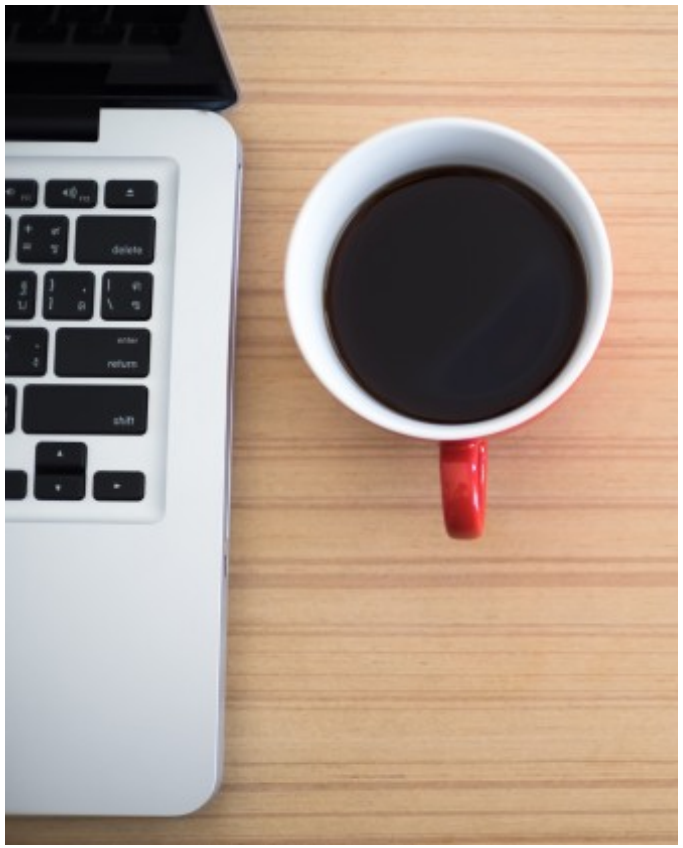
Fourth Industrial Summer School

Day 4

Data Preprocessing

Session Objectives

- ✓ Data Normalization
- ✓ Data Cleaning
- ✓ Handling Missing Data
- ✓ Replacing Values
- ✓ Removing Duplicates
- ✓ Handling Outliers
- ✓ Handling Noisy Data



Data Preprocessing

- The process of converting or mapping data from the initial "raw" form into another format, in order to prepare the data for further analysis.
- The process of making the data more suitable for data mining
- The tasks employed in this process are informed by the process of data understanding.

Normalization

- Normalization is a technique often applied as part of data preparation for machine learning.
- The goal of normalization is to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values.
- For machine learning, every dataset does not require normalization.
  - It is required only when features have different ranges.

Why do we normalize?

Age	Income
20	100000
30	20000
40	500000

Age	Income
0.2	0.2
0.3	0.04
0.4	1

Not normalized
- Different range
- Hard to compare
- Income will influence the result more

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

- → there is a need to "eliminate" the unit of measurement, and this operation is called normalizing the data.
- So, normalization brings any dataset to a comparable range.
  - It could be to squash down the data to fit between the range of [0,1] or [-1,1] or anything else!

Min Max Normalization

- Min Max Normalization transforms a value A to B which fits in the range [C,D].
- This ensures that no matter what scale your data is in, it will be converted to fall between the range of 0 to 1.

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

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

data= np.array([11,33,4,5,9,2])

scaled_data = minmax_scale(data)

print(scaled_data)
```

```
[0.29032258 1.         0.06451613 0.09677419 0.22580645 0.
```

Max Normalization

- Max is quite similar to Min Max normalization.
- The only difference being is that the the normalized values will fall between a range of 1 and to a value less than or equal to 0.

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

data= np.array([11,33,4,5,9,2])

normalize(data.reshape(1, -1), norm="max")
```

```
array([[0.33333333, 1.        , 0.12121212, 0.15151515, 0.27272727,
        0.06060606]])
```

Z-score normalization

- **Z-score normalization** (or Standardization) is the process of rescaling the features so that they'll have the properties of a Gaussian distribution with

$$X_{new}=(X_{old}-\mu) / \sigma$$

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

```
array([[-1.38218948, -1.40182605],
       [-0.95025527,  0.03594426],
       [ 1.2094158 , -0.50321961],
       [ 0.34554737,  0.21566555],
       [ 0.77748158,  1.65343586]])
```

Data Cleaning

- Data in real world is
  - Incomplete: lacking attribute values, lacking certain attributes of interest, or containing only aggregate data
  - Noisy: containing errors or outliers
  - Inconsistent: containing discrepancies in codes or names

- No quality data, no quality mining results
  - Quality decision must be based on quality data
  - Data warehouse needs consistent integration of quality data

- Thus, during the data analysis, there is a need to detect the presence of abnormal values within a data structure.

