2024S2 - IT2313

Programming for Data Science

Data Manipulation with Pandas (Part 2)

Data Transformation

Data Transformation

Combining Two DataFrames

In the following dataset, the first column contains information about student identifier and the second column contains their respective scores in any subject. The structure of the two data frames are the same in both case. Let try to concatenate both.

```
df1 = pd.DataFrame({'StudentID': [1, 3, 5, 7, 9],
                    'Score' : [89, 39, 50, 97, 22]})
df2 = pd.DataFrame({'StudentID': [2, 4, 6, 8, 10],
                    'Score': [98, 93, 44, 77, 69]})
```

```
df1
```

	StudentID	Score
0	1	89
1	3	39
2	5	50
3	7	97
4	9	22

```
df2
```

	StudentID	Score
0	2	98
1	4	93
2	6	44
3	8	77
4	10	69

```
# We can do that by using Pandas concat() method.

df_vertical = pd.concat([df1, df2], ignore_index=True)
df_vertical
```

	StudentID	Score
0	1	89
1	3	39
2	5	50
3	7	97
4	9	22
5	2	98
6	4	93
7	6	44
8	8	77
9	10	69

Data Transformation

Combining Two DataFrames

```
df1 = pd.DataFrame({'StudentID': [1, 3, 5, 7, 9],
                    'Score' : [89, 39, 50, 97, 22]})
df2 = pd.DataFrame({'StudentID': [2, 4, 6, 8, 10],
                    'Score': [98, 93, 44, 77, 69]})
```

```
df1
```

	StudentID	Score
0	1	89
1	3	39
2	5	50
3	7	97
4	9	22

```
df2
```

	StudentID	Score
0	2	98
1	4	93
2	6	44
3	8	77
4	10	69

```
df_horizontal = pd.concat([df1, df2], axis=1)
```

```
df_horizontal
```

	StudentID	Score	StudentID	Score
0	1	89	2	98
1	3	39	4	93
2	5	50	6	44
3	7	97	8	77
4	9	22	10	69

Data Transformation

Combining Two DataFrames

In the previous example, you received two files for same subject. Now, consider that you are teaching two courses. So, you will get two dataframes from each sections: two for Software Engineering course and another two for Machine Learning course.

```
df1SE =  pd.DataFrame({'StudentID': [9, 11, 13, 15, 17],
                       'ScoreSE' : [22, 66, 31, 51, 71]})
df2SE =  pd.DataFrame({'StudentID': [2, 4, 6, 8, 10],
                       'ScoreSE': [98, 93, 44, 77, 69]})

df1ML =  pd.DataFrame({'StudentID': [1, 3, 5, 7, 9],
                       'ScoreML' : [39, 49, 55, 77, 52]})
df2ML =  pd.DataFrame({'StudentID': [2, 4, 6, 8, 10],
                       'ScoreML': [93, 44, 78, 97, 87]})
```

```
dfSE = pd.concat([df1SE, df2SE], ignore_index=True)
dfML = pd.concat([df1ML, df2ML], ignore_index=True)

df = pd.concat([dfML, dfSE], axis=1)
df
```

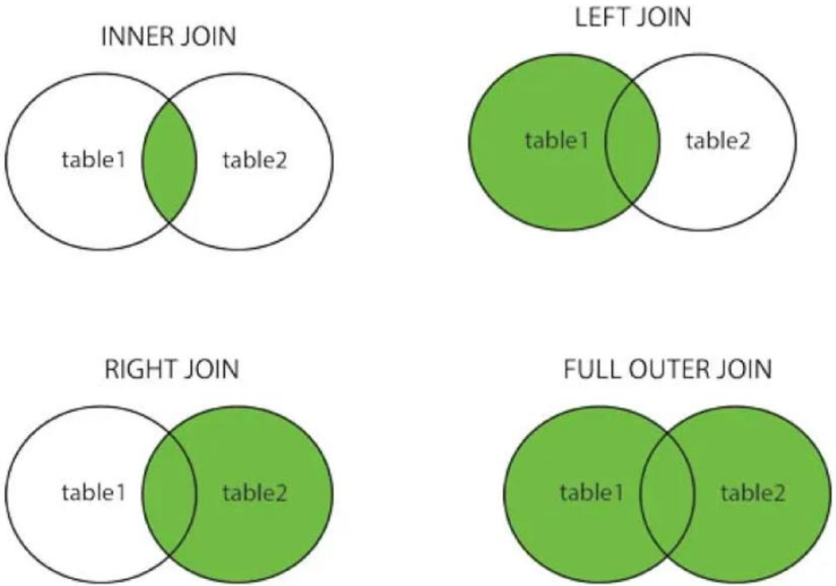
	StudentID	ScoreML	StudentID	ScoreSE
0	1	39	9	22
1	3	49	11	66
2	5	55	13	31
3	7	77	15	51
4	9	52	17	71
5	2	93	2	98
6	4	44	4	93
7	6	78	6	44
8	8	97	8	77
9	10	87	10	69

Data Transformation

Joins in DataFrame

In Pandas, joins can be performed using the merge function. The merge function combines two DataFrames based on the values of one or more columns. There are four types of joins in Pandas: **inner join**, **left join**, **right join**, and **outer join**.

