DSC 201: Data Analysis & Visualization

Data Merging

Dr. David Koop
Data Cleaning
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).

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]
Data Cleaning Operations

- Fill missing data
- Drop missing data
- Remove duplicates
- Modify data (map strings, arithmetic expressions)
- Replace values (e.g. -999 → NaN)
- Clamping values
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  1  0
  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"
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:
  • <numbers>/<numbers>/<numbers>
• Cannot search for every combination!
  • \d+/\d+/\d+
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
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!

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

data.str.contains('gmail')
data.str.split('@').str[1]
data.str[-3:]```
Assignment 4

- Clean and transform hurdat2.txt into hurdat2.csv
Quiz 2

• Next Tuesday
• Same format as Quiz 1: Multiple Choice and Short Answer
• Quiz at the beginning of the class
• Focus on material since the midterm
• Pandas, reading/writing data, data cleaning
Databases

• Databases:
  - Have been around for years
  - Organize data by tables, allow powerful queries
  - Most support concurrency: allowing multiple users to work with the database at once
  - Provide many features to ensure data integrity, security

• Database Management Systems (DBMS): software that manages databases and facilitates adding, updating, and removing data as well as queries over the data

• Main language used to interact with databases: Structured Query Language (SQL)
Relational Databases

- A specific model for databases [Codd, 1969]
- Extremely popular, supported by most major DBMS (IBM DB2, SQLServer, mySQL, etc.)
- Consists of relations (tables) made up of tuples (rows)
- Relations reference each other!
  - Types of relationships: one-to-one, many-to-one, many-to-many
- Each tuple has a key; to reference a tuple in another relation, use a foreign key in the current relation
Example: Football Game Data

- Data about football games, teams, and players
  - Game is between two Teams
  - Each Team has Players

- For each game, we could specify every player and all of their information... why is this bad?
Example: Football Game Data

- Data about football games, teams, and players
  - Game is between two Teams
  - Each Team has Players
- For each game, we could specify every player and all of their information… why is this bad?
- Normalization: reduce redundancy, keep information that doesn't change separate
- 3 Relations: Team, Player, Game
- Each relation only encodes the data specific to what it represents

<table>
<thead>
<tr>
<th>Player</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Name</td>
<td>Height</td>
<td>Weight</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Team</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Name</td>
<td>Wins</td>
<td>Losses</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Game</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Location</td>
<td>Date</td>
</tr>
</tbody>
</table>
Example: Football Game Data

- Have each game store the id of the home team and the id of the away team (one-to-one)
- Have each player store the id of the team he plays on (many-to-one)

What happens if a player plays on 2+ teams?
How does this relate to pandas?

- DataFrames in pandas are ~relations (tables)
- We may wish to normalize data in a similar manner in pandas
- However, operating on 2+ DataFrames at the same time can be unwieldy, can we merge them together?

  - Two potential operations:
    - Have football game data (just the Game table) from 2013, 2014, and 2015 and wish to merge the data into one data frame
    - Have football game data and wish to find the average temperature of the cities where the games were played
Concatenation

• Take two data frames with the same columns and add more rows
• `pd.concat([data-frame-1, data-frame-2, ...])`
• Default is to add rows (axis=0), but can also add columns (axis=1)
• Can also concatenate Series into a data frame.
• `concat` preserves the index so this can be confusing if you have two default indices (0,1,2,3...)—they will appear twice
  - Use `ignore_index=True` to get a 0,1,2...
Merges (aka Joins)

- Need to merge data from one DataFrame with data from another DataFrame
- Example: Football game data merged with temperature data

| Game | | | | |
|---|---|---|---|
| Id | Location | Date | Home | Away |
| 0 | Boston | 9/2 | 1 | 15 |
| 1 | Boston | 9/9 | 1 | 7 |
| 2 | Cleveland | 9/16 | 12 | 1 |
| 3 | San Diego | 9/23 | 21 | 1 |

| Weather | | | |
|---|---|---|
| wld | City | Date | Temp |
| 0 | Boston | 9/2 | 72 |
| 1 | Boston | 9/3 | 68 |
| ... | ... | ... | ... |
| 7 | Boston | 9/9 | 75 |
| ... | ... | ... | ... |
| 21 | Boston | 9/23 | 54 |
| ... | ... | ... | ... |
| 36 | Cleveland | 9/16 | 81 |

