TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN - TIN HỌC

NHÓM 22 - BÁO CÁO

ĐỒ ÁN CUỐI KÌ NHẬP MÔN PYTHON CHO KHOA HỌC DỮ LIỆU

Thành viên nhóm 22

Họ và tên	**MSSV**
Lưu Nguyễn Ngọc Hân	23280055
Vũ Thị Thanh Hà	23280053
Lê Ái Như	23280024

TP. Hồ Chí Minh, ngày 06 tháng 12 năm 2025

Mục lục

1 Giới thiệu bài toán

1.1 Bối cảnh

Trong bối cảnh cạnh tranh ngày càng gay gắt của ngành F&B, các thương hiệu cần hiểu rõ hành vi, nhu cầu và mức độ nhận thức của khách hàng để xây dựng chiến lược marketing hiệu quả hơn.

Do đó, bài toán **dự đoán phân khúc khách hàng của Highlands Coffee** có ý nghĩa quan trọng. Việc nhận diện đúng phân khúc cho phép thương hiệu cá nhân hoá ưu đãi, tối ưu truyền thông và phân bổ nguồn lực hiệu quả hơn.

Bài toán được xây dựng dựa trên **năm bộ dữ liệu** mô tả toàn diện về khách hàng của Highlands Coffee, gồm:

- **SA#var:** thông tin nhân khẩu học và hành vi tiêu dùng.
- **Segmentation:** phân khúc khách hàng và mức chi tiêu.
- **Brandhealth:** mức độ nhận biết, mức độ sử dụng và đánh giá thương hiệu.
- **Brand Image:** cảm nhận và liên tưởng về hình ảnh thương hiệu.
- **NeedstateDayDaypart:** nhu cầu và lý do ghé quán theo thời điểm trong ngày hoặc ngày trong tuần.

Bài toán mang tính liên ngành, kết hợp các lĩnh vực: phân tích dữ liệu, khai phá dữ liệu, mô hình hóa dự đoán, marketing chiến lược và nghiên cứu hành vi người tiêu dùng.

1.2 Mục tiêu

Mục tiêu của bài toán bao gồm hai nhóm chính: mục tiêu kinh doanh và mục tiêu kỹ thuật.

1. Mục tiêu kinh doanh

- Dự đoán phân khúc khách hàng của Highlands Coffee nhằm hỗ trợ cá nhân hoá ưu đãi và tối ưu hoá chiến dịch marketing.
- Phân tích đặc điểm nhân khẩu học và hành vi của từng phân khúc để hỗ trợ quyết định kinh doanh.
- Nâng cao hiệu quả phân bổ nguồn lực và chiến lược tiếp cận khách hàng.

2. Mục tiêu kỹ thuật

- Làm sạch, chuẩn hóa và tích hợp dữ liệu từ năm bảng dữ liệu khác nhau.
- Xây dựng pipeline tiền xử lý và trích chọn đặc trưng.
- Phát triển kiến trúc chương trình theo hướng hướng đối tượng (OOP), bao gồm các lớp: DataLoader, DataCleaner, FeatureEngineer, ModelTrainer và Evaluator.
- Huấn luyện và thử nghiệm nhiều mô hình dự đoán phân khúc.

- Đánh giá chất lượng phân cụm bằng các chỉ số như Silhouette Score, Davies–Bouldin Index, Calinski–Harabasz Index hay kết hợp theo trọng số giữa các chỉ số kết hợp lại để lựa chọn số cụm và mô hình phân cụm tối ưu.
- Trực quan hóa kết quả và phân tích các yếu tố ảnh hưởng đến phân khúc khách hàng.

1.3 Phạm vi

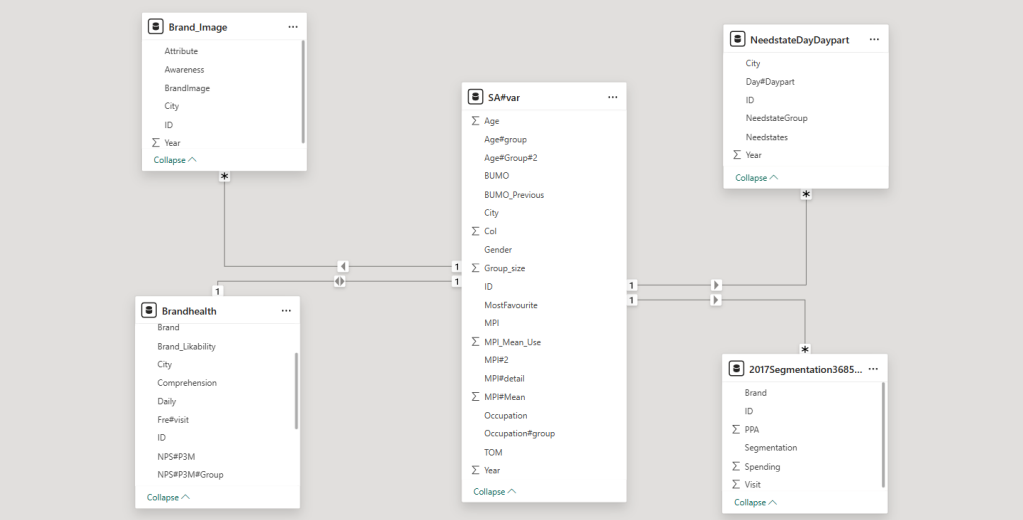
Đề tài được thực hiện trong phạm vi sau:

- Sử dụng Python với các thư viện: *pandas, numpy, matplotlib, seaborn, scikit-learn,....*
- Môi trường chạy: Google Colab, Jupyter Notebook, VisualStudio Code,...
- Tập trung vào các bước: EDA, tiền xử lý dữ liệu, xây dựng mô hình và đánh giá mô hình. Sử dụng các hàm thư viện như `pandas` (xử lý dữ liệu, merge, group), `scikit-learn` (train/test split, scaler, encoder, model training), `matplotlib`/`seaborn` (visualization),... và biết viết class/hàm riêng để tái sử dụng code hiệu quả.
- Không bao gồm triển khai hệ thống, tối ưu nâng cao hoặc xây dựng giao diện người dùng.

2 Giới thiệu dữ liệu

2.1 Nguồn dữ liệu

Dữ liệu được trích xuất từ bộ dữ liệu của cuộc thi *Business Intelligence 9*, gồm các bảng khảo sát khách hàng của Highlands Coffee. Bộ dữ liệu bao gồm thông tin nhân khẩu học, hành vi tiêu dùng, mức độ nhận biết thương hiệu, cảm nhận thương hiệu và nhu cầu theo thời điểm. Dữ liệu được cung cấp dưới dạng tệp CSV và được sử dụng trực tiếp cho mục đích phân tích trong đề tài.



Hình 1: Liên hệ giữa các bảng dữ liệu

2.2 Cấu trúc dữ liệu

Bộ dữ liệu sử dụng trong đề tài là bảng tổng hợp được tạo ra sau khi **merge 5 bộ dữ liệu gốc** (SA#var, Segmentation, Brandhealth, Brand Image, NeedstateDayDaypart). Các nhóm cột chính gồm:

- **Thông tin khách hàng:** `ID`, `City`, `Age`, `Gender`, `Occupation`.
- **Hành vi và nhu cầu:** `Group_size`, `Needstates`, `Day#Daypart`, `Visit`.
- **Nhận biết và mức độ sử dụng thương hiệu:** `TOM`, `BUMO`, `MostFavourite`, `Awareness`, `Trial`, `P3M`, `P1M`.
- **Chỉ số và hình ảnh thương hiệu:** `Brand_Likability`, `BrandImage`, `Attribute`.
- **Chi tiêu và tần suất:** `PPA`, `Spending`, `Fre#visit`, `Weekly`, `Daily`.
- **Phân khúc và nhãn mục tiêu:** `Segmentation`, `Segmentation_seg`.

2.3 Tiền xử lý dữ liệu

Quy trình tiền xử lý được thực hiện riêng cho từng bảng dữ liệu trong bộ khảo sát thương hiệu, bao gồm: *CustomerSegmentation_2017*, *Brand_Image*, *BrandHealth*, *SA#Var* và *Needstate*. Các bước xử lý được mô tả chi tiết như sau.

2.3.1 CustomerSegmentation_2017

- Kiểm tra và loại bỏ các dòng trùng lặp theo `ID`.
- Chuyển kiểu dữ liệu của `ID` sang `object`.
- Chuẩn hoá các giá trị trong `Segmentation` và `Brand`: kiểm tra khác biệt chữ hoa/thường, loại khoảng trắng dư thừa và chuẩn hoá định dạng.
- Kiểm tra tính hợp lý của số lượng nhóm phân khúc (`Segmentation`).
- Chuẩn hoá lại đơn vị `Spending` nếu dữ liệu đang được tính theo nghìn đồng (VND).
- Xác định và đánh dấu các giá trị ngoại lai của `Spending` và `PPA`, nhưng vẫn giữ nguyên để áp dụng log-transform ở giai đoạn mô hình hoá.
- Sửa lỗi chính tả trong cột `Brand`: ?Indepentdent? → ?Independent?.

2.3.2 Brand_Image

- Loại bỏ các dòng trùng hoàn toàn (duplicate rows).
- Chuyển kiểu dữ liệu của `ID` sang `object`.
- Kiểm tra và chuẩn hoá lỗi chính tả trong cột `Attribute`.
- Với cột `Awareness`, có khoảng 19 giá trị khuyết; các giá trị này được điền bằng *"Highlands Coffee"*.
- Gom nhóm (grouping) các thuộc tính (`Attribute`) trong giai đoạn Feature Engineering.

2.3.3 BrandHealth

- Loại bỏ các dòng trùng hoàn toàn.
- Chuyển `ID` về kiểu `object`.
- Xoá 4 dòng có giá trị `Awareness` bị thiếu.
- Chuẩn hoá tên cột:
  - `Fre#visit` → `Frequency_Visit`
  - `NPS#P3M` → `NPS_P3M`
  - `NPS#P3M#Group` → `NPS_P3M_Group`
- Xử lý missing values:
  - **Spontaneous**: điền theo `Awareness`; còn lại điền `None`.
  - **Trial**: điền *"Highlands Coffee"* nếu một trong các biến liên quan (P3M, P1M, Spontaneous, Brand_Likability, Weekly, Daily) mang giá trị Highlands Coffee; ngược lại điền `None`.
  - **P3M**: điền theo P1M, Daily hoặc Weekly; nếu không có thông tin thì điền `None`.
  - **Comprehension**: nếu `Awareness = Highlands Coffee` và `Comprehension` bị thiếu thì điền *"Know a little"*; ngược lại điền `None`.
  - **Frequency_Visit**: điền median nếu khách hàng có tương tác với Highlands Coffee; ngược lại điền 0.
  - **Brand_Likability**: nếu `ID` trùng với bảng Brand_Image thì sử dụng giá trị từ bảng này; hoặc điền bằng `Brand` nếu tần suất ghé thăm lớn hơn trung bình; còn lại điền `None`.
- Kiểm tra trùng lặp giữa `Spending` và `Spending_use`; nếu hai biến giống nhau thì loại bỏ `Spending_use`.
- Với `Spending`: nếu khách không `Trial` Highlands Coffee thì gán 0; nếu không đủ thông tin thì tạm gán -1 cho cả `Spending` và `PPA`.
- Kiểm tra và tính lại `PPA = Spending / Frequency_Visit` nếu khác biệt (bỏ qua các dòng có giá trị -1).
- Điền missing values của `Segmentation` theo ngưỡng:
  - $<$ 25,000 VND: Seg.01 – Mass
  - 25,000–59,000 VND: Seg.02 – Mass Aspiring
  - 60,000–99,000 VND: Seg.03 – Premium
  - $\geq$ 100,000 VND: Seg.04 – Super Premium
  - Spending = -1: Seg.00 – Unknown
- Điền missing values của `NPS_P3M` theo hành vi:

  - Nếu Brand_Likability = Highlands Coffee hoặc Segmentation thuộc nhóm Premium/Super Premium: gán 9.
  - Seg.02: gán 8.
  - Còn lại: gán -1.
- Xác định nhóm `NPS_P3M_Group`:
  - ≥ 9: Promoter
  - $= 8$: Passive
  - < 8: Detractor
- Xử lý ngoại lai (**Winsorization**) cho các biến: `Frequency_Visit`, `Spending`, `NPS_P3M`. Capping tại percentile 99 do tỷ lệ ngoại lai thấp.

2.3.4 SA#Var

- Loại bỏ các dòng thiếu `Group_size` và `Age#group`.
- Loại bỏ các biến: `MPI`, `MPI#2`, `MPI_Mean_Use`.
- Chuẩn hoá văn bản trong các biến `City`, `Occupation#group`, `Occupation`, `TOM`, `BUMO`, `MostFavourite`: loại bỏ khoảng trắng dư và sửa lỗi chính tả (ví dụ: `Indepedent Cafe` → `Independent Cafe`).
- Loại bỏ biến `Age#group` (trùng với `Age#Group#2`).
- Mapping thu nhập `MPI#Mean` sang `MPI#detail` theo ngưỡng giá trị.
- Với phần còn lại của `MPI#detail` bị thiếu → điền *"Unknown"*. Sau đó loại bỏ `MPI#Mean`.
- Với `BUMO_Previous`: giá trị thiếu được điền thành *"Unknown"*.

