Advanced Data Management (CSCI 490/680)

Data Wrangling

Dr. David Koop

DataFrame Access and Manipulation

- `df.values → 2D NumPy array`
- Accessing a column:
  - `df["<column>"]`
  - `df.<column>`
  - Both return Series
  - Dot syntax only works when the column is a valid identifier
- Assigning to a column:

```
- df["<column>"] = <scalar> # all cells set to same value
- df["<column>"] = <array>  # values set in order
- df["<column>"] = <series> # values set according to match
                            # between df and series indexes
```

Indexing

- Same as with NumPy arrays but can use Series's index labels
- Slicing with labels: NumPy is **exclusive**, Pandas is **inclusive**!
  - ```
    s = Series(np.arange(4))
    s[0:2] # gives two values like numpy
    ```
  - ```
    s = Series(np.arange(4), index=['a', 'b', 'c', 'd'])
    s['a':'c'] # gives three values, not two!
    ```
- Obtaining data subsets
  - `[]`: get columns by label
  - `loc`: get rows/cols by label
  - `iloc`: get rows/cols by position (integer index)
- For single cells (scalars), also have `at` and `iat`

Indexing

- `s = Series(np.arange(4.), index=[4,3,2,1])`
- `s[3]`
- `s.loc[3]`
- `s.iloc[3]`
- `s2 = pd.Series(np.arange(4), index=['a','b','c','d'])`
- `s2[3]`

Filtering

- Same as with numpy arrays but allows use of column-based criteria
  - `data[data < 5] = 0`
  - `data['three'] > 5]`

- `data < 5` → boolean data frame, can be used to select specific elements

Arithmetic

- Add, subtract, multiply, and divide are element-wise like numpy
- …but use labels to align
- …and missing labels lead to `NaN` (not a number) values

```
In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

```
In [29]: obj4
Out[29]:
California      NaN
Ohio          35000
Oregon        16000
Texas         71000
dtype: float64
```

```
In [30]: obj3 + obj4
Out[30]:
California        NaN
Ohio            70000
Oregon          32000
Texas          142000
Utah              NaN
dtype: float64
```

- also have `.add`, `.subtract`, … that allow `fill_value` argument
- `obj3.add(obj4, fill_value=0)`

Arithmetic between DataFrames and Series

- Broadcasting: e.g. apply single row operation across all rows
- Example:

```
In [148]: frame
Out[148]:
        b   d   e
Utah    0   1   2
Ohio    3   4   5
Texas   6   7   8
Oregon  9  10  11
```

```
In [149]: series
Out[149]:
b    0
d    1
e    2
Name: Utah, dtype: float64
```

```
In [150]: frame - series
Out[150]:
        b  d  e
Utah    0  0  0
Ohio    3  3  3
Texas   6  6  6
Oregon  9  9  9
```

- To broadcast over **columns**, use methods (`.add`, …)

```
In [154]: frame
Out[154]:
        b   d   e
Utah    0   1   2
Ohio    3   4   5
Texas   6   7   8
Oregon  9  10  11
```

```
In [155]: series3
Out[155]:
Utah       1
Ohio       4
Texas      7
Oregon    10
Name: d, dtype: float64
```

```
In [156]: frame.sub(series3, axis=0)
Out[156]:
        b  d  e
Utah   -1  0  1
Ohio   -1  0  1
Texas  -1  0  1
Oregon -1  0  1
```

Sorting by Index (sort_index)

- Sort by index (lexicographical):

```
In [168]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [169]: obj.sort_index()
Out[169]:
a    1
b    2
c    3
d    0
dtype: int64
```

- DataFrame sorting:

```
In [170]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
   .....:                   columns=['d', 'a', 'b', 'c'])
```

```
In [171]: frame.sort_index()
Out[171]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3
```

```
In [172]: frame.sort_index(axis=1)
Out[172]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

- axis controls sort rows (0) vs. sort columns (1)

Sorting by Value (sort_values)

- `sort_values` method on series
  - `obj.sort_values()`
- Missing values (`NaN`) are at the end by default (`na_position` controls, can be first)
- `sort_values` on DataFrame:
  - `df.sort_values(<list-of-columns>)`
  - `df.sort_values(by=['a', 'b'])`
  - Can also use `axis=1` to sort by index labels

Assignment 2

- Same data as A1, different version of the dataset
- Dealing with the raw data now
- Same questions as A1, but use pandas
- CS680 students + some questions about problems with the data

Ranking

- `rank()` method:

```
In [182]: obj = Series([7, -5, 7, 4, 2, 0, 4])

In [183]: obj.rank()
Out[183]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

- `ascending` and `method` arguments:

```
In [185]: obj.rank(ascending=False, method='max')
Out[185]:
0    2
1    7
2    2
3    4
4    5
5    6
6    4
dtype: float64
```

- Works on data frames, too

Statistics

- `sum`: column sums (`axis=1` gives sums over rows)
- missing values are excluded unless the whole slice is `NaN`
- `idxmax, idxmin` are like argmax, argmin (return index)
- `describe`: shortcut for easy stats!

