Tuần 1 - Tổng hợp kiến thức Buổi học số 1 và 2

Time-Series Team

12/08/2025

Buổi học số 1 (Thứ 3, 29/07/2025) và buổi học số 2 (Thứ 4, 30/07/2025) có nhiều nội dung tương đồng và kế thừa nhau nên nhóm mình sẽ tổng hợp lại thành 4 nội dung chính:

- ***Phần 1: Giới thiệu***
- ***Phần 2: Nhắc lại kiến thức cơ bản (Recap)***
- ***Phần 3: Các kỹ thuật nâng cao trong phân tích dữ liệu với Pandas***
- ***Phần 4: Kỹ thuật nâng cao trong trực quan hóa dữ liệu***

Phần 1: Giới thiệu

Từ Bài Toán Phân Loại Spam đến Tầm Quan Trọng của EDA và Giới Thiệu Pandas

Trong Module 2 chúng ta đã làm quen với dự án phân loại email Ham/Spam sử dụng Naive Bayes và cơ sở dữ liệu vector, một minh hoạ điển hình cho bài toán phân loại văn bản (text classification) trong Machine Learning. Dự án này sử dụng cả phương pháp truyền thống (Naive Bayes) lẫn cách tiếp cận hiện đại (đại diện tin nhắn dưới dạng vector nhúng và tra cứu trong cơ sở dữ liệu vector), nhằm xác định một tin nhắn có phải spam hay không. Thực tế, Naive Bayes là thuật toán thông dụng được sử dụng trong lọc thư rác (spam filtering). Phân loại spam/ham chính là một ví dụ tiêu biểu của phân loại văn bản, giúp người dùng tự động nhận biết và loại bỏ những nội dung không mong muốn.

Vấn đề dữ liệu ban đầu

Nếu bỏ qua bước xử lý và phân tích dữ liệu ban đầu, ta sẽ gặp phải nhiều vấn đề tiềm ẩn trong bộ dữ liệu thu thập được. Các khó khăn điển hình gồm có:

- **Dữ liệu mất cân bằng:** Tỉ lệ tin nhắn spam và ham thường không đều, khiến mô hình dễ thiên vị về lớp chiếm ưu thế. Ví dụ, nếu 90% tin nhắn là ham và chỉ 10% là spam, mô hình có thể đoán tất cả tin nhắn là ham mà vẫn đạt độ chính xác cao, nhưng lại bỏ sót hầu hết spam.
- **Dữ liệu nhiễu và không nhất quán:** Tin nhắn spam thường chứa các ký tự đặc biệt, lỗi chính tả hoặc cách viết lạ (như viết chữ in hoa sai quy tắc, dùng ký tự thay thế), gây khó khăn cho mô hình khi trích xuất đặc trưng. Ngoài ra, spammer có thể thay "free" thành "fr€e", "click" thành "cl1ck" để đánh lừa bộ lọc.
- **Bản ghi trùng lặp và dư thừa:** Trong bộ dữ liệu thu thập, có thể xuất hiện nhiều tin nhắn giống hệt nhau (trùng lặp) hoặc thông tin thừa thãi. Những bản ghi trùng làm sai lệch phân phối dữ liệu, tốn thêm bộ nhớ và làm giảm hiệu quả huấn luyện.
- **Đặc trưng chưa rõ nghĩa:** Dữ liệu văn bản thô chưa được phân loại, biến đổi thành đặc trưng hữu ích nên chúng ta chưa hiểu rõ dữ liệu đang chứa thông tin gì. Nếu không nắm được ý nghĩa các trường dữ liệu (ví dụ, các cột chứa chỉ số, danh mục chưa giải mã), sẽ rất khó để chọn cách xử lý và trích xuất đặc trưng phù hợp.

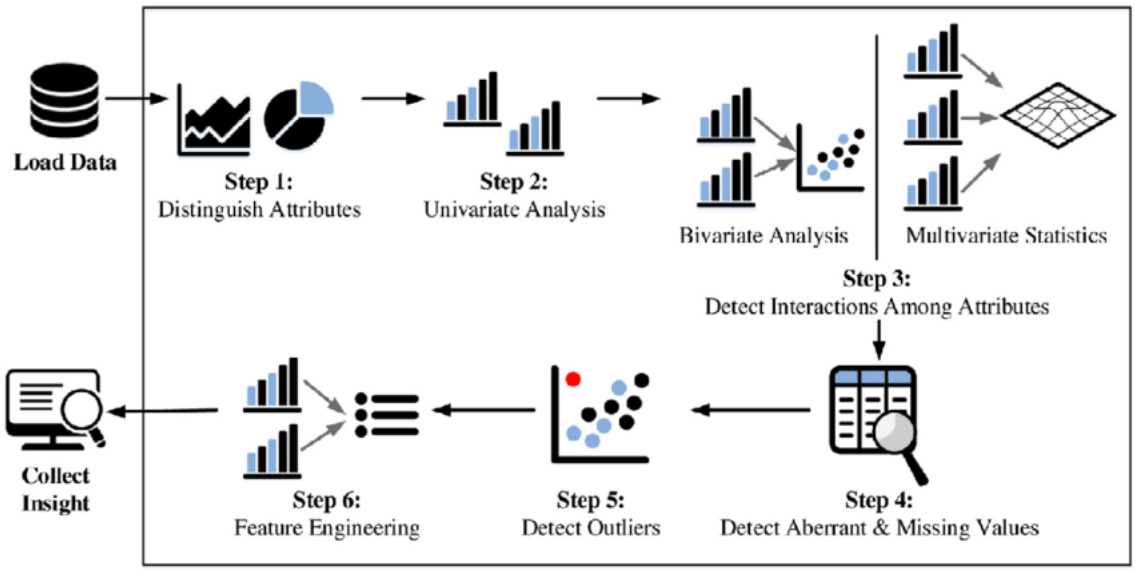
Ví dụ: Giả sử một tin nhắn spam viết: "Hurry up! fr€e c1cK đừng bỏ lỡ". Việc sử dụng ký tự lạ ("fr€e", "c1cK") có thể khiến bộ lọc spam dựa trên từ khoá khó nhận ra đây là spam. Ngược lại, một tin nhắn bình thường như "Gặp bạn lúc 3 giờ tại quán" có thể bị dự đoán sai thành spam nếu mô hình chưa được xử lý ngữ cảnh đầy đủ. Những vấn đề trên cho thấy việc kiểm tra và làm sạch dữ liệu (trước khi xây dựng mô hình) là rất quan trọng.

Khái niệm Khám phá Dữ liệu (EDA)

Để giải quyết các vấn đề dữ liệu trên, bước Khám phá Dữ liệu (Exploratory Data Analysis, viết tắt EDA) trở nên cần thiết. EDA là công đoạn đầu tiên giúp chúng ta hiểu rõ bộ dữ liệu trước khi xây dựng mô hình học máy. Theo tài liệu **Machine Learning cơ bản** của tác giả **Vũ Hữu Tiệp**,

> "Bước EDA này giúp chúng ta có cái nhìn đầu tiên về dữ liệu. Bạn cần có một cảm giác nhất định về những gì mình có trong tay trước khi có những chiến lược xây dựng mô hình".

Cũng vì vậy, EDA được xem là một bước quan trọng không thể thiếu trong pipeline ML.



Minh họa các bước EDA

Trong quá trình EDA, chúng ta thường thực hiện các công việc chính sau:

- **Kiểm tra phân phối dữ liệu:** Xem số lượng mẫu (hàng), số trường (cột), phân bố nhãn (spam/ham).

```
df['label'].value_counts()

```

- **Xử lý giá trị thiếu (NaN):**

```
df.isnull().sum()

```

- **Loại bỏ bản sao (trùng lặp):**

```
df.drop_duplicates()

```

- **Chuẩn hóa và làm sạch dữ liệu:** Ví dụ chuẩn hoá chữ hoa thành thường, loại bỏ ký tự đặc biệt ("fr€e" → "free"), áp dụng stemming/lemmatization.
- **Phát hiện điểm bất thường (outliers):** Dùng biểu đồ, boxplot, Z-score.

Những bước EDA trên giúp hiểu về độ phức tạp của bài toán và xác định bước xử lý tiếp theo. EDA không chỉ thực hiện một lần mà nên lặp lại xuyên suốt quá trình phát triển: trước khi trích xuất đặc trưng, sau khi tạo đặc trưng, thậm chí sau khi huấn luyện để đảm bảo dữ liệu đầu vào đã sạch.

Thư viện Pandas

Trong Python, một trong những công cụ đắc lực nhất hỗ trợ EDA là thư viện Pandas. Theo tài liệu **Description** của **TA Thái**,

"Pandas là một thư viện trong Python với ưu điểm là nhanh, mạnh, linh động, dễ sử dụng, mã nguồn mở, dùng để phân tích và thao tác dữ liệu."

Pandas được xây dựng trên NumPy, cung cấp nhiều hàm tiện ích giúp làm sạch, phân tích, mô hình hoá dữ liệu – đặc biệt hiệu quả với dữ liệu bảng (SQL table hoặc Excel spreadsheet).
Với Pandas, chúng ta có thể:

- Đọc/ghi nhiều định dạng dữ liệu (CSV, Excel, JSON, SQL, …).
- Xử lý giá trị thiếu: `fillna()`, `dropna()`.
- Loại bỏ trùng lặp: `drop_duplicates()`.
- Lọc, truy vấn, nhóm dữ liệu: `groupby`, `mean`, `sum`, `count`.
- Tốc độ cao, tích hợp tốt với NumPy, Matplotlib, Seaborn.

Ví dụ:

```python
import pandas as pd
df = pd.read_csv('emails.csv')
print(df['label'].value_counts())
```

```
==================  Output ==================

ham     4825
spam     747
Name: label, dtype: int64
```

Đoạn code trên đọc file `emails.csv` vào DataFrame và in ra số lượng nhãn mỗi loại (spam/ham). Tương tự có thể dùng:

```python
df.drop_duplicates(inplace=True)
```

để loại bỏ giá trị thiếu và

```python
df.dropna(inplace=True)
```

để xoá bản ghi trùng.

Như vậy, Pandas là nền tảng mạnh mẽ cho mọi công đoạn tiền xử lý và khám phá dữ liệu trong Pipeline của Machine Learning. Việc học Pandas giúp thực hiện các bước EDA hiệu quả, đảm bảo dữ liệu sạch, chất lượng trước khi đưa vào mô hình.

Phần 2: Các kiến thức cơ bản về Pandas

Pandas là thư viện cung cấp hai cấu trúc dữ liệu chính: Series và DataFrame, trong đó:

- **Series** là mảng 1 chiều có nhãn, bao gồm một giá trị và một chỉ mục (index) cho mỗi phần tử. DataFrame là bảng dữ liệu 2 chiều, được tạo từ nhiều Series ghép theo cột, với nhãn dòng (index) và nhãn cột (column).
- **DataFrame** hỗ trợ nhiều phép toán như lọc điều kiện, thống kê, nhóm (groupby), nối bảng... để phân tích dữ liệu.

1. Series Pandas

- **Tạo Series:** Dùng `pd.Series([...])`. Ví dụ khởi tạo Series với chỉ số mặc định (0,1,...):

```python
import pandas as pd
s = pd.Series([3, -5, 7, 4])
print(s)
```

```
==================  Output ==================

0    3
1   -5
2    7
3    4
dtype: int64
```

Mặc định Pandas tự tạo index dạng số nguyên. Ta cũng có thể tùy chỉnh index thủ công:

```python
s = pd.Series([3, -5, 7, 4], index=['a',
    'b', 'c', 'd'])
print(s)
```

```
==================  Output ==================

a    3
b   -5
c    7
d    4
dtype: int64
```

Với index thủ công, ta có thể truy xuất dữ liệu bằng tên nhãn thay vì vị trí số.

- **Truy cập phần tử:** Có thể lấy giá trị bằng vị trí hoặc nhãn. Ví dụ:

```python
print(s[2])  # Lay phan tu thu 2 (vi tri thu 2): 7
print(s['b'])  # Lay phan tu nhan 'b': -5
```

Trong Pandas, Series hoạt động giống từ điển có gán nhãn, nên có thể dùng cú pháp `s['nhãn']` để lấy giá trị.

- **Hàm tính toán cơ bản:** Series hỗ trợ các hàm thống kê như `min()`, `max()`, `sum()`, `mean()`, `std()` ... Ví dụ với Series số:

```python
data = pd.Series([1, 7, 5, 6, 5])
print("min:", data.min())   # 1
print("max:", data.max())   # 7
print("sum:", data.sum())   # 24
print("mean:", data.mean()) # 4.8
print("std:", data.std())   # ~2.28 (
    standard deviation)
```

```
==================  Output ==================
min: 1
max: 7
sum: 24
mean: 4.8
std: 2.280350850198276
```

Ngoài ra, `idxmax()` và `argmax()` trả về vị trí hoặc nhãn của phần tử lớn nhất trong Series.

- **Thao tác với chuỗi:** Nếu Series chứa dữ liệu chuỗi (string), ta có thể dùng thuộc tính `.str` để thao tác. Ví dụ:

```python
langs = pd.Series(['Java', 'C++', 'Python
    ', 'Java', 'C#'])
print(langs.str.count('a'))
print(langs.str.upper())
print(langs.replace('Java', 'C#'))
```

```
==================  Output ==================
0    1
1    0
2    0
3    1
4    0
dtype: int64

0      JAVA
1       C++
2    PYTHON
3      JAVA
4        C#
dtype: object

0      C#
1     C++
2    Python
3      C#
4       C#
dtype: object
```

  - `str.count('a')` đếm số lần xuất hiện của chữ "a" trong mỗi chuỗi.
  - `str.upper()` chuyển mỗi chuỗi thành chữ hoa. `replace('Java','C#')` thay thế tất cả phần tử `'Java'` thành `'C#'`.

- **Phép toán giữa các Series:** Hai Series có thể được cộng/trừ nhân chập tự động dựa trên nhãn. Ví dụ:

```python
s1 = pd.Series([10, 20, 30], index=['a','
    b','c'])
s2 = pd.Series([1, 2, 3], index=['a','b',
    'c'])
print(s1 + s2)
```

```
==================  Output ==================
a    11
b    22
c    33
dtype: int64
```

Trong trường hợp index khác nhau, Pandas sẽ căn chỉnh theo nhãn và những giá trị không khớp sẽ thành `NaN`.

2. DataFrame Pandas

- **Tạo DataFrame:** DataFrame là bảng 2D với các cột có thể khác kiểu dữ liệu. Ví dụ tạo DataFrame từ dict:

```
data = {'Name': ['Anh', 'Thai', 'Hoa'],
        'Country': ['Brazil', 'Vietnam', 'Vietnam']}
df = pd.DataFrame(data, columns=['Name', 'Country'])
print(df)
```

```
================== Output ==================
Name  Country
0   Anh   Brazil
1  Thai  Vietnam
2   Hoa  Vietnam
```

(Mỗi cột của DataFrame là một Series, có thể là `int64`, `float64`, `object` (chuỗi), v.v.)

- **Truy xuất dữ liệu:**
  - **Lấy cột:** `df['Name']` hoặc `df.Name` trả về một Series.
  - **Lấy nhiều cột:** `df[['Name','Country']]` trả về DataFrame con.
  - **Lấy dòng (bằng index mặc định):** có thể dùng `df.iloc[i]` hoặc `df.iloc[i:j]` để lấy theo chỉ số vị trí, hoặc `df.loc[label]` nếu index của bảng là nhãn định sẵn. Ví dụ:

```
print(df['Name'])
print(df.iloc[0])   # dong dau tien
```

```
================== Output ==================
0     Anh
1    Thai
2     Hoa
Name: Name, dtype: object

Name           Anh
Country     Brazil
Name: 0, dtype: object
```

Ta cũng có thể dùng slicing: `df[1:3]` lấy dòng 1 đến 2.

