Kelly Thompson
2018-08-05

IPUMS Data Engineer Exercise – Linking IPUMS v1.4

Report

Set-up

To begin the exercise, I first read through the instruction booklet in its entirety.

I used Git shell (via Git Desktop for Windows) to clone the repo with the input files. I saved a copy of the exercise instructions as well, for future reference.

Since I will be completing the exercise in Python, and the exercise documentation offered a helping hand in choosing a Python string comparison library, I went ahead and pip installed the jellyfish library and read in the library's documentation about how to call the Jaro Winkler comparison.

With the necessary libraries installed, I then launched a Jupyter Notebook in the directory with the files, in an Anaconda Python 3 environment.

Getting to know the data set

Before delving into answering the exercise questions, I want to get a sense of the datasets I'll be working with: content standards, completeness, how nulls are represented, etc. I know the data in the name fields are going to have inconsistencies because I've looked at digitized censuses and the corresponding transcribed data while doing genealogical research, and I know how messy they are - handwritten, inconsistent, varying levels of literacy of data collector and respondent..., but I want to get a feel for the numeric data and just how messy the textual data might be. I pull up the codebook for the variables on the IPUMS site as linked in the documentation.

I use the pandas read_csv function to read the two data sets each into their own pandas dataframe.

I use the pandas describe function to look at the 'age' data. I notice that the 1880 data has a weird max value of 999. I can't find this value in the codebook, but upon further investigation, this seems to represent missing data (given that there are no empty/null values in the 'age' column.) I make a note to be sure to consider that in any age comparisons I might do later on.

The only elements where I'm finding null values is in first names in the 1870 census and last names in the 1880 census. I'll need to keep this in mind when processing for name matches. In sorting the name data, I also find some textual non-null values (*, ---) which seem to represent unknown or missing data. I change these to numpy NaNs for the purposes of this exercise in the dataframe I've got in my Jupyter notebook.

The data in the 'sex' variable seems to be 100% complete and uses only codes 1 and 2.

I notice that all the data contains the code "10" in the BPL element. I know from the documentation that this represents birthplace, and that this data is supposed to be for people who reported being born in Connecticut in the 1870 and 1880 censuses. I double-check the code-book to make sure this corresponds to Connecticut, but it seems to correspond with Delaware (Connecticut is listed as code 009). Since this is consistent across the data sets and this isn't going to affect the content of the rest of the exercise [which birthplace is represented in the data doesn't matter for developing a matching

algorithm, as long as the data will contain matches we are looking for], but I'm going to ignore it but note it as part of this report.

Exercise – Question 1

At this point, I knew both the data sets were small enough to open in Excel. I know Excel is a very quick tool for me, and I can filter loosely very efficiently.

Using the import data from text feature in Excel, I imported the 1880 CSV data into a workbook. I sorted the sheet in ascending order by two ordered factors: NAMELAST, NAMEFRST. I then turned on filters for the name columns. I typed in "bee" and then selected the names that came up that were similar to "BEEBE" which were BEBEE, BEE, BEEBE, and BEEBY. This presented a subset of 24 rows. I noticed right away that there was a row with the name "CATHERINE". The Catherine Beebe we were looking for was 44 years old and coded as female in the 1870 census, and the match we found gives this Catherine Beebe's age as 54 and codes her as female, so this seems like a pretty high confidence match.

I wanted to try doing the same thing with regular expressions, too, so that is the next bit of the code. I looked for a loose combination of letters left-anchored in the field (since I knew the data was there from my eyes-on checking.)

Exercise – Question 2

For the second and third records, I decide to break out the jellyfish library and see how it works. I apply a lambda function to the 1880 dataframe row-wise in order to find the Jaro Winkler distance between each first and last name field and the data we are looking for. I decided to compare the name parts separately because I thought it would provide more accurate scores, but I'd like to read more about best practices for this. I pre-filtered the dataframes to avoid null values in the name columns.

I merge this data back into the dataframe, and sort in descending order to find the highest comparison values. This yields two matches that seem to be pretty good: "FANNIE E. BIRD", F, age 50 is the closest 1880 match for "FRANCES E BIRD", F, age 40 in the 1870 census; and "J. S. Luff", M, age 60 is the closest 1880 match for "J S Luff", M, age 49 in the 1870 data.

