

Cleaning Data

CMPT 353

The data that you get is rarely as accurate and as correct as you'd like.

Taking the data you find and removing/fixing problem data is called *data cleaning* (or *data cleansing*). It is often a big part of a real-world data analysis task (maybe 50–80%).

[For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights](#)

The process is generally fairly ad hoc: what you need to do depends on the data you get.

[A data cleaning example in an annotated Notebook](#)

Example problems: [[Data Cleaning & Integration, Chau](#)]

- duplicates
- empty rows
- (different) abbreviations
- difference in scales/inconsistency in description/sometimes include units
- typos
- missing values
- trailing spaces
- incomplete cells
- synonyms of the same thing
- skewed distribution, outliers
- bad formatting/not in relational format

In general, it's all just programming.

There are a few common problems and solutions we can talk about...

Validity

First problem: is the data valid?

- Actually present?
- Correct data type? number, boolean, etc.
- A legal value? length = “-10”? Date = “February 31 2021”? Phone number = “6” or “778-555-5555”?

If you have invalid data in your data set, you can fix, delete, etc, as appropriate.

Outliers

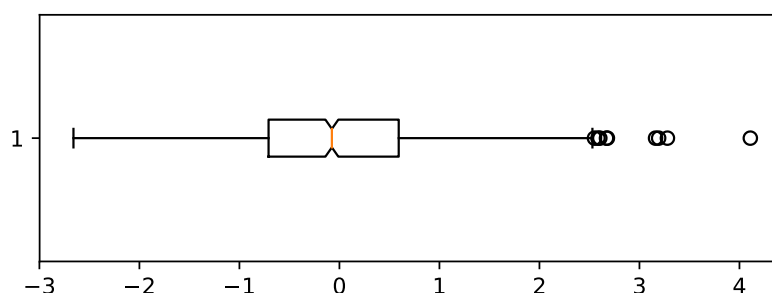
An *outlier* is a data point that's far away from the others.

Exactly how you define “far away” is going to depend on the data. What you do with outliers is also going to depend...

Outliers can occur in perfectly valid data that has no problems.

```
data = np.random.normal(0, 1, 1000)
print(data.mean(), data.std())
print(data.min(), data.max())
plt.boxplot(data, notch=True, vert=False)
```

```
-0.0478499351952 1.00437194396
-2.65944945638 4.10869262381
```



The point at $x \approx 4$ is four standard deviations from the mean. That would certainly be considered an outlier.

But we shouldn't just ignore it: it's a perfectly valid sample and should be included in any analysis.

Important point: you can't remove a data point just because you don't like it!

It makes sense to remove data if it's genuinely an error in measurement, or isn't fundamentally a true measurement of what you're looking at.

If it's just a slightly weird value, you have to justify it as an outlier, or keep it.

e.g. Suppose I am examining the temperature in my house, and take some measurements:

Object	Temperature
Air, bedroom	21°C
Air, living room	19°C
Air, oven	120°C
Desk surface	25°C

One value is clearly an outlier that measures a different thing. One is strangely high, but probably a valid measurement.

Finding Outliers

You can identify outliers by looking at how far a point is from the mean, often how many standard deviations from the mean.

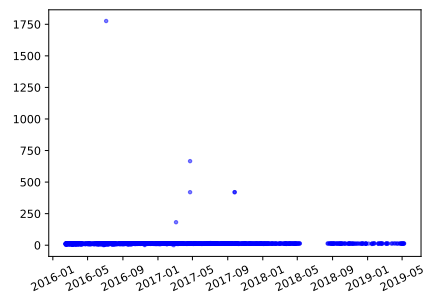
But that probably won't tell the whole story: different types of problem will show up in different ways.

[Wikipedia: Outliers](#)

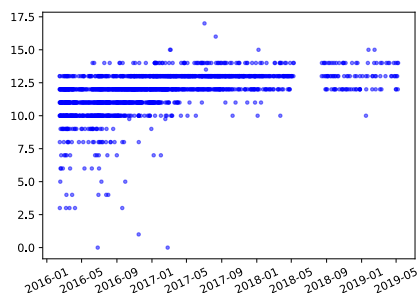
Suggestion: start by plotting.

