Pragmatics of Pedagogical Datasets

This guide is a collection of issues, opportunities, and examples for developing Pedagogical Datasets.

I have organized and summarized the most salient issues for future developers into 6 categories.

1. **General Advice**: Issues to keep in mind throughout the development of datasets, but particularly when beginning work with a new dataset.
2. **Collecting Data**: Issues that come up when data is being collected from external sources.
3. **Restructuring Data**: Issues related to changing the structure or format of data, without respect to the values themselves.
4. **Manipulating Data**: Issues related to the values within the dataset, as opposed to the structure encapsulating the data.
5. **Working with Data Types**: Issues related specifically to the various types of data, and their possible schema.
6. **Knowing Data**: Issues related to students' and designers' comprehension of the dataset that they are working with.

1. General Advice

The following issues and ideas should be kept in mind throughout the process. Much of this advice holds true for any software engineering project. However, this does not make it less valuable to review, especially since in some cases conventional wisdom may not hold true.

1.1. Have a plan:

Datasets are organized for a number of reasons, such as to facilitate manipulation, to conserve space, to facilitate access across certain indexes or specific paths. Typically, the developer has the intended purpose of the dataset in mind, and this can guide the process. Pedagogical datasets can be organized for specific problems or for open-ended exploration. The former, where an instructor has problems and activities already in mind, can be much easier to develop for, since the instructor will have particular characteristics in mind. The latter, where students will explore the dataset and derive their own research questions, is more difficult to design.

When designing open-ended datasets, the developer needs to provide a range of fields and values that can be used to explore many kinds of questions related to the dataset. I recommend a number of higher order strategies to make the students' experience as motivating as possible. First, try to intentionally lay out multiple obvious possible paths: imagine that you were using the dataset, and what kinds of questions you could conceivably want to answer with the data. Second, try to keep the

paths diverse, so that students can explore different directions. Third, allow for serendipity — you may not have an immediate use case for some data that you find, but if the quality is high, consider including it anyway. Finally, accept that not every potential use-case will be covered — students may wish you had collected some other kind of data, but you will never be able to predict all possible student preferences.

Example: Crime Data A primary path of questioning to design the State Crime dataset around is to facilitate research into the trend of crimes over time; however, with the right data collected, the dataset could be used to answer more wide-reaching questions related to public policy, enforcement strategies, and population habits. Incorporating the right statistics to allow students to explore all of these issues might make the dataset too big, but choosing a reasonable subset can make the dataset more widely appealing.

1.2. Build for your audience:

Typically, datasets are intentionally cleaned and organized so that they are as easy-to-use as possible. However, pedagogical datasets can be designed to provide structured difficulties, representing learning opportunities for novices to encounter specific classes of problems. The designer must consider the learning objectives and prior skills of their audience. In general, the goal is to remove all difficulties that are not explicitly related to the intended learning goals. For example, if a dataset incorporates null values for some observations, students need to know how to use conditionals to avoid plotting the null values — if the original lesson was simply about plotting lists of numbers, the learning objectives have suddenly been expanded (similar to the software engineering phenomenon of feature creep). It is important to choose your learners' battles carefully, so that they struggle with the right aspects. It can be tempting to leave a flaw in the dataset and casually wave it off as "a learning experience", but this must be done with the learners' best interests in mind.

1.3. Iterate:

As with any design project, you should work in an iterative manner. Expect to refine or discard earlier attempts. Decisions that seem reasonable early on may have surprising consequences when the results are brought to a learner. It is valuable to expose early iterations to expected users to gauge reaction.

1.4. Standardize your process:

Never treat code for preprocessing a dataset as single-use; plan early to make it a process that can be run repeatedly. Almost no dataset is final — new data is published, errors are found, or possible improvements are discovered. Structure the preprocessing as a pipeline; ideally include code to automate the retrieval, to clean the data, and to generate the completed dataset. At any time, you should be able to regenerate the completed datasets almost entirely automatically.

As with any coding project, include whatever documentation would be needed to explain the code. Document the source of data, both direct links and the pages that contain them (the former for convenience, the latter for redundancy and future-proofing). Keep track of citations and data documentation wherever possible, since information that seems obvious early on can become more obscure with time. Use assertions to verify expectations about the dataset. Make stylistic decisions and stick to them, so that it is easy to pick-up old build pipelines and navigate them.

1.5. Keep a clean workspace:

Verson Control Systems like GitHub can be a great asset in managing preprocessing code. However, many of these systems frown on keeping large data files around. Rather than driving up repository size by including all data files, consider incorporating rules into your repository (e.g., through .gitignore systems) that prevent storing data files (e.g., CSV, JSON, TXT, etc.). Then, add in code to automatically retrieve any data files on demand from a remote system and keep that data cached. This cache, along with intermediate and output files, should be removable as part of the build process. A self-hosted mirror of the original data source can provide safe redundancy, while also maintaining the separation of the processing and the data.

1.6. Manage dataset health:

Datasets are never perfect: Users will encounter mistakes and errors; Better structure and organization is realized; New data becomes available; Old data becomes obsolete; Better data is found. You need to plan early to track and manage these changes. Using an issue tracker (such as the one on systems like GitHub) to manage problems with the datasets is an excellent way to stay on top of these changes and to provide a record of the design decisions. It can be difficult to

remember the justifications for decisions that were made early-on, but issue trackers can be used to find the original reasoning.

