Fourth Industrial Summer School

Day 4

Data Transformation

Session Objectives

- ✓ Data Transformation
  - Mapping
  - Discretization
  - Binning
  - Permutation

- ✓ Data Grouping
  - Data Grouping
  - Data Aggregation

Data Transformation

- A function that maps the entire set of values of a given attribute to a new set of replacement values
  - s.t. each old value can be identified with one of the new values

- When dealing with DataFrames,
  - Data transformation refers to a reassembly of the data contained within a DataFrame, with possible additions by other DataFrame and removal of unwanted parts.

- Methods:
  - Mapping
  - Rename the Indexes of the Axes
  - Discretization
  - Binning
  - Permutation

Mapping

- The mapping is the creation of a list of matches between two different values, with the ability to bind a value to a particular label or string.
- Pandas library provides a set of functions which exploit mapping to perform some operations
- To define a mapping there is no better object than dict objects.

```
map = {
    'label1' : 'value1,
    'label2' : 'value2,
    ...
}
```

Replacing Values via Mapping

- Often in the data structure that you have assembled there are values that do not meet your needs.
  - For example, the text may be in a foreign language, or
  - may be a synonym of another value, or
  - may not be expressed in the desired shape.

- In such cases, a replace operation of various values is often a necessary process.

```
frame = pd.DataFrame(
    { 'item':['ball','mug','pen','pencil','ashtray'],
     'color':['white','rosso','verde','black','yellow'],
     'price':[5.56,4.20,1.30,0.56,2.75]})
```

Replacing Values via Mapping..

- Thus to be able to replace the incorrect values in new values is necessary to define a mapping of correspondences,
  - Key : old values
  - Value : new ones

```
newcolors = {
    'rosso': 'red',
    'verde': 'green'
 }
```

- Now the only thing you can do is to use the replace() function with the mapping as an argument

```
frame.replace(newcolors)
```

```
frame['color'].replace(newcolors)
```

Replacing Values via Mapping..

- A common case, for example, is the replacement of the NaN values with another value, for example 0.
  - Al performs its jol

```
import numpy as  np
ser = pd.Series([1,3,np.nan,4,6,np.nan,3])
```

```
ser.replace(np.nan,0)
```

```
0    1.0
1    3.0
2    0.0
3    4.0
4    6.0
5    0.0
6    3.0
dtype: float64
```

Adding Values via Mapping

- The mapping can be used also to add values in a column depending on the values contained in another.
  - Remember: the mapping will always be defined

```python
frame = pd.DataFrame(
    {'item':['ball','mug','pen','pencil','ashtray'],
     'color':['white','red','green','black','yellow']})
```

- For example, to add a column to indicate the price of the item shown in the DataFrame.

Adding Values via Mapping

- First, define a dict object that contains a list of prices for each

```
price = {
    'ball' : 5.56,
    'mug' : 4.20,
    'bottle' : 1.30,
    'scissors' : 3.41,
    'pen' : 1.30,
    'pencil' : 0.56,
    'ashtray' : 2.75
}
```

- Then, apply map() function to a Series or to a column of a DataFrame accepts a function or an object containing a dict with mapping.

```
frame['price'] = frame['item'].map(price)
frame
```

Rename the Indexes of the Axes

- To replace the label indexes, pandas provides the rename() function, which takes the mapping as argument, that is, a dict object.

	item	color	price
0	ball	white	5.56
1	mug	red	4.20
2	pen	green	1.30
3	pencil	black	0.56
4	ashtray	yellow	2.75

	item	color	price
first	ball	white	5.56
second	mug	red	4.20
third	pen	green	1.30
fourth	pencil	black	0.56
fifth	ashtray	yellow	2.75

```
reindex = {
    0: 'first',
    1: 'second',
    2: 'third',
    3: 'fourth',
    4: 'fifth'}
```

```
frame.rename(reindex)
```

Rename the Indexes of the Axes

- If you want to rename columns you must use the **columns** option.
- Thus, assign various mapping explicitly to the two **index** and **columns** options.

```
recolumn = {
    'item':'object',
    'price': 'value'}
```

```
frame.rename(index=reindex, columns=recolumn)
```

	object	color	value
first	ball	white	5.56
second	mug	red	4.20
third	pen	green	1.30
fourth	pencil	black	0.56
fifth	ashtray	yellow	2.75

Rename the Indexes of the Axes

- In Case, a single value need to be replaced, it can further explicate the arguments passed to the function of avoiding having to write and assign many variables.

```
frame.rename(index={1:'first'},
             columns={'item':'object'})
```

	object	color	price
0	ball	white	5.56
first	mug	red	4.20
2	pen	green	1.30
3	pencil	black	0.56
4	ashtray	yellow	2.75

Rename the Indexes of the Axes

- rename() function returns a DataFrame with the changes, leaving unchanged the original DataFrame.