```
In [204]: df.describe()
Out[204]:
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

```
In [205]: obj = Series(['a', 'a', 'b', 'c'] * 4)

In [206]: obj.describe()
Out[206]:
count     16
unique     3
top        a
freq       8
dtype: object
```

Statistics

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

[W. McKinney, Python for Data Analysis]

Unique Values and Value Counts

- `unique` returns an array with only the unique values (no index)
  - ```
    s = Series(['c','a','d','a','a','b','b','c','c'])
    s.unique() # array(['c', 'a', 'd', 'b'])
    ```
- Data Frames use `drop_duplicates`
- `value_counts` returns a Series with index frequencies:
  - ```
    s.value_counts() # Series({'c': 3,'a': 3,'b': 2,'d': 1})
    ```

Handling Missing Data

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return like-type object containing boolean values indicating which values are missing / NA.
`notnull`	Negation of `isnull`.

[W. McKinney, Python for Data Analysis]

Data Formats

Comma-separated values (CSV) Format

- Comma is a field separator, newlines denote records
  - ```
    a,b,c,d,message
    1,2,3,4,hello
    5,6,7,8,world
    9,10,11,12,foo
    ```
- May have a header (`a,b,c,d,message`), but not required
- No type information: we do not know what the columns are (numbers, strings, floating point, etc.)
  - Default: just keep everything as a string
  - Type inference: Figure out the type to make each column based on values
- What about commas in a value? → double quotes
    ```

Delimiter-separated Values

- Comma is a **delimiter**, specifies boundary between fields
- Could be a tab, pipe (|), or perhaps spaces instead
- All of these follow similar styles to CSV

Fixed-width Format

- Old school
- Each field gets a certain number of spots in the file
- Example:
  - ```
    id8141    360.242940   149.910199   11950.7
    id1594    444.953632   166.985655   11788.4
    id1849    364.136849   183.628767   11806.2
    id1230    413.836124   184.375703   11916.8
    id1948    502.953953   173.237159   12468.3
    ```
- Specify exact character ranges for each field, e.g. 0-6 is the id
    ```

Reading & Writing Data

Reading Data in Python

- Use the `open()` method to open a file for reading
  - `f = open('huck-finn.txt')`
- Usually, add an `'r'` as the second parameter to indicate "read"
- Can iterate through the file (think of the file as a collection of lines):
  - ```
    f = open('huck-finn.txt', 'r')
    for line in f:
        if 'Huckleberry' in line:
            print(line.strip())
    ```
- Using `line.strip()` because the read includes the newline, and print writes a newline so we would have double-spaced text
- Closing the file: `f.close()`
    ```

With Statement: Improved File Handling

- With statement does "enter" and "exit" handling (similar to the finally clause):
- In the previous example, we need to remember to call `f.close()`
- Using a with statement, this is done automatically:

```
- with open('huck-finn.txt', 'r') as f:
      for line in f:
          if 'Huckleberry' in line:
              print(line.strip())
```

- This is more important for writing files!

```
- with open('output.txt', 'w') as f:
      for k, v in counts.items():
          f.write(k + ': ' + v + '\n')
```

- Without `with`, we need `f.close()`

Reading & Writing Data in Pandas

Format	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

[https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html]

Types of arguments for readers

- Indexing: choose a column to index the data, get column names from file or user
- Type inference and data conversion: automatic or user-defined
- Datetime parsing: can combine information from multiple columns
- Iterating: deal with very large files
- Unclean Data: skip rows (e.g. comments) or deal with formatted numbers (e.g. 1,000,345)

read_csv

- Convenient method to read csv files
- Lots of different options to help get data into the desired format
- Basic: `df = pd.read_csv(fname)`
- Parameters:
  - `path`: where to read the data from
  - `sep` (or `delimiter`): the delimiter (`','`, `' '`, `'\t'`, `'\s+'`)
  - `header`: if `None`, no header
  - `index_col`: which column to use as the row index
  - `names`: list of header names (e.g. if the file has no header)
  - `skiprows`: number of list of lines to skip

More read_csv/read_tables arguments

Argument	Description
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip.
`na_values`	Sequence of values to replace with NA.
`comment`	Character(s) to split comments off the end of lines.
`parse_dates`	Attempt to parse data to `datetime`; `False` by default. If `True`, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns).
`keep_date_col`	If joining columns to parse date, keep the joined columns; `False` by default.
`converters`	Dict containing column number of name mapping to functions (e.g., `{'foo': f}` would apply the function `f` to all values in the `'foo'` column).
`dayfirst`	When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); `False` by default.
`date_parser`	Function to use to parse dates.
`nrows`	Number of rows to read from beginning of file.
`iterator`	Return a `TextParser` object for reading file piecemeal.
`chunksize`	For iteration, size of file chunks.

[W. McKinney, Python for Data Analysis]

Chunked Reads

- With very large files, we may not want to read the entire file
- Why?
  - Time
  - Want to understand part of data before processing all of it
- Reading only a few rows:
  - `df = pd.read_csv('example.csv', nrows=5)`
- Reading chunks:
  - Get an iterator that returns the next chunk of the file
  - `chunker = pd.read_csv('example.csv', chunksize=1000)`
  - ```
    for piece in chunker:
        process_data(piece)
    ```
    ```

Python csv module

- Also, can read csv files outside of pandas using csv module
  - ```
    import csv
    with open('persons_of_concern.csv', 'r') as f:
        for i in range(3):
            next(f)
        reader = csv.reader(f)
        records = [r for r in reader] # r is a list
    ```
- or
  - ```
    import csv
    with open('persons_of_concern.csv', 'r') as f:
        for i in range(3):
            next(f)
        reader = csv.DictReader(f)
        records = [r for r in reader] # r is a dict
    ```