1.7. Beware breaking convention:

A recurring danger when scaffolding a dataset is that students may inadvertently learn incorrect behavior. They may believe that the instructor has created the dataset with conventional best practices in mind, and they will (possibly subconsciously) adopt these strategies. Instructors should work with interested students to better understand why a dataset was organized the way it was, and other decisions that could have been made. This can be a teachable moment: consider an assignment where students work to improve a dataset for some purpose. Otherwise, the instructor should follow best practices wherever possible, assuming it does not conflict with managing students' struggles.

1.8. Work in phases:

Processing large data files can be time consuming, simply by the nature of data at scale. However, as with any coding process, the development process will involve errors and debugging time. If a later step fails, then resuming all the way from the beginning can greatly hamper development as long delays prevent proper debugging. Structure your build pipeline so that phases can be developed in chunks, and individual phases can be debugged independently. For example, by writing out intermediate results to disk, progress can pick up from where it left off.

1.9. Understand the context:

Datasets are typically generated for some purpose; studying more about that purpose can greatly aid in the development. Websites often publish complementary data dictionaries and information about datasets that can explain nuances. Sometimes, publications have been written that can better explain the context. Find and reach out to subject-matter experts in order to resolve questions. Ensure that the dataset you end up preparing aligns with the kind of data that professionals might expect to see, at least in nature if not shape or format. Generally, you should do research on the context before you collect and prepare the data.

2. Collecting data

The first step in preparing a dataset is obtaining the dataset. This section details issues and ideas related to finding, mining, generating, or otherwise obtaining datasets.

2.1. Hunting sources:

Because data is pervasive across the internet, the problem of obtaining a data source is largely one of filtering and searching. Search engines are an obvious first step in finding suitable datasets. However, knowing the capabilities and available techniques of your search engine can be invaluable. For instance, Google features a number of powerful ways to improve searches, including boolean logic, site-specific searches, and even a filetype search. This last feature has proven particularly useful, since we are typically interested in data files of certain formats. Knowing how to navigate search results effectively can also be helpful — recognizing what sites are particularly promising can help filter datasets more quickly. For instance, repository sites (e.g., GitHub), government sites, and non-profit organizations are often good places to start.

In the past few years, data enthusiasts have collected varied data sources into meta lists. Some meta lists are more useful than others. In my experience, however, many of these lists have considerable overlap, making it difficult to find new sources. Further, their datasets tend to be limited in size and scope, typically narrowed for a specific purpose. This is particularly true for datasets that are meant to teach specific subjects. The UCI Machine Learning Repository features a tremendous archive of Pedagogical Datasets, but the datasets tend to only contain a few limited features targeted for specific machine learning tasks (which can be too arcane or complex for beginners to tackle). Many datasets are created by and therefore tend to appeal to people who naturally find data interesting and useful — therefore, those tend to be about computational subjects that may have little cross-discipline appeal. Of course, there are still many excellent resources for finding datasets in meta-lists: for instance, the DataIsPlural project publishes novel, high-quality datasets twice a month , and a number of its datasets have been incorporated into the CORGIS project.

2.2. Working with file formats:

Data can come in many exotic formats, which can be troublesome to deal with. When collecting data, it is important to be aware of the various formats you might encounter. It is not possible to make an exhaustive list, but specific and general classes of data formats include the following:

- Spreadsheet data, in a row-column format, is one of the most conventional and stereotypical data formats. CSV (Comma-separated Values) and TSV (Tab-separated values) are common examples of this format. These data formats are usually easy to manipulate and

view in both text editors and spreadsheet software such as Microsoft Excel and Google Sheets.

- JSON is a hierarchical format that incorporates lists, objects, and four primitive types: integer, floats, strings, and booleans. I chose JSON as the designated format for completed datasets, since it allowed for considerable flexibility, expressiveness, and simplicity. There are variations on the JSON format, such as the JSON-line format, where every line of the file represents a parseable JSON object (removing the need to encapsulate the top-level structure as a list).
- XML data (and HTML data by extension) are document markup formats, not strictly intended for structuring data. Although popular in many industry settings, for most purposes, XML is a less efficient file format than JSON , in terms of space, transmission rate, and complexity. In my experience, processing datasets in XML is cumbersome and tedious compared to other data formats.
- Many popular statistics packages have their own proprietary formats: STATA and SAS, for instance, both have their own file types. These formats require specialized software, and can be inconvenient if the developer is not familiar with them. Typically, I tried to avoid these types if possible.
- PDF is a popular format for transporting documents reliably, and is sometimes used to publish data. Data stored in the PDF format is a tragedy, frozen in time and locked away from convenient access. Although PDF can be a useful format for conveying information to be reviewed by humans, it is highly problematic for computational purposes. Tools exist for parsing PDFs and extracting out their data, but this is extremely difficult, and usually not worth it. Typically, even the best tools introduce garbage data, fail to understand formatting and table structures, and require human effort to correct mistakes.

2.3. Scraping web data:

Not every website that collects data makes it easily available. In some cases, *scraping websites* (visiting pages using a script and downloading their contents) can be a great way to make your own dataset. In general, scraping can be considered abusive behavior, and some websites respond aggressively: limiting user access, banning IP addresses, and other defensive actions can greatly hamper your ability to access the site. It is critical to make sure that you are not violating the Terms of Service of the website you are scraping. You should also check carefully that the website does not have their data available in some other, more convenient way. You might even consider requesting the data directly, if contact information is available on the site. If scraping is the only option, then do so politely: use non-peak hours, spread out your scraping to reduce server load, and cache intelligently to reduce redundant server trips.

