BỘ NÔNG NGHIỆP VÀ MÔI TRƯỜNG

PH TRƯỜNG ĐẠI HỌC THỦY LỢI

BM CÔNG NGHỆ THÔNG TIN

BÁO CÁO MÔN HỌC MÁY

Tên đề tài:

DỰ ĐOÁN VẤN ĐỀ VỀ TIM MẠCH

Giảng viên hướng dẫn:	***Vũ Thị Hạnh***
Sinh viên thực hiện:	***Nguyễn Minh Thiện*** ***Nguyễn Ngọc Châu Giang*** ***Nguyễn Thị Yến Ngọc***
Mssv:	***2351067116*** ***2351067092*** ***2351067103***
Lớp:	***S26-65CNTT***

2025

Lời Cám Ơn

Để hoàn thành được bài tiểu luận này, em xin chân thành cảm ơn Ban Giám hiệu, các khoa, phòng và quý thầy, cô của trường Phân hiệu Đại học Thuỷ Lợi, những người đã tận tình giúp đỡ và tạo điều kiện cho em trong quá trình học tập. Đặc biệt, em xin gửi lời cảm ơn sâu sắc đến cô Vũ Thị Hạnh - người đã trực tiếp giảng dạy và hướng dẫn em thực hiện bài tiểu luận này bằng tất cả lòng nhiệt tình và sự quan tâm sâu sắc.

Trong quá trình thực hiện bài tiểu luận này, do hiểu biết còn nhiều hạn chế nên bài làm khó tránh khỏi những thiếu sót. Em rất mong nhận được những lời góp ý của quý thầy cô để bài tiểu luận ngày càng hoàn thiện hơn.

Em xin chân thành cảm ơn!

MỤC LỤC

MỤC LỤC HÌNH ẢNH

NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

TPHCM, ngày tháng năm 2025

Ký và ghi rõ họ tên

Vũ Thị Hạnh

I.Tổng quan:

1. Giới thiệu đề tài:

Các vấn đề về tim mạch đang ngày càng gia tăng và là nguyên nhân chính gây ảnh hưởng nghiêm trọng đến sức khỏe con người. Việc dự đoán sớm nguy cơ mắc các vấn đề tim mạch có vai trò quan trọng trong việc hỗ trợ bác sĩ đưa ra quyết định chẩn đoán và điều trị kịp thời.

Trong đề tài này, dữ liệu bệnh tim được sử dụng để xây dựng các mô hình Machine Learning nhằm dự đoán khả năng gặp vấn đề về tim mạch của bệnh nhân. Ba mô hình được áp dụng và so sánh gồm Decision Tree, Gradient Boosting và MLPClassifier. Quá trình thực hiện bao gồm phân tích dữ liệu, tiền xử lý, huấn luyện mô hình và đánh giá hiệu quả thông qua các chỉ số như Accuracy, Precision, Recall, F1-score và ROC-AUC.

2. Mục tiêu và bài toán đặt ra:

- Mục tiêu:
    - Xây dựng hệ thống dự đoán nguy cơ mắc vấn đề về tim mạch dựa trên dữ liệu y tế của bệnh nhân.
    - Áp dụng và so sánh hiệu quả của ba mô hình học máy: Decision Tree, Gradient Boosting và MLPClassifier.
    - Đánh giá mô hình dựa trên các chỉ số như Accuracy, Precision, Recall, F1-score và ROC-AUC để xác định mô hình phù hợp nhất.
- Bài toán đặt ra:
    - Đầu vào: Các đặc trưng y tế của bệnh nhân (tuổi, giới tính, huyết áp, cholesterol, nhịp tim,….).
    - Đầu ra: Dự đoán bệnh nhân có hoặc không gặp vấn đề về tim mạch (1: có vấn đề, 0: là không vấn đề).

II. Thực nghiệm:

1. Mô tả dữ liệu

Bộ dữ liệu Heart là một tập dữ liệu chi tiết các đặc trưng về bệnh tim. Đây là một nguồn dữ liệu quan trọng trong lĩnh vực y tế nhằm dự đoán bệnh nhân có mắc các căn bệnh về tim hay không?, từ đây bệnh nhân có thể tránh các trường hợp để bệnh đến mức báo động. Bộ dữ liệu này được cung cấp trên Kaggle, một nền tảng chuyên về kho dữ liệu trong lĩnh vực khoa học dữ liệu.

-Thông tin cơ bản:

- Nguồn dữ liệu: Kaggle
    - Link dataset: https://www.kaggle.com/datasets/tan5577/heart-failure-dataset

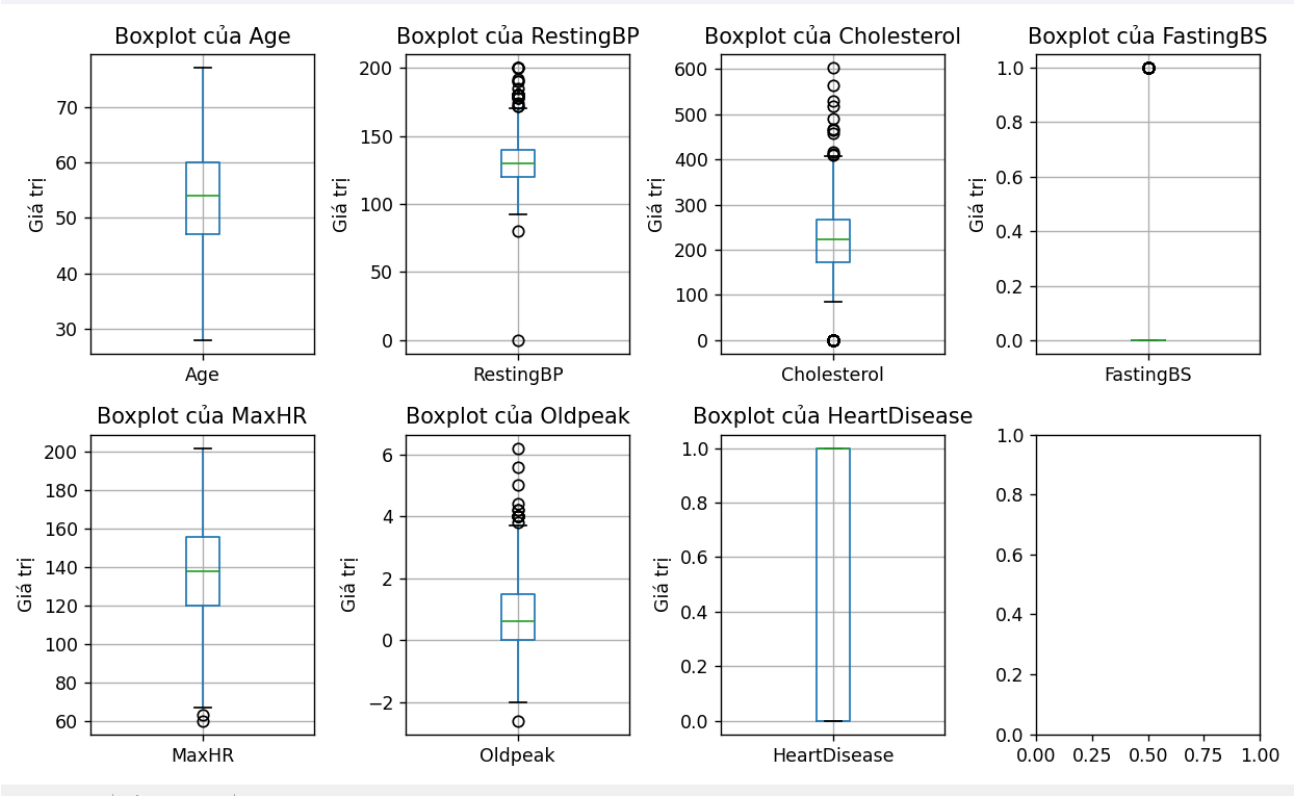
-Cấu trúc dữ liệu:

Bộ dữ liệu bao gồm 12 đặc trưng, chủ yếu là các đặc trưng liên quan để tim mạch. Dưới đây là các đặc trưng:

Tên đặc trưng	Mô tả	Dữ liệu
Age	Tuổi	int
Sex	Giới tính	M, F
ChestPainType	Loại đau ngực	TA, ASY, ATA, NAP
RestingBP	Huyết áp lúc nghỉ	int
Cholesterol	Cholesterol máu	int
FastingBS	Đường huyết lúc đói (>120 mg/dl = 1)	int
RestingECG	Kết quả điện tâm đồ	Normal, LVH, ST
MaxHR	Nhịp tim tối đa	int
ExerciseAngina	Đau thắt ngực khi vận động	N, Y
Oldpeak	Độ chênh ST	float

ST_Slope	Độ dốc đoạn ST	Flat, Up, Down
Heartdisease		int

a. Phân tích ngoại lai:



Hình 1 Biểu đồ phân tích ngoại lai

Từ biểu đồ ta có thể thấy được 5 đặc trưng không được hiển thị bao gồm: Sex, ChestPainType, RestingECG, ExerciseAngina, ST_Slope. Vì trong bộ dữ liệu Heart 5 đặc trưng trên là dạng chuỗi. Từ đây ta có thể dùng one hot encode hoặc label encode để xử lý dữ liệu chuỗi.

- 7 đặc trưng còn lại:
  - Age: phân bố khá đối xứng, không xuất hiện ngoại lai mang tính nghiêm trọng,

- RestingBP: các giá trị phần lớn nằm trong khoảng 120-140 mmHg, tuy nhiên có một số ngoại lai ở phía trên (>170mmHg)
- Cholesterol: phân bố trong khoảng 180 - 250 mg/dL, các giá trị lệch về phía bên phải đồng thời cũng có nhiều dữ liệu ngoại lai có giá trị cao bất thường (>400 mg/dL)
- MaxHR: phân bối khá đối xứng, tuy nhiên cũng có một số ngoại lai thấp (120-158)
- Oldpeak: Phân bố lệch phải nhiều, và đồng thời xuất hiện nhiều giá trị ngoại lai cao (>3.8)
- Trong đó có 2 đặc trưng mang biến 1/0 gồm: HeartDisease và FastingBS

- Sử dụng IQR để phát hiện ngoại lai theo công thức sau:

$$x < Q1 - 1.5 \times IQR \quad \text{hoặc} \quad x > Q3 + 1.5 \times IQR$$

Các biến số liên tục trong bộ dữ liệu Heart xuất hiện một số giá trị ngoại lai nằm chủ yếu ở các chỉ số sinh lý như huyết áp, cholesterol và độ chênh lệch ST. Tuy nhiên, trong bối cảnh y khoa thì các ngoại lai này phản ánh tình trạng bệnh lý thực tế và do đó không thể loại bỏ trong quá trình tiền xử lý dữ liệu.

```
 PHÂN TÍCH OUTLIERS BẰNG IQR:
RestingBP: 28 outliers (3.05%)
Cholesterol: 183 outliers (19.93%)
FastingBS: 214 outliers (23.31%)
MaxHR: 2 outliers (0.22%)
Oldpeak: 16 outliers (1.74%)
Using target: HeartDisease
```

b. Phân phối dữ liệu (biểu đồ Histogram)

Biểu đồ dùng để thể hiện phân phối dữ liệu của các biến số trong dataset như: tuổi, huyết áp, cholesterol, nhịp tim,…. Mỗi cột cho biết số lượng mẫu dữ liệu (tần suất) nằm trong khoảng giá trị nhất định.