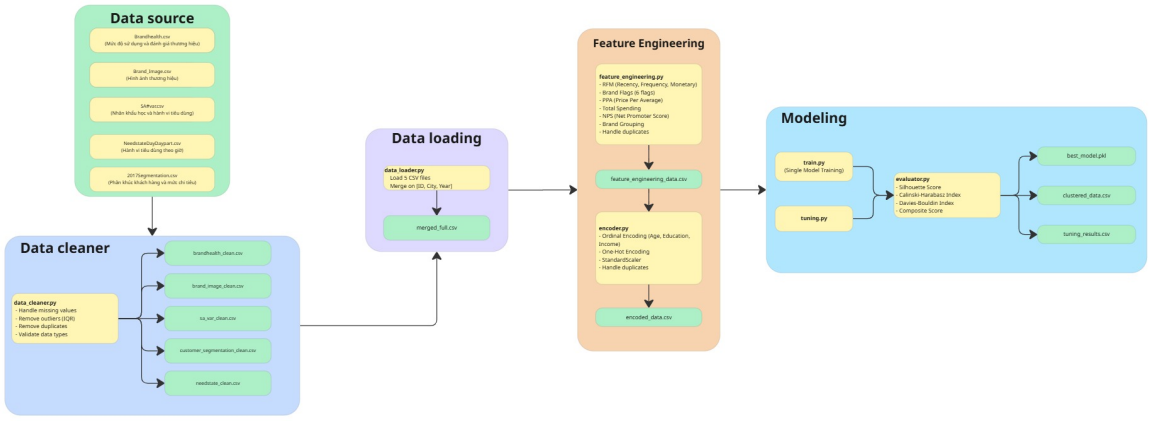
2.3.5 Needstate

- Chuyển `ID` sang kiểu `object`.
- Kiểm tra độ dài bất thường của `ID`; loại bỏ các giá trị sai định dạng.
- Loại bỏ các dòng trùng lặp theo toàn bộ hàng.
- Đổi tên cột `Day#Daypart` thành `Daypart`.
- Chuẩn hoá lỗi chính tả trong `Needstates`:
  - `Socialzing` → `Socializing`
  - `Enterntainment (watching movies, ...)` → `Entertainment`
- Chuẩn hoá nhóm `NeedstateGroup`, ví dụ:
  - `Relaxing & entertainment` → `Relaxing & Entertainment`
  - `Working & business meeting` → `Working & Business meeting`

3 Giới thiệu các Class và phương thức

3.1 Tổng quan kiến trúc code

Hệ thống mã nguồn được tổ chức theo mô hình module hoá, nhằm tối ưu tính rõ ràng, khả năng bảo trì và khả năng mở rộng. Toàn bộ dự án được chia thành các khối chức năng chính bao gồm: *quản lý dữ liệu*, *tiền xử lý*, *xây dựng mô hình*, *trực quan hoá* và *tiện ích hệ thống*. Cách tổ chức này đảm bảo quy trình xử lý dữ liệu và huấn luyện mô hình diễn ra một cách mạch lạc, đồng thời hỗ trợ khả năng tái sử dụng trong tương lai.



Hình 2: Flowchart Pipeline

(1) Khối dữ liệu (data/) Thư mục dữ liệu được chia thành ba nhóm:

- **raw/**: lưu trữ toàn bộ dữ liệu gốc (SA_var, Needstate, BrandHealth, Brand_Image, Segmentation). Các file này không được chỉnh sửa trực tiếp.
- **processed/**: lưu các dataset đã được xử lý và merge. Đây là dữ liệu đầu vào cho mô hình.

(2) Khối tiền xử lý (src/preprocessing/) Khối này bao gồm các class chịu trách nhiệm làm sạch, hợp nhất và chuẩn hoá dữ liệu:

- **DataLoader**: đọc từng bảng dữ liệu và merge theo ID – Year – City.
- **DataCleaner**: xử lý thiếu dữ liệu, chuẩn hoá text, khử nhiễu, xử lý ngoại lai, đổi tên cột, chuẩn hoá định dạng.
- **Encoder**: mã hoá các biến danh mục bằng One-hot, Label hoặc Target Encoding.
- **FeatureEngineering**: tạo biến mới phục vụ mô hình như nhóm tuổi, tầng thu nhập, nhóm phân khúc hoặc cờ hành vi.

Các module này tạo thành pipeline tiền xử lý hoàn chỉnh, đảm bảo dữ liệu đầu vào mô hình được thống nhất và phù hợp cho học máy.

(3) Khối mô hình (src/models/) Bao gồm các class phục vụ cho quá trình huấn luyện, đánh giá và tối ưu mô hình phân cụm.Chức năng lưu/tải mô hình được tích hợp trực tiếp trong từng class.

- **ModelTrainer**: xây dựng và huấn luyện các mô hình phân cụm (KMeans, GMM, DBSCAN, HDBSCAN), sinh nhãn phân cụm và hỗ trợ lưu/tải mô hình đã huấn luyện.
- **ClusteringEvaluator**: tính toán các chỉ số đánh giá chất lượng phân cụm (Silhouette, Calinski–Harabasz, Davies–Bouldin), đồng thời hỗ trợ lưu/tải kết quả đánh giá khi cần.
- **HyperparameterTuner**: sinh toàn bộ tổ hợp siêu tham số, huấn luyện và đánh giá mô hình, lựa chọn mô hình tối ưu và lưu lại cấu hình tốt nhất để sử dụng về sau.

Nhờ cấu trúc này, toàn bộ quy trình phân cụm được tổ chức rõ ràng, linh hoạt và dễ dàng tái lập mà không cần một class lưu/tải độc lập.

(4) Khối trực quan hoá (src/visualization/) Chịu trách nhiệm tạo biểu đồ phục vụ phân tích và trình bày:

- trực quan EDA (histogram, boxplot, heatmap),
- trực quan mô hình (ROC, confusion matrix, feature importance),
- biểu đồ giải thích mô hình (SHAP).

Các biểu đồ được tự động lưu vào thư mục `reports/figures/`.

(5) Quy trình pipeline tổng quát Toàn bộ hệ thống phân cụm được vận hành theo trình tự sau:

1. **Nạp dữ liệu**: đọc và hợp nhất các bảng khảo sát thông qua `DataLoader`.
2. **Tiền xử lý và chuẩn hoá**: làm sạch, chuyển kiểu dữ liệu, xử lý missing và chuẩn hoá văn bản bằng `DataCleaner`.
3. **Mã hoá và tạo đặc trưng**: biến đổi dữ liệu định tính/định lượng, sinh đặc trưng hành vi thông qua `Encoder` và `FeatureEngineering`.
4. **Huấn luyện mô hình phân cụm**: khởi tạo mô hình tương ứng (KMeans, GMM, DBSCAN, HDBSCAN) và sinh nhãn phân cụm bằng `ModelTrainer`.
5. **Đánh giá chất lượng phân cụm**: tính các chỉ số Silhouette, Calinski–Harabasz, Davies–Bouldin bằng `ClusteringEvaluator`.
6. **Tuning tham số**: thực thi toàn bộ tổ hợp tham số, so sánh kết quả và chọn mô hình tối ưu bằng `HyperparameterTuner`.
7. **Xuất kết quả**: tổng hợp bảng đánh giá, lưu mô hình tốt nhất và dataset đã gán nhãn để phục vụ phân tích tiếp theo.

3.2 Luồng hoạt động giữa các class

Quy trình được tổ chức theo dạng pipeline gồm hai phần chính: (1) xử lý dữ liệu đầu vào, (2) huấn luyện và tối ưu mô hình phân cụm. Các class tương tác tuần tự theo luồng như sau:

3.2.1 Khối xử lý dữ liệu

1. **Class DataLoader** Đọc và nạp dữ liệu thô từ nhiều nguồn vào `DataFrame`.
2. **Class DataCleaner** Làm sạch từng bảng dữ liệu, chuẩn hoá giá trị thiếu, sai định dạng và tiến hành ghép nối các bảng thành một dataset hoàn chỉnh phục vụ mô hình.
3. **Class FeatureEngineering** Sinh thêm các đặc trưng hành vi và marketing quan trọng, bao gồm đặc trưng thương hiệu, loyalty, switching, funnel depth và các chỉ số hành vi khác.
4. **Class FeatureEncoder** Tiền xử lý cuối cùng trước mô hình: mã hoá, scale, log-transform, chuẩn hoá dữ liệu. Class này sử dụng thêm:
   - **EncoderConfig**: cấu hình scaler và format đầu ra.
   - **LogTransformer**: thực hiện chuyển đổi log cho các biến cần thiết.

   Sau bước này, dữ liệu được chuyển về ma trận đầu vào $\mathbf{X}$ để sẵn sàng cho pipeline huấn luyện mô hình phân cụm.

3.2.2 Khối mô hình: Train – Evaluate – Tuning

1. **Class ModelTrainer** Nhận dữ liệu từ FeatureEncoder, xây dựng mô hình phân cụm tương ứng với cấu hình (`TrainingConfig`), huấn luyện và sinh ra nhãn phân cụm.
2. **Class ClusteringEvaluator** Đánh giá mô hình bằng các thước đo như Silhouette, Calinski–Harabasz và Davies–Bouldin. Kết quả mỗi lần đánh giá được lưu để phục vụ so sánh trong quá trình tuning.
3. **Class HyperparameterTuner** Là trung tâm điều phối cho quá trình thử nghiệm mô hình:
   - nạp dữ liệu để tuning,
   - sinh toàn bộ tổ hợp tham số (*grid search*),
   - huấn luyện mô hình qua `ModelTrainer`,
   - đánh giá qua `ClusteringEvaluator`,
   - lưu lại kết quả và cập nhật mô hình tốt nhất.

   Tuner thực thi lần lượt nhiều mô hình (`KMeans`, `GMM`, `DBSCAN`, `HDBSCAN`) và lựa chọn mô hình tối ưu theo chỉ số được cấu hình. Cuối cùng, HyperparameterTuner có trách nhiệm:
   - tổng hợp bảng kết quả,
   - lưu kết quả ra file,
   - lưu mô hình tốt nhất và dataset đã gán nhãn.

3.3 Mô tả từng Class

3.3.1 Class DataLoader

Class `DataLoader` chịu trách nhiệm đọc dữ liệu từ thư mục `data/raw/`, quản lý nhiều bảng dữ liệu và cung cấp các phương thức hỗ trợ hợp nhất các bảng theo ID, Year, City.

- **Phương thức** `__init__()`
  - Khởi tạo lớp với đường dẫn dữ liệu (raw hoặc cleaned), thiết lập danh sách tên file và separator tương ứng, đồng thời tạo các cấu trúc lưu trữ như `self.data` và `self.merged_df`.
- **Phương thức** `_read_csv(file_name, key)`
  - **Công dụng**: Đọc file CSV từ thư mục dữ liệu thô/cleaned, kiểm tra file có tồn tại, nạp nội dung bằng `pandas.read_csv()`, ghi log quá trình đọc và quản lý dữ liệu trong lớp.
  - **Input**:
    - `file_name`: tên file CSV cần đọc.
    - `key`: tên định danh dùng để lưu DataFrame vào `self.data`.
  - **Output**:
    - Trả về một `pandas DataFrame` chứa dữ liệu đã đọc.
    - Đồng thời lưu DataFrame này vào `self.data[key]`.
- **Phương thức** `load_all()`
  - **Công dụng**:
    - Tải tất cả file CSV đã định nghĩa và lưu vào bộ nhớ dưới dạng dictionary.
    - Cung cấp dữ liệu đã tải để sử dụng cho các bước xử lý tiếp theo.
  - **Input**: Không nhận tham số đầu vào (ngoại trừ `self`).
  - **Output**: Trả về một `Dict[str, DataFrame]` chứa tất cả DataFrame đã đọc từ các file CSV.
  - **Ngoại lệ**: Ném `FileNotFoundError` nếu bất kỳ file CSV bắt buộc nào không tồn tại.
- **Phương thức** `merge_all()`
  - **Công dụng**: Hợp nhất tất cả bảng thành một DataFrame duy nhất, chuẩn hóa và điền giá trị thiếu, lưu vào `self.merged_df`.
  - **Input**: Không có tham số ngoài `self`; tự gọi `load_all()` nếu dữ liệu chưa nạp.
  - **Output**: DataFrame đã gộp từ các bảng.
  - **Quy trình xử lý**:
    1. Lọc `SA_var` năm 2017 và kiểm tra cột bắt buộc.

2. Loại trùng, bỏ cột không cần thiết.
3. Merge tuần tự các bảng theo `ID`.
4. Làm sạch dữ liệu: xoá cột trùng, chuẩn hóa tên, điền NaN/giá trị mặc định, xoá `IncomeBand`.

– **Ngoại lệ**: Ném `ValueError` nếu bảng thiếu cột bắt buộc.

- **Phương thức** `save_merged(out_path=None)`
  - **Công dụng**: Lưu DataFrame đã được hợp nhất vào file CSV, tự động tạo thư mục nếu chưa tồn tại và ghi log quá trình lưu.
  - **Input**: `out_path` (tùy chọn): Đường dẫn file CSV đầu ra.Nếu không truyền, phương thức sẽ sử dụng đường dẫn mặc định từ `config.py`.
  - **Output**: Không trả về giá trị. Ghi DataFrame đã merge vào file CSV trên ổ đĩa.
  - **Quy trình xử lý**:
    * Kiểm tra xem đã có dữ liệu merge trong `self.merged_df` hay chưa.
    * Tạo thư mục đầu ra nếu chưa tồn tại.
    * Ghi toàn bộ bảng dữ liệu đã merge xuống file CSV.
    * Ghi log quá trình lưu file.
  - **Ngoại lệ**: `ValueError`: Ném ra khi `merge_all()` chưa được gọi nên không có dữ liệu để lưu.
- **Phương thức** `get_merged_info()`
  - **Công dụng**: Kiểm tra sự tồn tại của `self.merged_df`, sau đó trả về kích thước của DataFrame dưới dạng `shape`.
  - **Input**: Không có tham số đầu vào.
  - **Output**: Trả về một tuple `(n_rows, n_cols)` biểu thị số dòng và số cột của bảng đã merge.
  - **Ngoại lệ**: `ValueError` — nếu chưa gọi `merge_all()` và không có dữ liệu merge.

3.3.2 Class DataCleaner

Class `DataCleaner` chịu trách nhiệm làm sạch và tiền xử lý từng bảng dữ liệu khảo sát.

- **Phương thức** `__init__()`
  - **Công dụng**: Khởi tạo đối tượng `DataCleaner`. Phương thức này không nhận tham số và không thiết lập thuộc tính nào, đóng vai trò tạo instance để sử dụng các hàm làm sạch dữ liệu phía sau.
- **Phương thức** `_normalize_string_columns()`

- **Công dụng**: Chuẩn hoá các cột dạng chuỗi bằng cách loại bỏ khoảng trắng thừa.
- **Input**: DataFrame gốc và danh sách tên cột cần chuẩn hoá.
- **Output**: DataFrame mới với các cột string đã được làm sạch, không làm đổi dữ liệu gốc.
- **Tóm tắt xử lý**: Sao chép DataFrame, kiểm tra từng cột trong danh sách, chuyển sang kiểu chuỗi và áp dụng `strip()`.