2.4. Mining real-time data:

Mining a real-time data source is an easy way to translate a high-velocity, low-volume dataset into a low-velocity, high-volume dataset. The idea is to take a data source that updates regularly and to retrieve data from it on a consistent schedule, aggregating the data over time. This can be particularly useful for services that aggressively limit users' access rates. Mining requires time, planning, and coordination. It is a good idea to keep tabs on your data collection, in case the API service changes. Logs can help identify problems, especially if notification systems (e.g., email alerts) are used.

2.5. Legality of your data:

Some datasets make it obvious whether you are allowed to republish and use data, but sometimes it can be ambiguous. Keep the legality of your data in mind, and investigate the allowed uses before collecting the data. Typically, fair use policies are generous towards using data for educational purposes. When in doubt, reach out and request permission to use the data.

Remember that some datasets might need release forms from every individual student that intends to use it. The ICPSR (Inter-university Consortium for Political and Social Research) is a high-quality collection of datasets from the humanities and social sciences. Unfortunately, the vast majority of its data requires users to individually obtain release forms, since much of the data involves human subjects. In our experience, we have avoided working with ICPSR and other services that have these requirements, since it adds additional complexity disproportionate to our learning objectives.

2.6. Synthesizing datasets:

Many datasets can be improved and expanded by incorporating additional data from other datasets. These kind of conjoined datasets are popularly known as "Mash-ups". It can be tricky to find data with enough common indexes to perform the join operation. An example of a mashup would be to take state/year level data (such as for scholastic abilities in US states, or crime statistics) and to integrate information about population, income, or some other kind of metric. This sort of process is most appropriate with script support to recreate the mashup.

3. Restructuring data

Once data has been collected, it must be structured in a way that aligns with the projects the instructor has in mind. The following entries describe common operations performed when manipulating data. In general, we recommend using a scripting language (e.g,. Python, Perl) that

you are comfortable with in order to do any data processing. Many tools and guides already exist to aid in the process of manipulating data, so I consider reviewing low-level language details to be outside of the scope of this dissertation. Instead, I have tried to only include advice at a high level.

3.1. Choose your target structure:

For the CORGIS collection, every dataset is eventually represented as a list, where each element is a dictionary of either primitive fields or further dictionaries. This format was chosen since it allowed convenient filtering and sampling of elements while still providing opportunities to practice complex data traversal (both of heterogenous and homogenous data). There are many options available for the structure of data, however. In some cases, you may wish to allow the data to dictate its own structure, based on inherent characteristics. In other cases, you may want to plan a structure that will be easy to access using the students' intended learning environment. If preparing multiple datasets, maintain consistent structure across datasets, so that instructors can switch between datasets more quickly and easily.

3.1. Layering columnar data:

Restructuring datasets into list-of-dictionary form sometimes begins with spreadsheet data. The row/column format of spreadsheets is similar to the concept of a list-of-dictionaries, with the exception that dictionaries can contain other dictionaries whereas row/column typically holds singular primitive values. When working with spreadsheets that are particularly wide (i.e., have a lot of columns), chunking columns can be an excellent way to help users navigate the data's structure. These chunks of columns can represent sub-abstractions that group related fields. For example, an address can be grouped with latitude and longitude fields under a "location" field.

How big should a chunk be? A classic text in psychology says that the human memory is best at remembering 7 things in a chunk, plus or minus 2 . However, later research argues that the earlier estimate of 7 was based solely on remembering numeric sequences, and that other kinds of data have different optimal chunk sizes . For textual data (in this case, the names of the chunks which are keys of the dictionary), experts suggest smaller chunk sizes (4 has been suggested), but different experts draw on different factors to suggest a number, including the size of the words, how well-known the words are, and whether schemas can connect the words together. Ultimately, we aimed for chunks 4-5 keys wide, particularly at the top of our dictionary hierarchy, making allowances for the factors established above.

3.2. Converting XML to JSON:

As previously discussed, XML can be a cumbersome format, since it is meant for documents instead of data. This may be surprising, since technically XML can represent the same kinds of data as JSON (both are hierarchical data formats). There are many reasons why this is not as simple in practice, however. The syntax for XML requires more characters and rules than for JSON, for instance. Tools exist that can create a direct translation from XML into JSON - however, the results can be surprisingly garbled. One of the major reasons comes from XML's ability to annotate a node with text data, child nodes, and attributes. The comparable structure in JSON (dictionaries) expect to have either a single primitive value, a list, or a dictionary. Mapping between these two structures often requires some level of human oversight.

3.3. Working with indexes:

Datasets often have *natural keys* — attributes or sets of attributes that uniquely identify each instance in the dataset. Natural keys, unlike surrogate keys, have a natural existence within the dataset, and are not artificially generated for the dataset. The State Crime dataset, for instance, has two attributes that, together, form a natural key: state name and report year. In the CORGIS project, we always attempt to identify the attributes of natural keys as Indexes, as they are often useful for creating problems and mechanisms around the dataset. When sorting the data before release, we use the natural keys as sorting criteria.

Typically, we try to limit our datasets to have 2 indexes when building our natural key. When a dataset with two indexes is filtered on one of the indexes ("fixed), the remaining rows of the dataset are usually immediately useful for some purpose. Consider filtering the crime data by the state of Virginia — since the remaining index is report year, the remaining reports will represent all of the data for Virginia over time, requiring little effort to plot as a line graph.