Get a feeling for how your data is shaped, and look for problems by eye.

My scatter plot of date vs dog ratings for Exercise 2, without removing outliers:

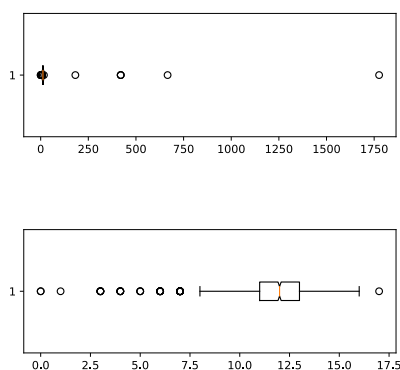


After removing the crazy outliers, we can actually see the data, and it looks okay:

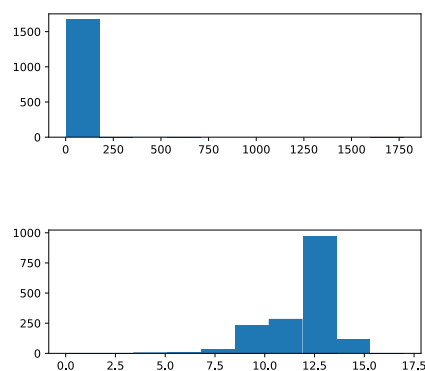


Are those points at 0 and 1 problems?

Or a box plot, with and without outliers:



Or a histogram:



With a basic plot, it seems pretty obvious that some of the “ $n/10$ ” ratings aren't really valid.

Including them in the analysis would overwhelm any other trends. We removed them from the data set.

Handling Outliers

A few things you can do if you identify “outliers”:

- Leave it as-is because it's actually valid.
- Remove the record entirely because it's invalid.
- Remove the *value* at treat it as a missing value (Pandas NaN or similar) in that record.
- Impute...

Imputation

If you have missing values (or deleted outliers), you could replace them by calculating a plausible value to replace them. This is called [imputation](#) (“imputing”, “to impute”).

Imputing is probably a bad idea: it's possible to introduce bias, introduce fake trends, wash out real trends,

Sometimes it's the least-bad option available.

Common ways to impute:

- Use (or average) nearby values.
- Average of known data. (*mean substitution*)
- Linear regression of nearby values. *
- Some other best-fit or maximum-likelihood result.

* i.e. do LOESS smoothing, and take the filtered prediction at that point.

Imputing can be useful when you have many variables and don't want to throw away a record for a relatively-low-importance variable.

Observation Time	X	Y
14:21	18	542.2
14:23	19	531.9
—	21	221.3
14:27	27	239.5

It might also be relevant to leave missing values in your data set. Maybe the fact that a value was missing is itself an important fact, and could be useful in later analysis?

Noise Filtering

As we talked about in the [Noise Filtering](#) notes.

Entity Resolution

Entity resolution or *record linkage* is the process of finding multiple values that actually refer to the same concept (entity).

e.g. Are “Steve Bergner” and “Steven Bergner” the same person?

e.g. Are these the same product? How could I (automatically) compare prices?

- “[AMD Ryzen 7 3700X 8-Core, 16-Thread Unlocked Desktop Processor with Wraith Prism LED Cooler](#)” from Amazon
- “[AMD RYZEN 7 3700X 8-Core 3.6 GHz \(4.4 GHz Max Boost\) Socket AM4 65W 100-100000071BOX Desktop Processor](#)” from NewEgg

In general, entity resolution is hard. Hope you have some structure or more precise values to work with.

e.g. employee/student number.

e.g. model number “Ryzen 7 3700X”. But are “CPU model” and “model - name” always the same concept?

An [approximate string matching](#) algorithm might help, like [difflib.get_close_matches](#) or [FuzzyWuzzy](#).

Or maybe you know about your data. e.g. two stores with the same name less than 500 m apart are unlikely: they're probably the same store.

Unless it's “Starbucks”.

Regular Expressions

Often, what you get is strings that have something useful in them. e.g. tweets, scraped HTML pages.

If there's a little structure, a *regular expression* can be a good way to extract what you need.

