

ECBS 5148 Data Architecture for Analysts

Individual Assignment - Report

Gerold Csendes (1901724)

Pump it Up: Data Mining the Water Table

Executive Summary

The context to my data product was to provide the data-related foundations to participating in DrivenData's Pump It Up machine learning challenge. A wide-range of learning objectives were planned to be demonstrated but some of them turned out to be not relevant to my use-case. This was due to the real-life nature of this data product. Though the 'base' data provided by DrivenData turned out to be a lot messier than expected and also the entities present were different from my expectations, I still think that my final data product can be considered as a suitable solution for the problem at hand. In the rest of the report, I will walk you through my work during which I will highlight the demonstrated learning objectives.

My whole work aside from the serialization section can be found at my [Github](#).

Data Flow

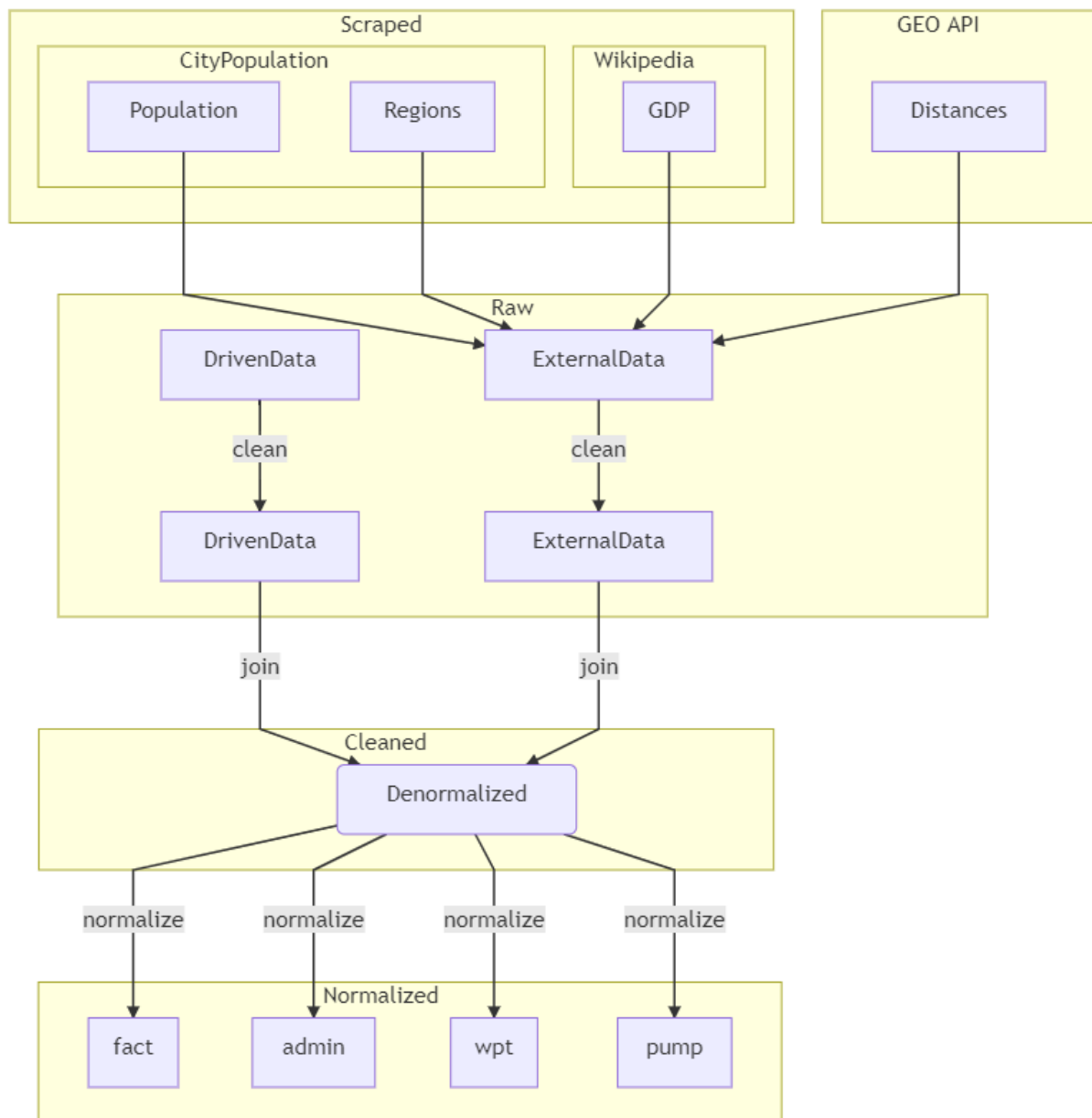
Nothing is more capable of explaining a data flow than a diagram. The below one shows the data sources involved and the actions involved. As we can see, this data architecture uses two main data sources: 1) DrivenData (internal) and 2) external data. I coined DrivenData's provided dataset as the internal data, since this serves as the basis for the machine learning competition and the other sources only serve enriching purposes to our internal dataset.