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 dur to
  - Equipment malfunction
  - Inconsistent with other recorded data and thus deleted
  - Data not enterted due to misunderstanding
  - Certain data may not be considered important at the time of entry
  - Not register history or changes of the data
- Missing data may need to be inferred.

How to Handle Missing Data?

- Ignore the tuple: usually done when class lable is missing (assuming the tasks in classification- not effective when the percentage of missing values per attribute varies considerably)

- Fill in the missing value
  - Manually: tedious + infeasible?
  - Use global constant to fill in the missing value: e.g., "unkowown", a new class?!
  - Use the attribute mean to fill in the missing value
  - Use the most probable value to fill in the missing value: inference-based such as Bayesian formula or decision tree

Filling-in Missing Data

- Ignore the instance:
  - often not very effective, especially when few features are missing.
- Fill in the missing value manually:
  - tedious and typically infeasible.
- Use a global constant to fill in the missing value:
  - e.g., "unknown". May be mistaken for concept.
- Fill in the missing value
  - using the feature mean or other analysis of the variable

Handling Missing Data using Pandas

- One of the goals of pandas is to make working with missing data as painless as possible.
  - For example, all of 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 using Pandas

- Pandas provides a number of methods to deal with missing values in the data frame:

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 using isnull()

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

```
# creating a dataframe from list
dict = {'Exam1':[100, 90, np.nan, 95],
        'Exam2': [30, 45, 56, np.nan],
        'Exam3':[np.nan, 40, 80, 98]}

scores = pd.DataFrame(dict)

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

	Exam1	Exam2	Exam3
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	NaN	56.0	80.0
3	95.0	NaN	98.0

	Exam1	Exam2	Exam3
0	False	False	True
1	False	False	False
2	True	False	False
3	False	True	False

Checking for missing values using isnull()

```
# making dataframe from csv file
data = pd.read_csv("employees.csv")

# creating bool series True for NaN values
bool_series = pd.isnull(data["Gender"])

# filtering data
# displaying data only with Gender = NaN
data[bool_series]
```

	Name	Gender	Start Date
20	Lois	NaN	4/22/95
22	Joshua	NaN	3/8/12
27	Scott	NaN	7/11/91
31	Joyce	NaN	2/20/05
41	Christine	NaN	6/28/15
49	Chris	NaN	1/24/80

As shown in the output image, only the rows having **Gender = NULL** are displayed.

Checking for missing values using notnull()

- notnull() function returns dataframe of Boolean values which are False for NaN values.

```
# making dataframe from csv file
data = pd.read_csv("employees.csv")

# creating bool series True for NaN values
bool_series = pd.notnull(data["Gender"])

# filtering data
# displaying data only with Gender = NaN
data[bool_series]
```

	Name	Gender	Start Date
0	Douglas	Male	8/6/93
1	Thomas	Male	3/31/96
2	Maria	Female	4/23/93
3	Jerry	Male	3/4/05
4	Larry	Male	1/24/98

Filling missing values

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

```
# filling missing value using fillna()
scores.fillna(0)
```

All missing scores will be filled by zero

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

```
# filling a missing value with previous ones
scores.fillna(method ='pad')
```

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

```
# filling a missing value with next ones
scores.fillna(method ='bfill')
```

Filling missing values

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

```
#Fillin with the mean (slective column)
scores["Exam1"].fillna(scores["Exam1"].mean())
```

Filling missing values

- To use groupby

	Level	Exam1	Exam2	Exam3
0	A	100.0	30.0	NaN
1	B	90.0	45.0	40.0
2	A	NaN	56.0	80.0
3	A	95.0	NaN	98.0
4	B	NaN	100.0	100.0

	Level	Exam1	Exam2	Exam3
0	A	100.0	30.0	NaN
1	B	90.0	45.0	40.0
2	A	97.5	56.0	80.0
3	A	95.0	NaN	98.0
4	B	90.0	100.0	100.0

```
# fill in the missing value in Exam1
# with the mean with considering the level groups

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

Filling missing values

- Filling a null values using replace() method

```
# replace Nan value in dataframe with value 0
scores.replace(to_replace = np.nan, value = 0)
```

- Using interpolate() function to fill the missing values using linear method.