Biểu đồ giúp ta quan sát dữ liệu phân bố đều hay lệch, phát hiện dữ liệu thất thường (outliers).

Kết quả chạy:

```
Using target: HeartDisease

Dtypes:
 Age                int64
Sex                object
ChestPainType      object
RestingBP           int64
Cholesterol         int64
FastingBS           int64
RestingECG         object
MaxHR               int64
ExerciseAngina     object
Oldpeak           float64
ST_Slope           object
HeartDisease        int64
dtype: object
```

- Chương trình tự động dò trong các tên có thể là biến mục tiêu, trong Dataset có cột HeartDisease, nên hệ thống chọn cột này làm biến mục tiêu (HeartDisease = 1: có bệnh tim, HeartDisease = 0: không có bệnh tim).
- Dataset đa dạng và phù hợp cho nhiều mô hình như Decision Tree, Gradient Boosting….

```
Missing values per column:
 Age               0
Sex                0
ChestPainType      0
RestingBP          0
Cholesterol        0
FastingBS          0
RestingECG         0
MaxHR              0
ExerciseAngina     0
Oldpeak            0
ST_Slope           0
HeartDisease       0
dtype: int64
```

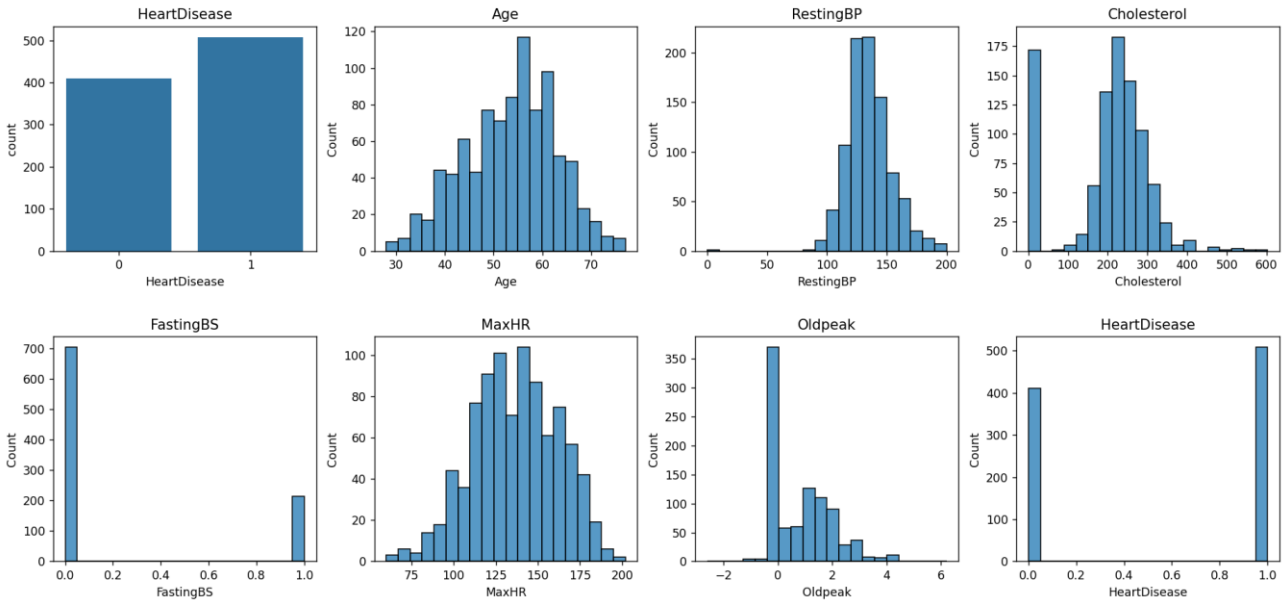
- Không có giá trị bị thiếu trong bất kì cột nào.

```
Target counts:
 HeartDisease
1    508
0    410
Name: count, dtype: int64
```

- Tổng số mẫu là 918, số người mắc bệnh tim là 508, số người không mắc bệnh tim là 410.

```
Target proportion:
 HeartDisease
1    0.553
0    0.447
Name: proportion, dtype: float64
```

- Kết quả trên thể hiện 55.3% là có bệnh tim, 44.7% là không có bệnh tim.



Hình 2 Biểu đồ phân phối dữ liệu

Trục hoành là giá trị của biến, trục tung là tần suất xuất hiện.

Kết quả cho thấy:

- Phân phối của Age (tuổi)
  - Độ tuổi có vấn đề về tim mạch tập trung nhiều nhất rơi vào khoảng 40-65 tuổi.
  - Có xuất hiện ở độ tuổi trẻ hơn hay già hơn, nhưng cũng ít hơn.
  - Dataset chủ yếu thu thập từ người trung niên nên phù hợp với nghiên cứu bệnh tim.
- Phân phối của RestingBP (huyết áp lúc nghỉ)
  - Phần lớn giá trị nằm trong khoảng 120-150.
  - Có một số giá trị cao bất thường (>180) có thể là outliers
- Phân phối của Cholesterol
  - Giá trị tập trung nhiều trong khoảng 200-300, phân phối lệch phải.
  - Có 1 giá trị cao bất thường, có thể là outliers.
- Phân phối của FastingBS (đường huyết lúc đói)
  - Dữ liệu chỉ có 2 giá trị là 0 và 1. Giá trị o chiếm đa số.
  - Đây là biến nhị phân, cho thấy ít người có đường huyết cao.

- Phân phối của MaxHR (nhịp tim tối đa)

o Giá trị phân bố tương đối khá đều, chủ yếu nằm trong khoảng 110-170

o Dữ liệu ổn định, ít outliers.

- Phân phối của Oldpeak

o Phần lớn giá trị nằm gần 0, phân phối lệch phải rõ rệt.

o Đa số bệnh nhân không có hoặc ít dấu hiệu. Một số bệnh nhân có chỉ số cao dẫn đến nguy cơ vấn đề về tim mạch.

- Phân phối của HeartDisease

o Chỉ có 2 giá trị là 0(không bệnh) và 1(có bệnh). Số lượng hai nhóm tương đối bằng nhau.

➢ Biểu đồ phân phối dữ liệu cho thấy các biến số trong tập dữ liệu có phân bố hợp lý, phản ánh đúng đặc điểm sinh lý và y học của bệnh tim.

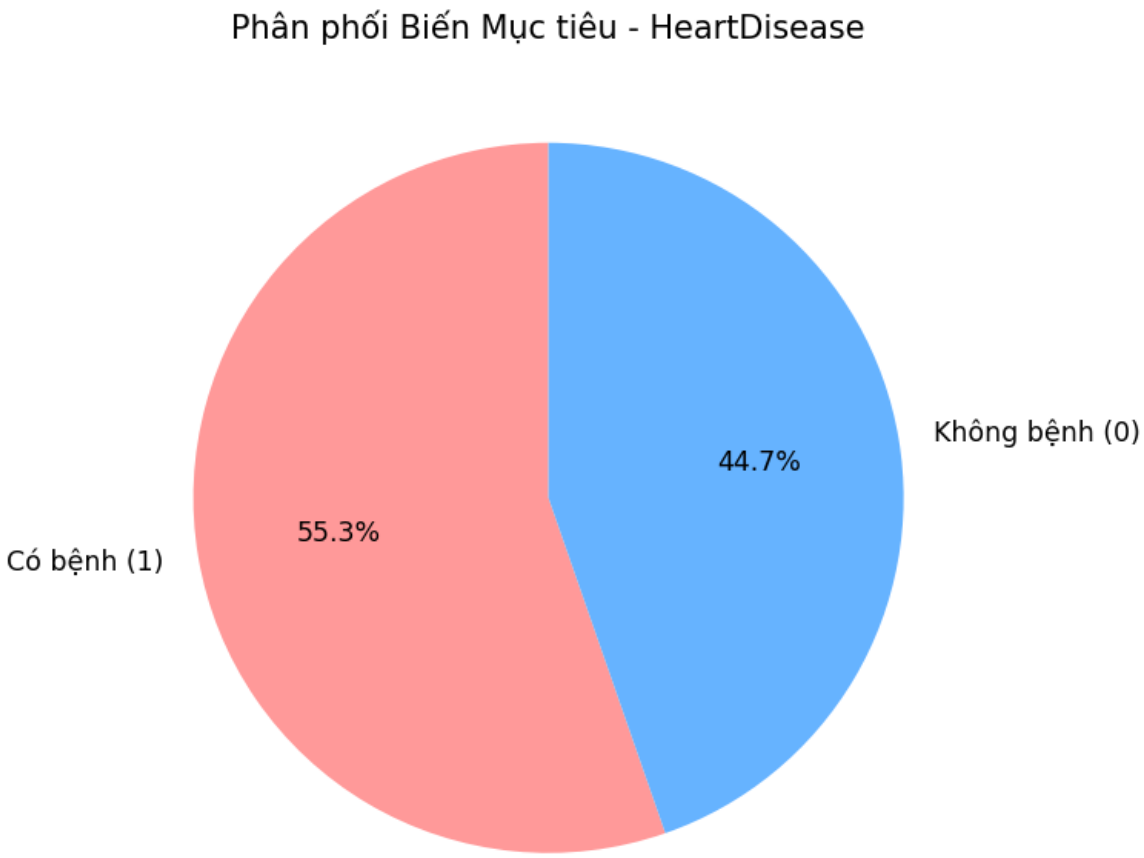
c. Phân phối theo mục tiêu – HeartDisease (biểu đồ tròn – Pie Chart)

Biểu đồ thể hiện tỷ lệ người không có bệnh tim (0), tỷ lệ người có bệnh tim (1). Đồng thời cũng kiểm tra xem Dataset có cân bằng hay mất cân bằng không. Tỷ lệ người mắc bệnh tim trong dữ liệu chiếm bao nhiêu %.

Nếu như có sự chênh lệch về dữ liệu quá lớn (ví dụ như 90% không bệnh và 10% có bệnh) thì mô hình sẽ dự đoán là không bệnh cho tất cả. Do đó biểu đồ này giúp đưa ra phần trăm và kết quả chính xác.

