Intermediate Data Science

Data Aggregation and Group Operations

Joanna Bieri DATA201

Intermediate Data Science

Important Information

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

Data Aggregation and Groups

Applying functions to separate groups of your data can be a critical component of data analysis. Often we have questions about how subgroups of the data differ or we might want to compute pivot tables for reporting or visualization. We are going to get more in depth into the `groupby()` function and really see if we can understand what it is doing and what object is returned from it. We will also see how to compute pivot tables and cross-tabulations.

Grouping

When grouping we use *split-apply-combine* to describe group operations.

- SPLIT - we first split the data based on one or more keys. Think about categorical values in a single column.
- APPLY - now we apply a function to each of the data subsets that we split above.
- COMBINE - finally the results of these functions are combined into a single summary output or result object.

Grouping

Here is some example data

	key1	key2	data1	data2
0	a	1	1.270613	-0.393596
1	a	2	0.050822	2.970859
2	None	1	1.006155	-0.681311
3	b	2	0.394552	-0.298086
4	b	1	1.959824	0.037971
5	a	<NA>	-1.810534	0.169288
6	None	1	2.196369	-0.734510

Grouping

What if you wanted to calculate the mean of the column data1, but on subsets or groups based on key1.

```
# First group the data
grouped = df["data1"].groupby(df["key1"])

# What is this object
grouped
```

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x7b8a9e3
```

Grouping

Notice that we can't see our data here. This has created a python object that contains all the information we need to apply a function, but it has not actually computed anything yet. We have prepared our split.

Now we will apply a function!

```
grouped.sum()
```

```
key1
a   -0.489098
b    2.354376
Name: data1, dtype: float64
```

We have now applied a function and combined the results into a series.

More Examples

```
# Calculate the mean of the data in column data1
# Group by two keys
# Results in hierarchical index
df["data1"].groupby([df["key1"], df["key2"]]).mean()
```

```
key1  key2
a     1       1.270613
      2       0.050822
b     1       1.959824
      2       0.394552
Name: data1, dtype: float64
```

More Examples

```
# Calculate the means of both column data1 and data2
# Group by both key1 and key2
df.groupby([df["key1"], df["key2"]]).mean()
```

More Examples

```
# This will automatically drop NaNs
df.groupby('key1').size()
```

```
key1
a    3
b    2
dtype: int64
```

```
df.groupby('key1',dropna=False).size()
```

```
key1
a      3
b      2
NaN    2
dtype: int64
```

More Examples

```
# Count the number of nonnull values
df.groupby('key1').count()
```

key1	key2	data1	data2
a	2	3	3
b	2	2	2

Iterating over Groups

You can iterate over the group object returned from groupby().

```
grouped = df["data1"].groupby(df["key1"])

for g in grouped:
    print(g[0])
    display(g[1])
```

Iterating over Groups

```
0    1.270613
1    0.050822
5   -1.810534
Name: data1, dtype: float64

3    0.394552
4    1.959824
Name: data1, dtype: float64
```

Iterating over Groups

You can see that each thing in our grouped object is a tuple. The first entry in the tuple is the group name (or category) the second object is a Series or a DataFrame depending on the number of columns sent in.

```
grouped = df[["data1","data2"]].groupby(df["key1"])
```

	data1	data2
0	1.270613	-0.393596
1	0.050822	2.970859
5	-1.810534	0.169288

	data1	data2
3	0.394552	-0.298086
4	1.959824	0.037971

Column vs Row Grouping

By default groups are created on the axis='index' meaning that it is breaking up the rows into groups based on labels in a column. But we could break up columns based on values in a row.

Column vs Row Grouping

Here is an example where we group the data based on column names.

1. We take the transpose of the dataframe.

Column vs Row Grouping

	0	1	2	3	4	5
key1	a	a	None	b	b	a
key2	1	2	1	2	1	<NA>
data1	1.270613	0.050822	1.006155	0.394552	1.959824	-1.81053
data2	-0.393596	2.970859	-0.681311	-0.298086	0.037971	0.169288

Column vs Row Grouping

2. Here we send in a dictionary that maps the values found in the index to either key or data.
3. The groupby now splits based on our old column names.

```
mapping = {"key1": "key", "key2": "key",
        "data1": "data", "data2": "data"}

grouped = df.T.groupby(mapping)
```

Column vs Row Grouping

	0	1	2	3	4	5
data1	1.270613	0.050822	1.006155	0.394552	1.959824	-1.81053
data2	-0.393596	2.970859	-0.681311	-0.298086	0.037971	0.169288

	0	1	2	3	4	5	6
key1	a	a	None	b	b	a	None
key2	1	2	1	2	1	<NA>	1

Grouping with Functions

You can also use python functions to specify groups.

For example, in the data below we have information about different people. What if you wanted to group a data set based on the length of their name. You can do that!

Grouping with Functions

Example data:

	a	b	c	d	e
Joe	0.999187	0.047659	1.462147	1.305697	0.412685
Steve	1.006778	0.939083	-0.091624	-0.549589	-0.475812
Wanda	0.486682	NaN	NaN	-0.671000	1.025035
Jill	0.440406	0.820044	0.367237	0.440910	0.972593
Trey	0.219433	1.888156	1.463126	-1.385857	1.880668

Grouping with Functions