[Joins In Pandas]



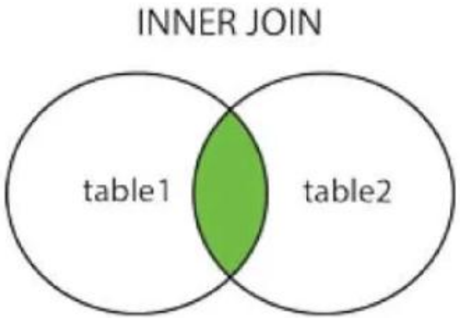
Data Transformation

Inner Join

Inner Join between two Pandas DataFrames combines rows that have matching values in specified columns, retaining only the rows with common keys in both Data Frames.

```
dfSE = pd.concat([df1SE, df2SE], ignore_index=True)
dfML = pd.concat([df1ML, df2ML], ignore_index=True)

df = dfSE.merge(dfML, how='inner')
df
```



	StudentID	ScoreSE	ScoreML
0	9	22	52
1	2	98	93
2	4	93	44
3	6	44	78
4	8	77	97
5	10	69	87

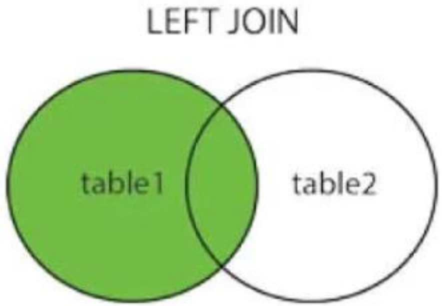
Data Transformation

Left Join

Left Join in Pandas combines rows from two DataFrames based on matching values in specified columns, keeping all rows from the left DataFrame and including matching rows from the right DataFrame. If there's no match in the right DataFrame, the result will have NaN values for columns from the right side.

```
dfSE = pd.concat([df1SE, df2SE], ignore_index=True)
dfML = pd.concat([df1ML, df2ML], ignore_index=True)

df = dfSE.merge(dfML, how='left')
df
```



	StudentID	ScoreSE	ScoreML
0	9	22	52.0
1	11	66	NaN
2	13	31	NaN
3	15	51	NaN
4	17	71	NaN
5	2	98	93.0
6	4	93	44.0
7	6	44	78.0
8	8	77	97.0
9	10	69	87.0

Data Transformation

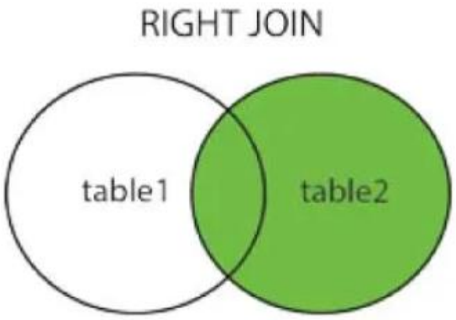
Right Join

A Right Join in Pandas merges two DataFrames based on matching values in specified columns, keeping all rows from the right DataFrame and including matching rows from the left DataFrame. If there's no match in the left DataFrame, the result will have NaN values for columns from the left side.

```
dfSE = pd.concat([df1SE, df2SE], ignore_index=True)
dfML = pd.concat([df1ML, df2ML], ignore_index=True)

df = dfSE.merge(dfML, how='right')
df
```

	StudentID	ScoreSE	ScoreML
0	1	NaN	39
1	3	NaN	49
2	5	NaN	55
3	7	NaN	77
4	9	22.0	52
5	2	98.0	93
6	4	93.0	44
7	6	44.0	78
8	8	77.0	97
9	10	69.0	87



Data Transformation

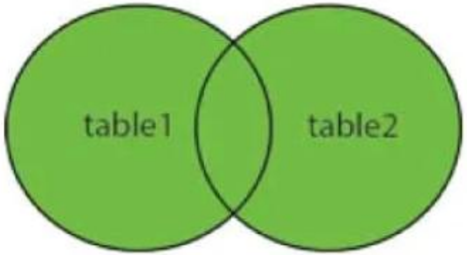
Outer Join

An Outer Join in Pandas merges two DataFrames based on matching values in specified columns, including all rows from both DataFrames and filling in with NaN values for non-matching columns.

```
dfSE = pd.concat([df1SE, df2SE], ignore_index=True)
dfML = pd.concat([df1ML, df2ML], ignore_index=True)

df = dfSE.merge(dfML, how='outer')
df
```

FULL OUTER JOIN



	StudentID	ScoreSE	ScoreML
0	9	22.0	52.0
1	11	66.0	NaN
2	13	31.0	NaN
3	15	51.0	NaN
4	17	71.0	NaN
5	2	98.0	93.0
6	4	93.0	44.0
7	6	44.0	78.0
8	8	77.0	97.0
9	10	69.0	87.0
10	1	NaN	39.0
11	3	NaN	49.0
12	5	NaN	55.0
13	7	NaN	77.0

Data Preparation

Data Preparation

Data preparation and cleaning in data science are crucial for reliable analytical results. Raw data is often incomplete and introducing errors which compromise model accuracy. Through data cleaning, we enhance data integrity and creating a robust foundation for analysis.

The following common dataset issues are discussed in this module:

- Columns and Data Renaming
- Conversion of Data Types
- Data Validation
- Data format revision
- Missing values / Empty Columns
- Duplicates
- Outliers
- Imbalances.