No data for San Diego
Merges (aka Joins)

- Want to join the two tables based on the location and date
- Location and date are the **keys** for the join
- What happens when we have missing data?
- Merges are **ordered**: there is a left and a right side
- Four types of joins:
  - Inner: intersection of keys (match on both sides)
  - Outer: union of keys (if there is no match on other side, still include with NaN to indicate missing data)
  - Left: always have rows from left table (no unmatched right data)
  - Right: like left, but with no unmatched left data
Inner Strategy

Merged

<table>
<thead>
<tr>
<th>Id</th>
<th>Location</th>
<th>Date</th>
<th>Home</th>
<th>Away</th>
<th>Temp</th>
<th>wId</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Boston</td>
<td>9/2</td>
<td>1</td>
<td>15</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Boston</td>
<td>9/9</td>
<td>1</td>
<td>7</td>
<td>75</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Cleveland</td>
<td>9/16</td>
<td>12</td>
<td>1</td>
<td>81</td>
<td>36</td>
</tr>
</tbody>
</table>

No San Diego entry
### Merged

<table>
<thead>
<tr>
<th>Id</th>
<th>Location</th>
<th>Date</th>
<th>Home</th>
<th>Away</th>
<th>Temp</th>
<th>wId</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Boston</td>
<td>9/2</td>
<td>1</td>
<td>15</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>NaN</td>
<td>Boston</td>
<td>9/3</td>
<td>NaN</td>
<td>NaN</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Boston</td>
<td>9/9</td>
<td>1</td>
<td>7</td>
<td>75</td>
<td>7</td>
</tr>
<tr>
<td>NaN</td>
<td>Boston</td>
<td>9/10</td>
<td>NaN</td>
<td>NaN</td>
<td>76</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>Cleveland</td>
<td>9/2</td>
<td>NaN</td>
<td>NaN</td>
<td>61</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cleveland</td>
<td>9/16</td>
<td>12</td>
<td>1</td>
<td>81</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>San Diego</td>
<td>9/23</td>
<td>21</td>
<td>1</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
## Merged

<table>
<thead>
<tr>
<th>Id</th>
<th>Location</th>
<th>Date</th>
<th>Home</th>
<th>Away</th>
<th>Temp</th>
<th>wId</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Boston</td>
<td>9/2</td>
<td>1</td>
<td>15</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Boston</td>
<td>9/9</td>
<td>1</td>
<td>7</td>
<td>75</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Cleveland</td>
<td>9/16</td>
<td>12</td>
<td>1</td>
<td>81</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>San Diego</td>
<td>9/23</td>
<td>21</td>
<td>1</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
# Right Strategy

## Merged

<table>
<thead>
<tr>
<th>Id</th>
<th>Location</th>
<th>Date</th>
<th>Home</th>
<th>Away</th>
<th>Temp</th>
<th>wId</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Boston</td>
<td>9/2</td>
<td>1</td>
<td>15</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>NaN</td>
<td>Boston</td>
<td>9/3</td>
<td>NaN</td>
<td>NaN</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>Boston</td>
<td>9/9</td>
<td>1</td>
<td>7</td>
<td>75</td>
<td>7</td>
</tr>
<tr>
<td>NaN</td>
<td>Boston</td>
<td>9/10</td>
<td>NaN</td>
<td>NaN</td>
<td>76</td>
<td>8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>NaN</td>
<td>Cleveland</td>
<td>9/2</td>
<td>NaN</td>
<td>NaN</td>
<td>61</td>
<td>22</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>Cleveland</td>
<td>9/16</td>
<td>12</td>
<td>1</td>
<td>81</td>
<td>36</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

No San Diego entry
Hierarchical Indexing (Multiple Keys)