Kết quả chạy:

```
==================================================
 PHÂN TÍCH BIẾN MỤC TIÊU (HeartDisease)
==================================================

 Số lượng cụ thể:
• Có bệnh tim (1): 508 (55.3%)
• Không bệnh tim (0): 410 (44.7%)
```



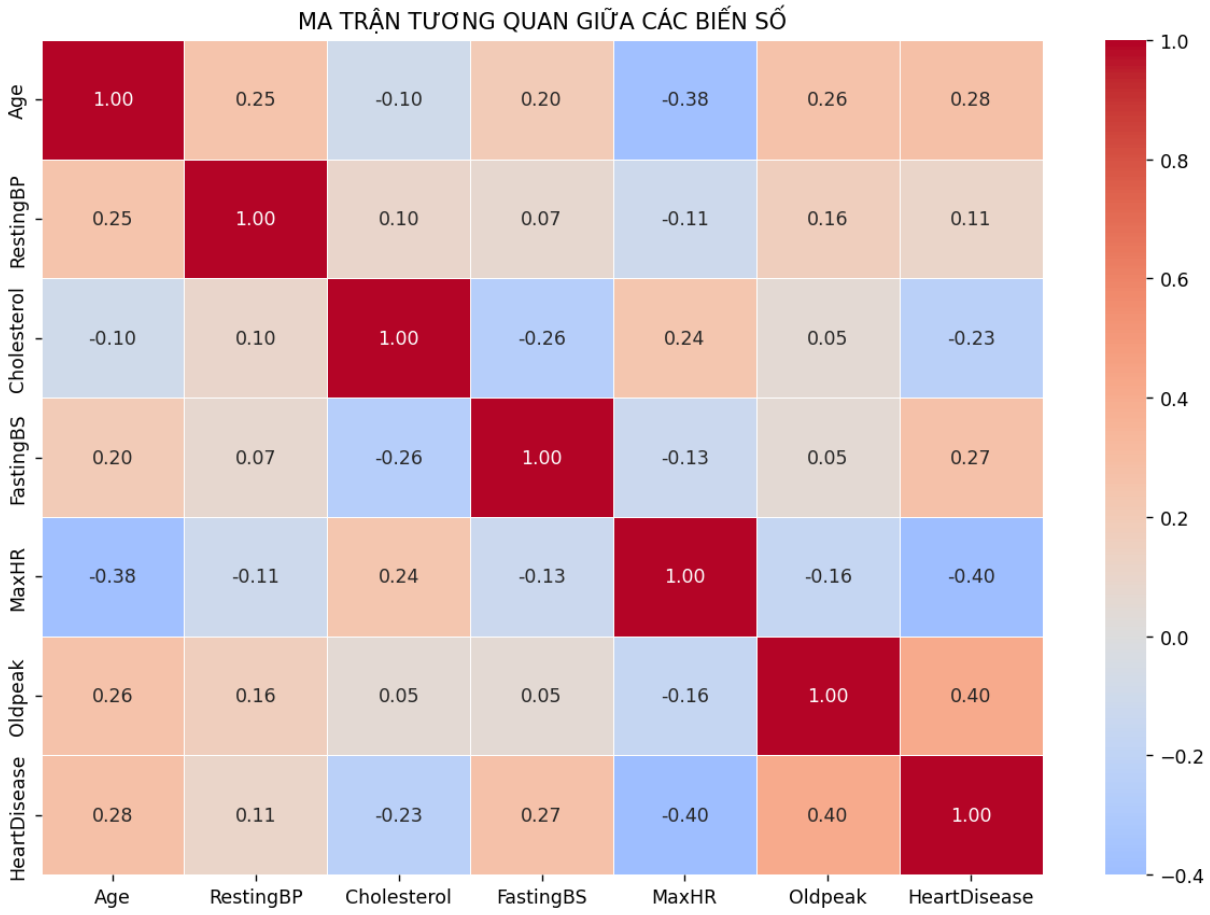
Hình 3 Biểu đồ phân phối biến mục tiêu

- Kết quả cho thấy:

  - Người mắc bệnh tim (1): 508 mẫu - chiếm 55.3%
  - Người không mắc bệnh tim (0): 410 mẫu - chiếm 44.7%
  - Nhóm người mắc bệnh tim chiếm tỷ lệ cao hơn so với nhóm người không mắc bệnh tim

  ➢ Tập dữ liệu hai lớp tương đối cân bằng nhau, không xảy ra tình trạng mất cân bằng nghiêm trọng.

d. Ma trận tương quan:



Hình 4 Ma trận tương quan

Sự tương quan giữa các biến với HeartDisease

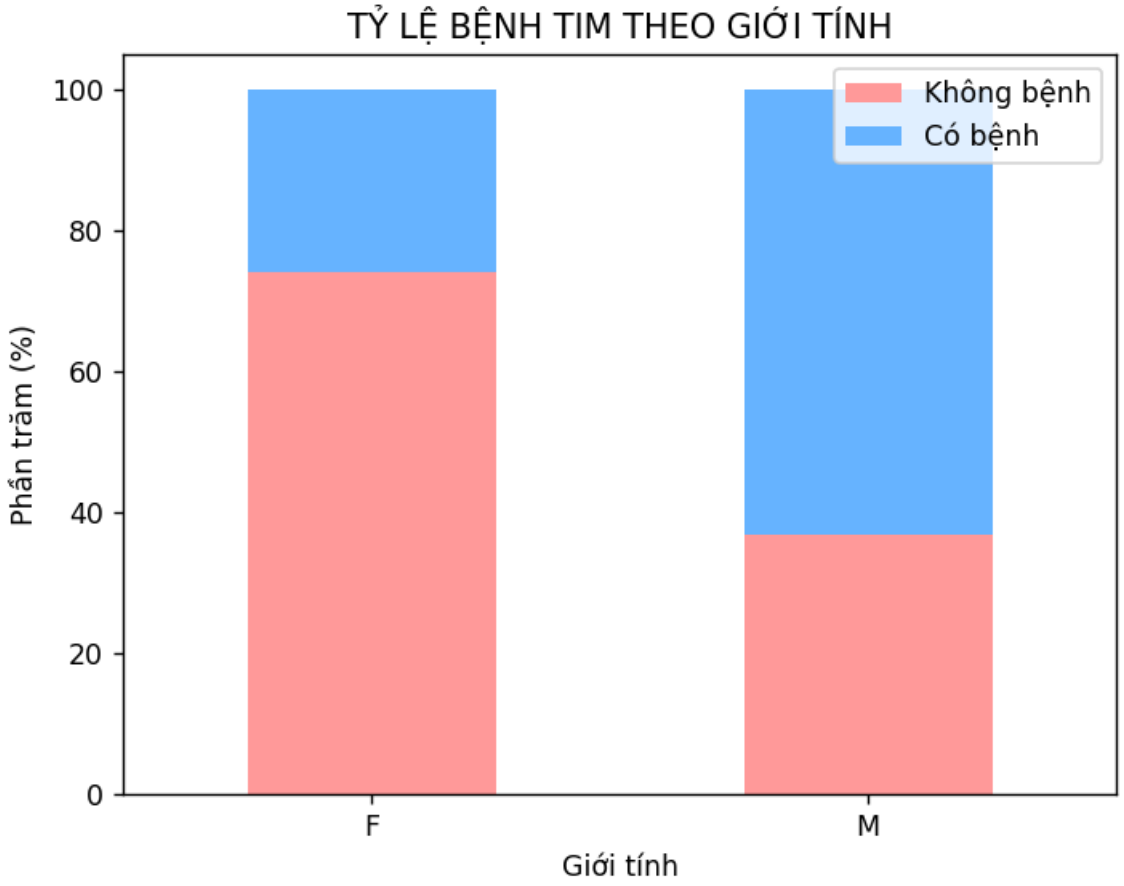
- Oldpeak có sự tương quan mạnh đối với HeartDisease: nếu oldpeak tăng thì nguy cơ bệnh tăng cao
- MaxHR có sự tương quan mạnh với HeartDisease: nếu MaxHR thấp thì nguy cơ bệnh tim cao
- Age có sự tương quan trung bình với HeartDisease
- Cholesterol có sự tương quan thấp với HeartDisease chứng tỏ mối quan hệ không rõ ràng giữa 2 yếu tố
- RestingBP có tương quan rất thấp với HeartDisease, vậy đặc trưng RestingBP cần có sự kết hợp với các yếu tố khác để có thể quyết định đến đặc trưng HeartDisease

Bài toán mang tính phi tuyến tính nên khả năng phải sử dụng đến model học được các quan hệ phức tạp như MLPClassifier.

e. Phân tích theo giới tính:

Biểu đồ thể hiện tỷ lệ không mắc bệnh tim hay có mắc bệnh tim theo giới tính Nam (M) và Nữ (F). Biểu đồ cho biết giới tính nào có nguy cơ mắc bệnh tim cao hơn.

```
==================================================
 PHÂN TÍCH THEO GIỚI TÍNH
==================================================

 Phân phối giới tính:
• Nam (M): 725 (79.0%)
• Nữ (F): 193 (21.0%)

 Tỷ lệ bệnh theo giới tính:
HeartDisease          0          1
Sex
F             74.093264  25.906736
M             36.827586  63.172414
```



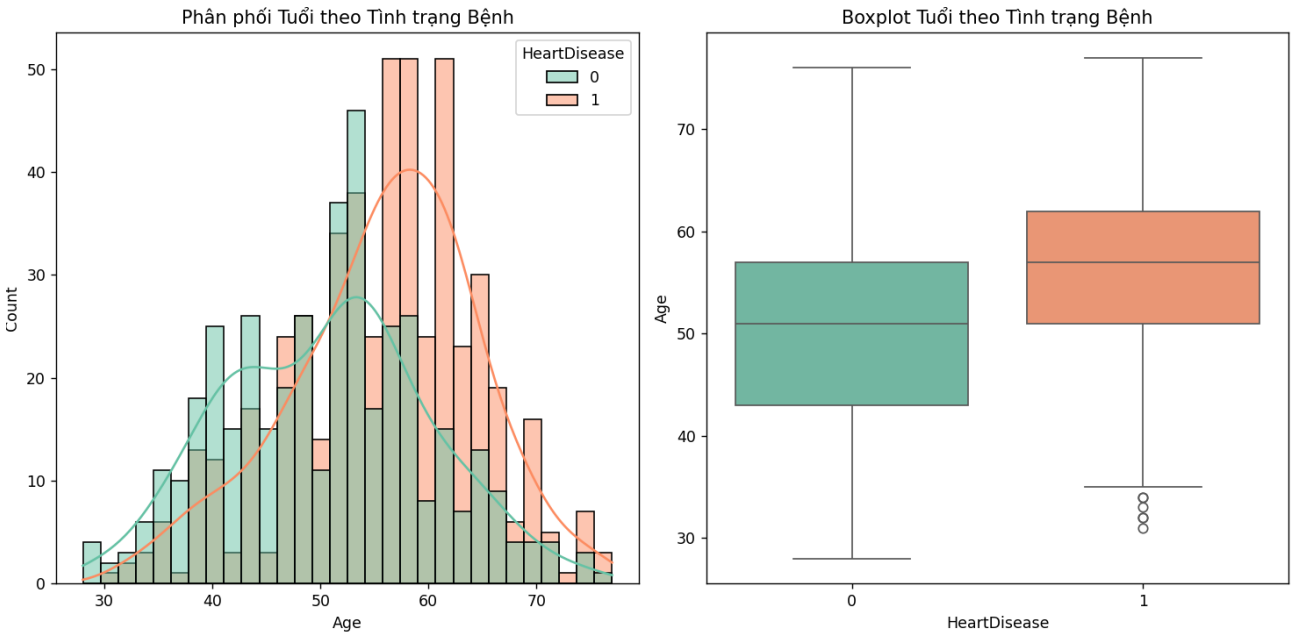
Hình 5 Biểu đồ phân tích theo giới tính

- Kết quả cho thấy:
    - Dataset thiên về nam (725 người – chiếm 79%) hơn với nữ (193 người – chiếm 21%)
    - Khoảng 60% nam giới trong dữ liệu mắc bệnh tim, trong khi nữ giới chỉ khoảng 25% (2/3 nữ giới không bị bệnh tim). Tỷ lệ nam giới mắc bệnh tim cao hơn nữ giới.