- **Lọc điều kiện:** Dùng biểu thức điều kiện trên DataFrame. Ví dụ: giả sử DataFrame về Pokémon (xem phần 2) có cột `'HP'`, ta lọc các Pokémon có HP > 50:

```
df_poke = pd.DataFrame({'Name':['A','B','C'], 'HP':[30, 60, 45]})
print(df_poke[df_poke['HP'] > 50])
```

```
================== Output ==================
Name  HP
1    B   60
```

Kết quả:chỉ giữ các hàng thỏa mãn điều kiện. Đây là cách chọn nhanh các bản ghi theo điều kiện.

- **Nhóm dữ liệu (GroupBy):** Dùng `df.groupby()` để phân nhóm rồi tính toán thống kê cho mỗi nhóm. Ví dụ: nhóm theo cột "Type 1" và tính tổng "Attack":

```
df = pd.DataFrame({'Type1':['Fire','Grass','Fire','Water'],
                   'Attack':[52,49,60,48]})
print(df.groupby('Type1')['Attack'].sum())
```

```
================== Output ==================
Type1
Fire     112
Grass     49
Water     48
Name: Attack, dtype: int64
```

Tương tự có thể dùng `.mean()`, `.count()`, `.max()`, v.v. để lấy trung bình, đếm, giá trị lớn nhất của từng nhóm.

- **Sắp xếp (`sort_values`):** Dùng `df.sort_values('cot', ascending=True)` hoặc `df.sort_values('cot', ascending=False)` để sắp xếp. Ví dụ: sắp xếp DataFrame Pokémon theo cột `"Speed"`:

```
df = pd.DataFrame({'Name':['X','Y','Z'],'Speed':[90,45,100]})
print(df.sort_values('Speed', ascending=False))
```

```
================== Output ==================
Name  Speed
2    Z    100
0    X     90
1    Y     45
```

Cú pháp `df.sort_values()` cho phép sắp xếp tăng/giảm dần hoặc theo nhiều cột.

- **Thao tác chỉ mục:** Có thể dùng `df.set_index('cột')` để biến cột thành index mới, hoặc `df.reset_index()` để hoàn nguyên. Ví dụ:

```
df2 = df.set_index('Name')
print(df2)
print(df2.loc['Anh'])   # truy xuat bang nhan index
df3 = df2.reset_index()
```

```
================== Output ==================
Country
Name
Anh     Brazil
Thai   Vietnam
Hoa    Vietnam

Country    Brazil
Name: Anh, dtype: object
```

- **Thêm/Xóa cột:** Dễ dàng thêm cột mới bằng cách gán. Ví dụ:
  - Thêm cột:

```
df = pd.DataFrame({'A':[1,2], 'B':[3,4]})
df['C'] = df['A'] + df['B']   # them cot C
print(df)
```

```
================== Output ==================
A  B  C
0  1  3  4
1  2  4  6
```

  - Để xóa cột, dùng `df.drop('col', axis=1)`. Ví dụ:

```
df2 = df.drop('C', axis=1)
print(df2.columns)
```

```
================== Output ==================
Index(['A', 'B'], dtype='object')
```

3. Thực hành với bộ dữ liệu Pokémon

Bộ dữ liệu Pokémon (CSV) chứa thông tin về các Pokémon với các thuộc tính như `Name`, `Type 1`, `Type 2`, `HP`, `Attack`, `Defense`, `Sp. Atk`, `Sp. Def`, `Speed`, `Generation`, `Legendary`, `Total`. Dưới đây là các bước thực hành cơ bản:

Đọc file CSV:

```
import pandas as pd
df = pd.read_csv("Pokemon.csv")
print(df.head(3))
```

Giả sử dữ liệu đã load, `df.head(3)` hiển thị 3 dòng đầu:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1	False

Khám phá dữ liệu:

Dùng `df.describe()`, `df.tail()`, v.v. để nắm thông tin tổng quan. Ví dụ:

```
print(df.describe())  # Thong ke mo ta cho cac cot so
```

	#	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
count	800.000000	800.00000	800.000000	800.000000	800.000000	800.000000	800.000000	800.000000	800.00000
mean	362.813750	435.10250	69.258750	79.001250	73.842500	72.820000	71.902500	68.277500	3.32375
std	208.343798	119.96304	25.534669	32.457366	31.183501	32.722294	27.828916	29.060474	1.66129
min	1.000000	180.00000	1.000000	5.000000	5.000000	10.000000	20.000000	5.000000	1.00000
25%	184.750000	330.00000	50.000000	55.000000	50.000000	49.750000	50.000000	45.000000	2.00000
50%	364.500000	450.00000	65.000000	75.000000	70.000000	65.000000	70.000000	65.000000	3.00000
75%	539.250000	515.00000	80.000000	100.000000	90.000000	95.000000	90.000000	90.000000	5.00000
max	721.000000	780.00000	255.000000	190.000000	230.000000	194.000000	230.000000	180.000000	6.00000

Hàm `describe()` cho biết các giá trị trung bình, min, max… của các cột số.

Truy xuất cột:

Lấy các cột cụ thể bằng ký hiệu `df['col']`. Ví dụ:

```
print(df['Name'])                    # Cot Name (Series)
print(df[['Name','HP','Attack']]) # Nhieu cot
```

```
==================  Output  ==================
0        Pikachu
1      Bulbasaur
2     Charmander
3       Squirtle
4            Mew
Name: Name, dtype: object

         Name   HP  Attack
0     Pikachu   35      55
1   Bulbasaur   45      49
2  Charmander   39      52
3    Squirtle   44      48
4         Mew  100     100
```

Ta cũng có thể chọn cột bằng `df.Type1` (nếu tên cột hợp lệ) hoặc chọn theo chỉ số cột bằng `df.iloc[:, [0, 3, 5]]`.

Lọc dữ liệu:

Sử dụng điều kiện boolean. Ví dụ, lấy các Pokémon có tổng điểm (`Total`) lớn hơn 300:

```
strong = df[df['Total'] > 300]
print(strong[['Name','Total']])
```

```
==================  Output  ==================
Name  Total
0  Pikachu    320
1  Bulbasaur  318
2  Charmander 309
3   Squirtle  314
4         Mew  600
```

Chú ý

Mew là `Legendary` nên `Total` rất cao. Lọc theo điều kiện cho phép xem nhanh các bản ghi thỏa mãn yêu cầu.

Sắp xếp dữ liệu:

Dùng `df.sort_values()`. Ví dụ, sắp xếp bảng theo cột `Speed` giảm dần:

```
sorted_df = df.sort_values('Speed', ascending=False)
print(sorted_df[['Name','Speed']])
```

```
==================  Output  ==================
Name  Speed
4      Mew    100
0  Pikachu     90
2  Charmander   65
1  Bulbasaur    45
3  Squirtle     43
```

Ví dụ trên cho thấy Mew có `Speed` cao nhất. Cú pháp `df.sort_values('col', ascending=False)` sắp xếp theo cột chỉ định.

Thêm và xóa cột:

- Thêm cột mới, ví dụ tính tổng chỉ số tấn công và phòng thủ:

```
df['Attack_Defense'] = df['Attack'] + df['Defense']
print(df[['Name','Attack','Defense','Attack_Defense']])
```

```
================== Output ==================
Name  Attack  Defense  Attack_Defense
0   Pikachu       55        40
     95
1  Bulbasaur       49        49
      98
2 Charmander       52        43
      95
3   Squirtle       48        65
     113
4        Mew      100       100
     200
```

- Xóa cột mới tạo:

```
df = df.drop('Attack_Defense', axis=1)
print(df.columns)
```

```
================== Output ==================
Index(['Name', 'Type 1', 'Type 2', 'HP',
    'Attack', 'Defense', 'Sp. Atk',
       'Sp. Def', 'Speed', 'Generation',
    'Legendary', 'Total'],
      dtype='object')
```

Cột `Attack_Defense` đã được loại bỏ, chỉ còn các cột ban đầu.

Nhóm dữ liệu và tính tổng điểm:

Giờ ta nhóm các Pokémon theo loại chính `Type 1` và tính tổng `Total` của mỗi nhóm:

```
print(df.groupby('Type 1')['Total'].sum())
```

```
================== Output ==================
Type 1
Electric    320
Fire        309
Grass       318
Psychic     600
Water       314
Name: Total, dtype: int64
```

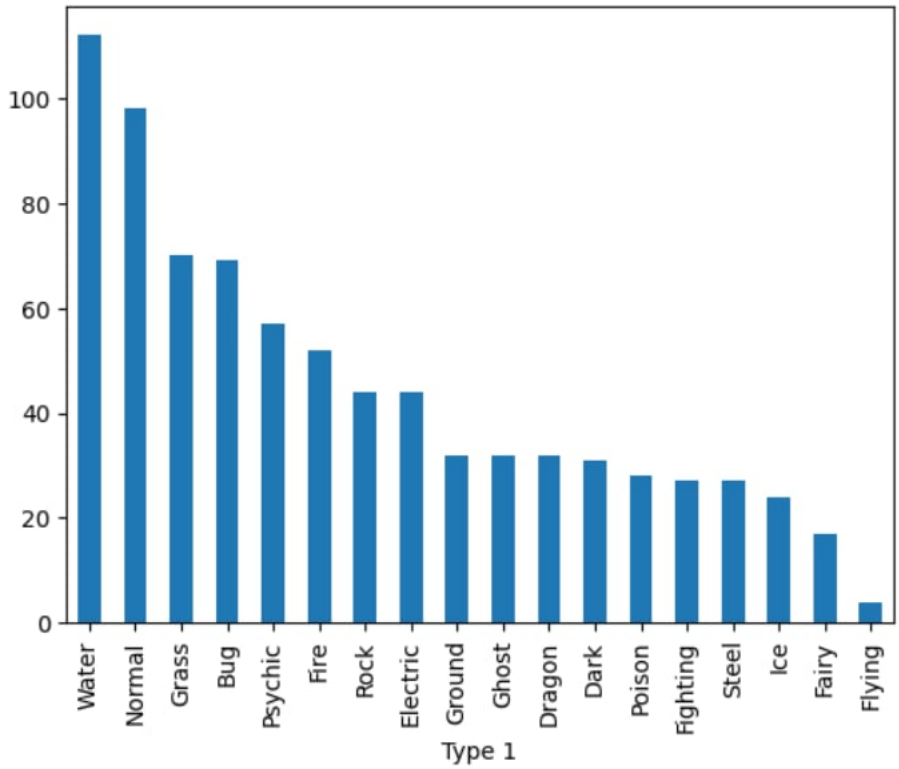
Trong đó Mew thuộc Psychic với Total=600. Tính tổng theo nhóm giúp so sánh sức mạnh tổng quát của từng loại Pokémon. Việc dùng `groupby` và các hàm như `sum()` là ví dụ điển hình của kỹ thuật GroupBy trong Pandas.

Trực quan hóa đơn giản:

Pandas tích hợp sẵn khả năng vẽ đồ thị từ DataFrame. Ví dụ, biểu diễn số lượng Pokémon theo mỗi `Type 1`:

```
df['Type 1'].value_counts().plot(kind='bar')
```

Đoạn code trên sử dụng `value_counts()` để đếm mỗi loại và vẽ biểu đồ cột (bar chart). Kết quả là hình cột thể hiện số lượng Pokémon thuộc mỗi type, giúp so sánh trực quan (ví dụ Electric, Grass, Fire, Water, Psychic… mỗi loại có bao nhiêu con).



Tóm lại, phần thực hành với bộ dữ liệu Pokémon minh họa các bước cơ bản của công đoạn làm việc với Pandas: đọc dữ liệu, khám phá (`describe, head, tail`), lấy cột, lọc điều kiện, sắp xếp, thêm/xóa cột, nhóm và tính toán (groupby, sum) và trực quan hóa. Các hàm `head()`, `tail()`, `describe()`… là những công cụ nhanh để hiểu sơ bộ dữ liệu. Việc thực hiện tuần tự các bước này cho phép sinh viên thực hành Pandas một cách rõ ràng, dễ tiếp cận và chuẩn bị tốt cho các bước phân tích tiếp theo.

Phần 3: Các kỹ thuật nâng cao trong phân tích dữ liệu với Pandas

1. Các kỹ thuật nâng cao trong phân tích dữ liệu với Pandas

1.1. Vấn đề thường gặp khi thao tác với dataset

Trong lý thuyết hoặc các ví dụ minh họa nhỏ, dữ liệu thường được trình bày dưới dạng "sạch sẽ" và hoàn chỉnh. Tuy nhiên, khi bước ra các bài toán thực tế, đặc biệt là trong lĩnh vực phân tích dữ liệu và học máy, chúng ta gần như luôn phải đối mặt với những khó khăn phát sinh từ chất lượng dữ liệu. Nếu không xử lý cẩn thận, các vấn đề này có thể làm sai lệch kết quả phân tích, dẫn đến những kết luận thiếu chính xác hoặc thậm chí phản tác dụng.

Một số vấn đề phổ biến nhất thường gặp khi thao tác với dataset là:

- **Thiếu dữ liệu (Missing Data):** xuất hiện khi một số quan sát không được ghi nhận, dẫn đến các ô bị bỏ trống hoặc chứa giá trị `NaN/null`. Ví dụ, trong dữ liệu y tế, không phải bệnh nhân nào cũng cung cấp đủ tất cả thông tin cần thiết.
- **Dữ liệu nhiễu (Noise):** bao gồm các giá trị bất thường (outlier) hoặc sai số do thiết bị đo lường, nhập liệu thủ công. Những điểm dữ liệu này nếu không được phát hiện và xử lý có thể làm méo mó phân phối và ảnh hưởng tiêu cực đến mô hình dự đoán.
- **Dữ liệu không đồng nhất:** các cột có thể chứa cả chuỗi lẫn số, định dạng ngày tháng không thống nhất (ví dụ `dd-mm-yyyy` và `yyyy/mm/dd` cùng xuất hiện). Điều này khiến việc phân tích hoặc trực quan hóa gặp khó khăn.
- **Dữ liệu phân bố theo thời gian (Time-dependent data):** với những bộ dữ liệu dạng chuỗi thời gian, yếu tố thứ tự xuất hiện là cực kỳ quan trọng. Không thể hoán đổi các quan sát một cách tùy ý, bởi vì thông tin về xu hướng, mùa vụ hoặc sự kiện bất thường đều phụ thuộc vào thời điểm ghi nhận.

Việc nhận diện và hiểu rõ những vấn đề này là bước đầu tiên quan trọng trước khi tiến hành các kỹ thuật nâng cao với Pandas.

1.2. Xử lý dữ liệu thiếu (Missing Data)

Loại bỏ dữ liệu thiếu

Khi thao tác với dữ liệu thực tế, một trong những vấn đề đầu tiên mà nhà phân tích thường gặp phải là sự xuất hiện của các giá trị bị thiếu (`NaN`). Cách xử lý đơn giản và trực quan nhất chính là loại bỏ hoàn toàn các quan sát hoặc thuộc tính chứa giá trị thiếu.

```python
import pandas as pd

# Tao DataFrame don gian có gia tri thieu
df = pd.DataFrame({
    "A":[1, 2, None, 4],
    "B":[5, None, 7, 8]
})

# Loai bo toan bo dong co gia tri NaN
df_clean = df.dropna()
print(df_clean)
```

```
==================  Output ==================
A    B
0  1.0  5.0
3  4.0  8.0
```

Kết quả cho thấy chỉ còn lại những dòng hoàn toàn không có giá trị bị thiếu (trong ví dụ này là dòng 0 và dòng 3).

Phương pháp `dropna()` đặc biệt hữu ích khi:

- Số lượng bản ghi rất lớn, việc mất đi một vài dòng không ảnh hưởng đáng kể đến phân tích.
- Các cột chứa giá trị thiếu không thực sự quan trọng đối với bài toán.

