DSC 201: Data Analysis & Visualization

Data Cleaning

Dr. David Koop
Dirty Data
Numeric Outliers

ages of employees (US)

median 37
mean 58.52632
variance 9252.041

[J. Hellerstein via J. Canny et al.]
FINDINGS

we got about the future of the data science, the most salient takeaway was how excited our respondents were about the evolution of the field. They cited things in their own practice, how they saw their jobs getting more interesting and less repetitive, all while expressing a real and broad enthusiasm about the value of the work in their organization.

As data science becomes more commonplace and simultaneously a bit demystified, we expect this trend to continue as well. After all, last year's respondents were just as excited about their work (about 79% were "satisfied" or better).

How a Data Scientist Spends Their Day

Here's where the popular view of data scientists diverges pretty significantly from reality. Generally, we think of data scientists building algorithms, exploring data, and doing predictive analysis. That's actually not what they spend most of their time doing, however.

As you can see from the chart above, 3 out of every 5 data scientists we surveyed actually spend the most time cleaning and organizing data. You may have heard this referred to as "data wrangling" or compared to digital janitor work. Everything from list verification to removing commas to debugging databases–that time adds up and it adds up immensely. Messy data is by far the more time-consuming aspect of the typical data scientist's work flow. And nearly 60% said they simply spent too much time doing it.

What data scientists spend the most time doing

- Building training sets: 3%
- Cleaning and organizing data: 60%
- Collecting data sets: 19%
- Mining data for patterns: 9%
- Refining algorithms: 4%
- Other: 5%

This takes a lot of time!

[CrowdFlower Data Science Report, 2016]
Why That’s a Problem

Simply put, data wrangling isn’t fun. It takes forever. In fact, a few years back, the New York Times estimated that up to 80% of a data scientist’s time is spent doing this sort of work.

Here, it’s necessary to point out that data cleaning is incredibly important. You can’t do the sort of work data scientists truly enjoy doing with messy data. It needs to be cleaned, labeled, and enriched before you can trust the output.

The problem here is two-fold. One: data scientists simply don’t like doing this kind of work, and, as mentioned, this kind of work takes up most of their time. We asked our respondents what was the least enjoyable part of their job.

They had this to say:

Note how those last two charts mirror each other. The things data scientists do most are the things they enjoy least. Last year, we found that respondents far prefer doing the more creative, interesting parts of their job, things like predictive analysis and mining data for patterns. That’s where the real value comes. But again, you simply can’t do that work unless the data is properly labeled. And nobody likes labeling data.

Do Data Scientists Have What They Need?

With a shortage of data scientists out there in the world, we wanted to find out if they thought they were properly supported in their job. After all, when you need more data scientists, you’ll often find a single person doing the work of several.

What’s the least enjoyable part of data science?

- Building training sets: 10%
- Cleaning and organizing data: 57%
- Collecting data sets: 21%
- Mining data for patterns: 3%
- Refining algorithms: 4%
- Other: 5%

[CrowdFlower Data Science Report, 2016]
Dirty Data: Data Scientist's View

• Combination of:
  - Statistician's View: data has non-ideal samples for model
  - Database Expert's View: missing data, corrupted data
  - Domain Expert's View: data doesn't pass the smell test

• All of the views present problems with the data

• The goal may dictate the solutions:
  - Median value: don't worry too much about crazy outliers
  - Generally, aggregation is less susceptible by numeric errors
  - Be careful, the data may be correct…
Be careful how you detect dirty data

• The appearance of a hole in the earth’s ozone layer over Antarctica, first detected in 1976, was so unexpected that scientists didn’t pay attention to what their instruments were telling them; they thought their instruments were malfunctioning.

  – National Center for Atmospheric Research
Types of Dirty Data Problems

- Separator Issues: e.g. CSV without respecting double quotes
- Naming Conventions: NYC vs. New York
- Missing required fields, e.g. key
- Different representations: 2 vs. two
- Truncated data: "Janice Keihanaikukauakahihuliheekahaunaele" becomes "Janice Keihanaikukauakahihuliheek" on Hawaii license
- Redundant records: may be exactly the same or have some overlap
- Formatting issues: 2017-11-07 vs. 07/11/2017 vs. 11/07/2017
Assignment 4

- hurdat2.txt to hurdat2.csv
Data Cleaning
Handling Missing Data

• Filtering out missing data:
  - Can choose rows or columns

• Filling in missing data:
  - with a default value
  - with an interpolated value

