Intermediate Data Science

Data Wrangling: Join, Combine, Reshape

Joanna Bieri DATA201

Important Information

- Email: joanna_bieri@redlands.edu
- Office Hours take place in Duke 209 – Office Hours Schedule
- Class Website
- Syllabus

Data Wrangling

Data Wrangling is the art of managing data that might be spread across many files or databases. It also involved organizing data that comes to you in an inconvenient format. We are going to explore some ways that Pandas can help us in organizing our data!

Hierarchical Indexing

Hierarchical indexing is a feature of pandas that allows you to have more than one index on a single axis in a DataFrame. This is like working with data in a table but allowing it to be higher dimensional. Here is the example from our book:

```
a  1    0.298598
   2    0.303884
   3    0.530289
b  1    0.966484
   3    0.321590
c  1    0.933737
   2    0.079616
d  2    0.744500
   3    0.033183
dtype: float64
```

Hierarchical Indexing

```
data.index
```

```
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
```

Notice here that the index now has two dimensions. You can reach in a grab the stuff inside the 'a' grouping and then within that choose 1,2,3. We can use partial indexing to grab different parts of the data.

Hierarchical Indexing

We can use the `.unstack()` function to send this data into a one dimensional data frame. Unstack takes the inner index and sends it to to separate columns, which keeping the outer index as the DataFrame index.

	1	2	3
a	0.298598	0.303884	0.530289
b	0.966484	NaN	0.321590
c	0.933737	0.079616	NaN
d	NaN	0.744500	0.033183

Hierarchical Indexing

You can also `.stack()` data in a one dimensional data frame.

```
a  1    0.298598
   2    0.303884
   3    0.530289
b  1    0.966484
   3    0.321590
c  1    0.933737
   2    0.079616
d  2    0.744500
   3    0.033183
dtype: float64
```

Hierarchical Indexing

In a DataFrame either the columns or the rows can have hierarchical levels.

		Ohio	Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

Figure 1: image.png

Hierarchical Indexing

The keys are two dimensional

```
MultiIndex([(    'Ohio', 'Green'),
            (    'Ohio',   'Red'),
            ('Colorado', 'Green')],
           )
```

Hierarchical Indexing

The indexes are two dimensional

```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

Hierarchical Indexing

I can choose to name the levels of index and column data

```
frame.index.names = ["key1", "key2"]
frame.columns.names = ["state", "color"]
```

	state		Ohio	Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

Figure 2: image.png

Reordering and Sorting Levels

In the example above the levels were state then color and a/b then number. So it is easy to select by the outer level, and a bit harder to get the inside levels.

```
frame['Ohio']
```

	color	Green	Red
key1	key2		
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

Figure 3: image.png

Reordering and Sorting Levels

So if we wanted to look at just the green data, we would need to reorder the levels, making color outer and state inner.

```
new_frame = frame.swaplevel('state','color', axis=1)
```

	color	Green	Red	Green
	state	Ohio	Ohio	Colorado
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

Figure 4: image.png

Summary Statistics by Level

If we run statistics on the DataFrame as a whole, it ignores the leves and does the operation to the whole thing. This might be okay in some instances, but in the data frame above maybe we want to sum the "a" and "b" groupings separately.

```
# Good old fashioned sum
frame.sum()
```

```
state     color
Ohio      Green    18
          Red      22
Colorado  Green    26
dtype: int64
```

Summary Statistics by Level

```
# Grouped sum
frame.groupby(level='key1').sum()
```

state		Ohio	Colorado
color	Green	Red	Green
key1			
a	3	5	7
b	15	17	19

Figure 5: image.png

Summary Statistics by Level

The grouped sum lets us look at the sums of the specific index levels. We could also group by the columns! If we just transpose the data frame then the column names become the index names!