- **Phương thức `_winsorize_series()`**
  - **Công dụng**: Giảm ảnh hưởng outlier bằng cách giới hạn giá trị Series trong khoảng giữa hai quantile.
  - **Input**: Series cần xử lý và ngưỡng quantile dưới/trên.
  - **Output**: Series mới đã được winsorize.
  - **Tóm tắt xử lý**: Tính hai quantile và dùng `clip()` để chặn giá trị vượt ngoài khoảng $[lower\_q, upper\_q]$.

- **Phương thức `_map_income_band_from_mean()`**
  - **Công dụng**: Chuyển giá trị thu nhập trung bình (`float`) sang nhóm thu nhập chuẩn hoá (`income band`) để dễ phân tích và trực quan hoá.
  - **Input**: `x` – giá trị thu nhập trung bình (số dương hoặc `NaN`).
  - **Output**: `str` – nhóm thu nhập nếu hợp lệ; `None` nếu `x` là `NaN` hoặc không thuộc khoảng nào.
  - **Tóm tắt xử lý**: Kiểm tra giá trị đầu vào và so sánh với các khoảng thu nhập đã định nghĩa để trả về nhãn tương ứng; nếu không thuộc khoảng nào, trả về `None`.

- **Phương thức `_fix_brand_spelling()`**
  - **Công dụng**: Chuẩn hóa và sửa lỗi chính tả phổ biến trong tên thương hiệu.
  - **Input**: `value` – chuỗi tên brand (`str`), có thể là `NaN`.
  - **Output**: `str` – tên đã sửa nếu có lỗi; giữ nguyên nếu không; trả về `value` nếu `NaN`.
  - **Tóm tắt xử lý**: Kiểm tra giá trị đầu vào, đối chiếu với bảng lỗi chính tả, thay thế nếu trùng, giữ nguyên nếu không.

- **Phương thức `clean_customer_segmentation()`**
  - **Công dụng**: Làm sạch và chuẩn hoá dữ liệu Customer Segmentation.
  - **Input**: DataFrame gốc.
  - **Output**: DataFrame đã được chuẩn hoá, loại trùng và thống nhất định dạng.
  - **Tóm tắt xử lý**: Sao chép dữ liệu, loại bỏ trùng lặp, chuẩn hoá ID và các cột văn bản, sửa lỗi thương hiệu và trả về dữ liệu sạch.

- **Phương thức** `clean_brand_image()`
  - **Công dụng**: Làm sạch bảng Brand Image.
  - **Input**: DataFrame gốc.
  - **Output**: DataFrame nhất quán, không trùng và không thiếu giá trị quan trọng.
  - **Tóm tắt xử lý**: Loại bỏ trùng lặp, chuẩn hoá văn bản, xử lý giá trị thiếu cho `Awareness` và trả về dữ liệu đã chuẩn hóa.
- **Phương thức** `clean_brandhealth()`
  - **Công dụng**: Làm sạch toàn diện bảng Brand Health.
  - **Input**: DataFrame gốc, tùy chọn bảng Brand Image.
  - **Output**: DataFrame thống nhất cấu trúc, hoàn chỉnh và sẵn sàng phân tích funnel/NPS.
  - **Tóm tắt xử lý**: Chuẩn hoá ID, xử lý các biến funnel, điền giá trị thiếu bằng logic nội bộ và dữ liệu liên bảng, chuẩn hoá tần suất – chi tiêu – segmentation và giảm nhiễu cho biến số.
- **Phương thức** `clean_sa_var()`
  - **Công dụng**: Làm sạch bảng SA Variables.
  - **Input**: DataFrame gốc.
  - **Output**: DataFrame đã chuẩn hoá cấu trúc, text và các biến nhân khẩu.
  - **Tóm tắt xử lý**: Chuẩn hoá tên cột, xoá biến dư thừa, chuẩn hoá văn bản, sửa lỗi thương hiệu và xây dựng lại biến thu nhập phục vụ phân tích.
- **Phương thức** `clean_needstate()`
  - **Công dụng**: Làm sạch bảng Needstate.
  - **Input**: DataFrame gốc.
  - **Output**: DataFrame đã chuẩn hoá và không còn lỗi định dạng.
  - **Tóm tắt xử lý**: Chuẩn hoá ID, loại bản ghi bất thường, sửa lỗi text các cột needstate và hoàn thiện dữ liệu cho phân tích nhu cầu.
- **Phương thức** `clean_all()`
  - **Công dụng**: Thực hiện toàn bộ quy trình làm sạch cho tất cả bộ dữ liệu.
  - **Input**: `dfs` – `Dict[str, pd.DataFrame]` với các key: `'customer_seg'`, `'brand_image'`, `'brandhealth'`, `'sa_var'`, `'needstate'`.
  - **Output**: `Dict[str, pd.DataFrame]` chứa datasets đã làm sạch.
  - **Xử lý**: Lần lượt gọi các hàm: `clean_customer_segmentation`, `clean_brand_image`, `clean_brandhealth`, `clean_sa_var`, `clean_needstate`, đồng bộ brand image với brandhealth và trả về dictionary mới.

- **Phương thức** `save_cleaned()`
  - **Công dụng**: Lưu các DataFrame đã làm sạch ra file CSV, tự tạo thư mục nếu chưa tồn tại.
  - **Input**: `dfs_clean` – `Dict[str, pd.DataFrame]` đã làm sạch; `base_dir` – đường dẫn thư mục lưu file (mặc định: `data/processed/cleaned`).
  - **Output**: Không trả về giá trị; ghi file ra ổ đĩa.
  - **Mapping tên file**: `customer_seg` → `customer_segmentation_clean.csv`, `brand_image` → `brand_image_clean.csv`, `brandhealth` → `brandhealth_clean.csv`, `sa_var` → `sa_var_clean.csv`, `needstate` → `needstate_clean.csv`.
  - **Xử lý**: Kiểm tra/thêm thư mục, lặp qua DataFrame, xác định tên file, lưu CSV, ghi log xác nhận.

3.3.3 Class FeatureEngineering

Class `FeatureEngineering` chịu trách nhiệm tạo ra các biến dẫn xuất phục vụ cho việc phân cụm (clustering) trong bộ dữ liệu khách hàng của Highlands Coffee.

- **Phương thức** `__init__(df)`
  - **Công dụng**: Khởi tạo đối tượng `FeatureEngineering` với dữ liệu đầu vào, danh sách brand giữ lại và mapping flag cho Highlands Coffee.
  - **Input**: `df`: `pd.DataFrame` dữ liệu khảo sát đã làm sạch.
  - **Thuộc tính khởi tạo**:
    - `self.df` — dữ liệu gốc.
    - `self.keep` — danh sách brand giữ lại: `"Independent Cafe"`, `"Street / Half street coffee"`, `"Highlands Coffee"`, `"Trung Nguyên"`.
    - `self.flag_map` — mapping các cột hành vi sang flag tương ứng (ví dụ: `"Awareness"` → `"Awareness_flag"`).
  - **Output**: Không trả về, chỉ thiết lập thuộc tính nội bộ.
- **Phương thức** `_add_brand_grouping()`
  - **Công dụng**: Gom các giá trị thương hiệu thành hai nhóm: thuộc `self.keep` và "Others".
  - **Input**: Không nhận input, sử dụng `self.df` đã khởi tạo.
  - **Output**: Trả về `self` với các cột nhóm mới: `TOM_Group`, `BUMO_Group`, `BUMO_Previous_Group`, `MostFavourite_Group`.
  - **Quy trình xử lý**: Kiểm tra từng giá trị; giữ nếu thuộc `self.keep`, gán "Others" nếu không; tạo các cột nhóm.
- **Phương thức** `_add_brand_flags()`
  - **Công dụng**: Tạo các biến flag (0/1) cho Highlands Coffee dựa trên các cột funnel.

- **Input**: Không nhận input; sử dụng `self.df` và `self.flag_map`.
- **Output**: Trả về `self` với các cột flag mới: `Awareness_flag`, `Spontaneous_flag`, `Trial_flag`, `P3M_flag`, `P1M_flag`, `Weekly_flag`, `Daily_flag`.
- **Quy trình xử lý**: Lặp qua `self.flag_map`; gán 1 nếu giá trị bằng "Highlands Coffee", ngược lại 0; tạo các cột flag mới.

- **Phương thức `_add_brand_loyalty()`**
  - **Công dụng**: Tạo điểm trung thành thương hiệu (0–4) dựa trên mức độ nhất quán giữa các lựa chọn thương hiệu.
  - **Input**: Không nhận input; sử dụng `self.df` đã có các cột nhóm thương hiệu.
  - **Output**: Trả về `self` với cột mới `Brand_Loyalty` (0–4).
  - **Quy trình xử lý**: So sánh sự trùng khớp giữa các cột nhóm thương hiệu (`TOM_Group`, `BUMO_Group`, `MostFavourite`, `MostFavourite_Group`, `BUMO_Previous_Group`); mỗi trùng khớp được 1 điểm; tổng điểm tạo `Brand_Loyalty`.
- **Phương thức `_add_brand_switcher()`**
  - **Công dụng**: Tạo biến `Brand_Switcher` nhận diện khách hàng thay đổi thương hiệu sử dụng chính.
  - **Input**: Không nhận input; sử dụng `self.df` với các cột `BUMO_Group` và `BUMO_Previous_Group`.
  - **Output**: Trả về `self` với cột `Brand_Switcher` (0/1).
  - **Quy trình xử lý**: So sánh `BUMO_Group` và `BUMO_Previous_Group`; khác nhau gán 1, giống nhau gán 0; tạo cột mới.
- **Phương thức `_add_funnel_depth()`**
  - **Công dụng**: Tính `Funnel_Depth` (0–7) dựa trên tổng số bước funnel mà khách hàng đã đạt.
  - **Input**: Không nhận input; sử dụng `self.df` với các cột flag từ `self.flag_map`.
  - **Output**: Trả về `self` với cột `Funnel_Depth`.
  - **Quy trình xử lý**: Cộng các flag theo hàng để xác định độ sâu funnel; tạo cột mới.
- **Phương thức `_add_awareness_usage_gap()`**
  - **Công dụng**: Tạo cột `Awareness_Usage_Gap` đo khoảng cách giữa biết đến và đã từng dùng.
  - **Input**: Không nhận input; sử dụng `self.df` với `Awareness_flag` và `Trial_flag`.
  - **Output**: Trả về `self` với cột mới `Awareness_Usage_Gap`.
  - **Quy trình xử lý**: Lấy hiệu `Awareness_flag - Trial_flag` theo hàng; tạo cột mới.
- **Phương thức `_add_need_is_drinking()`**

- **Công dụng**: Tạo cột `Need_is_Drinking` xác định nhu cầu chính liên quan đến "Drinking".
- **Input**: Không nhận input; sử dụng `self.df` với cột `Needstates`.
- **Output**: Trả về `self` với cột mới `Need_is_Drinking`.
- **Quy trình xử lý**: Kiểm tra chuỗi `Needstates` có chứa "Drinking"; gán 1 nếu có, 0 nếu không; tạo cột mới.

- **Phương thức** `build_all()`
  - **Công dụng**: Thực hiện toàn bộ các bước feature engineering và trả về DataFrame đã xử lý hoàn chỉnh.
  - **Input**: Không nhận input; sử dụng `self.df` đã khởi tạo.
  - **Output**: DataFrame cuối cùng (`self.df`) với tất cả các cột feature mới.
  - **Quy trình xử lý**: Lần lượt gọi các hàm tạo nhóm thương hiệu, flag funnel, điểm trung thành, Brand Switcher, độ sâu funnel, khoảng cách Awareness–Usage, và biến nhu cầu "Drinking"; mỗi hàm cập nhật `self.df` và hỗ trợ method chaining.

3.3.4 Class LogTransformer

- **Công dụng**: Custom transformer thực hiện `log1p` cho các biến numeric skewed (`Visit`, `Spending`, `PPA`) để giảm ảnh hưởng ngoại lai, ổn định phương sai và cải thiện hiệu quả thuật toán dựa trên khoảng cách.
- **Phương thức**:
  - `fit(X, y=None)`: Trả về chính instance.
  - `transform(X)`: Áp dụng log1p, giá trị âm thay bằng 0.
  - `inverse_transform(X)`: Hoàn nguyên dữ liệu từ log1p.
- **Quy trình xử lý**: Chuyển sang `float64`, thay giá trị âm bằng 0, áp dụng $\log(1+x)$; hồi lại dữ liệu bằng $\exp(x) - 1$.

3.3.5 Class EncoderConfig

- **Công dụng**: Lưu trữ cấu hình cho `FeatureEncoder`, bao gồm loại scaler, xử lý giá trị lạ và định dạng output.
- **Thuộc tính**:
  - `scaler_type` (`str`): `'standard'` hoặc `'minmax'`, mặc định `'standard'`.
  - `handle_unknown` (`str`): cách xử lý giá trị chưa xuất hiện, mặc định `'ignore'`.
  - `sparse_output` (`bool`): output có sparse matrix không, mặc định `False`.
- **Quy trình xử lý**: Khởi tạo tham số, truyền vào pipeline encoder, điều khiển hành vi scaler/encoder theo cấu hình.

3.3.6 Class FeatureEncoder

Class `FeatureEncoder` thực hiện quá trình mã hoá (*encoding*) cho toàn bộ dữ liệu sau giai đoạn *feature engineering*, chuẩn bị đầu vào cho các thuật toán phân cụm (*clustering*).