➢ Biểu đồ cho thấy tỷ lệ mắc bệnh tim ở nam giới (khoảng 60%) cao hơn nhiều so với nữ giới (khoảng 25%). Điều này chứng tỏ giới tính là một yếu tố có ảnh hưởng đáng kể đến khả năng mắc bệnh tim.

f. Phân tích tình trạng bệnh theo độ tuổi:

Bệnh nhân khả năng mắc bệnh liên quan đến tim mạch nằm trong khoảng từ 45 tuổi đổ lên. Xét về mặt thực tế thì điều này là phù hợp với dữ liệu về bệnh tim trong y khoa.



Hình 6 Biểu đồ phân tích theo độ tuổi

2. Khai phá dữ liệu

Xuất thông tin tổng quan của tập dữ liệu:

```
==================================================
THÔNG TIN TỔNG QUAN VỀ DỮ LIỆU
==================================================

 5 DÒNG ĐẦU TIÊN:
   Age Sex ChestPainType  RestingBP  ...  ExerciseAngina  Oldpeak ST_Slope  HeartDisease
0   40   M           ATA        140  ...               N      0.0       Up             0
1   49   F           NAP        160  ...               N      1.0     Flat             1
2   37   M           ATA        130  ...               N      0.0       Up             0
3   48   F           ASY        138  ...               Y      1.5     Flat             1
4   54   M           NAP        150  ...               N      0.0       Up             0

[5 rows x 12 columns]

 5 DÒNG CUỐI CÙNG:
     Age Sex ChestPainType  RestingBP  ...  ExerciseAngina  Oldpeak ST_Slope  HeartDisease
913   45   M           TA        110  ...               N      1.2     Flat             1
914   68   M          ASY        144  ...               N      3.4     Flat             1
915   57   M          ASY        130  ...               Y      1.2     Flat             1
916   57   F          ATA        130  ...               N      0.0     Flat             1
917   38   M          NAP        138  ...               N      0.0       Up             0
```

Xuất kiểu dữ liệu của các đặc trưng:

```
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Age             918 non-null    int64
 1   Sex             918 non-null    object
 2   ChestPainType   918 non-null    object
 3   RestingBP       918 non-null    int64
 4   Cholesterol     918 non-null    int64
 5   FastingBS       918 non-null    int64
 6   RestingECG      918 non-null    object
 7   MaxHR           918 non-null    int64
 8   ExerciseAngina  918 non-null    object
 9   Oldpeak         918 non-null    float64
 10  ST_Slope        918 non-null    object
 11  HeartDisease    918 non-null    int64
dtypes: float64(1), int64(6), object(5)
memory usage: 86.2+ KB
```

Xuất thống kê mô tả các feature có kiểu dữ liệu số:

THỐNG KÊ MÔ TẢ (SỐ):

	count	mean	std	min	25%	50%	75%	max
Age	918.0	53.510893	9.432617	28.0	47.00	54.0	60.0	77.0
RestingBP	918.0	132.396514	18.514154	0.0	120.00	130.0	140.0	200.0
Cholesterol	918.0	198.799564	109.384145	0.0	173.25	223.0	267.0	603.0
FastingBS	918.0	0.233115	0.423046	0.0	0.00	0.0	0.0	1.0
MaxHR	918.0	136.809368	25.460334	60.0	120.00	138.0	156.0	202.0
Oldpeak	918.0	0.887364	1.066570	-2.6	0.00	0.6	1.5	6.2
HeartDisease	918.0	0.553377	0.497414	0.0	0.00	1.0	1.0	1.0

Xuất thống kê mô các feature có kiểu dữ object:

```
 THỐNG KÊ MÔ TẢ (PHÂN LOẠI):

Sex:
Sex
M    725
F    193
Name: count, dtype: int64
Số lượng duy nhất: 2

ChestPainType:
ChestPainType
ASY    496
NAP    203
ATA    173
TA      46
Name: count, dtype: int64
Số lượng duy nhất: 4

RestingECG:
RestingECG
Normal    552
LVH       188
ST        178
Name: count, dtype: int64
Số lượng duy nhất: 3

ExerciseAngina:
ExerciseAngina
N    547
Y    371
Name: count, dtype: int64
Số lượng duy nhất: 2

ST_Slope:
ST_Slope
Flat    460
Up      395
Down     63
Name: count, dtype: int64
Số lượng duy nhất: 3
```

3. Các bước tiền xử lý

- Xử lý các giá trị bất thường ‘0’ trong cột Cholesterol và RestingBP
- Chuẩn hóa dữ liệu thành kiểu float
- Thay thế các giá trị ‘0’ bằng NaN
- Điều giá trị NaN bằng áp dụng thuật toán KNNImputer để đảm bảo dữ liệu được đầy đủ. Thuật toán KNNImputer lấy dữ liệu bằng cách tìm những hàng xóm gần trong tập dữ liệu, và lấy giá trị trung bình để thay thế giá trị NaN

```python
def clean_data(self):
    """
    Tiền xử lý dữ liệu an toàn:
    - Ép tất cả cột numeric sang float để tránh lỗi isnan
    - Thay giá trị 0 bằng NaN cho Cholesterol và RestingBP
    - Điền missing bằng KNNImputer
    """
    df_processed = self.df.copy()

    # Loại bỏ cột AgeGroup nếu có
    if 'AgeGroup' in df_processed.columns:
        df_processed = df_processed.drop('AgeGroup', axis=1)

    # ===== Xác định các cột số =====
    numeric_cols = ['Age', 'RestingBP', 'Cholesterol', 'MaxHR', 'Oldpeak']

    # Ép sang float
    for col in numeric_cols:
        df_processed[col] = pd.to_numeric(df_processed[col], errors='coerce')

    # ===== Thay 0 bằng NaN cho Cholesterol và RestingBP =====
    for col in ['Cholesterol', 'RestingBP']:
        if col in df_processed.columns:
            zero_count = (df_processed[col] == 0).sum()
            if zero_count > 0:
                print(f"{col}: {zero_count} giá trị 0 → sẽ thay bằng NaN")
                df_processed[col] = df_processed[col].replace(0, np.nan)

    # ===== Điền missing bằng KNNImputer =====
    from sklearn.impute import KNNImputer
    imputer = KNNImputer(n_neighbors=3)
    df_processed[numeric_cols] = pd.DataFrame(
        imputer.fit_transform(df_processed[numeric_cols]),
        columns=numeric_cols,
        index=df_processed.index
    )

    # ===== Thông tin kiểm tra =====
    print("Các giá trị missing sau imputer:")
    print(df_processed[numeric_cols].isnull().sum())

    self.df_processed = df_processed
```

- Xử lý
    - Sử dụng Label encoder cho các biến nhị phân: sex, ExerciseAngina

- Sex: Female ->0, Male->1
- ExerciseAngina: No->0, Yes->1
    - Sử dụng OneHotEncoder cho các bieense đa giá trị
- ChestPainType thành 4 cột sau :ChestPainType_ASY ,ChestPainType_ATA, ChestPainType_NAP, ChestPainType_TA

- RestingECG thành 3 cột sau: RestingECG_LVH, RestingECG_Normal, RestingECG_ST
- ST__Slope thành 3 cột sau: ST_Slope_Down, ST_Slope_Flat, ST_Slope_Up

```
def encode_features(self):
    df_processed = self.df_processed
    target_col1 = ['Sex', 'ExerciseAngina']
    target_col2 = ['ChestPainType', 'RestingECG', 'ST_Slope']

    # ---- LabelEncoder cho target_col1 ----
    for col in target_col1:
        le = LabelEncoder()
        df_processed[col] = le.fit_transform(df_processed[col])
        self.label_encoders[col] = le

    # ---- OneHotEncoder cho target_col2 ----
    self.onehot_encoder = OneHotEncoder(drop=None, handle_unknown='ignore')
    onehot = self.onehot_encoder.fit_transform(df_processed[target_col2])
    onehot_df = pd.DataFrame(
        onehot.toarray(),
        columns=self.onehot_encoder.get_feature_names_out(target_col2),
        index=df_processed.index
    )

    df_processed = pd.concat([df_processed.drop(columns=target_col2), onehot_df], axis=1)
    self.df_processed = df_processed
```

Chuẩn hoá dữ liệu cho mô hình:

```
#Chuẩn bị dữ liệu cho model
def prepare_for_model(self):
    df_processed = self.df_processed
    if 'HeartDisease' in df_processed.columns:
        X = df_processed.drop('HeartDisease', axis=1)
        y = df_processed['HeartDisease']
        self.X = X
        self.y = y

        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.2, random_state=42, stratify=y
        )

        self.X_train, self.X_test = X_train, X_test
        self.y_train, self.y_test = y_train, y_test

        self.scaler = StandardScaler()
        X_train_scaled = pd.DataFrame(
            self.scaler.fit_transform(X_train),
            columns=X_train.columns,
            index=X_train.index
        )
        X_test_scaled = pd.DataFrame(
            self.scaler.transform(X_test),
            columns=X_test.columns,
            index=X_test.index
        )

        return X_train_scaled, X_test_scaled, y_train, y_test
    else:
        return None
```

