Fourth Industrial Summer School

Day 3

Data Transformation, Grouping & Preprocessing

What topics will be covered in M2?

Day 1	Python for Data Science	▪ NumPy arrays ▪ Pandas data structures
Day 2	Data Loading	▪ Working with files ▪ Reading/writing data from/to various sources
	Data Manipulation	▪ Indexing and selecting, sampling, sorting and ranking, merging and joining reshaping and pivoting
Day 3	Data Transformation & Grouping	▪ Mapping, encoding, discretization, binning, ▪ Grouping and aggregation
	Data Preprocessing	▪ Data cleaning, removing duplicates, handling missing data and outliers ▪ Normalization/Scaling

Session Objectives

- ✓ Data Transformation
  - Mapping
  - Encoding
  - Discretization
  - Binning
- ✓ Data Grouping
  - GroupBy: Split, Apply, Combine
  - Data Aggregation

Data Transformation

- Data is usually collected from different sources
  - Data in its raw form is often unorganized and it requires processing to be valuable for analysis
  - **Issues**: Missing Data, useless variables, incorrect data, etc.

- Thus, data processing is needed to make data coherent and analyzable
  - Convert categorical data to numerical
  - Ensure data types are appropriate for analysis
  - Cleanse data by removing nulls or duplicate records
  - Summarize data through aggregation methods
  - Rescale values, often to the 0-1 range, for algorithms that require it

Data Transformation...

- All ML algorithms are based on mathematics, thus
  - The data should be in a format that is consistent with ML methods
  - The entire set of values of a given attribute should be in the same format and sense

  - Format Consistency:
    - **Numerical Data**: e.g. not a mix of numbers and text.
    - **Date/Time Data**: e.g., YYYY-MM-DD
    - **Categorical Data**: e.g., 'Male' and 'Female' instead of 'M', 'F', 'male', 'female'

  - Sense Consistency:
    - **Units of Measurement**: e.g., all in meters rather than a mix of meters and feet

Data Transformation is the process of changing the data format, structure, or values into a structured format suitable for analysis

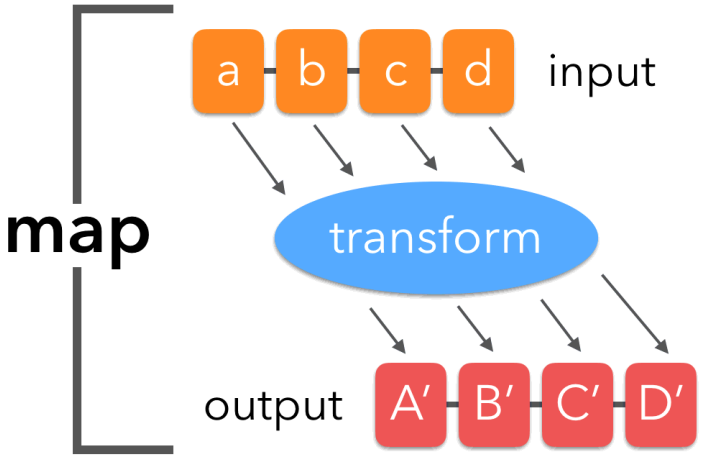
Data Transformation in Pandas

- Mapping
  - Replacing values
  - Adding values
- Encoding categorical variables
- Discretization
  - Equal-width binning
  - Equal-depth binning

Mapping

- The mapping is the creation of a list of matches between two different values, with the ability to bind a value to a specified label or string

- Pandas library provides a set of functions which employ **mapping** to perform some operations



Replacing Values via Mapping…

- Often in the data structure that you have assembled, some values do not meet your needs

- For example, the given value may be
  - in a foreign language, or
  - a synonym of another value, or
  - not be expressed in the desired shape.

- In such cases, a replacement operation of various values is often necessary
  - The dictionary object is the best way to define a mapping

Replacing Values via Mapping…

Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team	Position
Douglas	Male	8/6/93	12:42 PM	97308	6.945	True	Marketing	PG
Thomas	Male	3/31/96	6:53 AM	61933	4.170	True	NaN	SF
Maria	Female	4/23/93	11:17 AM	130590	11.858	False	Finance	SG
Jerry	Male	3/4/05	1:00 PM	138705	9.340	True	Finance	SG
Larry	Male	1/24/98	4:47 PM	101004	1.389	True	Client Services	PF
...	...	...	...	...	...	...	...	...
Henry	NaN	11/23/14	6:09 AM	132483	16.655	False	Distribution	PF
Phillip	Male	1/31/84	6:30 AM	42392	19.675	False	Finance	C
Russell	Male	5/20/13	12:39 PM	96914	1.421	False	Product	SG
Larry	Male	4/20/13	4:45 PM	60500	11.985	False	Business Development	SG
Albert	Male	5/15/12	6:24 PM	129949	10.169	True	Sales	SG

1000 rows × 8 columns

The value “Distribution” will be replaced by → ‘Logistics’

The values “Marketing” and “Sales” will be replaced by → ‘Commercial’

Replacing Values via Mapping…

1. Investigate the values in the target column
   COULMN.value_counts()

```
employees["Team"].value_counts()
```

```
Client Services          106
Finance                  102
Business Development     101
Marketing                 98
Product                   95
Sales                     94
Engineering               92
Human Resources           91
Distribution              90
Legal                     88
Name: Team, dtype: int64
```

2. Define a mapping of correspondences

```
new_teams = {
    'Marketing' : 'Commercial',
    'Sales' : 'Commercial',
    'Distribution' : 'Logistics'
            }
```

3. Preform the mapping

```
employees['Team'].replace(new_teams, inplace=True)
```

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 separately

Adding Values via Mapping..

- For example, to add a new column "Transportation allowance", where the column has three classes

1. define the required dictionary

```
# define the values of the new column
Trans_allowance = {
    "Commercial" : 'A',
    "Product": 'B',
    "Client Services" :'C'
}
```

Assuming that the allowance are given to these three classes

2. Then, apply the map() function on the relevant column (or Series)

```
employees["Transportation Allowance"] = employees["Team"].map(Trans_allowance)
```

Mapping for encoding the data

- Encoding data is useful for converting categorical data into a format that ML algorithms can use effectively.

```
# make the dictionary manually
gender_map = {'Male':1,
              'Famale':2
             }

employees['Gender'] = employees['Gender'].map(gender_map)
```

- The mapping can be created based on the existing values

```
# Get the values using the code
uniqueValues=employees['Gender'].unique().tolist()
print(uniqueValues)
gender_map = dict(zip(uniqueValues,list(range(1,len(uniqueValues)))))
employees['Gender'] = employees['Gender'].map(gender_map)
```

Mapping for encoding the data

- The mapping can be created based on the existing values

```
# Get the values using the code

uniqueValues=employees['Gender'].unique().tolist()
print(uniqueValues)

gender_map = dict(zip(uniqueValues,list(range(0,len(uniqueValues)))))

print(gender_map)

employees['Gender'] = employees['Gender'].map(gender_map)
```

```
['Male', 'Female', nan]
{'Male': 0, 'Female': 1, nan: 2}
```

LabelEncoder and One-Hot Encoding

- LabelEncoder is used to transform categorical data into numerical labels, where each unique category is assigned an integer

Original Data

Team	Points
A	25
A	12
B	15
B	14
B	19
B	23
C	25
C	29

Label Encoded Data

Team	Points
0	25
0	12
1	15
1	14
1	19
1	23
2	25
2	29

- One-Hot Encoding transforms categorical features into binary vectors, where each category becomes a new column with a binary value (1 or 0)

One-Hot Encoded Data

Team_A	Team_B	Team_C	Points
1	0	0	25
1	0	0	12
0	1	0	15
0	1	0	14
0	1	0	19
0	1	0	23
0	0	1	25
0	0	1	29

LabelEncoder

```
from sklearn.preprocessing import LabelEncoder

# Initialize LabelEncoder
le = LabelEncoder()

# Fit and transform
employees['encoded_Position'] = le.fit_transform(employees['Position'])
```

Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team	Position
Douglas	1.0	8/6/93	12:42 PM	97308	6.945	True	Marketing	PG
Thomas	1.0	3/31/96	6:53 AM	61933	4.170	True	NaN	SF
Maria	2.0	4/23/93	11:17 AM	130590	11.858	False	Finance	SG
Jerry	1.0	3/4/05	1:00 PM	138705	9.340	True	Finance	SG
Larry	1.0	1/24/98	4:47 PM	101004	1.389	True	Client Services	PF
...	...	...	...	...	...	...	...	...
Henry	NaN	11/23/14	6:09 AM	132483	16.655	False	Distribution	PF