```
frame.T
```

	key1	a		b	
	key2	1	2	1	2
state	color				
Ohio	Green	0	3	6	9
	Red	1	4	7	10
Colorado	Green	2	5	8	11

Figure 6: image.png

Combining and Merging Datasets

Sometimes in your analysis you will want to grab data from more than one file, or maybe you scrape data from more that one website. In these cases you need to be able to merge the data into a single dataset for analysis. There are a few great Pandas commands for this:

Combining and Merging Datasets

- pd.merge() - connects the rows in separate DataFrames based on one or more keys. It implements the database join operations.
- pd.concat() - this is short of concatenate. Concatenate stacks objects along an access. For example stacking rows to add more observations or stacking columns to add more variables to the existing observations.
- combine_first() - splices together overlapping data to fill in missing values in one object with values from another.

Pandas Merge

Merge connects separate DataFrames based on comparing keys (or column labels). There are different merge types available:

- `inner` is the most restrictive and only includes cases where the keys match across both datasets.
- `left` includes all entries in the left dataset and only those that match from the right dataset.
- `right` includes all entries in the right dataset and only those that match from the left dataset.
- `outer` includes all entries in both datasets.

Pandas Merge

We will start with two example DataFrames and explore the results for the different merge types.

Pandas Merge

	employee_id	name	department_id
0	1	Alice	10
1	2	Bob	20
2	3	Charlie	10
3	4	Diana	30
4	5	Eve	99

	department_id	department_name
0	10	Engineering
1	20	HR
2	30	Marketing
3	40	Sales

Pandas Merge

Looking at these two data sets we see that there are 5 employees and 4 departments. The two DataFrames share the key department_id. **You need a shared key or shared data to merge!** You will also notice some missmatch between the datasets. For example none of our employees have the department_id 40=Sales, and one of our employees has a department_id 99, which does not appear in our departments data frame.

Pandas Merge

Lets look at the merges below. Note: we are using employees and the left and departments as the right dataset. This could be switched. Both datasets have the department_id key!

Pandas Merge

```
pd.merge(employees, departments,
        on='department_id', how='inner')
```

	employee_id	name	department_id	department_name
0	1	Alice	10	Engineering
1	2	Bob	20	HR
2	3	Charlie	10	Engineering
3	4	Diana	30	Marketing

Pandas Merge

```
pd.merge(employees, departments,
        on='department_id', how='left')
```

	employee_id	name	department_id	department_name
0	1	Alice	10	Engineering
1	2	Bob	20	HR
2	3	Charlie	10	Engineering
3	4	Diana	30	Marketing
4	5	Eve	99	NaN

Pandas Merge

```
pd.merge(employees, departments,
        on='department_id', how='right')
```

	employee_id	name	department_id	department_name
0	1.0	Alice	10	Engineering
1	3.0	Charlie	10	Engineering
2	2.0	Bob	20	HR
3	4.0	Diana	30	Marketing
4	NaN	NaN	40	Sales

Pandas Merge

```
pd.merge(employees, departments,
        on='department_id', how='outer')
```

	employee_id	name	department_id	department_name
0	1.0	Alice	10	Engineering
1	3.0	Charlie	10	Engineering
2	2.0	Bob	20	HR
3	4.0	Diana	30	Marketing
4	NaN	NaN	40	Sales
5	5.0	Eve	99	NaN

Pandas Merge

Merge Type	Includes All Employees	Includes All Departments	Notes
Inner	Only matched	Only matched	Most restrictive
Left	Yes	Only matched	Focus on employees
Right	Only matched	Yes	Focus on departments
Outer	Yes	Yes	Full outer view

Pandas Merge

How would you merge data sets if one had the correct data, but did not have the correct key? Well, one option would be to change the column labels to match, but you could also tell pandas.merge() two different keys.

```
pd.merge(employees, departments,
        left_on='department_id', right_on='dept_code')
```

Pandas Merge

You can also specify that you want to use the indexes as the merge values.