• In pandas:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropna</td>
<td>Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.</td>
</tr>
<tr>
<td>fillna</td>
<td>Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.</td>
</tr>
<tr>
<td>isnull</td>
<td>Return boolean values indicating which values are missing/NA.</td>
</tr>
<tr>
<td>notnull</td>
<td>Negation of isnull.</td>
</tr>
</tbody>
</table>

In [10]:
string_data = pd.Series(["aardvark", "artichoke", np.nan, "avocado"])

In [11]:
string_data
Out[11]:
0    aardvark
1   artichoke
2      NaN
3    avocado
dtype: object

In [12]:
string_data.isnull()
Out[12]:
0    False
1    False
2    True
3    False
dtype: bool

There is work ongoing in the pandas project to improve the internal details of how missing data is handled, but the user API functions, like pandas.isnull, abstract away many of the annoying details. See Table 7-1 for a list of some functions related to missing data handling.

Table 7-1. NA handling methods

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropna</td>
<td>Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.</td>
</tr>
<tr>
<td>fillna</td>
<td>Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.</td>
</tr>
<tr>
<td>isnull</td>
<td>Return boolean values indicating which values are missing/NA.</td>
</tr>
<tr>
<td>notnull</td>
<td>Negation of isnull.</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
Filling in missing data

- `fillna` arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value</code></td>
<td>Scalar value or dict-like object to use to fill missing values</td>
</tr>
<tr>
<td><code>method</code></td>
<td>Interpolation; by default 'ffill' if function called with no other arguments</td>
</tr>
<tr>
<td><code>axis</code></td>
<td>Axis to fill on; default <code>axis=0</code></td>
</tr>
<tr>
<td><code>inplace</code></td>
<td>Modify the calling object without producing a copy</td>
</tr>
<tr>
<td><code>limit</code></td>
<td>For forward and backward filling, maximum number of consecutive periods to fill</td>
</tr>
</tbody>
</table>
Filtering and Cleaning Data

• Find duplicates
  - `duplicated`: returns boolean Series indicating whether row is a duplicate—first instance is not marked as a duplicate

• Remove duplicates:
  - `drop_duplicates`: drops all rows where `duplicated` is True
  - `keep`: which value to keep (first or last)

• Can pass specific columns to check for duplicates, e.g. check only key column
Changing Data

- Convert strings to upper/lower case
- Convert Fahrenheit temperatures to Celsius
- Create a new column based on another column

```python
In [56]: lowercased
Out[56]:
0    bacon
1  pulled pork
2      bacon
3   pastrami
4  corned beef
5      bacon
6   pastrami
7    honey ham
8    nova lox
Name: food, dtype: object

meat_to_animal = {
    'bacon': 'pig',
    'pulled pork': 'pig',
    'pastrami': 'cow',
    'corned beef': 'cow',
    'honey ham': 'pig',
    'nova lox': 'salmon'
}

In [57]: data['animal'] = lowercased.map(meat_to_animal)

In [58]: data
Out[58]:
food    ounces    animal
   0     bacon      4.0      pig
   1  pulled pork    3.0      pig
   2     bacon     12.0      pig
   3  Pastrami     6.0      cow
   4  corned beef    7.5      cow
   5   Bacon       8.0      pig
   6   pastrami     3.0      cow
   7    honey ham    5.0      pig
   8    nova lox    6.0  salmon
```

[W. McKinney, Python for Data Analysis]
Replacing Values

- fillna is a special case
- What if -999 in our dataset was identified as a missing value?

```
In [61]: data
Out[61]:
   0   1.0
  1  -999.0
  2   2.0
  3  -999.0
  4  -1000.0
  5    3.0
dtype: float64
```
```
In [62]: data.replace(-999, np.nan)
Out[62]:
   0   1.0
  1  NaN
  2   2.0
  3  NaN
  4  NaN
  5    3.0
dtype: float64
```

- Can pass list of values or dictionary to change different values
Clamping Values

- Values above or below a specified threshold are set to a max/min value

```python
In [93]: data.describe()
Out[93]:

         0         1         2         3
count 1000.000000 1000.000000 1000.000000 1000.000000
mean   0.049091   0.026112  -0.002544  -0.051827
std    0.996947   1.007458   0.995232   0.998311
min   -3.645860  -3.184377  -3.745356  -3.428254
25%   -0.599807  -0.612162  -0.687373  -0.747478
50%   0.047101   -0.013609  -0.022158  -0.088274
75%   0.756646    0.695298   0.699046   0.623331
max   2.653656    3.525865   2.735527   3.366626
```

```python
In [97]: data[np.abs(data) > 3] = np.sign(data) * 3
```

```python
In [98]: data.describe()
Out[98]:

         0         1         2         3
count 1000.000000 1000.000000 1000.000000 1000.000000
mean   0.050286   0.025567  -0.001399  -0.051765
std    0.992920   1.004214   0.991414   0.995761
min   -3.000000  -3.000000  -3.000000  -3.000000
25%   -0.599807  -0.612162  -0.687373  -0.747478
50%   0.047101   -0.013609  -0.022158  -0.088274
75%   0.756646    0.695298   0.699046   0.623331
max   2.653656    3.000000   2.735527   3.000000
```
Computing Indicator Values