Binary Encoding

```
import category_encoders as ce

be = ce.BinaryEncoder(cols=['Gender'])
employees = be.fit_transform(employees)
```

Name	Gender_0	Gender_1	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
Douglas	0	1	8/6/93	12:42 PM	97308	6.945	True	Marketing
Thomas	0	1	3/31/96	6:53 AM	61933	4.170	True	NaN
Maria	1	0	4/23/93	11:17 AM	130590	11.858	False	Finance
Jerry	0	1	3/4/05	1:00 PM	138705	9.340	True	Finance
Larry	0	1	1/24/98	4:47 PM	101004	1.389	True	Client Services
...	...	...	...	...	...	...	...	...
Henry	1	1	11/23/14	6:09 AM	132483	16.655	False	Distribution

One-Hot Encoding

```
employees = pd.get_dummies(employees, columns=['Gender'],
                           drop_first=True)
employees.head()
```

Name	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team	Position	Gender_Male
Douglas	8/6/93	12:42 PM	97308	6.945	True	Marketing	PG	1
Thomas	3/31/96	6:53 AM	61933	4.170	True	NaN	SF	1
Maria	4/23/93	11:17 AM	130590	11.858	False	Finance	SG	0
Jerry	3/4/05	1:00 PM	138705	9.340	True	Finance	SG	1
Larry	1/24/98	4:47 PM	101004	1.389	True	Client Services	PF	1

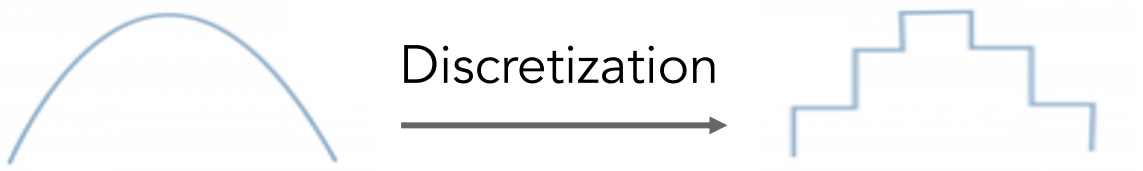
drop_first=False

Gender_Female	Gender_Male
0	1
0	1
1	0
0	1
0	1

Discretization

- Discretization is the process of converting continuous data into discrete values by partitioning the continuous range into intervals and assigning a unique value or label to each interval
  - It is a data-reduction mechanism

- To analyze the vast amounts of data generated in sequence, it might be required to transform this data into discrete categories



Continuous attributes, features or variables

Discretized or nominal attributes/features/variables/intervals.

Discretization…

- For example, to
  - divide the range of a continuous attribute into smaller intervals
  - count the occurrence or statistics related to each of them
  - interval labels can then be used to replace actual data values
  - reduce data size
  - prepare for further analysis, e.g., classification

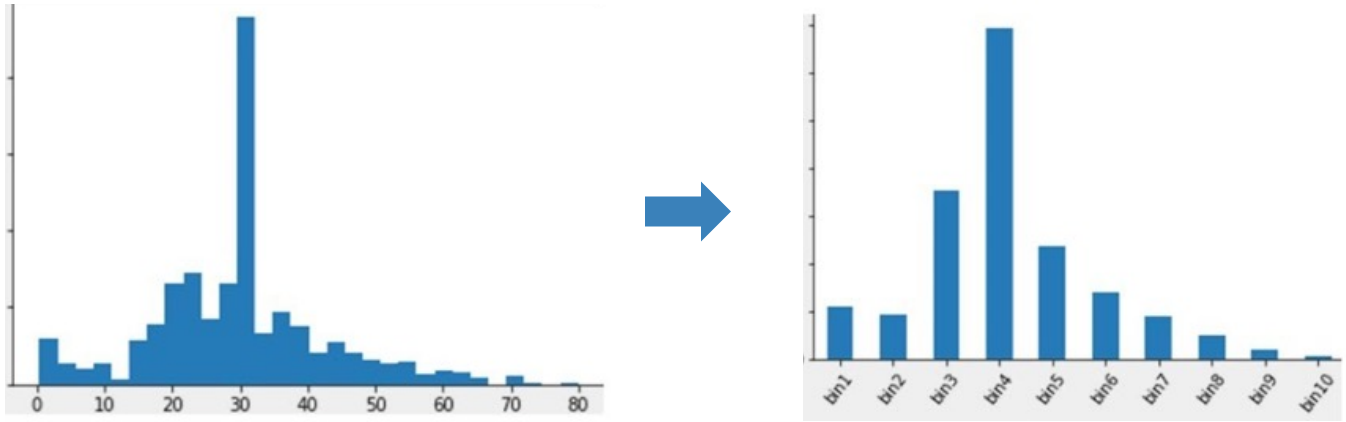
- Some learning methods show remarkable improvements in learning speed and accuracy
  - e.g., decision tree-based algorithms produce shorter, more compact, and accurate results when using discrete values

Data Discretization Methods

- Binning
  - Equal-width binning
  - Equal-depth binning
- Histogram analysis
- Clustering analysis
- Decision-tree analysis
- ....

Binning

- The binning method
  - sorts the data first and then
  - the sorted values are distributed into several **buckets** or **bins**
- There are three approaches to performing smoothing
  - by bin means- each value in a bin is replaced by the mean value of the bin
  - by bin median- … replaced by its bin median value.
  - by bin boundary- the minimum and maximum values in a given bin are identified as the bin boundaries.
    - Each bin value is then replaced by the closest boundary value.



Equal-width binning

- Equal-width (distance) partitioning
- The simplest binning approach is to partition the range of the variable into **k** equal-width intervals.

- Divides the range into N intervals of equal size: uniform grid
- The width of intervals W = (B –A)/N,
  - where A and B are the lowest and highest values

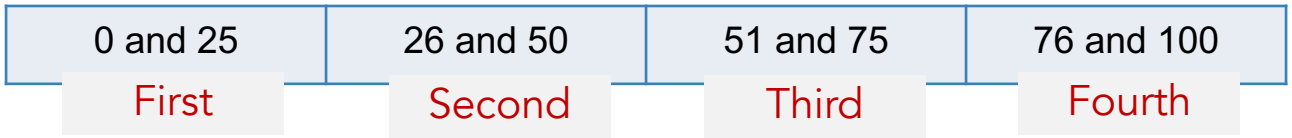
- Weakness
  - Outliers may dominate presentation
  - Skewed data is not handled well

Equal-width binning …

- When dealing with continuous numeric data, it is often helpful to bin the data into multiple buckets for further analysis.
  - For example, a list of experimental values between 0 and 100

```
values = [12,25,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]
```

  - To uniformly divide this interval,



- Pandas supports these approaches using the cut and qcut functions
  - useful to perform some statistical analysis on a large number of scalar data
  - to covert from a continuous variable to a categorical variable.

Equal-width binning …

- Pandas cut() function is used to segregate array elements into separate bins
  - returns a special object of categorical type
  - works only on 1-D array-like objects

```
values = [12,25,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]

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

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

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

Each class has a lower limit with a bracket and an upper limit with a parenthesis.

Equal-width binning …

- Bins labels can be passed also

```
bin_lables= ['Very Low','Low','High','Very High']
categ =pd.cut(values, bins, labels= bin_lables)
```

```
[Very Low, Very Low, High, High, Low, ..., Very High, Very Low, Low, Very High, Very High]
Length: 17
Categories (4, object): [Very Low < Low < High < Very High]
```

- value_counts() can be used to get the occurrences for each bin,
  - how many results fall into each category

```
categ.value_counts()
```

```
Very Low     5
Low          3
High         4
Very High    5
```

Equal-width binning …

- You can specify the number of categories, instead of explicating the bin edges,
  - Pass the bins argument as an integer
  - Accordingly, the cut() function 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

```
values = [12,25,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]
```

```
categ = pd.cut(values, 5)
```

```
[(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, 41.4], (41.4, 60.
6], (79.8, 99.0], (79.8, 99.0]]
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 binning

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

Equal-depth binning…

- qcut() is described as a "Quantile-based discretization function"

- It means that qcut tries to divide up the underlying data into equal sized bins.
  - The function defines the bins using percentiles based on the distribution of the data, not the actual numeric edges of the bins.

