSV or TSV
.csv	Comma-separated values	Yes	Yes	
.tsv	Tab-separated values	Yes	Yes	
.pdf	Portable Document Format, developed by Adobe	No	No; while not proprietary, contains non-textual elements	Convert to XML or plain text
.xml	Extensible Markup Language with nested data structure	Yes	Yes	
.html, .htm	HyperText Markup Language, web pages	Yes	No; typically contains scripts and many web-related code bits	Clean up scripts and other extras; convert to XML
.md	Markdown, essentially a plain-text file with light formatting	Yes	No; contains formatting information	
.json	JavaScript Object Notation; popular format in social media	Yes	Yes	
.txt	Plain-text file	Yes	Yes	

content. But because these are still plain text files underneath, they can readily be opened and modified as such through simple text editors; in this type of interface, formatting constructs such as "," and "<" lose their special-character status and are treated as literal characters like any other that can be freely edited. This brings the associated risk of accidentally damaging the all-important data structure of a file itself, but the upside is that this affords the end user the ultimate control over the file content.

In dealing with file formats, it is important to understand this duality of format-specific meta-characters. For instance, because the comma "," is a special delimiter character in CSV, *literal* tokens of comma as in "Blood, sweat, and tears" or "10,000,000" in a CSV file will somehow need to be discernable as such. How is this accomplished? There are a few standard approaches. One is to utilize the practice of *escaping*, where a single meta-character is designated with the function of turning literalness on and off. The backslash "\" is the most common such escape character. In CSV, hence, any instances of "," will be understood as a field delimiter except for those prefixed by "\" as in "\,"; whitespace characters such as the newline and the tab characters are commonly represented in an escaped form, as in the tab character "\t". In such schemes, a literal backslash is produced through self-escaping, that is, as "\\". Another common method is to turn on literalness within a certain scope: in some implementation of CSV, commas are understood to be literal when appearing inside a string sequence enclosed in a pair of double quotes ("..."). Lastly, a literal version of a special character may be expressed as a designated code: in HTML, the literal versions of the less-than ("<") and greater-than (">") characters are written as `<` and `>` utilizing `&` and `;` which themselves are special characters in HTML. The ability to correctly recognize these elements and translate them accordingly is at the core of file format converters; however, having an overall understanding of this underlying principle comes in handy in occasional trouble shooting, often through the use of a text editor.

Embracing text editors as a go-to application in one's workflow is a hallmark of experienced data practitioners. Novices will find it tempting to retreat to the familiarity of Microsoft Word and Excel, which after all seem reasonably capable of importing, editing, and exporting back most of these formats. However, one should

consider the fact that these consumer-oriented applications are hobbled by their propensity for applying alterations without the user's knowledge (see section 6), with often fatal consequences to data integrity. Text editors, by contrast, take more principled and transparent approaches when it comes to file content manipulation and tend to give the end users explicit and finer control. There are many excellent text editors available, including Notepad++[5] (Windows), Sublime Text[6] (Mac), and Atom[7] (all platforms), that are recommended over the bare-bones text editors that come preinstalled in Windows 10 and the Mac OS operating systems.

Returning to the matter of conversion, how does one go about converting a file from one format to another? When dealing with only a handful of files, the simplest choice is manual export through the format's known native application. For example, Microsoft Word documents can be "saved as" XML or a plain text file; Excel likewise provides an option to save a workbook file as a CSV (.csv) file. Additionally, one may utilize third-party applications specializing in document conversion; Pandoc,[8] in particular, bills itself as a universal document converter and is capable of handling such diverse formats as Microsoft Word, Open Office, HTML, XML, and LaTeX. The sheer number of compatible formats seems impressive enough, but the true power of third-party tools such as Pandoc lies with their command-line capability: most can be called in a command-line interface along with optional parameters for fine-tuning, making it possible to automate conversion of a large number of files. As with any tools, though, conversion tools are not fail-proof and the end result will need careful inspection. We will return to command-line–based automation in section 5.1.

4.2 Text files: Character encoding, line ending

As we have seen, data-exchange file formats are fundamentally plain text files, for which one can identify two central issues—character encoding and line ending. Unicode (Unicode Consortium 2019) is gaining increasingly wide adoption as the universal character encoding standard, yet the reality is that text files still come in many different and often hard-to-detect character encodings. Unlike Linux and the Mac OS which use UTF-8, a subtype of Unicode, as the OS-default character set, Windows adopts non-Unicode character-encoding systems for its various language versions, which is a common source of encoding conflict. For its English and Western language versions it uses a system called Windows-1252, also known as CP-1252 or ANSI (for American National Standards Institute). For the sake of cross-platform compatibility, text files should be encoded in Unicode whenever possible, in UTF-8 (a subtype of Unicode that uses eight bits as the minimum character width) or UTF-16 (same but uses sixteen bits as the minimum width).