  - If you want the changes to take effect on the object on which you call the function, you will set the **inplace**

```
frame.rename(columns={'item':'object'},
             inplace=True)
```

Discretization

- Discretization process can happen in some experimental cases, to handle large quantities of data generated in sequence.

- To carry out an analysis of the data it is necessary to transform this data into discrete categories,
  - for example, by dividing the range of values in smaller intervals and counting the occurrence or statistics related to each of them.

- Discretization: Divide the range of a continuous attribute into intervals
  - Interval labels can then be used to replace actual data values
  - Reduce data size by discretization

Data Discretization Methods

- Typical methods:
  - Binning
    - Top-down split, unsupervised
  - Histogram analysis
    - Top-down split, unsupervised
  - Clustering analysis
    - Unsupervised
  - Decision-tree analysis
    - Supervised

Simple Discretization: Binning

- Equal-width (distance) partitioning
  - Divides the range into N intervals of equal size: uniform grid
  - if A and B are the lowest and highest values of the attribute, the width of intervals will be: W = (B –A)/N.
  - The most straightforward, but outliers may dominate presentation
  - Skewed data is not handled well

- Equal-depth (frequency) partitioning
  - Divides the range into N intervals, each containing approximately same number of samples
  - Good data scaling
  - Managing categorical attributes can be tricky

Discretization

- for example,
  - you may have a reading of an experimental value between 0 and 100.
  - These data are collected in a list.

```
results = [12,34,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]
```

  - You know that the experimental values have a range from 0 to 100; therefore you can uniformly divide this interval,
  - For example, into four equal parts, i.e., bins.
    - The first contains the values between 0 and 25,
    - the second between 26 and 50,
    - the third between 51 and 75, and
    - the last between 76 and 100.

Equal-width partitioning

```
results = [12,34,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]
```

```
bins = [0,25,50,75,100]
```

```
categ = pd.cut(results, bins)
categ
```

each class has the lower limit with a bracket and the upper limit with a parenthesis.

```
[(0, 25], (25, 50], (50, 75], (50, 75], (25, 50], ..., (75, 100], (0, 25], (25,
Length: 17
Categories (4, interval[int64]): [(0, 25] < (25, 50] < (50, 75] < (75, 100]]
```

- Use pandas.cut when you need to segment and sort data values into bins.
  - This function is also useful for going from a continuous variable to a categorical variable.

Equal-width partitioning

- The object returned by the **cut()** function is a special object of **Categorical** type.
- Internally it contains:
  - a **levels** array indicating the names of the different internal categories and
  - a **labels** array that contains a list of numbers equal to the elements of **results** (i.e., the array subjected to

```
bin_names = ['unlikely','less likely','likely','highly likely']
categ =pd.cut(results, bins, labels= bin_names)
categ
```

```
[unlikely, less likely, likely, likely, less likely, ..., highly likely, unlikel
Length: 17
Categories (4, object): [unlikely < less likely < likely < highly likely]
```

Equal-width partitioning

- Finally to know the occurrences for each bin, that is, how many results fall into each category, you have to use the value_counts() function.

```
pd.value_counts(categ)
```

```
highly likely    5
likely           4
less likely      4
unlikely         4
dtype: int64
```

Equal-width partitioning

- If the **cut()** function is passed as an argument to an integer instead of explicating the bin edges,
  - it divides the range of values of the array in many intervals as specified by the number.

- The limits of the interval will be taken by the minimum and maximum of the sample data

Equal-width partitioning

- Example

```
results = [12,34,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]
```

```
categ1 = pd.cut(results, 5)
categ1
```

```
[(2.904, 22.2], (22.2, 41.4], (60.6, 79.8], (41.4, 60.6], (22.2, 41.4], ..., (79.8, 99.0], (22.2, 41.
Length: 17
Categories (5, interval[float64]): [(2.904, 22.2] < (22.2, 41.4] < (41.4, 60.6] < (60.6, 79.8] <
                                    (79.8, 99.0]]
```

Equal-depth partitioning

- **qcut()** function divides the sample directly into quintiles.

- The qcut() will ensure that the number of occurrences for each bin is equal, but the edges of each bin to vary.