- qcut() function divides the sample directly into quintiles.
  - It ensure that the number of occurrences for each bin is equal, but the edges of each bin to vary.

Equal-depth binning …

- Example

```
values = [12,25,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]
```

```
categ_d = pd.qcut(values, 5)
```

```
[(2.999, 23.4], (23.4, 46.0], (62.6, 87.0], (46.0, 62.6], (23.4, 46.0], ..., (62.6, 87.0], (2.999, 23.4], (46.0, 62
.6], (87.0, 99.0], (62.6, 87.0]]
Length: 17
Categories (5, interval[float64]): [(2.999, 23.4] < (23.4, 46.0] < (46.0, 62.6] < (62.6, 87.0] < (87.0, 99.0]]
```

```
categ_d.unique()
```

```
[(2.999, 23.4], (23.4, 46.0], (62.6, 87.0], (46.0, 62.6], (87.0, 99.0]]
```

```
pd.value_counts(categ_d)
```

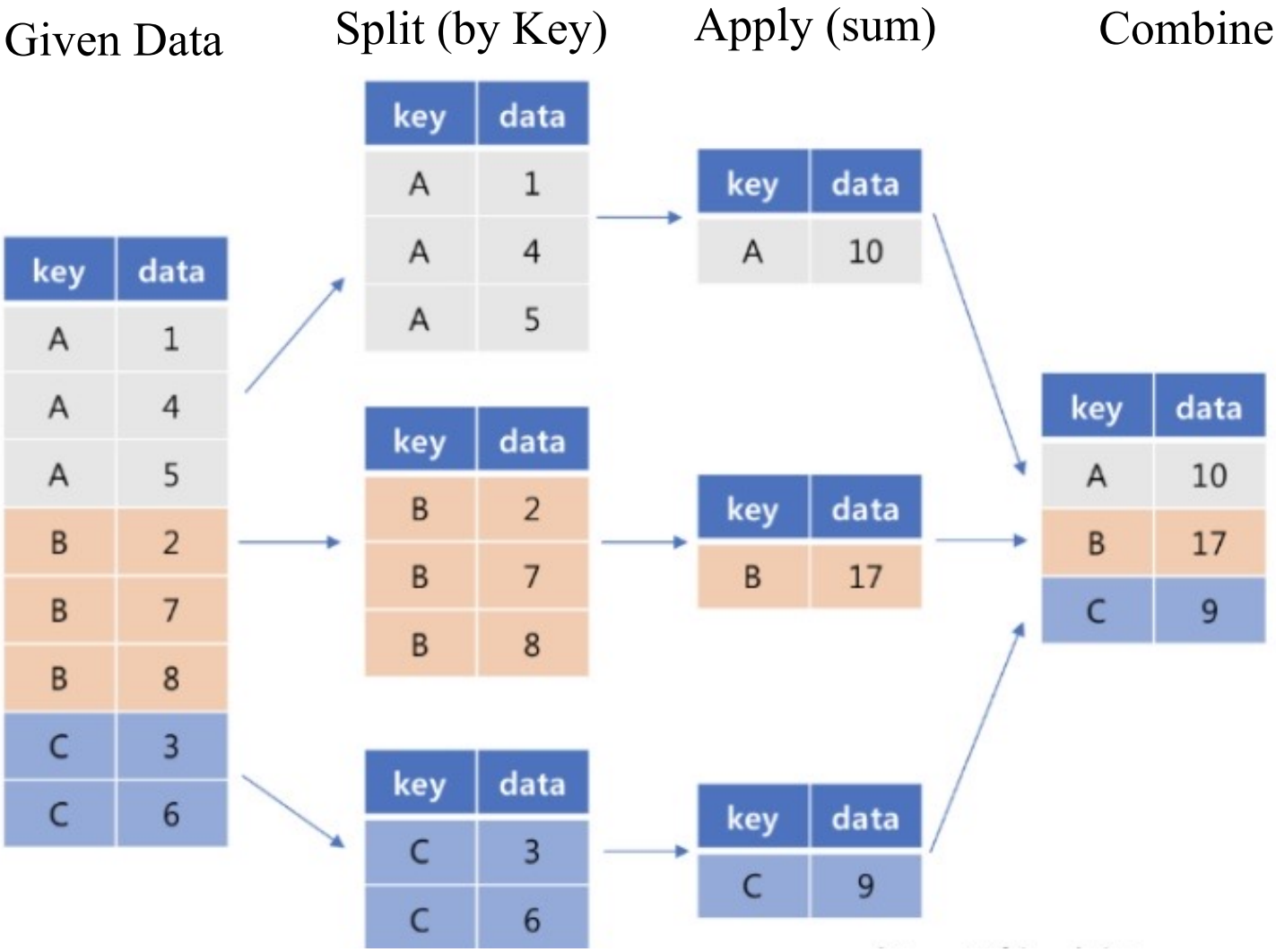
```
(62.6, 87.0]     4
(2.999, 23.4]    4
(87.0, 99.0]     3
(46.0, 62.6]     3
(23.4, 46.0]     3
```

Data Grouping & Aggregation

Data Grouping

- In real data science projects, you'll be dealing with large amounts of data
- Groupby concept is used to get insights into the data
  - Although its simplicity, it is a valuable technique widely used in Data Science.
- After dividing the data into groups, you can apply a function that converts or transforms the data in some way depending on the group they belong to.
  - E.g.
    - How many employees are in each department?
    - What is the average of the salaries in each department?
    - etc.

Groupby concept



Groupby concept..

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

- Split data into group by applying some conditions on the dataset
  - often linked to indexes or just certain values in a column
- Apply a function or calculate statistics to each group independently
  - which will produce a new and single value, specific to that group
- Combine different datasets after applying groupby and results into a data structure

Groupby concept..

- In the **apply** step, we might wish to do one of the following:
  - **Aggregation**: compute summary statistic(s) for each group. Examples:
    - Compute group sums or means.
    - Compute group sizes/counts.
  - **Transformation**: perform some group-specific computations and return a like-indexed object. Examples:
    - Standardize data (Z-score) within a group.
    - Filling NAs within groups with a value derived from each group.
  - **Filtration**: discard some groups, according to a group-wise computation that evaluates True or False. Examples:
    - Discard data that belongs to groups with only a few members.
    - Filter out data based on the group sum or mean.

DataFrames groupby method

- Split Data into Groups using one column

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

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

employees.csv

  - Multiple columns can be used to do the grouping

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

- To view the created groups, use .groups property

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

```
{'Business Development': [9, 33, 36,
4, 151, 155, 164, 180, 182, 203, 249,
1, 414, 419, 441, 473, 477, 478, 481,
```

refers to the row index

Selecting a group

- get_group() can be used to filter the grouped data
  - For example, to select the "Engineering" group

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

	Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team	Position
8	Angela	Female	11/22/05	6:29 AM	95570	18.523	True	Engineering	PG
30	Christina	Female	8/6/02	1:19 PM	118780	9.096	True	Engineering	SG
50	Nancy	Female	9/23/00	8:05 AM	94976	13.830	True	Engineering	SF
54	Sara	Female	8/15/07	9:23 AM	83677	8.999	False	Engineering	PG
58	Theresa	Female	4/11/10	7:18 AM	72670	1.481	True	Engineering	SG
...	...	...	...	...	...	...	...	...	...

Count values

- groupby.count() is counting values within each group
  - The missing values are excluded

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

Team	Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Position
Business Development	99	90	101	101	101	101	99	101
Client Services	100	90	106	106	106	106	100	106
Distribution	77	72	90	90	90	90	77	90
Engineering	86	84	92	92	92	92	86	92

  - Notes:
    - It is one way to explore the dataset for any missing value
    - If we don't have any missing values, the number should be the same for each column and group.