Besides the advantage of 2-indexes simplifying the filtering of datasets, I have found that introductory students struggle with the implications of datasets that have more than 2 indexes. Earlier versions of our cancer dataset incorporated 5 distinct indexes (gender, race, state, year, and location of cancer on the body), which was often overwhelming for students. Students seem to find it difficult to consider what happens when an index is fixed, that the data that remains will be over the

other possible indexes. For the cancer dataset, if the state is fixed to be Virginia, it is still unsuitable to plot the remaining data as a line plot — all of the non-year indexes must be given fixed values.

3.4. Collapsing fields:

If you find your dataset is too big, you can consider collapsing on a field (also known as pivoting a table). That is, group each instance in the dataset by the different values of the collapsing field, and then use an aggregation function (sum, average, count, min, max, etc.) to flatten the groups into single values. This drops information, but does so in a consistent way and retains some of the original information. For example, the Airlines dataset (which gives information about flights at airports over the past decade) was several standard deviations larger than our other datasets, so we decided to collapse its Carrier field (which indicated which company controlled the flight). This substantially reduced the size of the dataset, at the cost of some fidelity of information.

There are often specialized ways to perform the grouping operation. For instance, increasing the time scale of data (daysmonths, monthsyears) or generalizing geographic region (countiesstates, citiescountries) can reduce the dataset dramatically. There are few general principles here, but instead careful application of domain knowledge is required.

A field might be promoted into an index by grouping values carefully, in a process known as "binning". Consider the Cars dataset, which has information about many cars over multiple years. Each year, there is usually more than one car present, and the year is not an index for the data. A possible transformation of this dataset would be to bin the cars for each year, presenting averaged data, which would make the year a suitable index for the data. Although for the Car dataset this would make the dataset prohibitively small, there are other cases where this might be preferable.

3.5. Stacking data:

In the opposite operation of collapsing data, *stacking data* means to extract out a series into its own column. This process removes embedded lists from inside a dataset. The CORGIS project avoids having nested lists (to reduce user complexity), so this technique was used quite often when designing datasets. Every time you stack data, however, you naturally introduce a new index, so consider the ramifications carefully. This can also be used to increase the number of records your dataset has, and by extension its size. Data that is fully stacked is often known as "Tidy Data" ,

representing how this form of structure can be particularly easy to work with (compared to nested lists).

3.6. Redundant total field:

A common occurrence in datasets is a set of a keys within a single dictionary that perfectly partition a whole. In the Airlines dataset, for example, the time that flights were delayed given in a dictionary as 5 categories, one for each of the possible causes of a delay. Summed, these fields represent the total time that flights were delayed. Although students could be tasked with calculating this number, they would be required to perform either an iteration over a heterogeneous collection (which might not be in the original learning objectives, the way a homogenous collection iteration would) or to write code that individually adds each of the keys. The latter is particularly bad for datasets with tremendously large numbers of keys, such as the Building Construction dataset (which has over 20 categories of construction types, represented as dictionary fields). A common solution to avoid this is to provide an extra field that represents the total of the other fields. However, this total field hampers the student from doing an iteration over the keys, since the result will be twice what it would otherwise be. Caution and careful consideration of learning objectives should guide the decision to include summation, or other aggregation, fields for convenience.

4. Manipulating the data

Beyond working with the structure of the data, the developer must also consider the contents of individual fields. This includes both the value and the key associated with that value. In this section, I detail a number of issues related to the massaging of the data before it is ready for use. Discussion about the Types used in creating data is largely reserved for the following section.

4.1. Standardize fields:

Be as consistent as possible across fields names, types, and their values. Ensure that every field name has the same style of capitalization, spelling, use of symbols, and punctuation, and make sure there are no errors in any of the above. Make sure that every instance of a field has the same type and uses the same kinds of units. If any of the data is nullified in some way, ensure that that nullification happens in a reasonable, consistent way. I strongly recommend using a consistent structure across objects, unless you explicitly want to teach students about how to deal with inconsistent structures. Automated tools should be used to supplement the process and verify that these conditions hold. Although using such tools to check the structure of the datasets is obvious, they can also be used to check other conditions, such as correct spelling. Develop a tangible list of

standards and conventions that can be referenced during development. This is useful not only for large teams, but for maintaining consistency across multi-dataset projects.

4.2. Names are important:

When developing fields for your dataset, meaningful names are critical. The field name is students' first impression of the data, and an important time to convey as much information as you can. Nobody reads documentation , and students are not exceptions — in fact, they are probably even bigger offenders of the rule. Just like designing user interfaces, documentation is not a substitute for bad design. When naming things, consider both fields and values: although the field is obvious, sometimes you will want to recode values to be more clear. For example, if a string value was given as an abbreviation, you might use the fully expanded phrase.

Names should be as long as necessary to convey all the needed information, but not longer. Names that are too long will force students to do more typing, make it more likely for typos to occur, and potentially increase frustration unnecessarily. Be careful when using abbreviations, especially if they are not common domain-specific ones — be sure to provide documentation if you must use an abbreviation. If possible, you can also use names to hint at units (e.g., instead of a "Time" field, you could have an "Hours" field).

Be careful with choices in language. It can be easy to introduce offensive or politically correct terminology, especially as a layman. When developing the Immigration dataset, which has information about legal and illegal immigrants entering the United States, the term "Alien" was used in some places to describe illegal immigrants; after a user raised concerns about the vocabulary, we chose to find more accurate and less offensive terminology. Proper research on the source context can help dramatically with navigating these kinds of issues, but in general it helps to simply be open to the possibility and to be responsive to user issues.