Exercise – Question 3

I turn the steps I took in Question 2 above into a function which I call find_name_match(). The function takes as input two dataframes of different census year data which we're trying to link. The function iterates through the rows of the first dataframe, and then uses the Jaro Winkler comparison from the jellyfish library to assign a distance score from 0 to 1, with 1 being the closest match. The function looks separately at last and first names. It them merges the scores back into the original dataframe and sorts in descending order (highest scores first) by last name and first name consecutively. The first row in the dataframe is the closest match by this measure. The criteria used are closest distance between last name data between the years and first name data between the years.

Exercise – Question 4

Next I add in the fourth row of the 1870 dataframe – "John Smith". The program returns many matches that have high comparison scores. This means the name was very common for people born in

Connecticut recorded in the 1880 census. Because there are so many people with this name, we need to add more matching criteria to filter out the noise.

Exercise – Question 5

To accommodate data with many matches, I add 'age' as a comparison criteria. If the person described by the 1870 microdata is the same as the person described by the 1880 microdata, we would expect their age to be 10 years greater in 1880 compared to 1870. Because I know that census data is often very inaccurate when it comes to age data due to survey methodology (the census was often responded to by someone else in the household instead of the individual themselves, leading to guessed ages, etc.) I would expect the actual difference to slightly vary. I add a calculation to the function to find the absolute value of the difference between the ages minus ten years. I don't care if the age is over- or under-estimated, I want a sense of the distance from the expected value. The smaller this value, the better the match.

While this was giving okay matches in the top five returned results, I still wasn't happy with the rankings. I decided to add two more criteria – sex code and household members in common from one year to another. Sex code was a calculated distance (like age above). I compared sets of household members by grouping the data by 'SERIAL' variable, or household id, and combining any names of people associated with that household into a set of unique values. By comparing the sets, I was able to come up with a count of exact matches of household member names between 1870 and 1880 censuses. This number could be used to boost scores where overlap was observed. While comparison scores would have been an ideal way to compare this data, due to time constraints I simply looked for exact matches.

This time, instead of sorting by comparison scores, I created a weighted total score combining last name comparison score, first name comparison score, age distance, sex code distance, and household member overlap for first and last names. I assigned coefficients to the factors based on hunches I had about the data and in response to the tests I ran on the data sets. I'm sure there is a more mathematical way to approach this, but I went with a rough cut method to try to get something workable quickly.

Exercise – Question 6

When I ran the refactored function on all 1,000 1870 records, the estimated run time was going to be 2 hours and 3 minutes. The average run time per 1870 record was 8.00 seconds after letting the program run for 10 minutes and 8 seconds.

Given that the function compares last name data to last name data and first name data to first name data separately, the function is running comparison scores for every row in the 1880 file twice per 1870 row. If we doubled the size of the 1880 file, the run time would double, at the least.

This creates a performance bottleneck because the string comparison functions are the slowest part of the analysis the function is performing. The more fields to compare and the more data in the list of possible matches, the longer the run time.

One way to modify the program to run faster could be to use the split-apply-combine method. This would mean splitting the data set of potential matches into smaller sets before applying the comparison function. This could mean filtering the dataset based on criteria that seem to have lower error rates or

which contain certain constellations of data values that we are looking for, such as females between the ages of 55 and 65 or last names with “mc” or “mac”. [That we are already working on a dataset of only people with a birthplace code of “10” is an example of this.]

To parallelize this program, I would have a script run the program in parallel on multiple batches of data at once instead of consecutively. The subsets or batches of data could be based on any of the variables that are used as matching criteria, such as birthplace, approximate decade of birth, etc.

To get better matches, I would explore several avenues. While my program starts to explore data normalization (conversion to uppercase and removal of periods) and addition of match criteria as ways to get better matches, these avenues could be explored even further, especially in the data normalizing realm (removal of spaces would be the first low-hanging fruit I’d go after.) Many common names are abbreviated in the microdata (“Wm” for “William”, “Saml” for “Samuel”), so it would be interesting to try converting some of these to longer forms before running the matching function. I think that middle initials were also throwing off the accuracy of matches for the very common names. When a middle initial was present on one year’s data and missing in another’s, it created disparities in the comparison scores that were due to missing data, not the true expression of the name. It would be interesting to attempt splitting these values out into a third name column and scoring them separately for the weighted total. These are just some quick examples I’ve thought of as I’ve been working through the example data sets at hand.

Output and Source code

The output of this work is a csv data file containing the fields id_1870 and id_1880. I will be submitting my source code as a Jupyter notebook (.ipynb). While in a production environment I would want to extract the final elements of the program that I would be parallelizing or automating as a .py file that could be run from a command line or by a shell script, I think the notebook format helps tell the story of how I approached the problem, got to know the data, and tested my work.