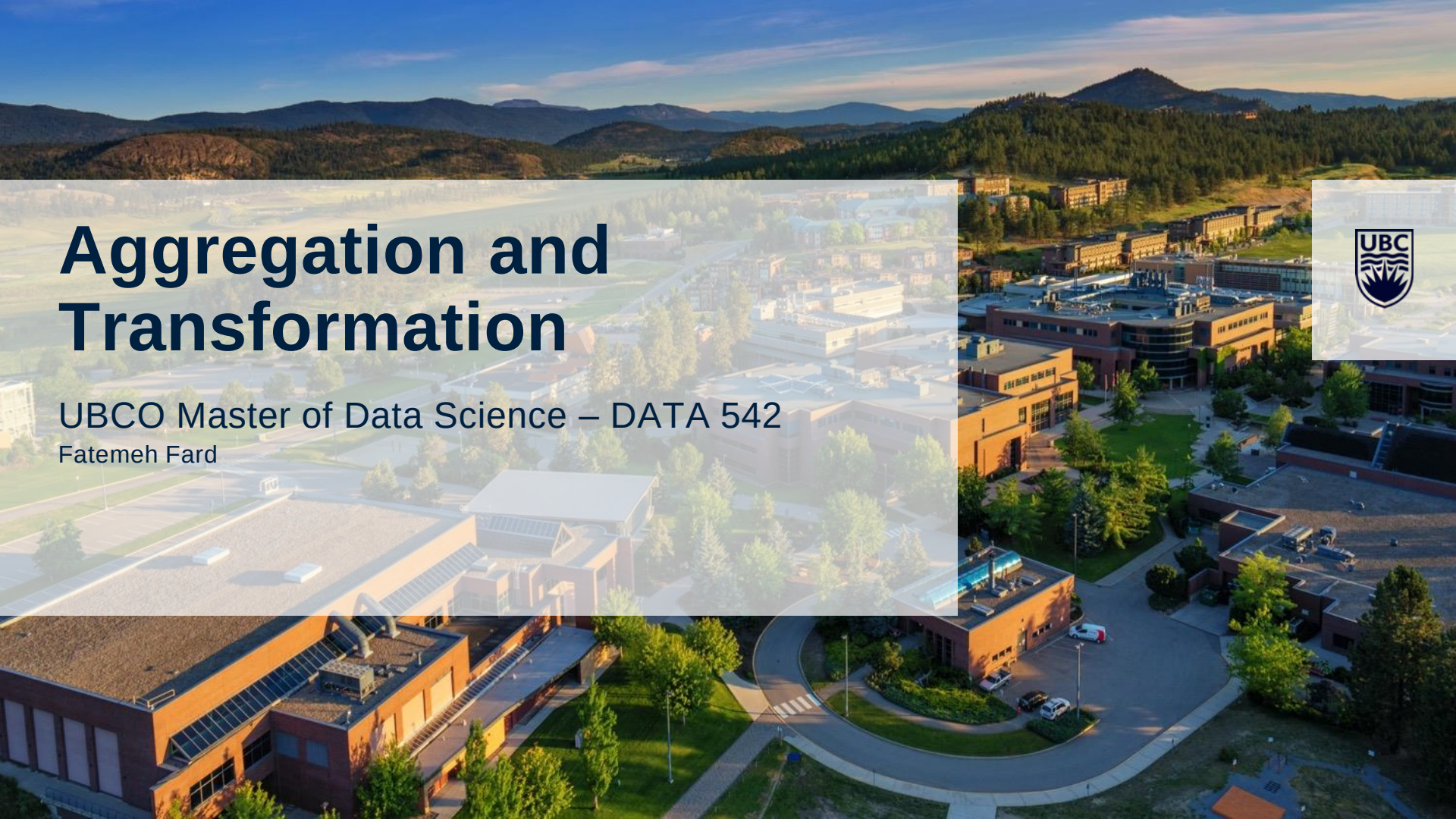

Aggregation and Transformation

UBCO Master of Data Science – DATA 542

Fatemeh Fard

Motivation

SQL:

- Data can be joined, filtered, transformed, and aggregated easily.
- Constrain in the kinds of group operations that can be performed.

With Python and pandas DataFrames, we can perform quite complex group operations by utilizing any function that accepts a pandas object or NumPy array.

Iterating over groups

The **GroupBy object** supports iteration, generating a sequence of 2-tuples containing the group name along with the chunk of data, returning each group as a Series or DataFrame.

```
for  name,  group  in  df.groupby ( 'key' ):
    print (name)
    Print (group)
```

```
for (method, group) in planets.groupby('method'):
     print("{0:30s} shape={1}".format(method,
group.shape))
```

Iterating over groups cont.

In the case of multiple keys, the first element in the tuple will be a tuple of key values:

```
for (k1,k2), group in df.groupby(['key1','key2']):
    print (( k1 ,  k2 ))
    print ( group )
```

TRY IT

Lazy evaluation

- No computation is applied until an aggregation is applied.

  ↳ e.g.
  - sum()
  - size()
  - mean()

Aggregation Review

Select a column and group by keys from another column

```
df.groupby('key').mean()
df.groupby('key')['data1'].sum()
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation

`sum()`, `mean()`, `mad()`, `std()`, `min()`, `max()`

`aggregate()`

It can take

- a string
- a function
- a list

Compute all the aggregates at once

```
df.groupby('key').aggregate(['min', np.median, max])
```

Aggregation cont.

`aggregate()`

Pass a dictionary mapping column names to operations to be applied on that column:

```
df.groupby('key').aggregate({'data1': 'min',
'data2': 'max'})
```

Specifying the split key for groupby()

Passing column name or list of column names

A list, array, series, or index providing the grouping keys

A dictionary or series mapping index to group

Any Python function

A list providing the grouping keys

```
L = [0, 1, 0, 1, 2, 0]
df.groupby(L).sum()
```

L
0
1
0
1
2
0

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

A dictionary mapping index to grouping keys

```
df3 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
df3.groupby(mapping).sum()
```

	data1	data2
key		
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

Python functions and list of keys as grouping keys

Any Python function

```
df3.groupby(str.lower).mean()
```

	data1	data2
a	1.5	4.0
b	2.5	3.5
c	3.5	6.0

A list of valid keys

```
df3.groupby([str.lower, mapping]).mean()
```

TRY IT

Filtering

A filtering operation allows you to drop data based on the group properties using **`filter()`**.