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

To group records using more than one column

Apply functions on a column

- It is possible to apply a function on a specific column
- Examples:
  - Find statistics for each group

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

  - Get the number of positions in each team

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

  - Find the total male-only salaries for each team

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

  - Find the number of male and female entries in each team

```
dataset.groupby(['Team', 'Gender'])['Name'].count()
```

Rename the resulting columns

- To change the column names to reflect the meaning of the groups' calculation

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

```
Team
Business Development    91866.316832
Client Services         88224.424528
Distribution            88500.466667
Engineering             94269.195652
Finance                 92219.480392
Human Resources         90944.527473
Legal                   89303.613636
Marketing               90435.591837
Product                 88665.505263
Sales                   92173.436170
Name: Salary, dtype: float64
```

```
dataset.groupby(['Team'])['Salary'].mean().reset_index().rename(
    columns={'Salary' : 'Total Salary'}).round(2)
```

	Team	Total Salary
0	Business Development	91866.32
1	Client Services	88224.42
2	Distribution	88500.47
3	Engineering	94269.20
4	Finance	92219.48
5	Human Resources	90944.53
6	Legal	89303.61
7	Marketing	90435.59
8	Product	88665.51
9	Sales	92173.44

To round off to decimal places
round(x)

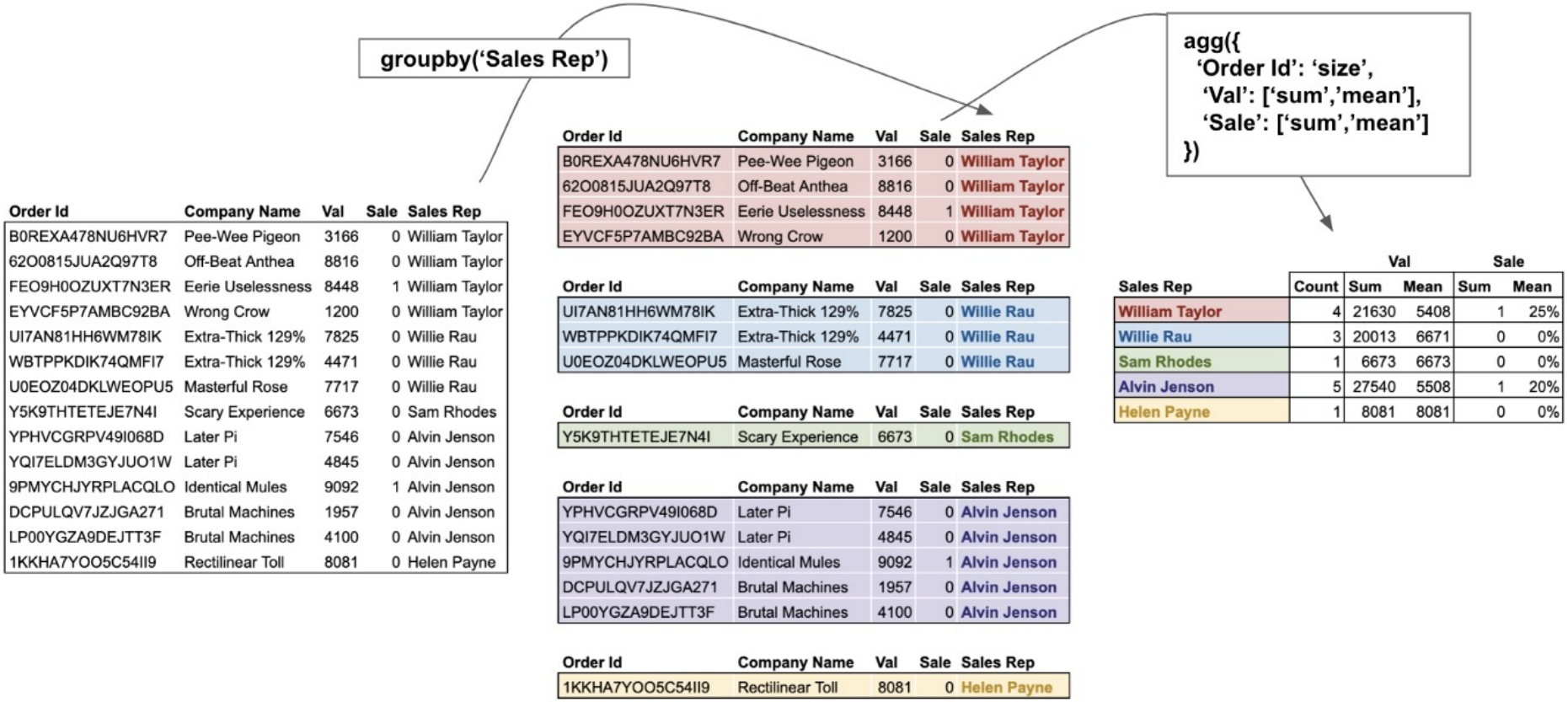
Aggregation

- Once the GroupBy object has been created, several methods are available to perform a **computation** on the grouped data

- Effective summarization is an essential part of big data analysis
  - Methods such as sum(), mean(), median(), min(), and max(), give single numbers that provide insight into the nature of a potentially large data set.

- Aggregation assists to get a summary of the operations applied to the groups
  - E.g., calculate the mean and median salary, by groups, using agg method

Aggregation: Example



Order Id	Company Name	Val	Sale	Sales Rep
B0REXA478NU6HVR7	Pee-Wee Pigeon	3166	0	William Taylor
62O0815JUA2Q97T8	Off-Beat Anthea	8816	0	William Taylor
FEO9H0OZUXT7N3ER	Eerie Uselessness	8448	1	William Taylor
EYVCF5P7AMBC92BA	Wrong Crow	1200	0	William Taylor
UI7AN81HH6WM78IK	Extra-Thick 129%	7825	0	Willie Rau
WBTPPKDIK74QMFI7	Extra-Thick 129%	4471	0	Willie Rau
U0EOZ04DKLWEOPU5	Masterful Rose	7717	0	Willie Rau
Y5K9THTETEJE7N4I	Scary Experience	6673	0	Sam Rhodes
YPHVCGRPV49I068D	Later Pi	7546	0	Alvin Jenson
YQI7ELDM3GYJUO1W	Later Pi	4845	0	Alvin Jenson
9PMYCHJYRPLACQLO	Identical Mules	9092	1	Alvin Jenson
DCPULQV7JZJGA271	Brutal Machines	1957	0	Alvin Jenson
LP00YGZA9DEJTT3F	Brutal Machines	4100	0	Alvin Jenson
1KKHA7YOO5C54II9	Rectilinear Toll	8081	0	Helen Payne

Sales Rep	Val			Sale	
	Count	Sum	Mean	Sum	Mean
William Taylor	4	21630	5408	1	25%
Willie Rau	3	20013	6671	0	0%
Sam Rhodes	1	6673	6673	0	0%
Alvin Jenson	5	27540	5508	1	20%
Helen Payne	1	8081	8081	0	0%

Aggregation…

- An aggregated function returns a single aggregated value for each group

```
import numpy as np
dataset.groupby('Team')['Salary'].agg(np.mean)
```

  - For simplicity, you may assign the resultant DataFrame to a new one

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

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

Multiple Aggregation

- Applying multiple aggregation functions at once

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

grouped['Salary'].agg([np.sum, np.mean, np.std])
```

Team	sum	mean	std
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
Marketing	8862688	90435.591837	34175.230632
Product	8423223	88665.505263	31997.406562

Multiple Aggregation..

- Another example

```
grouped = dataset.groupby(['Team'])
grouped['Salary'].agg(['mean',
                       'median',
                       'std',
                       'min',
                       'max'])
```

Team	mean	median	std	min	max
Business Development	91866.316832	93997.0	33461.860802	36844	147417
Client Services	88224.424528	90356.0	31272.598888	35095	147183
Distribution	88500.466667	86842.0	33538.473345	35575	149105
Engineering	94269.195652	95273.0	32349.531179	36946	147362
Finance	92219.480392	88873.5	34475.515066	35381	149908
Human Resources	90944.527473	93022.0	33107.945736	35203	149903
Legal	89303.613636	87994.5	32755.649720	35061	148985
Marketing	90435.591837	85381.5	34175.230632	36643	149456

```
grouped = dataset.groupby(['Team','Position'])
grouped['Salary'].agg(['mean',
                       'median',
                       'std',
                       'min',
                       'max']).round(2)
```

Multiple Aggregation..

- It is also allowed to pass a dictionary to agg() and specify what operations to apply to each column.
  - Instructions for aggregation are provided in the form of a python dictionary or list.

