

Open Ended Capstone – Market Research

Yeh, Margaret

Dataset

American Community Survey 5-Year Data (2009-2022)

- <https://www.census.gov/data/developers/data-sets/acs-5year.html>
- <https://api.census.gov/data/2022/acs/acs5/profile/variables.html>

Groups: <https://censusreporter.org/topics/table-codes/>

- Group DP02 – Citizenship, Education, Household composition
- Group DP03 - Selected Economic Characteristics: County level economic characteristics, income of various households, retirement, social security, health insurance
- Group DP04 – Rent and Housing information
- Group S0804 – Means of Transportation to Work by Selected Characteristics for Workplace Geography
- Group S0701 – Geographic Mobility by Selected Characteristics in the United States
- Group S2303 – Work status in the past 12 months
- Group CP03 - Comparative Economic Characteristics

Target Questions to Answer

- Where would it be good to invest a new retail business?
 - Locations of rapid increase in income, increased people moving in, consider housing costs. Employment statistics
- Where are the areas that can benefit from improved public transport?
 - Locations of high population and lots of people commuting by car.
- Is there anything interesting that can correlate with increasing or decreasing housing prices?

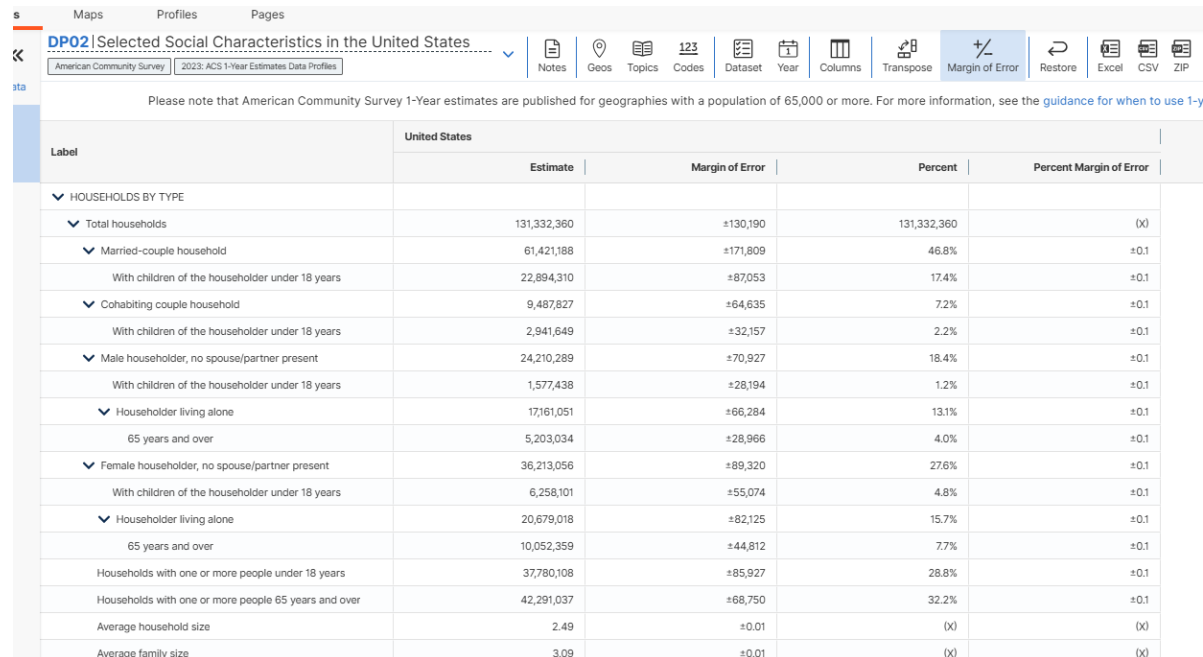
Data Exploration

- For some groups, there are years that the API request failed to get. It seems to happen at random
- Sample image of group DP03 (Rent and Housing information)

[12]:	DP03_0001E	DP03_0001EA	DP03_0001M	DP03_0001MA	DP03_0001PE	DP03_0001PEA	DP03_0001PM	DP03_0001PMA	DP03_0002E	DP03_0002EA	...	DP03_0003E
0	1367247	None	837	None	1367247	None	-888888888	(X)	922490	None	...	-888888888
1	1084	None	187	None	1084	None	-888888888	(X)	575	None	...	-888888888
2	34850	None	147	None	34850	None	-888888888	(X)	16238	None	...	-888888888
3	178693	None	318	None	178693	None	-888888888	(X)	103769	None	...	-888888888