The mental errors that students make will not always be be obvious, although students do seem to be predisposed to find certain kinds of patterns in the data. The Publishers dataset features information about the top 10,000 books sold on Amazon, where each row represents a single book at a specific point in time. A number of properties for each book is recorded, including the daily number of books sold on Amazon. This field was named "Daily Books Sold", and encoded as an integer. Almost every student that interacted with the dataset developed the misconception that, because the word "Daily" was in the name, the data was indexed by time. Although the

documentation was quite explicit that the dataset was a snapshot of a collection of books at a point in time, students would attempt to plot the Daily Average; since the dataset was sorted by the book's rank in terms of sales, this would invariable lead to a relatively smooth downward curve. The conclusions and extrapolations that students made from this data are irrelevant, because they were fundamentally led astray by the misleading name.

4.3. Working with bad data:

Datasets are rarely perfect. In fact, more often than not they have errors in their data: typos, dropped fields, encoding errors, or simply non-existent data. As many of these imperfections should be *cleaned* as possible before they are given to students. Many different strategies abound, depending on the nature of the malformed data.

When data are missing, one option is to substitute a null value. Null data can be complex for students to handle: it can require more conditional checks, breaks some standard tools and operations, and needs more critical thinking to reason about. Different formats and languages have different ideas of non-existence. The JSON format provides a "null" type, which is often present in programming languages (`None` in Python, `null` in Java). Some languages support multiple kinds of non-existence, such as `undefined`, `null`, and `NaN`(Not a Number) in JavaScript. If these forms of non-existence are not available, you might replace the missing data with 0, 99999, or -1 or some other "Bad data" value. The added value is that the fields will be of the same type — the downside is that it becomes considerably less obvious that the data was incomplete, and learners might not realize what has happened. Learning objectives should be considered carefully to determine whether students should need to confront bad data, or whether it is suitable to ignore the problems for pedagogical purposes.

The process of substituting values for non-existent data is called *imputation*, and is a more sophisticated technique than using nullified values. Imputation can be performed in several different ways, each of which has trade-offs. The "last-observation-carried-forward" rule simply carries over the value from an adjacent instance in the natural ordering. Mean substitution replaces the missing value with an average from the entire dataset, or from a subset of instances that share a similar key. Although sometimes effective, mean substitution can become problematic when the fields being interpolated are independent from the indexes being considered. For example, in the Earthquake dataset, there is no relationship between the "Gap" metric and the datasets' natural keys; if missing "Gap" data was being interpolated using mean substitution, the values would be worse than

guesses, since it would introduce a non-existent relationship. The final form of imputation, and the most sophisticated, is to use regression to fill in missing values. Stochastic variations of regression can be used to incorporate noise into the dataset, increasing its feeling of authenticity. Although typically more accurate than other forms of imputation, regression suffers from the inverse problem that it can introduce new alternative realities: patterns become reinforced, even if they didn't really exist. The Global Development dataset's source material (the World Bank) uses linear interpolation to fill in years worth of data, leading to smooth gentle curves that belied the true nature of population data, and in some cases whitewashes tragedies and diseases in foreign countries. It is crucial to document the use of imputation for students, and communicate with the learners about the nature of the data.

Listwise deletion is a simple and direct method for handling bad data — deleting any instances that have bad data. This can have the side benefit of reducing the size and complexity of the dataset, if it is too large already. Consider carefully whether the data being removed are random or represent a specific case of interest. If possible, retain any important statistical relationships with a proper sampling algorithm that gets a random distribution, or intentionally picks values that maintain the relationship. Removing outliers to smooth data can lead to odd results, and needs to be approached with caution. In the CORGIS Immigration dataset, outliers were naively (and briefly) trimmed, leading to a complete halt of Mexico's immigration to the United States. Deletion has other variations beyond deleting just the instance: *pairwise deletion* removes all instances that share index values, for instance, and there is always the option of just removing a field across the dataset.

Removing or changing the data can be useful for avoiding complex situations. For example, earthquake magnitudes can be negative - but explaining this to students is an extra headache that may be more distraction than helpful as a context. Therefore, the Earthquake data filters out earthquakes smaller than magnitude 1. Although the documentation details this fact, it is likely that students would never realize this — which can be dangerous if they are relying on the dataset to be completely honest!

4.4. Cleaning up by hand:

Sometimes, automated tools are insufficient for cleaning up a dataset. When developing the Medal of Honor dataset, for instance, award dates and locations were often mangled and incomplete, which became a problem when geocoding was applied to the dataset. The strategy to overcome this limitation was to automate as many broad cases as possible, and to simplify the process of handling

exceptions as much as possible. This process needs to be formalized and repeatable; consider that it might need to be run multiple times, that it might need to be interrupted (whether because of a developer error or some other issue), or that it might need to be reviewed. One suitable approach is to have a step in the preprocessing pipeline write the data out to a spreadsheet or other convenient data format, for human annotation. This file can be updated in-place, without deleting existing annotations, and then read back in during a later step to fix the most exceptional cases. Mechanical Turks and undergraduate assistants can be powerful tools for fixing data by hand.

4.5. Reshaping data:

Visualizing data that is heavily skewed or has an alternative distribution can be difficult. For example, when creating histograms of data that is log-normal, the graph may appear to be completely skewed in one direction; after applying a logarithm function, however, the data becomes normally distributed. Data scientists will often transform data using other common mathematical functions, such as a Square-Root, in order to reshape the data into a more easily visualized distribution. For data that is already normal, it is also common to apply a standardization operation, which uses the mean and standard deviation to recenter the data around zero. Rather than expecting students to apply this operation themselves, a field can be preprocessed using these operations. Doing so requires the developer to explain the field further, both through its name and in the formal documentation; documenting and explaining these kinds of transformations can be challenging, however.