```
pd.merge(employees,departments,
        left_on='department_id', right_index=True)
```

Hirearchical index values:

When you have hierarchical index values or columns, things get more confusing my merges are still possible!

Concatenate

When you need to stack new rows or columns onto an existing data set pd.concat() is a great way to do that.

Concatenate

Let's imagine that we are working with the employee and department information above. Now suddenly HR sends us information about two new employees and data that contains all the salaries. They have confirmed that the salaries are in increasing order of the employee id. How do we get all this data into a single dataframe?

Concatenate

When using concat the dimensions must match!

- `axis=0` must have the same number of columns - you are adding rows
- `axis=1` must have the same number of rows - you are adding columns

Concatenate

	employee_id	name	department_id
0	6	Joanna	30
1	7	Bella	20

	emp_num	salary
0	emp_1	60000
1	emp_2	55000
2	emp_3	62000
3	emp_4	58000
4	emp_5	500000
5	emp_6	40000
6	emp_7	40000

Concatenate

1. Concat the new_hires onto the employees data

   all_employees = pd.concat([employees, new_hires],ignore_index=True)

	employee_id	name	department_id
0	1	Alice	10
1	2	Bob	20
2	3	Charlie	10
3	4	Diana	30
4	5	Eve	99
5	6	Joanna	30
6	7	Bella	20

department_id	department_name

Concatenate

2. Merge the employees and department data - keeping all information

   all_employees = pd.merge(all_employees,departments, on='department_id',how='outer')

	employee_id	name	department_id	department_name_x	departm
0	1.0	Alice	10	Engineering	Enginee
1	3.0	Charlie	10	Engineering	Enginee
2	2.0	Bob	20	HR	HR
3	7.0	Bella	20	HR	HR
4	4.0	Diana	30	Marketing	Marketi
5	6.0	Joanna	30	Marketing	Marketi
6	NaN	NaN	40	Sales	Sales
7	5.0	Eve	99	NaN	NaN

Concatenate

3. Concat the new columns onto the full data set.

```
full_data = pd.concat([all_employees, salaries], axis=1)
```

	employee_id	name	department_id	department_name_x	departm
0	1.0	Alice	10	Engineering	Enginee
1	3.0	Charlie	10	Engineering	Enginee
2	2.0	Bob	20	HR	HR
3	7.0	Bella	20	HR	HR
4	4.0	Diana	30	Marketing	Marketi
5	6.0	Joanna	30	Marketing	Marketi
6	NaN	NaN	40	Sales	Sales
7	5.0	Eve	99	NaN	NaN

Combining Data with Overlap

Sometimes you have two datasets that have an overlap, but one or both of them are incomplete and you want to use one to fill in NaNs in the other. You can think of the `comnbine_first()` operation as patching up the data. It basically does and if-else statement that inserts values if there are null values in the original dataset.

Combining Data with Overlap

Here is a scenario where maybe you have incomplete employee data. However each of the datasets has missing data and you want one complete dataset.

Combining Data with Overlap

	name	dept_code	email
1	Alice	10.0	None
2	None	20.0	bob@example.com
3	Charlie	NaN	None

	name	dept_code	email	phone
1	Alice A.	10	alice@example.com	111-1111
2	Bob B.	20	None	222-2222
3	Charlie C.	10	charlie@example.com	333-3333
4	Diana D.	30	diana@example.com	444-4444

Combining Data with Overlap

```
employee_profiles.combine_first(backup_profiles)
```

	dept_code	email	name	phone
1	10.0	alice@example.com	Alice	111-1111
2	20.0	bob@example.com	Bob B.	222-2222
3	10.0	charlie@example.com	Charlie	333-3333
4	30.0	diana@example.com	Diana D.	444-4444

Combining Data with Overlap

combine_first() does not overwrite existing data in the first DataFrame. It aligns on both index and column names — mismatches are handled gracefully. You can think of it as a data patching tool.

Combining Data with Overlap

combine_first() is perfect when:

- You have a primary source of data that may be incomplete.
- You have a secondary or backup source you want to use to fill in the blanks.
- You want row-wise alignment based on index.

But what happens if the indexes dont align?

Reshaping and Pivoting

The goal with reshaping and pivoting is to rearrange tabular data in a way that is better for your specific analysis. We have already seen some of these commands.

- `stack()` rotates or pivots from the columns in the data to the rows when provided hierarchical indexes.
- `unstack()` rotates of pivots from the rows into the columns creating hierarchical indexes.
- `.pivot()` reshapes the data from long(tall) format to a wide format.
- `melt()` reshapes the data from wide to long(tall) melting the column names into the data

Reshaping and Pivoting

Long (Tall) Format Also called "tidy" data in some contexts:

- One row per observation.
- Repeated categories or measurements in a single column.
- More rows, fewer columns.
- One row per person per test

Reshaping and Pivoting

Wide Format

- Unique values in a categorical column become column headers.
- More columns, fewer rows.
- Easier for humans to read, but not always ideal for statistical analysis.
- One row per person, one column per test

Reshaping and Pivoting

Below we will look at the ways we might rearrange or reshape our data!

	person	month	sales	expenses
0	Alice	Jan	200	150
1	Alice	Feb	180	120
2	Bob	Jan	210	160
3	Bob	Feb	190	140

Pivot

When you pivot a DataFrame you tell it which columns to use for the new data set

- `index` which column should be used as the new row (index) labels.
- `column` which column should be used as the new column labels.
- `values` which column should be used to fill in values in the new data set.

Pivot