Data Preparation

Columns and Data Renaming

- DataFrame columns can be updated by providing a list of strings and assigning to the column's properties

```
marks = { "col1":[82, 70, 68, 75, 58],
          "col2":['A', 'B', 'C+', 'B+', 'C+'],
        }
df = pd.DataFrame(marks)
df
```

	col1	col2
0	82	A
1	70	B
2	68	C+
3	75	B+
4	58	C+

```
df.columns = ["Marks","Grade"]
df
```

	Marks	Grade
0	82	A
1	70	B
2	68	C+
3	75	B+
4	58	C+

Data Preparation

Columns and Data Renaming

- Replace values throughout the DataFrame.
- The replace() function replaces all occurrences of the value with the desired value.

```
# Sample dataframe
df = pd.DataFrame({'A': ['a','b','c'],
                   'B':['b','c','d']})
df
```

	A	B
0	a	b
1	b	c
2	c	d

```
# Replace b with e
df_rep = df.replace('b', 'e')
df_rep
```

	A	B
0	a	e
1	e	c
2	c	d

Data Preparation

Columns and Data Renaming

- Replace values throughout the DataFrame.
- Making using of the lambda function.
- These functions are defined using the lambda keyword, followed by a list of parameters, a colon, and the expression to be evaluated

```
age = { "col1":[82, 70, 68, 75, 58],
        "col2":[20, 28, 79, 88, 45],}
df = pd.DataFrame(age)
df
```

	col1	col2
0	82	20
1	70	28
2	68	79
3	75	88
4	58	45

```
df.apply(lambda age: age + 1)
```

	col1	col2
0	83	21
1	71	29
2	69	80
3	76	89
4	59	46

Data Preparation

Columns and Data Renaming

- Replace values in a particular column.

```
# Sample dataframe
df = pd.DataFrame({'A': ['a','b','c'],
                   'B':['b','c','d']})
df
```

	A	B
0	a	b
1	b	c
2	c	d

```
# Replace b with e
df_rep = df.replace({'A': 'b'}, 'e')
df_rep
```

	A	B
0	a	b
1	e	c
2	c	d

Data Preparation

Data Validation

- Range Check – It ensures the integrity of our data by verifying if a value falls within the expected boundaries. For instance, when dealing with a person's age, we might perform a range check to ensure it is a realistic and reasonable value.

- Type Check – It focuses on the data's structure. It ensures that a value is of the correct data type. For instance, storing a person's age as a string instead of a numerical value can lead to complications. Type checks safeguard against such mismatches, maintaining the accuracy and consistency of our data."

Age	
300	Not Valid
67	Valid
43	Valid

Age	dtype	
30	String	Not Valid
67	String	Not Valid
43	string	Not Valid

Data Preparation

Conversion of Data Types

- Data cleaning is a crucial step in the data analysis process, ensuring that the data is in the right format and structure for analysis. One common task is dealing with data types and formats.
- In a case where column numerical values are stored in string format. The function "to_numeric()" can be use to convert strings to numerical values.

```python
# Create a DataFrame with a column containing numeric
# values as strings
data = {'NumericColumn': ['1', '2', '3', '4']}
df = pd.DataFrame(data)

# Convert the 'NumericColumn' to numeric
df['NumericColumn'] = pd.to_numeric(df['NumericColumn'])

# Now 'NumericColumn' is of numeric type
df
```

	NumericColumn
0	1
1	2
2	3
3	4

```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 1 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   NumericColumn  4 non-null      int64
dtypes: int64(1)
memory usage: 164.0 bytes
```

Data Preparation

Data Format Revision

- Update values to an unexpected format
  - Dates
  - Names of options
  - Capitalization
- Ensures output in a consistent format

CustomerID	User_Name	Join_Date
440	CABBY13	2022-06-20
230	taxi#1	2020-08-03
559	NY_taxi	2021-12-05

CustomerID	Last_Ride_Date
440	7/01/2022
230	8/3/2020
559	1/31/2021

CustomerID	User_Name	Join_Date	Last_Ride_Date
440	cabby13	2022-06-20	2022-07-01
230	taxi#1	2020-08-03	2020-08-03
559	ny_taxi	2021-12-05	2021-01-31

Data Preparation

Date Format Revision

- Conversion of Date or Time Format before Calculating Mean or Median.
- When working with time data, it's essential to ensure that the time columns are in the right format. If not, you might encounter issues when trying to perform calculations like Mean or Median.

```
# Create a DataFrame with a column containing time values as strings
data = {'TimeColumn': ['12:30:45.678', '15:45:30.123', '18:20:15.999']}
df = pd.DataFrame(data)

# Convert the 'TimeColumn' to datetime format
df['TimeColumn'] = pd.to_datetime(df['TimeColumn'], format='%H:%M:%S.%f')

# Split the datetime column into separate date and time columns
df['Date'] = df['TimeColumn'].dt.date
df['Time'] = df['TimeColumn'].dt.time

# Display the DataFrame
df
```

	TimeColumn	Date	Time
0	1900-01-01 12:30:45.678	1900-01-01	12:30:45.678000
1	1900-01-01 15:45:30.123	1900-01-01	15:45:30.123000
2	1900-01-01 18:20:15.999	1900-01-01	18:20:15.999000

Data Preparation

```
# Create a DataFrame with a column containing date & time values as strings
data = {'DateTimeColumn': ['2023-11-19 12:30:45.678',
                           '2023-11-19 15:45:30.123',
                           '2023-11-19 18:20:15.999']}
df = pd.DataFrame(data)

# Convert the 'DateTimeColumn' to datetime format
df['DateTimeColumn'] = pd.to_datetime(df['DateTimeColumn'],
                                      format='%Y-%m-%d %H:%M:%S.%f')

# Split the datetime column into separate date, time, year, month, day,
# hour, minute, and second columns
df['Date'] = df['DateTimeColumn'].dt.date
df['Time'] = df['DateTimeColumn'].dt.time
df['Year'] = df['DateTimeColumn'].dt.year
df['Month'] = df['DateTimeColumn'].dt.month
df['Day'] = df['DateTimeColumn'].dt.day
df['Hour'] = df['DateTimeColumn'].dt.hour
df['Minute'] = df['DateTimeColumn'].dt.minute
df['Second'] = df['DateTimeColumn'].dt.second

# Display the DataFrame
df
```

	DateTimeColumn	Date	Time	Year	Month	Day	Hour	Minute	Second
0	2023-11-19 12:30:45.678	2023-11-19	12:30:45.678000	2023	11	19	12	30	45
1	2023-11-19 15:45:30.123	2023-11-19	15:45:30.123000	2023	11	19	15	45	30
2	2023-11-19 18:20:15.999	2023-11-19	18:20:15.999000	2023	11	19	18	20	15

Data comes from various sources in different formats. Converting dates and times to a standardized format allows for seamless integration and analysis of datasets.