Chuẩn hóa dữ liệu mới nhập cho mô hình để đảm bảo dữ liệu mới nhập phải trải qua tiền xử lý trước khi đưa vào huấn luyện mô hình:

```python
#Chuẩn hóa dữ liệu mới trc khi đưa vào model
def transform_new_data(self, df_new: pd.DataFrame):
    if not self.label_encoders or not self.onehot_encoder or not self.scaler:
        raise ValueError("Preprocessor chưa được huấn luyện. Chạy encode_features() và prepare_for_model() trước.")

    df = df_new.copy()

    # Map số sang nhãn string cho các cột phân loại
    chest_map = {0:"TA", 1:"ATA", 2:"NAP", 3:"ASY"}
    rest_map = {0:"Normal", 1:"ST", 2:"LVH"}
    slope_map = {0:"Up", 1:"Flat", 2:"Down"}

    if "ChestPainType" in df.columns:
        df["ChestPainType"] = df["ChestPainType"].map(chest_map)
    if "RestingECG" in df.columns:
        df["RestingECG"] = df["RestingECG"].map(rest_map)
    if "ST_Slope" in df.columns:
        df["ST_Slope"] = df["ST_Slope"].map(slope_map)

    # ---- LabelEncoder cho cột nhị phân ----
    for col in ['Sex', 'ExerciseAngina']:
        le = self.label_encoders[col]
        df[col] = df[col].apply(lambda x: le.transform([x])[0] if x in le.classes_ else -1)

    # ---- OneHotEncoder ----
    onehot = self.onehot_encoder.transform(df[['ChestPainType','RestingECG','ST_Slope']])
    onehot_df = pd.DataFrame(onehot.toarray(),
                             columns=self.onehot_encoder.get_feature_names_out(['ChestPainType','RestingECG','ST_Slope']),
                             index=df.index)
    df = pd.concat([df.drop(columns=['ChestPainType','RestingECG','ST_Slope']), onehot_df], axis=1)

    # ---- Ép kiểu numeric cho các cột số ----
    numeric_cols = ['Age', 'RestingBP', 'Cholesterol', 'MaxHR', 'Oldpeak']
    for col in numeric_cols:
        if col in df.columns:
            df[col] = pd.to_numeric(df[col], errors='coerce')

    # ---- Chuẩn hóa ----
    return self.scaler.transform(df)
```

4. Mô hình học máy

a. Decision Tree

Decision Tree (Cây quyết định) là mô hình học máy phân loại dựa trên việc chia dữ liệu thành các nhánh theo điều kiện và đưa ra kết quả dự đoán (có bệnh/ không có bệnh).

❖ Ưu điểm:

- Dễ hiểu, xử lý tốt cả biến số và biến phân loại.
- Thời gian huấn luyện nhanh, dễ triển khai.

❖ Nhược điểm:

- Dễ bị overfitting, khi tăng max_depth, cây càng sâu gây học nhiễu làm dự đoán kém.
- Độ chính xác của mô hình này cũng thấp nhất trong các mô hình được huấn luyện trong bài.
- Chỉ cần thay đổi dữ liệu nhỏ cũng làm cấu trúc cây thay đổi.

❖ Phần chuẩn bị và khởi tạo

```
## Phần chuẩn bị & khởi tạo pipeline
def __init__(self, X_train_scaled, X_test_scaled, y_train, y_test, feature_names=None):
    self.X_train_scaled = X_train_scaled #Lưu tập dữ liệu huấn luyện (đã được chuẩn hoá)
    self.X_test_scaled = X_test_scaled #Lưu tập dữ liệu kiểm tra
    self.y_train = y_train #Lưu nhãn của tập huấn luyện (0, 1)
    self.y_test = y_test #Lưu nhãn dùng để đánh giá
    self.feature_names = feature_names #Lưu tên các cột đặc trưng
    self.model = None #Lưu mô hình cơ bản
    self.best_model = None #Lưu mô hình tốt nhất sau tuning
```

- Hàm khởi tạo "__init__" được sữ dụng để chuẩn bị dữ liệu và thiết lập pipeline cho mô hình Decision Tree. Dữ liệu huấn luyện và kiểm tra đã được chuẩn hóa, được lưu trữ để phục vụ quá trình huấn luyện và đánh giá.

❖ Phần huấn luyện mô hình

```
## Phần huấn luyện
def train_decision_tree(self
                        , max_depth=3 #Giới hạn độ sâu của cây tránh overfitting
                        , random_state=42 #Cố định kết quả sinh ngẫu nhiên
                        ):
    """Huấn luyện mô hình Decision Tree cơ bản"""
    self.model = DecisionTreeClassifier(max_depth=max_depth #Kiểm soát độ phức tạp của cây
                                        , random_state=random_state #Đảm bảo kết quả ổn định
                                        )
    self.model.fit(self.X_train_scaled, self.y_train)
    return self.model
```

- Huấn luyện mô hình Decision Tree nhằm tránh hiện tượng overfitting và sử dụng tham số random_state để đảm bảo tính tái lập của kết quả.

❖ Phần đánh giá mô hình

```
## Phần đánh giá mô hình
def evaluate_model(self, model=None):
    """Đánh giá mô hình với các metrics"""
    if model is None:
        model = self.model
    y_pred = model.predict(self.X_test_scaled)
    y_proba = model.predict_proba(self.X_test_scaled)[:, 1] if hasattr(model, "predict_proba") else None

    metrics = {
        "accuracy": accuracy_score(self.y_test, y_pred), #Tỷ lệ dự đoán đúng trên tổng số mẫu
        "precision": precision_score(self.y_test, y_pred), #trong số dự đoán có bệnh, bao nhiêu ca đúng
        "recall": recall_score(self.y_test, y_pred), #mô hình phát hiện bao nhiêu người trong số người 1
        "f1": f1_score(self.y_test, y_pred), #đánh giá cân bằng mô hình (precision và recall)
        "roc_auc": roc_auc_score(self.y_test, y_proba) if y_proba is not None else None, # khả năng phân biệt 0 và 1
        "y_pred": y_pred,
        "y_proba": y_proba
    }
    return metrics
```

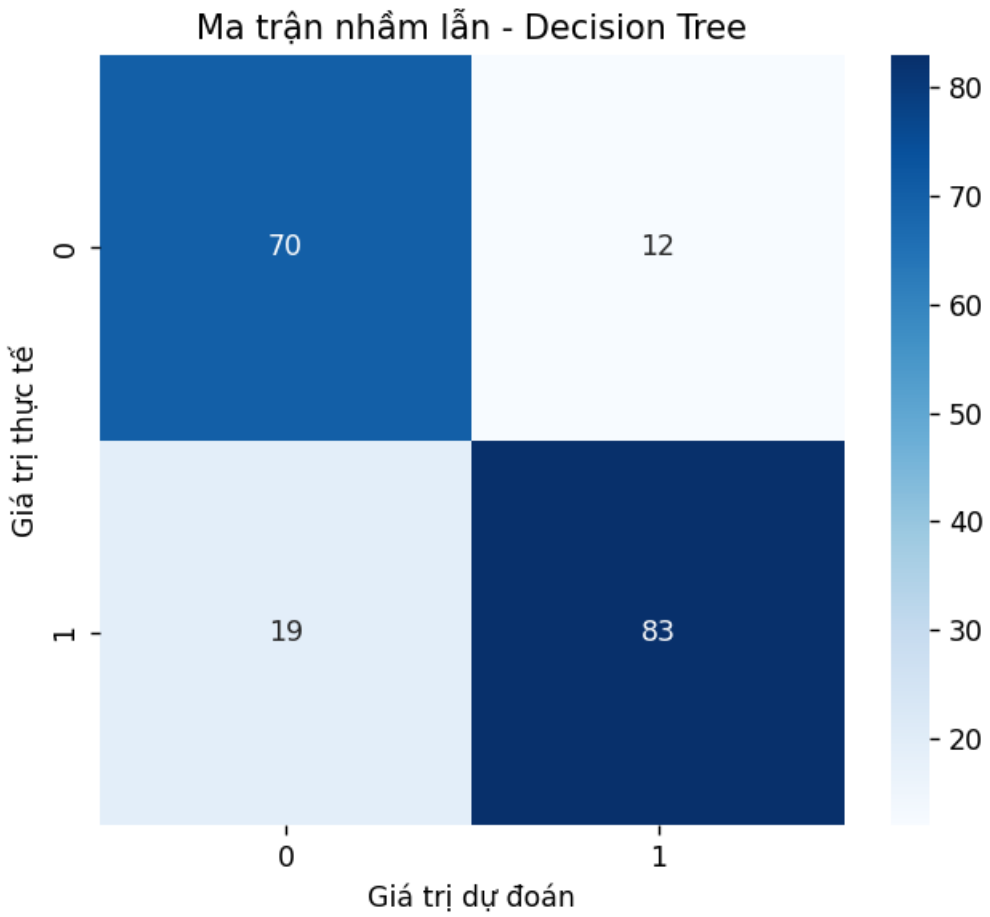
- Đánh giá hiệu năng của mô hình trên dữ liệu bằng nhiều chỉ số khác nhau, từ đó xác định mức độ chính xác và khả năng phát hiện bệnh tim của mô hình.
- Kết quả:

```
Accuracy: 0.8315
Precision: 0.8737
Recall: 0.8137
F1-Score: 0.8426
ROC-AUC: 0.8987
```

❖ Phần Confusion Matrix

```
## Phần Confusion Matrix
def plot_confusion_matrix(self, y_pred, title="Ma trận nhầm lẫn - Decision Tree"):
    cm = confusion_matrix(self.y_test, y_pred) # tạo ma trận
    plt.figure(figsize=(6, 5)) #Tạo khung hình cho biểu đồ
    sns.heatmap(cm, annot=True #Hiển thị giá trị trong từng ô
                , fmt='d' #Hiển thị số nguyên
                , cmap='Blues'
                )
    plt.title(title)
    plt.xlabel('Giá trị dự đoán')
    plt.ylabel('Giá trị thực tế')
    plt.show()
```

- Dùng để phân tích chi tiết kết quả dự đoán của mô hình Decision Tree, cho biết mô hình dự đoán đúng và sai như thế nào đối với từng lớp.
- Kết quả:

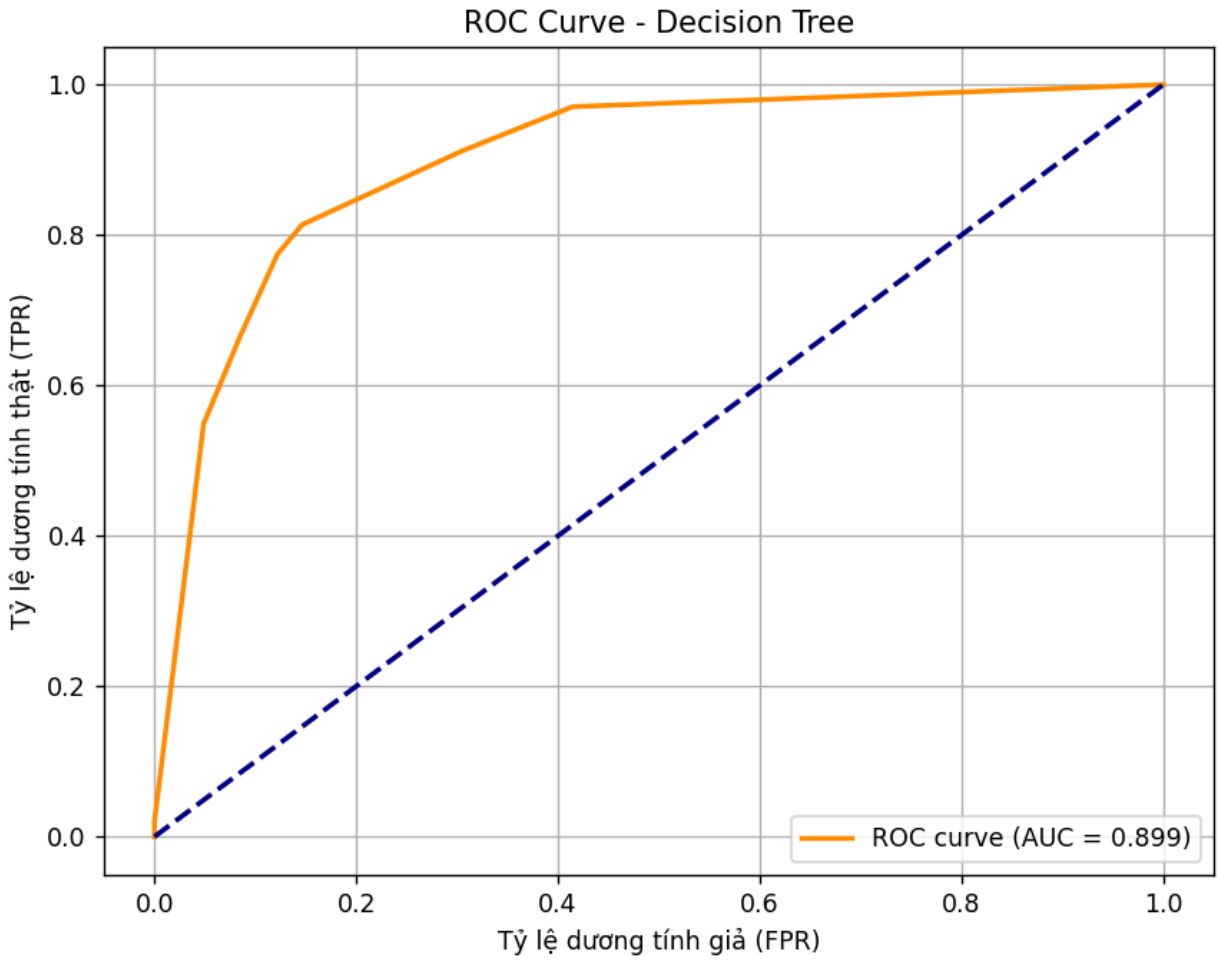


- Kết quả trên cho thấy mô hình Decision Tree có khả năng phát hiện khá tốt với số lượng giá trị thực tế cao hơn, tuy nhiên vẫn còn vài trường hợp bị dự đoán nhầm, cần cân nhắc.

❖ Phần ROC_Curve & AUC

```
## Phần ROC_Curve (Đánh giá khả năng phân biệt)
def plot_roc_curve(self, y_proba, model_name="Decision Tree"):

    ## Tính False Positive Rate (FPR) và True Positive Rate (TPR)
    fpr, tpr, _ = roc_curve(self.y_test, y_proba)

    # Tính diện tích dưới đường cong ROC (AUC)
    roc_auc = roc_auc_score(self.y_test, y_proba)
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.3f})') #Vẽ đường cong ROC
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--') #vẽ đường chéo
    plt.xlabel('Tỷ lệ dương tính giả (FPR)')
    plt.ylabel('Tỷ lệ dương tính thật (TPR)')
    plt.title(f'ROC Curve - {model_name}')
    plt.legend(loc="lower right")
    plt.grid(True)
    plt.show()
```

- ROC Curve được sử dụng để đánh giá khả năng phân biệt giữa hai lớp (có bệnh và không có bệnh tim) của mô hình Decision Tree ở nhiều ngưỡng phân loại khác nhau, thay vì chỉ dựa vào một ngưỡng cố định như Accuracy.
- Kết quả:

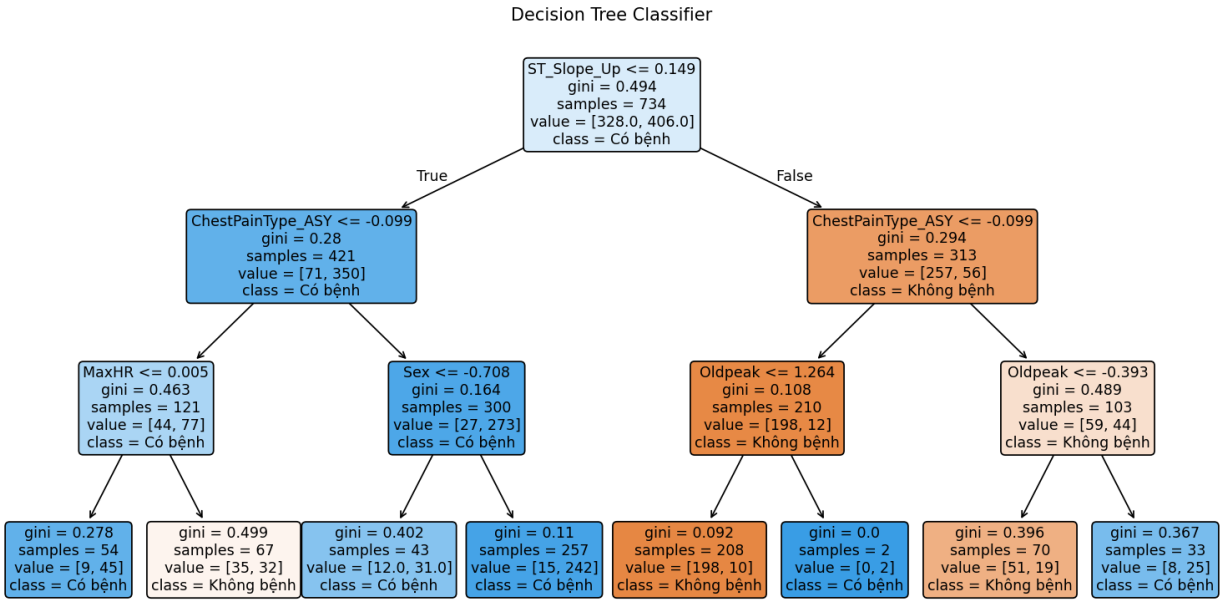


- Đường cong ROC cho thấy mô hình Decision Tree có khả năng phân biệt tốt giữa hai lớp có và không có bệnh tim. Giá trị AUC đạt 0.899, chứng tỏ mô hình hoạt động hiệu quả và vượt trội so với dự đoán ngẫu nhiên

❖ Phần vẽ cây quyết định

```python
## Phần vẽ cây
def plot_decision_tree(self, model=None):
    #Nếu không truyền mô hình vào hàm thì dùng mô hình đã train trước đó
    if model is None:
        model = self.model
    plt.figure(figsize=(22, 12))
    plot_tree(model,
              feature_names=self.feature_names if self.feature_names is not None else None, #giúp hiện thị ra tên thuộc tính thay vì X[]
              class_names=['Không bệnh', 'Có bệnh'],
              filled=True, rounded=True, fontsize=10)
    plt.title("Decision Tree Classifier")
    plt.show()
```

- Trực quan hoá cấu trúc của mô hình Decision Tree giúp hiểu rõ mô hình sử dụng những đặc trưng nào, cách mô hình đưa ra quyết định phân loại.



- Cây quyết định được trực quan hóa nhằm minh họa cách mô hình Decision Tree đưa ra quyết định phân loại. Các đặc trưng quan trọng như ST_Slope, ChestPainType, Sex, Oldpeak và MaxHR được sử dụng để phân chia dữ liệu.

❖ Phần tuning cơ bản

```python
## Phần tuning
def tune_max_depth(self, max_depth_values=range(3, 20)):
    best_score = 0.0 #Lưu độ chính xác cao nhất
    best_depth = None #Lưu giá trị max_depth tương ứng
    for depth in max_depth_values:
        # Khởi tạo mô hình Decision Tree với độ sâu hiện tại
        clf = DecisionTreeClassifier(max_depth=depth, random_state=42)

        # Huấn luyện mô hình trên tập train
        clf.fit(self.X_train_scaled, self.y_train)

        # Đánh giá mô hình trên tập test
        score = clf.score(self.X_test_scaled, self.y_test)
        print(f"Max Depth = {depth}: Test Accuracy = {score:.4f}")

        #Nếu Accuracy cao hơn giá trị tốt nhất trước đó thì cập nhật
        if score > best_score:
            best_score = score
            best_depth = depth
    print(f"\nBest Max_depth = {best_depth} : Best Accuracy = {best_score:.4f}")
    return best_depth, best_score
```

- Tìm giá trị Max_Depth phù hợp cho mô hình Decision Tree, giúp cân bằng độ chính xác và hạn chế bị overfitting.
- Kết quả:

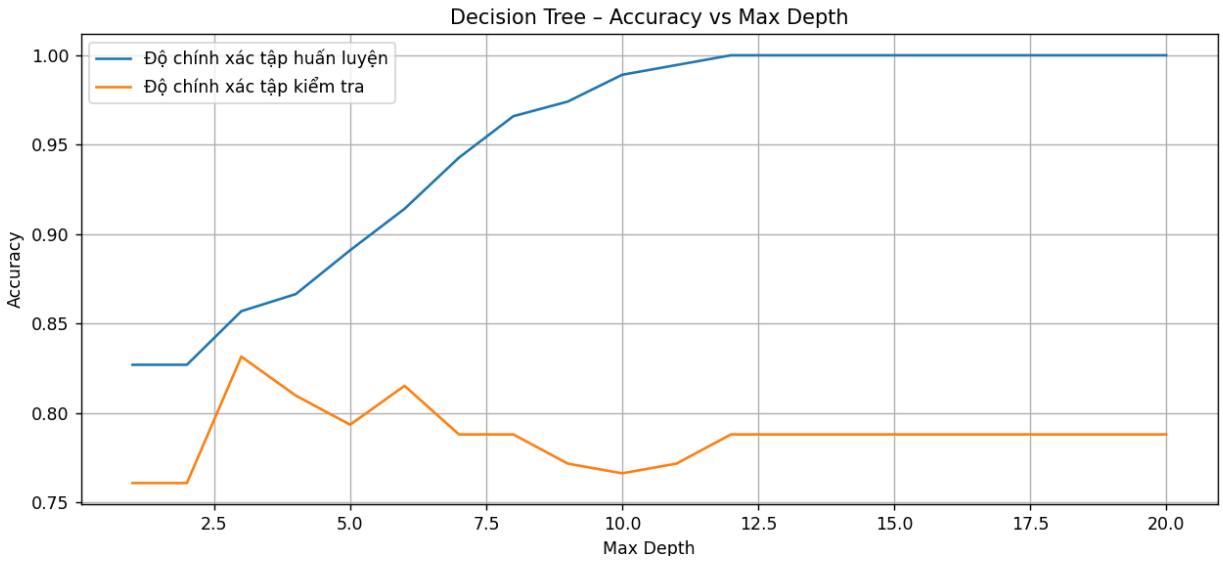
```
Max Depth = 3: Test Accuracy = 0.8315
Max Depth = 4: Test Accuracy = 0.8098
Max Depth = 5: Test Accuracy = 0.7935
Max Depth = 6: Test Accuracy = 0.8152
Max Depth = 7: Test Accuracy = 0.7880
Max Depth = 8: Test Accuracy = 0.7880
Max Depth = 9: Test Accuracy = 0.7717
Max Depth = 10: Test Accuracy = 0.7663
Max Depth = 11: Test Accuracy = 0.7717
Max Depth = 12: Test Accuracy = 0.7880
Max Depth = 13: Test Accuracy = 0.7880
Max Depth = 14: Test Accuracy = 0.7880
Max Depth = 15: Test Accuracy = 0.7880
Max Depth = 16: Test Accuracy = 0.7880
Max Depth = 17: Test Accuracy = 0.7880
Max Depth = 18: Test Accuracy = 0.7880
Max Depth = 19: Test Accuracy = 0.7880

Best Max_depth = 3 : Best Accuracy = 0.8315
```

- Kết quả tuning cho thấy mô hình Decision Tree đạt độ chính xác cao nhất khi max_depth = 3. Khi độ sau của cây tăng thì độ chính xác có xu hướng giảm, cho thấy hiện tượng overfitting.