Data Exploration

- Data.census.gov hosts their data and shows some of the variables in table format on their website. It is useful for initial exploration and checking for interesting datasets to add. For example, for the group DP02: <https://data.census.gov/table/ACSDP1Y2023.DP02?q=dp02>



The screenshot shows the Data.census.gov interface. The top navigation bar includes 'Maps', 'Profiles', and 'Pages'. The main title is 'DP02 | Selected Social Characteristics in the United States'. Below the title, there are tabs for 'American Community Survey' and '2023: ACS 1-Year Estimates Data Profiles'. A toolbar contains icons for 'Notes', 'Geos', 'Topics', 'Codes', 'Dataset', 'Year', 'Columns', 'Transpose', 'Margin of Error', 'Restore', 'Excel', 'CSV', and 'ZIP'. A note states: 'Please note that American Community Survey 1-Year estimates are published for geographies with a population of 65,000 or more. For more information, see the [guidance for when to use 1-y](#)'. The table below displays household data for the United States, with columns for 'Label', 'United States', 'Estimate', 'Margin of Error', 'Percent', and 'Percent Margin of Error'.

Label	United States			
	Estimate	Margin of Error	Percent	Percent Margin of Error
▼ HOUSEHOLDS BY TYPE				
▼ Total households	131,332,360	±130,190	131,332,360	(x)
▼ Married-couple household	61,421,188	±171,809	46.8%	±0.1
With children of the householder under 18 years	22,894,310	±87,053	17.4%	±0.1
▼ Cohabiting couple household	9,487,827	±64,635	7.2%	±0.1
With children of the householder under 18 years	2,941,649	±32,157	2.2%	±0.1
▼ Male householder, no spouse/partner present	24,210,289	±70,927	18.4%	±0.1
With children of the householder under 18 years	1,577,438	±28,194	1.2%	±0.1
▼ Householder living alone	17,161,051	±66,284	13.1%	±0.1
65 years and over	5,203,034	±28,966	4.0%	±0.1
▼ Female householder, no spouse/partner present	36,213,056	±89,320	27.6%	±0.1
With children of the householder under 18 years	6,258,101	±55,074	4.8%	±0.1
▼ Householder living alone	20,679,018	±82,125	15.7%	±0.1
65 years and over	10,052,359	±44,812	7.7%	±0.1
Households with one or more people under 18 years	37,780,108	±65,927	28.8%	±0.1
Households with one or more people 65 years and over	42,291,037	±68,750	32.2%	±0.1
Average household size	2.49	±0.01	(x)	(x)
Average family size	3.09	±0.01	(x)	(x)

Data Exploration

- There are some missing columns, and the headers are coded. Descriptions of the codes are included in the dataset documentation but not in the dataset itself
- 1. Is the data homogenous in each column?
 - Yes, within the column. If there are empty values, then the whole column is empty.
- 2. How do you anticipate this data will be used by data analysts and scientists downstream?
 - Data analysts can draw correlations between the data of different groups. For example, with the population group, people can separate out demographics in each county and see how they correlate with the target they are studying.
- 3. Does your answer to the last question give you an indication of how you can store the data for optimal querying speed and storage file compression?
 - It hints that people might want to be able to access datasets from multiple groups at the same time. Maybe storing with a cloud service and using cloud analytics is the best method of optimizing both the speed and storage.
- 4. What cleaning steps do you need to perform to make your dataset ready for consumption?
 - Removal of empty columns and columns that provide no information. Replace the coded headers or put the description of the coded headers somewhere it is easy to look up.
- 5. What wrangling steps do you need to perform to enrich your dataset with additional information?
 - The datasets can be joined so that not so many files need to be processed. Maybe in each group, join all the years into one table.