Tuy nhiên, cách tiếp cận này cũng có nhiều hạn chế:

- **Mất mát thông tin:** nếu tỉ lệ dữ liệu bị thiếu cao, việc loại bỏ có thể làm dataset bị thu hẹp nghiêm trọng, thậm chí gây mất cân bằng dữ liệu.
- **Sai lệch mẫu:** nếu dữ liệu bị thiếu không ngẫu nhiên (ví dụ: nhóm khách hàng có thu nhập cao thường bỏ trống thông tin độ tuổi), thì việc loại bỏ có thể làm sai lệch phân phối và dẫn đến thiên lệch trong phân tích.

Vì vậy, **loại bỏ dữ liệu thiếu chỉ nên coi là lựa chọn cuối cùng** trong xử lý dữ liệu, áp dụng khi tỉ lệ missing nhỏ và không gây ảnh hưởng nhiều đến kết quả nghiên cứu.

Điền giá trị thay thế và Nội suy (Fillna & Interpolation)

Khi gặp dữ liệu thiếu, ngoài việc loại bỏ hoặc điền bằng các giá trị thống kê đơn giản (trung bình, trung vị, mode), Pandas còn hỗ trợ kỹ thuật **nội suy (interpolation)**. Kỹ thuật này ước lượng giá trị bị thiếu dựa trên các quan sát lân cận, giúp dữ liệu trở nên mượt hơn và phản ánh xu hướng thực tế tốt hơn so với việc dùng một hằng số cố định.

- **Điền bằng giá trị thống kê (Fillna với mean/median):**

```python
import pandas as pd

# Series co gia tri thieu
s = pd.Series([2.0, 2.2, None, 2.7, 4.9])

# Dien gia tri thieu bang trung bibinh
s_filled = s.fillna(s.mean())
print(s_filled)
```

```
==================  Output ==================
0    2.000000
1    2.200000
2    2.950000
3    2.700000
4    4.900000
dtype: float64
```

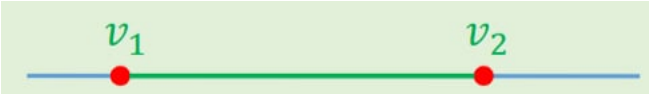
Ở đây giá trị thiếu được thay bằng trung bình của toàn bộ Series (2.95). Cách này giữ nguyên kích thước dữ liệu nhưng có thể làm giảm độ biến thiên tự nhiên.

- **Nội suy tuyến tính (Linear Interpolation):**

```
 Noi suy tuyen tinh
s_linear = s.interpolate(method='linear')
print(s_linear)
```

```
==================  Output ==================
0    2.00
1    2.20
2    2.45
3    2.70
4    4.90
dtype: float64
```

Phương pháp tuyến tính tính toán giá trị bị thiếu dựa trên 2 điểm liền kề theo công thức đường thẳng. Kết quả ở vị trí thứ 2 là trung bình cộng của 2.2 và 2.7, tức 2.45.



Minh họa nội suy tuyến tính trên đồ thị (đường thẳng nối giữa hai điểm đã biết)

- Nội suy theo thời gian (Time Interpolation):

```
import pandas as pd

# Series co chi muc thoi gian
dates = pd.date_range("2023-01-01",
    periods=5, freq="D")
s_time = pd.Series([2.0, None, None, 2.7,
     4.9], index=dates)

# Noi suy theo thoi gian
s_interp_time = s_time.interpolate(method
    ='time')
print(s_interp_time)
```

```
==================  Output ==================
2023-01-01    2.000000
2023-01-02    2.233333
2023-01-03    2.466667
2023-01-04    2.700000
2023-01-05    4.900000
Freq: D, dtype: float64
```

Khi dữ liệu có chỉ mục thời gian, `interpolate(method='time')` sẽ phân bổ giá trị dựa trên khoảng cách thời gian. Trong ví dụ trên, 2 ngày trống (02/01 và 03/01) được nội suy theo tỉ lệ tuyến tính giữa ngày 01/01 (2.0) và 04/01 (2.7).

Các phương pháp nội suy khác:

- **Polynomial:** nội suy theo đa thức bậc n, ví dụ `method='polynomial', order=2`.
- **Spline:** dùng spline bậc n để ước lượng, mượt hơn linear.

- **Pad/ffill, bfill:** về bản chất cũng là một dạng nội suy đặc biệt, lấy giá trị trước (forward fill) hoặc sau (backward fill).

Nội suy giúp tái tạo dữ liệu thiếu một cách tự nhiên và giữ được xu hướng. Tuy nhiên, nếu dữ liệu gốc nhiễu hoặc có quá nhiều missing liên tiếp, nội suy có thể tạo ra giá trị không phản ánh đúng thực tế.

Forward Fill và Backward Fill

Một trong những cách xử lý dữ liệu thiếu đơn giản nhưng hữu ích là **Forward Fill (ffill)** và **Backward Fill (bfill)**. Thay vì dùng một hằng số hay giá trị trung bình, ta tận dụng chính các giá trị quan sát lân cận để thay thế cho ô bị thiếu:

- **Forward Fill (ffill):** điền giá trị bị thiếu bằng *giá trị gần nhất phía trước.*
- **Backward Fill (bfill):** điền giá trị bị thiếu bằng *giá trị gần nhất phía sau.*

Phương pháp này đặc biệt hữu ích trong dữ liệu chuỗi thời gian (time series), nơi mà giá trị ở các thời điểm liên tiếp thường có mối liên hệ chặt chẽ.

```python
import pandas as pd

# Vi du Series voi gia tri thieu
s = pd.Series([2.0, None, None, 2.7, None, 4.9])

# Forward fill và Backward fill
s_ffill = s.fillna(method='ffill')
s_bfill = s.fillna(method='bfill')

print("Forward Fill:\n", s_ffill)
print("\nBackward Fill:\n", s_bfill)
```

```
==================  Output ==================
Forward Fill:
0    2.0
1    2.0
2    2.0
3    2.7
4    2.7
5    4.9
dtype: float64

Backward Fill:
0    2.0
1    2.7
2    2.7
3    2.7
4    4.9
5    4.9
dtype: float64
```

So sánh:

- Với `ffill`, giá trị thiếu ở index 1 và 2 được điền bằng 2.0 (giá trị ở index 0), còn giá trị thiếu ở index 4 được điền bằng 2.7.
- Với `bfill`, giá trị thiếu ở index 1 và 2 được điền bằng 2.7 (giá trị ở index 3), còn giá trị thiếu ở index 4 được điền bằng 4.9.

Ưu điểm:

- Giữ được mối liên kết thời gian trong dữ liệu.
- Đơn giản, tính toán nhanh, dễ hiểu.

Nhược điểm:

- Nếu khoảng missing kéo dài nhiều bước, việc điền bằng một giá trị lân cận có thể gây sai lệch lớn so với thực tế.
- Không phù hợp với dữ liệu không tuần tự hoặc không có ý nghĩa theo thời gian.

Minh họa Forward Fill và Backward Fill

Index	Original	Forward Fill	Backward Fill
0	2.0	2.0	2.0
1	NaN	2.0	2.7
2	NaN	2.0	2.7
3	2.7	2.7	2.7
4	NaN	2.7	4.9
5	4.9	4.9	4.9

1.3. Giảm nhiễu dữ liệu (Noise Reduction)

Trung bình trượt (Moving Average): SMA, EMA và TMA

Mục tiêu. Moving Average (MA) là nhóm kỹ thuật làm mượt chuỗi tín hiệu nhằm: (i) giảm nhiễu ngẫu nhiên (noise), (ii) làm nổi bật xu hướng ngắn/trung hạn, và (iii) chuẩn bị dữ liệu cho phân tích tiếp theo (phát hiện bất thường, dự báo). Ba biến thể phổ biến: **SMA** (Simple), **EMA** (Exponential), **TMA** (Triangular).

Định nghĩa

Giả sử chuỗi quan sát $\{s_t\}$, cửa sổ trượt kích thước $k \in \mathbb{N}$, tại thời điểm t:

- **SMA (Simple Moving Average):**

$$\mathrm{SMA}_t(k) \;=\; \frac{1}{k}\sum_{i=0}^{k-1} s_{t-i}.$$

  Mỗi điểm trong cửa sổ có trọng số bằng nhau.

- **EMA (Exponential Moving Average):**

$$\mathrm{EMA}_t(\alpha) \;=\; \alpha s_t + (1-\alpha)\,\mathrm{EMA}_{t-1}(\alpha), \quad \alpha \in (0,1].$$

  Thực hành thường lấy $\alpha = \frac{2}{k+1}$ để so sánh tương ứng với cửa sổ k của SMA.

- **TMA (Triangular Moving Average):** trung bình *có trọng số tam giác*, lớn nhất ở giữa, nhỏ dần về hai biên.

$$\mathrm{TMA}_t(k) \;=\; \frac{\sum_{i=0}^{k-1} w_i\, s_{t-i}}{\sum_{i=0}^{k-1} w_i}, \quad \text{với } w_i \text{ tăng tuyến tính rồi giảm (dạng tam giác).}$$

  Một cách thực hiện phổ biến: SMA của SMA (áp dụng SMA hai lần) tạo ra phân bố trọng số tam giác.

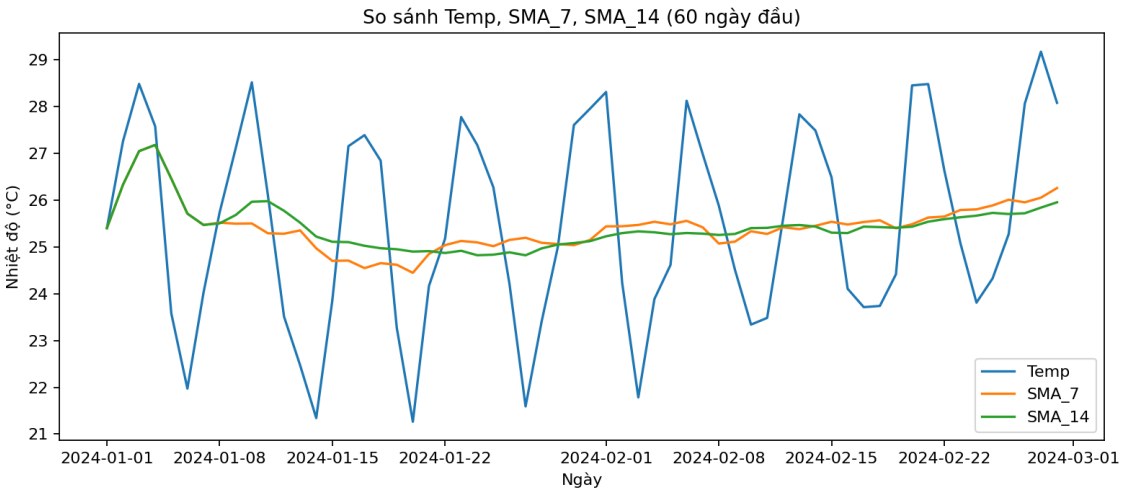
- SMA làm mượt mạnh nhưng gây trễ (lag) đáng kể; EMA phản ứng nhanh hơn nhờ trọng số lớn cho s_t;
- TMA cân bằng giữa mượt và trễ, giảm nhiễu tốt hơn SMA cùng k nhờ nhấn mạnh vùng trung tâm cửa sổ.

(a) SMA — Simple Moving Average

Khi dùng SMA:

- Chọn k nhỏ (3–7) để mượt nhẹ, giữ nhịp biến động ngắn hạn.
- Chọn k trung bình (7–21) để xem xu hướng tuần/tháng (tùy miền: thời tiết, kinh tế...).
- **Cảnh báo:** k càng lớn thì trễ càng nhiều; `min_periods` giúp kiểm soát số điểm tối thiểu trong cửa sổ.

```python
import pandas as pd

# Vi du: chuoi nhiet do theo ngay (gia dinh da co cot 'Temp' và index dang DatetimeIndex)
# df = pd.read_csv("Daily-min-temp.csv", parse_dates=["Date"], index_col="Date")

# SMA voi cua so 7 ngay va 14 ngangay
sma_7  = df["Temp"].rolling(window=7,  min_periods=1, center=False).mean()
sma_14 = df["Temp"].rolling(window=14, min_periods=1, center=False).mean()

df_out = pd.DataFrame({"Temp": df["Temp"], "SMA_7": sma_7, "SMA_14": sma_14}).head(10)
print(df_out)
```



So sánh Temp, SMA_7, SMA_14 cho 60 ngày đầu (minh họa)

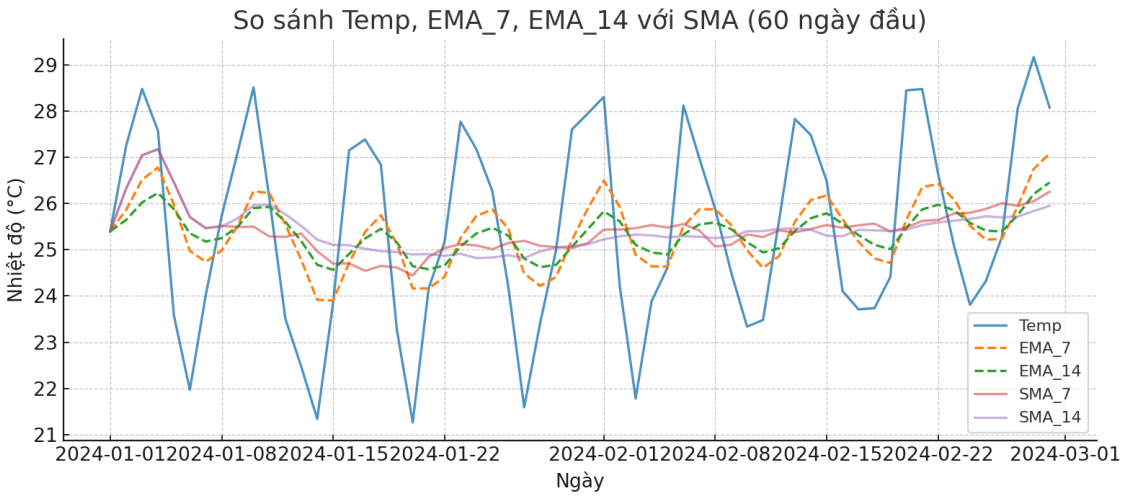
Ghi chú

- `center=True` vẽ SMA "căn giữa" cửa sổ (giảm trễ thị giác), đổi lại cuối chuỗi sẽ thiếu giá trị.
- `min_periods` kiểm soát số mẫu tối thiểu để tính trung bình (tránh NaN đầu chuỗi).

(b) EMA — Exponential Moving Average

Ưu điểm: phản ứng nhanh với thay đổi mới (trọng số lớn cho s_t), trễ nhỏ hơn SMA cùng "độ mượt".
Thực hành: chọn `span=k` để tương đương trực giác với cửa sổ k; `adjust=False` dùng dạng đệ quy chuẩn trong phân tích dòng thời gian.

```
ema_7  = df["Temp"].ewm(span=7,  adjust=False).mean()
ema_14 = df["Temp"].ewm(span=14, adjust=False).mean()

df_out = pd.DataFrame({"Temp": df["Temp"], "EMA_7": ema_7, "EMA_14": ema_14}).head(10)
print(df_out)
```



So sánh Temp, EMA_7, EMA_14 với SMA cho 60 ngày đầu (minh họa). EMA phản ứng nhanh hơn với biến động dữ liệu.