Equal-depth partitioning

- Example

```
categ3 = pd.qcut(results, 5)
categ3
```

```
[(2.999, 24.0], (24.0, 46.0], (62.6, 87.0], (46.0, 62.6], (24.0, 46.0], ..., (62.6, 87.0], (2.999,
Length: 17
Categories (5, interval[float64]): [(2.999, 24.0] < (24.0, 46.0] < (46.0, 62.6] < (62.6, 87.0] <
                                    (87.0, 99.0]]
```

```
pd.value_counts(categ3)
```

```
(62.6, 87.0]     4
(2.999, 24.0]    4
(87.0, 99.0]     3
(46.0, 62.6]     3
(24.0, 46.0]     3
dtype: int64
```

Permutation

- The operations of permutation (random reordering) of a Series or the rows of a DataFrame are easy to do using the numpy.random.permutation() function.
  - create a DataFrame containing integers in ascending order.
  - Now create an array of five integers from 0 to 4

```
nframe = pd.DataFrame(np.arange(25).reshape(5,5))
new_order = np.random.permutation(5)
nframe.take(new_order)
```

Hands on session

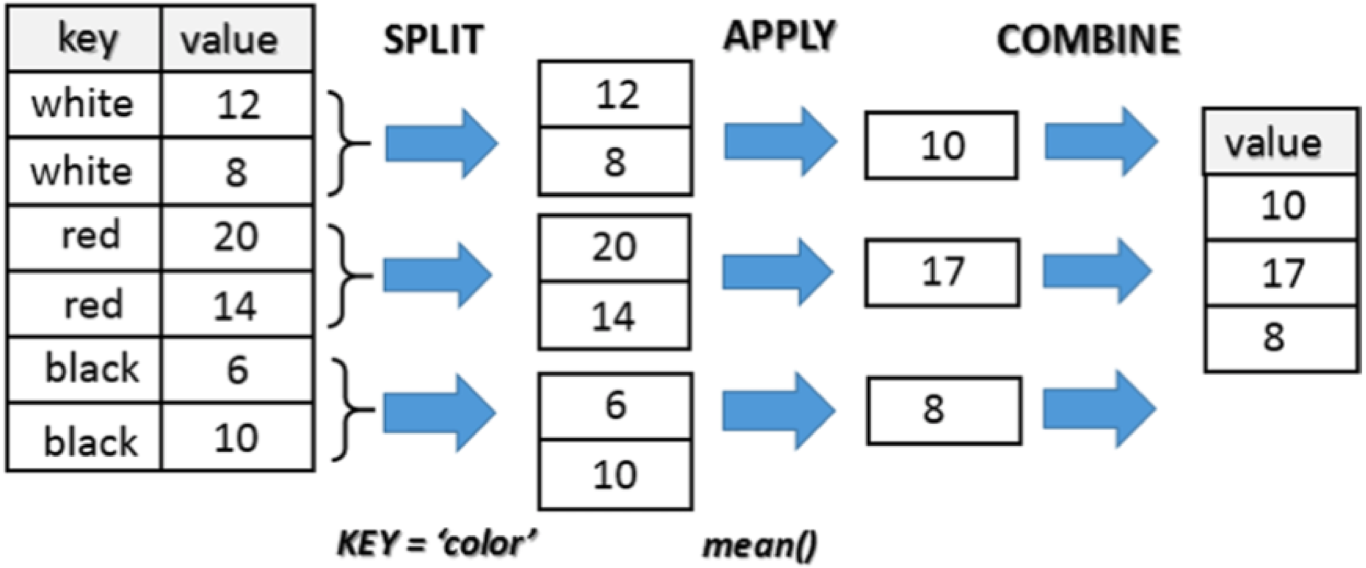
Problem Solving

Data Grouping

- In real data science projects, you'll be dealing with large amounts of data and trying things over and over, so for efficiency, we use Groupby concept.
- It's a simple concept but it's an extremely valuable technique that's widely used in data science.
- It is a process of transformation since after the division into different groups, you can apply a function that converts or transforms the data in some way depending on the group they belong to.

Pandas GroupBy

- Groupby mainly refers to a process involving one or more of the following steps they are:



GroupBy

- Split - a process in which we split data into group by applying some conditions on the dataset
  - often linked to indexes or just certain values in a column.

- Apply- a process in which we apply a function or calculate statistics to each group independently
  - which will produce a new and single value, specific to that group.