Extracting specific components (year, month, day, hour, etc.) from dates and times allows for the creation of new features.

Converting date and time formats to a uniform format simplifies data cleaning, making it easier to identify and handle discrepancies.

Data Preparation

Missing Values / Empty Columns

- Can lead to errors
- Unrepresentative, biased results
- Can be null/empty/missing values
- Applied to column or collection of columns

Row/ Col	1	2	3	4	5	6
1	0.24	-0.1		0.18	0.42	-0.25
2	0.19	-0.22	-0.2	0.12	0.21	-0.26
3	0.21	0.09	0.57	-0.14	0.29	0.01
4	0.76	0.07	0.04	-0.06	0.3	-0.47
5	0.46	0.12	0.49	-0.42	0.28	-0.3
6	0.43	-0.23	-0.3	-0.24	0.23	
7	0.44	-0.32	0.26	-0.77	0.31	-0.09
8	0.11	0.03		-0.24	0.36	-0.11
9	0.32	0	0.26	-0.5	0.31	0.1
10	0.12	-0.01	-0.13	0.12	0.47	-0.3
11	0.53	0.25	0.49	-0.3	0.13	-0.12
12	0.17	0.06	0.06	0.28	0.38	-0.23
13	0.19	-0.06	0.05	-0.25	0.23	-0.05

Dealing with Missing Values / Empty Columns

- Drop missing or sparse rows/columns if given column has too many missing values.
- Use df.isnull() to check for null/empty/missing values.
- 5% or less of total values

```
# count missing values
print(df['oldpeak'].isnull().sum())

# Drop empty column(s) and row(s)
columns_dropped = heart_disease_df.drop(['oldpeak'], axis='columns')
rows_and_columns_dropped = columns_dropped.dropna(how='all')
```

Data Preparation

Dealing with Missing Values / Empty Columns

- Let use the following DataFrame as **Example 1**.

	Pet	Color	Eyes	Length	Weight
0	Cat	Brown	Black	17.49	5.41
1	Dog	Golden	Black	29.01	24.40
2	Dog	Golden	NaN	24.64	21.65
3	Dog	Golden	Brown	21.97	9.25
4	Cat	Black	Green	13.12	NaN
5	Rabbit	White	Blue	13.12	8.67
6	Cat	Gray	Yellow	11.16	11.08
7	Dog	Spotted	Brown	27.32	15.50
8	Rabbit	Yellow	Red	22.02	13.64
9	Dog	Black	Blue	24.16	10.82

```
pets.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Pet     10 non-null     object
 1   Color   10 non-null     object
 2   Eyes    9 non-null      object
 3   Length  10 non-null     float64
 4   Weight  9 non-null      float64
dtypes: float64(2), object(3)
memory usage: 532.0+ bytes
```

```
pets.isnull().sum()
```

```
Pet       0
Color     0
Eyes      1
Length    0
Weight    1
dtype: int64
```

Data Preparation

- **Cleaning by dropping the row**

	Pet	Color	Eyes	Length	Weight
0	Cat	Brown	Black	17.49	5.41
1	Dog	Golden	Black	29.01	24.40
2	Dog	Golden	NaN	24.64	21.65
3	Dog	Golden	Brown	21.97	9.25
4	Cat	Black	Green	13.12	NaN
5	Rabbit	White	Blue	13.12	8.67
6	Cat	Gray	Yellow	11.16	11.08
7	Dog	Spotted	Brown	27.32	15.50
8	Rabbit	Yellow	Red	22.02	13.64
9	Dog	Black	Blue	24.16	10.82

```
pets_cleaned = pets.dropna()
pets_cleaned
```

	Pet	Color	Eyes	Length	Weight
0	Cat	Brown	Black	17.49	5.41
1	Dog	Golden	Black	29.01	24.40
3	Dog	Golden	Brown	21.97	9.25
5	Rabbit	White	Blue	13.12	8.67
6	Cat	Gray	Yellow	11.16	11.08
7	Dog	Spotted	Brown	27.32	15.50
8	Rabbit	Yellow	Red	22.02	13.64
9	Dog	Black	Blue	24.16	10.82

Cleaned DataFrame by dropping rows with null entry

```
pets_cleaned.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8 entries, 0 to 9
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Pet     8 non-null      object
 1   Color   8 non-null      object
 2   Eyes    8 non-null      object
 3   Length  8 non-null      float64
 4   Weight  8 non-null      float64
dtypes: float64(2), object(3)
memory usage: 384.0+ bytes
```

```
pets_cleaned.isnull().sum()
```

```
Pet       0
Color     0
Eyes      0
Length    0
Weight    0
dtype: int64
```

Data Preparation

- Fill missing values with substitutes when there are only a few missing values?

  Fill with mean or median
  Use constant or previous value

```
# Calculate the mean cholestrol value
mean_value = heart_disease_df['chol'].mean()

# Fill missing cholestrol values with the mean
heart_disease_df['chol'].fillna(mean_value, inplace=True)
```

Data Preparation

Dealing with Missing Values / Empty Columns

- Let use the following DataFrame as **Example 2**. Michael Phelps best times in 100m swimming.
- **Cleaning by filling the missing numerical values with mean or median.**

	100m Freestyle	100m Butterfly
0	NaN	49.82
1	48.74	50.48
2	48.78	NaN
3	48.87	50.77
4	48.97	50.86
5	49.05	50.89

For 100m Freestyle let use the mean value to fill the null value.