❖ Phần biểu đồ Accuracy và Max_Depth

```
## Phần biểu đồ Accuracy vs Max Deth
def plot_accuracy_vs_depth(self, depths=range(1, 21)):
    train_acc, val_acc = [], []
    for depth in depths:
        dt = DecisionTreeClassifier(max_depth=depth, random_state=42)
        dt.fit(self.X_train_scaled, self.y_train)

        #Accuracy trên tập huấn luyện
        train_acc.append(dt.score(self.X_train_scaled, self.y_train))

        #Accuracy trên tập kiểm tra
        val_acc.append(dt.score(self.X_test_scaled, self.y_test))
    plt.figure(figsize=(12, 5))
    plt.plot(depths, train_acc, label='Độ chính xác tập huấn luyện')
    plt.plot(depths, val_acc, label='Độ chính xác tập kiểm tra')
    plt.xlabel('Max Depth')
    plt.ylabel('Accuracy')
    plt.title('Decision Tree - Accuracy vs Max Depth')
    plt.legend()
    plt.grid(True)
    plt.show()
```

- Được sử dụng để phân tích ảnh hưởng của tham số max_depth đến hiệu năng của mô hình Decision Tree, đồng thời phát hiện hiện tượng overfitting thông qua sự chênh lệch giữa độ chính xác trên tập huấn luyện và tập kiểm tra.
- Kết quả:



- Biểu đồ Accuracy và Max Depth cho thấy khi độ sâu của cây tăng lên, độ chính xác trên tập huấn luyện tăng, trong khi độ chính xác trên tập kiểm tra không cải thiện và có xu hướng giảm. Điều này cho thấy mô hình Decision Tree dễ bị overfitting khi cây quá sâu. Giá trị max_depth = 3 được lựa chọn vì đạt độ chính xác cao và khả năng tổng quát hóa tốt.

❖ Phần tuning nâng cao

```
## Phần tunning nâng cao
def grid_search_decision_tree(self):
    dt = DecisionTreeClassifier(random_state=42)
    param_grid = {
        'max_depth': [2, 3, 4, 5, 6, 8, 10, None], #Độ sâu tối đa của cây
        'min_samples_split': [2, 5, 10, 20], # Số mẫu tối thiểu để tách 1 node
        'min_samples_leaf': [1, 2, 5, 10], #Số mẫu tối thiểu tại node lá
        'criterion': ['gini', 'entropy']
    }
    grid_search = GridSearchCV(dt, param_grid, cv=5, scoring='accuracy', n_jobs=-1, verbose=0)
    grid_search.fit(self.X_train_scaled, self.y_train)
    print("Thông số tốt nhất:", grid_search.best_params_)
    print("Độ chính xác CV tốt nhất:", grid_search.best_score_)
    self.best_model = grid_search.best_estimator_ #Lưu lại mô hình tốt nhất
    return self.best_model
```

- Sau khi tuning đơn giản với tham số max_depth, mô hình Decision Tree vẫn có nguy cơ overfitting. Vì vậy, Grid Search được sử dụng để tìm tổ hợp tham số tối ưu nhất bằng cách đánh giá nhiều tham số .
- Kết quả:

```
Thông số tốt nhất: {'criterion': 'gini', 'max_depth': 3, 'min_samples_leaf': 10, 'min_samples_split': 2}
Độ chính xác CV tốt nhất: 0.8433044450656976
```

- Kết quả cho thấy mô hình tối ưu đạt độ chính xác trung bình 84.33% với các tham số max_depth = 3, min_samples_leaf = 10 và criterion = gini. Điều này cho thấy việc giới hạn độ sâu cây và tăng số mẫu tại node lá giúp mô hình giảm overfitting và cải thiện khả năng tổng quát hóa.

b. Gradient Boosting

❖ Giới thiệu về Gradient Boosting:

-XGBoost (Extreme Gradient Boosting) là một giải thuật được base trên gradient boosting, tuy nhiên kèm theo đó là những cải tiến to lớn về mặt tối ưu thuật toán, về sự kết hợp hoàn hảo giữa sức mạnh phần mềm và phần cứng, giúp đạt được những kết quả vượt trội cả về thời gian training cũng như bộ nhớ sử dụng.

-Mã nguồn mở với ~350 contributors và ~3,600 commits trên Gihub, XGBoost cho thấy những khả năng ứng dụng đáng kinh ngạc của mình như :

-XGBoost có thể được sử dụng để giải quyết được tất cả các vấn đề từ hồi quy (regression), phân loại (classification), ranking và giải quyết các vấn đề do người dùng tự định nghĩa.

-XGBoost hỗ trợ trên Windows, Linux và OS X.

-Hỗ trợ tất cả các ngôn ngữ lập trình chính bao gồm C ++, Python, R, Java, Scala và Julia.

-Hỗ trợ các cụm AWS, Azure và Yarn và hoạt động tốt với Flink, Spark và các hệ sinh thái khác.

-Dự đoán xác suất trên tập dữ liệu thử nghiệm bằng predict_proba()phương pháp của mô hình đã được huấn luyện.

- Nguyên lý hoạt động
  - Sequential Learning: Các mô hình được huấn luyện tuần tự thay vì song song
  - Gradient Descent: Sử dụng phương pháp giảm độ dốc để tối ưu hóa hàm mất mát

- Weighted Combination: Kết hợp các mô hình yếu với trọng số phù hợp

❖ Ưn điểm:

1. Hiệu suất cao

Accuracy: 88.59% (cao nhất trong bài toán của bạn)

AUC-ROC: 0.943 (xuất sắc)

F1-score cân bằng: 0.90

2. Linh hoạt

Xử lý cả dữ liệu số và phân loại

Không yêu cầu chuẩn hóa mạnh

Tự động phát hiện tương tác phi tuyến

3. Dễ hiểu

Có feature importance rõ ràng

Biết được yếu tố quan trọng nhất:

ST_Slope_Flat (21.5%)

Oldpeak (18.2%)

MaxHR (14.8%)

4. Giảm overfitting

Dùng subsampling (0.8 trong bài)

Có regularization tự nhiên

Cân bằng tốt bias-variance

❖ Nhược điểm:

1. Tốn tài nguyên

Training time: 0.32s (chậm hơn Random Forest)

Memory usage cao (lưu nhiều cây)

Khó parallelize training

2. Khó tuning

Quá nhiều hyperparameters (8+)

Cần GridSearchCV/RandomizedSearchCV

Nhạy cảm với hyperparameters

3. Giải thích hạn chế

Khó giải thích từng dự đoán cụ thể

Cần SHAP/LIME để giải thích chi tiết

Phức tạp hơn so với Decision Tree

4. Khác

Nhạy cảm với nhiễu (noise)

Không hỗ trợ online learning

Dễ overfit nếu không điều chỉnh tốt

❖ Mục đích

- Lớp này thực hiện việc xây dựng, huấn luyện, tối ưu và đánh giá các mô hình Gradient Boosting cho bài toán phân loại bệnh tim. Gradient Boosting là một kỹ thuật ensemble học mạnh mẽ kết hợp nhiều cây quyết định yếu thành một mô hình mạnh.

❖ Kiến trúc tổng quan

GradientBoostingModel:

- Khởi tạo (__init__)
- Quản lý dữ liệu

- Huấn luyện mô hình
- Tinh chỉnh siêu tham số
- Ensemble methods
- Đánh giá và visualization
- Lưu/tải model

❖ Khởi tạo và thiết lập

- def __init__(self, X_train, X_test, y_train, y_test, feature_names=None, save_dir='models'):
- Input: Dữ liệu train/test đã chuẩn hóa, nhãn và tên đặc trưng
- Output: Khởi tạo đối tượng với các thuộc tính cần thiết
- Tự động tạo thư mục lưu trữ: models/ và models/datasets/

❖ Quản lý dữ liệu

- save_datasets()
  - Mục đích: Lưu dataset train/test thành file CSV để tái sử dụng
  - Đầu ra: 3 file:
- gb_train_dataset.csv: Dữ liệu huấn luyện.
- gb_ test_dataset.csv: Dữ liệu kiểm tragb_dataset_info.pkl: Thông tin metadata.

- load_datasets()
  - Mục đích: Tải lại datasets đã lưu để tiếp tục làm việc.
  - Ứng dụng: Tiết kiệm thời gian khi cần thử nghiệm nhiều lần.

❖ Các mô hình Gradient Boosting được triển khai

- Mô hình cơ bản (model_gb_basic())

```
GradientBoostingClassifier(
        n_estimators=100,      # 100 cây
        learning_rate=0.1,     # Tốc độ học
        max_depth=3,           # Độ sâu tối đa
        random_state=42        # Đảm bảo tái lập kết quả
```

)

- Mô hình nâng cao (model_gb_advanced())

```
GradientBoostingClassifier(
        n_estimators=200,      # Tăng số cây
        learning_rate=0.05,    # Giảm tốc độ học
        max_depth=5,           # Tăng độ sâu
        max_features='sqrt'   # Giới hạn đặc trưng mỗi cây
)
```

❖ Tinh chỉnh siêu tham số

- RandomizedSearchCV (finetuning_randomized())
  - Phương pháp: Tìm kiếm ngẫu nhiên trong không gian tham số
  - Ưu điểm: Nhanh, hiệu quả với không gian tham số lớn
  - Tham số tìm kiếm:

```
param_dist = {
        'n_estimators': [50, 100, 150, 200],
        'learning_rate': [0.01, 0.05, 0.1, 0.2],
        'max_depth': [3, 4, 5, 6],
        'subsample': [0.7, 0.8, 0.9, 1.0]
}
```

- GridSearchCV (finetuning_grid())
  - Phương pháp: Tìm kiếm toàn bộ lưới tham số
  - Ưu điểm: Tìm được tham số tối ưu chính xác
  - Nhược điểm: Chậm với không gian tham số lớn

❖ Kĩ thuật Ensemble

- Voting Ensemble (ensemble_gb())

- VotingClassifier(

  estimators=[

  ('gb1', GradientBoostingClassifier(...)),

  ('gb2', GradientBoostingClassifier(...)),

  ('rf', RandomForestClassifier(...)),

  ('lr', LogisticRegression(...))