```
def filter_func(x):
    return x['data2'].std() > 4

df.groupby('key').filter(filter_func)
```

Transform

While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. Use `transform()`.

The output is the same shape as the input.

lambda

Example: center the data by subtracting the group-wise mean

```
df.groupby('key').transform(lambda x: x - x.mean())
```

Map

Map: It iterates over each element of a series.

```
df['column1'].map(lambda x: 10+x)
```

This will add 10 to each element of column1.

```
df['column2'].map(lambda x: 'AV'+x)
```

This will concatenate “AV“ at the beginning of each element of column2 (column format is string).

Apply

Apply: As the name suggests, applies a function along any axis of the DataFrame. It can be applied on a DataFrame or a Series.

```
df[['column1','column2']].apply(sum)
```

It will returns the sum of all the values of column1 and column2.

Apply

The `apply()` method lets you apply an **arbitrary function** to the **group results**.

The function should **take a DataFrame**, and **return either a Pandas object (e.g., DataFrame, Series) or a scalar**; the combine operation will be tailored to the type of output returned.

```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby('key').apply(norm_by_data2)
```

ApplyMap

ApplyMap: This helps to apply a function to each element of dataframe.

```
func = lambda x: x+2
df.applymap(func)
```

It will add 2 to each element of dataframe (all columns of dataframe must be numeric type)

Differences between Map, Apply and ApplyMap

DEFINITION

`map` is defined on Series ONLY

`applymap` is defined on DataFrames ONLY

`apply` is defined on BOTH

INPUT ARGUMENT

`map` accepts dicts, Series, or callable

`applymap` and `apply` accept callables only

Note: A **callable** object is an object that can accept some arguments (also called parameters) and possibly return an object (often a tuple containing multiple objects). A function is the simplest **callable** object in **Python**, but there are others, such as classes or certain class instances.

REF: https://stackoverflow.com/questions/19798153/difference-between-map-applymap-and-apply-methods-in-pandas

Differences between Map, Apply and ApplyMap

BEHAVIOR

`map` is elementwise for Series

`applymap` is elementwise for DataFrames

`apply` also works elementwise but is suited to more complex operations and aggregation. The behaviour and return value depends on the function.

REF: https://stackoverflow.com/questions/19798153/difference-between-map-applymap-and-apply-methods-in-pandas

Differences between Map, Apply and ApplyMap

USE CASE

`map` is meant for mapping values from one domain to another, so is optimised for performance

```
df['A'].map({1:'a', 2:'b', 3:'c'})
```

`applymap` is good for elementwise transformations across multiple rows/columns

```
df[['A', 'B', 'C']].applymap(str.strip)
```

`apply` is for applying any function that cannot be vectorised

```
df['sentences'].apply(nltk.sent_tokenize)
```

REF: https://stackoverflow.com/questions/19798153/difference-between-map-applymap-and-apply-methods-in-pandas

Differences between Map, Apply and ApplyMap

`DataFrame.apply` operates on entire rows or columns at a time.

`DataFrame.applymap`, `Series.apply`, and `Series.map` operate on one element at time.

https://stackoverflow.com/questions/19798153/difference-between-map-applymap-and-apply-methods-in-pandas

https://www.geeksforgeeks.org/difference-between-map-applymap-and-apply-methods-in-pandas/

Differences between Transform and Apply

`apply` implicitly passes all the columns for each group as a **DataFrame** to the custom function, while `transform` passes each column for each group as a **Series** to the custom function

The custom function passed to `apply` can return a scalar, or a Series or DataFrame (or numpy array or even list). The custom function passed to `transform` must return a sequence (a one dimensional Series, array or list) the same length as the group.

REF: https://stackoverflow.com/questions/27517425/apply-vs-transform-on-a-group-object

Vectorization

Vectorization is the process of executing operations on entire arrays.

Looping over DataFrame rows using indices (slow)

- Looping with iterrows()
- Looping with apply()
- Vectorization with Pandas series
- Vectorization with NumPy arrays

1000x faster data manipulation: vectorizing with Pandas and Numpy, https://www.youtube.com/watch?v=nxWginnBklU&ab_channel=PyGotham2019

https://engineering.upside.com/a-beginners-guide-to-optimizing-pandas-code-for-speed-c09ef2c6a4d6

Lecture 5 learning outcome

At the end of this lecture you should be able to:

- Transform your data with your desired functions
- Perform apply() method to your DataFrame to apply your desired functions on groups or the whole data
- Perform functions and aggregations on groups in your dataframe

TRY IT
Filling Missing Values with Group-Specific Values

Fill null values with appropriate mean from the group

Dataset:

```
states = ['Ohio',’New York','Vermont','Florida','Oregon','Nevada','California','Idaho’]

data = pd.Series(np.random.randn( 8 ), index = states )
```

Write function to fill the null values with the mean value of each group

```
data.groupby(group_key).mean()
fill_mean  =  lambda  g :  g.fillna ( g.mean ())
data.groupby(group_key).apply(fill_mean)
```


THE UNIVERSITY OF BRITISH COLUMBIA