```
df.pivot(index='person',
        columns='month', values='sales')
```

month person	Feb	Jan
Alice	180	200
Bob	190	210

Pivot

```
df.pivot(index='month',
        columns='person', values='expenses')
```

person month	Alice	Bob
Feb	120	140
Jan	150	160

Pivot

```
df.pivot(index='person',
        columns='month', values=['sales','expenses'])
```

	sales		expenses	
month	Feb	Jan	Feb	Jan
person				
Alice	180	200	120	150
Bob	190	210	140	160

Stack

Stack takes the index values and shifts from wide to tall format. You will see one grouping of data for each index. In many cases you might want to set the index values to be more interesting to get a better breakdown of the data.

Stack

We go from wide data to tall data, meaning one observation per index per column.

```
df.stack()
```

```
0  person      Alice
   month         Jan
   sales         200
   expenses      150
1  person      Alice
   month         Feb
   sales         180
   expenses      120
2  person        Bob
   month         Jan
   sales         210
   expenses      160
```

Stack

Stack with index_set

```
df.set_index(['person', 'month']).stack()
```

```
person  month
Alice   Jan    sales       200
               expenses    150
        Feb    sales       180
               expenses    120
Bob     Jan    sales       210
               expenses    160
        Feb    sales       190
               expenses    140
dtype: int64
```

Unstack

Lets say you are given data with observations for each person. But what you want is a wide data frame, with fewer rows and more categorical columns. This is what unstack can do!

Unstack

Let's start with some stacked data - hierarchical indexes:

```
person  month
Alice   Jan    sales       200
               expenses    150
        Feb    sales       180
               expenses    120
Bob     Jan    sales       210
               expenses    160
        Feb    sales       190
               expenses    140
dtype: int64
```

Unstack

```
data.unstack()
```

		sales	expenses
person	month		
Alice	Feb	180	120
	Jan	200	150
Bob	Feb	190	140
	Jan	210	160

Figure 7: image.png

Unstack

Here use the Person level index as the columns

`unstack(level=0)`

	person	Alice	Bob
month			
Feb	sales	180	190
	expenses	120	140
Jan	sales	200	210
	expenses	150	160

Figure 8: image.png

Unstack

Here use the month level index as the columns

`unstack(level=1)`

person	month	Feb	Jan
Alice	sales	180	200
	expenses	120	150
Bob	sales	190	210
	expenses	140	160

Figure 9: image.png

Unstack

Here use the innermost values as the columns

`unstack(level=2)`

		sales	expenses
person	month		
Alice	Feb	180	120
	Jan	200	150
Bob	Feb	190	140
	Jan	210	160

Figure 10: image.png

Melt

The melt command lets you choose a column (or use all columns) to be used as an additional row in the data. Here is some data:

	person	month	sales	expenses
0	Alice	Jan	200	150
1	Alice	Feb	180	120
2	Bob	Jan	210	160
3	Bob	Feb	190	140

Melt

`pd.melt(df)`

this is a bit drastic in this case!

	variable	value
0	person	Alice
1	person	Alice
2	person	Bob
3	person	Bob
4	month	Jan
5	month	Feb
6	month	Jan
7	month	Feb
8	sales	200
9	sales	180
10	sales	210
11	sales	190

Melt

You keep the columns ‘person’ and ‘month’ the rest are melted

```
pd.melt(df,id_vars=['person','month'])
```

	person	month	variable	value
0	Alice	Jan	sales	200
1	Alice	Feb	sales	180
2	Bob	Jan	sales	210
3	Bob	Feb	sales	190
4	Alice	Jan	expenses	150
5	Alice	Feb	expenses	120
6	Bob	Jan	expenses	160
7	Bob	Feb	expenses	140