One commonly encountered source of variability within Unicode encoding is what is known as the "byte order mark" or BOM, represented as a non-printing Unicode character code U+FEFF at the beginning of a file. Basically, BOMs work as a signature that specifies the encoding and byte order of a file to an application that reads it in. However, their presence or absence is something that is mandated by an individual application's protocol, which creates complications. The details surrounding BOM usage are highly technical, so let us simply make a note of a few fundamental issues here: (1) use of a BOM may be required at the protocol level adopted by a particular application, and (2) conversion therefore must be done with an application in mind, and finally (3) BOMs should be avoided to the extent possible. For further details, readers should consult the Unicode Consortium's standard document[9] and its frequently asked questions section on BOMs.[10]

The second important aspect of text file encoding is line ending, which again shows OS dependency. Historically Unix and Linux have used LF (line feed, a single "\n" character) to encode the end of a line, which the Mac OS has since adopted; Windows, on the other hand, uses CRLF (carriage return followed by line feed, encoded as a sequence of two characters "\r\n"). This creates a fundamental dissonance between the two groups of operating systems regarding what exactly constitutes a line in a text file; a text file with "\n" line breaks, when opened in a Windows application, may show up as a single continuous line of text without a break. Windows OS however is reportedly on track to increase compatibility with the Unix-style line breaks. Here, too, the recommendation is to adopt Unix-style LF line breaks, irrespective of the OS.

Further complicating the matter is the concept of *locale*, a set of system-wide parameters that defines the default language, country, regions, and other variations. The problem of default character encoding that we have

discussed is closely related to locales, which follows from the OS's region and language setting. Importantly, applications depend on the locale setting when opening and interpreting a file, which then means that the same file, opened on different OSs or under different language versions of the same OS, may behave differently. Crucially, applications will also apply locale-specific settings when saving out files, which may lead to unwitting modifications of the fundamental characteristics such as line ending and character encoding. It is therefore imperative, especially in a collaborative environment, to be aware of and look for locale-induced problems.

Converting a text file to use a different character encoding or a different line ending is something that all text editors are equipped to handle. Figure 6.1 is a screenshot of Notepad++; through its "Encoding" menu, the text file currently in ANSI (i.e., Windows-1252) encoding can be converted to UTF-8. The line ending is currently set to Unix (LF) as shown in the bottom status bar, which can be converted via "Edit" → "EOL Conversion" menu (EOL: end-of-line).

Beyond a single file, converting a folder full of text files is best handled through batch processing in a command-line environment. Popular tools specializing in these conversion tasks are Unix-native applications: once considered tools of the trade for those exclusively inhabiting the Unix/Linux world, their general availability has drastically expanded in recent years. These tools now come preinstalled or otherwise easily installable in the Mac OS; on the Windows side, the Git Bash package[11] comes bundled with most popular Unix command-line tools. To name a few, `dos2unix` and its reverse `unix2dos` are used for line break conversion; `iconv` on the other hand is a venerable tool used for encoding conversion. The example command that follows converts an input file's content from (`-f`) the Windows-1252 encoding to (`-t`) UTF-8, the output stream of which is redirected into (`>`) a new file called `outputfile.txt`.

```
iconv -f windows-1252 -t utf-8 inputfile.txt >
outputfile.txt
```

4.3 File and directory operations

A need for conversion often extends to multiple files, at times hundreds or even thousands of them, at which point automating the process through the command line becomes an absolute necessity. Figure 6.2 illustrates how the same operation using `iconv` is repeated on all text files (ending in .txt extension) in the directory, using the Bash shell's `for` loop syntax and variables. The converted files, bearing the original file name designated through the variable `$file`, are saved in a new directory called `converted`.

This is an example of a relatively simple operation executed on the fly, but more complex tasks can be planned out and executed through the use of a shell script. Section 5.1 lists a few learning resources on the Bash shell and scripting.

More generally, reorganization of the entire file and directory structure of a data set will often involve command-line operations. File format conversion and

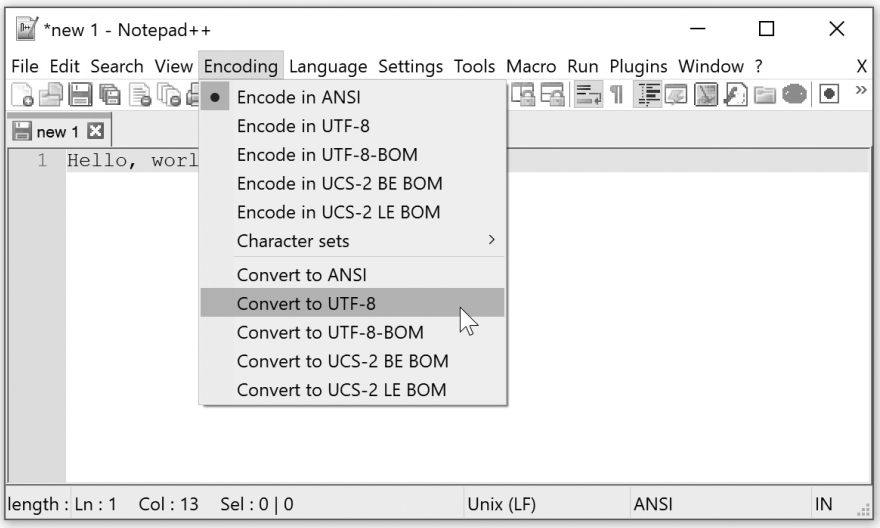


Figure 6.1
Encoding conversion in Notepad++.

