Antonopoulos Ilias
p3352004
iantonopoulos@aueb.gr

Entity Resolution with MonetDB

21/03/2021

A proof-of-concept approach on an entity resolution task, with Tensorflow for the machine learning part and everything happening inside a MonetDB instance. This project is part of the course Large Scale Data Management at the MSc in Data Science of AUEB, carried out during the Winter Quarter 2020-21.

Contents

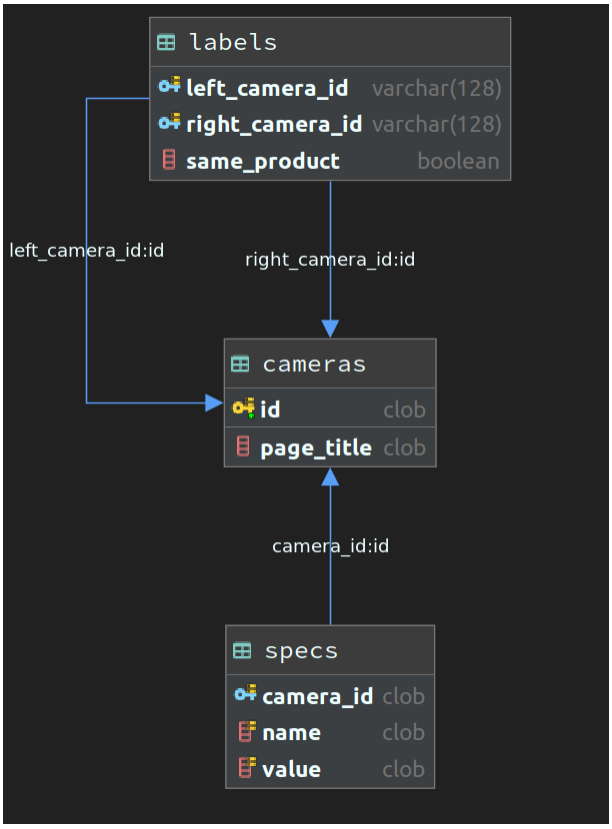
1 Data Loading & Database Schema

Let's begin by exploring the data.

By extracting the **camera_specs.tar.gz**, we can see the following structure:

```
buy.net/

      4233.json
      4236.json
      ...
      ...
      ...
      6785.json

cammarkt.com/

...
...
...

www.wexphotographic.com/
```

Given that each camera entity, apart from the page title information, can have any number of features (specs) available and that feature-overlap and consistency between any two products is far from the norm, we will utilize the following schema:



We will write 2 pythonic loaders (UDFs) that will create and populate the tables **cameras** (holding the primary identifier and the page title) and **specs** (holding a reference to the camera and the different camera specs with their corresponding values.

A point-of-interest here is that some of the JSON files contain an array of legitimate values for a particular key (e.g. *./www.camerafarm.com.au/705.json*). In that case, our specs loader function will deduplicate the array of values before attempting to generate all the specs-values mappings.

For these two tables, we will utilize the MonetDB command CREATE TABLE FROM LOADER.

Now, on with the **sigmod_medium_labelled_dataset.csv** data. This is our annotated data, in a CSV format. We will utilize the COPY INTO FROM MonetDB command to efficiently bulk-insert the CSV data into a new table called **labels**.

Having transferred our dataset to a MonetDB instance, we will additionally perform some ALTERations to our tables in order to add ensure referential integrity and thus come up with a working chema:

a) enforce a primary key on **cameras.id** column.

b) enforce a foreign key constraint to the **specs.camera_id** (referencing **cameras.id**)

c) enforce a unique constraint on **specs** table, for the columns (**camera_id**, **name**, **value**)

d) enforce a primary key on labels (**left_camera_id**, **right_camera_id**) combination.

e) enforce a foreign key constraint to the **labels.left_camera_id** (referencing **cameras.id**)

f) enforce a foreign key constraint to the **labels.right_camera_id** (referencing **cameras.id**)

Now that we have a working database schema, we can proceed with some data exploration to build an understanging towards the entity resolution task.

2 Exploratory Data Analysis

We have **29787** products (i.e. cameras), and an astonishing **4661**(4660 + the page title) different specs across these products. This unusually high number of specs stems from the fact that the JSON files were essentially unstructured information, with little overlap, even between products of the same retailer.

We are also equipped with **46665** product pairs (out of the pool of all possible product pairs that can be formulated from our dataset), for which we know the ground truth i.e. if they are the same or not.

Table 1: select same_product, count(*) as count from labels group by same_product;

same_product?	count
1	3582
0	43083

As we've expected - given the fact that the label corresponds to a product pair - the problem is imbalanced. We have, comparatively, very few products that are "matching".