```
grouped = people.groupby(len)
```

	a	b	c	d	e
Joe	0.999187	0.047659	1.462147	1.305697	0.412685

	a	b	c	d	e
Jill	0.440406	0.820044	0.367237	0.440910	0.972593
Trey	0.219433	1.888156	1.463126	-1.385857	1.880668

	a	b	c	d	e
Steve	1.006778	0.939083	-0.091624	-0.549589	-0.475812
Wanda	0.486682	NaN	NaN	-0.671000	1.025035

Data Aggregation

Data aggregation is when you take an array (or list) of values and apply a transformation that produces a single scalar output. Think about a list of grades and then taking an average.

The lecture notes have a long list of functions!

Now sometimes you want to apply multiple functions to a single grouped object. One way to do this is with the `.agg()` function.

Data Aggregation

Lets read in the tips data we have seen before:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

Data Aggregation

Now lets say that we want to understand the tip percentage from different groups: smokers vs nonsmokers, vs day of the week. Maybe you want to know the average and standard deviation of the tip percentage and you want to define a calculation of your own called max_to_min that calculates the difference between the max and min percent.

```
# Now we can define our own custom function
def max_to_min(arr):
    return arr.max() - arr.min()
```

Data Aggregation

Now group the data by day and smoker and use .agg to apply the functions:

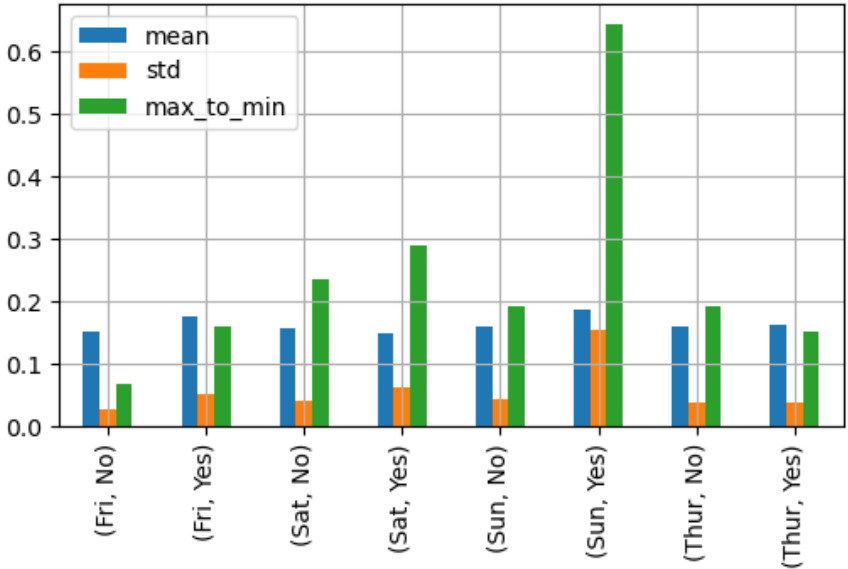
```
grouped = tips.groupby(["day", "smoker"])

functions = ["mean", "std", max_to_min]
agg_data = grouped['tip_pct'].agg(functions)
```

Data Aggregation

This is a case where I would use pandas .plot()

```
agg_data.plot.bar()
plt.grid()
plt.show()
```

This is a case where I would use pandas .plot()



Groupwise Linear Regression

We talked about linear regression in DATA101 and DATA100. It is a way to fit a straight line to some given data.

We will use sklearn to do this. If you don't have it installed, you should install it.

Groupwise Linear Regression

We will read in some data

	AAPL	MSFT	XOM	SPX
2003-01-02	7.40	21.11	29.22	909.03
2003-01-03	7.45	21.14	29.24	908.59
2003-01-06	7.45	21.52	29.96	929.01
2003-01-07	7.43	21.93	28.95	922.93
2003-01-08	7.28	21.31	28.83	909.93

Groupwise Linear Regression

We want to group the data by year, but notice that the index values of this data are Timestamps!