```
dataset.groupby(['Team', 'Gender']).agg(
    {
        'Position': "count",
        'Salary':'mean',
        'Bonus %': 'max'
    }
)
```

Keys→ Columns
Values→ Functions

		Position	Salary	Bonus %
Team	Gender			
Business Development	Female	50	92669.060000	19.626
	Male	40	89071.750000	18.845
Client Services	Female	48	86430.083333	19.731
	Male	42	93141.833333	19.894

Multiple Aggregation..

- To pass a list of functions to be applied on the target column

```
grouped = dataset.groupby(['Gender'])

grouped.agg({'Salary': ['sum', 'mean'],
             'Bonus %': ['sum', 'mean'],
             'Position': ['count']}).reset_index().round(2)
```

	Gender	Salary		Bonus %		Position
		sum	mean	sum	mean	count
0	Female	38800311	90023.92	4306.13	9.99	431
1	Male	38660604	91180.67	4392.57	10.36	424

Multiple Aggregation..

- For simplicity, define the aggregation procedure and pass it to the function

```
# Define the aggregation procedure outside of the groupby operation

aggregations = {
    'Salary': 'mean',
    'Bonus %':'max'
}

dataset.groupby(['Team']).agg(aggregations)
```

Multiple Aggregation…

- The agg(..) syntax is flexible and simple to use
  - You can also define functions inline using lambda functions to extract statistics

```
dataset.groupby(['Team']).agg(
    {
    'Salary': 'mean',
    'Bonus %':lambda x: max(x)
})
```

Team	Salary	Bonus %
Business Development	91866.316832	19.626
Client Services	88224.424528	19.894
Distribution	88500.466667	19.908
Engineering	94269.195652	19.850
Finance	92219.480392	19.930

Multiple Aggregation…

- Usually, lambda functions are used used to provide statistics that are not provided by the built-in options

You can also assign labels to the new columns

```
dataset.groupby('Team').agg(
    Ave_Bonus = ('Bonus %', 'mean'),
    # Apply a lambda to Salary column
    Salary_gap = ('Salary', lambda x: x.max()-x.min()),

).reset_index().round(2)
```

	Team	Ave_Bonus	Salary_gap
0	Business Development	10.57	110573
1	Client Services	10.50	112088
2	Distribution	9.62	113530
3	Engineering	10.46	110416
4	Finance	10.19	114527
5	Human Resources	9.99	114700
6	Legal	10.32	113924

Hands-On Exercises

Data Preprocessing

Session Objectives

- ✓ Data Preprocessing
  - Handling Missing Data
  - Eliminating Duplicate data
  - Normalization/Scaling
  - Handling Outliers

Why Data Preprocessing?

Data Quality Improvement	Cleaning/removing any errors or inconsistencies in the data, such as missing values, outliers, or duplicate entries
Data Integration	In real-world scenarios, data is often collected from multiple sources in different formats
Noise Reduction	Data may contain noise, which refers to irrelevant or misleading information
Normalization and Scaling	Ensuring that all variables are on a similar scale prevents certain features from dominating the analysis due to their larger magnitude
Feature Selection and Extraction	Identification and selection of relevant features for analysis, where less important or redundant features are eliminated, resulting in simplified and more efficient models

Data issues !!

1. Missing data-
   - Information is not collected (e.g., people decline to give some info…)
   - Attributes may not be applicable to all cases (e.g., annual income is not applicable to children)

2. Duplicate data –
   - A major issue when merging data from heterogeneous sources

3. Different scaling of numeric columns in the dataset
   - Change the values of numeric columns in the dataset to a common scale
   - Without distorting differences in the ranges of values

4. Outliers are data objects with characteristics that are considerably different than most of the other data objects in the dataset
   - The noisy data may be resulting from the modification of the original values
   - Many reasons may cause the noise

1. Missing Data

- Missing data occurs commonly in many data analysis applications
- Data is not always available
  - E.g.. some tuples have no reordered values for some attributes
- Missing data may be due to
  - Equipment malfunction
  - Inconsistent with other recorded data and thus deleted
  - Data not entered due to misunderstanding
  - Certain data may not be considered important at the time of entry
  - Not register history or changes of the data
- Filling-in Missing Data
  - Ignore the instance (e.g. a class label is missing)
  - Use a global constant to fill in the missing value
  - using the feature mean or other analysis of the variable

Handling Missing Data in Pandas

- One of the goals of pandas is to make working with missing data as painless as possible.
  - For example, all the descriptive statistics on Pandas objects exclude missing data by default.

- In Pandas, missing data is represented by two values: None or NaN
  - Pandas treat None and NaN as essentially interchangeable for indicating missing or null values.

Handling Missing Data in Pandas

- Pandas provides a number of methods to deal with missing values in the DataFrame

df.method()	Description
isnull()	Returns True for NaN values.
notnull()	Returns False for NaN values.
dropna()	Drop missing observations
fillna(0)	Replace missing values with zeros
replace()	Replace a value by new one
interpolate()	Fill the missing values using different methods

Checking for missing values

```
# creating a DataFrame
dic = {
   'Name': ['Tom', 'Bob', 'Sam', 'Maria', 'Eve', 'Frank', 'Grace'],
   'Exam1':[100, np.nan, 90, np.nan, 95,  np.nan, 85],
    'Exam2': [30, 45, 56, np.nan, 75, 87, 90],
    'Exam3':[ 40, np.nan, 80, 98, np.nan, 90, np.nan]}
scores = pd.DataFrame(dic)
scores.set_index("Name", inplace=True)
```

- isnull() function returns a DataFrame of Boolean values which are True for NaN values

Name	Exam1	Exam2	Exam3
Tom	100.0	30.0	40.0
Bob	NaN	45.0	NaN
Sam	90.0	56.0	80.0
Maria	NaN	NaN	98.0
Eve	95.0	75.0	NaN
Frank	NaN	87.0	90.0
Grace	85.0	90.0	NaN

```
# using isnull() function
scores.isnull()
```

Name	Exam1	Exam2	Exam3
Tom	False	False	False
Bob	True	False	True
Sam	False	False	False
Maria	True	True	False
Eve	False	False	True
Frank	True	False	False
Grace	False	False	True

Checking for missing values…

```
# filtering data - displaying data only with Exam1 = NaN
scores[pd.isnull(scores["Exam1"])]
```

Name	Exam1	Exam2	Exam3
Bob	NaN	45.0	NaN
Maria	NaN	NaN	98.0
Frank	NaN	87.0	90.0

```
# filtering data - displaying data if Exam1 has a value
scores[pd.notnull(scores["Exam1"])]
```

Name	Exam1	Exam2	Exam3
Tom	100.0	30.0	40.0
Sam	90.0	56.0	80.0
Eve	95.0	75.0	NaN
Grace	85.0	90.0	NaN

Imputation (Filling Missing Data)

Name	Exam1	Exam2	Exam3
Tom	100.0	30.0	40.0
Bob	NaN	45.0	NaN
Sam	90.0	56.0	80.0
Maria	NaN	NaN	98.0
Eve	95.0	75.0	NaN
Frank	NaN	87.0	90.0
Grace	85.0	90.0	NaN

scores.fillna(0)

fillna(n) to fill null values with a specific value 'n'

Name	Exam1	Exam2	Exam3
Tom	100.0	30.0	40.0
Bob	0.0	45.0	0.0
Sam	90.0	56.0	80.0
Maria	0.0	0.0	98.0
Eve	95.0	75.0	0.0
Frank	0.0	87.0	90.0
Grace	85.0	90.0	0.0

scores.fillna(method ='pad')

fillna(method='pad') to fill null values with the previous ones

Name	Exam1	Exam2	Exam3
Tom	100.0	30.0	40.0
Bob	100.0	45.0	40.0
Sam	90.0	56.0	80.0
Maria	90.0	56.0	98.0
Eve	95.0	75.0	98.0
Frank	95.0	87.0	90.0
Grace	85.0	90.0	90.0

Imputation…

Name	Exam1	Exam2	Exam3
Tom	100.0	30.0	40.0
Bob	NaN	45.0	NaN
Sam	90.0	56.0	80.0
Maria	NaN	NaN	98.0
Eve	95.0	75.0	NaN
Frank	NaN	87.0	90.0
Grace	85.0	90.0	NaN

df.fillna(method ='bfill')

- fillna(method='bfill') to fill null value with the next ones

Name	Exam1	Exam2	Exam3
Tom	100.0	30.0	40.0
Bob	90.0	45.0	80.0
Sam	90.0	56.0	80.0
Maria	95.0	75.0	98.0
Eve	95.0	75.0	90.0
Frank	85.0	87.0	90.0
Grace	85.0	90.0	NaN

df.fillna(scores.mean())

- To fill in the missing values in each column using the mean of that column

Name	Exam1	Exam2	Exam3
Tom	100.0	30.000000	40.0
Bob	92.5	45.000000	77.0
Sam	90.0	56.000000	80.0
Maria	92.5	63.833333	98.0
Eve	95.0	75.000000	77.0
Frank	92.5	87.000000	90.0
Grace	85.0	90.000000	77.0