Data Exploration: Entity-Relationship Model

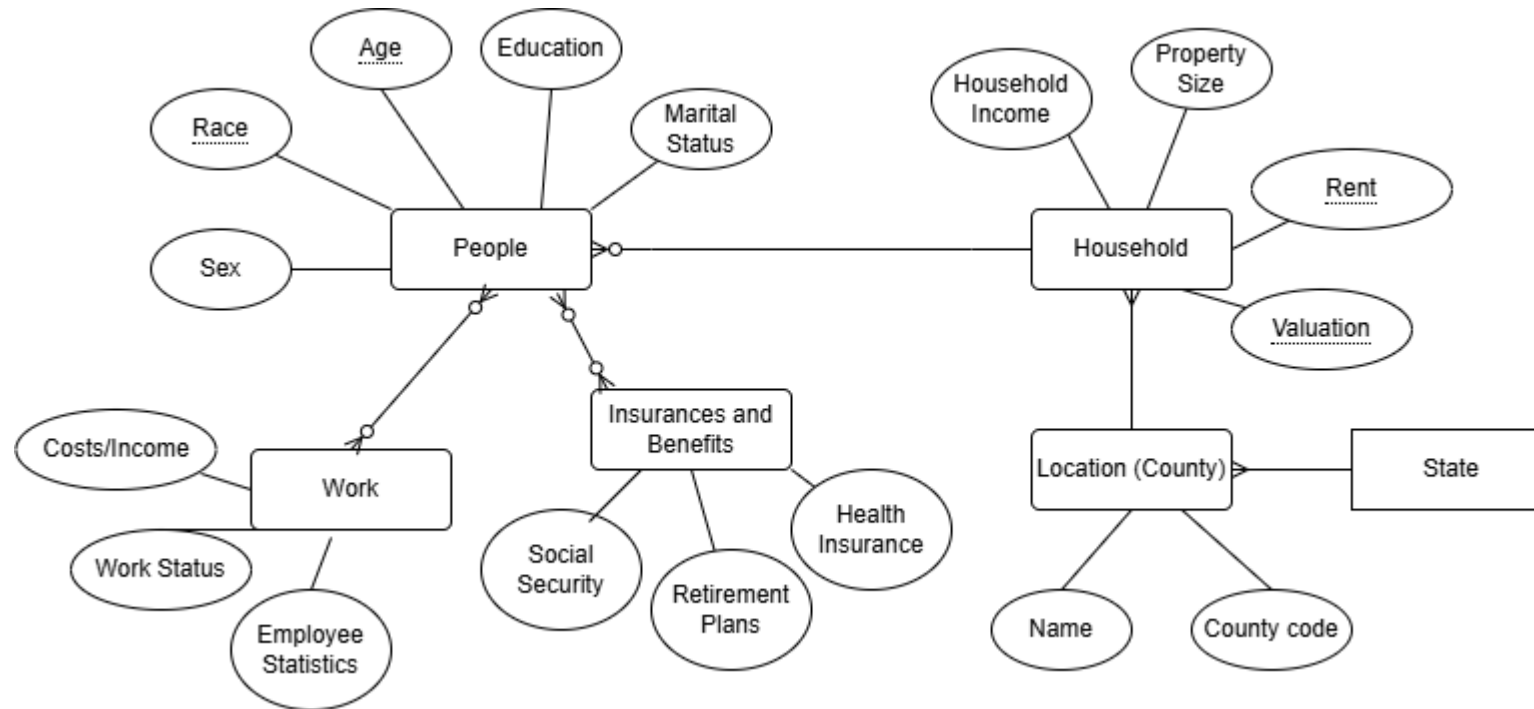


Figure made
using Draw.io

Data Exploration:

Goals with transforming the dataset

- Keep most of the information untouched.
- Combine most of the separate datasets together, see if its possible to have all the years in one table (sliding window? Originally the data for each year is a pooling of the previous 5 years). I want to be able to graph the data for a specific county over time.
- Maybe include the json that holds information that corresponds the header code to descriptions, put that in a table for easy lookup

Pipeline Prototype

- Original, testing with DP02: 3221 rows x 1196 columns
- First try was to combine all the same year jsons into one database, but this caused the program to hang. Change to try and clean null values first, then add each json to the dataframe one by one.
- After dropping null columns, testing with DP02, went from 1196 columns down to 721 columns. Added a column for year
- After dropping rows that are all null except for ['GEO_ID', 'NAME', 'state', 'county'], 3143 x 721
- There are many values that don't make sense like (x), I dropped all non numeric columns other than ['GEO_ID', 'NAME', 'state', 'county'], now there are 576 columns

Pipeline Prototype

- Sample data after cleaning: There are still some zero columns, but overall it is much cleaner than the start. Zero columns might not have been removed because of non-zero values that are outside of this observed window. I decided not to replace the header codes for now, may change it if it becomes a problem later on.