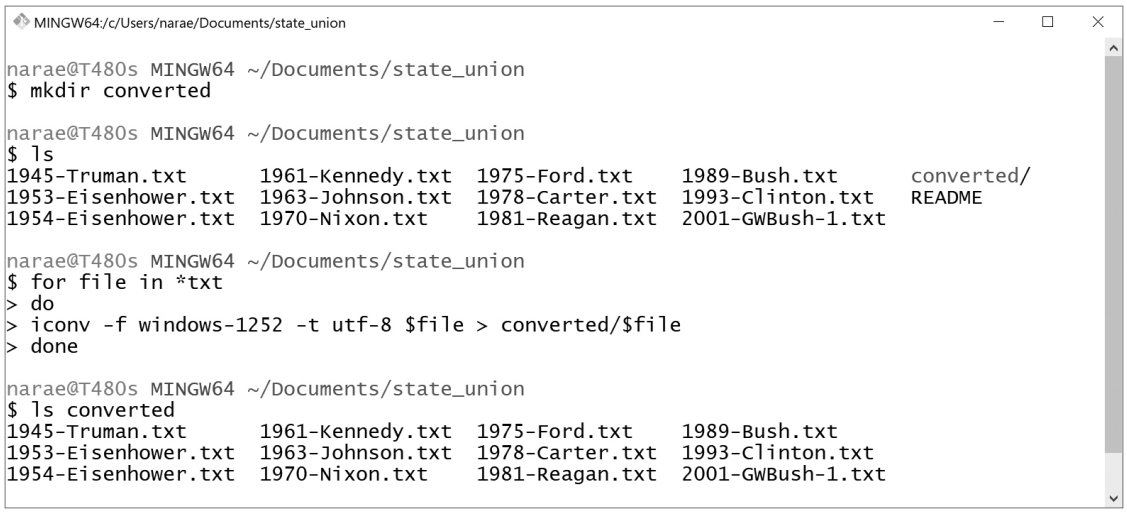


Figure 6.2
Batch processing on multiple files through command line.

encoding conversion are two particularly common use cases. At other times, files may be joined, split up, grouped into new directories, and renamed. There are Unix tools specializing in such operations, which are typically coupled with a loop syntax available through popular shells such as Bash and Zsh. The Mac OS, based on Unix, supplies the command-line interface and both shells natively through its Terminal app. On the Windows side, the Bash shells and Unix tools are available through Windows Subsystem for Linux[12] and also Git Bash installation. See section 5.1 for details.

As briefly noted in section 3, reorganization of file and directory structure is often prompted by a deeper consideration surrounding the design of a data set. Folders, files, and text lines are largely platform-independent units of information storage and hence the primary device for imposing organization. An optimal setup therefore may see them aligned with discrete categories intrinsically present in a data set, with separate folders provisioned for salient top-level categories, and each text file or line of text corresponding to a single data sample, and so forth. Overhauling this alignment is a leading cause for undertaking file and directory restructuring. An archive of student writing samples that came as a single, long word-processed file is best split into multiple plain text files with a single file corresponding to a single writing sample. On the other hand, language samples from social media tend to be short with rich metadata therefore are best stored as a single file with a tabulated format, with each line corresponding to a row entry.

How this overall alignment is configured has a fundamental impact on the constitution of a data set. The extent to which metadata should be encoded in the file and directory structure is in fact a design-critical issue. A flat directory with tens of thousands of unordered, poorly or inconsistently named files could use folder organization; conversely, excessive depth in directory hierarchy should be avoided, as deeply nested directories can be a hindrance to navigation. File names, on the other hand, are often co-opted to encode central parameters such as demographic information, a practice that can quickly become intractable. Given these considerations, beyond one or possibly two top-tier parameters, it is best to keep directory structures and file names simple and instead record metadata in a separate, tabulated documentation file, where each file name is listed with as many parameter attributes as there are without being bound by space concerns. Again, shuffling around these integral bits of information—extracting them from file and directory names; redirecting them into a new file; renaming, moving, and deleting files and directories—are all tasks that are best automated through command-line operations and, needless to say, with utmost care.

4.4 Data cleaning and normalization

Data, especially if sourced from naturally occurring communication streams, is bound to be messy. It may contain

irregularities that necessitate manual or automated correction. It may also contain a mass of irrelevant content that distracts from the intended purpose of the data set. Personal information may be present, which will require redaction and anonymization. Standardization and normalization steps will ensure consistent codification of attribute values. These tasks, especially if done through automated means, require methodical approaches as well as a tight control on quality. Inadequately executed data cleaning and reorganizing operations are a common source of data corruption.

The first type of data cleaning effort concerns culling of extraneous content. When data is first exported from the original source, it will likely contain an excess of extralinguistic information that might not provide much utility. For instance, surveys conducted through popular online platforms typically record entries such as Internet protocol addresses of respondents, time and duration of access, and more; they will then show up as columns when the survey data are exported as a spreadsheet form. When publishing these data alongside the research, it may be beneficial to drop such data fields and categories entirely, which not only addresses the privacy concern but also keeps the data content on point.