Out of the 3 main requirements from data architecture, I consider maintainability as the most important feature, since my use-case is an analysis and not production. In my case, maintainability may not seem super vital at first glance because my goal is to use the denormalized dataset for several machine learning methods with different calibrations. However, I still consider maintainability as a key issue, since it may happen that during the time of the analysis that I come across discrepancy in the data or find out to involve more data. In that case, the data flow shall be easy to reproduce to make those modifications.

Hereinafter, let's break down the data architecture to smaller (yellow boxes) pieces and see what tools and methods were applied in detail.

Learning Outcomes Demonstrated

- Represent a mental model visually.
- Create diagrams with Mermaid or other tool.
- Create concept map of data products.
- Identify key trade-offs in data architecture.



Scraped and GEO API Data

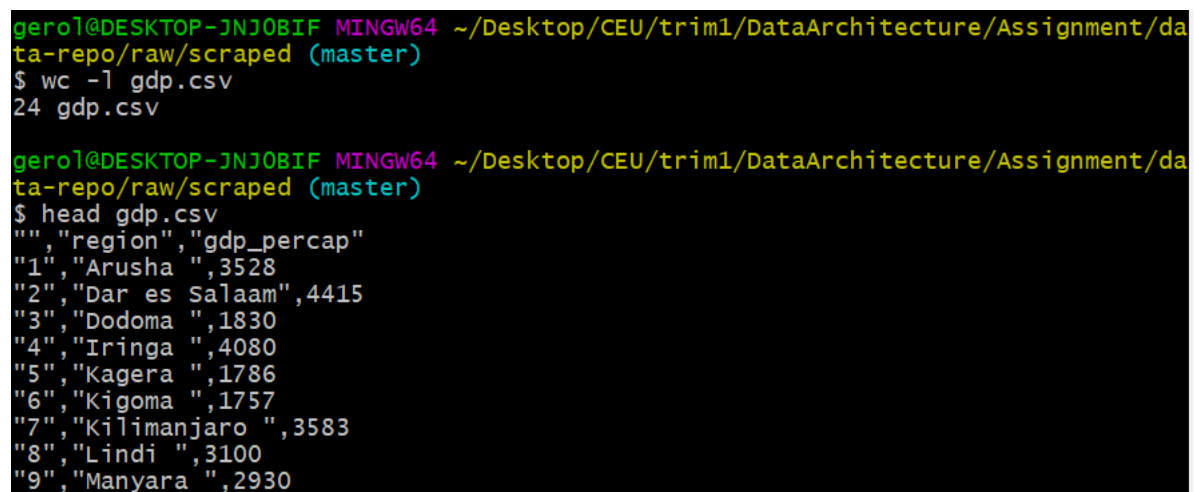
I scraped the web for two purposes: 1) to demonstrate this learning objective and 2) to enrich my data with population, capitals and GDP attributes on a regional level.