```
#Fill in a selected column
df[["Exam1"]].fillna(df["Exam1"].mean())
```

Imputation…

```
# creating a DataFrame
dic = {'Section':['A', 'B', 'A', 'A', 'B', 'A', 'B'],
        'Name': ['Tom', 'Bob', 'Sam', 'Maria', 'Eve', 'Frank', 'Grace'],
        'Exam1':[100, np.nan, 90, np.nan, 95,  np.nan, 85],
        'Exam2': [30, 45, 56, np.nan, 75, 87, 90],
        'Exam3':[ 40, np.nan, 80, 98, np.nan, 90, np.nan]}
 scores = pd.DataFrame(dic)
```

- Filling a null values using groupby() method

Name	Exam1	Exam2	Exam3
Tom	100.0	30.0	40.0
Bob	NaN	45.0	NaN
Sam	90.0	56.0	80.0
Maria	NaN	NaN	98.0
Eve	95.0	75.0	NaN
Frank	NaN	87.0	90.0
Grace	85.0	90.0	NaN

```
# using isnull() function
scores.isnull()
```

Name	Exam1	Exam2	Exam3
Tom	False	False	False
Bob	True	False	True
Sam	False	False	False
Maria	True	True	False
Eve	False	False	True
Frank	True	False	False
Grace	False	False	True

Imputation…

- Filling a null values using groupby() method

```
scores["Exam1"].fillna(scores.groupby("Section")
                              ["Exam1"].transform("mean"),
                              inplace=True)
```

	Section	Name	Exam1	Exam2	Exam3
0	A	Tom	100.0	30.0	40.0
1	B	Bob	NaN	45.0	NaN
2	A	Sam	90.0	56.0	80.0
3	A	Maria	NaN	NaN	98.0
4	B	Eve	95.0	75.0	NaN
5	A	Frank	NaN	87.0	90.0
6	B	Grace	85.0	90.0	NaN

	Section	Name	Exam1	Exam2	Exam3
0	A	Tom	100.0	30.0	40.0
1	B	Bob	90.0	45.0	NaN
2	A	Sam	90.0	56.0	80.0
3	A	Maria	95.0	NaN	98.0
4	B	Eve	95.0	75.0	NaN
5	A	Frank	95.0	87.0	90.0
6	B	Grace	85.0	90.0	NaN

Imputation…

- Interpolation is a type of estimation, a method of constructing new data points within the range of a discrete set of known data points.

- Using interpolate() function

```
#to interpolate the missing values
scores.interpolate(method ='linear',
                   limit_direction ='forward')
```

  - method : {‘linear’, ‘time’, ‘index’, ‘values’, ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘barycentric’, ‘krogh’, ‘polynomial’, ‘spline’, ‘piecewise_polynomial’, ‘from_derivatives’, ‘pchip’, ‘akima’}

  - limit_direction *:* {‘forward’, ‘backward’, ‘both’}

Removing Incomplete data

- Incomplete records might need to be removed from the dataset

- dropna() function can be used to drop null values from a DataFrame
  - It can drop Rows/Columns of datasets with Null values in different ways.
  - To drop all rows with any NA values: **data.dropna()**
  - To drop rows that have all NA values: **data.dropna(how='all')**
  - To put a limitation on how many non-null values need to be in a row in order to keep it **data.dropna(thresh=5)**

Default :
- How='any'
- Axis =0

2. Duplicate Data

- If two objects represent the same entity, the values of their corresponding features might differ, and these inconsistencies need to be resolved
  - Care must be taken to avoid accidentally combining data objects that are similar but not duplicates, such as two distinct individuals with identical names.

- The term deduplication is often used to refer to the process of dealing with these issues.

Eliminating Duplicate Entries

- DataFrame.drop_duplicates() returns DataFrame with duplicate rows removed, optionally only considering certain columns

df

	Name	Age	Sex
0	James	24	Male
1	Alice	28	Female
2	Phil	40	Male
3	James	24	Male

```
new_df= df.drop_duplicates()
new_df
```

	Name	Age	Sex
0	James	24	Male
1	Alice	28	Female
2	Phil	40	Male

Eliminating Duplicate Entries…

- To remove duplicates of only one column or a subset of columns
  - Specify the column(s) that should be unique

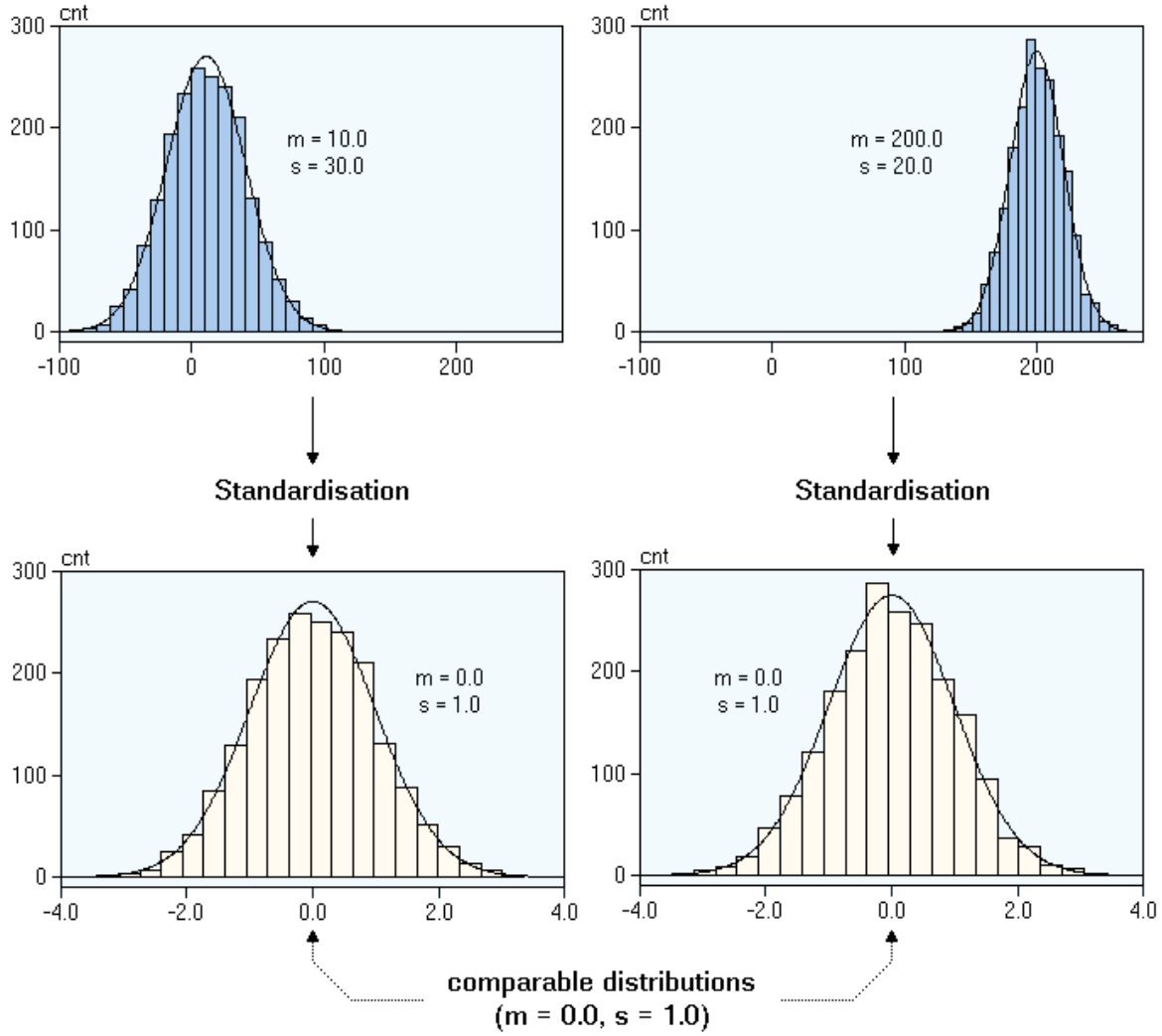
```
data.drop_duplicates(subset='Name')
```

- To do this conditional on a different column's value,
  - Use sort_values(col_name) and specify **keep equals either first or last**