This type of categorical removal tends to be fairly straightforward, but more nuanced assessments are called for when deciding on inclusion or removal of data samples. With a set of thousands of essays submitted by students, some may be of questionable value. Should you be dropping writings that solely consist of "asdfasdfasdf"? How about one that simply reads "I don't know, this question is hard" or multiple copies submitted by a single author with only slight variation? Should you be dropping writings by respondents who skipped over a demographic background question? Furthermore, if handling a large data set, it will be impossible to manually inspect all data entries, which means one will then need to decide what criteria and computational means should be employed to precisely spot and filter out undesirable samples.

Additionally, data culling may be motivated by design considerations. For a data set aiming to focus on a few key parameters, dropping of certain samples may be justified if they are not a good fit in those regards. Likewise, if achieving a balance among certain parameter values is part of the design goals, a decision may be made to remove excess samples. In conducting these quota-based removals, care must be taken to ward off selection bias by adopting randomization in the process.

Another chief motivation for data cleaning is normalization. Missing values are part of naturally occurring data and will call for a decision on their treatment: leaving as is, filling in with a default value, filling in through approximation, or dropping of samples are possible choices. Validation of data values is another critical step: what is presumed to be a closed set of values is often not, which will then require correction. For instance, when you have part-of-speech tagged text data, it may be entirely possible for invalid tags to be present; they can only be discovered via a data set–wide validation, which involves rounding up all tags and reducing them into a unique set. User-reported values present yet another challenge. A data column that contains information on a language user's first language, if self-reported, will contain an unruly set of string values ranging from, say, "Sinhalese," "sinhala," "Sinhala language," "Sinhales [*sic*] and Tamil," which will need to be interpreted and then mapped onto unique, closed-set values, such as the International Organization for Standardization's ISO 639 language code *si* or *sin*, and in the last case possibly *sin;tam*.

In some other instances, contextually dependent data may be converted into related yet more readily usable values. For example, survey data may only include information on respondents' birth year; it may be advantageous to covert the information into their age at the time of data collection and, depending on the purpose of the data, even be necessary to bin the age values into a few key groups. It is also common for a data set to start with certain categorical values encoded in community-internal shorthand, which may benefit from remapping into a more universally interpretable system of representation before publication.

Lastly, anonymization is often a crucial step before publication. Personal names, user identifications and e-mails are all sensitive information that needs to be redacted. User identifications are typically used as a record identifier, which will then need to be replaced across the data set with randomly generated unique identifiers. In-text mentions of personal names and email addresses will need to be identified and then replaced by a designated placeholder. Correctly identifying all instances, however, will be a challenge, because these are typically an open set; such operations are therefore undertaken by use of pattern matching via regular expressions or

off-the-shelf natural language processing (NLP) applications capable of named entity recognition.

All in all, data cleaning and normalization are no doubt the most underrated aspects of data-side work: they routinely reveal themselves to be more complex and more time- and effort-intensive than one would initially anticipate. At the same time, the payout of good, quality work tends to remain largely invisible—people rarely take time to appreciate a data set on its cleanliness and thoughtful construction—while the stakes of data corruption and data loss are high. Another related paradox is that the tasks involved are viewed as grunt work, yet executing them right will require considerable experience and high levels of technical sophistication. It is therefore not uncommon to encounter cases of time-strapped faculty members coming to regret tasking undergraduate research assistants with data-cleaning work, because they end up spending more time either teaching the students the processes or tracking down and fixing their mistakes.

Despite these challenges, or perhaps because of them, the importance of getting these steps right cannot be emphasized enough. The creators of a data set, with access to the full context surrounding its origin, are in the best position to apply informed interpretation and bring order to the data in the way that minimizes loss of meaningful distinctions. Publishing insufficiently cleaned or normalized data in turn has many negative consequences: it creates unnecessary duplication of efforts on the part of individual end users; without access to the full context, users run the risk of arriving at an incorrect interpretation; and research founded on disparate postprocessing treatments will be neither readily comparable nor easily reproducible.[13]

Lastly on the tools side, cleaning operations on a handful of data files may be accomplished manually via software platforms that are native to the data format. For example, Microsoft Excel can handle pruning of columns or simple remapping of values in tabular data; text editor programs such as Notepad++ and Atom all come with find-and-replace functions that can even work on user-supplied regular expression patterns. With a larger data set, and with more complex data cleaning tasks, it becomes necessary to bring in more comprehensive computational tools such as command-line scripting, Python, and R. On these platforms, a user can conduct a full range of data inspection—sampling, probing, compiling descriptive statistics, rounding up attribute values—all the while diligently checking for anomalies. Then, custom functions can be defined that will target particular values, patterns, or constructions and produce transformed values, which then can be applied to the underlying data to obtain a transformed output. This sort of bulk transformation however is powerful and is all too frequently a source of data corruption. We will return to this topic in section 6.2.

4.5 Synthesis, merging, text processing

This section covers processes by which new information is added into a data set. There are two main modes: first, pulling from additional data sources and merging in relevant bits of data points, and second, augmenting text content with interpretive knowledge such as word tokens and parts of speech often through the use of popular NLP applications.