- **Phân loại các cột dữ liệu:**
  - **DROP**: Các cột không sử dụng cho mô hình: `ID, Year, TOM, BUMO, BUMO_Previous, MostFavourite, Brand, Spontaneous, Trial, P3M, P1M, Weekly, Daily, Needstates, Awareness, Attribute, BrandImage, Occupation`
  - **LOG_NUMERIC**: Các cột số có phân phối lệch (*skewed*) sẽ được áp dụng log-transform và scaling: `Visit, Spending, PPA`
  - **NUMERIC**: Các cột số thông thường chỉ được scaling: `Age, Group_size, NPS_P3M, Brand_Loyalty, Brand_Switcher, Funnel_Depth, Awareness_Usage_Gap, Need_is_Drinking, Awareness_flag, Spontaneous_flag, Trial_flag, P3M_flag, P1M_flag, Weekly_flag, Daily_flag`
  - **ORDINAL**: Các cột có thứ tự tự nhiên (*ordinal*) sẽ được mã hoá thứ tự: `Segmentation, Age_Group_2, Comprehension`
  - **CATEGORICAL**: Các cột không có thứ tự (*nominal*) sẽ được one-hot encoding: `City, Gender, Occupation_Group, Daypart, NeedstateGroup, TOM_Group, BUMO_Group, BUMO_Previous_Group, MostFavourite_Group, NPS_P3M_Group, Brand_Likability`
- **Thuộc tính:**
  - `config`: `EncoderConfig` object chứa các cấu hình encoding và scaling
  - `preprocessor_`: `ColumnTransformer` đã được fit
  - `feature_names_`: Danh sách tên feature sau khi encode
- **Phương thức** `__init__()` là phương thức khởi tạo của class `FeatureEncoder`.
  - **Công dụng**:
    - Tạo một instance của `FeatureEncoder`.
    - Thiết lập cấu hình encoder dựa trên `EncoderConfig`.
    - Khởi tạo các thuộc tính nội bộ `preprocessor_` và `feature_names_` bằng `None`.
  - **Input**:
    - `config` (`Optional[EncoderConfig]`): cấu hình encoder. Nếu không truyền, sẽ sử dụng cấu hình mặc định:
      - `scaler_type='standard'`
      - `handle_unknown='ignore'`
      - `sparse_output=False`

- **Output**:
  - Trả về một instance của `FeatureEncoder` với các thuộc tính:
    - `self.config`: lưu cấu hình encoder.
    - `self.preprocessor_`: `None` ban đầu, sẽ được gán khi fit dữ liệu.
    - `self.feature_names_`: `None` ban đầu, sẽ được gán sau khi fit.
- **Quy trình xử lý**:
  - Nhận `config` từ người dùng; nếu không có thì dùng cấu hình mặc định.
  - Gán cấu hình vào thuộc tính `self.config`.
  - Khởi tạo `self.preprocessor_` và `self.feature_names_` với giá trị `None`, chờ được xây dựng khi fit.

- **Phương thức** `_get_ordinal_categories()` là phương thức tĩnh (`staticmethod`) của class `FeatureEncoder` dùng để định nghĩa thứ tự cho các biến ordinal trước khi thực hiện *ordinal encoding*.
  - **Công dụng**:
    - Xác định thứ tự các category cho từng biến ordinal, đảm bảo encoding phản ánh đúng quan hệ thứ tự tự nhiên.
    - Được sử dụng trong pipeline khi fit `OrdinalEncoder`.
  - **Input**: Không có input (static method).
  - **Output**:
    - Trả về `List of Lists`, mỗi list chứa thứ tự các category cho một biến ordinal. Thứ tự này quan trọng để giữ mối quan hệ tự nhiên giữa các mức:
      1. **Segmentation** (theo mức chi tiêu tăng dần):
         - `'Seg.01 - Mass (<VND 25K)'`
         - `'Seg.02 - Mass Asp (VND 25K - VND 59K)'`
         - `'Seg.03 - Premium (VND 60K - VND 99K)'`
         - `'Seg.04 - Super Premium (VND 100K+)'`
      2. **Age_Group_2** (theo độ tuổi tăng dần):
         - `'16 - 19 y.o.'`, `'20 - 24 y.o.'`
           `'25 - 29 y.o.'`
           `'30 - 34 y.o.'`, `'35 - 39 y.o.'`
           `'40 - 44 y.o.'`
           `'45+ y.o.'`
      3. **Comprehension** (từ không biết đến biết rõ):
         - `"Don't know"`, `'Know a little'`, `'Know well'`
  - **Quy trình xử lý**:

    - Xác định các mức của từng biến ordinal theo thứ tự logic (chi tiêu tăng dần, tuổi tăng dần, mức độ hiểu biết tăng dần).
    - Gom các thứ tự này vào một danh sách gồm ba list con.
    - Trả về danh sách này để sử dụng trong `OrdinalEncoder`.
  - **Lưu ý**:
    - Nếu biến ordinal chỉ có 2 giá trị (Yes/No), encoding sẽ tự động xử lý mà không cần danh sách thứ tự dài.
    - Thứ tự các category được định nghĩa để `OrdinalEncoder` encode các giá trị thành số nguyên theo thứ tự tăng dần.

- **Phương thức** `_build_preprocessor()` xây dựng một `ColumnTransformer` kết hợp các pipeline xử lý cho từng nhóm cột, chuẩn bị dữ liệu đầu vào cho các thuật toán clustering.
  - **Công dụng**:
    - Xây dựng pipeline xử lý tất cả các nhóm feature:
      1. **LOG_NUMERIC**: áp dụng `LogTransformer` + scale (Standard hoặc MinMax).
      2. **NUMERIC**: chỉ scale với scaler đã chọn.
      3. **ORDINAL**: encode các biến ordinal theo thứ tự tự nhiên với `OrdinalEncoder`.
      4. **CATEGORICAL**: one-hot encode các biến categorical, drop first category để tránh multicollinearity.
  - **Input**:
    - Không có input trực tiếp.
    - Sử dụng các thuộc tính của class `FeatureEncoder`:
      - `self.config`: cấu hình encoder, xác định loại scaler, cách xử lý giá trị unknown và output (dense/sparse).
      - `self.LOG_NUMERIC_COLS`, `self.NUMERIC_COLS`, `self.ORDINAL_COLS`, `self.CATEGORICAL_COLS`: danh sách các cột theo nhóm.
  - **Output**:
    - Trả về một `ColumnTransformer` với các pipeline xử lý tương ứng cho từng nhóm cột.
    - Các cột không nằm trong nhóm sẽ bị loại bỏ (`remainder='drop'`).
    - ColumnTransformer này có thể được fit và transform trực tiếp trên DataFrame đầu vào.
  - **Quy trình xử lý**:
    - Chọn loại scaler (`StandardScaler` hoặc `MinMaxScaler`) dựa trên `config`.
    - Tạo pipeline cho nhóm `LOG_NUMERIC`: áp dụng `LogTransformer` rồi scale.

    - Tạo pipeline scale cho nhóm `NUMERIC` thông thường.
    - Khởi tạo `OrdinalEncoder` với thứ tự các giá trị đã định nghĩa.
    - Khởi tạo `OneHotEncoder` cho các biến categorical.
    - Gộp toàn bộ thành một `ColumnTransformer` và loại bỏ các cột không được chỉ định.
  - **Lưu ý**:
    - Loại scaler được xác định dựa trên `self.config.scaler_type` (`StandardScaler` hoặc `MinMaxScaler`).
    - Các giá trị unknown trong ordinal sẽ được gán `-1`.
    - Output của one-hot encoding có thể là dense hoặc sparse tùy cấu hình `self.config.sparse_output`.
    - Drop first category trong one-hot encoding giúp tránh *multicollinearity* khi sử dụng cho các mô hình học máy tuyến tính.
- **Phương thức** `_prepare_dataframe(df)`
  - **Công dụng**: Loại bỏ các cột không cần thiết khỏi DataFrame trước khi tiến hành encoding, đảm bảo chỉ giữ lại các biến phục vụ mô hình.
  - **Input**: `df` (`pd.DataFrame`): DataFrame gốc chứa tất cả các cột sau feature engineering.
  - **Output**:
    - `pd.DataFrame`: bản sao DataFrame đã loại bỏ các cột trong `DROP_COLS`.
    - Thông báo console (print) tên 5 cột đầu tiên đã bị drop và tổng số cột bị loại bỏ.
  - **Quy trình xử lý**:
    - Tạo bản sao của DataFrame để tránh thay đổi dữ liệu gốc.
    - Xác định những cột nằm trong `DROP_COLS` và thật sự xuất hiện trong DataFrame.
    - Loại bỏ các cột này khỏi DataFrame.
    - Trả về DataFrame đã được làm sạch.
  - **Lưu ý**:
    - Chỉ drop các cột tồn tại trong DataFrame, tránh lỗi khi cột không có.
    - Giữ nguyên dữ liệu gốc, phương thức hoạt động trên bản sao (`df.copy()`).
- **Phương thức** `_extract_feature_names()`
  - **Công dụng**: Trích xuất danh sách tên feature sau khi dữ liệu đã được transform bằng `preprocessor_` (`ColumnTransformer`).
  - **Input**:

* Không có input trực tiếp.
* Sử dụng `self.preprocessor_` (ColumnTransformer đã fit).

– **Output**:

* Trả về một list (`List[str]`) chứa tên tất cả các feature sau khi transform:
  1. **LOG_NUMERIC và NUMERIC**: giữ nguyên tên cột gốc.
  2. **ORDINAL**: giữ nguyên tên cột gốc.
  3. **CATEGORICAL**: lấy tên từ `OneHotEncoder.get_feature_names_out()`. Nếu transformer không hỗ trợ phương thức này, fallback giữ nguyên tên cột gốc.

– **Quy trình xử lý**:

* Duyệt qua từng transformer trong `ColumnTransformer`.
* Giữ nguyên tên cột cho các nhóm numeric và ordinal.
* Lấy tên các biến one-hot bằng `get_feature_names_out`.
* Ghép tất cả tên biến lại thành một danh sách và trả về.

– **Lưu ý**:

* Các cột không được encode (remainder) sẽ bị bỏ qua.
* Kết quả giúp xác định thứ tự và tên các feature khi xuất DataFrame transform ra hoặc khi dùng trong các mô hình ML.

- **Phương thức** `fit_transform(df)`
  – **Công dụng**: Thực hiện việc fit encoder trên DataFrame đầu vào và transform dữ liệu sang dạng numeric chuẩn hoá, sẵn sàng cho các thuật toán clustering.
  – **Input**: `df` (`pd.DataFrame`): DataFrame đã qua feature engineering (ví dụ: `df_final`).
  – **Output**:
    * `X` (`np.ndarray`): numpy array chứa dữ liệu đã được encode, có shape $(n\_samples, n\_features)$.
    * `self.feature_names_`: list tên tất cả các feature sau encode, phục vụ cho việc tạo DataFrame transform hoặc interpret kết quả.
  – **Quy trình xử lý**:
    * Chuẩn bị DataFrame bằng cách loại bỏ các cột không sử dụng.
    * Xây dựng `ColumnTransformer` gồm các pipeline xử lý cho từng nhóm biến.
    * Thực hiện `fit` và `transform` trên dữ liệu đã chuẩn bị để tạo ma trận đặc trưng.
    * Trích xuất và lưu lại tên các biến sau khi encoding.

    - Trả về ma trận đặc trưng dạng `numpy array`.
  - **Lưu ý**:
    - Phương thức trả về numpy array, không phải DataFrame.
    - Feature names được lưu trong `self.feature_names_` để tái sử dụng hoặc xuất DataFrame.
    - Các bước xử lý được in ra console để theo dõi tiến trình.

- **Phương thức** `transform(df)`
  - **Công dụng**: Sử dụng encoder đã fit để transform dữ liệu mới, áp dụng các bước encoding và scaling giống như dữ liệu huấn luyện.
  - **Input**:
    - `df` (`pd.DataFrame`): DataFrame mới cần encode.
    - DataFrame có thể chứa tất cả các cột feature giống dữ liệu huấn luyện; các cột không cần thiết sẽ bị drop.
  - **Output**: `X` (`np.ndarray`): numpy array chứa dữ liệu đã được encode, có shape $(n\_samples, n\_features)$, sẵn sàng cho mô hình.
  - **Quy trình xử lý**:
    - Kiểm tra encoder đã được `fit`. Nếu chưa, báo lỗi.
    - Chuẩn hoá và làm sạch dữ liệu mới bằng hàm `_prepare_dataframe()`.
    - Áp dụng `preprocessor.transform()` để encode toàn bộ dữ liệu.
    - Trả về ma trận đặc trưng dạng `numpy array`.
  - **Lưu ý**:
    - Nếu encoder chưa được fit, phương thức sẽ raise `ValueError`.
    - Phương thức chỉ transform dữ liệu mới, không fit lại encoder.
    - Output là numpy array; tên feature có thể tham khảo từ `self.feature_names_`.

- **Phương thức** `save(filepath)`
  - **Công dụng**: Dùng để lưu toàn bộ `FeatureEncoder` (bao gồm `preprocessor` và `feature_names`) vào một file `pkl` để tái sử dụng sau này.
  - **Input**: `filepath` (`str`): đường dẫn file để lưu encoder.
    Mặc định: `'../../models_saved/encoders/feature_encoder.pkl'`.
  - **Output**:
    - Không có output trả về.
    - File `.pkl` được tạo ra tại đường dẫn `filepath` chứa toàn bộ object `FeatureEncoder`.
  - **Quy trình xử lý**:

* Kiểm tra encoder đã được `fit`. Nếu chưa, báo lỗi.
* Tạo thư mục lưu trữ nếu chưa tồn tại.
* Ghi toàn bộ object `FeatureEncoder` vào file bằng `joblib.dump()`.
* Thông báo đường dẫn đã lưu thành công.

– **Lưu ý**:

* Nếu encoder chưa được fit, phương thức sẽ raise `ValueError`.
* File lưu giữ nguyên trạng encoder, bao gồm các pipeline, preprocessor và feature names.