Writing CSV data with pandas

- Basic: `df.to_csv(<fname>)`
- Change delimiter with sep kwarg:
  - `df.to_csv('example.dsv', sep='|')`
- Change missing value representation
  - `df.to_csv('example.dsv', na_rep='NULL')`
- Don't write row or column labels:
  - `df.to_csv('example.csv', index=False, header=False)`
- Series may also be written to csv

eXtensible Markup Language (XML)

- Older, self-describing format with nesting; each field has tags
- Example:

```
- <INDICATOR>
    <INDICATOR_SEQ>373889</INDICATOR_SEQ>
    <PARENT_SEQ></PARENT_SEQ>
    <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
    <INDICATOR_NAME>Escalator Avail.</INDICATOR_NAME>
    <PERIOD_YEAR>2011</PERIOD_YEAR>
    <PERIOD_MONTH>12</PERIOD_MONTH>
    <CATEGORY>Service Indicators</CATEGORY>
    <FREQUENCY>M</FREQUENCY>
    <YTD_TARGET>97.00</YTD_TARGET>
  </INDICATOR>
```

- Top element is the **root**

XML

- No built-in method
- Use lxml library (also can use ElementTree)
- ```
  from lxml import objectify
  path = 'datasets/mta_perf/Performance_MNR.xml'
  parsed = objectify.parse(open(path))
  root = parsed.getroot()
  data = []
  skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
                 'DESIRED_CHANGE','DECIMAL_PLACES']
  for elt in root.INDICATOR:
      el_data = {}
      for child in elt.getchildren():
          if child.tag in skip_fields:
              continue
          el_data[child.tag] = child.pyval
      data.append(el_data)
  perf = pd.DataFrame(data)
  ```

[W. McKinney, Python for Data Analysis]
  ```

JavaScript Object Notation (JSON)

- A format for web data
- Looks very similar to python dictionaries and lists
- Example:
  - ```
    {"name": "Wes",
     "places_lived": ["United States", "Spain", "Germany"],
     "pet": null,
     "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
                  {"name": "Katie", "age": 33, "pet": "Cisco"}] }
    ```
- Only contains literals (no variables) but allows null
- Values: strings, arrays, dictionaries, numbers, booleans, or null
  - Dictionary keys must be strings
  - Quotation marks help differentiate string or numeric values
    ```

What is the problem with reading this data?

- ```
  [{"name": "Wes",
    "places_lived": ["United States", "Spain", "Germany"],
    "pet": null,
    "siblings": [
       {"name": "Scott", "age": 25, "pet": "Zuko"},
       {"name": "Katie", "age": 33, "pet": "Cisco"}]
   },
   {"name": "Nia",
    "address": {"street": "143 Main",
                "city": "New York",
                "state": "New York"},
    "pet": "Fido",
    "siblings": [
       {"name": "Jacques", "age": 15, "pet": "Fido"}]
   },
  …
  ]
  ```
  ```

Reading JSON data

- Python has a built-in `json` module
  - ```
    with open('example.json') as f:
        data = json.load(f)
    ```
  - Can also load/dump to strings:
    - `json.loads, json.dumps`
- Pandas has `read_json, to_json` methods
    ```