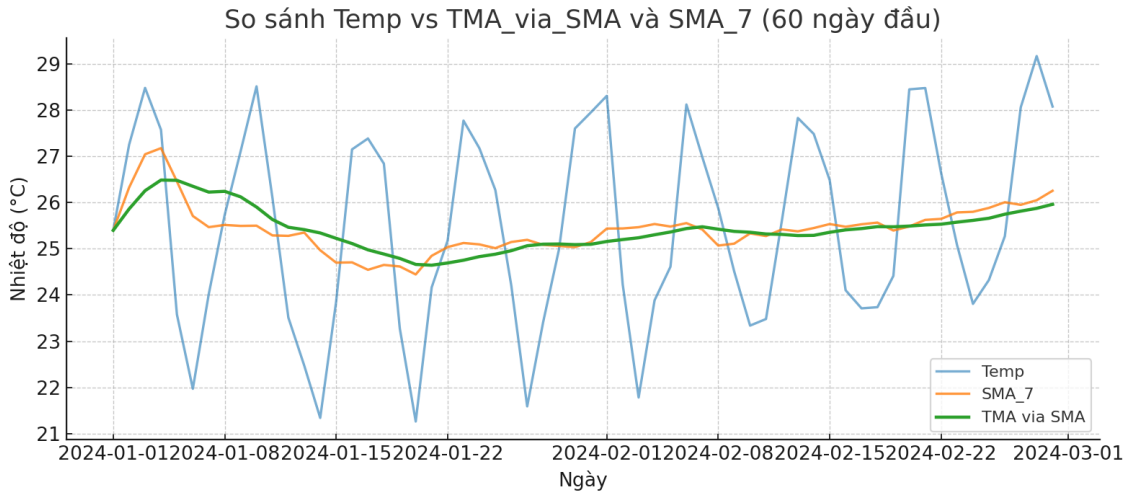
Công thức tương đương: với `span = k`, $\alpha = \frac{2}{k+1}$. EMA khởi tạo thường lấy $\text{EMA}_0 = s_0$ (hoặc trung bình vài điểm đầu).

(c) TMA — Triangular Moving Average

Ý tưởng. TMA dùng *trọng số tam giác*: nhỏ ở biên, lớn dần về giữa cửa sổ, rồi giảm dần. Điều này làm mượt mạnh hơn SMA (cùng k) nhưng vẫn giữ được độ "trung tâm" của khu vực lân cận, thường cho đường mượt & ít méo pha hơn ở biên so với SMA thuần.

- **Cách 1 — TMA là SMA của SMA (đơn giản, không phụ thuộc gói ngoài).**

```python
# TMA bang cach ap dung SMA 2 lan (k co dinh)
k = 7
sma_once = df["Temp"].rolling(window=k, min_periods=1).mean()
tma_sma  = sma_once.rolling(window=k, min_periods=1).mean()

df_out = pd.DataFrame({"Temp": df["Temp"], "TMA_via_SMA": tma_sma}).head(10)
print(df_out)
```



So sánh Temp với TMA (qua SMA) và SMA_7 cho 60 ngày đầu. TMA mượt hơn SMA.

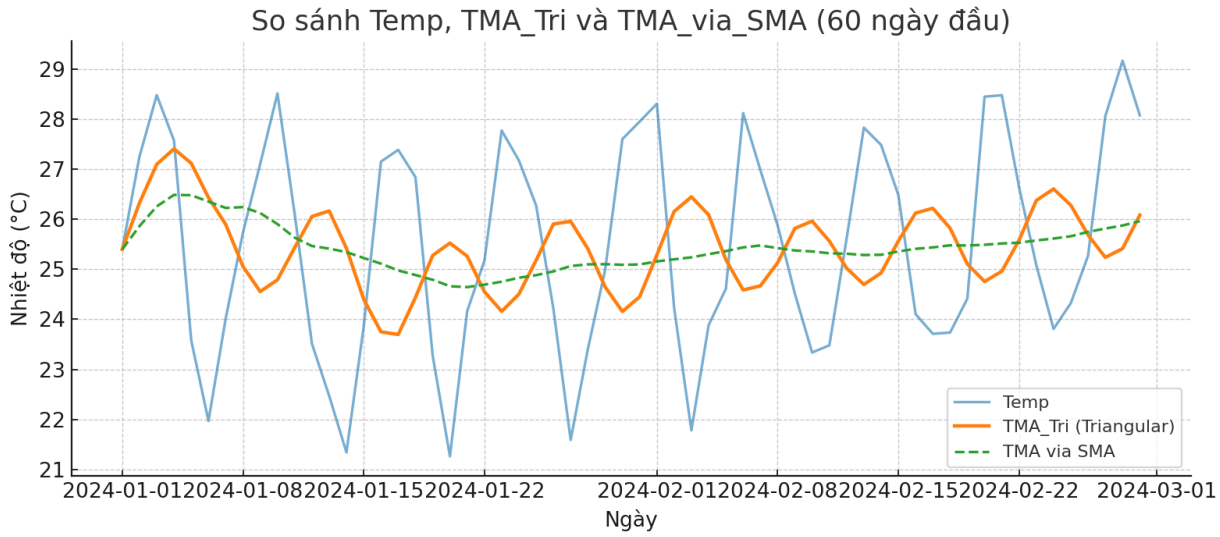
- **Cách 2 — TMA với trọng số tam giác tường minh.**

  – Với k lẻ, trọng số tam giác có dạng $w = [1, 2, \ldots, m, \ldots, 2, 1]$ với $k = 2m - 1$;

  – Với k chẵn, dùng cặp trung tâm cân xứng. Có thể hiện thực bằng `rolling().apply()`.

```python
import numpy as np

def triangular_weights(k: int):
    if k \% 2 == 1:              # k le: dinh o chinh giua
        m = (k + 1) // 2
        w = np.r_[np.arange(1, m+1), np.arange(m-1, 0, -1)]
    else:                        # k chan: hai dinh giua bang nhau
        m = k // 2
```

```
        w = np.r_[np.arange(1, m+1), np.arange(m, 0, -1)]
    return w / w.sum()

def tma_triangular(x):
    # x la numpy array do dai k; co the chua NaN -> bo NaN tuong ung trong trong so
    w_all = triangular_weights(len(x))
    mask = ~np.isnan(x)
    if mask.sum() == 0:
        return np.nan
    w = w_all[mask]
    return np.dot(x[mask], w) / w.sum()

k = 7
tma_tri = df["Temp"].rolling(window=k, min_periods=1).apply(tma_triangular, raw=True)

df_out = pd.DataFrame({"Temp": df["Temp"], "TMA_Tri": tma_tri}).head(10)
print(df_out)
```



Đồ thị Temp với TMA_Tri (trọng số tam giác) và TMA_via_SMA (60 ngày đầu). Hai cách cho kết quả tương đồng.

So sánh nhanh:

- **SMA:** dễ hiểu, mượt vừa phải, trễ lớn khi k tăng.
- **EMA:** phản ứng nhanh, trễ nhỏ hơn, phù hợp dữ liệu nhạy cảm theo thời gian.
- **TMA:** mượt hơn SMA cùng k, trọng số tập trung vùng trung tâm $\Rightarrow$ thường ổn định hơn trước nhiễu cục bộ.

Thực hành và mẹo chọn tham số

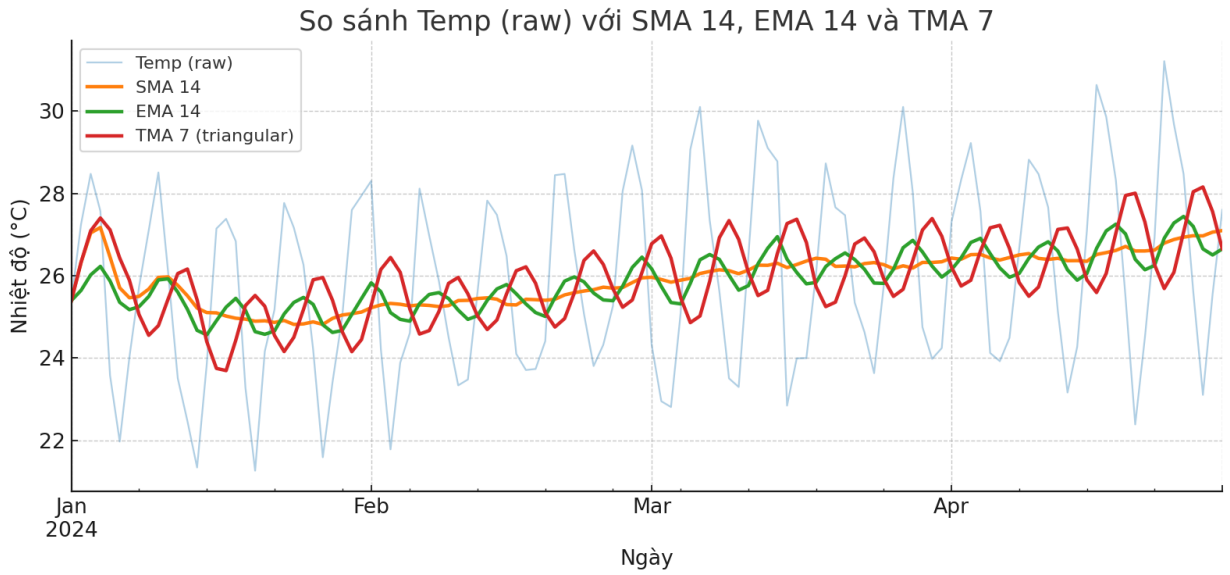
Chọn k (hoặc span) theo miền bài toán:

- Dữ liệu thời tiết/ngày: $k = 7$ (chu kỳ tuần) để mượt nhiễu ngày–ngày; $k = 14, 21$ để nhìn xu hướng dài hơn.
- Dữ liệu cảm biến nhiễu cao: thử TMA hoặc EMA `span` nhỏ để giữ phản ứng nhanh.

Tuỳ chọn triển khai:

- `center=True` để vẽ đường mượt "căn giữa" cửa sổ (đẹp khi minh hoạ), nhưng lưu ý giá trị cuối chuỗi sẽ thiếu.
- `min_periods` để tránh NaN ở đầu cửa sổ (ví dụ `min_periods=1`).
- Kết hợp làm sạch trước (drop/ffill/interpolate) giúp MA ổn định hơn.

```
ax = df["Temp"].plot(alpha=0.35, linewidth=1.0, label="Temp (raw)")
sma_14.plot(ax=ax, linewidth=2.0, label="SMA 14")
ema_14.plot(ax=ax, linewidth=2.0, label="EMA 14")
tma_tri.plot(ax=ax, linewidth=2.0, label="TMA 7 (triangular)")
ax.legend()
```



Đồ thị chồng Temp (raw), SMA 14, EMA 14 và TMA 7 để so sánh độ mượt và độ trễ.

1.4. Nhóm và tổng hợp dữ liệu nâng cao (Advanced GroupBy & Aggregation)

Khi thao tác với dữ liệu thực tế, việc chỉ tính trung bình (`mean()`) hay tổng (`sum()`) là chưa đủ. Trong nhiều trường hợp, ta cần tổng hợp nhiều chỉ số cùng lúc, hoặc tự viết hàm tuỳ chỉnh để tính toán phức tạp hơn. Pandas cung cấp công cụ mạnh mẽ thông qua `groupby()` kết hợp với `agg()`.

Một số điểm nổi bật:

- Có thể áp dụng nhiều hàm thống kê cùng lúc cho các cột khác nhau.
- Hỗ trợ viết hàm tuỳ chỉnh bằng `lambda` hoặc định nghĩa riêng.
- Kết quả trả về có cấu trúc bảng gọn gàng, dễ dàng xuất báo cáo.

Ví dụ 1 — Phân tích doanh số quảng cáo

Giả sử ta có bảng dữ liệu `advertising_simple` gồm 2 cột: `Sales` (doanh số) và `Budget` (ngân sách), kèm theo kênh `Channel`. Ta muốn biết trung bình và giá trị lớn nhất của doanh số theo từng kênh, đồng thời tính tổng ngân sách (quy đổi nghìn đơn vị).

```
df.groupby('Channel').agg({
    'Sales': ['mean','max'],
    'Budget': lambda x: x.sum()/1000
})
```

Channel	Sales_mean	Sales_max	Budget_sum_k
Online	296.6666666666667	310	6.15
Radio	116.66666666666667	130	1.65
TV	223.33333333333334	250	3.3

Bảng tổng hợp: Sales (mean, max) và Budget (tổng chia 1000) theo từng Channel.

Giải thích:

- Cột `Sales` được áp dụng cả `mean` và `max`.
- Cột `Budget` áp dụng hàm `lambda` để chia tổng cho 1000.

Điều này cho phép ta vừa thống kê cơ bản, vừa tùy biến theo nhu cầu báo cáo.

Ví dụ 2 — Đếm số lượng bản ghi theo nhãn

Với tập dữ liệu `Tweets dataset`, giả sử mỗi bản ghi có nhãn `label` (như `positive`, `negative`, `neutral`), ta muốn đếm số lượng tweet thuộc từng nhãn:

```
tweets.groupby('label')['text'].count()
```

label	text
negative	2
neutral	3
positive	4

Số lượng tweet theo từng nhãn: positive, negative, neutral.

Ý nghĩa: Đây là thao tác thường gặp trong phân tích dữ liệu văn bản (text mining). Khi làm sentiment analysis, bước đầu tiên chính là thống kê số lượng dữ liệu cho mỗi loại nhãn để xem tập dữ liệu có cân bằng hay không. Nếu dữ liệu lệch (ví dụ `positive` quá nhiều so với `negative`), ta cần cân nhắc kỹ khi huấn luyện mô hình học máy.

Ví dụ 3 — Áp dụng nhiều phép thống kê phức tạp

Không chỉ gói gọn trong `mean` hay `max`, ta có thể kết hợp: độ lệch chuẩn (`std`), giá trị đầu tiên (`first`), giá trị cuối cùng (`last`), hoặc thậm chí viết riêng hàm để phát hiện bất thường:

```
def iqr(x):
    q1, q3 = x.quantile([0.25, 0.75])
    return q3 - q1

df.groupby('Channel').agg({
    'Sales': ['mean','std',iqr],
    'Budget': ['sum','min','max']
})
```

('Channel', '')	('Sales', 'mean')	('Sales', 'std')	('Sales', 'iqr')	('Budget', 'sum')	('Budget', 'min')	('Budget', 'max')
Online	296.6666666666667	15.275252316519461	15.0	6150	2000	2100
Radio	116.66666666666667	15.275252316519465	15.0	1650	500	600
TV	223.33333333333334	25.166114784235834	25.0	3300	1000	1200

Bảng tổng hợp: Sales (mean, std, IQR) và Budget (sum, min, max) theo từng Channel.

Nhận xét:

- Dùng `iqr` để phát hiện kênh có phân phối doanh số biến động mạnh.
- Có thể dễ dàng mở rộng sang nhiều chỉ số khác mà không cần chỉnh sửa dữ liệu gốc.

1.5. Xử lý và phân tích dữ liệu chuỗi thời gian (Time Series)

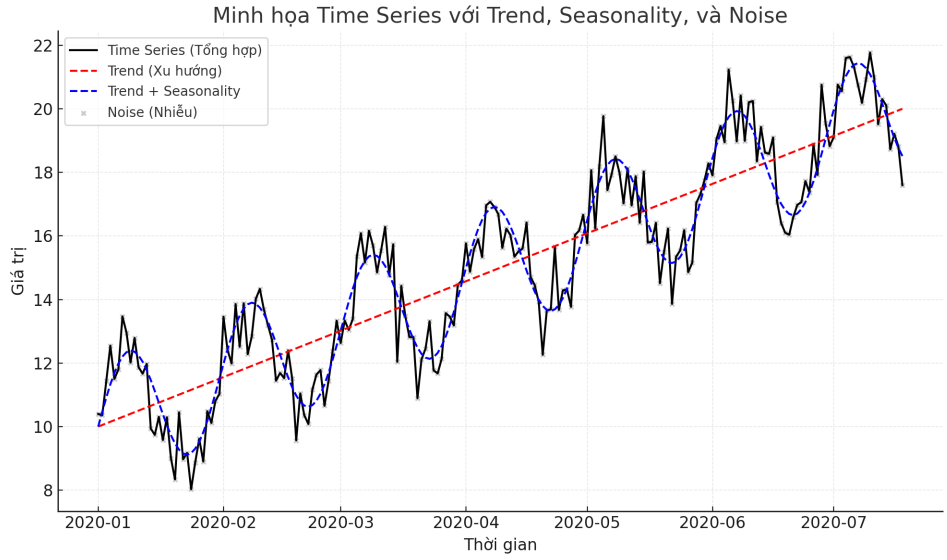
Khái niệm Time Series

Time Series (chuỗi thời gian) là tập dữ liệu mà các quan sát được thu thập theo *thứ tự thời gian*. Khác với dữ liệu bảng thông thường, ở đây vị trí (index) đóng vai trò rất quan trọng vì nó gắn liền với mốc thời gian.
Ví dụ:

- Nhiệt độ hàng ngày tại Hà Nội.
- Giá cổ phiếu Apple theo từng phút.
- Lượng tweet liên quan đến một hashtag theo từng giờ.