- **Phương thức** `load(filepath)`
  – **Công dụng**: Tải lại `FeatureEncoder` đã lưu từ file `.pkl`, cho phép sử dụng encoder mà không cần fit lại.
  – **Input**: `filepath` (`str`): đường dẫn file chứa encoder đã lưu.
  Mặc định: `'../../models_saved/encoders/feature_encoder.pkl'`.
  – **Output**: Trả về object `FeatureEncoder` đã được load.
  – **Quy trình xử lý**:
    * Đọc file `pkl` bằng `joblib.load()`.
    * Khôi phục object `FeatureEncoder` đầy đủ (gồm preprocessor và thuộc tính đi kèm).
    * In thông báo đã load thành công.
    * Trả về encoder đã được load.
  – **Lưu ý**:
    * File `.pkl` phải tồn tại; nếu không sẽ gây lỗi.
    * Encoder sau khi load đã sẵn sàng để gọi `transform()` hoặc các phương thức khác.
- **Phương thức** `get_feature_names()`
  – **Công dụng**: lấy danh sách tên các features sau khi đã encoding. Đây là danh sách được tạo trong bước `fit_transform()`.
  – **Input**: Không có input trực tiếp.
  – **Output**: `List[str]`: danh sách tên các feature sau encode.
  – **Quy trình xử lý**:
    * Kiểm tra thuộc tính `feature_names_` đã tồn tại hay chưa.
    * Nếu chưa, raise `ValueError` yêu cầu gọi `fit_transform()` trước.
    * Nếu đã có, trả về bản sao của `self.feature_names_`.
  – **Lưu ý**:

* Phương thức chỉ có thể được gọi sau khi encoder đã fit dữ liệu bằng `fit_transform()`.
* Trả về bản sao (`copy()`) để tránh thay đổi ngoài ý muốn trên danh sách feature gốc.

3.3.7 Class TrainingConfig

Cấu hình này chuẩn hóa tất cả thông số cần thiết cho huấn luyện mô hình clustering, bao gồm dữ liệu, loại model, số cụm, seed, các feature, đường dẫn lưu kết quả và model, cũng như hyperparameters cố định, giúp code gọn gàng, dễ bảo trì và tái sử dụng.

- **data_path:** Đường dẫn tới file CSV chứa dữ liệu đã encode. Mặc định là `"data/processed/encoded_data.csv"`.
- **model_type:** Loại mô hình clustering sử dụng. Giá trị có thể là `'kmeans'`, `'gmm'`, `'dbscan'`, `'hdbscan'`. Mặc định là `'kmeans'`.
- **n_clusters:** Số lượng cụm (dùng cho KMeans/GMM). Mặc định là 5.
- **random_state:** Giá trị seed để đảm bảo kết quả tái lập. Mặc định là 42.
- **selected_features:** Danh sách tên cột dữ liệu sẽ được sử dụng để clustering. Nếu để `None`, tất cả các cột numeric sẽ được sử dụng.
- **results_path:** Đường dẫn file CSV lưu kết quả đánh giá clustering, bao gồm các metric như Silhouette, Calinski-Harabasz, Davies-Bouldin.
  Mặc định là `"results/cluster_results.csv"`.
- **model_path:** Đường dẫn file để lưu mô hình đã train dưới định dạng pickle (.pkl). Mặc định là `"results/best_cluster_model.pkl"`.
- **model_params:** Dictionary chứa các hyperparameters cố định cho từng loại model.
  Quy trình xử lý:
  - Nhận đường dẫn dữ liệu đã encode và loại mô hình cần huấn luyện.
  - Thiết lập số cụm (`n_clusters`) và random seed.
  - Xác định danh sách features cần sử dụng (nếu không có, dùng toàn bộ).
  - Đọc các tham số mô hình trong `model_params`.
  - Khởi tạo đường dẫn lưu kết quả đánh giá và file mô hình đã train.

3.3.8 Class ModelTrainer

Chịu trách nhiệm huấn luyện các mô hình phân cụm dựa trên cấu hình từ `TrainingConfig`, đánh giá kết quả bằng `ClusteringEvaluator` và lưu model đã huấn luyện.

- **Quy trình làm việc:**
  1. `load_data()` - Tải dữ liệu đã encode từ CSV, chọn các feature nếu cần và chuyển thành numpy array.
  2. `build_model()` - Khởi tạo mô hình clustering dựa trên loại model và các tham số trong config.

3. `train_model()` - Huấn luyện mô hình với dữ liệu đã load.
4. `evaluate()` - Đánh giá mô hình bằng các metrics như Silhouette, Calinski-Harabasz, Davies-Bouldin.
5. `save_model()` - Lưu mô hình đã huấn luyện ra file pickle.
6. `get_cluster_labels()` - Lấy cluster labels từ mô hình đã huấn luyện.
7. `save_labels(output_path)` - Lưu cluster labels ra file CSV.

Thuộc tính (Attributes):

- **config:** Object `TrainingConfig` chứa tất cả thông số huấn luyện.
- **evaluator:** Object `ClusteringEvaluator` dùng để đánh giá mô hình.
- **logger:** Logger dùng để hiển thị thông tin quá trình huấn luyện.
- **df:** `DataFrame` chứa dữ liệu đã tải.
- **X:** `numpy array` của các đặc trưng đã chọn.
- **model:** Object mô hình đã được huấn luyện (KMeans, GMM, DBSCAN hoặc HDBSCAN).

- **Phương thức** `__init__()` là phương thức khởi tạo của class `ModelTrainer`.
  - **Công dụng**: Thiết lập toàn bộ cấu hình và các thành phần cần thiết cho quá trình huấn luyện mô hình phân cụm.
  - **Input**:
    - `config` (`TrainingConfig`): cấu hình huấn luyện, bao gồm loại model, số cụm, feature, đường dẫn lưu model, hyperparameters.
    - `evaluator` (`ClusteringEvaluator`): evaluator dùng để đánh giá clustering.
    - `logger` (`Optional[logging.Logger]`): logger để ghi thông tin. Nếu không truyền, sẽ tạo logger mặc định với format hiển thị đơn giản và mức `INFO`.
  - **Output**:
    - Trả về một instance của `ModelTrainer` với các thuộc tính:
      - `self.config`: lưu cấu hình training.
      - `self.evaluator`: evaluator dùng để đánh giá clustering.
      - `self.logger`: logger hiển thị thông tin.
      - `self.df`: `None` ban đầu, lưu DataFrame dữ liệu sau khi load.
      - `self.X`: `None` ban đầu, lưu numpy array các feature.
      - `self.model`: `None` ban đầu, lưu mô hình đã huấn luyện.
  - **Quy trình xử lý:**
    - Nhận `TrainingConfig` làm cấu hình huấn luyện và `ClusteringEvaluator` để phục vụ đánh giá model.

    - Thiết lập hệ thống logging:
        - Nếu không truyền `logger`, tạo logger mặc định với `StreamHandler`.
        - Định dạng log ở mức `INFO`.
        - Đảm bảo logger không thêm nhiều handler trùng lặp.

- **Phương thức** `load_data()`
  - **Công dụng**: Tải dữ liệu đã encode từ CSV, kiểm tra cột feature và chuyển sang `numpy array`.
  - **Input**: Không có input trực tiếp; sử dụng `self.config.data_path` và `self.config.selected_features`.
  - **Output**: Cập nhật `self.df` (DataFrame) và `self.X` (numpy array). Nếu file/cột không hợp lệ, ném lỗi tương ứng.
  - **Quy trình xử lý**:
    - Kiểm tra file tồn tại, ném `FileNotFoundError` nếu không.
    - Đọc CSV vào DataFrame.
    - Lọc các cột feature nếu có, ném `ValueError` khi thiếu cột.
    - Chuyển sang `numpy array` và log kích thước dữ liệu.

- **Phương thức** `build_model(n_clusters=None)`
  - **Công dụng**: Khởi tạo mô hình phân cụm dựa trên cấu hình trong `TrainingConfig`. Tự động gán tham số mặc định, ghi đè bằng `model_params` nếu có, và trả về instance mô hình đã sẵn sàng để train.
  - **Input**: `n_clusters (Optional[int])`: số cụm cần tạo (chỉ áp dụng cho KMeans/GMM). Nếu `None`, dùng `config.n_clusters`.
  - **Output**:
    - Trả về một instance của model tương ứng:
      - `KMeans` nếu `config.model_type='kmeans'`
      - `GaussianMixture` nếu `config.model_type='gmm'`
      - `DBSCAN` nếu `config.model_type='dbscan'`
      - `HDBSCAN` nếu `config.model_type='hdbscan'`
    - Nếu model không hợp lệ, ném `ValueError`.
    - Nếu HDBSCAN chưa cài, ném `ImportError`.
  - **Quy trình xử lý:**
    - Chuẩn hoá `model_type` về dạng lowercase và sao chép `model_params`.
    - Với từng loại mô hình:

· **KMeans**: tạo bộ tham số mặc định (`n_clusters`, `n_init`, `random_state`), sau đó merge với params.

· **GMM**: thiết lập `n_components`, `random_state`, rồi merge params.

· **DBSCAN**: tạo tham số mặc định (`eps`, `min_samples`, `n_jobs`) và merge params.

· **HDBSCAN**: kiểm tra thư viện đã cài hay chưa, gán `min_cluster_size`, `min_samples`, rồi merge params.

* Trả về instance model tương ứng.

* Nếu model_type không hợp lệ → ném lỗi `ValueError`.

- **Phương thức** `train_model()`
  - **Công dụng**: Huấn luyện mô hình phân cụm theo cấu hình đã định. Hàm tự động build model từ config và fit trên toàn bộ tập dữ liệu đã encode.
  - **Input**:
    * Không có input trực tiếp.
    * Sử dụng dữ liệu và cấu hình từ:
      · `self.X`: numpy array các feature đã load.
      · `self.config`: cấu hình huấn luyện.
  - **Output**:
    * Không trả về giá trị.
    * Cập nhật thuộc tính `self.model` với mô hình đã được huấn luyện.
    * Nếu dữ liệu chưa được load (`self.X=None`), ném `ValueError`.

    Quy trình xử lý:

    * Kiểm tra dữ liệu đã được load hay chưa (`self.X`). Nếu chưa → báo lỗi.
    * Xác định loại mô hình từ `config.model_type`.
    * Ghi log trạng thái training (kèm số cụm nếu là KMeans/GMM).
    * Gọi `build_model()` để khởi tạo mô hình phù hợp.
    * Fit mô hình trên toàn bộ dữ liệu `X`.
    * Ghi log xác nhận mô hình đã huấn luyện xong.
- **Phương thức** `evaluate()`
  - **Công dụng**: Đánh giá chất lượng mô hình phân cụm sau khi huấn luyện bằng ba chỉ số quan trọng: Silhouette, Calinski–Harabasz và Davies–Bouldin. Kết quả trả về dưới dạng dictionary.
  - **Input**:
    * Không có input trực tiếp.

    - Sử dụng dữ liệu và mô hình từ:
        - `self.X`: numpy array các feature đã load.
        - `self.model`: mô hình đã huấn luyện.
        - `self.config.model_type`: loại model để ghi vào đánh giá.
  - **Output**:
    - Trả về một dictionary chứa các metrics:
        - `'model'`: tên model.
        - `'n_clusters'`: số cụm (nếu áp dụng).
        - `'silhouette'`: Silhouette Score.
        - `'calinski_harabasz'`: Calinski-Harabasz Score.
        - `'davies_bouldin'`: Davies-Bouldin Index.
    - Nếu dữ liệu chưa load hoặc model chưa train, ném `ValueError`.
  - **Quy trình xử lý:**
    - Kiểm tra mô hình đã được huấn luyện hay chưa.
    - Kiểm tra dữ liệu đã được load hay chưa.
    - Log trạng thái "Evaluating model...".
    - Lấy nhãn cụm bằng `get_cluster_labels()`.
    - Gọi `evaluator.evaluate_once()` để tính các chỉ số.
    - Log kết quả từng metric.
    - Trả về dictionary gồm tên model, số cụm và các metrics.

- **Phương thức** `save_model(path=None)`
  - **Công dụng**: Lưu mô hình clustering đã huấn luyện vào file `.pkl` để tái sử dụng sau này.
  - **Input (Arguments)**:
    - `path (Optional[str])`: đường dẫn file pickle để lưu mô hình. Nếu `None`, dùng `self.config.model_path`.
  - **Output**:
    - Không trả về giá trị.
    - Ghi file pickle chứa mô hình đã huấn luyện.
    - Nếu `self.model` chưa được train, ném `ValueError`.
  - **Quy trình xử lý:**
    - Kiểm tra xem mô hình đã được huấn luyện hay chưa.
    - Xác định đường dẫn lưu model (tham số `path` hoặc `config.model_path`).

    * Tạo thư mục lưu trữ nếu chưa tồn tại.
    * Ghi mô hình xuống file bằng `joblib.dump()`.
    * Ghi log thông báo đường dẫn file đã lưu.

- **Phương thức** `load_model(path)`
  - **Công dụng**: Tải lại mô hình clustering từ file `.pkl` để sử dụng hoặc đánh giá tiếp.
  - **Input**:
    * `path (str)`: đường dẫn file pickle chứa mô hình.
  - **Output**:
    * Trả về instance của mô hình đã lưu (KMeans, GaussianMixture, DBSCAN hoặc HDBSCAN).
    * Nếu file không tồn tại, ném `FileNotFoundError`.
  - **Quy trình xử lý:**
    * Kiểm tra sự tồn tại của file tại đường dẫn `path`. Nếu không tồn tại → ném `FileNotFoundError`.
    * Sử dụng `joblib.load()` để đọc model từ file.
    * Trả về instance mô hình đã load sẵn sàng để sử dụng.

- **Phương thức** `get_cluster_labels()`
  - **Công dụng**: Trích xuất nhãn cụm từ mô hình phân cụm đã huấn luyện, trả về dưới dạng `numpy array`. Với DBSCAN/HDBSCAN, có thể xuất hiện nhãn `-1` cho các điểm nhiễu (noise).
  - **Input**:
    * Không có input trực tiếp.
    * Sử dụng:
      · `self.model`: mô hình đã huấn luyện.
      · `self.X`: numpy array dữ liệu.
      · `self.config.model_type`: loại mô hình.
  - **Output**:
    * Trả về một `numpy array` chứa cluster labels.
    * Nếu mô hình chưa train, ném `ValueError`.
    * Nếu dữ liệu chưa load, ném `ValueError`.
  - **Quy trình xử lý:**
    * Kiểm tra mô hình đã được huấn luyện chưa. Nếu chưa → báo lỗi.
    * Kiểm tra dữ liệu đã load chưa. Nếu chưa → báo lỗi.

