Last May, the US Library of Congress made the largest [release](https://www.si.umich.edu/news/library-congress-opened-its-catalogs-why-it-matters#gsc.tab=0) of digital records in its history – metadata for over 25 million books, maps and recordings. People immediately started making some pretty cool visualizations to explore [patterns in the data](http://sappingattention.blogspot.com/2017/05/a-brief-visual-history-of-marc.html), or demonstrate the [incredible](https://medium.com/@thisismattmiller/library-of-congress-lists-57ddd177f1e2?loclr=blogsig) size of the release. I set out to design a geospatial exploration of the data, with the primary goal of learning a few technologies along the way. This post catalogues some of the challenges in processing the data into a format useful for a D3 powered mapping visualization.

**Choosing MARC fields**

The library of congress organizes their record metadata following the MARC (MAchine Readable Cataloging) standard. There are many different metadata fields available for Library of Congress records. I wanted to do a project with a spatial component, so I settled on manipulations of the following:

* [050 a: Classification number](https://www.loc.gov/marc/bibliographic/bd050.html): Letters at the beginning of classification numbers indicate subject area. May be useful to explore in the future.
* [260 c: Date of publication, distribution, etc.](https://www.loc.gov/marc/bibliographic/bd260.html): Our main temporal variable.
* [650 a: Subject Added Entry; Topical term or geographic name entry element](https://www.loc.gov/marc/bibliographic/bd650.html): Strings about the subject, including location of the subject matter.

There are obviously many other options, which can be [explored here](https://www.loc.gov/marc/bibliographic/bd650.html).

**Parsing XML record files**

I restricted myself to just book records, which are available in for download in 41 separate compressed .XML files [here](http://www.loc.gov/cds/products/MDSConnect-books_all.html). Each uncompressed XML file is more than 500 megabytes. With files this large, loading them entirely into memory as you would with standard XML parsing methods becomes impractical (especially on my laptop). Instead, we have to use iterative parsing to step through the XML node by node without loading the entire tree structure into memory. The Python package lxml provides tools for doing exactly this, namely an implementation of the etree.iterparse() function. There are [some examples](http://lxml.de/tutorial.html#event-driven-parsing) of this on the lxml site, but I found this overall [explanation of iterative XML parsing](https://www.ibm.com/developerworks/xml/library/x-hiperfparse/) to be very helpful as well. We will want to iterate through “record” items. Then for each one, find all subfields with the tags corresponding to the chosen MARC fields.

At this stage, we have some initial cleaning that is also necessary. We might expect 260c – *Date of publication, distribution, etc.* to be a simple integer year telling us the year if publication. No such luck:

[EXAMPLE OF BAD DATE STRING]

We can roughly extract years from these strings by taking the first instance of exactly four consecutive digits using the [regular expression](https://docs.python.org/3/library/re.html#finding-all-adverbs) function:

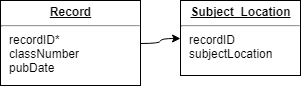
re.findall(**r'(?<!\d)\d{4}(?!\d)'**, date\_string)

Similarly we might want to strip some of the punctuation out of the subject location strings with the translate method.

.translate({ord(c): **None for** c **in '[];:?,.'**})

**Storing the parsed fields**

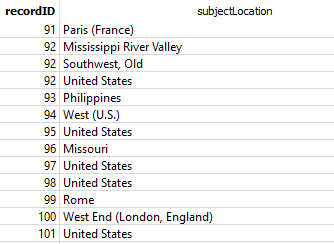
Great, we can parse the records, but we’re going to need to save them somewhere if we hope to use them in our visualization. Each record may have several subject locations, so our data structure needs to handle a one to many relationship. Given that we may be working with several million records that have these fields populated, we’ll use a SQLite database with a couple tables to start:



Once you have [installed SQLite](http://www.sqlitetutorial.net/download-install-sqlite/), the [sqlite3](https://docs.python.org/3/library/sqlite3.html) python package makes it easy to create and interact with our SQLite database from our Python script. [This thorough guide](http://sebastianraschka.com/Articles/2014_sqlite_in_python_tutorial.html) to using sqlite3 might be really helpful, especially if you’re just getting started with SQLite like me.

**Geocoding subject locations**

The big challenge of this project is turning the subjectLocation strings into actual locations that can be mapped. Here’s an example of some of the strings stored in the Subject\_Location table:



We have a mix of countries, cities, states, and descriptions of regions. The plan is to send each of these strings to a service that will geocode it, then return some associated data that we can use to make a map. We’ll use the Python package geopy, which provides access to several different geocoding service APIs (see [A Practical Guide to Geopy](https://sunnykrgupta.github.io/a-practical-guide-to-geopy.html)). Let’s consider three different services:

|  |  |  |
| --- | --- | --- |
| Service | API key required? | Query limits |
| [Bing Map API](https://msdn.microsoft.com/en-us/library/gg585136.aspx) | Yes | See documentation |
| [Google Maps Geocoding](https://developers.google.com/maps/documentation/geolocation/intro) | Yes | 2500 / day |
| [Nominatim (Open Street Map)](https://wiki.openstreetmap.org/wiki/Nominatim) | No | 1 / second |