4.6. Extending a dataset with divined data:

Sometimes, a dataset can be extended without introducing any external data. The method leverages metadata and extrapolate information already contained within the dataset. A relatively simple case is to add a rank field to the dataset, if there is a natural ordering of instances. However, more sophisticated metrics can be added too. The Classics Dataset originally incorporated large texts of classic, public-domain books; however, it featured few numeric metrics on the books. A number of textual analysis techniques, including sentiment analysis, word count, and text difficulty estimations, were applied to expand the dataset.

5. Working with Data Types

Types are an important topic in computing and when creating datasets. The developer must be careful about choosing what types to incorporate into their database. What types do you want your students to be familiar with? Most datasets will incorporate at least some basic numeric types

(integer, decimal) and text types. It is also common to have boolean types. Some datasets benefit from having enumerated types. However, enumerated types exist as metadata, not necessarily as an inherent part of the structure. There are also the "Non-existence" types (`null`, `None`, `undefined`, `NaN`, etc.) that require further explanation.

The CORGIS collection limits its data types to integers, decimals, booleans, and string types (in addition to lists and dictionaries). However, even within this scope, there are more complex schemas and nuances to be captured. Specialization types, such as currencies, dates, and addresses, can be represented in many different ways, with trade-offs for each. In this section, I describe concerns and opportunities provided by these types and schemas.

5.1. Numbers:

In some ways, numbers are the quintessential type of data. Most formats and environments separate between integer and decimal (floating point) types, but there are other variations: some languages have special types dedicated to massive numbers (`long` in Java and Python), tiny numbers (`char` in Java), and decimal numbers with more or less precision (`double`, `single`, in Java).

The CORGIS project generates datasets ready for multiple languages and environments, which requires a lowest-common denominator approach. Therefore, the only two numeric types are float and integer. We have a mild recommendation to prefer integer numbers over floating point numbers, since decimal numbers can end up sacrificing some level of accuracy in certain environments. Further, it is advisable to keep the size of numbers manageable — it can be mentally difficult to keep track of numbers greater than 2^{64}, and some languages coerce these numbers to a special numeric type (as in Pyton 2.7) or, even worse, simply cut off the unused data (as in the formal JavaScript spec).

Percentages are common in many datasets. These are typically in the range of 0% to 100%, but not always. A small decision is required when encoding percentages: do you multiply it by 100, or leave it as a decimal number? That is, if you were representing a half, would it be 50 (50%) or .5? Whichever option you choose, you should be consistent and make it clear in the documentation. Given that either format can lead to confusion (is that 5000%? .5%?), there is no clear solution. In the CORGIS project, we use whole numbers to represent percentages.

5.2. Textual:

Textual data, often referred to as `string` data, comes with a host of headaches. Any data can be encoded as a string. However, students must then struggle with type conversions and string parsing,

which may or may not be relevant to the intended learning objectives. Review data carefully to make sure that no numeric fields have been accidentally encoded as strings — while this sounds like a silly mistake, we find it happens in almost every dataset at least once. For example, persistent bug reports with the Cars Dataset led to the discovery that several fields were accidentally encoded as strings for certain observations, because the units were included in certain strings (i.e. `15kg` instead of the integer `15`).

Text data opens up a host of lessons and opportunities to teach students about string parsing, manipulation, and other common operations. However, in more mathematically driven learning contexts (e.g., Data Science) where these skills might be considered distractions, it can be difficult to find a purpose for text data. Analyzing the text itself for metrics can provide numbers, assuming the text is of a sufficient quality and interest (e.g., a note-worthy book, forum posts with associated metadata). These metrics can be applied at multiple levels: individual characters, words, sentences, paragraphs, chapters, equally-sized chunks, or even on complete text blobs. The following is a brief description of computational textual analysis techniques that have been explored or applied in the CORGIS project. This list is biased towards techniques that work particularly well for English-language text. Not all of these metrics lead to particularly useful results.

- Sentiment Analysis: The positivity or negativity of a text, calculated by the frequency of certain words that have previously been tagged as positive or negative. Many popular packages exist off-the-shelf to perform this kind of analysis.
- Word and Sentence Statistics: Simple measures of the way that the text is written, such as number of characters, number of words, number of sentences, average length of word, average length of sentence, etc.
- Word Difficulty: An estimate of the difficulty of all the words in a text. Many formula and packages exist to calculate this, but the two general approaches rely on either the number of syllables in the text or using an index of word difficulty.
- Word Frequency: Word frequency analysis can be used to isolate the patterns of use of certain words, assuming a list of suitable words to mine can be created.
- Word Etymology: As shown in , the etymological source of words (what language they were imported from, such as German, French, Latin, etc.) can be estimated using language source dictionaries.
- Grammatical Mood Analysis: The English language expresses uncertainty through certain kinds of grammatical constructions. The frequency of these constructions can be used to classify text as "imperative", "indicative", "subjunctive", or "conditional" .
- Vocabulary Richness: The diversity of words in a text, also known as Yules I , which the original creator argues is a measure of complexity.
- Tonal Analysis: Advances in deep learning have led to tools such as IBM Watson, which features an application to analyze a text for emotional content (anger, sadness, joy, disgust,

fear), language style (analytical, confident, tentative), and social tendencies (openness, conscientiousness, extraversion, agreeableness, emotional range).