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

Notice that, NaN values in the first row could not get filled as the direction of filling of values is forward and there is no previous value which could have been used in interpolation.

Remove incomplete rows

- dropna() function can be used to drop a 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()**

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

- 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)**

Deleting Missing Values in NumPy Arrays

- numpy.isnan(array [, out]) **t**est element-wise whether it is NaN or not return the result as a boolean array.

```
# Create feature matrix
X = np.array([[1, 2],
              [6, 3],
              [8, 4],
              [9, 5],
              [np.nan, 4]])

# Remove observations with missing values
X[~np.isnan(X).any(axis=1)]
```

```
array([[1., 2.],
       [6., 3.],
       [8., 4.],
       [9., 5.]])
```

Removing Duplicate Data

- Removing duplicate data raises two issues:
  - If there are two objects that actually represent a single object, then the values of corresponding features may differ, and these inconsistent values must be resolved.
  - Care needs to be taken to avoid accidentally combining data objects that are similar, but not duplicates, such as two distinct people with identical names.

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

Removing Duplicate Data

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

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

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

df

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

new_df

Removing Duplicate Data

- To remove duplicates of only one or a subset of columns:
  - specify subset as the individual column or list of columns that should be unique.
- To do this conditional on a different column's value, you can sort_values(colname) and specify keep equals either first or last.

```
df = df.sort_values('Age',
                    ascending=False)
df = df.drop_duplicates(subset='Name',
                        keep='first')
df
```

	Name	Age	Sex
0	James	24	Male
1	Alice	28	Female
2	Phil	40	Male
3	James	25	Male

	Name	Age	Sex
2	Phil	40	Male
1	Alice	28	Female
3	James	25	Male

Detecting and Filtering Outliers

- Outliers are data that differ significantly from other data in a dataset.
  - i.e. one or more values are too high or too low when compared to the majority of the values.

- 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.

Detecting and Filtering Outliers

- Outliers is one of the trickiest problem to solve,
  - because you don't always have a unique definition of outliers, or
  - a clear reason to have them in your data.

- Most common causes of outliers on a data set:
  - Data entry errors (human errors)
  - Measurement errors (instrument errors)
  - Experimental errors (data extraction or experiment planning/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.

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]

