A Tale of Three Fuzzy Matchers

Welcome

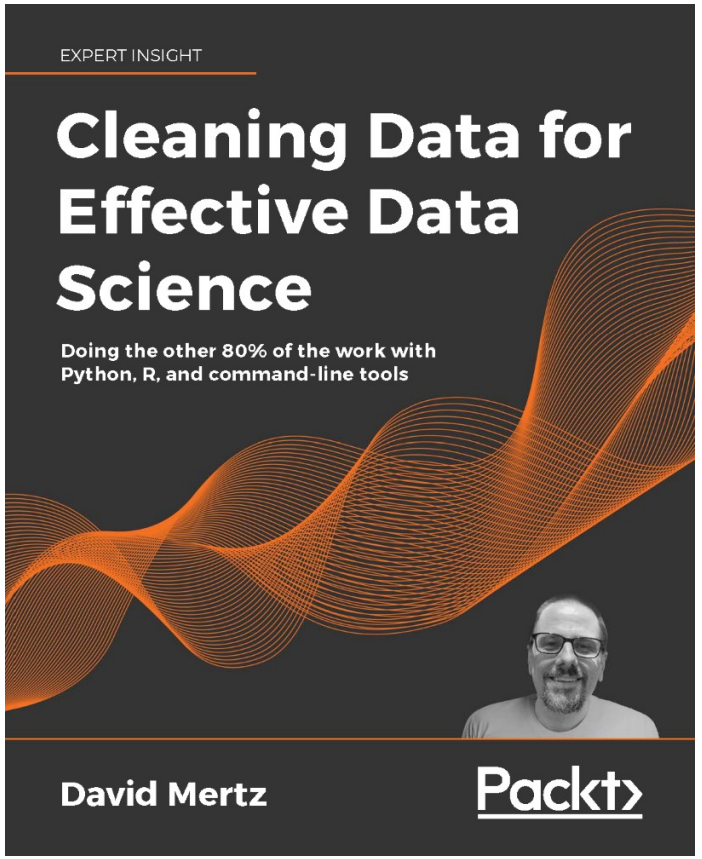
Heuristic matching is frequently needed to identify the individual people who are (imperfectly) described by data records.

I developed several algorithms, each having different performance characteristics, for the Ada software system.

david.mertz@seiu.org
www.seiu.org

About Me

Many of the concepts addressed in this talk are also discussed in my *Cleaning Data* book, which may be read and downloaded at the link below.



mertz@gnosis.cx
gnosis.cx/cleaning

About SEIU

SEIU is a labor union representing 2 million workers in the United States, Puerto Rico, and Canada, founded in 1921.

We fight for a just society where all workers are valued and all people respected—no matter where we come from or what color we are; where all families and communities can thrive; and where we leave a better and more equitable world for generations to come.

About Ada

A computer system, named after 19th century computer pioneer Ada Lovelace, processes and enhances membership data provided by the 150+ union locals of SEIU.

As with all data, ours is partial, often flawed, sometimes inconsistent, and absolutely necessary for our union's political lobbying and organization drives.

Watercolor portrait of Ada King, Countess of Lovelace, c. 1840, possibly by Alfred Edward Chalon

Public domain

https://en.wikipedia.org/wiki/File:Ada_Lovelace_portrait.jpg

First *fuzzy match*: Ada assigns an ID to a person without leaking information about their identity.

```
SELECT substr(raw_hash, 1, 8) AS raw_hash,
       substr(canonical, 1, 9) AS canonical,
       field_names
FROM identifiers LIMIT 8;
```

```
raw_hash|canonical|field_names
5cccc2ee|5cccc2ee2|first_middle,lastname,dob,external_id,affiliate
6c6efb90|5cccc2ee2|first_middle,lastname,dob,external_id
7f6faa8d|5cccc2ee2|first_middle,lastname,dob,affiliate
230b75a9|5cccc2ee2|first_middle,lastname,external_id,affiliate
ccf08fa8|5cccc2ee2|first_middle,dob,external_id,affiliate
4fb45087|5cccc2ee2|lastname,dob,external_id,affiliate
f2e1b668|f2e1b6687|first_middle,lastname,dob,external_id,affiliate
be9e2564|f2e1b6687|first_middle,lastname,dob,external_id
```