```
print(first_index.year)
print(first_index.month)
print(first_index.month_name())
print(first_index.day)
print(first_index.day_name())
```

```
2003
1
January
2
Thursday
```

Groupwise Linear Regression

We can define a function to get the year

```
def get_year(x):
    return x.year

# Then group by the year
by_year = close_px.groupby(get_year)
```

Groupwise Linear Regression

Now do the linear regression:

```
from sklearn.linear_model import LinearRegression
# Define a liner regression and return intercept and slope
def regress(data,xvars,yvar):
    X = data[xvars]
    y = data[yvar]
    LM = LinearRegression()
    LM.fit(X, y)
    slope = LM.coef_[0]
    intercept = LM.intercept_
    return intercept, slope
```

Groupwise Linear Regression

```
# Apply the linear regression to the groups
by_year.apply(regress,yvar='AAPL',xvars=['SPX'])
```

Groupwise Linear Regression

```
2003     (-8.58984966958583, 0.018505966710274123)
2004     (-132.12116289682774, 0.1325654494610963)
2005     (-299.5951029082234, 0.28683118762773924)
2006    (-133.18691255199056, 0.15566846429728873)
2007     (-432.9175504440354, 0.37990617598041215)
2008      (-36.33814322777991, 0.1461565642803874)
2009     (-164.38487241406185, 0.3282529241664158)
2010    (-210.84981430613925, 0.41290045011855553)
2011      (603.8769113704016, -0.1938338932016932)
dtype: object
```

Pivot Tables and Cross-Tabulation

Pivot Table

Pivot tables are often found in spreadsheet programs and are a way to summarize data. We have seen the `.pivot` operation as a way to wrangle the data. Here we will look at the `pivot_table()` method. The results in many cases can be produced using the groupby function, but this acts as a shortcut and can add partial totals or margins to the data.

Pivot Table

Remember the tips data:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

Pivot Table

This will create a DataFrame with index values taken from the day and smoker rows and the data columns from the values.

```
tips.pivot_table(index=["day", "smoker"],
                 values=["size", "tip", "tip_pct", "total_bill"])
```

Pivot Table

		size	tip	tip_pct	total_bill
day	smoker				
Fri	No	2.250000	2.812500	0.151650	18.420000
	Yes	2.066667	2.714000	0.174783	16.813333
Sat	No	2.555556	3.102889	0.158048	19.661778
	Yes	2.476190	2.875476	0.147906	21.276667
Sun	No	2.929825	3.167895	0.160113	20.506667
	Yes	2.578947	3.516842	0.187250	24.120000
Thur	No	2.488889	2.673778	0.160298	17.113111

Pivot Table

NOTE by default the pivot table returns the mean()

We could have done the same operation with groupby!

Pivot Table

Arguments you might want to pass into the pivot_table() function:

- index – the values that you are grouping by
- values – the numbers you are aggregating
- columns – categories to subset the columns - adding extra columns to the output
- margin=True – include the margin or the value for the whole
- aggfunc – aggregation function if you want something other than mean.
- fill_value – what you want to full if the computation runs into a NaN

More examples in the lecture notes!

Crosstab

Cross tabulation is a type of pivot table that returns frequency observations. You can very quickly reach into your data and get counts of the number of observations that fall into each subset.

Crosstab

Here we will cross tabulate the counts for day of the week, time, and smoker yes/no. This will let us see how the counts line up.

```
cdata = pd.crosstab([tips["time"], tips["day"]], tips["smoker"]
```

Crosstab

	smoker	No	Yes
time	day		
Dinner	Fri	3	9
	Sat	45	42
	Sun	57	19
	Thur	1	0
Lunch	Fri	1	6
	Thur	44	17

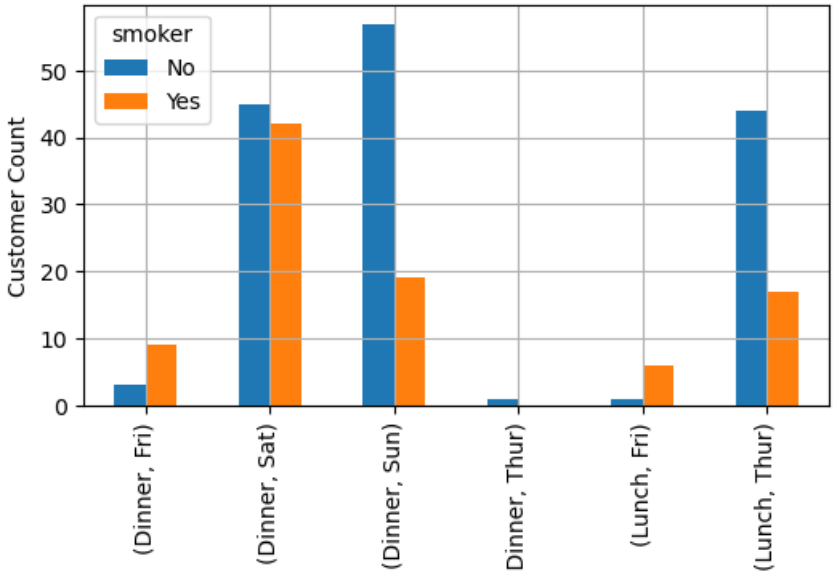
Figure 2: Crosstab

Crosstab

Use pandas to do a quick plot!

```
cdata.plot.bar()
plt.grid()
plt.ylabel('Customer Count')
plt.show()
```

Crosstab


smoker
No
Yes
Customer Count
0
10
20
30
40
50
(Dinner, Fri)
(Dinner, Sat)
(Dinner, Sun)
Dinner, Thur)
(Lunch, Fri)
(Lunch, Thur)