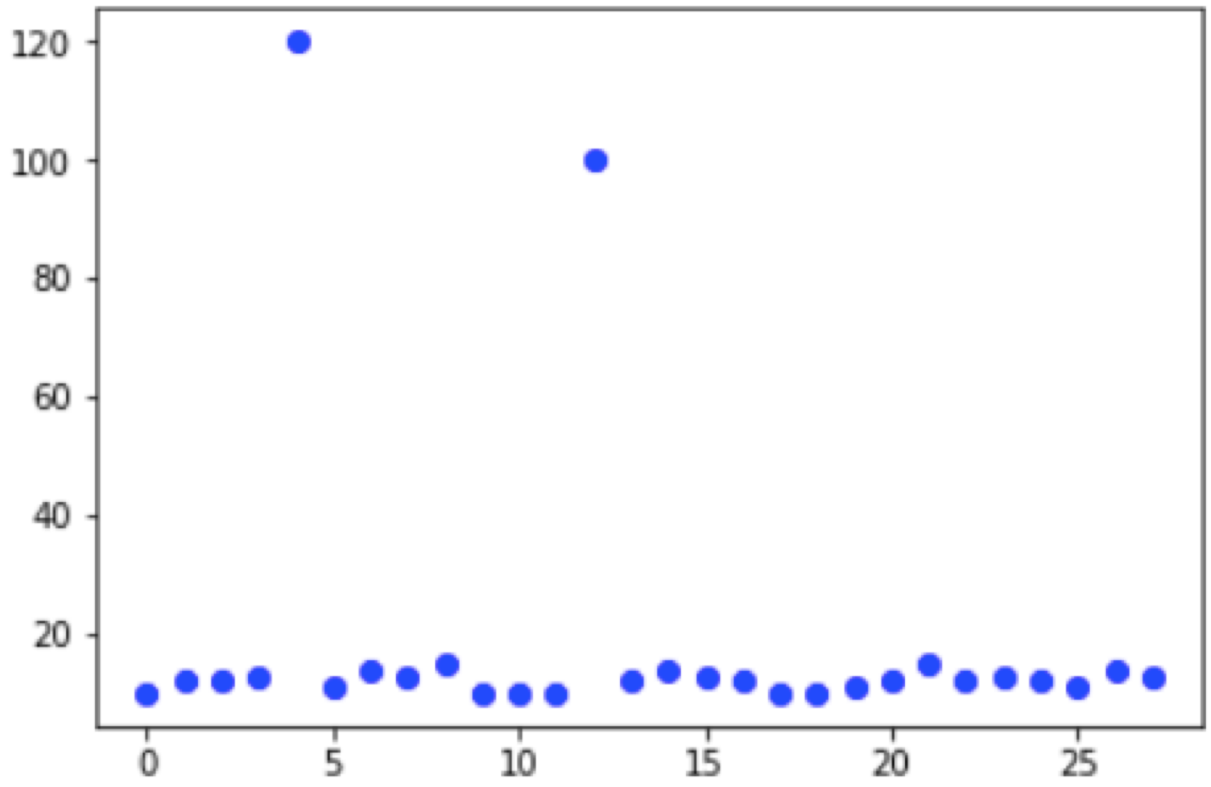
```
lower_bound=0;
upper_bound=100;

df['score'] = np.where((df['score'] < lower_bound) |
                       (df['score'] < upper_bound),
                       0, 1)
```

Using Plot

```
import matplotlib.pyplot as plt


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



Using IQR

- IQR tells how spread the middle values are.
- It can be used to tell when a value is too far from the middle.

- An outlier is a point which falls more than 1.5 times the interquartile range above the third quartile or below the first quartile.

- Steps:
  - Arrange the data in increasing order
  - Calculate first(q1) and third quartile(q3)
  - Find interquartile range (q3-q1)
  - Find lower bound q1*1.5
  - Find upper bound q3*1.5
  - Anything that lies outside of lower and upper bound is an outlier

Using IQR

```
#Using IQR

dataset= np.array([10,12,12,13,120,11,14,
           13,15,10,10,10,100,12,
           14,13, 12,10, 10,11,12,
           15,12,13,12,11,14,13])

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

#Find the IQR which is
#the difference between third and first quartile
iqr = q3 - q1

# Find lower and upper bound

lower_bound = q1 -(1.5 * iqr)
upper_bound = q3 +(1.5 * iqr)

print(lower_bound)
print(upper_bound)

#Create dataframe for the values
df = pd.DataFrame(dataset,columns=['score'])

df['score'] = np.where((df['score'] < lower_bound) |
                       (df['score'] < upper_bound),
                       0, 1)
# Show data
df
```

Using Z-Score

- $z = (X — \mu) / \sigma$

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

```
# using z-score

import numpy as np
import pandas as pd

#data
dataset= [10,12,12,13,120,11,14,
          13,15,10,10,10,100,12,
          14,13, 12,10, 10,11,12,
          15,12,13,12,11,14,13]

outliers=[]

# to be compared with z-score
threshold= 3

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

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

#show the detected values
outliers
```

```
[120, 100]
```

Noisy Data

- Noise: random error or variance in a measured variable
- Incorrect attribute values may due to
  - Faulty data collection instruments
  - Data entry problems
  - Data transmission problems
  - Technology limitation
  - Inconsistency in naming convention
- Other data problems which requires data cleaning
  - Duplicate records
  - Incomplete data
  - Inconsistent data

How to Handle Noisy Data ?

- Binning method:
  - First sort data and partition into (equal-depth) bins
  - Then smooth by bin means, smooth by bin median, smooth by bin boundaries, etc.
- Clustering
  - Detect and remove outliers
- Combined computer and human inspection
  - Detect suspicious values and check by human
- Regression
  - smooth by filling the data into regression functions

Hands on session

Problem Solving