Three operations are performed:

- Subsetting fields
- Normalization of field values
- Cryptographic hash

```
raw_hash|canonical|field_names
5cccc2ee|5cccc2ee2|first_middle,lastname,dob,external_id,affiliate
6c6efb90|5cccc2ee2|first_middle,lastname,dob,external_id
7f6faa8d|5cccc2ee2|first_middle,lastname,dob,affiliate
230b75a9|5cccc2ee2|first_middle,lastname,external_id,affiliate
```

Field values are never stored in the table shown, but hypothetically, the rows we've seen could have had:

```
raw_hash|canonical|field_values
5cccc2ee|5cccc2ee2|DAVIDQ,MERTZ,1964-09-12,abc123,seiu-iu
6c6efb90|5cccc2ee2|DAVIDQ,MERTZ,1964-09-12,abc123
7f6faa8d|5cccc2ee2|DAVIDQ,MERTZ,1964-09-12,seiu-iu
230b75a9|5cccc2ee2|DAVIDQ,MERTZ,abc123,seiu-iu
```

Each subset of fields has a cryptographic hash mapping it to the first-encountered representation of that individual person.

Generation/lookup of an ID occurs at (much) better than 1,000 rows per second and deterministically generates an identical ID from row data alone.

The core operation takes a *powerset* of the available fields for a given record, with a threshold for *enough data*.

```
from itertools import chain, combinations
def powerset(iterable, min_length=0):
    """
    Return a powerset of an iterable.

    Powerset([1,2,3]) → () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
    Powerset([1,2,3], 2) → (1,2) (1,3) (2,3) (1,2,3)
    """
    s = list(iterable)
    return chain.from_iterable(
        combinations(s, r) for r in range(min_length, len(s) + 1)
    )
```

Typical Typos

Second fuzzy match: Spelling variation in employer names.

Acme Products, Incorporated, *Acme Products Inc.*, *Acme Products*, and (misspelled) *Acma Products, Incorperated* might all intend to represent the same entity.

- Normalization of field values
- Phonetic similarity with Jaro-Winkler or Levenshtein
- Adjust score based on supplemental information

Typical Typos

A DataFrame has company names and relevant "metadata."

The same employer tends to be in the same city and state, and have members from the same affiliate.

```
def with_metaphones(df):
  return pd.DataFrame({
    "employer": df.employer.astype(str),
    "affiliate": df.affiliate.str.strip().str.upper(),
    "city": df.city.str.strip().str.upper(),
    "state": df.state.str.strip().str.upper(),
    "metaphones": df.employer.apply(doublemetaphone)
  })
```

Typical Typos

The Pandas-style normalization is commonplace (but helpful). The interesting part is the metaphones.

```
>>> from metaphone import doublemetaphone
>>> doublemetaphone("East 70th Street")
('ASTT0STRT', 'ASTTTTSTRT')
>>> doublemetaphone("West 4th Street")
('AST0STRT', 'FSTTTSTRT')
>>> doublemetaphone("East 7th Street")
('AST0STRT', 'ASTTTSTRT')
```

We do not always get multiple options:

```
>>> doublemetaphone("Acme Products, Incorporated")
('AKMPRTKTSSNKRPRTT', '')
>>> doublemetaphone("Acma Products, Incorperated")
('AKMPRTKTSSNKRPRTT', '')
```

Typical Typos

Phoneticization with metaphone (or soundex) helps.

String similarity is the next step. Jaro-Winkler is somewhat more useful than Levenshtein in overweighting prefixes.

```
>>> from itertools import combinations
>>> from Levenshtein import jaro_winkler
>>> street_phonemes = ['ASTTTTSTRT', 'ASTT0STRT', 'FSTTTSTRT']
>>> for a, b in combinations(street_phonemes, 2):
...     print(f"{a:<10s} {b:<10s} {jaro_winkler(a, b):0.3f}")

ASTTTTSTRT ASTT0STRT  0.913
ASTTTTSTRT FSTTTSTRT  0.813
ASTT0STRT  FSTTTSTRT  0.757
```

Actual heuristic matching of company names uses phoneticization and string distance as a baseline.