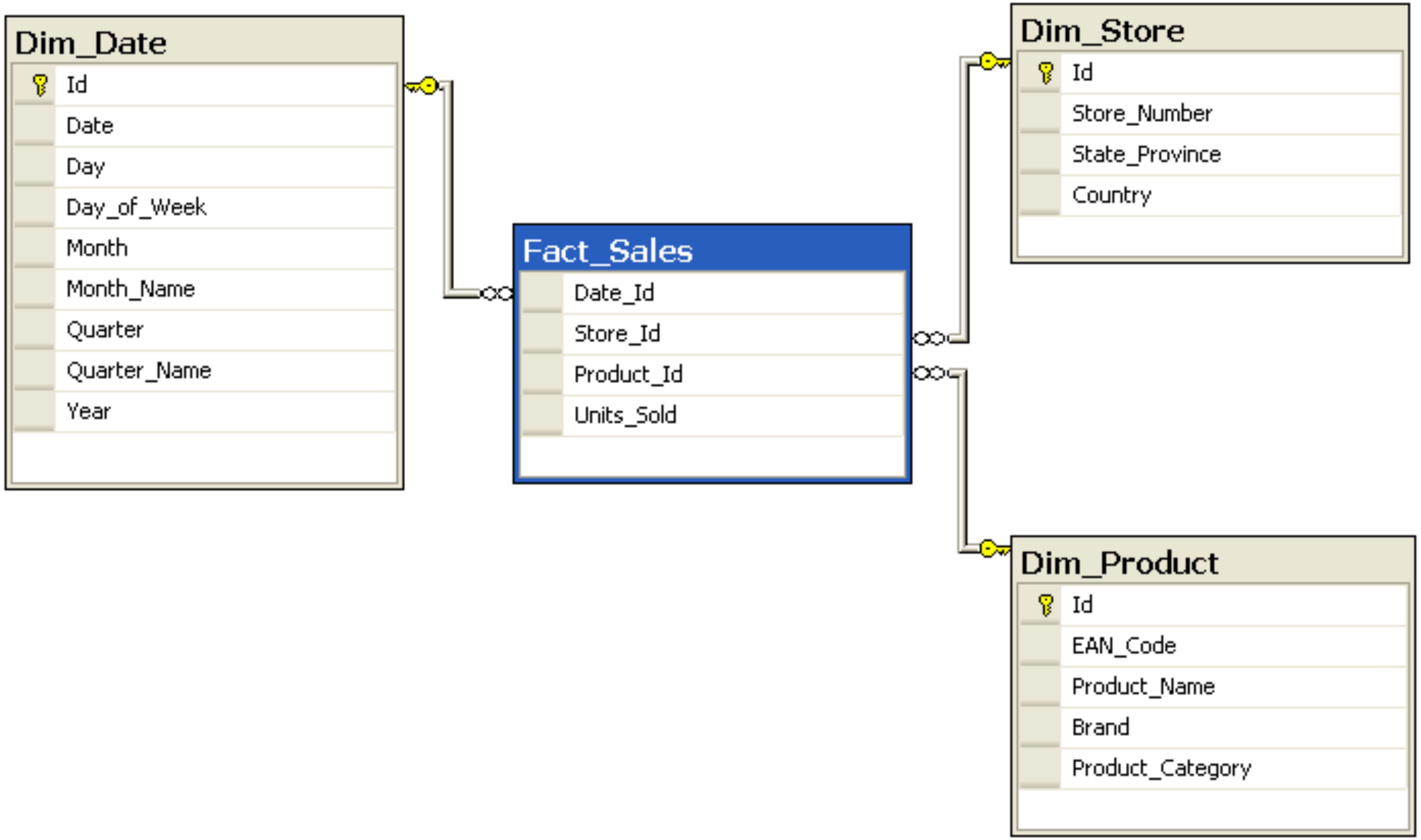
JSON Orientation

- Indication of expected JSON string format. Compatible JSON strings can be produced by `to_json()` with a corresponding orient value. The set of possible orients is:
  - `split`: dict like `{index -> [index],`
    `columns -> [columns],`
    `data -> [values]}`
  - `records`: list like `[{column -> value}, ... , {column -> value}]`
  - `index`: dict like `{index -> {column -> value}}`
  - `columns`: dict like `{column -> {index -> value}}`
  - `values`: just the values array

Binary Formats

- CSV, JSON, and XML are all text formats
- What is a binary format?
- Pickle: Python's built-in serialization
- HDF5: Library for storing large scientific data
  - Hierarchical Data Format, supports **compression**
  - Interfaces in C, Java, MATLAB, etc.
  - Use `pd.HDFStore` to access
  - Shortcuts: `read_hdf/to_hdf`, need to specify object
- Excel: need to specify sheet when a spreadsheet has multiple sheets
  - `pd.ExcelFile` or `pd.read_excel`

Databases



[Wikipedia]

Databases

- Relational databases are similar to multiple data frames but have many more features
  - links between tables via foreign keys
  - SQL to create, store, and query data
- sqlite3 is a simple database with built-in support in python
- Python has a database API which lets you access most database systems through a common API.

Python DBAPI Example

```
import sqlite3
query = """CREATE TABLE test(a VARCHAR(20), b VARCHAR(20),
                             c REAL, d INTEGER);"""
con = sqlite3.connect('mydata.sqlite')
con.execute(query)
con.commit()
# Insert some data
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"
con.executemany(stmt, data)
con.commit()
```

[W. McKinney, Python for Data Analysis]