It is illuminating to calculate the support for each spec (i.e. the ratio of non-null values for this spec). We will include <page title> in the analysis, as the most prominent camera spec, even though we chose to include it in the cameras table and not in the specs table:

Table 2: top 10 specs, based on support (rounded to 2 decimal digits)

spec	support
<page title>	1
brand	0.53
model	0.5
megapixels	0.46
type	0.46
screen size	0.41
optical zoom	0.39
mpn	0.35
condition	0.33
upc	0.26

By taking a closer look at these 10 specs we can suspect that they contain significant discriminative ability and will most probably be of vital importance to our machine learning component.
It is also highly informative to get the number of specs with support below a given threshold:

Table 3: no. of specs with support below x%

# specs	support threshold
4638	10%
4600	5%
4502	2%
4383	1%
3527	0.1%
62	0.01%

This is in line with our initial impression of the data being very much unstructured: for example, 4383 out of the total 4662 specs have less than 1% support!

3 Blocking

We would like to block together cameras of the same brand. This will help us restrict the ”potential matching” space, as cameras that belong to different blocks should not, in principle, refer to the same camera!

To implement the blocking step, we will introduce a **brand_id** to our **cameras** table, that is going to be a reference to a new **brands** table. We will seed the **brands** table with a curated set of common camera brands (Canon, Kodak etc.) that are indirectly available to our dataset. All we have to do then is develop a MonetDB/Python UDF that, given a text, extracts the camera brand based on our curated selection.

We can now populate the **brand_id** attribute according to the return values of this UDF, when applied in the <page title> camera attribute.

brand extracted ?	count
yes	24562
no	5225

So, we have managed to accurately retrieve the brand from <page title> in the vast majority of cases. We will deal with the rest, where brand was not extracted, later.

Table 4: top 10 blocks, based on size

brand	size
canon	5504
nikon	4736
sony	3235
hikvision	2007
fujifilm	1509
olympus	1447
panasonic	1318
samsung	1069
kodak	757
ricoh	710

Table 5: bottom 5 blocks, based on size

brand	size
garmin	7
epson	6
wespro	4
contour	4
motorola	3

In total, we have formed **47** blocks.

Before continuing, we should note that a number of pre-processing actions were taken in order to normalize the <page title> attribute, pre-blocking, such as:

- converted to lowercase
- sanitization: punctuation removal, dashes, etc.
- fixing common brand aliases (e.g. cannon → canon)

4 Filtering

After block formation, we would also like to filter out ”easy” matches, by extracting-via-heuristics and comparing the models (D3200, 600D, 2032i etc.) of the cameras. We would thus end up - for each block - with a subset of cameras that match both on brand and model (and basically refer to the same camera).

To implement the filtering step, we will have to develop a MonetDB/Python UDF that, given a text, extracts the camera model based on heuristics. We will use:

- regular expression(s) to capture the **key observation** that almost all camera models are numbers that are followed or preceded by a selection of letters.
- special handling of any measurement units (cm, mm, mp etc.) that are present in text, to further solidify the regular expression matches.
- some special adjustments on very, very few popular models (eg. canon mark i, ii, iii, iv) that appear frequently in the dataset.

We will, in the same fashion as before, base the model extraction on the <page title> attribute alone.

From this point on, we will make use of the term **signature** and say that a camera has a signature if and only if it has a brand extracted (i.e. it has been classified to a block) and also a model extracted.

So, by the end of the filtering step, we expect that a subset of our cameras will each be equipped with a signature (and thus constitute ”solved” cases, as we are aware of information sufficient for matching).

found signature ?	count
yes	20805
no	8982

Table 6: top 10 signatures, based on size (i.e. how many cameras share the same signature)

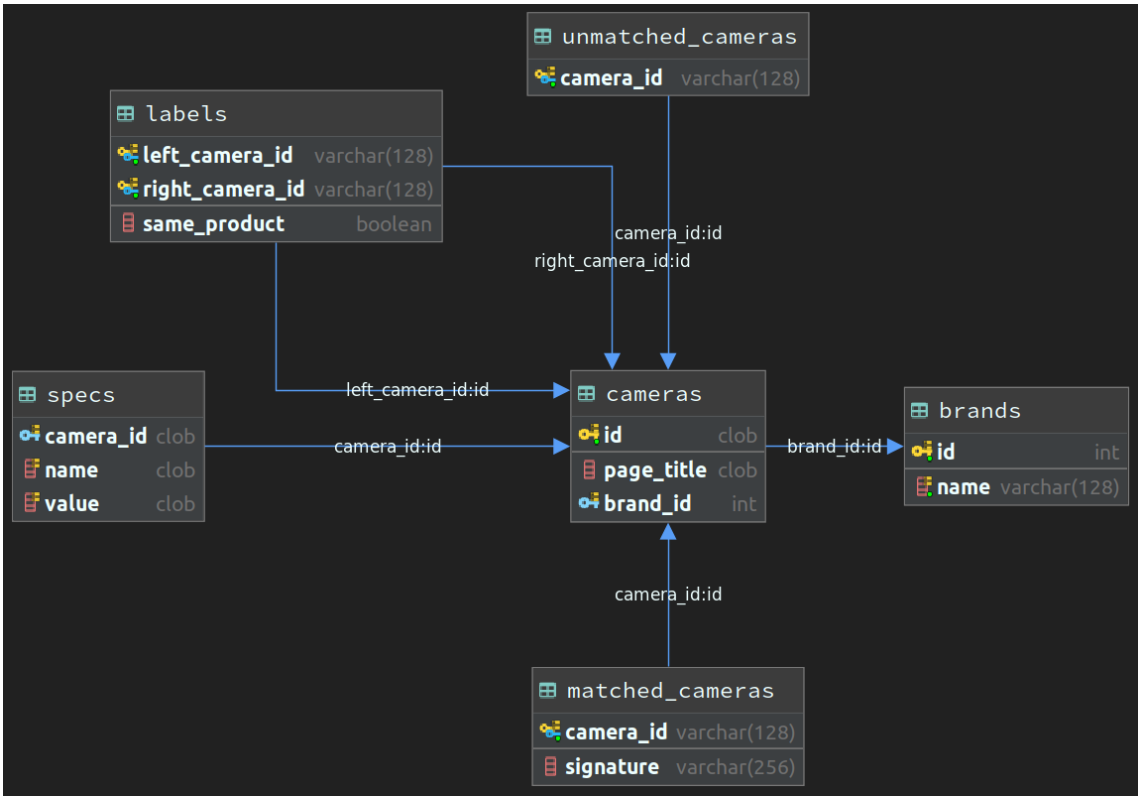
signature	size
canon_600d	250
hikvision_i5	203
canon_7d	192
nikon_d3200	185
hikvision_ip66	183
canon_60d	169
hikvision_3332i	169
canon_t3	166
nikon_d3100	156
hikvision_2012i	156