Augmenting your data with additional information available from a published source can aid discovery and enhance its value for the research community. For example, if you have collected an archive of geotagged Tweets, correlating the geotags with zip code and further the location's demographic makeup may provide a key insight for sociolinguistic studies. In text data, marking individual words with their frequency ranking from a published corpus will likewise prove useful. These are all instances of merging information from an external source. There are a few important considerations.

First, external data may contain inaccuracies or spurious entries that could then become a permanent part of your data set. In a memorable episode in my own research group, tokens of *also* were found to have *conjurer* listed as the lemma in our augmented data set; an investigation tracked down the source as the single errant line in the popular Someya lemma list[14] seen in figure 6.3.

This anecdote illustrates perils of blindly trusting external data; one must always carefully vet their sources and then scrutinize the end result of incorporation.

Second, merging from external sources is rarely a problem-free, one-step procedure and will require some, often extensive, measure of postprocessing. Examples include cases that are not covered by the external data source, subtle categorical mismatches and inconsistencies, and edge cases. They will need to be resolved in some way with clear documentation. The quality of attention paid in this postprocessing stage often makes

```
2626 conifer -> conifers
2627 conjecture -> conjectures,conjecturing,conjectured
2628 conjunction -> conjunctions
2629 conjure -> conjures,conjuring,conjured
2630 conjurer -> conjurers,also,spelled
2631 conk -> conks,conking,conked
```

Figure 6.3
An illustration of an error lurking in a widely used external data source.

or breaks the usefulness of the knowledge being added. Lastly, if publication of your data set is the ultimate goal, special attention needs to be paid to the copyright and licensing of the external data set, as they may have an implication on the publication of the incorporated data portions (see Collister, chapter 9, this volume).

Beyond merging from a static data resource, adding basic types of linguistic interpretation obtained through the use of text processing applications is another way of augmenting textual data. Sentence-level or word-level tokenization, lemmatization, and part-of-speech tagging are among popular steps (Lu 2014). Certain languages have a fairly well-established tokenization and part-of-speech conventions adopted by scholarly communities, which tend to coincide with the availability of NLP tools that are known to produce highly accurate automated output. The level of accuracy will vary depending on the task at hand: for English, word tokenization is something that is typically done in a deterministic fashion with high regularity and accuracy, while part-of-speech tagging accuracy currently tops out at 97.85%,[15] which some may argue crosses the threshold of being good enough to replace human annotation whose reported interannotator agreement rate is around 97% (Manning 2011).

But are these reported figures truly representative? With the exception of tokenization for certain languages, virtually every modern NLP tool is based on stochastic models trained and evaluated on the same pool of often highly homogeneous text; the reported performance figure is therefore liable to drop, sometimes significantly, when applied to out-of-domain text. Consequently, rather than taking a reported performance figure at face value, researchers would do well to conduct their own evaluation of how an NLP tool actually fares on their own text. This can be achieved by manually inspecting a small sample portion of the processed output. Additionally, conducting an output-wide sweep will help discover data-specific weak spots that the application is particularly ill-equipped to handle, which then can be fixed through postprocessing. In other words, in augmenting data through the use of NLP applications the same set of considerations apply as before, which is to thoroughly test the resource itself and be prepared to apply rigorous postprocessing and correction. Omitting these steps will invariably lead to noisy output that adds little value, or worse, a layer of interpreted linguistic knowledge whose lack of reliability ends up weakening subsequent linguistic research founded on it.

If these were the stakes, why go to the trouble of creating these new interpreted layers and publishing them? There are many benefits. First of all, certain earlier text processing tasks in the pipeline such as tokenization are viewed as essential: a running piece of raw text is nothing but a sequence of characters, and breaking it down into discrete sentences and/or word tokens is considered a foundational early step on which all later, higher-level linguistic inquiries will rest. A layer of word tokens published along with the original, raw text will therefore not only save time on the part of end users but also ensure all subsequent linguistic analyses build on common ground. All in all, a layer of automatically generated linguistic interpretation can be a true value-add when the output quality is sufficiently high, results are vetted through validation, and anomalies are corrected through postprocessing to the extent possible. One final and critical requirement: the entire process and the evaluation outcome must be clearly documented, so that future users of the data set can make informed decisions.

4.6 Manual annotation

The need for manual annotation arises in a couple of different settings. First, data samples collected for a personal research project may require further linguistic interpretation in the form of categorical judgment. For

example, sentence elicitations may need to be supplemented by a judgment on the speech act they encode. This type of annotation effort is referred to as *coding* in some research communities and is typically undertaken on a small scale by the individual researcher who is the creator and end user of the produced linguistic interpretation. A second scenario is more institutional: a manual annotation project is undertaken with an express purpose of building and publishing a large-scale linguistic data set. This type of institutional annotation project typically involves a team of linguists who may take on varying roles, a system of quality-checking in place for interannotator agreement, creation of annotation guidelines, a managed project plan and a budget, and a specialized annotation software platform. In either scenario, the process leads to *creation* of linguistic data, rather than transformation thereof, and therefore is outside of the scope of this chapter. Interested readers should consult Ide and Pustejovsky (2017).

5 Essential tools

This section provides an overview of essential tools that are commonly employed in data transformation projects. Perhaps unsurprisingly, they comprise core computational skill sets as well as the platforms popularized by the recently emerging field of data science.