For 100m Butterfly let use the median value to fill the null value.

	100m Freestyle	100m Butterfly
0	48.88	49.82
1	48.74	50.48
2	48.78	50.77
3	48.87	50.77
4	48.97	50.86
5	49.05	50.89

```
# Fill null values with mean values for '100m Freestyle' and '100m Butterfly'
phelps_100m_df['100m Freestyle'] = phelps_100m_df['100m Freestyle'].fillna(phelps_100m_df['100m Freestyle'].mean())
phelps_100m_df['100m Butterfly'] = phelps_100m_df['100m Butterfly'].fillna(phelps_100m_df['100m Butterfly'].median())
```

Data Preparation

Duplicates

- Refer to identical or nearly identical instances or entries
- Can arise due to various reasons such as data collection errors, system glitches, or intentional replication.
- Identifying and handling duplicates is crucial in data analysis to ensure the accuracy and reliability of the results.

DoctorID	DoctorName
275	Miach
300	Debbie
310	Berry

+

DoctorID	DoctorName
274	Hull
275	Miach
276	Clemency
277	Lydon
278	Chapin
279	Noel

Data Preparation

Dealing with Duplicates

- Drop duplicate rows based on certain columns
- drop_duplicates() function to identify the duplicates based on only certain columns by passing them as a list to the subset argument.

```python
# create a sample dataframe with duplicate rows
data = {
    'Pet': ['Cat', 'Dog', 'Dog', 'Dog', 'Cat'],
    'Color': ['Brown', 'Golden', 'Golden', 'Golden', 'Black'],
    'Eyes': ['Black', 'Black', 'Black', 'Brown', 'Green']
}

df = pd.DataFrame(data)
df
```

	Pet	Color	Eyes
0	Cat	Brown	Black
1	Dog	Golden	Black
2	Dog	Golden	Black
3	Dog	Golden	Brown
4	Cat	Black	Green

```python
df_unique = df.drop_duplicates(subset=['Pet', 'Color'])
df_unique
```

	Pet	Color	Eyes
0	Cat	Brown	Black
1	Dog	Golden	Black
4	Cat	Black	Green

Data Preparation

Outliers

- A data point that is way beyond the other data points in the data set
- It can skew your data which can lead to incorrect inferences.
- Identifying and handling outliers is crucial in maintaining the reliability of our analyses.

Detecting Outliers

- Detecting outliers is crucial in data analysis to ensure accurate and reliable results.
- The Interquartile Range (IQR) is a measure of statistical dispersion, representing the range between the first quartile (Q1) and the third quartile (Q3) of a dataset.
- IQR = Q3 - Q1
- The 1.5 IQR rule is a widely used method to identify outliers.
- It establishes a threshold beyond which data points are considered outliers.

Data Preparation

Outliers Formula:

- Lower Bound: Q1 - 1.5 * IQR
- Upper Bound: Q3 + 1.5 * IQR
- Data points below the lower bound or above the upper bound are flagged as potential outliers.
- The rule assumes a normal distribution and provides a balance between sensitivity and avoiding excessive false positives.

Example:

- If Q1 is 150, Q3 is 200, and IQR is 50, then the lower bound is 75 and the upper bound is 275.
- Any data point below 75 or above 275 would be considered an outlier according to the 1.5 IQR rule.

Data Preparation

Dealing with Duplicates

	Pet	Color	Eyes
0	Cat	Brown	Black
1	Dog	Golden	Black
2	Dog	Golden	Black
3	Dog	Golden	Brown
4	Cat	Black	Green

	Pet	Color	Eyes
0	Cat	Brown	Black
1	Dog	Golden	Black
3	Dog	Golden	Brown
4	Cat	Black	Green

- Data must be clean, concise, and rich
- Redundancies are unhelpful
- Duplicates can bias or confuse model
- Look at unique identifiers as a criteria for dropping records / rows
- Use df.drop_duplicates() to drop duplicate rows
- By default, the drop_duplicates() function identifies the duplicates **taking all the columns into consideration.**

```python
# create a sample dataframe with duplicate rows
data = {
    'Pet': ['Cat', 'Dog', 'Dog', 'Dog', 'Cat'],
    'Color': ['Brown', 'Golden', 'Golden', 'Golden', 'Black'],
    'Eyes': ['Black', 'Black', 'Black', 'Brown', 'Green']
}

df = pd.DataFrame(data)
```

```python
# drop duplicates
df_unique = df.drop_duplicates()
print("\nAfter dropping duplicates:\n")
df_unique
```

Data Preparation

```python
# Given DataFrame
data = {
    'StudentID': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15],
    'Height (cm)': [188, 178, 228, 157, 170, 188, 168, 172, 160, 160, 120, 185, 189, 173, 152]
}

df = pd.DataFrame(data)
df
```

	StudentID	Height (cm)
0	1	188
1	2	178
2	3	228
3	4	157
4	5	170
5	6	188
6	7	168
7	8	172
8	9	160
9	10	160
10	11	120
11	12	185
12	13	189
13	14	173
14	15	152

```python
df.shape
```

```
(15, 2)
```

Data Preparation

```
# Calculate IQR
Q1 = df['Height (cm)'].quantile(0.25)
Q3 = df['Height (cm)'].quantile(0.75)
IQR = Q3 - Q1

# Use the 1.5 IQR formula to filter the DataFrame
filtered_df = df[(df['Height (cm)'] >= (Q1 - 1.5 * IQR)) & (df['Height (cm)'] <= (Q3 + 1.5 * IQR))]

# Display the cleaned dataset without outliers
filtered_df
```

	StudentID	Height (cm)
0	1	188
1	2	178
3	4	157
4	5	170
5	6	188
6	7	168
7	8	172
8	9	160
9	10	160
11	12	185
12	13	189
13	14	173
14	15	152

```
filtered_df.shape
```

```
(13, 2)
```

Data Preparation

- **Imbalances**

Data imbalance is a situation in which the distribution of classes within a dataset is uneven. In simpler terms, some classes have significantly fewer instances than others.

