Data 101: Data Engineering Note 6 – October 6, 2022

Data Preparation

Lecturer: Lakshya Jain *Scribe: Shreya Shankar*

Adapted from Joe Hellerstein's notebook on data preparation and wrangling in an earlier version of DATA 101.

1 Introduction

Data preparation is a very broad subject, covering everything from data models to statistical assessments of data to string algorithms to scalable data processing. In some sense, most of Data Engineering—most of data science!—boils down to Data Preparation. The lecture has actual examples of data unboxing and transformations, but we'll cover some of them here.

2 "Unboxing" Data

"Unboxing" is the act of looking at your data to get a high-level sense of what's going on. Some things to look out for:

- Header info/schema
- Metadata
- Comments

Take a look at some rows or records in your data. Most files will run into one of these categories: record per line (e.g., csv), dictionaries/objects (e.g., JSON), natural language (prose), images, and more. Check out the lecture for specific commands you can run to unbox the data, like `head`.

3 Structural Transformations

We can convert relations to matrices and vice versa. Many datasets are not immediately stored in relations, so it's useful to do these structural transformations. We can use the `UNPIVOT` operation to convert matrices to relations—or the UI in Trifacta, as done in the lecture. Similarly, the `PIVOT` operation converts relations to matrices.

Suppose we have a table with year and month columns, and we want to pivot it into year by month matrix form. We have many rows that have the same (year, month) pair, which means our `PIVOT` needs to pack many values into a single cell. To do this, Trifacta asks us to choose an aggregate function—a reasonable choice might be `AVERAGE(Inches of Precipitation)`. If you prefer, Trifacta (like Postgres) actually has an aggregate function that will just store a nested list (array) of all the values in a single cell—this is the `LIST` aggregate. We can do `PIVOT/UNPIVOT` in Trifacta, in Pandas, and in Spreadsheets.

Can we do `PIVOT` in Relational Algebra? No: think about how we declare column values in relational algebra: we write an expression like $\pi_{c_1,c_2}(T)$. The subscripts of the operator are part of the syntax of your relational expression—they do not change as the relation instance (the data in the database!) changes.

By contrast, for `PIVOT` the subscript of the π operator essentially needs to be "the set of distinct values in the relation instance", which absolutely changes as the relation instances changes. Similarly, `UNPIVOT` returns data values (an output instance) that come from the input schema which isn't allowed. "Pure" SQL as we've learned it—an equivalent to the relational algebra—shouldn't be able to express `PIVOT` or `UNPIVOT`. However, given how useful this is, many SQL systems have (proprietary) extensions.

4 Type Induction and Coercion

To begin let's review "statistical" data types. This is a slight refinement from the terms in DS100:

- nominal / categorical: types that have no inherent ordering, used as names for categories
- ordinals: types that are used to encode order. Typically the integers
- cardinals: types that are used to express cardinality ("how many"). Typically the integers . Cardinals are common as the output of statistics (frequencies).
- numerical / measures: types that capture a numerical value or measurement. Typically a real (floating point) number.

4.1 Data types in the wild

We've seen that some systems like databases keep data types as metadata, and enforce strong typing in storage when data is inserted or modified. Used carefully, databases will carry the data-type metadata along with the data when they communicate with tools or other databases.

But it's very, very common to work with data that has little or no metadata. In that case, we have to interpret the data somehow. As a very first step, we need to guess ("induce") types for the data.

4.2 Techniques for Type Induction

Suppose I give you a column of potentially dirty data. Suppose you have a set of types H. You need to write an algorithm to choose a type. How does it choose?

- Hard rules: Try types from most- to least-specific. (e.g. boolean, int, float, string). Choose the first one that matches all the values.
- Minimum Description Length (MDL): see below
- Machine learning / classification (not really discussed in this class)

4.3 MDL

MDL is used to find a good "default" type for a column. Each type has a different number of bits required to store each value, so the goal is to find the type that minimizes the total number of bits stored for that column.

Let's say $len(v)$ is the bit-length for encoding of a value v explicitly. Given a type T with $|T|$ distinct values, the bit-length of encoding a value in that type is $log|T|$. (E.g. there are 2^64 64-bit integers, and each one is $log(2^64) = 64$ bits long.) Let's say that indicator variable $I_T(v) = 1$ if $v \in T$, and 0 otherwise.

For MDL, we choose the type that minimizes the description length for the set of data c in a column:

$$\min_{T \in H} \sum_{v \in c} (I_T(v) log(|T|) + (1 - I_T(v)) len(v))$$

Consider a column of values: {'Joe', 2, 12, 4750}. Assume the default type is string, which costs us 8 bits per character. We can encode this as 3 16-bit integers and 'Joe': length is $3 * 16 + 3 * 8 = 72$. Or we can encode it all as strings: $(3 + 1 + 2 + 4) * 8 = 80$. MDL would favor "int16" over "string" in this example.

Note that one can enhance MDL in various ways. One approach that's interesting to consider is to induce *compound* types: i.e. the string "12/31/2021" could be *string* or *date* or a compound type like *int4 '/' int8 '/' int16*. Another approach is to use compression techniques to get tighter measures for the length of encoding—for both type-matches and for strings.

In practice, some systems will break if the chosen type doesn't fit all the data in the column, in which case they'll choose a "hard rules" approach. For systems that can handle a mix of types, something like MDL is not unusual, though it may be more naive (e.g. pick the type that matched the largest number of entries).

5 Numerical Transformations

There are various types of calculations to consider: scalar functions, aggregate functions, and window functions, all of which we have previously learned in SQL.

5.1 Scalar Functions

Recall a scalar is a tensor of dimension 0, or a value in some field. A scalar function is a function on scalar values. In relational algebra, they are subscripts of $\pi, \sigma, and \bowtie$, like $\pi_{a,f(b)}(R)$. Scalar functions are typically quick: they are computed individually on each record and run in parallel.

A custom form of scalar functions is a user-defined function. You use them as you would use built-in functions. In Postgres:

```
CREATE LANGUAGE plpythonu;

CREATE OR REPLACE FUNCTION pyhash(s text)
  RETURNS text
AS $$
  ## Python text goes here, can reference variable s
  import hashlib
  m = hashlib.sha256()
  m.update(s)          # here's s!
  return m.hexdigest() # return a text
$$ LANGUAGE plpythonu;
```

5.2 Aggregate functions

Here, the input is a set or vector of values. Typical univariate functions include: min, max, sum, avg, stddev, variance. There are also bivariate functions: corr (correlation).

5.3 Window Functions

Recall that there is 1 value in output for each "window" of input values. Any aggregate function can be used in a window! Aggregates can be ordered (because windows can be ordered). Refer back to the windowing notes earlier in the semester.