Đặc điểm cốt lõi của Time Series:

- **Xu hướng (Trend):** xu hướng tăng hoặc giảm kéo dài, ví dụ giá nhà đất tăng dần qua các năm.
- **Mùa vụ (Seasonality):** sự lặp lại theo chu kỳ, ví dụ doanh thu bán lẻ tăng mạnh vào dịp Tết/Noel.
- **Nhiễu (Noise):** biến động ngẫu nhiên khó dự đoán, ví dụ do lỗi cảm biến, biến cố bất ngờ.

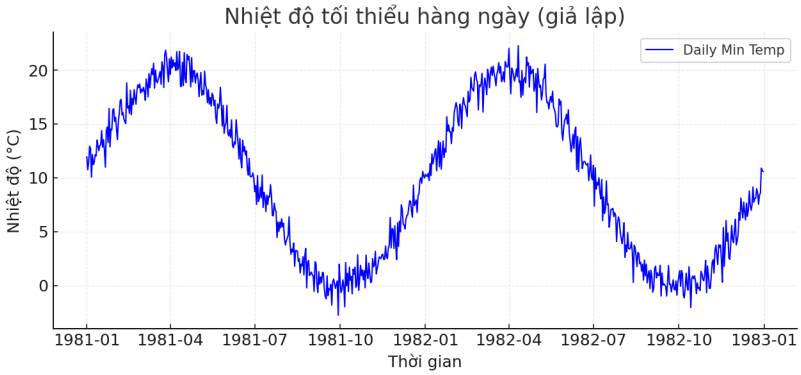


Hình 1: Ví dụ minh họa ba thành phần chính của Time Series: trend, seasonality và noise.

Thao tác cơ bản với Time Series trong Pandas

```
1 import pandas as pd
2
```

```
# Doc du lieu nhiet do toi thieu hang ngay
df = pd.read_csv("Daily-min-temp.csv",
                 parse_dates=['Date'], index_col='Date')

# In 5 dong dau
print(df.head())

# Lay du lieu 6 thang dau nam 1981
df['1981-01':'1981-06']
```



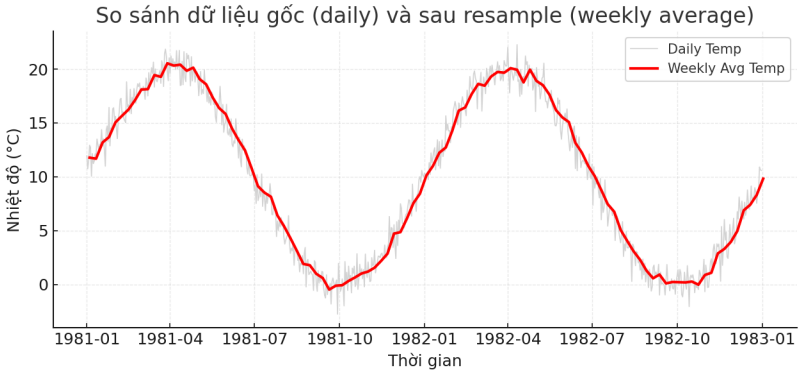
Trực quan hóa dữ liệu nhiệt độ tối thiểu hàng ngày (giả lập).

Trong Pandas, việc đặt `Date` làm index giúp ta có thể:

- Truy xuất theo khoảng thời gian, ví dụ `df['1981-01':'1981-06']`.
- Tính toán trực tiếp dựa trên khoảng thời gian (time-aware).
- Dễ dàng **`resample`**, **`rolling`**, hoặc kết hợp nhiều mức tần suất.

Resampling (Thay đổi tần suất quan sát)

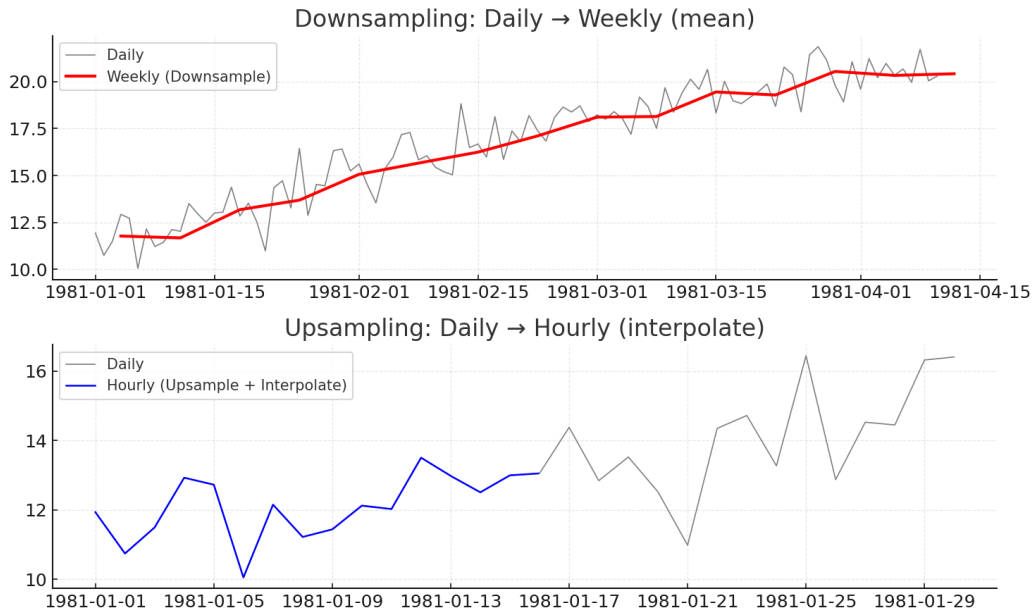
```
df['Temp'].resample('W').mean() # Lay trung binh nhiet do theo tuan
```



So sánh dữ liệu nhiệt độ gốc theo ngày và dữ liệu trung bình theo tuần (resample).

Resampling gồm hai hướng:

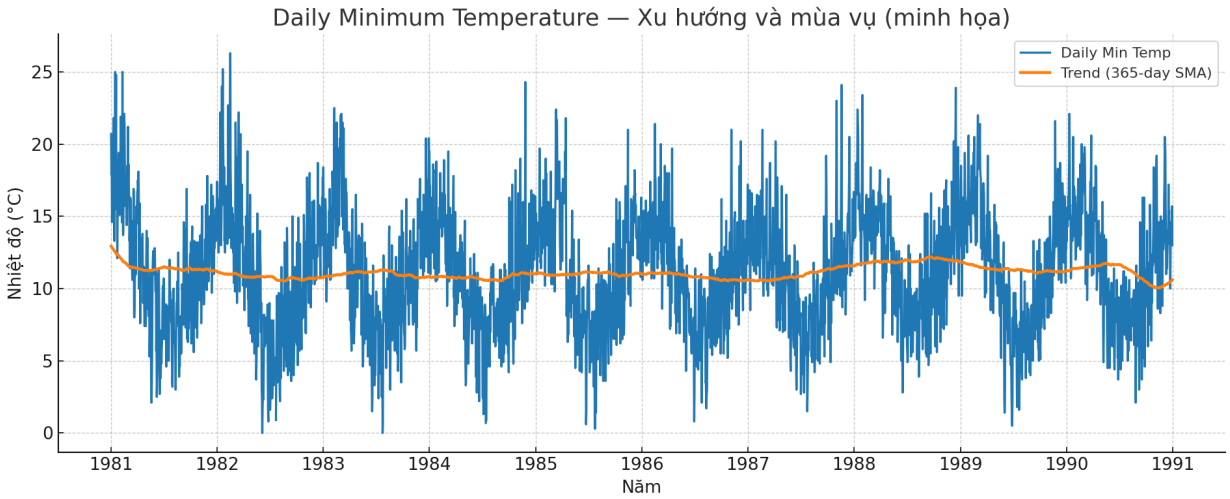
- **Downsampling:** giảm tần suất, ví dụ từ ngày → tuần. Cần chỉ rõ hàm tổng hợp (mean, sum...).
- **Upsampling:** tăng tần suất, ví dụ từ ngày → giờ. Thường phải đi kèm `ffill()` hoặc `interpolate()` để điền giá trị.



Ví dụ minh họa Downsampling (daily → weekly) và Upsampling (daily → hourly với interpolate).

Ứng dụng thực tế

- **Weather data:** Dữ liệu thời tiết (ví dụ: `Daily-min-temp`) thường có tính chu kỳ rõ ràng theo mùa. Ta có thể resample theo tháng hoặc năm để phát hiện xu hướng biến đổi khí hậu dài hạn. Ngoài ra, việc phân tích độ lệch chuẩn giữa các năm giúp nhận diện mức độ biến động nhiệt độ.
- **Financial data:** Chuỗi giá cổ phiếu hoặc chỉ số chứng khoán thường nhiều nhiễu. Các kỹ thuật `SMA` (Simple Moving Average) và `EMA` (Exponential Moving Average) được dùng để:
  - Làm mượt dữ liệu ngắn hạn.
  - So sánh đường SMA và EMA để phát hiện tín hiệu mua/bán (crossover).
- **Social media data:** Với dữ liệu tweet hoặc lượt xem video, ta thường đếm số lượng sự kiện theo ngày. Tuy nhiên, dữ liệu gốc có thể nhiễu và biến động mạnh theo giờ hoặc ngày. Resample sang tuần (downsampling) sẽ giảm nhiễu, giúp nhìn rõ xu hướng dài hạn (ví dụ: sự kiện nóng dần theo từng tuần).



Case study với `Daily-min-temp`: trực quan hóa dữ liệu nhiệt độ tối thiểu hằng ngày, kèm xu hướng dài hạn qua `SMA 365 ngày`. Đường SMA giúp phát hiện xu hướng khí hậu đang tăng nhẹ theo thời gian, trong khi dữ liệu gốc dao động mạnh theo mùa.

2. Ứng dụng vào case study thực tế

Case study: Dự báo thời tiết (Temperature Forecasting)

Bối cảnh & mục tiêu. Ta sử dụng dữ liệu `Daily minimum temperature` (nhiệt độ tối thiểu hằng ngày) để xây dựng một quy trình dự báo ngắn hạn theo phong cách *baseline*. Quy trình bao gồm: (i) nạp & rà soát chất lượng dữ liệu, (ii) xử lý thiếu & giảm nhiễu bằng Moving Average (SMA/EMA), (iii) khám phá xu hướng (trend) & mùa vụ (seasonality), (iv) tạo đặc trưng *lag/rolling*, (v) chia tập theo thời gian và đánh giá các mô hình cơ sở (persistence, SMA).

2.1. Nạp dữ liệu và kiểm tra ban đầu

```
import pandas as pd

# 1) Nap du lieu: cot Date -> datetime, dat lam index theo thoi gian
df = pd.read_csv("timeseries_daily-minimum-temperatures.csv", parse_dates=["Date"])
df = df.set_index("Date").sort_index()

# Dat ten cot nhiet do (neu khac Temp thi lay cot cuoi)
temp_col = "Temp" if "Temp" in df.columns else df.columns[-1]

# 2) Ra soat nhanh
print(df.head(5))      # 5 dong dau
print(df.tail(5))      # 5 dong cuoi
print(df.info())       # kieu du lieu, so dong
print(df.isna().sum())# dem missing
```

5 dòng đầu tiên

Date	Temp
1981-01-01 00:00:00	20.7
1981-01-02 00:00:00	17.9
1981-01-03 00:00:00	18.8
1981-01-04 00:00:00	14.6
1981-01-05 00:00:00	15.8

5 dòng cuối cùng

Date	Temp
1990-12-27 00:00:00	14
1990-12-28 00:00:00	13.6
1990-12-29 00:00:00	13.5
1990-12-30 00:00:00	15.7
1990-12-31 00:00:00	13

Thông tin DataFrame & Missing values

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3650 entries, 1981-01-01 to 1990-12-31
Data columns (total 1 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   Daily minimum temperatures  3650 non-null   object
dtypes: object(1)
memory usage: 57.0+ KB

Missing values:
Daily minimum temperatures    0
```

Bảng 5 dòng đầu & 5 dòng cuối + summary info

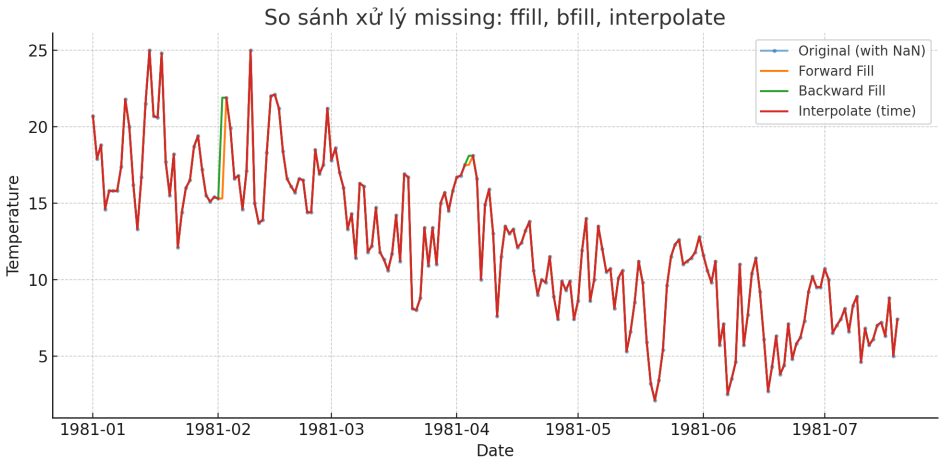
Lý do: Với chuỗi thời gian, **chỉ mục thời gian** (DatetimeIndex) cực kỳ quan trọng: giúp `slice` theo mốc, `resample`, `rolling`, v.v. Việc `sort_index()` đảm bảo thứ tự thời gian chính xác.

2.2. Xử lý dữ liệu thiếu (Missing): ffill/bfill vs. interpolate

Chiến lược. Nếu `NaN` rải rác, `interpolate(method="time")` thường mượt và trung thực hơn so với điền hằng số. Nếu `NaN` xuất hiện thành cụm ngắn, `ffill/bfill` là lựa chọn đơn giản và ổn định.