• We might have multiple keys to identify a single piece of data
• Example: Football records for each team over multiple years
  - Identify a specific row by both the team name and the year
  - Can think about this as a tuple (team_name, year)
• pandas supports this via hierarchical indexing (MultiIndex)
• display mirrors the hierarchical nature of the data
Example

- data = ["W": 11, "L": 5], ["W": 6, "L": 10],
  ["W": 12, "L": 4], ["W": 8, "L": 8],
  ["W": 2, "L": 14]
index = [["Boston", "Boston", "San Diego", "San Diego", "Cleveland"],
df = pd.DataFrame(data, columns=["W", "L"],
                  index=pd.MultiIndex.from_arrays(
                  index, names=('Team', 'Year')))

<table>
<thead>
<tr>
<th>Team</th>
<th>Year</th>
<th>W</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>2007</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2008</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>San Diego</td>
<td>2007</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2008</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Cleveland</td>
<td>2007</td>
<td>2</td>
<td>14</td>
</tr>
</tbody>
</table>
How do we access a row? or slice?
MultiIndex Row Access and Slicing

- `df.loc["Boston", 2007]`
- Remember that `loc` uses the index values, `iloc` uses integers
- **Note:** `df.iloc[0]` gets the first row, **not** `df.iloc[0, 0]`
- Can get a subset of the data using partial indices
  - `df.loc["Boston"]` returns both 2007 and 2008 data
- **What about slicing?**
  - `df.loc["Boston":"Cleveland"]` → ERROR!
  - Need to have sorted data for this to make sense
  - `df = df.sort_index()`
  - `df.loc["Boston":"Cleveland"]` → inclusive!
  - `df.loc[(slice("Boston","Cleveland"),2007),:]`
Reorganizing the MultiIndex

• swaplevel: switch the order of the levels
  - df = df.swaplevel("Year","Team")
  - df.sort_index()

• Can do summary statistics by level
  - df.sum(level="Team")

• Reset the index (back to numbers)
  - df.reset_index()

• Promote columns to be the indices
  - df.set_index(["Team", "Year"])

<table>
<thead>
<tr>
<th>Year</th>
<th>Team</th>
<th>W</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>Boston</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Cleveland</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>San Diego</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>2008</td>
<td>Boston</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>San Diego</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Data Merging in Pandas

- `pd.merge(left, right, ...)`
- Default merge: join on matching column names
- Better: specify the column name(s) to join on via `on` kwarg
  - If column names differ, use `left_on` and `right_on`
  - Multiple keys: use a list
- `how` kwarg specifies the type of join
  ("inner", "outer", "left", "right")
- Can add suffixes to column names when they appear in both tables, but are not being joined on
- Can also merge using the index by setting `left_index` or `right_index` to True
## Merge Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>DataFrame to be merged on the left side.</td>
</tr>
<tr>
<td>right</td>
<td>DataFrame to be merged on the right side.</td>
</tr>
<tr>
<td>how</td>
<td>One of 'inner', 'outer', 'left', or 'right'; defaults to 'inner'.</td>
</tr>
<tr>
<td>on</td>
<td>Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in left and right as the join keys.</td>
</tr>
<tr>
<td>left_on</td>
<td>Columns in left DataFrame to use as join keys.</td>
</tr>
<tr>
<td>right_on</td>
<td>Analogous to left_on for left DataFrame.</td>
</tr>
<tr>
<td>left_index</td>
<td>Use row index in left as its join key (or keys, if a MultiIndex).</td>
</tr>
<tr>
<td>right_index</td>
<td>Analogous to left_index.</td>
</tr>
<tr>
<td>sort</td>
<td>Sort merged data lexicographically by join keys; True by default (disable to get better performance in some cases on large datasets).</td>
</tr>
<tr>
<td>suffixes</td>
<td>Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y') (e.g., if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result).</td>
</tr>
<tr>
<td>copy</td>
<td>If False, avoid copying data into resulting data structure in some exceptional cases; by default always copies.</td>
</tr>
<tr>
<td>indicator</td>
<td>Adds a special column _merge that indicates the source of each row; values will be 'left_only', 'right_only', or 'both' based on the origin of the joined data in each row.</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]