This imbalance can lead to implications for machine learning models, as they will affect the accurately learn and predict the minority classes. It's crucial to address this issue for unbiased and effective model training.

Data Preparation

- **Imbalances**

Imbalanced datasets often lead to model bias. Models tend to favor the majority class, as they are exposed to it more frequently during training. This bias can result in poor predictions for the minority classes.

Traditional metrics like accuracy may be misleading. Introduce precision, recall, F1-score, and AUC-ROC as metrics that provide a more comprehensive evaluation of model performance in imbalanced datasets

Strategies to Address Data Imbalance include using Over Sampling which involves creating copies of instances from the minority class to balance the class distribution. Under Sampling, on the other hand, entails removing instances from the majority class. Both methods aim to create a more balanced dataset.

Data Preparation

- **Imbalances**

Up Sampling the Minority

Down Sampling the Majority

GroupBy and Aggregations

GroupBy and Aggregations

Let consider the following DataFrame, let calculate the Total sales per region and Average sales per salesperson in each region.

```
data = {
    'Region': ['North', 'North', 'South', 'South', 'East', 'East', 'West'],
    'Salesperson': ['Alice', 'Bob', 'Alice', 'Carol', 'Eve', 'Bob', 'Carol'],
    'Product': ['A', 'B', 'A', 'B', 'A', 'B', 'A'],
    'Sales': [200, 150, 300, 350, 400, 250, 500]
}

df = pd.DataFrame(data)
df
```

	Region	Salesperson	Product	Sales
0	North	Alice	A	200
1	North	Bob	B	150
2	South	Alice	A	300
3	South	Carol	B	350
4	East	Eve	A	400
5	East	Bob	B	250
6	West	Carol	A	500

GroupBy and Aggregations

Total sales per region

```python
# Total sales per region
total_sales_per_region = df.groupby('Region')['Sales'].sum()
# Display Results
print("Total Sales Per Region:")
total_sales_per_region
```

```
Total Sales Per Region:
Region
East     650
North    350
South    650
West     500
Name: Sales, dtype: int64
```

```python
# Total sales per region
total_sales_per_region = df.groupby('Region')['Sales'].sum().reset_index()
# Display Results
print("Total Sales Per Region:")
total_sales_per_region
```

Total Sales Per Region:

	Region	Sales
0	East	650
1	North	350
2	South	650
3	West	500

GroupBy and Aggregations

Average sales per salesperson in each region

```
# Average sales per salesperson in each region
avg_sales_per_salesperson = df.groupby(['Region', 'Salesperson'])['Sales'].mean()
# Display Results
print("\nAverage Sales Per Salesperson in Each Region:")
avg_sales_per_salesperson
```

```
Average Sales Per Salesperson in Each Region:

Region  Salesperson
East    Bob            250.0
        Eve            400.0
North   Alice          200.0
        Bob            150.0
South   Alice          300.0
        Carol          350.0
West    Carol          500.0
Name: Sales, dtype: float64
```

GroupBy and Aggregations

Average sales per salesperson in each region

```python
# Average sales per salesperson in each region
avg_sales_per_salesperson = df.groupby(['Region', 'Salesperson'])['Sales'].mean().reset_index()
# Display Results
print("\nAverage Sales Per Salesperson in Each Region:")
avg_sales_per_salesperson
```

```
Average Sales Per Salesperson in Each Region:
```

	Region	Salesperson	Sales
0	East	Bob	250.0
1	East	Eve	400.0
2	North	Alice	200.0
3	North	Bob	150.0
4	South	Alice	300.0
5	South	Carol	350.0
6	West	Carol	500.0

Adding .reset_index() to the aggregation command will convert the result of the GroupBy operation back into a regular DataFrame, with the grouped columns becoming normal columns instead of part of the index.

Reshaping Data with melt()

Reshaping Data with melt()

- The melt() function in pandas DataFrame is used to reshape data by unpivoting it, converting wide-format data into long-format, making it easier to analyze and visualize.

Wide Format

	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

Long Format

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130
3	Mary	Bo	weight	150

Reshaping Data with melt()

Wide Format

	financial	company	2019	2018	2017	2016
0	total_revenue	twitter	3459329	3042359	2443299	2529619
1	gross_profit	twitter	2322288	2077362	1582057	1597379
2	net_income	twitter	1465659	1205596	-108063	-456873
3	total_revenue	facebook	70697000	55838000	40653000	27638000
4	gross_profit	facebook	57927000	46483000	35199000	23849000
5	net_income	facebook	18485000	22112000	15934000	10217000

```
fin_tall_df = fin_df.melt(id_vars=['financial','company'])
fin_tall_df.head(10)
```

Long Format

	financial	company	variable	value
0	total_revenue	twitter	2019	3459329
1	gross_profit	twitter	2019	2322288
2	net_income	twitter	2019	1465659
3	total_revenue	facebook	2019	70697000
4	gross_profit	facebook	2019	57927000
5	net_income	facebook	2019	18485000
6	total_revenue	twitter	2018	3042359
7	gross_profit	twitter	2018	2077362
8	net_income	twitter	2018	1205596
9	total_revenue	facebook	2018	55838000

Reshaping Data with melt()