```
# Vi du pipeline xu ly thieu:
df["Temp_fill_ffill"] = df[temp_col].ffill()
df["Temp_fill_bfill"] = df[temp_col].bfill()
df["Temp_fill_time"]  = df[temp_col].interpolate(method="time")

# So sanh tren mot doan ngan
window = df.iloc[:200][[temp_col, "Temp_fill_ffill", "Temp_fill_bfill", "
    Temp_fill_time"]]
```

```
print(window.head(10))
```

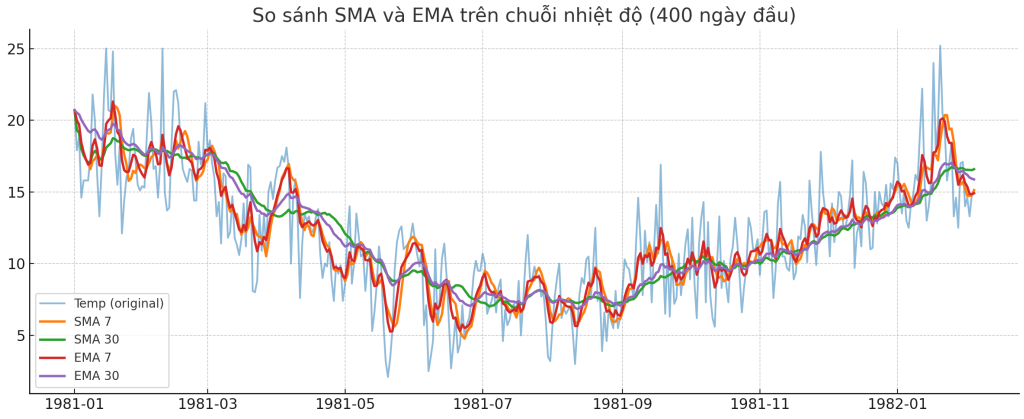


Đồ thị đoạn ngắn: đường gốc vs. ffill/bfill vs. interpolate

2.3. Giảm nhiễu bằng Moving Average (SMA/EMA)

Nguyên tắc. SMA làm mượt mạnh nhưng tạo độ trễ; EMA phản ứng nhanh hơn vì đặt trọng số lớn cho quan sát gần nhất. Chọn cửa sổ k theo chu kỳ tự nhiên (ví dụ $k = 7$ cho tuần).

```
s = df["Temp_fill_time"] # Su dung cot da dien thieu theo thoi gian

# SMA k=7, 30; EMA span=7, 30
df["SMA_7"]  = s.rolling(window=7,  min_periods=1).mean()
df["SMA_30"] = s.rolling(window=30, min_periods=1).mean()
df["EMA_7"]  = s.ewm(span=7,  adjust=False).mean()
df["EMA_30"] = s.ewm(span=30, adjust=False).mean()

print(df[[temp_col, "SMA_7", "EMA_7"]].head(15))
```

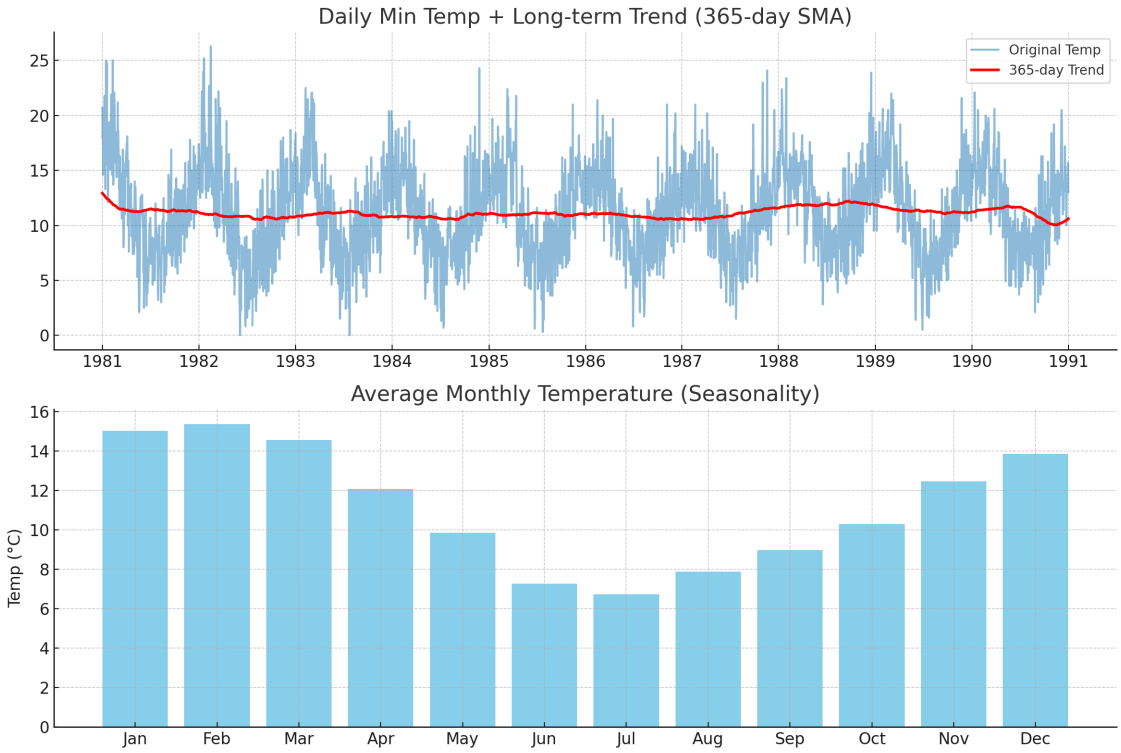


Đồ thị chồng: Temp gốc, SMA_7, SMA_30, EMA_7, EMA_30 để so sánh độ mượt & độ trễ

2.4. Khám phá xu hướng (Trend) và mùa vụ (Seasonality)

Ý tưởng. Dùng `rolling` dài hạn (vd. 365 ngày) để quan sát xu hướng; dùng `groupby` theo tháng để quan sát mùa vụ (climatology theo tháng).

```
# Trend dai han: trung binh truot 365 ngay (xap xi 1 nam)
df["Trend_365"] = s.rolling(window=365, center=True, min_periods=1).mean()

# Mua vu theo thang: trung binh nhiet do theo thang trong nam
monthly_clim = s.groupby(s.index.month).mean().round(2)
print(monthly_clim)
```



(1) đồ thị chuỗi gốc + Trend_365; (2) biểu đồ cột trung bình theo 12 tháng để thấy seasonality

2.5. Tạo đặc trưng dự báo (Feature Engineering)

Nguyên tắc thời gian: tránh *data leakage*. Mọi đặc trưng `lag/rolling` phải dùng *quá khứ* để dự báo tương lai.

```
# Lags: hom qua, tuan truoc
df["lag_1"]  = s.shift(1)
df["lag_7"]  = s.shift(7)
df["lag_14"] = s.shift(14)

# Rolling features (7 ngay gan nhat)
```

```
df["roll_mean_7"] = s.rolling(7).mean()
df["roll_std_7"]  = s.rolling(7).std()

# EMA nhu dac trung phan ung nhanh
df["ema_7"] = s.ewm(span=7, adjust=False).mean()

# Seasonality flags
df["month"]     = df.index.month
df["dayofyear"] = df.index.dayofyear

print(df[[temp_col,"lag_1","lag_7","roll_mean_7","ema_7","month","dayofyear"]].head
    (12))
```

)aily minimum temperatu	lag_1	lag_7	roll_mean_7	ema_7	month	dayofyear
20.7	nan	nan	nan	20.7	1.0	1.0
17.9	20.7	nan	nan	20.0	1.0	2.0
18.8	17.9	nan	nan	19.7	1.0	3.0
14.6	18.8	nan	nan	18.42	1.0	4.0
15.8	14.6	nan	nan	17.77	1.0	5.0
15.8	15.8	nan	nan	17.28	1.0	6.0
15.8	15.8	nan	17.06	16.91	1.0	7.0
17.4	15.8	20.7	16.59	17.03	1.0	8.0
21.8	17.4	17.9	17.14	18.22	1.0	9.0
20.0	21.8	18.8	17.31	18.67	1.0	10.0
16.2	20.0	14.6	17.54	18.05	1.0	11.0
13.3	16.2	15.8	17.19	16.86	1.0	12.0

Bảng nhỏ 12 dòng minh họa các đặc trưng đã tạo

2.6. Chia tập theo thời gian & thiết lập baseline

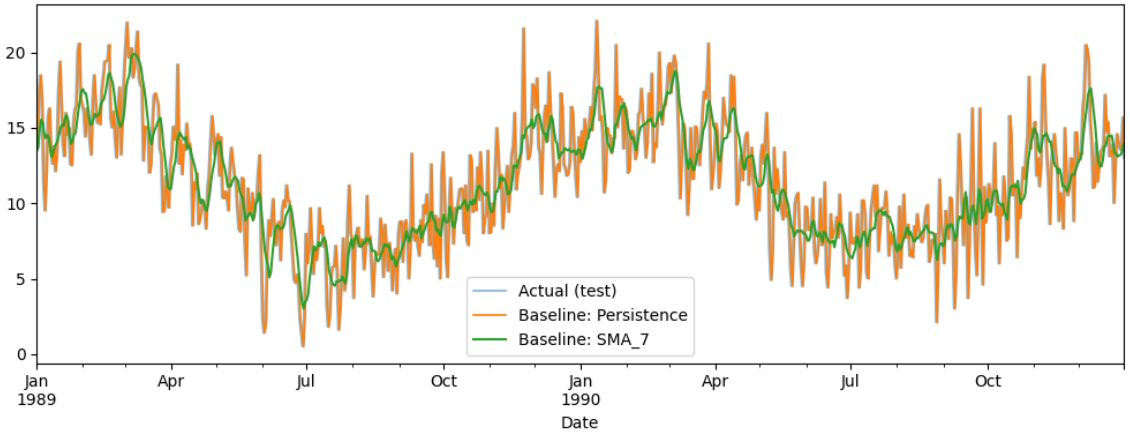
Lý do. Chuỗi thời gian phải *chia theo mốc thời gian* (không xáo trộn). Hai baseline phổ biến: **Persistence** (hôm nay = hôm qua) và **SMA_7** (trung bình 7 ngày).

```
import numpy as np

# Chia 80% dau lam train, 20% cuoi lam test
split_time = int(len(df) * 0.8)
train, test = df.iloc[:split_time].copy(), df.iloc[split_time:].copy()

# Baseline 1: Persistence
test["pred_persist"] = test["lag_1"]  # du bao hom nay = hom qua

# Baseline 2: SMA_7 (duong trung binh 7 ngay gan nhat)
test["pred_sma7"] = test["roll_mean_7"]

# Ham danh gia
def mae(y, yhat): return np.mean(np.abs(y - yhat))
def rmse(y, yhat): return np.sqrt(np.mean((y - yhat)**2))

y_true = test["Temp_fill_time"]
mae_p,  rmse_p  = mae(y_true, test["pred_persist"]), rmse(y_true, test["pred_persist"
    ])
mae_s7, rmse_s7 = mae(y_true, test["pred_sma7"]),   rmse(y_true, test["pred_sma7"])

print("Baseline-Persist  MAE/RMSE:", round(mae_p,3),  "/", round(rmse_p,3))
print("Baseline-SMA7     MAE/RMSE:", round(mae_s7,3), "/", round(rmse_s7,3))
```

Baseline	MAE	RMSE
Persistence	1.953	2.481
SMA_7	1.759	2.237

Bảng so sánh MAE/RMSE của 2 baseline

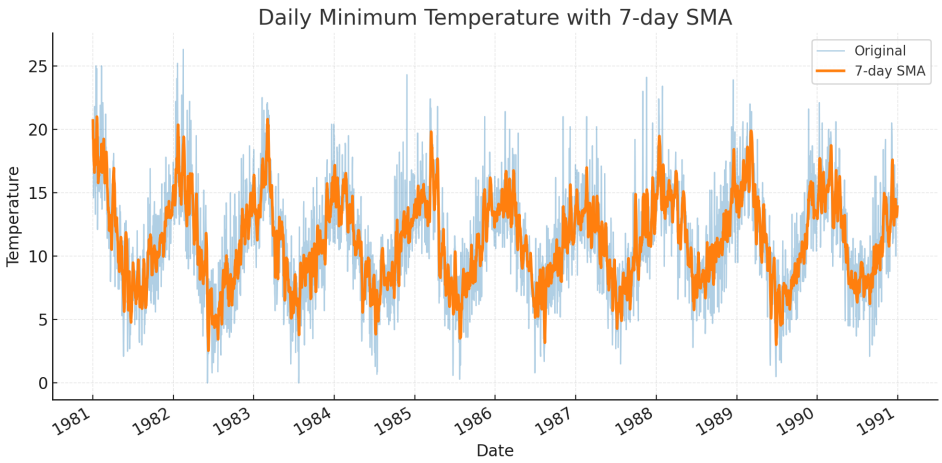
```
# Minh hoa truc quan tren tap test
ax = y_true.plot(label="Actual (test)", alpha=0.55, linewidth=1.2)
test["pred_persist"].plot(ax=ax, label="Baseline: Persistence", linewidth=1.5)
test["pred_sma7"].plot(ax=ax,     label="Baseline: SMA_7", linewidth=1.8)
ax.legend()
```



Đồ thị: Actual vs. Baseline Persistence vs. Baseline SMA_7

2.7. Minh họa: biểu đồ dự báo/tham chiếu bằng SMA

```
import matplotlib.pyplot as plt

# Ve toan chuoi voi duong SMA 7 ngay
ax = df["Temp_fill_time"].plot(label="Original", alpha=0.35, linewidth=1.0)
df["SMA_7"].plot(ax=ax, label="7-day SMA", linewidth=2.0)
plt.legend()
```



Biểu đồ dự báo/tham chiếu với SMA; đường SMA mượt, thể hiện xu hướng ngắn hạn

2.8. Tổng kết

Tóm tắt: (1) đặt chỉ mục thời gian, (2) xử lý thiếu bằng `interpolate` hoặc `ffill/bfill`, (3) giảm nhiễu với SMA/EMA theo chu kỳ thực tế, (4) khám phá trend/mùa vụ, (5) tạo đặc trưng `lag/rolling`, (6) chia theo thời gian, (7) đánh giá baseline bằng MAE/RMSE.
Gợi ý mở rộng: Thử `TMA` (trọng số tam giác) để mượt hơn SMA, và các mô hình chuyên biệt (ARIMA/ETS, Prophet) khi cần dự báo nâng cao.

Phần 4: Kỹ thuật nâng cao trong trực quan hóa dữ liệu

Sau khi đã làm chủ các thao tác tiền xử lý và phân tích dữ liệu bằng **Pandas** (Phần 3), bước tiếp theo để biến những con số khô khan thành **câu chuyện trực quan** chính là trực quan hóa.

Nếu coi dữ liệu là "nguyên liệu", thì trực quan hóa chính là "ngôn ngữ" giúp chúng ta **nhìn ra xu hướng, so sánh mối quan hệ, và phát hiện điều bất thường** mà các phép tính thuần túy khó làm lộ rõ.

Trong phần này, ta sẽ đi xa hơn các biểu đồ cơ bản, khám phá ba hướng chính:

- **Trực quan hóa đa biến (Multivariate Visualization):** khi một biểu đồ không chỉ kể câu chuyện của một cột dữ liệu, mà của cả nhiều chiều cùng lúc (ví dụ bộ dữ liệu Iris).
- **Phát hiện bất thường qua trực quan hóa (Anomaly Detection):** khi mắt người chính là "bộ lọc anomaly" đầu tiên và mạnh mẽ nhất.
- **Tùy biến và kết hợp biểu đồ:** ghép nhiều góc nhìn vào một canvas để tạo nên bức tranh dữ liệu toàn cảnh.

1. Trực quan hóa đa biến với Iris (Multivariate Visualization)

Bộ dữ liệu **Iris** gồm 150 quan sát với 4 đặc trưng định lượng:

- `sepal_length`, `sepal_width` (chiều dài, rộng đài hoa).
- `petal_length`, `petal_width` (chiều dài, rộng cánh hoa).
- `species` (loài: `setosa`, `versicolor`, `virginica`).

Mục tiêu: khám phá quan hệ giữa các cặp biến, sự khác biệt phân phối theo `species`, và bức tranh đa biến tổng thể.
Gợi ý thực hành: với dữ liệu dày (nhiều điểm chồng) hãy dùng `alpha` nhỏ, `hexbin`, hoặc `kde` để tránh overplotting.

1.1. Chuẩn bị dữ liệu & thiết lập phong cách

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Tai Iris (2 cach: sklearn hoac CSV cua ban)
from sklearn.datasets import load_iris
iris_bunch = load_iris(as_frame=True)
iris = iris_bunch.frame.rename(columns={
    "sepal length (cm)":"sepal_length",
    "sepal width (cm)":"sepal_width",
    "petal length (cm)":"petal_length",
    "petal width (cm)":"petal_width",
    "target":"species_id"
})
iris["species"] = iris["species_id"].map(dict(enumerate(iris_bunch.target_names)))