    - Với DBSCAN/HDBSCAN: gọi `fit_predict(X)` để lấy nhãn cụm.
    - Với các mô hình khác:
        - Nếu model có thuộc tính `labels_`, trả về `model.labels_`.
        - Ngược lại, gọi `model.predict(X)` để lấy nhãn cụm.
    - Trả về mảng nhãn cụm dạng `numpy array`.

- **Phương thức** `save_labels(output_path)`
  - **Công dụng**: Lưu nhãn cụm (`cluster labels`) của dữ liệu ra file CSV, kèm các thông tin gốc từ DataFrame.
  - **Input**: `output_path (str)`: đường dẫn file CSV để lưu cluster labels.
  - **Output**:
    - Không trả về giá trị.
    - Ghi file CSV chứa cluster labels.
    - Nếu mô hình chưa train hoặc dữ liệu chưa load, sẽ ném lỗi từ `get_cluster_labels()`.
  - **Quy trình xử lý:**
    - Lấy nhãn cụm bằng `get_cluster_labels()`.
    - Tạo bản sao DataFrame gốc và thêm cột `cluster`.
    - Tạo thư mục lưu trữ nếu chưa tồn tại.
    - Ghi DataFrame kèm nhãn cụm ra file CSV với encoding `utf-8-sig`.
    - Ghi log xác nhận file đã được lưu.

3.3.9 Class BaseMetric(ABC)

Đóng vai trò lớp trừu tượng, cung cấp khuôn mẫu chung cho các metric và buộc các lớp con phải triển khai phương thức tính toán theo một chuẩn thống nhất.

- **Phương thức** `__init__()`
  - **Công dụng**: Thiết lập tên metric và quy tắc đánh giá (`higher_is_better`) cho từng đối tượng.
  - **Input**:
    - `name` – tên của metric;
    - `higher_is_better` – giá trị boolean cho biết metric càng cao càng tốt hay không.
  - **Output**: Hàm không trả về giá trị, nhưng khởi tạo đối tượng bằng cách gán `name` và `higher_is_better` vào các thuộc tính nội bộ của lớp.

- **Quy trình xử lý**: Hàm nhận hai tham số đầu vào, sau đó gán `name` và `higher_is_better` vào thuộc tính của đối tượng để các lớp con sử dụng trong quá trình tính toán và đánh giá mô hình.

- **Phương thức** `compute()`
  - **Công dụng**: Định nghĩa giao diện chung để các lớp con bắt buộc phải triển khai hàm tính toán giá trị metric cho mô hình phân cụm.
  - **Input**:
    - `X` – ma trận đặc trưng dạng `numpy.ndarray`.
    - `labels` – nhãn cụm được gán cho từng quan sát.
  - **Output**: Trả về một giá trị dạng `float` biểu diễn metric đã được tính toán.
  - **Quy trình xử lý**: Vì đây là phương thức trừu tượng, hàm không thực hiện xử lý trực tiếp mà buộc các lớp con phải override để tự triển khai thuật toán tính metric dựa trên `X` và `labels`.

- **Phương thức** `is_better()`
  - **Công dụng**: So sánh hai giá trị metric và xác định xem `score1` có tốt hơn `score2` hay không, dựa trên quy tắc đánh giá `higher_is_better`.
  - **Input**:
    - `score1` – giá trị metric thứ nhất.
    - `score2` – giá trị metric thứ hai.
  - **Output**: Trả về giá trị boolean `True` nếu `score1` được xem là tốt hơn `score2`.
  - **Quy trình xử lý**: Hàm kiểm tra thuộc tính `higher_is_better`: nếu đúng, so sánh `score1 > score2`; nếu sai, so sánh `score1 < score2`. Kết quả so sánh được trả về dưới dạng boolean.

3.3.10 Các lớp Metric kế thừa từ BaseMetric và triển khai đa hình

- **Lớp** `SilhouetteMetric`
  - **Công dụng**: Cài đặt metric Silhouette Score, kế thừa từ *BaseMetric* và override phương thức `compute()` để tính giá trị đánh giá mức độ tách biệt giữa các cụm.
  - **Input**:
    - `X`: Ma trận đặc trưng (`numpy.ndarray`).
    - `labels`: Nhãn cụm tương ứng với từng quan sát.
  - **Output**: Trả về giá trị Silhouette Score dạng `float`.
  - **Quy trình xử lý**: Phương thức `compute()` gọi hàm `silhouette_score(X, labels)` từ thư viện `sklearn`, sau đó chuyển đổi kết quả sang kiểu `float` và trả về.

- **Lớp** `CalinskiMetric`

- **Công dụng**: Cài đặt Calinski–Harabasz Score, kế thừa từ *BaseMetric* và override phương thức `compute()` để đánh giá mức độ phân tách giữa các cụm.
- **Input**:
  - `X`: Ma trận đặc trưng.
  - `labels`: Nhãn cụm của từng điểm dữ liệu.
- **Output**: Trả về giá trị Calinski–Harabasz Score kiểu `float`.
- **Quy trình xử lý**: Phương thức `compute()` gọi hàm `calinski_harabasz_score(X, labels)`, chuyển đổi sang kiểu `float` và trả về.

- **Lớp** `DaviesMetric`
  - **Công dụng**: Cài đặt Davies–Bouldin Index, kế thừa từ *BaseMetric* và override phương thức `compute()`. Metric này đánh giá mức độ tương tự giữa các cụm (giá trị thấp hơn tốt hơn).
  - **Input**:
    - `X`: Ma trận đặc trưng.
    - `labels`: Nhãn cụm đã gán từ thuật toán clustering.
  - **Output**: Trả về Davies–Bouldin Index dạng `float`.
  - **Quy trình xử lý**: Phương thức `compute()` gọi hàm `davies_bouldin_score(X, labels)`, chuyển kết quả sang `float` và trả về.

3.3.11 Class ClusteringEvaluator

Chịu trách nhiệm đánh giá chất lượng mô hình phân cụm bằng cách tính đồng thời ba chỉ số: Silhouette Score, Calinski–Harabasz Score và Davies–Bouldin Index. Ngoài ra, lớp còn quản lý log và lưu kết quả đánh giá cho nhiều mô hình khác nhau.

- **Phương thức** `__init__()`
  - **Công dụng**:Khởi tạo đối tượng `ClusteringEvaluator`, thiết lập hệ thống ghi log, tạo ba metric chính (Silhouette, Calinski–Harabasz, Davies–Bouldin) dựa trên cơ chế kế thừa và đa hình, đồng thời chuẩn bị cấu trúc lưu kết quả đánh giá.
  - **Input**: `logger`: (tuỳ chọn) đối tượng `logging.Logger`. Nếu không được truyền vào, phương thức sẽ tạo logger mặc định ở mức `INFO`.
  - **Output**: Không trả về giá trị. Phương thức khởi tạo các thuộc tính nội bộ: `logger`, `metrics`, `results_`.
  - **Quy trình xử lý**:
    - Kiểm tra tham số `logger`: Nếu không có, tạo logger mặc định tên `ClusteringEvaluator`, cấu hình mức `INFO`, tạo `StreamHandler` và tránh trùng lặp log bằng cách tắt `propagate`.

    * Khởi tạo ba metric theo cơ chế kế thừa từ `BaseMetric`: `SilhouetteMetric`, `CalinskiMetric`, `DaviesMetric`. Các metric được lưu trong dictionary `self.metrics`.
    * Khởi tạo danh sách `self.results_` để lưu kết quả đánh giá của từng mô hình phân cụm.

- **Phương thức** `evaluate_once()`
  - **Công dụng**: Đánh giá một mô hình phân cụm duy nhất bằng cách tính ba metric chính (Silhouette, Calinski–Harabasz, Davies–Bouldin). Phương thức sử dụng cơ chế đa hình (*polymorphism*) để gọi `compute()` tương ứng với từng metric.
  - **Input**:
    * `X`: Ma trận đặc trưng dưới dạng `numpy.ndarray` hoặc `DataFrame`.
    * `labels`: Mảng nhãn cụm ứng với mỗi điểm dữ liệu.
    * `model_name`: Tên mô hình phân cụm (ví dụ: `"KMeans_k5"`, `"GMM_k4"`).
  - **Output**:
    * Trả về một `dict` chứa thông tin đánh giá mô hình:
      · `model`: Tên mô hình.
      · `n_clusters`: Số cụm được tạo.
      · `silhouette`: Giá trị Silhouette Score.
      · `calinski_harabasz`: Calinski–Harabasz Score.
      · `davies_bouldin`: Davies–Bouldin Index.
  - **Quy trình xử lý**:
    * Chuyển đổi `X` về dạng `numpy.ndarray` nếu đang ở dạng `DataFrame`.
    * Chuyển `labels` sang `numpy.ndarray`.
    * Kiểm tra số cụm hợp lệ: phải có ít nhất hai cụm và nhãn phải khớp số lượng mẫu.
    * Khởi tạo `result` chứa tên mô hình và số cụm.
    * Duyệt qua từng metric trong `self.metrics` và gọi `compute()` theo cơ chế đa hình.
    * Lưu kết quả vào `self.results_`.
    * Trả về dictionary kết quả.

- **Phương thức** `evaluate_many()`
  - **Công dụng**: Đánh giá đồng thời nhiều mô hình phân cụm bằng cách lần lượt gọi `evaluate_once()` cho từng mô hình trong `labels_dict`, sau đó tổng hợp kết quả thành một bảng duy nhất.

- **Input**:
  - `X`: Ma trận đặc trưng (`numpy.ndarray` hoặc `DataFrame`).
  - `labels_dict`: Từ điển ánh xạ tên mô hình sang nhãn cụm, dạng `{model_name: labels}`.
- **Output**:
  - Trả về một `DataFrame` tổng hợp kết quả đánh giá của tất cả mô hình, gồm các cột:
    - `model`
    - `n_clusters`
    - `silhouette`
    - `calinski_harabasz`
    - `davies_bouldin`
- **Quy trình xử lý**:
  - Ghi log số lượng mô hình cần đánh giá.
  - Duyệt từng mô hình trong `labels_dict` và gọi `evaluate_once()`.
  - Nếu mô hình nào lỗi, ghi cảnh báo và bỏ qua.
  - Sau khi xử lý tất cả mô hình:
    - Nếu không có kết quả nào, trả về `DataFrame` rỗng.
    - Nếu có dữ liệu, tạo `DataFrame` từ `self.results_`.
  - Ghi log bảng kết quả và trả về.
- **Ngoại lệ**: Không trực tiếp ném ngoại lệ; lỗi trong từng mô hình được bắt, ghi log cảnh báo và mô hình đó được bỏ qua.

- **Phương thức** `save_results()`
  - **Công dụng**:
    - Lưu toàn bộ kết quả đánh giá clustering (được lưu trong thuộc tính `results_`) ra một file CSV.
    - Hỗ trợ việc lưu trữ, báo cáo và sử dụng lại kết quả đánh giá trong các bước phân tích sau.
  - **Input**: `filepath` (`str`): Đường dẫn file CSV sẽ được lưu.
  - **Output**:
    - Hàm không trả về giá trị (return `None`).
    - Tạo file CSV chứa kết quả đánh giá dưới dạng bảng với các cột:
      - `model`
      - `n_clusters`

        · `silhouette`
        · `calinski_harabasz`
        · `davies_bouldin`
    - **Quy trình xử lý**:
        * Kiểm tra xem `results_` có dữ liệu hay không.
            · Nếu rỗng, ghi log cảnh báo và kết thúc hàm.
        * Chuyển `results_` thành `DataFrame`.
        * Kiểm tra và tạo thư mục (folder) nếu chưa tồn tại, dựa trên phần thư mục của `filepath`.
        * Ghi `DataFrame` ra file CSV với mã hoá `utf-8-sig`.
        * Ghi log xác nhận lưu thành công.
        * Nếu xảy ra lỗi (ghi file, quyền truy cập, đường dẫn sai), hàm bắt ngoại lệ và log lỗi.
- **Phương thức** `plot_all_scores()`
    - **Công dụng**:
        * Vẽ đồng thời cả ba metric đánh giá cụm (Silhouette, Calinski-Harabasz, Davies-Bouldin) trong cùng một biểu đồ gồm 3 subplot.
        * Giúp so sánh trực quan hiệu suất của nhiều mô hình theo từng tiêu chí trong cùng một hình duy nhất.
    - **Input**:
        * `figsize` (`tuple`): Kích thước toàn bộ figure theo dạng `(width, height)`.
        * `save_path` (`str` hoặc `None`): Đường dẫn để lưu hình. Nếu `None`, chỉ hiển thị mà không lưu file.
    - **Output**:
        * Không trả về giá trị (`None`).
        * Hiển thị figure chứa 3 biểu đồ hoặc tạo file ảnh nếu có `save_path`.
    - **Quy trình xử lý**:
        * Kiểm tra xem danh sách kết quả `results_` có dữ liệu hay không. Nếu rỗng, ghi cảnh báo và dừng.
        * Chuyển `results_` thành `DataFrame` để trích xuất các metric.
        * Tạo một figure gồm 3 subplot đặt cạnh nhau.
        * Với mỗi subplot:
            · Lấy danh sách mô hình và giá trị metric tương ứng.
            · Vẽ biểu đồ cột với màu được tạo từ colormap `viridis`.

· Ghi giá trị score lên mỗi cột.

· Thiết lập tiêu đề, trục, xoay nhãn và thêm đường lưới.

* Điều chỉnh layout để các biểu đồ không bị chồng chéo.
* Nếu có `save_path`: tạo thư mục (nếu cần) và lưu hình xuống file.
* Nếu không có `save_path`: hiển thị trực tiếp bằng `plt.show()`.
* Đóng figure bằng `plt.close()` để giải phóng bộ nhớ.