Databases

- Similar syntax from other database systems (MySQL, Microsoft SQL Server, Oracle, etc.)
- SQLAlchemy: Python package that abstracts away differences between different database systems
- SQLAlchemy gives support for reading queries to data frame:
  - ```
    import sqlalchemy as sqla
    db = sqla.create_engine('sqlite:///mydata.sqlite')
    pd.read_sql('select * from test', db)
    ```
    ```

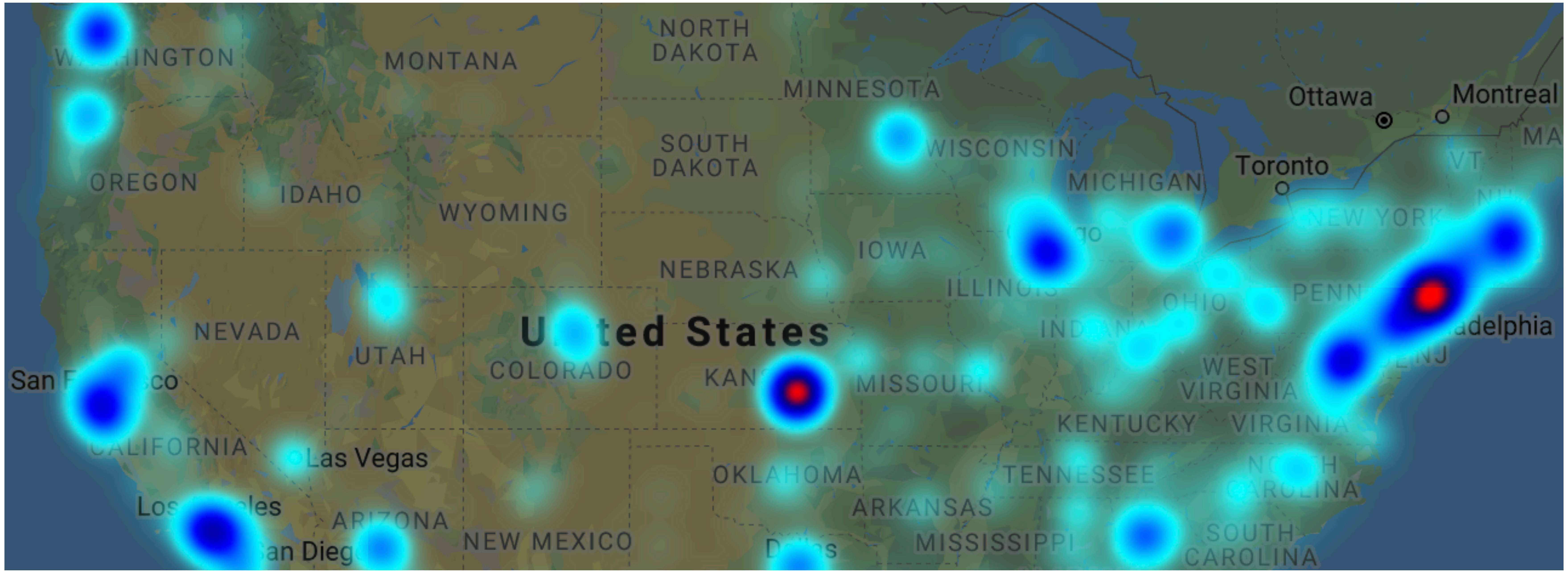
What if data isn't correct/trustworthy/in the right format?

Dirty Data

[Flickr]

Geolocation Errors

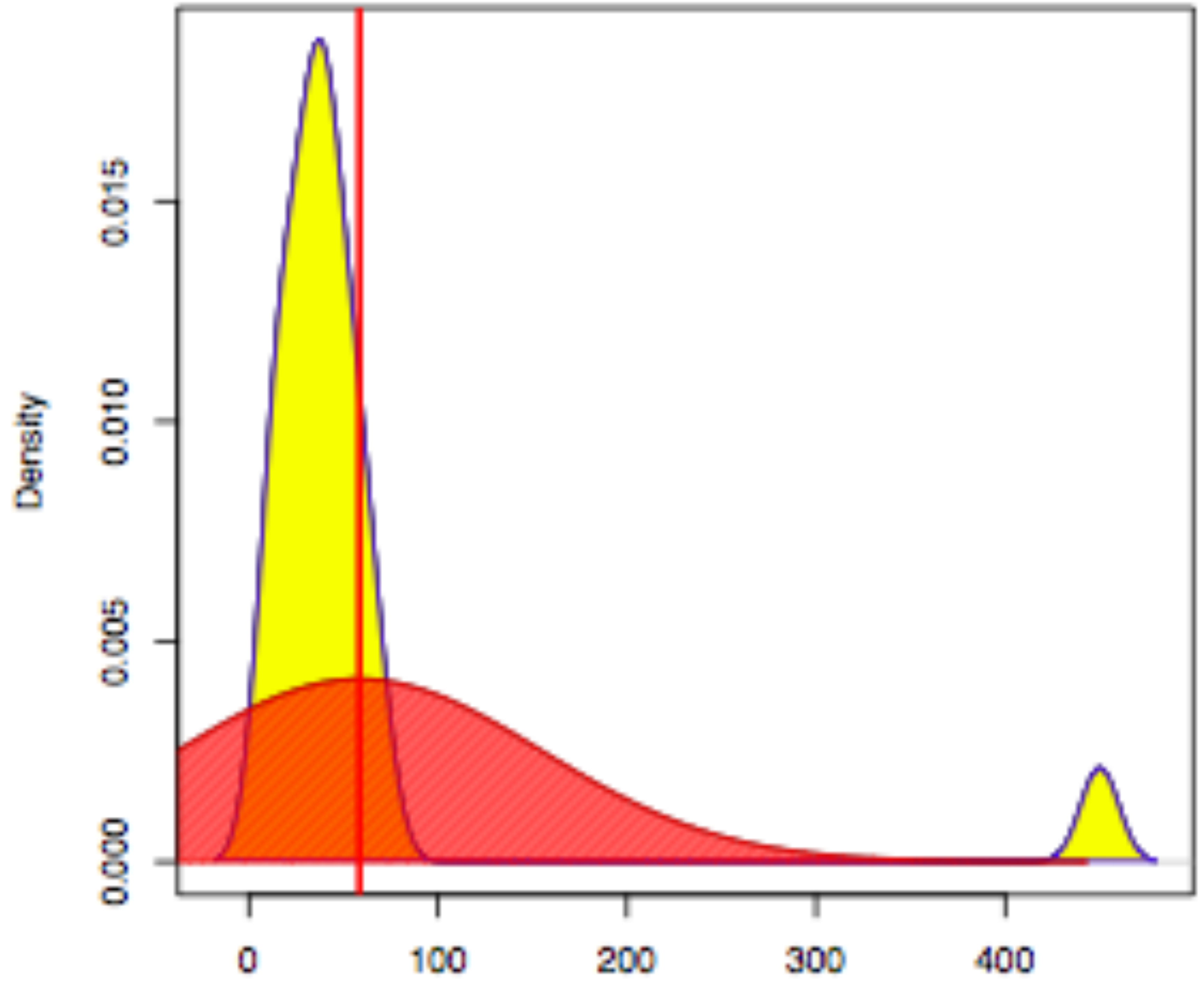
- Maxmind helps companies determine where users are located based on IP address
- "How a quiet Kansas home wound up with 600 million IP addresses and a world of trouble" [Washington Post, 2016]



Numeric Outliers

12	13	14	21	22	26	33	35	36	37	39	42	45	47	54	57	61	68	450

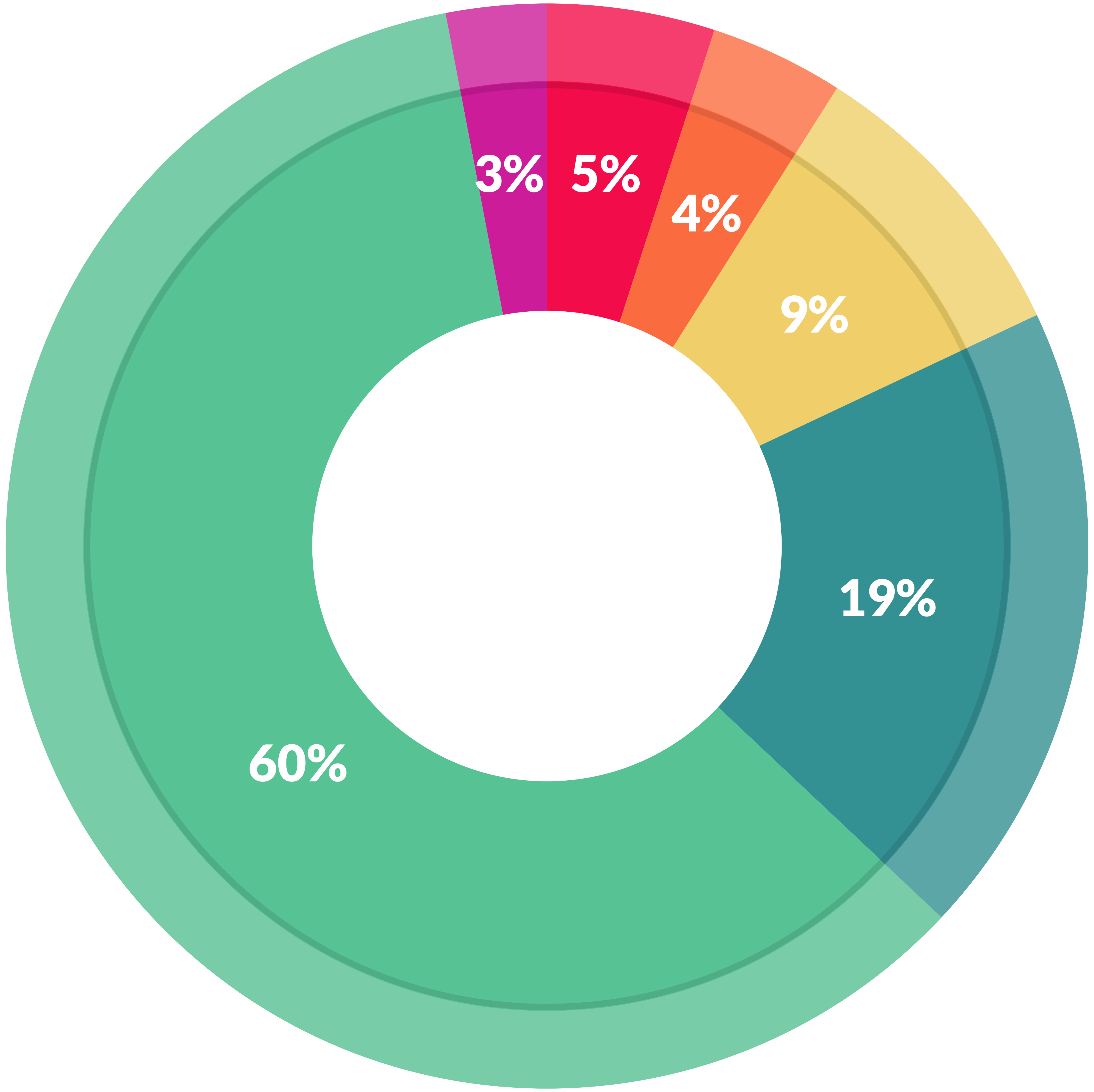
ages of employees (US)



- median 37
- mean 58.52632
- variance 9252.041

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

This takes a lot of time!

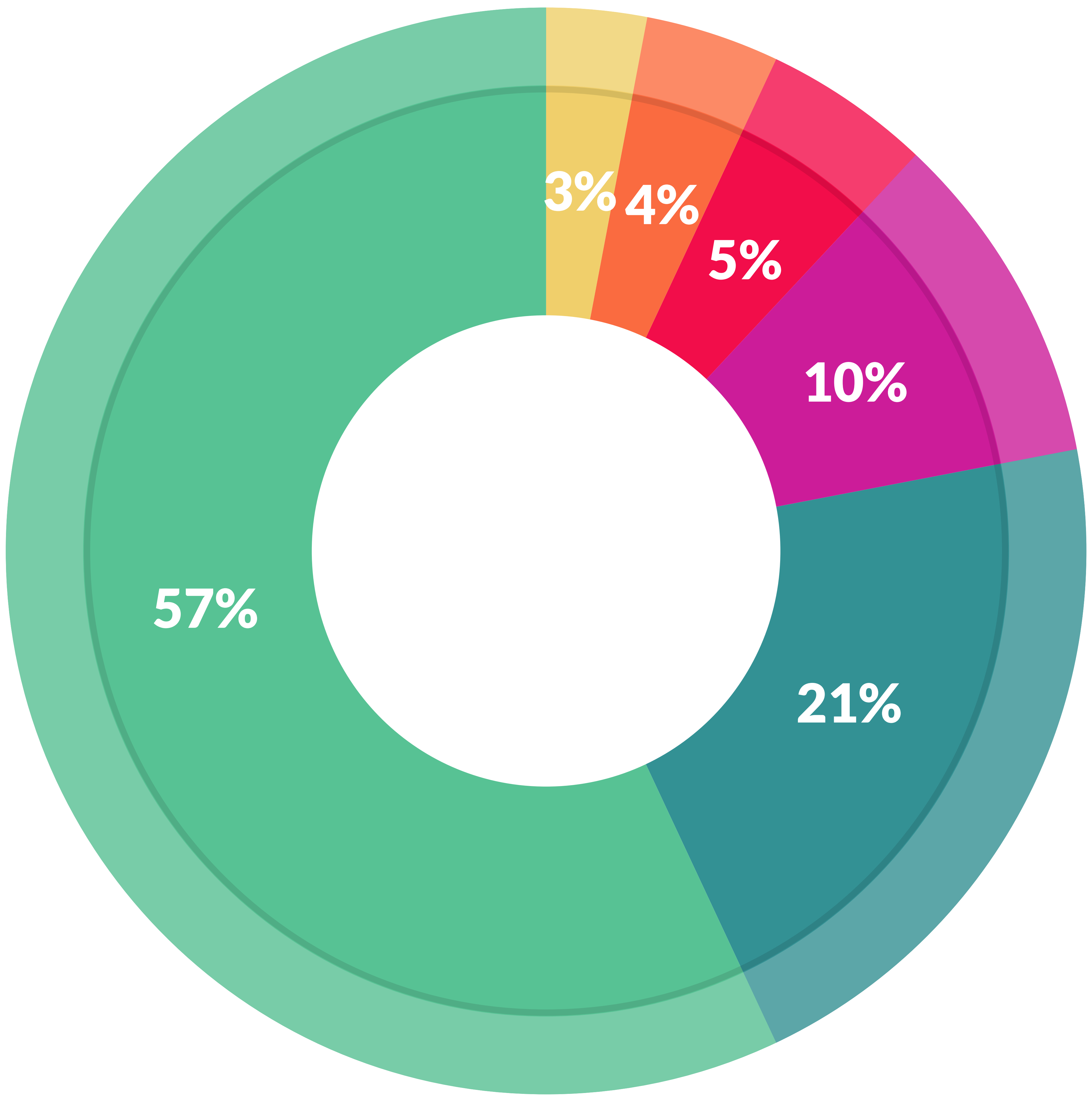


What data scientists spend the most time doing

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