```

```
# Tuy chon hien thi
sns.set(context="notebook", style="whitegrid")
```

sepal_length	sepal_width	petal_length	petal_width	species_id	species
5.1	3.5	1.4	0.2	0	setosa
4.9	3.0	1.4	0.2	0	setosa
4.7	3.2	1.3	0.2	0	setosa
4.6	3.1	1.5	0.2	0	setosa
5.0	3.6	1.4	0.2	0	setosa

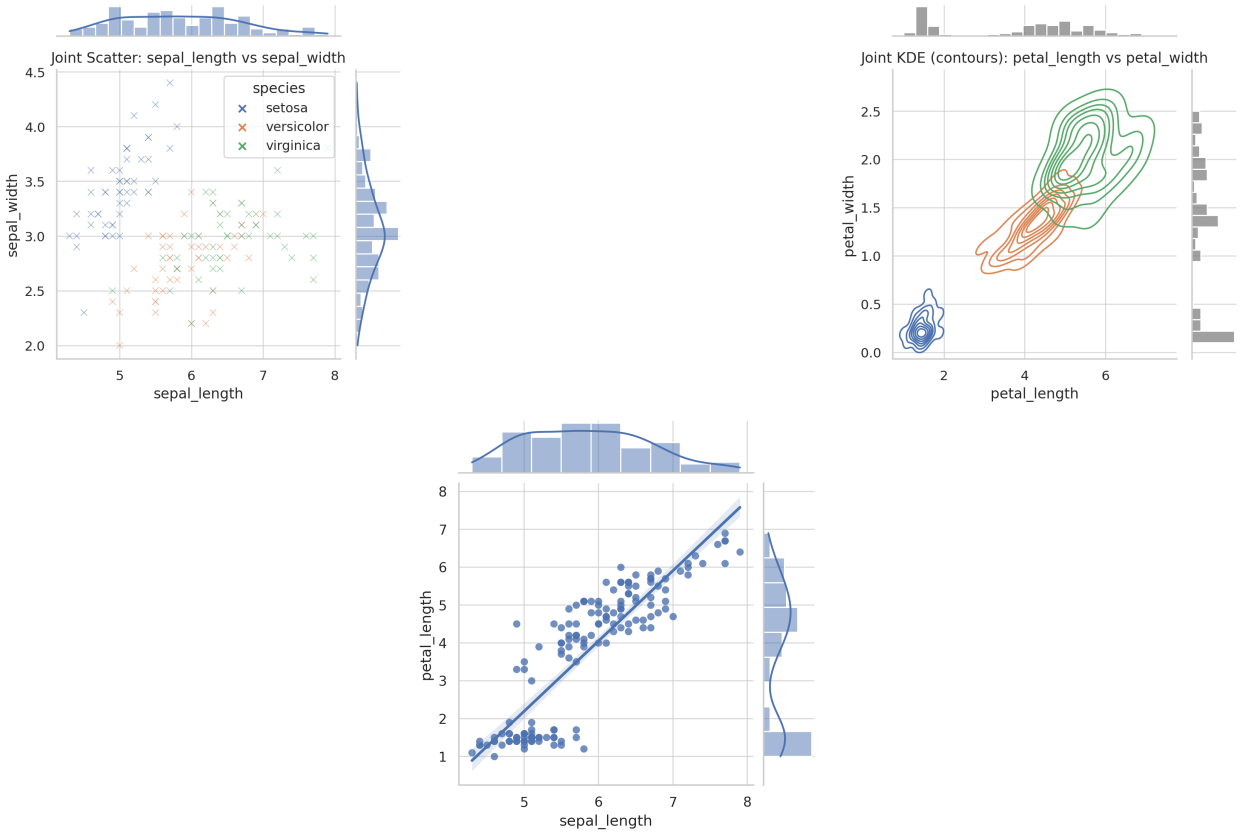
Bảng `head()` 5 dòng đầu của Iris (các cột đã chuẩn hoá tên).

1.2. Jointplot — phân tích tương quan cặp biến (kèm phân bố biên)

Khi dùng jointplot:

- `kind="scatter"`: phân tán cơ bản; thêm `alpha` để giảm chồng điểm.
- `kind="kde"`: đường đẳng mật độ (contour), phù hợp xem cụm; đẹp với `fill=True`.
- `kind="reg"`: hồi quy tuyến tính kèm CI; hữu ích khi kiểm tra xu thế tuyến tính.

```
# 1) Scatter + hue theo species
g = sns.jointplot(
    data=iris, x="sepal_length", y="sepal_width",
    hue="species", kind="scatter", alpha=0.7, height=5
)

# 2) KDE (mat do) de thay cum ro rang
g2 = sns.jointplot(
    data=iris, x="petal_length", y="petal_width",
    hue="species", kind="kde", fill=True, thresh=0.05, height=5
)

# 3) Hoi quy tuyen tinh cho mot cap bien
g3 = sns.jointplot(
    data=iris, x="sepal_length", y="petal_length",
    kind="reg", height=5
)
```



a) Joint Scatter: `sepal_length` vs `sepal_width` theo `species`.
b) Joint KDE (contours): `petal_length` vs `petal_width` cho từng `species`.
c) Joint Regression: `sepal_length` vs `petal_length` (đường hồi quy).

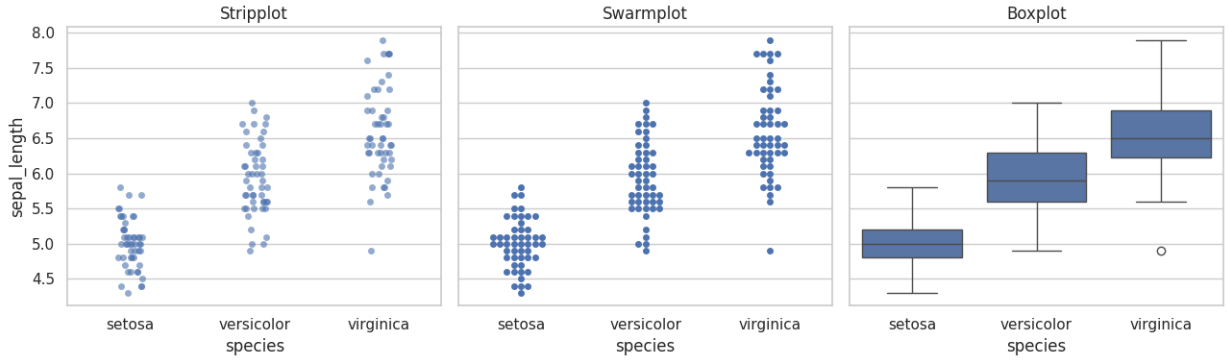
1.3. Stripplot, Swarmplot & Boxplot — so sánh phân bố theo nhóm

Chọn biểu đồ phù hợp:

- `stripplot`: thể hiện từng điểm; dùng `jitter=True`, `alpha<1`.
- `swarmplot`: sắp xếp điểm tránh đè; đẹp nhưng chậm khi dữ liệu rất lớn.
- `boxplot`: tóm tắt phân vị, median, outlier; bỏ chi tiết điểm đơn lẻ.

```
fig, axes = plt.subplots(1, 3, figsize=(13, 4), sharey=True)

sns.stripplot(data=iris, x="species", y="sepal_length",
              alpha=0.6, jitter=True, ax=axes[0])
axes[0].set_title("Stripplot")

sns.swarmplot(data=iris, x="species", y="sepal_length",
              ax=axes[1])
axes[1].set_title("Swarmplot")

sns.boxplot(data=iris, x="species", y="sepal_length",
            ax=axes[2])
```

```
axes[2].set_title("Boxplot")

plt.tight_layout()
```

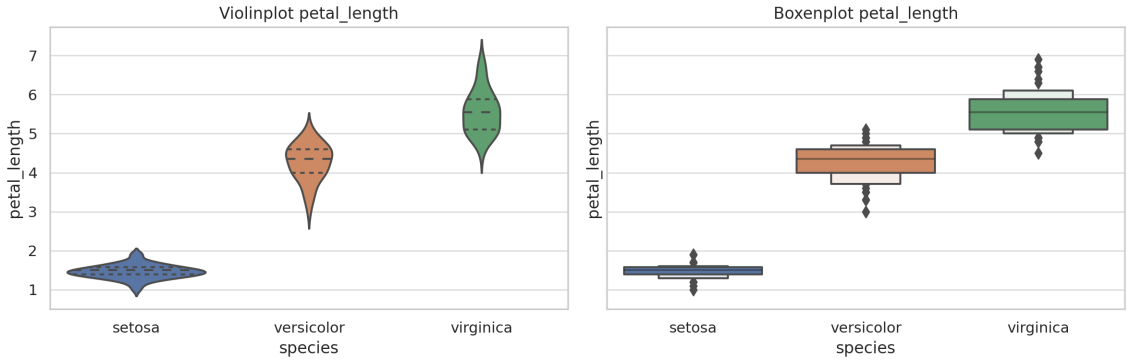


Ba biểu đồ song song (strip, swarm, box) cho `sepal_length` theo `species`.

1.4. Violin & Boxen — hình dạng phân phối & đuôi dày

Violinplot hiển thị ước lượng mật độ kernel (KDE) theo chiều dọc; **boxenplot** (letter-value plot) cho chi tiết phân vị ở đuôi phân phối, hữu ích khi cần xem đuôi dày.

```
fig, axes = plt.subplots(1, 2, figsize=(12,4), sharey=True)
sns.violinplot(data=iris, x="species", y="petal_length",
               inner="quartile", ax=axes[0])
axes[0].set_title("Violinplot petal_length")

sns.boxenplot(data=iris, x="species", y="petal_length",
              ax=axes[1])
axes[1].set_title("Boxenplot petal_length")
plt.tight_layout()
```



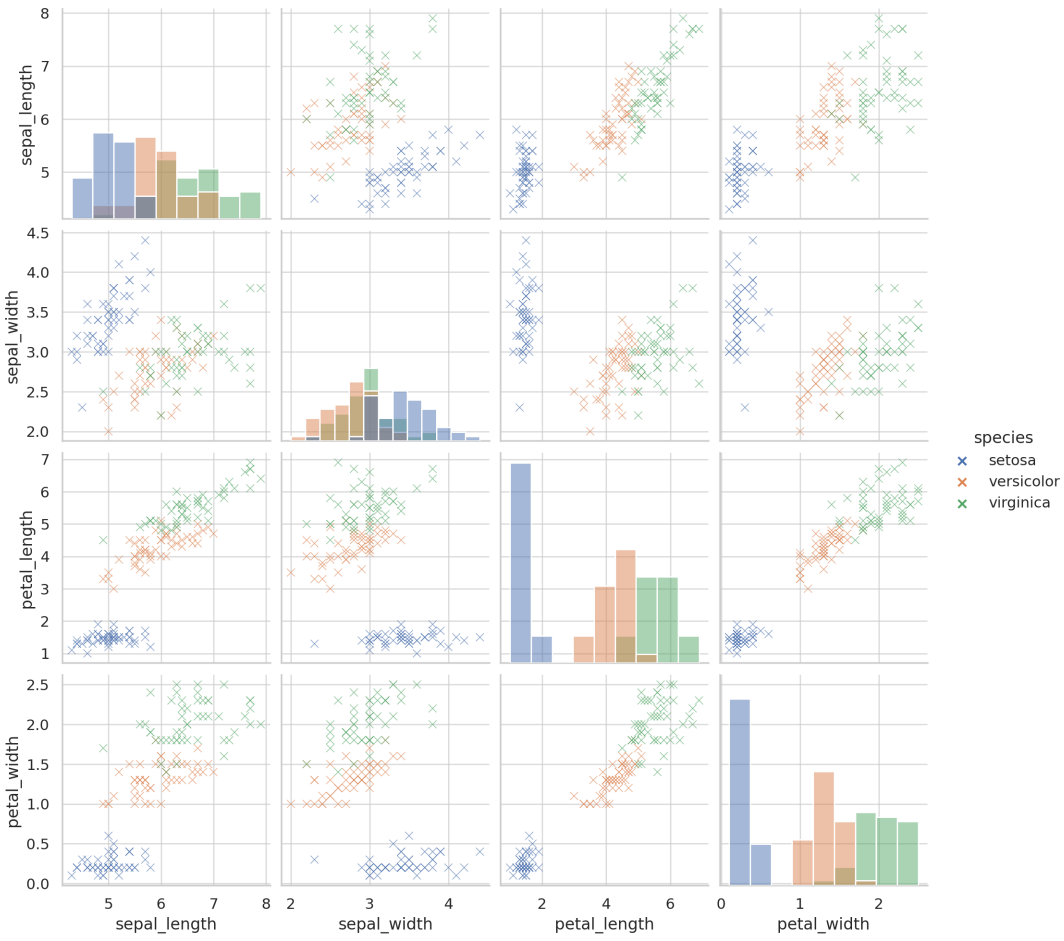
violinplot và boxenplot `petal_length` theo `species`.

1.5. Pairplot & Heatmap — toàn cảnh đa biến

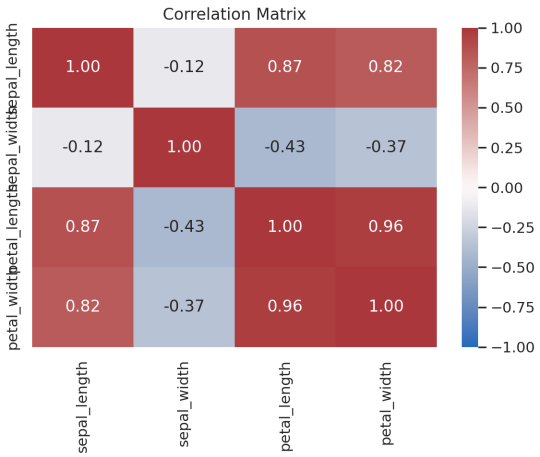
Pairplot tạo lưới scatter/hist cho mọi cặp biến; **heatmap** cho ma trận tương quan.

- Với dữ liệu dày, dùng `diag_kind="kde"`, `plot_kws={"alpha":0.6}`.
- Có thể chọn tập biến con để tập trung vào cặp có ý nghĩa.

```
# Pairplot cho 4 bien, to mau theo species
g = sns.pairplot(
    data=iris,
    vars=["sepal_length","sepal_width","petal_length","petal_width"],
    hue="species", diag_kind="kde",
    plot_kws={"alpha":0.7, "s":35}
)

# Heatmap tuong quan
corr = iris[["sepal_length","sepal_width","petal_length","petal_width"]].corr()
ax = sns.heatmap(corr, annot=True, fmt=".2f", cmap="vlag", vmin=-1, vmax=1)
ax.set_title("Correlation Matrix")
```



Pairplot Iris: 4 biến, phân biệt theo *species*.

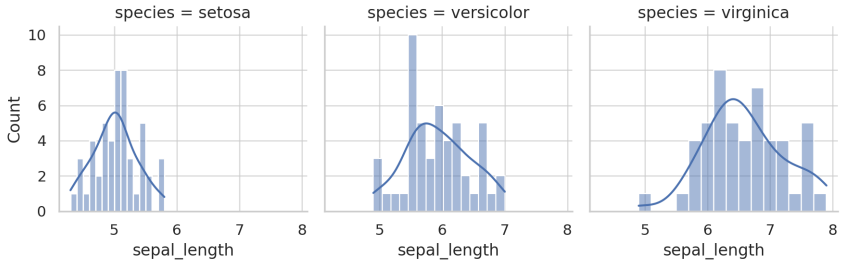


Heatmap ma trận tương quan.