R was used to scrape the two websites: 1) Wikipedia for its regional GDP data and 2) Citypopulation.de for population and capital city data on a regional level. The code for the scrapers can be found in the root directory as *scrapers.R* and the datasets at *data-repo/raw/scraped* along with the htmls that were scraped. The robots.txts were of course carefully inspected prior to scraping data. It turned out that Wikipedia warmly welcomes “friendly, low-speed bots” that view and scrape articles. The other site has no robots.txt, thus, I supposed, it allows web-scraping from a legal and technical point of view too.

The resulting datasets required some tiny cleaning which meant slightly changing the names of the regions in the data from Citypopulation data. In the case of two regions, their names contained geographical locations in brackets, like Kagera [West Lake] which I simply changed to Kagera.

Although the GEO API data source has been added to the data-flow visualization, I will only say a few words about this because it has rather less to do with this content of the course or to be more specific, it doesn't demonstrate any more learning objectives than other parts of the architecture. The idea of using this source was to calculate a waterpoint's distance from the regional center. The code doing that can be found at [geo/tavolsag.ipynb](#). It first applies geocoding to fetch the GPS (latitude, longitude) of a city center and then uses the HERE API to calculate the route distance. This API accepts GPS as a single string, within which the latitude and longitude is separated with a comma. A for loop is used to join DrivenData's latitude and longitude of a waterpoint.

When cleaning the internal dataset, I used linux commands like *head* and *wc* to have glance over my scraped data which is proven by the below picture.

A terminal window screenshot showing the execution of two Linux commands. The first command, 'wc -l gdp.csv', returns '24 gdp.csv'. The second command, 'head gdp.csv', displays the first 10 lines of a CSV file, which include a header and nine data rows for different regions and their GDP per capita values.

```
gero1@DESKTOP-JNJOBIF MINGW64 ~/Desktop/CEU/trim1/DataArchitecture/Assignment/da
ta-repo/raw/scraped (master)
$ wc -l gdp.csv
24 gdp.csv

gero1@DESKTOP-JNJOBIF MINGW64 ~/Desktop/CEU/trim1/DataArchitecture/Assignment/da
ta-repo/raw/scraped (master)
$ head gdp.csv
", "region", "gdp_percap"
"1", "Arusha ", 3528
"2", "Dar es Salaam", 4415
"3", "Dodoma ", 1830
"4", "Iringa ", 4080
"5", "Kagera ", 1786
"6", "Kigoma ", 1757
"7", "Kilimanjaro ", 3583
"8", "Lindi ", 3100
"9", "Manyara ", 2930
```

Learning Outcomes Demonstrated

- Understand basic robots.txt structure.
- Use wget, curl or other programmatic tool to download data from the web.
- Use for loops to automate data transformations.

Cleaning the data

DrivenData's dataset must have been cleaned to facilitate high-quality data analysis. Missing values must have been normalized because its encoding has shown heterogeneity. Construction year, for example contained '0'-s, while the payment variable had 'unknown' values. I replaced missing values of all forms to python pandas *np.nan* for normalization purposes.

The data had been explored using python's *pandas-profiling* package which showed that a variable was constant and some variables were just a recoding of some other variable. Since such variables do not have much use in an analysis, those were dropped. After carefully inspecting the data dictionary of DrivenData, some additional variables were dropped out of analytical reasons. There were many geographical data on a different level of aggregation and for the sake of simplicity, only those seeming the most vital have been kept.

The funder was one of the messiest out of all column from the entity resolution perspective. The nearest neighbor method was applied after double-checking the resulting entities to provide consistence. The below picture shows that it seems to work well at clustering alike values. To provide transparency, the applied transformations have been saved in Openrefine in jsons. This also good for maintainability, as these changes can be later revised and also easily edited.