- Combine- a process in which we combine different datasets after applying groupby and results into a data structure

- Pandas provides a tool very flexible and high performance: **GroupBy**

DataFrames groupby method

- Split Data into Groups

Employees.csv

```
dataset.groupby('Team')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object
```

```
dataset.groupby('Team').groups
```

.groups is used to view the groups

```
{'Business Development': Int64Index([   9,
            ...
            928, 933, 936, 949, 950, 959,
           dtype='int64', length=101),
 'Client Services': Int64Index([   4,   18,
            ...
            918, 920, 924, 932, 937, 938,
           dtype='int64', length=106),
 'Distribution': Int64Index([ 40,   60,   65,
            240, 248, 260, 266, 267, 278,
```

Refers to the row index

DataFrames groupby method

- Using multiple multiple columns to do the grouping

```
#Group by with multiple columns
dataset.groupby(['Team','Position']).groups
```

DataFrames groupby method

- We can also use the groupby method get_group to filter the grouped data.
  - For example, to select the "Engineering" group

```
grouped = dataset.groupby('Team')
grouped.get_group("Engineering")
```

DataFrames groupby method

- **Pandas Groupby Count**
- To find out how big each group is (e.g., how many observations in each group),
  - **.size()** to count the number of rows in each group:

- In addition Pandas groupby count() method can be used to count by group(s) and get the entire dataframe.

```
grouped = dataset.groupby(['Team','Position'])
grouped.count()
```

Note: If we don't have any missing values the number should be the same for each column and group. ☾ Thus, this is a way we can explore the dataset and see if there are any missing values in any column.

DataFrames groupby method

- In some cases we may want to find out the number of unique values in each group.
  - This can be done using the groupby method **nunique**

```
grouped.nunique()
```

DataFrames groupby method

- Once groupby object is create we can calculate various statistics for each group:

```
dataset.groupby('Team')[['Salary']].sum()
```

Note: If single brackets are used to specify the column (e.g. salary), then the output is Pandas Series object. When double brackets are used the output is a Data Frame

DataFrames groupby method

- In some cases, you may need to change the column names to reflect the meaning of the groups calculation

```
grouped['Salary'].mean().reset_index().rename(
     columns={'Team':'Sector',
               'Salary' : 'Total Salary'})
```

```
grouped['Salary'].median().reset_index().rename(
     columns={'Team':'Sector',
               'Salary' : 'MedianSalary'})
```

Aggregation

- Most of the time we want to have our summary statistics in the same table. We can calculate the mean and median salary, by groups, using the *agg* method.
- Thus, Aggregation assists to get a summary about the operations applied to the groups.

Aggregation

- An aggregated function returns a single aggregated value for each group.
- Once the group by object is created, several aggregation operations can be performed on the grouped data

```python
import numpy as np
grouped = dataset.groupby('Team')

print(grouped['Salary'].agg(np.mean))
```

```
Team
Business Development    91866.316832
Client Services         88224.424528
Distribution            88500.466667
Engineering             94269.195652
Finance                 92219.480392
Human Resources         90944.527473
```

Aggregation

- Another way to see the size of each group is by applying the size() function

```
grouped = dataset.groupby('Team')
print(grouped.agg(np.size))
```

```
                      Name  Gender
Team
Business Development   101     101
Client Services        106     106
Distribution            90      90
Engineering             92      92
Finance                102     102
Human Resources         91      91
Legal                   88      88
Marketing               98      98
```

Aggregation

- Applying Multiple Aggregation Functions at Once

```
import numpy as np
grouped = dataset.groupby('Team')
print (grouped['Salary'].agg([np.sum, np.mean, np.std]))
```

```
                          sum          mean           std
Team
Business Development  9278498  91866.316832  33461.860802
Client Services       9351789  88224.424528  31272.598888
Distribution          7965042  88500.466667  33538.473345
Engineering           8672766  94269.195652  32349.531179
Finance               9406387  92219.480392  34475.515066
Human Resources       8275952  90944.527473  33107.945736
Legal                 7858718  89303.613636  32755.649720
```

Aggregation

- Another example

```
grouped['Salary'].agg(['mean',
                       'median',
                       'std',
                       'min',
                       'max']).reset_index()
```

	Team	mean	median	std	min	max
0	Business Development	91866.316832	93997.0	33461.860802	36844	147417
1	Client Services	88224.424528	90356.0	31272.598888	35095	147183
2	Distribution	88500.466667	86842.0	33538.473345	35575	149105
3	Engineering	94269.195652	95273.0	32349.531179	36946	147362

Hands on session

Problem Solving