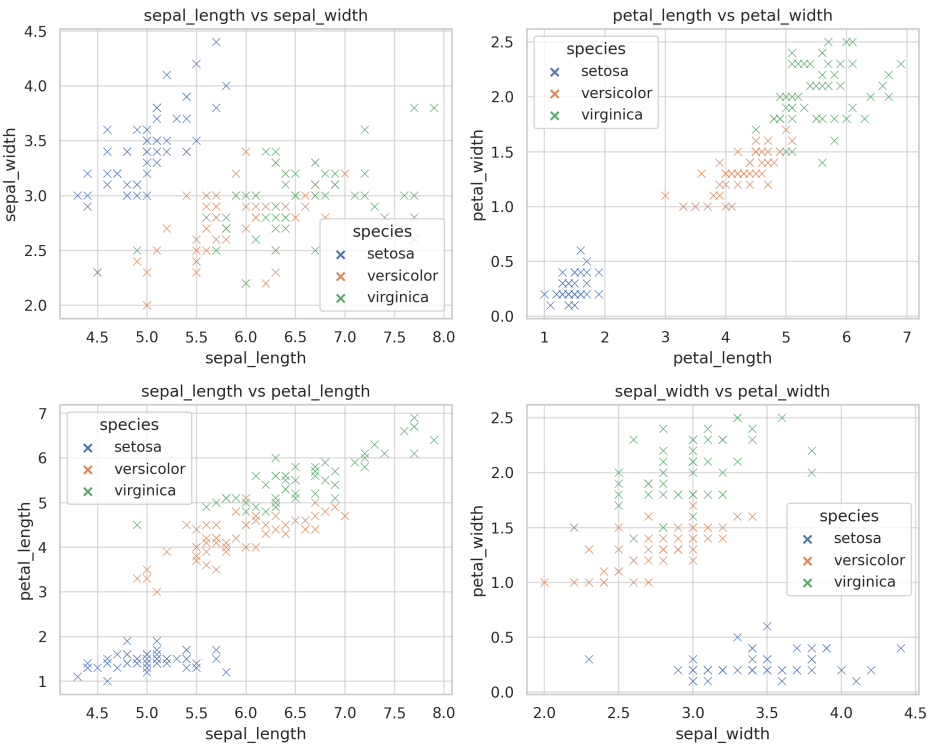
1.6. FacetGrid & Subplots — so sánh theo nhiều điều kiện

Facet chia nhỏ không gian vẽ theo nhóm (như `species`) để so sánh cạnh nhau. Rất hữu ích khi muốn đặt các phân phối/cụm tương ứng ở cùng một tỉ lệ trục.

```
# FacetGrid: histogram phan phoi sepal_length theo species
g = sns.FacetGrid(iris, col="species", sharex=True, sharey=True, height=3)
g.map_dataframe(sns.histplot, x="sepal_length", bins=15, kde=True)

# Subplots tuy bien: 2x2 bon scatter khac nhau
fig, axes = plt.subplots(2, 2, figsize=(10,8), sharex=False, sharey=False)
sns.scatterplot(data=iris, x="sepal_length", y="sepal_width", hue="species", ax=axes
    [0,0])
sns.scatterplot(data=iris, x="petal_length", y="petal_width", hue="species", ax=axes
    [0,1])
sns.scatterplot(data=iris, x="sepal_length", y="petal_length", hue="species", ax=axes
    [1,0])
sns.scatterplot(data=iris, x="sepal_width", y="petal_width", hue="species", ax=axes
    [1,1])
plt.tight_layout()
```



Histogram sepal_length theo species (FacetGrid).

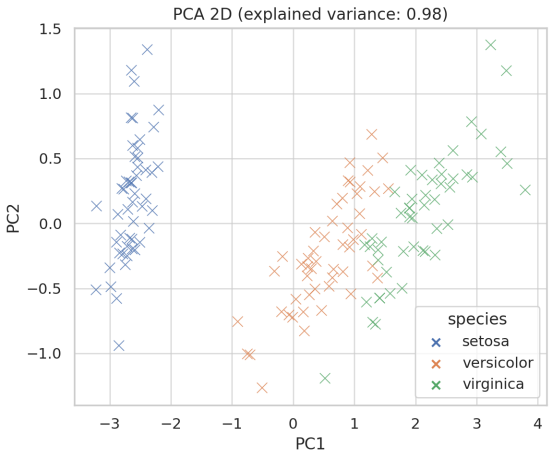


Scatterplot 2x2 cho các cặp biến Iris.

1.7. PCA 2D — "nén chiều" để trực quan hoá cụm

PCA (Principal Component Analysis) giúp chiếu dữ liệu 4 chiều xuống 2D, vừa giảm chiều, vừa tối đa hóa phương sai theo trục mới, để trực quan hoá cụm `species`.

```
from sklearn.decomposition import PCA

X = iris[["sepal_length","sepal_width","petal_length","petal_width"]].values
y = iris["species"].values

pca = PCA(n_components=2, random_state=42)
X2 = pca.fit_transform(X)

pca_df = pd.DataFrame(X2, columns=["PC1","PC2"])
pca_df["species"] = y

ax = sns.scatterplot(data=pca_df, x="PC1", y="PC2", hue="species")
ax.set_title(f"PCA 2D (explained variance: {pca.explained_variance_ratio_.sum():.2f})"
    )
```



scatter PCA 2D tô màu theo `species`

1.8. Thực hành tốt (Best practices) & mẹo trình bày

- **Gắn nhãn** đầy đủ: tiêu đề, trục, đơn vị; legend rõ ràng (`title` của legend nên ngắn).
- **Tránh overplotting**: dùng `alpha`, `markersize` nhỏ, hoặc `hexbin/kde`.
- **So sánh công bằng**: giữ `sharex/sharey` khi đặt nhiều subplot cạnh nhau.
- **Kể chuyện**: mỗi hình nên trả lời một câu hỏi cụ thể (ví dụ: "biến nào phân tách loài tốt nhất?").

2. Phát hiện bất thường qua trực quan hóa (Anomaly Detection)

Tiếp theo, một trong những ưu thế lớn của trực quan hóa dữ liệu là khả năng phát hiện nhanh chóng các quan sát bất thường (**outlier**). Khi nhìn vào biểu đồ, mắt người thường dễ dàng nhận ra các điểm "lạc lõng" so với xu thế hoặc cụm chính của dữ liệu. Đây chính là tín hiệu quan trọng để nhà phân tích chú ý và đặt câu hỏi.

Việc phát hiện bất thường (**outlier detection**) là bước quan trọng, vì outlier có thể:

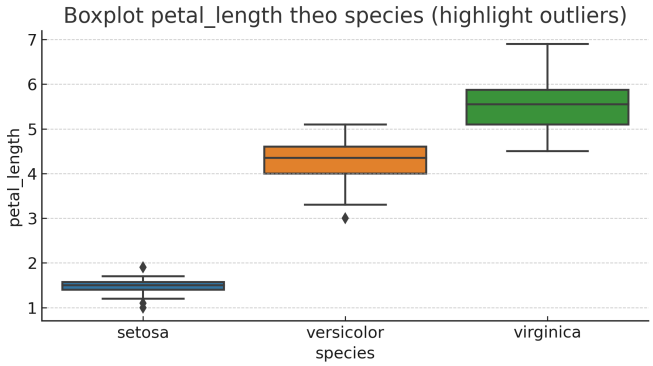
- Là lỗi nhập liệu (ví dụ: gõ nhầm đơn vị hoặc số âm không hợp lý).
- Là các quan sát hiếm nhưng có ý nghĩa (ví dụ: một bệnh nhân đặc biệt, hay một loài hoa lai hiếm).
- Gây ảnh hưởng mạnh đến thống kê và mô hình (trung bình, hồi quy, PCA đều nhạy cảm với outlier).

Trực quan hóa là công cụ nhanh nhất để phát hiện bất thường: bằng mắt thường, ta có thể nhận ra điểm "khác biệt" ngay cả khi các chỉ số thống kê chưa thể hiện rõ.

2.1. Boxplot — tìm điểm nằm ngoài khoảng IQR

Boxplot hiển thị median, phân vị Q1–Q3, và **IQR (Interquartile Range)**. Các điểm nằm ngoài `[Q1 - 1.5*IQR, Q3 + 1.5*IQR]` được coi là outlier.

```
sns.boxplot(x="species", y="petal_length", data=iris)
```



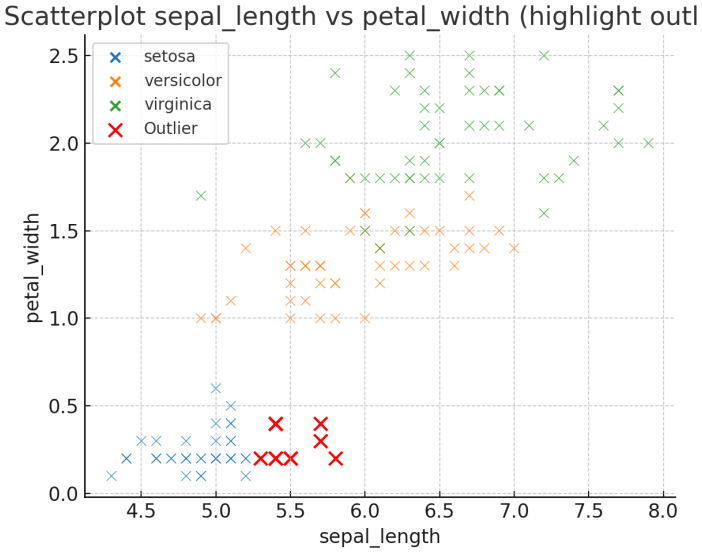
boxplot `petal_length`, highlight outlier.

Ví dụ: với loài *versicolor*, có vài điểm cao bất thường so với cụm chính.

2.2. Scatterplot — nhận diện điểm lệch cụm

Scatterplot cho phép quan sát tương quan hai chiều. Các điểm nằm xa khỏi đám mây điểm (cluster) thường là outlier.

```
sns.scatterplot(x="sepal_length", y="petal_width",
                hue="species", data=iris)
```



scatterplot `sepal_length` vs `petal_width`.

Một số điểm *setosa* có `sepal_length` bất thường dài/rộng hơn cụm chính, có thể là outlier.

2.3. Kết hợp với PCA hoặc Heatmap

Ngoài boxplot và scatter, ta còn có thể dùng:

- **PCA 2D**: trực quan hóa dữ liệu sau khi nén chiều, điểm lệch cụm hiện rõ.
- **Heatmap tương quan**: phát hiện biến nào có mối liên hệ lạ, gợi ý nguyên nhân outlier.

Tóm lại: trực quan hóa giúp phát hiện bất thường dễ dàng, sau đó ta cần quyết định: *xử lý outlier* (loại bỏ, Winsorize, hoặc thay thế) hay *giữ lại* (nếu outlier mang thông tin quan trọng).

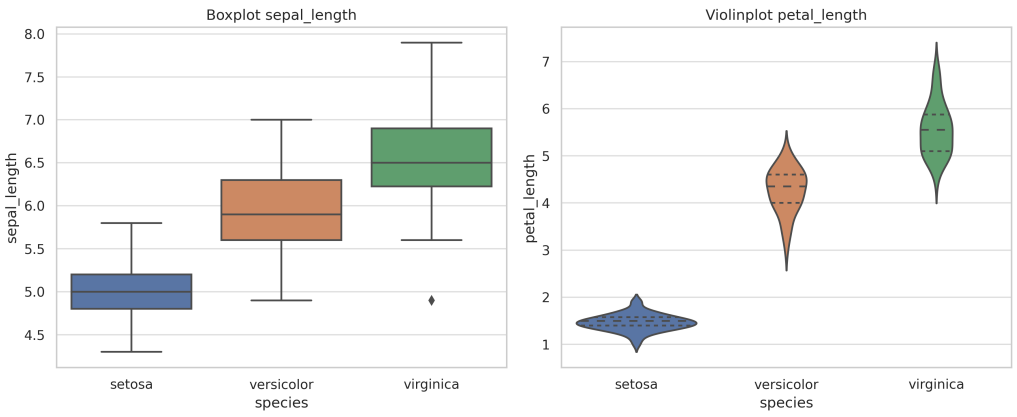
3. Tùy biến và kết hợp nhiều biểu đồ

Trong thực tế, việc trực quan hóa dữ liệu không chỉ dừng lại ở từng biểu đồ đơn lẻ. Để khai thác tối đa thông tin, ta cần kết hợp nhiều loại biểu đồ khác nhau hoặc tuỳ biến chúng sao cho phù hợp với mục đích phân tích.

3.1. Subplot nhiều biểu đồ trong cùng Figure

Ý tưởng: Dùng `subplots()` của `matplotlib` để tạo lưới nhiều trục (axes), sau đó vẽ các loại biểu đồ khác nhau lên từng trục. Điều này giúp so sánh trực quan hai (hoặc nhiều) đặc trưng khác nhau trong cùng một khung hình, thay vì phải mở nhiều biểu đồ rời rạc.

```
import matplotlib.pyplot as plt
fig, axes = plt.subplots(1, 2, figsize=(12,5))

# Boxplot: tom tat phan vi va outlier
sns.boxplot(x="species", y="sepal_length", data=iris, ax=axes[0])
axes[0].set_title("Boxplot sepal_length")

# Violinplot: phan phoi muot va median
sns.violinplot(x="species", y="petal_length", data=iris, ax=axes[1])
axes[1].set_title("Violinplot petal_length")

plt.tight_layout()
```

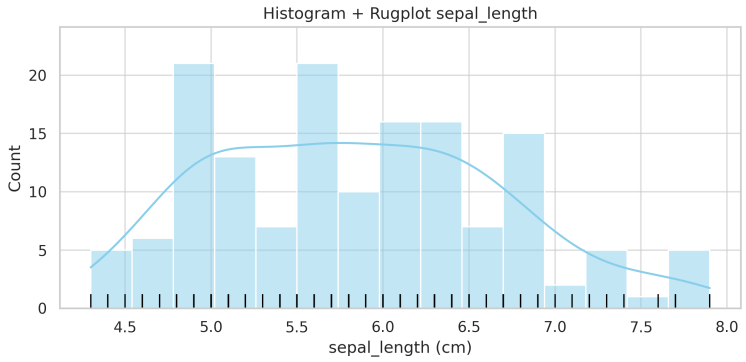


subplot boxplot (sepal_length) + violinplot (petal_length).

3.2. Overlay nhiều lớp thông tin trên cùng biểu đồ

Ý tưởng: Nhiều khi ta muốn biểu đồ hiển thị không chỉ một loại trực quan, mà kết hợp các lớp khác nhau (layering). Ví dụ: histogram để thấy phân phối tổng quan, kết hợp thêm rugplot để hiển thị vị trí từng quan sát gốc.

```
sns.histplot(iris["sepal_length"], kde=True, color="skyblue", bins=15)
sns.rugplot(iris["sepal_length"], color="black")
plt.title("Histogram + Rugplot sepal_length")
```



histogram + rugplot sepal_length (dấu vạch nhỏ dưới trục X).

3.3. Kết hợp nhiều yếu tố thẩm mỹ (Customization)

Ngoài việc kết hợp nhiều loại biểu đồ, ta còn có thể tuỳ biến:

- Màu sắc (`palette`, `hue`) để phân biệt nhóm.
- Chú thích (legend, annotation) để nhấn mạnh outlier hoặc cụm quan trọng.
- Kiểu dáng (linewidth, marker, alpha) để kiểm soát trực quan.

Ví dụ:

```
ax = sns.scatterplot(data=iris, x="sepal_length", y="petal_length",
                     hue="species", style="species", s=60)
ax.axhline(5, ls="--", color="red", alpha=0.7)
ax.set_title("Scatter ővi đường tham ếchiu")
```

Kết quả: Biểu đồ scatter không chỉ phân biệt 3 loài, mà còn có đường ngang (threshold) giúp phân tích rõ hơn.