- **Melting with value_vars**

Wide Format

	financial	company	2019	2018	2017	2016
0	total_revenue	twitter	3459329	3042359	2443299	2529619
1	gross_profit	twitter	2322288	2077362	1582057	1597379
2	net_income	twitter	1465659	1205596	-108063	-456873
3	total_revenue	facebook	70697000	55838000	40653000	27638000
4	gross_profit	facebook	57927000	46483000	35199000	23849000
5	net_income	facebook	18485000	22112000	15934000	10217000

```python
fin_tall_df = fin_df.melt(id_vars=['financial','company'],
value_vars=['2018','2017'])
fin_tall_df.head(9)
```

Long Format

	financial	company	variable	value
0	total_revenue	twitter	2018	3042359
1	gross_profit	twitter	2018	2077362
2	net_income	twitter	2018	1205596
3	total_revenue	facebook	2018	55838000
4	gross_profit	facebook	2018	46483000
5	net_income	facebook	2018	22112000
6	total_revenue	twitter	2017	2443299
7	gross_profit	twitter	2017	1582057
8	net_income	twitter	2017	-108063

Reshaping Data with melt()

- **Melting with column names**

Wide Format

	financial	company	2019	2018	2017	2016
0	total_revenue	twitter	3459329	3042359	2443299	2529619
1	gross_profit	twitter	2322288	2077362	1582057	1597379
2	net_income	twitter	1465659	1205596	-108063	-456873
3	total_revenue	facebook	70697000	55838000	40653000	27638000
4	gross_profit	facebook	57927000	46483000	35199000	23849000
5	net_income	facebook	18485000	22112000	15934000	10217000

```
fin_tall_df = fin_df.melt(id_vars=['financial','company'],
value_vars=['2018','2017'],
var_name=['year'], value_name='dollars')
fin_tall_df.head(8)
```

Long Format

	financial	company	year	dollars
0	total_revenue	twitter	2018	3042359
1	gross_profit	twitter	2018	2077362
2	net_income	twitter	2018	1205596
3	total_revenue	facebook	2018	55838000
4	gross_profit	facebook	2018	46483000
5	net_income	facebook	2018	22112000
6	total_revenue	twitter	2017	2443299
7	gross_profit	twitter	2017	1582057

Reshaping Data with pivot_table()

Reshaping Data with pivot_table()

- The pivot_table() allows you to reshape and summarize data by specifying columns for rows, columns, values, and aggregation functions, providing a flexible way to analyze and present your data.
- Assuming the following DataFrame is known as baseball

	Team	Player	Batting Avg
0	Team 1	W	0.245299
1	Team 2	U	0.329035
2	Team 3	V	0.234873
3	Team 4	K	0.338188
4	Team 5	N	0.277347
5	Team 1	A	0.387346
6	Team 2	T	0.227504
7	Team 3	G	0.268213
8	Team 4	X	0.222695
9	Team 5	L	0.384939

```
baseball.pivot_table(values="Batting Avg", columns="Team", aggfunc=np.sum)
```

Team	Team 1	Team 2	Team 3	Team 4	Team 5
Batting Avg	0.822097	0.549938	0.625926	0.914331	0.836686

```
baseball.pivot_table(values="Batting Avg", columns="Team", aggfunc=np.mean)
```

Team	Team 1	Team 2	Team 3	Team 4	Team 5
Batting Avg	0.411048	0.274969	0.312963	0.457166	0.418343

```
baseball.pivot_table(values="Batting Avg", columns="Team", aggfunc=np.median)
```

Team	Team 1	Team 2	Team 3	Team 4	Team 5
Batting Avg	0.411048	0.274969	0.312963	0.457166	0.418343

Reshaping Data with pivot_table()

- Here's a table listing common aggregation functions used with the pivot_table function in Pandas:

Aggregation Function	Description
'mean'	Computes the mean (average) of values in each group.
'sum'	Calculates the sum of values in each group.
'count'	Counts the number of occurrences in each group.
'min'	Finds the minimum value in each group.
'max'	Finds the maximum value in each group.
'median' or '50%'	Calculates the median (middle value) in each group.
'std'	Computes the standard deviation of values in each group.
'var'	Calculates the variance of values in each group.

Pivot Tables and Crosstab

Pivot Tables and Crosstab

- Pivot Tables and Crosstab functions in Pandas are powerful tools for transforming complex datasets into clear, summarized views, enabling us to analyze and interpret data patterns with ease.
- Key Differences Between Pivot Table and Crosstab

Feature	**Pivot Table**	**Crosstab**
Use Case	Flexible data summarization with aggregation.	Typically used for frequency counts or specific aggregations.
Syntax Complexity	Requires more parameters but is highly flexible.	Simpler syntax for basic operations.
Missing Values Handling	Can handle missing values using fill_value.	Doesn't directly handle missing values.

Pivot Tables and Crosstab

- Let's work with a dataset that tracks sales of different products across regions and salespeople.

```
data = {
    'Region': ['North', 'North', 'South', 'South', 'East', 'East', 'West', 'West'],
    'Product': ['A', 'B', 'A', 'B', 'A', 'B', 'A', 'B'],
    'Salesperson': ['Alice', 'Bob', 'Alice', 'Carol', 'Eve', 'Bob', 'Carol', 'Eve'],
    'Sales': [200, 150, 300, 350, 400, 250, 500, 450]
}

df = pd.DataFrame(data)
df
```

	Region	Product	Salesperson	Sales
0	North	A	Alice	200
1	North	B	Bob	150
2	South	A	Alice	300
3	South	B	Carol	350
4	East	A	Eve	400
5	East	B	Bob	250
6	West	A	Carol	500
7	West	B	Eve	450