For additional score adjustments, we utilize the other fields seen previously to tweak scores. Is it the same city? Did we get the company name from the same union local?

Using tweaked scores, we can decide the "best guess" about which entity is the best attribution for the data we actually see.

Let's reflect on the computational complexity of the first two matchers.

Create/locate an ID is an *O(n log m)* operation. Where n is the number of records being processed, and m is the number of previously stored rows.

Storage might be two orders of magnitude larger than current record set, but is not unbounded.

- For *each* of the N records being processed:
  - Generate a fixed number of subsets: *O(1)*
  - Perform a cryptographic hash on each subset: *O(1)*
  - Search an indexed table for existing match: *O(log m)*
  - If record is not matched, insert subset records: *O(1)*

Let's reflect on the computational complexity of the first two matchers.

Fuzzy matching a name (with scoring metadata) is an $O(n^2)$ operation... or $O(n \times m)$ if the current record set is considered separately from previously canonicalized initial records.

- For *each* of the N records being processed:
  - Generate a canonical representation: $O(1)$
  - Compare the edit distance of each phoneticized version to each stored version: $O(m)$
  - Compare each canonicalized metadata field to stored versions: $O(m)$

Third *fuzzy match*: Enhance political and demographic information based on an external data provider.

A vendor we work with has data about nearly all U.S. persons (over 300M), with a focus on demographic data, voting history, and political sentiment models.

The vendor will perform their own fuzzy matching based on some basic information—using algorithms and procedures that are confidential to them—but generally utilizing fields like name, birth date, address.

Our vendor will perform matching for us, but we also have a copy of their raw data if we wish to query it.

A tentative implementation of our own *entity resolution* has been created.

Essentially, the problem is to match batches of tens of thousands of person records against a database where essentially any specific identifying feature might be wrong or missing in our data.

This is a pretty standard and common database task.

There are a few fuzzing techniques that we have found useful. One common variation in data about persons is that they go by various nicknames.

The Python library *nicknames* is based on genealogy information from the Center for African American Research, Inc.

This library, as good as it is mostly centers on English language names, and will be of less use for names of other linguistic origins (but English has borrowed quite a bit from many sources).

```
from nicknames import NickNamer
def nicknames(name: str) -> set:
    """Get nicknames for a list of names.

    NOTE: connections among nicknames and canonical names forms a graph.
    The exact "shape" of this graph is not clear; we do not know
    whether it forms a reasonable number of equivalence classes by
    Recursive vertex contraction.

    As a practical heuristic,examine a small depth for direct nicknames,
    and for the nicknames of canonical names.  This will create a
    Reasonably sized collection of plausible nicknames.
    """
    nn = NickNamer()
    nicknames = nn.nicknames_of(name)
    for canonical in nn.canonicals_of(name):
        nicknames.add(canonical)
        nicknames |= nn.nicknames_of(canonical)
    return {n.upper() for n in nicknames}
```

The technique used creates different, but sometimes overlapping, expansions for related names.

It does a *pretty good* job of providing a good range of values to use in an SQL "`WHERE name IN (nicknames)`" query.

```
>>> nicknames("david")
{'DAVEY', 'DAY', 'DAVE'}
>>> nicknames("dave")
{'DAVID', 'DAVEY', 'DAY', 'DAVE'}
>>> nicknames("abigail")
{'GAIL', 'NABBY', 'ABBI', 'ABBE', 'ABBEY', 'ABBY', 'ABBIE'}
>>> nicknames("esther")
{'HETTY', 'HESTER', 'HESSY', 'ESSIE', 'ESTHER'}
>>> nicknames("chimezie")
set()
```

A few other techniques are used to "pre-fuzz" some data in records.

Last names can be divided on spaces and hyphens, and their components chosen as alternatives.

For postal addresses, the Python library *usaddress-scourgify* normalizes and pulls out components... of course, the format conventions are very US-specific.

Pandas can usually do a good job of normalizing dates to a needed format.

In designing a fuzzy string matcher:

- The "feel" of the data being searched
- The size of the dataset and match space
- Performance characteristics & compromises

david.mertz@seiu.org
www.seiu.org