```
data = data.sort_values('Start Date')
data.drop_duplicates(subset='Name', keep='first')
```

- 'first' (default): Drop duplicates except for the first occurrence.
- 'last': Drop duplicates except for the last occurrence.
- False: Drop all duplicates.

3. Different Ranges



Not normalized
- Different range
- Hard to compare
- May influence the result

Normalized
- Similar value range
- Similar influence on the analytical model

https://medium.com/

Why Normalization?

Scale Independence	▪ Normalization rescales the features to a standard range, typically between 0 and 1 or -1 and 1. ▪ This makes the features more comparable and eliminates any potential bias introduced by different scales.
Avoidance of Dominance	▪ In some ML algorithms, features with larger scales can dominate the learning process and have an unequal influence on the results. ▪ Normalization prevents this by ensuring that all features contribute equally to the analysis.
Outlier Handling	▪ Normalization can help in handling outliers by bringing the values within a similar range. ▪ Outliers, which are extreme values that deviate significantly from the majority of the data, can unevenly impact analysis.
Algorithm Compatibility	▪ Some ML algorithms, such as K-nearest neighbors (KNN) or support vector machines (SVM), rely on distance-based calculations. ▪ Normalizing the data ensures that the distances between samples are calculated accurately and consistently

Normalization

- Normalization improves the reliability and effectiveness of data analysis by creating a more meaningful and fair representation of the data for modeling and analysis.

- Normalizing techniques solve this problem:
  - Min-max normalization – it guarantees all features will have the exact same scale but does not handle outliers well
  - Z-score normalization- it handles outliers but does not produce normalized data with the *exact* same scale

Min Max Normalization

- It performs the linear transformation on original data
  - The data will be converted to fall between the range of 0 to 1. (Default)

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

```
import numpy as np
from sklearn.preprocessing import minmax_scale

data= np.array([5, 20, 30, 45, 70, 100])

scaled_data = minmax_scale(data)

print(scaled_data)
```

```
[0.         0.15789474 0.26315789 0.42105263 0.68421053 1.        ]
```

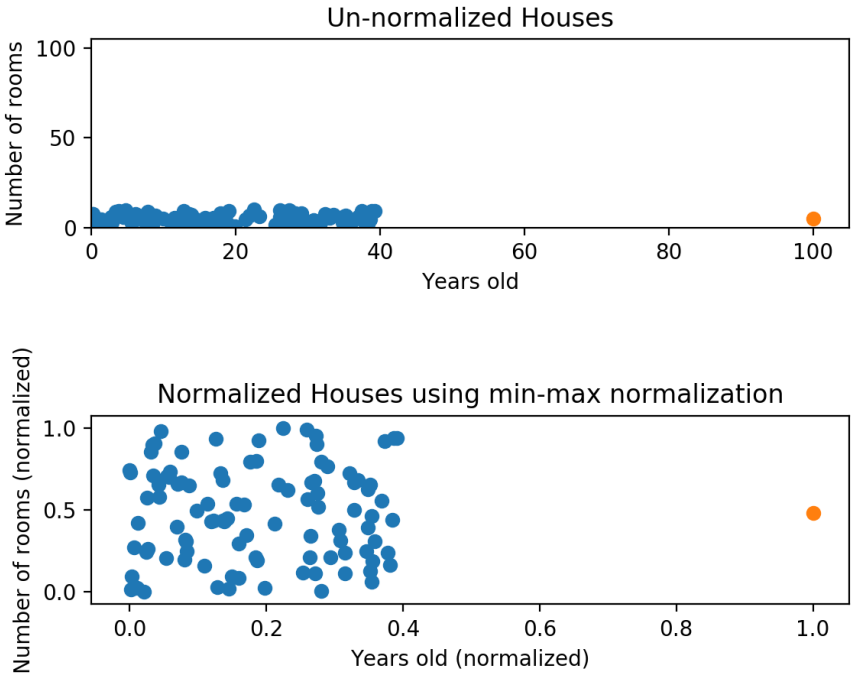
  - feature_range (min, max) specifies the desired range of transformed data.

```
scaled_data = minmax_scale(data, feature_range=(1, 10))
```

```
[ 1.          2.42105263  3.36842105  4.78947368  7.15789474 10.        ]
```

Min Max Normalization…

- Min-max normalization has a significant downside. It does not handle outliers very well
  - E.g., if you have 99 values between 0 and 40, and one value is 100, then the 99 values will all be transformed to a value between 0 and 0.4.
  - That data is just as squished as before!



Source of figure: https://www.codecademy.com/

Z-score normalization

- Z-score gives you an idea of how far from the mean a data point is
- Z-score normalization is the process of rescaling the features so that they'll have the properties of a Gaussian distribution with μ=0 and σ=1
  - where μ is the mean and σ is the standard deviation from the mean

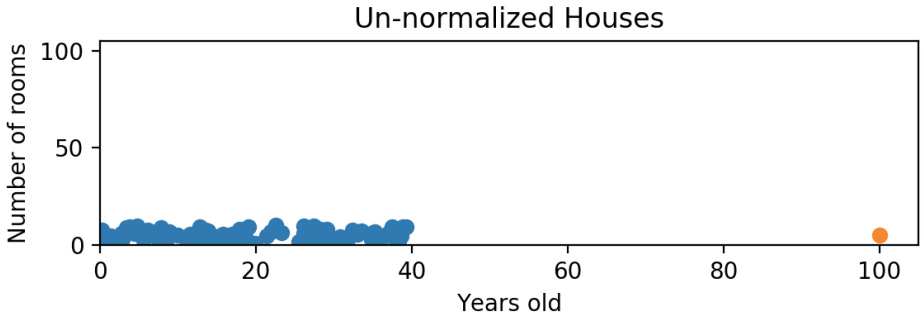
$$z = \frac{x - \mu}{\sigma}$$

  - If a value is exactly **equal** to the mean of all the values of the feature, it will be normalized to **0**.
  - If it is **below** the mean, it will be a **negative** number, and if it is **above** the mean it will be a **positive** number.

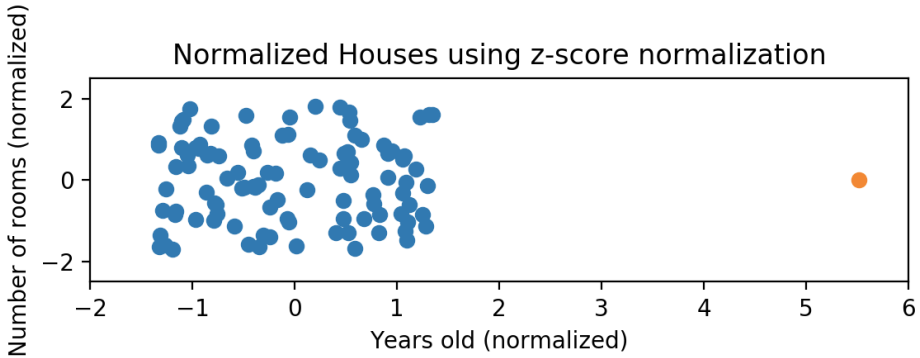
Z-score normalization…

- Z-score normalization is a strategy of normalizing data that avoids the outliers' issue.