• Useful for machine learning
• Want to take possible values and map them to 0-1 indicators
• Example:

```python
In [109]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                       'data1': range(6)})
.....:

In [110]: pd.get_dummies(df['key'])
Out[110]:
   a  b  c
0  0  0  1
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

• Example: Genres in movies
String Transformation

- One of the reasons for Python's popularity is string/text processing
  - `split(<delimiter>)`: break a string into pieces:
    - `s = "12,13, 14"
      slist = s.split(',', '') # ["12", "13", " 14"]`
  - `<delimiter>.join([<str>])`: join several strings by a delimiter
    - `":".join(slist) # "12:13: 14"`
  - `strip()`: remove leading and trailing whitespace
    - `[p.strip() for p in slist] # ["12", "13", "14"]`
  - `replace(<from>,<to>)`: change substrings to another substring
    - `s.replace(',', ':') # "12:13: 14"
  - `upper()/lower()`: casing
    - "AbCd".upper () # "ABCD"
    - "AbCd".lower() # "abcd"
String Transformations

- **index(<str>):** find where a substring first occurs (Error if not found)

  - `s = "12,13, 14"
    - `s.index(',')` # 2
    - `s.index(':')` # ValueError raised

- **find(<str>):** same as index but -1 if not found

  - `s.find(',')` # 2
    - `s.find(':')` # -1

- **startswith() / endswith():** boolean checks for string occurrence

  - `s.startswith("1")` # True
    - `s.endswith("5")` # False
## String Methods

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Return the number of non-overlapping occurrences of substring in the string.</td>
</tr>
<tr>
<td>endswith</td>
<td>Returns True if string ends with suffix.</td>
</tr>
<tr>
<td>startswith</td>
<td>Returns True if string starts with prefix.</td>
</tr>
<tr>
<td>join</td>
<td>Use string as delimiter for concatenating a sequence of other strings.</td>
</tr>
<tr>
<td>index</td>
<td>Return position of first character in substring if found in the string; raises ValueError if not found.</td>
</tr>
<tr>
<td>find</td>
<td>Return position of first character of first occurrence of substring in the string; like index, but returns –1 if not found.</td>
</tr>
<tr>
<td>rfind</td>
<td>Return position of first character of last occurrence of substring in the string; returns –1 if not found.</td>
</tr>
<tr>
<td>replace</td>
<td>Replace occurrences of string with another string.</td>
</tr>
<tr>
<td>strip, rstrip, lstrip</td>
<td>Trim whitespace, including newlines; equivalent to x.strip() (and rstrip, lstrip, respectively) for each element.</td>
</tr>
<tr>
<td>split</td>
<td>Break string into list of substrings using passed delimiter.</td>
</tr>
<tr>
<td>lower</td>
<td>Convert alphabet characters to lowercase.</td>
</tr>
<tr>
<td>upper</td>
<td>Convert alphabet characters to uppercase.</td>
</tr>
<tr>
<td>casefold</td>
<td>Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.</td>
</tr>
<tr>
<td>ljust, rjust</td>
<td>Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.</td>
</tr>
</tbody>
</table>

Regular Expressions

Regular expressions provide a flexible way to search or match (often more complex) string patterns in text. A single expression, commonly called a regex, is a string formed according to the regular expression language. Python's built-in re module is responsible for applying regular expressions to strings; I'll give a number of examples of its use here.

The art of writing regular expressions could be a chapter of its own and thus is outside the book's scope. There are many excellent tutorials and references available on the internet and in other books.

The `re` module functions fall into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes. Let's look at a simple example:
Regular Expressions

- AKA regex
- A syntax to better specify how to decompose strings
- Look for patterns rather than specific characters
- "31" in "The last day of December is 12/31/2016."
- May work for some questions but now suppose I have other lines like: "The last day of September is 9/30/2016."
- ...and I want to find dates that look like:
  - \d+/%d+/\d+
- Cannot search for every combination!
Regular Expressions

• Character classes:
  - \d = digits
  - \s = spaces
  - \w = word character [a-zA-Z0-9_]
  - [a-z] = lowercase letters (square brackets indicate a set of chars)

• Repeating characters or patterns
  - + = one or more (any number)
  - * = zero or more (any number)
  - ? = zero or one
  - {<number>} = a specific number (or range) of occurrences
Regular Expressions in Python

- import re
- re.search(<pattern>, <str_to_check>)
  - Returns None if no match, information about the match otherwise
- Capturing information about what is in a string → parentheses
- (\d+)/\d+/\d+ will capture information about the month
- match = re.search('(^\d+)/\d+/\d+', '12/31/2016')
  if match:
    match.group() # 12
- re.findall(<pattern>, <str_to_check>)
  - Finds all matches in the string, search only finds the first match
- Can pass in flags to alter methods: e.g. re.IGNORECASE
Regular Expression Methods

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findall</td>
<td>Return all non-overlapping matching patterns in a string as a list</td>
</tr>
<tr>
<td>finditer</td>
<td>Like findall, but returns an iterator</td>
</tr>
<tr>
<td>match</td>
<td>Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None</td>
</tr>
<tr>
<td>search</td>
<td>Scan string for match to pattern; returning a match object if so; unlike match, the match can be anywhere in the string as opposed to only at the beginning</td>
</tr>
<tr>
<td>split</td>
<td>Break string into pieces at each occurrence of pattern</td>
</tr>
<tr>
<td>sub, subn</td>
<td>Replace all (sub) or first n occurrences (subn) of pattern in string with replacement expression; use symbols \1, \2, ... to refer to match group elements in the replacement string</td>
</tr>
</tbody>
</table>

Table 7-4. Regular expression methods

Sub also has access to groups in each match using special symbols like \1 and \2. The symbol \1 corresponds to the first matched group, \2 corresponds to the second, and so forth:

In \[166\]:

```python
print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
```

Dave Username: dave, Domain: google, Suffix: com

Steve Username: steve, Domain: gmail, Suffix: com

Rob Username: rob, Domain: gmail, Suffix: com

Ryan Username: ryan, Domain: yahoo, Suffix: com

There is much more to regular expressions in Python, most of which is outside the book’s scope. Table 7-4 provides a brief summary.

Table 7-4. Regular expression methods

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findall</td>
<td>Return all non-overlapping matching patterns in a string as a list</td>
</tr>
<tr>
<td>finditer</td>
<td>Like findall, but returns an iterator</td>
</tr>
<tr>
<td>match</td>
<td>Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None</td>
</tr>
<tr>
<td>search</td>
<td>Scan string for match to pattern; returning a match object if so; unlike match, the match can be anywhere in the string as opposed to only at the beginning</td>
</tr>
<tr>
<td>split</td>
<td>Break string into pieces at each occurrence of pattern</td>
</tr>
<tr>
<td>sub, subn</td>
<td>Replace all (sub) or first n occurrences (subn) of pattern in string with replacement expression; use symbols \1, \2, ... to refer to match group elements in the replacement string</td>
</tr>
</tbody>
</table>

Cleaning up a messy dataset for analysis often requires a lot of string munging and regularization. To complicate matters, a column containing strings will sometimes have missing data:

In \[167\]:

```python
data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com', '...': 'Rob': 'rob@gmail.com', 'Wes': np.nan}
```

In \[168\]:

```python
data = pd.Series(data)
```

In \[169\]:

```python
data
```

Dave dave@google.com
Rob rob@gmail.com
Steve steve@gmail.com
Wes NaN
dtype: object

[W. McKinney, Python for Data Analysis]
Pandas String Methods

• Any column or series can have the string methods (e.g. replace, split) applied to the entire series

• Fast (vectorized) on whole columns or datasets

• use .str.<method_name>

• .str is important!

- data = pd.Series({'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com', 'Rob': 'rob@gmail.com', 'Wes': np.nan})

  data.str.contains('gmail')
  data.str.split('@').str[1]
  data.str[-3:]
Pandas String Methods with RegExs

In [172]: pattern
Out[172]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.(\[A-Z\]{2,4})'

In [173]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[173]:
Dave     [(dave, google, com)]
Rob      [(rob, gmail, com)]
Steve    [(steve, gmail, com)]
Wes      NaN
dtype: object

In [174]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [175]: matches
Out[175]:
Dave      True
Rob       True
Steve     True
Wes       NaN
dtype: object

[W. McKinney, Python for Data Analysis]