	DP02_0001E	DP02_0001M	DP02_0001PE	DP02_0001PM	DP02_0002E	DP02_0002M	DP02_0002PE	DP02_0002PM	DP02_0003E	DP0
0	7322	189	7322	0	6101	195	83.3	1.7	3156	
1	6691	189	6691	0	4915	271	73.5	3.3	2435	
2	52146	537	52146	0	36164	770	69.4	1.3	16444	
3	14200	400	14200	0	8268	332	58.2	2.4	3668	
4	3329	207	3329	0	2176	198	65.4	4.9	671	

Pipeline Prototype

- Metrics are logged to census_data.log
- Results from logger: The first line of transformer means DP02 has been combined into a single dataframe of 43997 rows and 576 columns
- Currently, the final dataframe is saved locally as csv for easier examination. But this probably will be changed into the future, maybe will need to change it to save as parquet into cloud storage

```
226 2024-12-16 15:10:24,936 -- __main__ -- INFO -- Running main
227 2024-12-16 15:10:51,633 -- transformer -- INFO -- Final combined DataFrame for acs_acs5_profile_DP02: (43997, 576)
228 2024-12-16 15:11:16,327 -- transformer -- INFO -- Final combined DataFrame for acs_acs5_profile_DP03: (45089, 439)
229 2024-12-16 15:11:42,003 -- transformer -- INFO -- Final combined DataFrame for acs_acs5_profile_DP04: (45089, 556)
230 2024-12-16 15:11:47,352 -- __main__ -- INFO -- Saved combined DataFrame for acs_acs5_profile_DP02 to data\acs_acs5_profile_DP02_combined.csv
231 2024-12-16 15:11:51,985 -- __main__ -- INFO -- Saved combined DataFrame for acs_acs5_profile_DP03 to data\acs_acs5_profile_DP03_combined.csv
232 2024-12-16 15:11:57,311 -- __main__ -- INFO -- Saved combined DataFrame for acs_acs5_profile_DP04 to data\acs_acs5_profile_DP04_combined.csv
233
```

Pipeline Prototype

- Slide deck updates:
- Choices I had to make about cleaning/transforming this data in the prototype
 - Some of the things I wanted to do (replace all large negative numbers with n/a) ended up taking an extremely long time to run on each dataframe. I ended up making choices to reduce runtime.
 - I decided against combining information across different groups in this prototype to keep things simple.
- Choices I made about the automation of the data pipeline that impact its performance or reliability
 - Each part (collector and transformer) can be run separately as shown in driver.py
 - I chose to hardcode the year in this implementation (After which it will collect all data up to that given year), but there is a part where if uncommented, it will automatically use the current year instead of the hardcoded year. If using the uncommented current year, then driver.py can be run every year (or every year with a US census release) and it will automatically grab all data up to the current year, transform it, and then save the combined dataframes as individual CSVs.
 - The Collector class is also able to grab a single year (Collector.getyear()), and the Transformer class is able to combine data from a single year to an existing dataframe (Transformer.combine()). If adding just one new year, it is possible to use these methods to update an existing dataframe with the new incoming json. This makes it faster if only updating with the new year. My current implementation assumes that the user starts from nothing, so it will download all years from 2009 every time.
 - The desired groups are listed in GROUPLIST in configs.ini and commented out if they need to be ignored

Pipeline Scaling

- Refactoring the code to work in Spark
- Initial problems:
 - Spark cluster does not run locally on my computer, so I ported the transformer.py and collector.py to Databricks Community Edition and did testing on a sample (DP02, 2009).
 - The json downloaded by collector.py was a list of lists, which would read into spark as a single corrupted string. I needed to change collector.py to use spark and directly parse the response into a spark dataframe before saving.
 - The manipulations take longer to do in Spark than on pandas, some things like null column removal needed to consider run time (A couple minutes on Spark, nearly instant in Pandas). Any sort of iteration through columns can't be done in a reasonable timeframe.
 - Plan is to remove Null columns, remove null rows, and remove columns containing non-numerical data.