```
from scipy import stats
stats.zscore(data)
```



- Now, the points are on roughly the same scale for both features
- Almost all points are between -2 and 2 on both the x-axis and y-axis.



→ This is not the case with Max-Min Normalization

Source of figures: https://www.codecademy.com/

4. Outliers & Noisy Data

- Outliers are data values that differ significantly from others in a dataset

- Outliers skew your data distributions and affect all your basic central tendency statistics
  - Means are pushed upward or downward, influencing all other descriptive measures
  - An outlier will always inflate variance and modify correlations, so you may obtain incorrect assumptions about your data and the relationships between variables

Detection of Outliers

- Outliers are one of the trickiest problems to solve
  - because there is no single definition of outliers, and their presence in your data may not always have a clear reason

- Most common causes of outliers on a data set:
  - Data entry errors (human errors)
  - Measurement errors (instrument errors)
  - Experimental errors (executing errors)
  - Intentional (dummy outliers made to test detection methods)
  - Data processing errors (data manipulation or data set unintended mutations)
  - Sampling errors (extracting or mixing data from wrong or various sources)
  - Natural (not an error, novelties in data)

As a result, much is left to your investigation and evaluation.

Detection of Outliers…

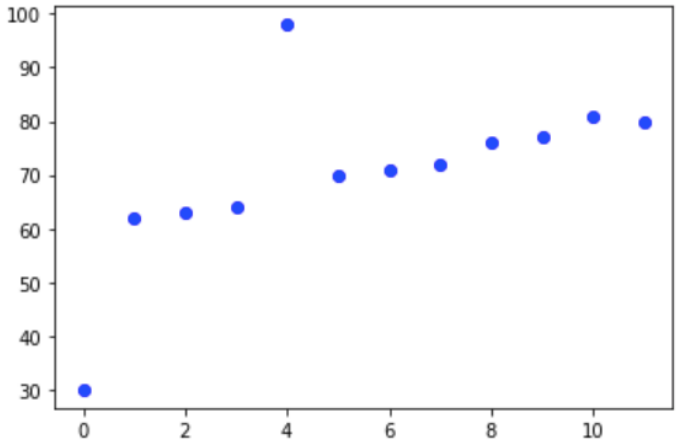
- Common Methods for Detecting Outliers
  - Plots
  - Known bounds
  - Interquartile range (IQR)
  - Z-Score

Detection of Outliers: Using Plots

- The quickest and easiest way to identify outliers is by visualizing them using plots.
  - If your dataset is not huge (approx. up to 10k observations & 100 features), build scatter plots & box-plots of variables

```
dataset=np.array([30, 62, 63, 64, 98, 70,71, 72, 76, 77, 81, 80])
```

```
import matplotlib.pyplot as plt


plt.plot(dataset, 'o', color='blue');
plt.show()
```



Detection of Outliers: Using known bounds

- If you have the lower-bound and the upper-bound of the dataset you have, you can check according to those known bounds.
  - e.g. Students scores [0,100]

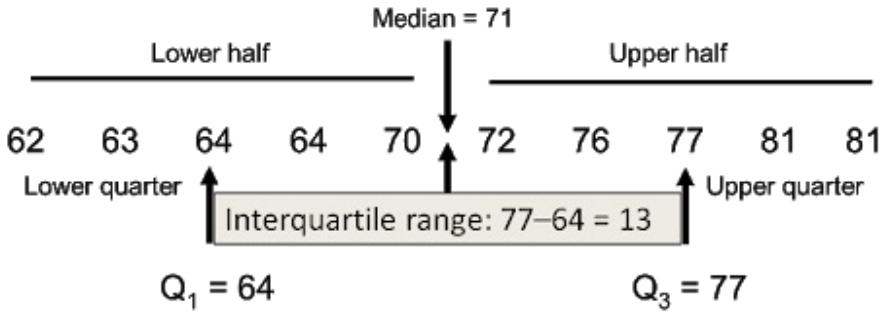
```
lower_bound = 0;
upper_bound = 100;

scores = np.array([30, 62, 63, 64, 110, 70,71, 72, 76, 77, 65, 54, -5])
np.where((scores< lower_bound) | (scores > upper_bound))
```

```
(array([ 4, 12]),)
```

Detection of Outliers: Using IQR

- Interquartile range can be used to tell when a value is too far from the middle

64 - (1.5*13) =44.5



77 + (1.5*13) =96.5

An outlier is a point that falls more than 1.5 times the IQR above Q3 or below Q1

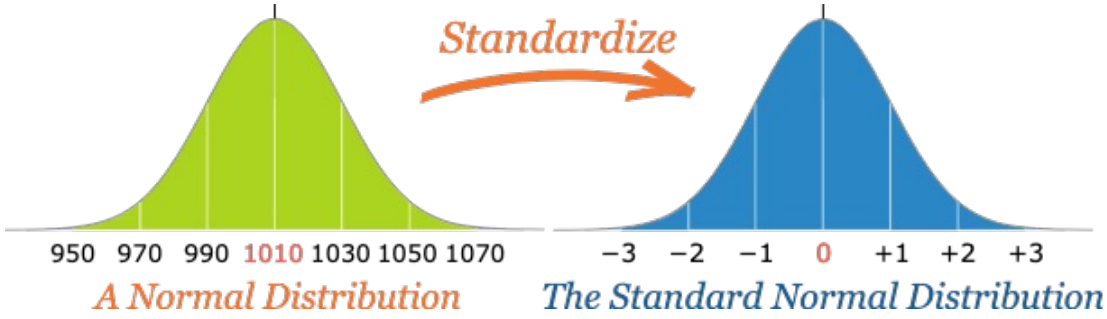
- Steps:
  - Arrange the data in an increasing order
  - Find the median of the lower and upper half of the data (i.e. Q1 and Q3)
  - Find IQR = Q3-Q1
  - Add (1.5 x IQR) to the Q3. Any number greater than this is a suspected outlier.
  - Subtract (1.5 x IQR) from Q1. Any number less than this is a suspected outlier

Detection of Outliers: Using IQR…

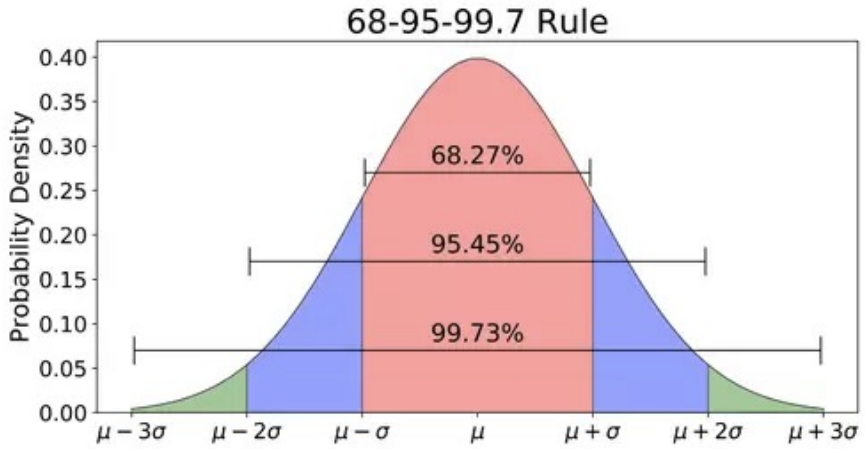
```
#Using IQR

data =np.array([30, 62, 63, 64, 98, 70,71, 72, 76, 77, 81, 80])

#Finding first quartile and third quartile
q1, q3= np.percentile(data,[25,75])

#Find the IQR which is
iqr = q3 - q1

# calculate the outlier cutoff
cut_off = iqr * 1.5

# Find lower and upper bound
lower, upper = q1 - cut_off, q3 + cut_off

print(F"Lower = {lower}, Upper = {upper}" )

# identify outliers
outliers = [x for x in data if x < lower or x > upper]
print('Identified outliers: %d' % len(outliers))

# remove outliers
outliers_removed = [x for x in data if x >= lower and x <= upper]
print('Non-outlier observations: %d' % len(outliers_removed))
```

Detection of Outliers: Using Z-Score

- Z-score helps to understand if a data value is greater or smaller than mean and how far away it is from the mean.



- about 68% of measurements will be between -1 and +1 standard deviations from the mean
- about 95.5% of measurements will be between -2 and +2 standard deviations from the mean
- about 99.7% of measurements will be between -3 and +3 standard deviations from the mean



Source of figures:
https://www.simplypsychology.org/z-score.html

Detection of Outliers: Using Z-Score

```
# using z-score
import numpy as np

data =np.array([30, 62, 63, 64, 98, 70,71, 72, 76, 77, 81, 80])

# to be compared with z-score
threshold= 2

# find the mean and standard deviation
# of the all the data points
mean_1 = np.mean(data)
std_1 =np.std(data)

outliers=[]

# find z-score for each point & identify outliers
for y in data:
  z_score= (y - mean_1)/std_1
  if np.abs(z_score) > threshold:
    outliers.append(y)

#show the detected values
print('Identified outliers: %d' % len(outliers))
print(outliers)
```

→ if the z score is greater than 3 than we can classify that point as an outlier.

```
Identified outliers: 1
[30]
```

Hands-On Exercises