- **Phương thức** `get_best_model()`
  - **Công dụng**:
    * Xác định mô hình phân cụm tốt nhất dựa trên một metric đánh giá được chọn.
    * Hỗ trợ so sánh và lựa chọn mô hình tối ưu trong các thuật toán clustering.
  - **Input**:
    * `metric` (`str`): Tên metric dùng để xác định mô hình tốt nhất. Giá trị hợp lệ gồm:
      · `'silhouette'` (cao hơn tốt hơn)
      · `'calinski_harabasz'` (cao hơn tốt hơn)
      · `'davies_bouldin'` (thấp hơn tốt hơn)
  - **Output**:
    * Một `dict` chứa thông tin đầy đủ của mô hình tốt nhất:
      · `model`
      · `n_clusters`
      · `silhouette`
      · `calinski_harabasz`
      · `davies_bouldin`
    * Nếu chưa có kết quả đánh giá: trả về `{}`.
  - **Quy trình xử lý**:
    * Kiểm tra danh sách kết quả `results_`. Nếu rỗng, ghi cảnh báo và trả về dict trống.
    * Chuyển `results_` thành `DataFrame` để tiện tra cứu.
    * Xác định tiêu chí so sánh:
      · Nếu metric là `davies_bouldin`: chọn giá trị nhỏ nhất.
      · Ngược lại: chọn giá trị lớn nhất.

        * Lấy dòng tương ứng với mô hình tốt nhất (best row) và chuyển thành `dict`.
        * Ghi log thông tin mô hình tốt nhất.
        * Trả về `dict` kết quả.
- **Phương thức** `clear_results()`
    - **Công dụng**:
        * Xóa toàn bộ kết quả đánh giá clustering đã được lưu trong thuộc tính `results_`.
        * Giúp khởi tạo lại bộ đánh giá khi muốn chạy đánh giá mới từ đầu.
    - **Input**: Hàm không nhận tham số đầu vào (ngoài `self`).
    - **Output**:
        * Không trả về giá trị (`None`).
        * Trạng thái bên trong của đối tượng thay đổi: `results_` được làm trống.
    - **Quy trình xử lý**:
        * Gán thuộc tính `results_` thành danh sách rỗng `[]`.
        * Ghi log thông báo rằng toàn bộ kết quả đã được xóa.

3.3.12 Class TuningConfig

Cung cấp cấu hình cho quá trình tinh chỉnh siêu tham số (hyperparameter tuning), bao gồm đường dẫn dữ liệu, vị trí lưu kết quả, random state và lựa chọn metric để chọn mô hình tối ưu.

- **Thuộc tính**:
    - `data_path`: Đường dẫn tới dữ liệu đã encode.
    - `results_path`: Đường dẫn lưu file kết quả tuning.
    - `random_state`: Seed cố định cho việc khởi tạo mô hình.
    - `metric_selection`: Metric dùng để chọn mô hình tốt nhất (`silhouette`, `calinski_harabasz`, `davies_bouldin`, hoặc `composite`).
    - `silhouette_weight, calinski_weight, davies_weight`: Trọng số khi dùng composite score.
- **Output**: Một đối tượng `TuningConfig` chứa toàn bộ tham số cấu hình cho quá trình tuning.
- **Quy trình xử lý**:
    - Gán giá trị mặc định hoặc giá trị người dùng truyền vào cho từng thuộc tính.
    - Lưu lại dưới dạng data class giúp code dễ đọc, rõ ràng và tự động tạo constructor.

– Được sử dụng bởi module `Hyperparameter Tuner` để điều khiển toàn bộ pipeline tuning.

3.3.13 Class HyperparameterTuner

Lớp `HyperparameterTuner` chịu trách nhiệm tự động thử nghiệm nhiều bộ siêu tham số khác nhau cho các mô hình phân cụm, đánh giá chúng bằng `ClusteringEvaluator`, và chọn ra mô hình có hiệu suất tốt nhất. Lớp này hỗ trợ quy trình tuning có cấu hình linh hoạt thông qua `TuningConfig`.

- **Phương thức** `__init__()`
  - **Công dụng**: Khởi tạo bộ tinh chỉnh siêu tham số (hyperparameter tuner) cho bài toán phân cụm. Thiết lập cấu hình, evaluator để tính metric, hệ thống logging, vùng lưu kết quả, và các biến theo dõi mô hình tốt nhất.
  - **Input**:
    * `config`: Đối tượng `TuningConfig` chứa toàn bộ cấu hình tuning.
    * `evaluator`: Đối tượng `ClusteringEvaluator` phục vụ tính toán các metric (Silhouette, CH, DBI,...).
    * `logger` (tuỳ chọn): Logger bên ngoài truyền vào; nếu không, lớp sẽ tự tạo logger mặc định.
  - **Output**: Không trả về giá trị; khởi tạo một đối tượng `HyperparameterTuner` sẵn sàng để chạy quá trình tuning.
  - **Quy trình xử lý**:
    * Gán cấu hình và evaluator vào thuộc tính nội bộ của lớp.
    * Thiết lập `logger`:
      · Nếu không truyền vào, tạo logger mới tên `HyperparameterTuner`.
      · Thiết lập mức độ log là `INFO`.
      · Tạo `StreamHandler` với định dạng log đơn giản.
    * Khởi tạo các biến trạng thái:
      · `df_data`: nơi lưu dataset sau khi load.
      · `all_results`: danh sách lưu kết quả của tất cả tổ hợp siêu tham số đã thử.
      · Bộ biến theo dõi mô hình tốt nhất: `best_model`: mô hình phân cụm tốt nhất, `best_labels`: nhãn phân cụm ứng với mô hình tốt nhất, `best_params`: cấu hình siêu tham số của mô hình tốt nhất, `best_metric`: giá trị metric tốt nhất đạt được.
    * Hoàn tất việc khởi tạo đối tượng và sẵn sàng chạy quy trình tuning.
- **Phương thức** `load_data()`

- **Công dụng**: Tải dữ liệu đã được xử lý (encoded data) từ đường dẫn quy định trong `TuningConfig` để phục vụ cho quá trình tinh chỉnh siêu tham số (hyperparameter tuning).
- **Input**: Không có tham số đầu vào; sử dụng đường dẫn từ `self.config.data_path`.
- **Output**: Không trả về giá trị. Gán dữ liệu đã load vào biến `self.df_data` (kiểu `pandas.DataFrame`).
- **Quy trình xử lý**:
  * Ghi log thông báo rằng hệ thống bắt đầu quá trình tải dữ liệu.
  * Kiểm tra sự tồn tại của file dữ liệu:
    · Nếu file không tồn tại: ném lỗi `FileNotFoundError`.
  * Đọc file CSV bằng `pandas.read_csv` và lưu vào `self.df_data`.
  * Ghi log kích thước của dữ liệu (`shape`) sau khi tải thành công.

- **Phương thức** `_generate_param_grid()`
  - **Công dụng**: Sinh toàn bộ tập hợp các tổ hợp siêu tham số (parameter combinations) cho một mô hình cụ thể dựa trên lưới tham số do người dùng cung cấp. Phương thức hỗ trợ quá trình Grid Search thủ công cho nhiều thuật toán clustering.
  - **Input**:
    * `model_type`: tên hoặc loại mô hình (ví dụ: `"kmeans"`, `"gmm"`, `"hierarchical"`).
    * `param_grid`: từ điển mô tả lưới siêu tham số, dạng:

      ```
      {parameter_name: list_of_values}
      ```

  - **Output**: Trả về danh sách chứa tất cả tổ hợp tham số có thể, mỗi tổ hợp là một `dict`. Mỗi dict còn được bổ sung thêm trường `model_type`.
  - **Quy trình xử lý**:
    * Lấy danh sách các tên tham số (`keys`) và danh sách các giá trị tương ứng (`values`).
    * Dùng `itertools.product` để lấy tích Đề-các các giá trị nhằm tạo toàn bộ tổ hợp.
    * Ghép từng tổ hợp giá trị với `keys` để tạo ra một `dict` tham số.
    * Với mỗi dict sinh ra, thêm trường:

      ```
      c["model_type"] = model_type
      ```

    * Trả về danh sách tất cả các tổ hợp.

- **Phương thức** `run_grid_search(model_type, param_grid)`
  - **Công dụng**: Dùng để thực hiện Grid Search toàn diện cho một thuật toán phân cụm với tập tham số cho trước.

– **Input**:

  * `model_type`: tên mô hình phân cụm (ví dụ: `"kmeans"`, `"gmm"`, `"hierarchical"`).
  * `param_grid`: từ điển mô tả tập giá trị thử nghiệm cho từng siêu tham số. Ví dụ:

    {`n_clusters: [3,4,5]`, `init: ["k-means++", "random"]`}

– **Output**:

  * Không trả về giá trị.
  * Cập nhật nội bộ:
    · `self.all_results`: danh sách toàn bộ kết quả đánh giá.
    · `self.best_model`: mô hình tốt nhất.
    · `self.best_params`: bộ tham số tốt nhất.
    · `self.best_metric`: giá trị metric tốt nhất.
    · `self.best_labels`: nhãn phân cụm của mô hình tốt nhất.

– **Quy trình xử lý**:

1. **Kiểm tra dữ liệu**: Nếu dữ liệu chưa được tải, gọi `load_data()`.
2. **Sinh tổ hợp tham số**: Dùng hàm `_generate_param_grid()` để tạo danh sách tất cả các tổ hợp siêu tham số.
3. **Lặp qua từng tổ hợp**:
   (a) Tách `n_clusters` và các tham số mô hình khác.
   (b) Tạo chuỗi định danh mô hình (`model_name_id`) từ tham số.
   (c) Tạo `TrainingConfig` cho tổ hợp hiện tại.
   (d) Huấn luyện mô hình qua `ModelTrainer`.
   (e) Lấy nhãn cụm từ mô hình huấn luyện.
4. **Đánh giá mô hình**: Gọi `evaluator.evaluate_once()` để tính:

   Silhouette, Calinski-Harabasz, Davies-Bouldin

5. **Lưu kết quả**: Gộp siêu tham số và metric thành một bản ghi rồi đưa vào:

   `self.all_results.append(record)`

6. **So sánh để tìm mô hình tốt nhất**:
   * Nếu metric là `davies_bouldin` thì chọn giá trị **nhỏ nhất**.
   * Ngược lại chọn giá trị **lớn nhất**.
   * Nếu cấu hình chọn `metric_selection = "composite"` thì tạm thời dùng Silhouette để theo dõi.

7. **Xử lý lỗi**:
    * Bắt `ValueError` (lỗi tham số không hợp lệ) và tiếp tục chạy.
    * Bắt lỗi chung để tránh dừng toàn bộ quá trình.
8. **Ghi log**: In tiến trình chạy và kết quả từng mô hình theo dạng:

$$\text{Sil},\ \text{CH},\ \text{DB},\ \text{Clusters}$$

- **Phương thức** `run_all_models()`
  - **Công dụng**: Thực hiện chạy toàn bộ quá trình Grid Search cho tất cả các thuật toán phân cụm được hỗ trợ.
  - **Input**: Không có tham số đầu vào.
  - **Output**: Không trả về; cập nhật `all_results`, `best_model`, `best_labels`, `best_params`.
  - **Quy trình xử lý**:
    * Gọi `run_grid_search()` cho KMeans với các bộ siêu tham số cố định.
    * Gọi `run_grid_search()` cho GMM.
    * Gọi `run_grid_search()` cho DBSCAN.
    * Nếu thư viện HDBSCAN tồn tại, chạy tiếp Grid Search cho HDBSCAN.
    * Mỗi lần chạy sẽ lưu kết quả và cập nhật mô hình tốt nhất.
- **Phương thức** `_calculate_composite_score()`
  - **Công dụng**:
    * Thực hiện tính toán chỉ số tổng hợp (composite score) dựa trên 3 metrics đã chuẩn hóa:
      · Silhouette (0–1, càng cao càng tốt) — giữ nguyên.
      · Calinski–Harabasz (không giới hạn, càng cao càng tốt) — chuẩn hóa Min–Max.
      · Davies–Bouldin (không giới hạn, càng thấp càng tốt) — đảo giá trị rồi chuẩn hóa Min–Max.
  - **Input**: `df`: DataFrame chứa các cột `silhouette`, `calinski_harabasz`, `davies_bouldin`.
  - **Output**: `pd.Series` — chuỗi điểm composite score tương ứng từng dòng của `df`.
  - **Quy trình xử lý**:
    * Giữ nguyên giá trị Silhouette (đã nằm trong khoảng 0–1).
    * Chuẩn hoá Calinski–Harabasz bằng min–max scaling.
    * Đảo chiều Davies–Bouldin (vì thấp tốt hơn) rồi chuẩn hoá.

* Kết hợp ba giá trị chuẩn hoá theo trọng số đã cấu hình để tạo composite score.

- **Phương thức** `get_summary()`
  - **Công dụng**: Tạo bảng tổng hợp kết quả phân cụm và xếp hạng mô hình theo metric hoặc composite score.
  - **Input**: Không nhận tham số ngoài; sử dụng `self.all_results` và cấu hình trong `self.config.metric_selection`.
  - **Output**: DataFrame đã được sắp xếp theo metric được chọn (Silhouette, Calinski-Harabasz, Davies-Bouldin hoặc composite score).
  - **Quy trình xử lý**:
    * Kiểm tra nếu chưa có kết quả thì trả về DataFrame rỗng.
    * Chuyển `self.all_results` thành DataFrame.
    * Nếu dùng composite score:
      · Gọi `_calculate_composite_score()` để tạo cột `composite_score`.
      · Sắp xếp giảm dần theo composite score.
    * Nếu dùng single metric:
      · Đặt thứ tự sắp xếp: Davies–Bouldin (tăng dần), các metric khác (giảm dần).
      · Sắp xếp DataFrame theo metric đã chọn.
- **Phương thức** `save_results()`
  - **Công dụng**: Lưu kết quả tuning ra CSV, hiển thị Top 3 mô hình và cập nhật `self.best_params` nếu dùng composite score.
  - **Input**: Không nhận tham số; sử dụng `self.all_results` và `self.config.results_path`.
  - **Output**: File CSV kết quả, log Top 3 mô hình, cập nhật `self.best_params`.
  - **Quy trình xử lý**: Lấy bảng tóm tắt (sort theo metric/composite), lưu CSV, log Top 3; nếu dùng composite, lấy dòng đầu tiên, truy ngược `self.all_results` và cập nhật `self.best_params`.
- **Phương thức** `save_best_model_and_df()`
  - **Công dụng**: Lưu mô hình tốt nhất (.pkl) và xuất DataFrame gốc kèm cột `cluster` ra CSV.
  - **Input**: `model_path` (mặc định `results/best_model.pkl`), `df_path` (mặc định `results/clustered_data.csv`).
  - **Output**: File mô hình (.pkl) và file CSV dữ liệu kèm nhãn cụm.
  - **Quy trình xử lý**: Kiểm tra `self.best_model` và `self.best_labels`; tạo thư mục nếu cần; lưu mô hình; thêm cột `cluster` vào DataFrame và xuất CSV.