Pipeline Scaling

- Try 1: Originally, to avoid the corrupt string error, data needed to be read through `pandas.read_json`, and then the Spark dataframe was made from the pandas dataframe. This defeats the purpose of using Spark, so to fix this, `Collector.py` needed changes to parse the API response correctly. I also changed `collector.py` to save in a parquet format rather than json
- Original had column names in the first row, changed `collector.py` to fix this before saving as parquet
- Dropping the null columns results in 720 columns. This matches the previous test with pandas, minus the added year column
- Dropping null rows also matches previous test with pandas, 3221 row

Pipeline Scaling: Thoughts

- Pandas is less restrictive, fixes many problems automatically. When converting to Spark, some things like detection of non-numerical columns would not work unless the schema was determined beforehand. Manually checking each column to convert takes too long. When first reading in the file, all columns are cast as StringType. Pandas is able to ignore the schema while doing manipulations, but Spark needs to add an extra step and correctly pick the data type before saving as parquet.

Pipeline Scaling: Difficulties

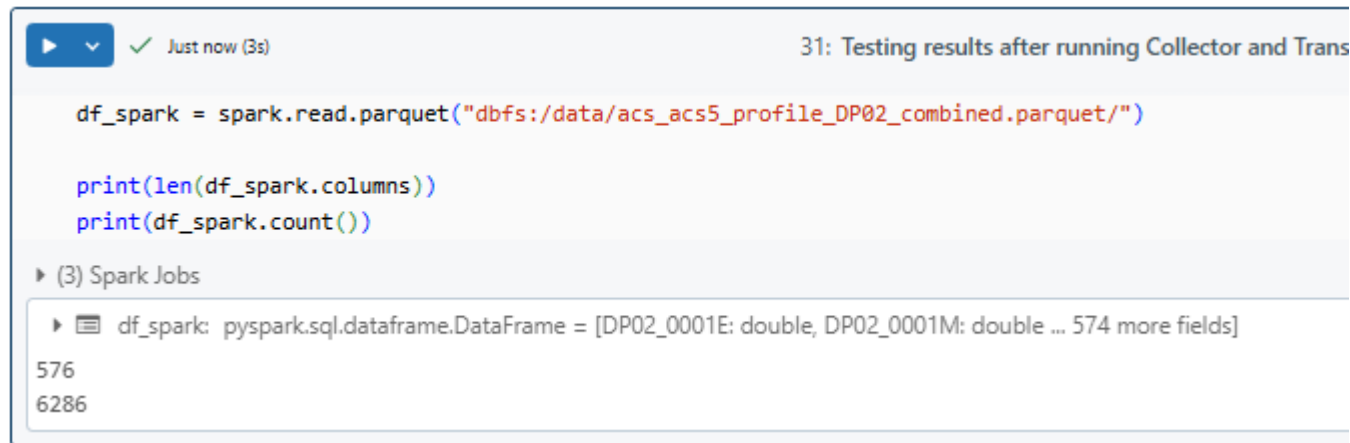
- Databricks Community Edition has difficulties importing classes from multiple notebooks (Collector.py, Transformer.py). It needs to %run the location of the imported notebook before importing classes and functions, but this also runs the imported notebook as if it was `__main__`. I wasn't able to figure out how to let it only let me import into the driver.
- Instead of using the driver to import both then create Collect and Transform objects to collect/transform, I altered collector and transformer to work without the driver. Later on I can schedule them to run directly in databricks on Azure one after the other. The driver is kept for local development but won't be used when running on Databricks. This makes it so each of them needs their own SparkSession and logger file which isn't ideal. Will try to figure out a better way in the future.

Pipeline Scaling: Difficulties

- Overall... it is slow, much slower than my pandas prototype (took 23 min for 3 groups, 3 years). I only worked with years 2009-2011 when confirming my code worked in Databricks. Maybe it is slow because I didn't properly clean the data when making the prototype. There are a lot more steps included in this part of the project to make sure the schema is correct. Also to simplify things, I made all numbers into floats instead of int types.

Pipeline Scaling: Results

- Final count of rows and columns for combined with 3 years of DP02. 576 columns corresponds to my prototype done in pandas
- Rest of the examinations and testing were done in the test notebook



The screenshot shows a Databricks notebook cell with the following content:

```
df_spark = spark.read.parquet("dbfs:/data/acs_acs5_profile_DP02_combined.parquet/")

print(len(df_spark.columns))
print(df_spark.count())
```

The cell status is "Just now (3s)" with a green checkmark. The title is "31: Testing results after running Collector and Trans". The output shows "(3) Spark Jobs" and a summary for "df_spark: pyspark.sql.dataframe.DataFrame = [DP02_0001E: double, DP02_0001M: double ... 574 more fields]". The final output lines are "576" and "6286".