[CrowdFlower Data Science Report, 2016]

…and it isn't the most fun thing to do



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: Statistician's View

- Some process produces the data
- Want a model but have non-ideal samples:
  - Distortion: some samples corrupted by a process
  - Selection bias: likelihood of a sample depends on its value
  - Left and right censorship: users come and go from scrutiny
  - Dependence: samples are not independent (e.g. social networks)
- You can add/augment models for different problems, but cannot model everything
- Trade-off between accuracy and simplicity

[J. Canny et al.]

Dirty Data: Database Expert's View

- Got a dataset
- Some values are missing, corrupted, wrong, duplicated
- Results are absolute (relational model)
- Better answers come from improving the quality of values in the dataset

[J. Canny et al.]

Dirty Data: Domain Expert's View

- Data doesn't look right
- Answer doesn't look right
- What happened?
- Domain experts carry an implicit model of the data they test against
- You don't always need to be a domain expert to do this
  - Can a person run 50 miles an hour?
  - Can a mountain on Earth be 50,000 feet above sea level?
  - Use common sense

[J. Canny et al.]

Dirty Data: Data Scientist's View

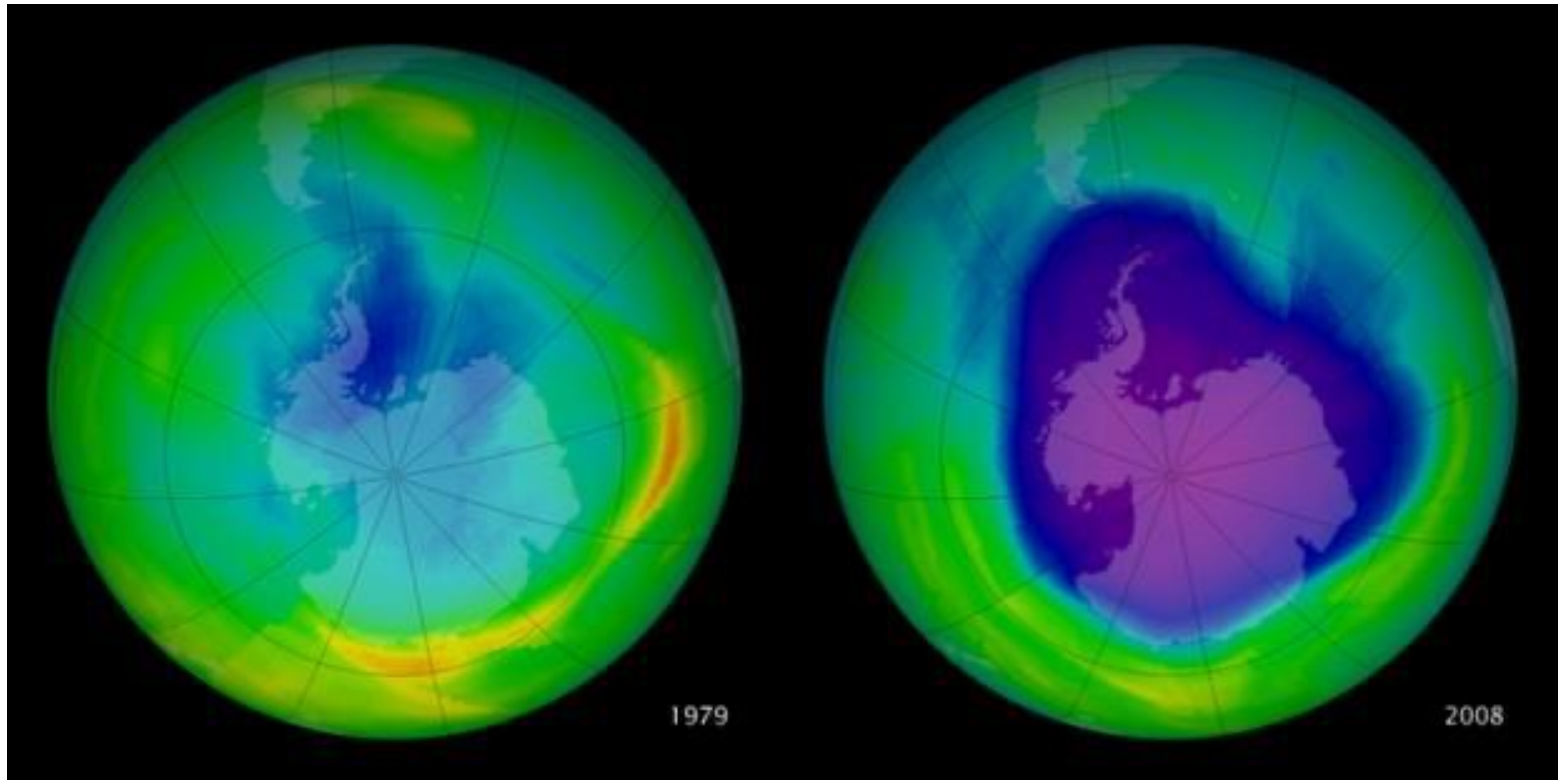
- Combination of the previous three views
- All of the views present problems with the data
- The goal may dictate the solutions:
  - Median value: don't worry too much about crazy outliers
  - Generally, aggregation is less susceptible by numeric errors
  - Be careful, the data may be correct…

[J. Canny et al.]

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



[Wikimedia]

Where does dirty data originate?

- Source data is bad, e.g. person entered it incorrectly
- Transformations corrupt the data, e.g. certain values processed incorrectly due to a software bug
- Integration of different datasets causes problems
- Error propagation: one error is magnified

[J. Canny et al.]

Types of Dirty Data Problems

- Separator Issues: e.g. CSV without respecting double quotes
  - `12, 13, "Doe, John", 45`
- 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`

[J. Canny et al.]

Data Wrangling

- Data wrangling: transform raw data to a more meaningful format that can be better analyzed
- Data cleaning: getting rid of inaccurate data
- Data transformations: changing the data from one representation to another
- Data reshaping: reorganizing the data
- Data merging: combining two datasets

Data Cleaning

Wrangler: Interactive Visual Specification of Data Transformation Scripts

S. Kandel, A. Paepcke, J. Hellerstein, J. Heer

Data Wrangler Demo

- http://vis.stanford.edu/wrangler/app/

Transform Script Import Export

- Split **data repeatedly** on **newline** into **rows**
- Split **split repeatedly** on **','**
- Promote **row 0** to header

Text | Columns | Rows | Table | Clear

Delete **row 7**

Delete **empty rows**

Fill **row 7** by **copying** values from **above**

	Year	Property_crime_rate
0	Reported crime in Alabama	
1		
2	2004	4029.3
3	2005	3900
4	2006	3937
5	2007	3974.9
6	2008	4081.9
7		
8	Reported crime in Alaska	
9		
10	2004	3370.9
11	2005	3615
12	2006	3582

Wrangler

- Data cleaning takes a lot of **time** and **human effort**
- "Tedium is the message"
- Repeating this process on multiple data sets is even worse!
- Solution:
  - interactive interface (mixed-initiative)
  - transformation language with natural language "translations"
  - suggestions + "programming by demonstration"