For example, looking for tweets with text “vote for party” or “voting for party”. This can be done with [Series.str.extract](#):

```
tweets = pd.DataFrame({'text': [
    "I'm voting for Green",
    'Get out and vote for @terrybeech!',
    'How do I vote?'
]})
parties = tweets['text'].str.extract(
    r'vot(e|ing) for (.*?)', expand=False)[1]
print(parties)
```

```
0      Green
1  @terrybeech!
2      NaN
Name: 1, dtype: object
```

... or with [Python's re](#):

```
import re
party_re = re.compile(r'vot(e|ing) for (.*?)')
def get_party(txt):
    match = party_re.search(txt)
    if match:
        return match.group(2)
    else:
        return None

parties = tweets['text'].apply(get_party)
print(parties)
```

```
0      Green
1      @terrybeech!
2      None
Name: text, dtype: object
```

Python re

A few basics about Python's regular expression library (which is the same syntax as Pandas' regular expressions)...

Regular expressions contain a lot of backslashes, but those are special in Python strings. It's common to use [Python raw string literals](#) for regex patterns

```
'\n' # one character: newline
r'\n' # two characters: backslash, n
```

If you're using a regex a lot (and you probably are), you should compile it **once** and use many times:

```
import re
input_pattern = re.compile(r'...')
```

Then you can use the `.match()` (look at start of string) or `.search()` method (look anywhere in the string) to find the pattern (or not):

```
m = input_pattern.match(input_string)
if m:
    # handle the match
else:
    # no match: handle that
```

Some things you can do in regular expressions:

- `r'x'` matches an “x” character
- `r'.'` matches any single character
- `r'\.'` matches a single “.” character
- `r'x*'` matches 0 or more “x”
- `r'x+'` matches 1 or more “x”
- `r'x?'` matches “x” but it's optional (0 or 1)

- `r'^'` matches start of string
- `r'$'` matches end of string
- `r'[ab\.]+'` matches 1+ “a” or “b” or “.”
- `r'\d+'` matches 1+ digits
- `r'\s*'` matches 0+ spaces (`[\t\n\r\f\v]`)
- `r'\S*'` matches 0+ non-spaces characters

e.g. we have (simplified) web log files like this:

```
192.168.10.11 "GET / HTTP/1.1" 200
10.11.12.13 "GET /missing.html HTTP/1.0" 404
192.168.10.11 "GET /page.html HTTP/1.1" 200
```

... and want to disassemble to get the parts out.

Can do that with a pattern like this:

```
import re
log_pattern = re.compile(
    r'^\d+\.\d+\.\d+\.\d+ "GET \S+ HTTP/\d\.\d" \d+$')
```

But could there be IPv6 addresses? Is `HTTP/\d\.\d` always going to match the HTTP version? Other optional stuff?

Suggestion: be a little defensive.

```
with open('re-input.txt') as logfile:
    for line in logfile:
        m = log_pattern.match(line)
        if not m:
            raise ValueError('Bad input line: %r' % (line,))
        # :
```

But we're going to actually want to extract the parts. A (...) pattern captures the “...” part so it can be extracted later.

```
log_pattern = re.compile(
    r'^(\d+\.\d+\.\d+\.\d+) "GET (\S+) HTTP/\d\.\d" (\d+)$')
```

Then on a match object, `m.group(1)` will be the IP address match, etc.

If using `Series.str.extract`, it will become column '0' in the result.


```
log_pattern = re.compile(
    r'^(\d+\.\d+\.\d+\.\d+) "GET (\S+) HTTP/\d\.\d" (\d+)$')
with open('re-input.txt') as logfile:
    for line in logfile:
        m = log_pattern.match(line)
        if not m:
            raise ValueError('Bad input line: %r' % (line,))
        print((m.group(1), m.group(2), m.group(3)))
```

```
('192.168.10.11', '/', '200')
('10.11.12.13', '/missing.html', '404')
('192.168.10.11', '/page.html', '200')
```

Regex Summary

You can get a lot done with regular expressions.

... but don't get fooled into thinking they're the best tool for every string processing task.

[CMPT 353](#). Copyright © 2017–2021 Steven Bergner. 

[Course Notes Home](#). CMPT 353, Fall 2021. Copyright © 2017–2021 Steven Bergner & Greg Baker.

