Data 101: Data Engineering Note 7 – October 13, 2022

Data Cleaning

Lecturer: Lakshya Jain *Scribe: Shadaj Laddad*

Data cleaning is a key part of the pipeline for processing raw data from a variety of sources, as it is usually necessary to perform transformations on the data before it can be used in analysis logic. When writing data cleaning pipelines, our goal is to build reproducible transformations that produce a new version of the original data, rather than replacing it. This makes it easy to tweak the cleaning process later on and helps others understand how the cleaned data was derived.

1 Outliers

Data often contains outliers, either due to events that were very unlikely to ever occur or errors in entry. Either way, outliers can cause trouble with analyses (especially statistical ones) of our data, so we may want to remove such values.

1.1 Gaussian Outliers

The simplest definition of outliers is based on the Gaussian distribution, which forms a ”bell curve” defined by a mean (center of the curve) and standard deviation (spread of the curve). For a given value, we can compute it's **z-score**, which is how many standard deviations it is from the mean. When the absolute value of the z-score is large, that means the value was less likely to occur according to the distribution.

Using z-scores, we can easily remove outliers from our data. For example, if we remove any values with an absolute z-score larger than 2, we will keep only the values from the 2.5th percentile to the 97.5th percentile.

1.2 Trimming

Using just Gaussian distributions to find outliers has some caveats, however. If there is a single value that is much much larger than the others, its presence will significantly increase the measured standard deviation and result in outliers that aren't *as* far to be kept in the dataset.

This problem is isomorphic to the difference between mean and median, as the latter is more robust to the presence of extreme outliers. We can take a page from this book and remove the lowest and highest n values instead, which will not suffer from the same sensitivity to outliers.

A slight variation on this is to replace outlier values that would have been trimmed with the value *just before* those that are removed. So instead of outputting NULL values or removing rows, the edges of the dataset will just have a tail of repeated values. This approach is called Winsorizing, and helps preserve the standard deviation of the original dataset.

1.3 Robustness

Finally, let's discuss how we decide *how much* data to preserve when removing outliers. One approach to this is to focus on how robust our outlier removal is to the introduction of corrupt values.

We started with means and standard deviations, but as we saw earlier this model of distributions is especially susceptible to outliers. But if we start with medians, we can define the median absolute deviation (MAD) as the equivalent to standard deviations: $MAD(X) = \text{median}(|X_i - \tilde{X}|)$.

Now, let's map this back to the percentiles we used when picking our z-score ranges. We find that 1 standard deviation is equivalent to 1.4826 MADs (when evaluated on a standard normal distribution with mean 0 and stddev 1). So to compute an equivalent range to 2 standard deviations with more robust MADs, we can filter values that are $2 * 1.4826$ MADs from the median of the dataset. This strategy is called Hampel x84.

2 Imputation

Raw data will often include some missing or corrupted data, which we may want to fill in with replacements because downstream computations (such as machine learning pipelines) often depend on having all values for each entry.

Consider a scenario where we are tracking the number of students attending each discussion section. A TA may have forgotten to record this number for one week's section, but we can use SQL to replace this value. Specifically, we can use `CASE` statements to introduce conditionals that can choose between passing through the existing value or using another expression to replace it.

We start by computing the averages for each TA:

```sql
WITH avg_attendance AS
SELECT id, AVG(num_students) AS avg
FROM discussion_metrics
GROUP BY id
```

Then, we can compute a version of the original table with null values replaced by the corresponding average:

```sql
SELECT attendance.id,
    CASE WHEN attendance.num_students IS NOT NULL
    THEN attendance.num_students
    ELSE a.avg
FROM discussion_metrics AS attendance INNER JOIN avg_attendance AS a
ON attendance.id = a.id
```

But this may not be a good model for attendance, which (unfortunately) has a tendency to drop off over the semester. We can handle this by using linear regression to create a model of attendance over time.

First, we compute slopes and intercepts for each TA:

```sql
WITH avg_attendance AS
SELECT id,
    regr_slope(num_students, date) AS slope,
    regr_intercept(num_students, date) AS intercept
FROM discussion_metrics
GROUP BY id
```

And then can use this to replace NULLs:

```sql
SELECT attendance.id,
    CASE WHEN attendance.num_students IS NOT NULL
    THEN attendance.num_students
    ELSE (a.slope * attendance.date + a.intercept)::int
FROM discussion_metrics AS attendance INNER JOIN avg_attendance AS a
ON attendance.id = a.id
```

This covers the basics of imputation, but there are even fancier techniques for better models, such as interpolating over chunks of NULL values in ordered rows (see lecture notebook for an example in SQL).

3 Entity Resolution

Once we have each table cleaned up and ready to process, there is still one significant barrier to performing data analysis: joins. Although we have cleaned data locally, within each table, we have not cleaned up the connections between these tables such as the strings we may want to join on.

For example, consider a dataset with two tables consisting of human-entered data: one with product names and their prices, and one with product names and their popularity. A natural analysis query may be to join these:

```sql
SELECT a.name, a.price, b.popularity
FROM prices AS a INNER JOIN popularity AS b
ON a.name = b.name
```

But because these tables have been entered by humans, there may be slight differences in the names used to identify equivalent products. For example, the prices table may have a row with name ”Between 1&2 (CD)”, but the popularity table may instead use the name ”Between 1&2 [CD]”. Small differences like this will fail the equality condition in our join, resulting in many pairs missing from the query result.

3.1 Matching Data

What can we do in such situations? Well, if we consider this example, what we want to do is perform a join with a richer predicate that compares the string but tolerates slight differences between them. One algorithm we can use to compute this ”tolerance” is the **Levenshtein distance**.

At its core, the Levenshtein distance computes the fewest number of insertions (adding a character), deletions (removing a character), and mutations (replacing one character with another) required to transform one string to another. For example, in our example above, the Levenshtein distance is 2, because we replace the ”(” with ”[” and the ”)” with ”]”.

With a function to compute the Levenshtein distance (which is available in many databases), we can rewrite this query to tolerate slight differences:

```sql
SELECT a.name, a.price, b.popularity
FROM prices AS a, popularity AS b
WHERE levenshtein(a.name, b.name) < 5
```

Note that because we are performing an advanced calculation on every pair of tuples from the input tables, we cannot optimize this query using strategies such as a hash join. Instead, this query will perform a nested iteration over both tables and compute the distance for each pair, which will be *extremely slow*. Thankfully, there are smarter indexing strategies that can optimize this, which we will learn about later in this course.

3.2 Blocking and Matching

In some situations, we may not be performing a specific join but want to identify chunks of our data that correspond to equivalent real-world entities. We break this down into two phases: blocking to identify subsets of the data that contain values that are likely to be equivalent and matching to determine exactly which values correspond to the same entity.

Blocking is typically performed with a regular `GROUP BY`, with some key that is used as an approximation for equivalence (note that we may generate several keys for one row, because blocks may overlap). For example, we can use q**-gram** blocking, where we create blocks for every substring of length q that show up in the source data.

Once our blocks are formed, the next step is to cluster our entities into equivalence classes. First, we define a metric to numerically compute a ”distance” between values, which can either be univariate (between pairs of values) or multivariate (between pairs of tuples). Then, we can employ a clustering algorithm to merge values that are close to each other. These results can then be used in a data analysis to traverse the graph of identical and related values.