5.1 Command-line tools

As illustrated throughout this chapter, command-line Unix utilities are essential in facilitating automation of certain transformation tasks, especially those that involve format conversion operations. They are readily available on a Mac OS: the Mac OS is based on Unix and thusly supplies the command-line interface and the Bash shell (lately the Z shell) natively through its Terminal app. On the Windows side, its native command-line environment called Command Prompt is entirely separate from the Unix-based suite of tools. However, recent developments have made Unix tools easily accessible in the Microsoft Windows platform. First, Windows 10 now lets users install and run a copy of Linux natively within Windows via Windows Subsystem for Linux. Second, Git's Windows installation includes Git Bash, a Bash command-line emulation layer, as well as most of the popular Unix command-line tools we have seen previously; this path comes with a minimal overhead and therefore is highly recommended for beginners and experienced users alike. Additionally, there are a number of non-Unix applications that are nevertheless designed to be used in a command-line environment such as Pandoc and, on the speech file side, SoX.[16]

Making the leap from the familiar graphical user interface into a command-line environment can be disorienting, but there are an ever-increasing number of learning resources online: The Unix Shell by Software Carpentry[17] (Devenyi et al. 2019) is particularly well received. Lu (2014) has an excellent chapter on how to process text in the command-line interface with detailed illustration and use cases tailored specifically for corpus linguistic research. Poser (2018) has been maintaining an extensive list of computational resources useful for linguistic research, which is found online.[18]

5.2 End-to-end processing tools

Operations that reach deep into the content of data, such as data cleaning, merging, and text processing, are best approached by enlisting more versatile computational tools such as Python and R. Python is a general-purpose programming language that has recently become the dominant platform of choice in NLP and, more broadly, artificial intelligence. R on the other hand has been a tool of choice for statistics-oriented research that has gained a wide adoption among linguists. They are capable of directly handling common formats such as HTML/XML, CSV, SQL, JSON, and even PDFs. Each boasts a large user base, active development communities, and open-source libraries for essential text transformation tasks. Python in particular was aided by a prominent library called Natural Language Toolkit (Bird, Loper, & Klein 2009) in emerging as the dominant platform for conducting primary text processing steps such as tokenization and part-of-speech tagging. The recent rise of the Pandas library has reinforced Python's handling of tabular data, traditionally a stronghold of R. Many text processing operations are also available in R through its open-source libraries. Using these programming languages, data transformation can be conducted in a streamlined, end-to-end workflow while maintaining a finer control over the outcome of each processing step.

5.3 Version control and documentation

If there is one cardinal rule in data manipulation, it is that every transformation step must be reversible. Version

control systems are indispensable in this regard: they serve as your safeguard for recovery as well as a powerful tool for managing the entire data processing pipeline while keeping a bird's-eye view over the history of a transformation cycle. Initially designed as a system for administering software development, version control systems have since emerged as a principal tool of trade for data practitioners. The most popular platform by far is Git,[19] an open-source project that is seeing universal adoption. In a personal project, it operates as a pure versioning system on a folder and its file content, called *repository*. In a multiuser project, Git additionally acts as a collaboration platform, as it offers solutions for managing file contents contributed by multiple users. Git forces users to structure development of a project into a succession of version commits, while providing tools to navigate and document changes throughout the version history. Becoming a successful user of Git entails breaking from one's old ingrained work habits and adopting an entirely new mindset on managing a project.

Extending Git's core capability is GitHub,[20] a popular repository hosting service. Keeping an online copy of a repository as the master copy serves multiple purposes: first is a data backup, and the second is a venue for publication. Lately, data development efforts as well as their resulting data sets are publicly hosted and distributed on GitHub. Data-focused research is increasingly finding a public-facing project home on GitHub, as do research projects small and large, including those that draw participants from across the globe. One such example is the Universal Dependencies Project[21] (Nivre et al. 2017), whose aim is to construct a data set of massively cross-linguistic syntactic annotation: it has over two hundred contributors producing more than one hundred treebanks in over seventy languages. In this connected era of the Internet, running a data-focused project out of GitHub is becoming a primary vehicle for keeping up the visibility of its public profile.

6 Guarding against data corruption

When operating on data, it is all too easy for unintended errors to slip in. Instances of data corruption lead to a waste of valuable time and resources required for correction. Worse yet, they may go unnoticed for a long period of time; a belated discovery may threaten to invalidate research findings that have been produced since. Having a versioning system in place is an absolute minimum that will enable recovery, but this is helpful only in the event when an error gets discovered and does so before its publication. Prevention of corruption in the first place is therefore paramount. This section covers common pitfalls and recommended strategies for combating the threat of corruption and ensuring integrity of data.

6.1 Accidental corruption: Humans and applications

One of the common sources of data corruption has to do with the realities of modern computing: textual data displayed for the human eye within a given software platform is fundamentally a rendered view. What looks like an ordinary stretch of text may be hiding invisible control characters and formatting-related tags, which may then get copied along with the genuine text content when an unsuspecting researcher goes for copy-and-paste. In addition, two characters that look alike to the naked eye may in fact be two entirely different ones, which then introduces a spurious distinction. Case in point: an ill-considered copy-paste job may introduce *co—worker* as a token that will then be interpreted by text processing tools as entirely distinct from *co-worker*.