Cluster & Edit column "funder"

This feature helps you find groups of different cell values that might be alternative representations of the same thing. For example, the two strings "New York" and "new york" are very likely to refer to the same concept and just have capitalization differences, and "Gödel" and "Godel" probably refer to the same person. [Find out more...](#)

Method: ppm: Radius: Block Chars: 92 clusters found

Cluster Size	Row Count	Values in Cluster	Merge?	New Cell Value
3	32	<ul style="list-style-type: none"> Quick Wings (16 rows) Quick Wins (15 rows) Quick Win (1 rows) 	<input type="checkbox"/>	Quick Wings
3	13	<ul style="list-style-type: none"> Germany Misionary (11 rows) German Missionary (1 rows) Germany Missionary (1 rows) 	<input type="checkbox"/>	Germany Misionary
3	41	<ul style="list-style-type: none"> Roman Cathoric-same (24 rows) Roman Cathoric Same (15 rows) Roman Cathoric (2 rows) 	<input type="checkbox"/>	Roman Cathoric-same
3	32	<ul style="list-style-type: none"> Quick Wings (16 rows) Quick Wins (15 rows) Tquick Wings (1 rows) 	<input type="checkbox"/>	Quick Wings
3	132	<ul style="list-style-type: none"> Villagers (106 rows) Village (25 rows) Villages (1 rows) 	<input type="checkbox"/>	Villagers
3	29	<ul style="list-style-type: none"> Rotary Club (19 rows) Lottery Club (8 rows) Lottery Club (2 rows) 	<input type="checkbox"/>	Rotary Club

Choices in Cluster

Rows in Cluster

Average Length of Choices

Length Variance of Choices

Most of the data cleaning have been done in the *integrator.py* (for the internal data) in the root directory and the resulting dataset saved in *data-repo/cleaned* as *train.csv*. The cleaned – and joined – external dataset is saved as *economic.csv*, as it contains economics data. Take a note, that the latter doesn't follow a tidy format, since that data is wide-shaped but it is no key issue here, as I figured to later only use one year of data as DrivenData's dataset ranges mainly over 2012-2015.

These two datasets were joined and saved as *denormalized.csv*. This server as the *analysis-ready* data object because it contains all variables in a single table (in denormalized form) which comes handy when analyzing data but performs poorly when it comes to storing data efficiently.

Learning Outcomes Demonstrated

- Use string functions in OpenRefine or other tool to normalize text data.
- Explore data in OpenRefine or other tool with facets and filters.
- Save, edit and replay changes in OpenRefine on different datasets.
- Separate important from unimportant features

Normalizing the data

Data was normalized in order to story my analysis-ready dataset: *data-repo/cleaned/demormlaized.csv* effectively. Since we learned that the cost of a join is very low even for datasets exceeding 100m rows, and storing data in a normalized way reduces redundancy, it was wort normalizing my data.

This dataset already satisfies the firs, atomic cell condition. The dataset was split into 4 flat files, 1 fact and 3 dimension table at *./normalized*. The *fact.csv* contains the *inspection_id*, *date_recorded* the waterpoint id (unit that is beng observed): *ws_id* the result of the inspection: *status_group*. Here, the primary key is *inspection_id* and the foreign key is *ws_id*. The rest 3 dimension tables contain different type of information of a waterpoint. The conditions of the second and third forms are also satisfied.

The *admin.csv* tracks what method a waterpoint is using to pay its bills. Thus, it only has a primary key *ws_id* and the *payment* (string) variable. The *pump.csv* contains technical information of a waterpoint,

or to be more precise, the technical information on the water pumping process. It has again the primary key of *ws_id*. The last dimension, *wpt.csv* contains general information about a waterpoint, such as its name and geographical (region, GPS coordinates, distance from region capital) information. It made sense not to pool such variable to any of the former two dimensions, because the variables in this dataset didn't change at all for none of the entities, while the rest two experiences changes in values.