After parsing all 4.4 GB (compressed) of XML records, we end up with over 2.6 million location strings. Clearly the query limits are going to be a big factor in deciding which geocoding service to use, and we’ll need to reduce the number of queries sent as much as we can. Restricting to unique strings we end up with 131,170. We can reduce this further by using the tool [OpenRefine](http://openrefine.org/) to cluster the collection of unique strings. This groups together similar strings and assigns a single value to each. We end up with 123,008 strings that we will need to geocode. We could manually check the data and eliminate some rows that don’t necessarily make sense for our project (“Solar system” for example), but I’ll just leave them in and consider whatever latitude and longitude is returned to be acceptable noise.

**Caching your geocoded data**

Bing had the most generous free usage limits (~50,000 / day) at the time I was doing this geocoding. Even so, it’s fairly critical to create a cache of returned data from geocoding queries. Repeatedly sending the same queries to a geocoding service during development could get you blocked from that service. We may also decide in the future to pull some other elements out of the query return, which would be unavailable if we just saved the latitude and longitude and threw out the rest.

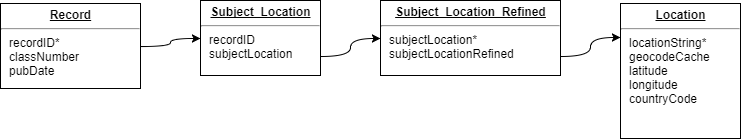
The strategy here is to create another SQLite table to store the query terms and results. Query results will be complicated nested dictionary structures. To cache this in our SQL database, we will have to “flatten” it using the Python package [pickle](https://docs.python.org/3/library/pickle.html). An excellent example of the entire caching process can be found in the answers to [this question on stackoverflow](https://stackoverflow.com/questions/28397847/most-straightforward-way-to-cache-geocoding-data).

**Point vs. Country data**

Let’s consider two possible formats that could be associated with each string: latitude/ longitude points, or country codes. Geocoding services should always return a longitude and latitude point for a queried string, but they may not always identify a country code. However, the lat/long point returned for a string like “United States” will fall in the center of the country, and since “United States” is likely to be by far the most common string in the subject location field, plotting longitude and latitude points may incorrectly make Missouri look like the subject of way more writing than the rest of the country.

[INSERT GRAPHIC HERE]

For this reason, we’ll want to just associate each record with a country. We can turn our longitude and latitude points into country codes pretty simply – by loading them in ArcGIS (or your favorite open source GIS) and spatially joining each point with the country it lies in. For this I used [Natural Earth country boundaries](http://www.naturalearthdata.com/downloads/10m-cultural-vectors/10m-admin-0-countries/). Great, now for each parsed subject location string we should have a refined string, a cached geocoded object, a longitude, a latitude, and an [ISO country code](https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2). Here’s our database structure:



Joining these tables and summarizing by pubDate and countryCode, we have the following spreadsheet ready for a D3 visualization:

[EMBEDDED CSV?]

------------------------------------------------------------------

Blog 1 Outline

* Release of LoC metadata (LINK: <http://fortune.com/2017/05/17/library-of-congress-free-record-release/> )
  + Available for download here: <http://www.loc.gov/cds/products/marcDist.php>
* Project goals:
  + Geospatial data project experience
  + SQL python integration
  + Learn some D3
* Exploring available fields (LINK: <https://www.loc.gov/marc/bibliographic/> )
  + 050 a - LoC Call #
    - The alphabetic code at the start can be used to define subjects
  + 260 a - location of publication
  + 260 c - date of publication
  + 650 a - subject location
    - There may be multiple listed!
* Major challenges working with LoC data:
  + Large XML files: streaming
    - Package, tutorial, blog posts?
      * <https://www.ibm.com/developerworks/xml/library/x-hiperfparse/>
  + Working with SQL
    - SQLite tutorial
  + Geocoding text strings:
    - Geopy – different service restrictions

|  |  |  |  |
| --- | --- | --- | --- |
| Service (link to more info) | API key required? | Query limits | Data limits? |
| Bing | Yes |  |  |
| Google | Yes |  |  |
| Nominatim | No |  |  |

* + - Reducing number of strings to query
      * Unique strings only
      * OpenRefine those
      * VISUALIZE THE NUMBERS?
        + 1,010,260,818 book records scanned

IS THIS ACCURATE? Might be number of XML nodes.

* + - * + 1,765,584 book records with populated metadata

2,636,367 location fields with strings

* + - * + 131,171 unique location strings
        + 123,008 refined location strings
    - Archiving geocoded results
      * Pickle
      * POSSIBLE BLOG POST
    - Point data vs Region data?
      * Example map with radius of points determined by count?
    - Extracting country codes
      * Map of points
  + Final data to work with
    - Image of sheet? Embedded sheet?