Another, related, danger comes from the fact that software platforms will invariably apply their own rendering and transformation in the course of opening a file, which then may inadvertently be saved out, overwriting the original. Microsoft Word's conversion of plain ASCII quotation markers (") into *smart* quotes (“or”) is a well-known issue; opening a CSV file in Python's Pandas library may convert integer values (e.g., 1, 2, 3) into real-valued ones (e.g., 1.0, 2.0, 3.0) unless proper configuration is specified. Pandas also may apply its own conversion: on the Windows platform, it is known to auto-convert Unix-style line ending “\n” into Windows-style “\r\n” when dealing with textual data types. Your copy of Excel, unless it was patched in recent months, may insert an invisible character “\uFEFF” at the beginning of a CSV file as a BOM. The list goes on.

Human error is another common cause. An errant keystroke slipping into a data file is distressingly easy, so is replacing or deleting a file copy entirely. A vital practice in this regard is establishing an operating procedure for *viewing* file content as entirely separate from one for *editing*. The moment you open up a data file in edit-enabled applications such as Microsoft Word, Excel, and even text editor programs, you risk unintentionally modifying the

data. Many text-based data formats such as plain text (.txt extension), CSV, and HTML can be opened through popular browsers such as Chrome and Safari, which limit users to the viewing mode. The command-line interface also offers a good mode of operation for viewing and exploring data content. Interacting with text data in a command-line environment through standard Unix tools such as `less`, `cat`, `grep`, and other input/output piping methods will afford the user an added benefit: the power to sift through large quantities of data on the fly.

6.2 Corruption introduced during transformation operations

The last source of corruption is arguably the most intractable: inadequately managed data transformation efforts conducted on a mass scale. For example, an operation that is meant to anonymize personal names in a corpus may end up overapplying and removing homonymous words such as *April*, *frank*, and *park* across the board. Moreover, insufficiently restricted conversion rules may end up matching strings mid-word, changing, say, "Parking is a problem" into "<ANONYMOUS_NAME>ing is a problem." These are essentially a transformation operation applying where it should not, that is, a case of poor *precision*. These examples may seem contrived, but linguists, many of whom by training tend to focus on salient linguistic exemplars, are liable to underestimate the long tails of edge cases that make up a huge bulk in naturally occurring language data. To make matters worse, the sheer volume of data that likely had necessitated these automations would rule out manual inspection of the output and may ultimately lead to unexpected errors going undiscovered.

If regulating a vast number of positive hits is a difficult task, harder still is keeping tabs on a transformation operation failing to apply where it should, which is a matter of *recall*. When a transformation does not apply across the board to all intended targets, this too can be seen as type of data corruption, as the end result is unevenly applied transformation and therefore increased noise in the data. Going back to the anonymization example, in a vast amount of text data, how does one make sure that all personal names have been correctly detected? There is no easy and sure-fire way to ensure this, especially if the text contains non-standard capitalization and other irregularities.

The only defense against transformation-induced data corruption is a practice of rigorous postevaluation. The effect of a given transformation operation may not be vetted individually across all instances, but as a principle a methodical probing procedure should be put in place that rounds up frequent transformation cases along with some spot-checks performed on edge cases. An experienced practitioner of data manipulation will know to devise a comprehensive set of test cases that is intended to keep both the precision and recall of the transformation close to 100%, which then should be supplemented by exhaustive testing on the output. In addition, the practice of sustained monitoring through routine exploratory data analysis goes a long way: compiling data set–wide statistics on several key data points and actively monitoring them will help expose any anomalies that were introduced via an earlier processing step.

Lastly but most importantly: documentation is key. All aspects of a transformation task—the application, function, or code that was used; the target and its scope; expected and actual outcomes; quality control steps taken; postprocessing steps taken; and everything else—should be noted and recorded as part of a version control history. This is not only an instrument for troubleshooting any case of corruption discovered down the line but also a way of inculcating sound working habits in the daily workflow of a data practitioner. Eventually, the key details of the transformation operations should accompany the data set at the time of publication.

6.3 Warding off corruption: Summary of strategies

To sum up, this is a list of ground principles for warding off data corruption.

1. A version control system enables recovery and therefore is an absolute necessity.
2. Data corruption instances tend to escape immediate notice; prevention therefore is paramount.
3. Understand that all software platforms apply their own rendering and transformation to data files they open; each additional application in a work pipeline will introduce another risk factor that will have to be managed.
4. Beware of what is visible within an application's view; underlying code representation may be entirely different or hiding extraneous code bits.
5. Stem off accidental modification by separating file viewing methods from editing procedures.

6. Mass transformation operations, especially those based on pattern-matching, should be executed with care, with a tight control on both precision and recall.
7. Every transformation operation should be accompanied by methodical postevaluation as well as spot-checks.
8. Employ exploratory data analysis at regular intervals to ensure integrity of your data as they move through each stage of transformation.
9. Keep meticulous documentation of these processing steps.