In retrospect, normalizing the data didn't make much sense for this case. Prior to an extensive data exploration I was hoping that same units will be frequently observed but this turned out to be not the case. You can check this by using LINUX's `wc` command to compare the records in the fact vs a dimension. There are 54,097 in the fact and 54,028 in any of the dimensions. I checked, and there was only one unit 3 times observed and 67 twice. Thus, redundancy shouldn't be such a great issue. You can check it by comparing the storage needed for the denormalized vs normalized form. The denormalized.csv is 11,354 KB, while the normalized csvs add up to 9705 KB. So, normalization does mean some efficiency improvement.

The normalized data was also saved into an SQL database. I used Python's `Sqlite3` package to achieve this. After creating the database, I wanted to check how join performance improves after indexing each table. I originally wanted to test the time of a join several times but I didn't succeed because after the first join (of all datasets into the analysis-ready dataset), Python always showed 0 second. There may be some caching going on in the background that causes this. Thus, I was able to compare the joins on a sample of one observation for each type of join (one with and the other without indexing). Without indexing, the result was 0.00099802 seconds, with indexing 0.0009975 seconds. So there is seems to be a slight increase in performance but I wouldn't want to deal with indexing in a real-life situation because a human couldn't tell those times apart (and storing indexes require more disk space). The code for this is to be found at the end of *normalizer.py*.

Learning Outcomes Demonstrated

- Create and query a simple database in SQLite or other RDBMS.
- Understand and apply normal forms 1-3 to simple relational data.
- Build a binary tree from simple ordered data

Data Serialization

I didn't yet explain my choice of serialization format, the csv. Considering the size of my data, I figured that it is not worth differing from the the most-widely used format (csv). To demonstrate that I do understand trade-offs in data serialization and that I am aware of the most popular serialization formats, I conducted a small research in the [data-structures.py](#). I random sampled my denormalized.csv for 50,000 observations, thus resulting with 300,000 observations in total. Then, I compared JSON's, Parquet's and CSV's performance of reading and writing time and also the storage efficiency.

Let's start with storage. Parquet is the winner with 9,875 KB, then comes CSV with 65,959 and JSON with 165,853. You can see that Parquet dominates by far which is very nice but it has the downside that this is not a human-readable format, while CSV and JSON are. You can explore these latter two tih Notepad for example which I did.

Parquet also dominated with its super fast read and write time. To compare these times, I wrote and read the dataset 10 times for each format. I took Parquet on average 0.48 seconds to read and 1.07 to write the data. JSON proved to be better performing when writing the data with its 7.07 seconds vs. CSV's 17.98. CSV read the data 2.57 seconds on average, while JSON 8.38 seconds.

Parquet's dominance cannot be questioned, especially when it comes to comparing storing efficiency. In real-life I still may not opt for that format but I would consider changing to Parquet if my dataset were to grow to 1m rows. Parquet's better performance comes at a price. It uses a lot more memory to deserialize the data but this shall be neglected for my use-case.

I also experimented around with saving my dataset in different encodings. I always use utf-8 as default but the latin worked just as fine for my dataset.

Learning Outcomes Demonstrated

- Compare popular serialization formats fixed width, CSV, JSON, XML, YAML, JSONlines, Parquet.
- Explain the tradeoffs in data serialization.
- Explore JSON files with jq or other tool.
- Load and save text file with different character encodings.

Documenting the data

My data is documented using frictionless data's data package software. I only documented the denormalized.csv because it has the same variables with the normalized tables, and I didn't find documenting the original (DrivenData's) dataset necessary, as you can find a data dictionary online.

Learning Outcomes Demonstrated

- Create a Data Package to share data and metadata together.

Conceptual Plan

In the conceptual plan document the following learning outcomes were demonstrated:

- Create logical model for simple relational data and represent it with Entity-Relation Diagrams

Total number of learning outcomes demonstrated: **20**. There were **at least 2 objectives** demonstrated from each topic.