4 Thư viện mô hình máy học và tối ưu tham số

4.1 Thư viện mô hình máy học

Dự án sử dụng các thư viện machine learning chính sau:

- **scikit-learn (>=1.0.0)**: Thư viện học máy phổ biến nhất của Python, cung cấp:
  - Các thuật toán phân cụm: KMeans, DBSCAN, GaussianMixture
  - Công cụ tiền xử lý: StandardScaler, OneHotEncoder, OrdinalEncoder
  - Metrics đánh giá: Silhouette Score, Calinski-Harabasz Index, Davies-Bouldin Index
  - Công cụ giảm chiều dữ liệu: PCA, t-SNE
- **HDBSCAN (>=0.8.0)**: Thư viện chuyên biệt cho thuật toán HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise), cải tiến của DBSCAN với khả năng:
  - Tự động xác định số lượng cụm
  - Xử lý hiệu quả dữ liệu có mật độ thay đổi
  - Phát hiện outliers và noise points
- **NumPy (>=1.21.0)**: Thư viện tính toán số học hiệu năng cao, hỗ trợ:
  - Xử lý mảng đa chiều (N-dimensional arrays)
  - Các phép toán đại số tuyến tính
  - Tính toán vectorized để tăng tốc độ
- **pandas (>=1.3.0)**: Thư viện phân tích và xử lý dữ liệu:
  - Đọc/ghi file CSV, Excel
  - DataFrame cho thao tác dữ liệu dạng bảng
  - Merge, join, group-by các dataset

4.2 Các mô hình sử dụng

Dự án triển khai pipeline linh hoạt hỗ trợ 4 thuật toán phân cụm khác nhau, được cấu hình thông qua tham số `model_type` trong class `TrainingConfig`:

4.2.1 KMeans Clustering

- **Thuật toán**: Phân cụm dựa trên khoảng cách Euclidean, tối thiểu hóa tổng bình phương khoảng cách trong cụm (inertia)
- **Ưu điểm**: Đơn giản, nhanh, hiệu quả với cụm hình cầu
- **Hyperparameters chính**:
  - `n_clusters`: Số lượng cụm (grid search: 3-10)

  - `n_init`: Số lần khởi tạo centroid (10, 20)
  - `max_iter`: Số vòng lặp tối đa (300)
  - `init`: Phương pháp khởi tạo (k-means++)
- **Kết quả**: KMeans với `n_clusters=5` đạt Composite Score cao nhất (0.82), được chọn làm mô hình tốt nhất

4.2.2 Gaussian Mixture Model (GMM)

- **Thuật toán**: Mô hình xác suất dựa trên Expectation-Maximization (EM), giả định dữ liệu là hỗn hợp của các phân phối Gaussian
- **Ưu điểm**: Soft clustering (xác suất thuộc cụm), linh hoạt với hình dạng cụm ellipsoidal
- **Hyperparameters chính**:
  - `n_clusters`: Số component Gaussian (3-8)
  - `covariance_type`: Dạng ma trận hiệp phương sai (full, tied, diag, spherical)
  - `n_init`: Số lần khởi tạo (10)
  - `max_iter`: Số vòng lặp EM (200)
- **Ứng dụng**: Phù hợp khi cụm có hình dạng phức tạp hoặc cần xác suất membership

4.2.3 DBSCAN (Density-Based Spatial Clustering)

- **Thuật toán**: Phân cụm dựa trên mật độ, không yêu cầu số cụm trước
- **Ưu điểm**: Phát hiện cụm hình dạng bất kỳ, tự động loại bỏ outliers
- **Hyperparameters chính**:
  - `eps`: Bán kính vùng lân cận (0.5-4.0)
  - `min_samples`: Số điểm tối thiểu để tạo cụm (5, 10, 15, 20)
  - `metric`: Độ đo khoảng cách (euclidean, manhattan)
- **Thách thức**: Nhạy cảm với lựa chọn `eps`, khó khăn với dữ liệu nhiều chiều

4.2.4 HDBSCAN (Hierarchical DBSCAN)

- **Thuật toán**: Cải tiến DBSCAN với phân cấp dựa trên cây khung tối thiểu (Minimum Spanning Tree)
- **Ưu điểm**: Tự động chọn số cụm, xử lý cụm có mật độ khác nhau
- **Hyperparameters chính**:
  - `min_cluster_size`: Kích thước cụm tối thiểu (10, 20, 30, 50, 100)
  - `min_samples`: Số điểm lân cận để ước lượng mật độ (10, 20)
  - `metric`: Độ đo khoảng cách (euclidean, manhattan)

- `cluster_selection_method`: Phương pháp chọn cụm (eom - excess of mass)

- **Ứng dụng**: Phù hợp với dữ liệu phức tạp, nhiều mức mật độ khác nhau

4.3 Thư viện tối ưu tham số

4.3.1 Grid Search tùy chỉnh

Dự án triển khai module `tuning.py` với class `HyperparameterTuner` để tự động hóa grid search cho tất cả các mô hình. Điểm đặc biệt là hệ thống hỗ trợ 4 chế độ lựa chọn metric:

1. **Silhouette Score**: Đo độ tương đồng trong cụm và phân tách giữa các cụm ([-1, 1], càng cao càng tốt)
2. **Calinski-Harabasz Index**: Tỷ lệ phương sai giữa cụm và trong cụm (unbounded, càng cao càng tốt)
3. **Davies-Bouldin Index**: Trung bình khoảng cách giữa các cụm (unbounded, càng thấp càng tốt)
4. **Composite Score**: Kết hợp cả 3 metrics với trọng số có thể tùy chỉnh.

$$\text{Composite} = w_{\text{Sil}} \cdot \text{Sil}_{\text{norm}} + w_{\text{CH}} \cdot \text{CH}_{\text{norm}} + w_{\text{DB}} \cdot \text{DB}_{\text{norm}} \tag{1}$$

4.3.2 Không gian tìm kiếm hyperparameter

Bảng 1 mô tả không gian tìm kiếm cho từng model:

Bảng 1: Không gian Grid Search cho các mô hình

Model	**Hyperparameter**	**Số cấu hình**
KMeans	n_clusters: [3, 4, 5, 6, 7, 8, 9, 10] (8 giá trị) init: [k-means++] (1 giá trị) n_init: [10, 20] (2 giá trị) max_iter: [300] (1 giá trị)	$8 \times 1 \times 2 \times 1 = 16$
GMM	[h] n_clusters: [3, 4, 5, 6, 7, 8] (6 giá trị) covariance_type: [full, tied, diag, spherical] (4 giá trị) n_init: [10] (1 giá trị) max_iter: [200] (1 giá trị)	$6 \times 4 \times 1 \times 1 = 24$
DBSCAN	eps: [0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0] (8 giá trị) min_samples: [5, 10, 15, 20] (4 giá trị) metric: [euclidean, manhattan] (2 giá trị)	$8 \times 4 \times 2 = 64$
HDBSCAN	min_cluster_size: [10, 20, 30, 50, 100] (5 giá trị) min_samples: [10, 20] (2 giá trị) metric: [euclidean, manhattan] (2 giá trị) cluster_selection_method: [eom] (1 giá trị)	$5 \times 2 \times 2 \times 1 = 20$
Tổng		**124 cấu hình**

4.3.3 Thư viện hỗ trợ khác

- **joblib (>=1.1.0)**: Lưu/load models dưới dạng .pkl file với nén hiệu quả
- **itertools**: Tạo tổ hợp tham số cho grid search (Cartesian product)
- **tqdm (>=4.62.0)**: Progress bar để theo dõi quá trình tuning

5 Phân tích kết quả thí nghiệm

5.1 Kết quả mô hình tốt nhất

Sau khi thực hiện grid search với 124 cấu hình trên 4 thuật toán clustering, mô hình KMeans với 3 cụm được chọn làm mô hình cuối cùng dựa trên composite score. Bảng 2 trình bày các chỉ số đánh giá của mô hình này:

Bảng 2: Kết quả đánh giá mô hình KMeans tốt nhất

Metric	**Giá trị**
Model	KMeans
Số cụm (n_clusters)	3
Silhouette Score	0.2544
Calinski-Harabasz Index	1269.22
Davies-Bouldin Index	1.5338
Composite Score	**0.7018**

- **Silhouette Score (0.2544)**: Giá trị dương cho thấy các điểm dữ liệu được gán vào đúng cụm của chúng. Mặc dù không cao (khoảng 0-1), điều này phản ánh đặc điểm thực tế của dữ liệu khách hàng có sự chồng lấn nhất định giữa các nhóm, không tách biệt hoàn toàn.
- **Calinski-Harabasz Index (1269.22)**: Giá trị cao cho thấy tỷ lệ tốt giữa phương sai giữa các cụm (between-cluster variance) và phương sai trong cụm (within-cluster variance), chứng tỏ các cụm được phân tách rõ ràng.
- **Davies-Bouldin Index (1.5338)**: Giá trị càng thấp càng tốt (tối ưu là 0). Với 1.5338, cho thấy độ tương đồng giữa các cụm ở mức chấp nhận được, các cụm có sự phân biệt nhất định.
- **Composite Score (0.7018)**: Điểm tổng hợp cân bằng cả 3 metrics trên với trọng số $0.4 \times \text{Sil} + 0.3 \times \text{CH} + 0.3 \times \text{DB}$, đạt 0.7018 cho thấy mô hình có chất lượng tốt tổng thể.

5.2 Đặc điểm các phân khúc khách hàng

Mô hình KMeans với 3 cụm đã phân chia 3685 khách hàng Highlands Coffee thành 3 nhóm với đặc điểm rõ rệt.

5.2.1 Cluster 0 – Nhóm Trung Niên Ghé Nhiều Nhưng Chi Ít

Đây là phân khúc khách hàng có **độ tuổi cao nhất** (khoảng 40 tuổi), đặc trưng bởi hành vi ghé quán thường xuyên nhưng chi tiêu hạn chế:

- **Hành vi tiêu dùng**:
  - Tần suất ghé quán cao nhất trong 3 nhóm
  - Chi tiêu trên mỗi lần (PPA - Price Per Average) thấp
  - Tổng chi tiêu (Total Spending) thấp
  - Giá trị đơn hàng nhỏ, thường chỉ mua đồ uống cơ bản
- **Thái độ thương hiệu**:
  - NPS (Net Promoter Score) thấp → mức độ hài lòng kém
  - Ưa chuộng Independent Cafe (quán địa phương) hơn Highlands
  - Ít có khả năng giới thiệu thương hiệu cho người khác
- **Đặc điểm địa lý**: Tập trung chủ yếu tại **Cần Thơ**
- **Insight kinh doanh**: Nhóm này có thể là khách hàng "ghé qua"hoặc sử dụng Highlands như một lựa chọn tiện lợi hơn là yêu thích thương hiệu. Chiến lược retention cần tập trung vào tăng engagement và tạo giá trị cảm nhận.

5.2.2 Cluster 1 – Nhóm Trẻ Chi Tiêu Cao, Hài Lòng Cao

Phân khúc khách hàng **trẻ nhất** (khoảng 33 tuổi) và có giá trị cao nhất về mặt doanh thu:

- **Hành vi tiêu dùng**:
  - Chi tiêu mỗi lần cao nhất (high PPA)
  - Tổng chi tiêu cao nhất trong 3 cụm
  - Tần suất ghé quán thấp hơn Cluster 0
  - Xu hướng mua combo hoặc sản phẩm cao cấp
- **Thái độ thương hiệu**:
  - NPS rất cao → hài lòng mạnh với thương hiệu
  - Highlands là thương hiệu yêu thích (top of mind)
  - Khả năng giới thiệu thương hiệu cao
  - Khách hàng trung thành (brand advocates)
- **Đặc điểm địa lý**: Tập trung chủ yếu tại **Hà Nội**
- **Insight kinh doanh**: Đây là phân khúc "vàng"(golden segment) cần được ưu tiên trong chiến lược marketing và loyalty programs. Mặc dù tần suất thấp hơn, giá trị mỗi giao dịch cao giúp họ mang lại doanh thu lớn nhất.

5.2.3 Cluster 2 – Nhóm Hài Lòng Cao Nhưng Chi Tiêu Thấp

Phân khúc có **độ tuổi trung bình**, đặc trưng bởi sự hài lòng cao nhưng chi tiêu hạn chế:

- **Hành vi tiêu dùng**:
  - Chi tiêu thấp (cả PPA và Total Spending)
  - Tần suất ghé quán ở mức trung bình – thấp
  - Có thể là sinh viên hoặc nhóm thu nhập hạn chế
- **Thái độ thương hiệu**:
  - NPS cao nhất trong 3 nhóm
  - Mặc dù hài lòng nhưng vẫn chủ yếu thích Independent Cafe hơn Highlands
  - Có tiềm năng chuyển đổi thành khách hàng trung thành nếu được tiếp cận đúng cách
- **Đặc điểm địa lý**: Tập trung chủ yếu tại **TP.HCM**
- **Insight kinh doanh**: Đây là nhóm có tiềm năng tăng trưởng. Mặc dù hiện tại chi tiêu thấp, NPS cao cho thấy họ có thiện cảm với thương hiệu. Chiến lược giá (pricing) và chương trình khuyến mãi phù hợp có thể chuyển họ thành khách hàng có giá trị cao hơn.

Bảng 3: Ma trận Giá trị - Trung thành của 3 cụm

	NPS Cao	**NPS Thấp**
Chi tiêu Cao	**Cluster 1** (Vàng)	-
Chi tiêu Thấp	**Cluster 2** (Tiềm năng)	**Cluster 0** (Rủi ro)