In total, we have managed to extract **3026** distinct signatures, across all blocks. It is informative to also assess how many distinct signatures were extracted per block.

Table 7: top 10 blocks, based on number of distinct signatures

brand	size
sony	529
canon	393
panasonic	295
fujifilm	279
nikon	273
samsung	227
olympus	200
kodak	180
casio	114
ricoh	103

And this is the database schema, at the end of filtering.



We have introduced 2 new proxy reference tables that host matched and unmatched cameras after the blocking & filtering process. We could, however, simply introduce a column **cameras.signature** and we would still be able to differentiate ”unmatched” and ”matched” cameras by checking for the presence of signature (null or not).

We will wait and see how the project will evolve at the later phases and reconsider accordingly.

4.1 Remarks

For now, we basically want to keep things simple and reach a baseline filtering performance that allows a fairly limited amount of false positives. Equipped with this baseline, we can later return to the filtering process and possibly enhance it by utilizing the available information outside <page title>.

However, with some initials attempts on brand / model extraction from other (than <page title>) camera specs, we came to the conclusion that:

- potential gains are negligible, as in most cases impact less than 100 cameras.
- even in the above cases, model and brand is almost always NOT part of the <page title>, so it should not concern us for this filtering phase, but instead could be something that can be leveraged by the Machine Learning component.

Given the <page title>, our belief is that we should not be tempted to introduce many heuristics (e.g. per-brand model aliases and modifications) as this will render our solution ”hacky” and difficult to extend-maintain-debug. A specific heuristic adjustment (e.g. model: t3i $\rightarrow$ 600d, mark ii, mark iii, mark iv) may be introduced only if we find out that this adjustment will yield a disproportionate benefit on the filtering process (i.e. it impacts many, many cameras).

5 Matching

To be developed soon.

6 References

[1] Pedro Holanda, Mark Raasveldt, and Martin Kersten. "Don't Hold My UDFs Hostage - Exporting UDFs For Debugging Purposes". In: Oct. 2017. URL: http://sbbd.org.br/2018/wp-content/uploads/sites/3/2018/02/p246-251.pdf.

[2] Torsten Kilias et al. "IDEL: In-Database Entity Linking with Neural Embeddings". In: *CoRR* abs/1803.04884 (2018). arXiv: 1803.04884. URL: http://arxiv.org/abs/1803.04884.

[3] Sidharth Mudgal et al. "Deep Learning for Entity Matching: A Design Space Exploration". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 19–34. ISBN: 9781450347037. DOI: 10.1145/3183713.3196926. URL: https://doi.org/10.1145/3183713.3196926.

[4] George Papadakis et al. "A Survey of Blocking and Filtering Techniques for Entity Resolution". In: *CoRR* abs/1905.06167 (2019). arXiv: 1905.06167. URL: http://arxiv.org/abs/1905.06167.

[5] Mark Raasveldt, Pedro Holanda, and Stefan Manegold. "devUDF: Increasing UDF development efficiency through IDE Integration. It works like a PyCharm!" In: *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT)* (2019). URL: https://openproceedings.org/2019/conf/edbt/EDBT19_paper_242.pdf.

[6] Mark Raasveldt and Hannes Mühleisen. "Vectorized UDFs in Column-Stores". In: *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. SSDBM '16. Budapest, Hungary: Association for Computing Machinery, 2016. ISBN: 9781450342155. DOI: 10.1145/2949689.2949703. URL: https://doi.org/10.1145/2949689.2949703.

[7] Mark Raasveldt et al. "Deep Integration of Machine Learning Into Column Stores". In: *Proceedings of the 21st International Conference on Extending Database Technology (EDBT)* (2018). URL: https://openproceedings.org/2018/conf/edbt/paper-293.pdf.