7 Conclusion

Transforming data is often necessary and crucial part of a data workflow, either in a personal research context or in a more concerted data set production effort. As data proliferate in our daily lives, linguists have access to ever-growing volumes of text whose transformation must rely on automation and computational tools. Transforming data at these elevated scales brings the risk of data corruption, which can be contained through adoption of rigorous data practices with an emphasis on quality control. All in all, there is an art to transforming data, which comprises specific skill sets as well as work habits and practices that could take years to develop. Fortunately, educational opportunities on computational and data-science methods have become abundant as of late: linguists who set their eyes on data-centric research should not delay their training.

Notes

1. Thieberger and Berez (2012) present a comprehensive overview of linguistic data management from the perspective of documentary linguistics.

2. Wikipedia.org (https://en.wikipedia.org/wiki/Data_science, accessed November 19, 2019) defines *data science* as "a multidisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from structured and unstructured data."

3. Detailed information on these file formats are presented in section 4.1.

4. An example: a lexical resource may utilize italics or boldface to mark a token as the citation form, head word, and so forth. Transformation to plain text will erase all such meaningful distinctions; a markup text format, such as HTML or XML, that can store stylistic information therefore will be a better candidate as the new format.

5. https://notepad-plus-plus.org/.

6. https://www.sublimetext.com/.

7. https://atom.io/.

8. https://pandoc.org/.

9. http://www.unicode.org/versions/Unicode12.0.0/UnicodeStandard-12.0.pdf.

10. https://unicode.org/faq/utf_bom.html#BOM.

11. https://gitforwindows.org/.

12. https://docs.microsoft.com/en-us/windows/wsl/about.

13. Increasingly, concerted efforts are being made in the linguistics community with the aim to promote recognition of data creation and publication work as an important part of scholarship. Alperin et al. (chapter 13, this volume) make a case, as do Champieux and Coates (chapter 12, this volume).

14. Available at https://www.laurenceanthony.net/software/antconc/.

15. Akbik, Blythe, and Vollgraf (2018), according to https://aclweb.org/aclwiki/POS_Tagging_(State_of_the_art).

16. http://sox.sourceforge.net/.

17. https://swcarpentry.github.io/shell-novice/.

18. http://billposer.org/Linguistics/Computation/Resources.html.

19. https://git-scm.com/.

20. https://github.com.

21. https://universaldependencies.org/, GitHub home at https://github.com/UniversalDependencies.

References

Akbik, Alan, Duncan Blythe, and Roland Vollgraf. 2018. Contextual string embeddings for sequence labeling. In *Proceedings of the 27th International Conference on Computational Linguistics*, 1638–1629.

Austin, Peter K. 2006. Data and language documentation. In *Essentials of Language Documentation*, ed. Jost Gippert, Nikolaus Himmelmann, and Ulrike Mosel, 87–112. Berlin: Mouton de Gruyter.

Bird, Steven, Edward Loper, and Ewan Klein. 2009. *Natural Language Processing with Python*. Sebastopol, CA: O'Reilly Media.

Bird, Steven, and Gary Simmons. 2003. Seven dimensions of portability for language documentation and description. *Language* 79:557–582.

Corti, Louise, Veerle Van den Eynden, Libby Bishop, and Matthew Woollard. 2019. *Managing and Sharing Research Data: A Guide to Good Practice*. London: Sage.

Devenyi, Gabriel A., Gerard Capes, Colin Morris, and Will Pitchers, eds. 2019. swcarpentry/shell-novice: Software Carpentry: the UNIX shell, June 2019 (Version v2019.06.1). Zenodo. http://doi.org/10.5281/zenodo.3266823.

Ide, Nancy, and Pustejovsky, James, eds. 2017. *Handbook of Linguistic Annotation*. Houten, the Netherlands: Springer Netherlands. http://doi:10.1007/978-94-024-0881-2.

Lu, Xiaofei. 2014. *Computational Methods for Corpus Annotation and Analysis*. Houten, the Netherlands: Springer Netherlands. http://doi:10.1007/978-94-017-8645-4.

Manning, Christopher. 2011. Part-of-speech tagging from 97% to 100%: Is it time for some linguistics? In *Proceedings of the 12th International Conference on Computational Linguistics and Intelligent Text Processing*, vol. 1, ed. Alexander Gelbukh, 171–189. Berlin: Springer-Verlag.

Nivre, Joakim, Daniel Zeman, Filip Ginter, and Francis Tyers. 2017. Universal dependencies. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Tutorial Abstracts*, E17-5001. Stroudsburg, PA: Association for Computational Linguistics.

Poser, Bill. 2018. Computational resources for linguistic research. Accessed version: last revised September 10, 2018. http://billposer.org/Linguistics/Computation/Resources.html.

POS Tagging (State of the art). n.d. The ACL Wiki, the Association for Computational Linguistics (website). Modified March 4, 2019. https://aclweb.org/aclwiki/POS_Tagging_(State_of_the_art).

Thieberger, Nicholas, and Andrea Berez. 2012. Linguistic data management. In *The Oxford Handbook of Linguistic Fieldwork*, ed. Nicholas Thieberger, 90–118. Oxford: Oxford University Press.

Unicode Consortium. 2019. The Unicode® Standard Version 12.1.0–Core Specification. https://unicode.org/versions/Unicode12.1.0/.