When preparing a text field, check strings carefully to make sure that special characters are correctly escaped and decoded, according to the expected use of the field. Data stored in Unicode encodings presents a particular challenge. Not every language, format, and platform has equivalent support for Unicode. Further, proper handling of Unicode is tricky without proper understanding of the mechanisms involved in bit-level string representation. In some cases, it may be simplest to simply ignore the problem of Unicode data by mapping characters to their nearest ASCII equivalents. Although this is the approach taken by the CORGIS project, it is most likely unsatisfactory for most serious text datasets. I recommend reviewing more substantive literature for any project interested in working with Unicode data correctly.

5.3. Dates and times:

Date and Time data is common in most datasets, since it opens up a wide range of interesting questions and problems. However, encoding and structuring such data is a difficult problem. In some cases it is useful to provide multiple, alternative representations of the value, but this increases file size and the total cognitive load needed to traverse the dataset. Although documentation can supplement the process, we have previously noted that documentation is not well-used by novices or experts.

Including a simple text representation of a a date/time can make it easier to immediately comprehend the value. If this text representation is structured correctly, it can also simplify sorting datasets on time. For instance, writing out a date in the format of "Year/Month/Day", using zeros to pad each number to a fixed width, makes the dataset easily sortable. Unfortunately, these formats may impact the readability of the value, since the zeros contribute noise and the format is not necessarily one most people are familiar with. Whatever format is used, it is important to document the schema as clearly as possible, since users may end up having to parse it.

A more sophisticated approach is to encode each component of the recorded date and/or time as a separate field in its own dictionary: a field for the year, a field for the month, etc. Even this introduces questions: should the month be encoded as a number, its full name ("January"), its abbreviation ("Jan"), or some combination of the above? A simpler approach, with its own caveats, is Epoch time, or the number of seconds after the Unix Epoch (Thursday, 1 January 1970, UTC). Epoch Time will

often be unfamiliar to students, requiring an explanation and increasing the complexity of the activity. Epochs are particularly large numbers, which can introduce its own form of complexity — 32-bit signed integers can only represent Epochs from 1901 to 2038, which may be problem for datasets with particularly long prospects. Although Epochs seem like a simple solution to a difficult problem, we tend to avoid using them in the CORGIS datasets for the reasons above, instead favoring a combination of individual fields and a full string representation.

Ultimately, handling all the intricacies of dates and time is impossible, and most programmers do not have any idea of the scope of the difficulties involved . As with any difficult situation, consider what you need your learners to be able to do and what they need to learn — choose a structure and format that is most suitable for this purpose. I recommend just doing your best and hoping that it works out.

5.4. Measurements:

Numbers rarely come without context, but programming languages rarely include units in their types. Numbers are used to store currencies, temperatures, distances, volume, and many other measurements. Are the units in dollars or euros? Centigrade, Kelvin, or Fahrenheight? Documentation is usually required to explain what a number means with respect to its field. Clever naming can be used to reduce the complexity — instead of a field named "Time Delayed", for instance, consider the name "Minutes Delayed", which is no more complicated but benefits from immediately explaining the units.

When designing for a specific audience, it can be helpful to use a measurement format they are already comfortable with. Inversely, putting data into an awkward format can provide an opportunity to have students practice manipulating data. In other situations, the context may demand or benefit from a certain format (e.g., using metric for scientific data, or sortable dates). Sometimes, converting data may result in unacceptable loss of precision, tainting the authenticity of the dataset. Floating point representations can be imprecise: doing conversations and then rounding can lead to less accurate results, for example.

5.5. Locations:

A common form of data is locations around the world. These locations can be present at varying levels: specific points, addresses, cities, counties, states, countries, continents. Each of the varying levels provides different problems and opportunities.

Many of the CORGIS datasets feature data from states in the USA. To conserve space, it may be tempting to use the two-letter state abbreviation ("VA" instead of "Virginia"). I strongly recommend against this approach, however, because of the ambiguities that can emerge when the location type expands, not to mention the increased cognitive load of converting from a two-letter abbreviation to the full state name. In fact, simply naming the field "State" often becomes a misnomer, tempting as it is. For instance, in many datasets, additional regions are also reported alongside states: the District of Columbia is not a state, but its data is often included. Puerto Rico and other United States territories are not states, but often contribute data to United States-oriented datasets. Their abbreviations will be unexpected to students. For some historical data, this situation becomes even worse: the Kingdom of Alaska and the Alaska Territory only formally joined the United States in 1959 (a scant 58 years before this document was published!).

The United States census breaks the country into 4 major regions with 2-3 divisions each. Other governmental bodies use other partitions, such as the Office of Management and Budget's 10 federal regions. Although these regions are often compelling divisions for data, students may not be familiar with them, requiring them to consult documentation. If the data is included alongside state and national level data, it can exacerbate the previously described redundant summing problem: with three distinct partitions of the data, summing all the instances will result in triple the expected total.

Global data often incorporates information about different countries. Unfortunately, the names, geographic sizes, and existence of countries varies widely over time. Keeping track of changes and aligning data time-wise in a meaningfully manner is exceptionally difficult. Be aware of political and cultural land-mines when choosing names, and what locations to include: some countries may refuse to recognize each other. I strongly recommend referring to experts and the literature to provide arguments for and against including or excluding countries — it is beyond the scope of this dissertation to make any arguments in any direction.