Pivot Tables and Crosstab

- The pivot_table method allows us to summarize data flexibly by defining rows, columns, and the aggregation function.

```python
# Pivot Table: Total sales per region for each product
pivot_table = df.pivot_table(
    index='Region',        # Rows
    columns='Product',     # Columns
    values='Sales',        # Values to aggregate
    aggfunc='sum',         # Aggregation function
    fill_value=0           # Fill missing values with 0
)

print("\nPivot Table - Total Sales Per Region for Each Product:")
pivot_table
```

```
Pivot Table - Total Sales Per Region for Each Product:
```

Product	A	B
Region		
East	400	250
North	200	150
South	300	350
West	500	450

Pivot Tables and Crosstab

- The crosstab function is useful for frequency counts or aggregations on combinations of rows and columns.

```python
# Crosstab: Frequency count of salespeople selling each product
crosstab = pd.crosstab(
    df['Salesperson'],      # Rows
    df['Product'],          # Columns
    values=df['Sales'],     # Values to aggregate
    aggfunc='sum',          # Aggregation function
    dropna=False,           # Keep all categories, even if empty
)

print("\nCrosstab - Total Sales Per Salesperson for Each Product:")
crosstab
```

```
Crosstab - Total Sales Per Salesperson for Each Product:
```

Product	A	B
Salesperson		
Alice	500.0	NaN
Bob	NaN	400.0
Carol	500.0	350.0
Eve	400.0	450.0

Pivot Tables and Crosstab

- For the previous example, let use Pivot Tables instead of Crosstab. Let compare the 2 outputs.

```python
# Pivot Table: Total sales per Salesperson for each product
pivot_table = df.pivot_table(
    index='Salesperson',        # Rows
    columns='Product',     # Columns
    values='Sales',        # Values to aggregate
    aggfunc='sum',         # Aggregation function
    fill_value=0           # Fill missing values with 0
)

print("\nPivot Table - Total Sales Per Salesperson for Each Product:")
pivot_table
```

```
Pivot Table - Total Sales Per Salesperson for Each Product:
```

Product	A	B
Salesperson		
Alice	500	0
Bob	0	400
Carol	500	350
Eve	400	450

Correlation

Correlation

- Describes direction and strength of relationship between two variables
- Can help us use variables to predict future outcomes
- corr() calculates Pearson correlation coefficient, measuring linear relationship

Using the following DataFrame with the following columns:

- Duration: Exercise session duration in minutes.
- Pulse: Average heart rate during the exercise session (bpm).
- Maxpulse: Maximum heart rate recorded during the exercise session (bpm).
- Calories: Estimated calories burned during the exercise session.

This dataset seems to capture information about exercise sessions, providing insights into the duration, intensity, and estimated calorie expenditure for each session.

Correlation

- Describes direction and strength of relationship between two variables
- Can help us use variables to predict future outcomes
- corr() calculates Pearson correlation coefficient, measuring linear relationship

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0
...	...	...	...	...
164	60	105	140	290.8
165	60	110	145	300.0
166	60	115	145	310.2
167	75	120	150	320.4
168	75	125	150	330.4

169 rows × 4 columns

Using the following DataFrame with the following columns:

- Duration: Exercise session duration in minutes.
- Pulse: Average heart rate during the exercise session (bpm).
- Maxpulse: Maximum heart rate recorded during the exercise session (bpm).
- Calories: Estimated calories burned during the exercise session.

This dataset seems to capture information about exercise sessions, providing insights into the duration, intensity, and estimated calorie expenditure for each session.

Correlation

- The Result of the corr() method is a table with a lot of numbers that represents how well the relationship is between two columns. The number varies from -1 to 1.
- 1 means that there is a 1 to 1 relationship (a perfect correlation), and for this data set, each time a value went up in the first column, the other one went up as well.
- 0.9 is also a good relationship, and if you increase one value, the other will probably increase as well.
- -0.9 is as good relationship as 0.9, but if you increase one value, the other will probably go down.
- 0.2 means NOT a good relationship, meaning that if one value goes up does not mean that the other will.

```
df.corr()
```

	Duration	Pulse	Maxpulse	Calories
Duration	1.000000	-0.155408	0.009403	0.922717
Pulse	-0.155408	1.000000	0.786535	0.025121
Maxpulse	0.009403	0.786535	1.000000	0.203813
Calories	0.922717	0.025121	0.203813	1.000000

Correlation

Duration and Calories:

There is a strong positive correlation of approximately 0.92 between 'Duration' and 'Calories'. This suggests that as the duration of the exercise session increases, the number of calories burned also tends to increase. This is an intuitive and expected relationship.

Pulse and Calories:

The correlation between 'Pulse' and 'Calories' is relatively low at 0.025. This indicates a weak linear relationship between the average heart rate during the exercise session ('Pulse') and the number of calories burned ('Calories'). The weak correlation suggests that other factors may influence calorie burn.

Correlation

Maxpulse and Calories:

There is a moderate positive correlation of approximately 0.20 between 'Maxpulse' and 'Calories'. This suggests that as the maximum heart rate recorded during the exercise session increases, the number of calories burned also tends to increase, though the correlation is not as strong as with 'Duration'.

Pulse and Maxpulse:

There is a strong positive correlation of approximately 0.79 between 'Pulse' and 'Maxpulse'. This indicates a strong linear relationship between the average heart rate and the maximum heart rate recorded during the exercise session. This relationship is expected, as one would generally expect the average and maximum heart rates to be correlated.

Strong positive correlations indicate a positive linear relationship, while weak correlations suggest a weaker or no linear relationship.

Thank You!

NANYANG
THE INNOVATIVE POLYTECHNIC

www.nyp.edu.sg