Encoding postal addresses raises similar concerns to representing dates and times. Should an address be broken into individual fields, encoded as a single string, or are multiple representations

appropriate? Before answering this question, it is usually also useful to ask: what parts are actually important? Cities, states, countries, and zip codes are typically usable within geographic plotting systems, and are often sufficient on their own. Addresses alone are usually not particularly useful to students, unless they have access to a convenient geocoding system which can give them latitude and longitude coordinates. Geotagging services can be expensive, slow, and rate-limited; they are best done as batch operations. The CORGIS project geocodes during the preprocessing phase, rather than expecting users of the datasets to do so.

5.7. URLs:

Similar to postal addresses, some datasets make URL data available. These URLs can connect students to further documentation, data, or other resources associated with an instance of a dataset. One of the more common use cases, for instance, is linking to an image stored on a remote server. This keeps the dataset's file size down while still allowing the user access to richer data. When using URLs, be sure to include the whole URL schema for convenience and clarity's sake (including the "http://" or other relevant protocol).

Distributed data that is not specifically under your control may change at any time, without notification. This change could be the simple disappearance of the data, but it could also be something more malicious (e.g., malicious code, offensive material). Consider mirroring the original linked data so that you can maintain control over the students' experience. Unfortunately, running your own mirror introduces a whole new collection of problems. In fact, many of the technical problems outlined in the RealTimeWeb project (Section [lnk:rtw]) apply directly to relying on linked web data. The challenges of working with distributed data are so great that you should carefully consider whether students will actually need to take advantage of the linked data, or whether you are simply padding the dataset unnecessarily.

5.8. Enumerated data:

Typically, a fields' value falls into a narrow range of options, which can be represented as an Enumerated Type. Many datasets use a numeric code to reduce the space that the data takes up. This requires external documentation to figure out (e.g., using a lookup table or a data dictionary). I strongly recommend against leaving a field as an encoded numeric value — instead, replace the

value with a more meaningful textual value (keeping the full data dictionary information in the documentation). Although redundant, the benefit to students' cognitive load is substantial.

6. Knowing the data

Eventually, a created dataset will be encountered by learners. These novices must quickly learn the structure and meaning of the fields, in order to use the dataset for the intended learning experience. This section highlights our experience with the problems and opportunities that occur when students encounter pedagogical datasets.

6.1. Nobody reads the documentation:

As previously stated, both students and experts skip the documentation . When they do use it, it tends to be for specific, targeted tasks. However, this does not make the documentation unimportant. Make sure that the documentation is always readily available, easy to access, and able to answer whatever questions they have. The fewer the barriers, and the better the documentation is at answering questions, the more students will learn to use it and build confidence in documentation as a concept. One method for improving documentation is to log what questions students typically ask about a dataset, and using that data to improve the existing documentation.

6.2. Learning the structure:

One of the earliest steps in learning to use a dataset is to learn its structure. Students can sometimes struggle to understand the structure of data, especially hierarchical datasets like the ones in the CORGIS project. For instance, we note many students who believe that the top level of our datasets are dictionaries, not lists, despite repeated instruction. This is a symptom of a bigger phenomenon: learners focus on the interesting part of a data and gloss over the finer details. Care and attention must be given to feedback for the students, and provide them opportunities to demonstrate their understanding (or misconceptions) of the nature of the data. This can be structured as an explicit activity where students map out the data's structure as a diagram, such as the one shown in figure [fig:data-map].

6.3. Learning the distribution:

As the students learn the structure of the data, they must also gain an understanding of the nature of the data. When they encounter a numeric value in an instance, is it a high value or a low one? Was it unlikely to have that value? What is the greater meaning of the number? These questions of

distribution, scale, and characteristics of the data might be questions that students are explicitly assigned to answer, but they can also be distracting side-effects of using data as a context. You might consider providing tools and supplementary information to document the nature of the data. Automatic analysis can reveal suggestions about data, such as median values, the range of data, and to some extent what kind of mathematical distribution is represented. You might consider using these statistical tools yourself to judge the nature of your data, so you can help guide your students to data appropriate to their tasks.

6.4. Disseminating materials:

Ultimately, a pedagogical dataset must be delivered to students. The CORGIS project releases datasets and their associated files through a central website. We strongly recommend that the dissemination problem be solved early in the development process. How big is the dataset, and will it fit on your server? If the datasets will be delivered through email, do the systems involved have an issue with file sizes or types? If the datasets will be delivered through a third-party service, do they provide a convenient download interface? Will the dataset fit on students' development machines, and will they be capable of loading the dataset into memory? Mechanical issues hamper the learning process and distract students from more important learning objectives.

We note a particular form of confusion that we have seen students struggle with, no matter how we phrase instructions. When working with the CORGIS datasets, some students will download the database file and possibly the associated code file, attempt to open the database file, and then get confused. These files are typically meant to be processed with a programming language or other specialized software, but students are used to being able to download and open files directly. In the CORGIS project, besides striving for clarity in our instructions, we often include comments in the file to make it clear to students that they should not be opening the data file directly. This at least reduces the incidences of confusion.

6.5. Monitor usage:

Pay attention to how students interact, struggle, and comprehend datasets. There is no substitute in design for user interaction. No matter how frustrating students' mistakes can be, we recommend adopting the principle from user interface design: Never blame the user. If it does not contribute to their learning, be prepared to treat a problem with a dataset as a mistake in its design. To facilitate this process, consider adding tracking systems to the dissemination platform, both for the datasets

themselves and for the documentation. If possible, provide a specific place for users to report problems and confusion, either on the site or in the learning environment (e.g., a post-activity survey).