  ],

  voting='soft' # Voting dựa trên xác suất

)

  - Nguyên lý: Kết hợp nhiều mô hình để tăng độ chính xác.
  - Chạy nhiều lần: Lấy kết quả tốt nhất sau n_runs lần chạy.

- Stacking (stacking_gb())
  - Base models: Gradient Boosting, Random Forest, Logistic Regression.
  - Meta-learner: Logistic Regression học từ dự đoán của base models.
  - Ưu điểm: Tận dụng điểm mạnh của từng mô hình.

❖ Đánh giá mô hình

- Các metrics sử dụng
  - metrics = {

    "accuracy": accuracy_score(y_true, y_pred),

    "precision": precision_score(y_true, y_pred),

    "recall": recall_score(y_true, y_pred),

    "f1": f1_score(y_true, y_pred),

    "roc_auc": roc_auc_score(y_true, y_proba)

    }

- Visualizations
  - Confusion Matrix + ROC Curve

- Confusion Matrix: Phân tích lỗi phân loại
- ROC Curve: Đánh giá khả năng phân biệt
- Feature Importance
- Xác định đặc trưng quan trọng nhất
- Giúp hiểu mô hình và cải thiện dữ liệu
- Learning Curve
- Phân tích bias-variance tradeoff
- Xác định overfitting/underfitting

❖ Quản lý mô hình

- Lưu mô hình (save_model())
  - joblib. dump(self.best_model, model_path)
  - Lưu model tốt nhất
  - Lưu metrics và feature importance
  - Lưu metadata và kết quả so sánh
- Tải mô hình (load_model())
  - Khôi phục mô hình đã huấn luyện
  - Tiếp tục sử dụng hoặc dự đoán
- Flowchart xử lý
  - graph TD
  - A[Khởi tạo dữ liệu] --> B[Lưu datasets]
  - B --> C[Huấn luyện mô hình cơ bản]
  - C --> D[Huấn luyện mô hình nâng cao]
  - D --> E[Tinh chỉnh RandomizedSearch]
  - E --> F[Tinh chỉnh GridSearch]
  - F --> G{use_ensemble?}
  - G -->|Yes| H[Ensemble Voting]
  - G -->|No| I
  - H --> I{use_stacking?}
  - I-->|Yes| J[Stacking Ensemble]
  - I -->|No| K

- J--> K[So sánh kết quả]
- K --> L[Chọn model tốt nhất]
- L--> M[Visualization]
- M--> N[Lưu kết quả]

- Pipeline chính (run_models())
  - Chuẩn bị dữ liệu
  - Huấn luyện 4-6 mô hình khác nhau
  - So sánh và chọn model tốt nhất
  - Visualize kết quả
  - Lưu trữ toàn bộ thông tin

❖ Ưu điểm của thiết kế

- Tính linh hoạt
- Tuỳ chọn ensemble: Có thể bật/tắt ensemble và stacking
- Lưu datasets: Tái sử dụng dữ liệu đã xử lý
- Nhiều phương pháp tinh chỉnh: Randomized + Grid Search
- Tính toàn diện
- Đa dạng mô hình: Từ cơ bản đến nâng cao
- Đánh giá đa chiều: 5 metrics + visualization
- Feature analysis: Hiểu rõ đặc trưng quan trọng
- Tính tái sử dụng
- Modular design: Dễ dàng mở rộng
- Auto-save: Tự động lưu tất cả kết quả
- Easy loading: Dễ dàng tải lại để sử dụng
- Khả năng mở rộng
- Thêm mô hình mới

```
def model_gb_custom(self, custom_params):
    model = GradientBoostingClassifier(**custom_params)
```

- …. training và evaluation

❖ Ứng dụng thực tế

- Trong bài toán bệnh tim
- Dự đoán rủi ro: Phân loại bệnh tim dựa trên đặc trưng lâm sàng
- Feature importance: Xác định yếu tố nguy cơ quan trọng
- Độ tin cậy: Xác suất dự đoán giúp bác sĩ ra quyết định
- Các ứng dụng khác
- Tài chính: Phát hiện gian lận
- Marketing : Dự đoán khách hàng rời bỏ
- Y tế: Chẩn đoán bệnh từ dữ liệu lâm sàng

❖ Kết luận

- Lớp GradientBoostingModel là một giải pháp toàn diện cho việc xây dựng và triển khai mô hình Gradient Boosting với các tính năng:
- Đầy đủ: Từ data management đến model deployment
- Linh hoạt: Nhiều tùy chọn và cấu hình
- Chuyên sâu: Đánh giá đa chiều và visualization
- Thực tế: Dễ dàng tích hợp vào hệ thống dự đoán

c. MLPClassifier

MLP Classifier (Multilayer Perceptron Classifier – MLP) là một thuật toán học máy có giám sát (Machine Learning) thuộc lớp Mạng nơ-ron nhân tạo, là tập hợp của các perceptron chia làm nhiều nhóm, mỗi nhóm tương ứng với một layer. Thuật toán về cơ bản được đào tạo trên dữ liệu để học một hàm phi tuyến tính nhằm mục đích phân loại hay hồi quy, với một tập hợp các tính năng và một biến mục tiêu (ví dụ: nhãn).

- Ưu điểm:
  - Mô hình hóa tốt các mối quan hệ phi tuyến tính phức tạp giữa các đặc trưng nhờ các tầng ẩn và hàm kích hoạt
  - Tính linh hoạt cao, và xử lý được các đặc trưng đa chiều
  - Tổng quát hóa tốt khi tinh chỉnh thông số đúng
  - Hiệu quả với bài toán phân loại nhị phân: chẩn đoán y khoa, phát hiện gian lận,...
- Nhược điểm:

- Nhạy cảm với siêu tham số: việc tinh chỉnh thông số để model tối ưu là vô cùng quan trọng
- Yêu cầu nhiều tài nguyên và thời gian huấn luyện
- Độ phức tạp cao trong quá trình cài đặt và huấn luyện, và cần phải hiểu rõ về mạng neuron
- Dễ bị overfitting đặc biệt với các dữ liệu nhỏ vì nó học dữ liệu nhiều lần

Thông số của model:

Tên thuộc tính	Mô tả	Giá trị
hidden_layer_sizes	số neuron ở mỗi tầng ẩn. Có thể có nhiều tầng ẩn. Thông thường sẽ là 1-3 tầng	Int Vd: (100,) : 1 tầng ẩn với 100 neuron
activation	hàm kích hoạt	Relu: - công thức: f(x) = max(0,x) - Hàm relu chỉ trả về các giá trị đầu vào dương, nếu không có thì sẽ trả về không - Relu giúp quá trình huấn luyện mạng neuron nhanh hơn vì lấy các giá trị dương nên gradient sẽ không bị suy giảm theo thời gian Tanh: - Công thức: $f(x) = \frac{2}{1+e^{-2x}} - 1$

		- Tanh có các giá trị đầu ra trong khoảng -1 đến 1, từ đây model có thể học được cả giá trị âm và dương, đồng thời đối xứng qua 0. Logistic: - Công thức: $f(x) = \frac{1}{1+e^{-x}}$ - Logistic có các giá trị đầu ra trong khoảng từ 0 đến 1. Vì hàm giúp chuyển đổi đầu vào thành xác suất nên mô hình có thể đưa ra quyết định mỗi lớp theo xác suất. Hàm Logistic phù hợp với các bài toán phân loại nhị phân Identity - Công thức: $f(x) = x$ - Indentity là hàm kích hoạt tuyến tính, và từ công thức có thể thấy được đầu ra chính là đầu vào. - Hàm phù hợp với các bài toán hồi quy vì giá trị đầu ra có thể thay đổi trực tiếp với đầu vào.
solver	Thuật toán tối ưu	**lbfgs:** Phương pháp tối ưu bậc hai, nhanh và hiệu quả đối với các mô hình có ít tham số hơn và ít dữ liệu hơn.

		sgd (Stochastic Gradient Descent): Thuật toán gradient descent ngẫu nhiên, có thể điều chỉnh tốc độ học và thường được sử dụng cho dữ liệu lớn. **adam:** Một phương pháp tối ưu hiện đại kết hợp giữa momentum và adaptive learning rate, rất phổ biến vì hiệu suất tốt và dễ sử dụng.
max_tier	số vòng lặp tối đa trong quá trình huấn luyện	Int - Giá trị mặc định: 200
learning_rate	Tốc độ học	constant: tốc độ học cố định invscaling: giảm dần theo 1/t adaptive: giảm khi không cải thiện
alpha	hệ số L2 penalty để tránh overfitting	float: 0.001, 0.0001,…. - Giá trị mặc định: 0.0001 - Giá trị thuộc đoạn [0.0001, 1]
batch_size		auto: min(200, n_samples)
shuffle	xáo trộn dữ liệu mỗi epoch	true/false
random_state	seed để tái thiết lập kết quả	int

early_stopping	dừng sớm khi không cải thiện trên tập validation	True/False
validation_fraction	tỉ lệ dữ liệu validation khi early_stopping = true	float
n_tier_no_change	số epoch không cải thiện trước khi dừng	int
momentum	trọng số gradient chỉ dành cho solver =’sgd’	- Giá trị thuộc: 0.8-0.95

Xây dựng model với MLPClassifier với hàm kích hoạt là relu và thuật toán tối ưu là adam. Hai thuật toán trên được chọn vì ReLU chạy ổn định với dữ liệu dương mà hầu hết các giá trị y tế đều là dương. Đồng thời ReLU giúp gradient không bị nhỏ dần nên mô hình sẽ học được hiệu quả hơn. Thuật toán tối ưu Adam là sự kết hợp giữa Momentum (tích lũy gradient quá khứ) và RMSPop (điều chỉnh tốc độ học theo từng tham số) phù hợp dữ liệu y tế phức tạp (không đồng đều dữ liệu), từ đây mô hình chạy ổn định hơn.

Xây dựng MLPClassifier cơ bản:

```python
# ================== MLP cơ bản ==================
def model_MLP(self):
    model = MLPClassifier(
        activation='relu', solver='adam', #chọn hàm kích hoạt: relu, thuật toán tối ưu:adam
        max_iter=1000, #Học lặp 1000
        learning_rate_init=0.001, #Tốc độ học ban đầu của thuật toán
        hidden_layer_sizes=(100,), #Học 1 layer với 100 neuron
        random_state=42, #seed
        alpha=0.1 #hệ số regularization
    )
    model.fit(self.X_train_scaled, self.y_train) #fit tập dữ liệu vào model
    y_pred = model.predict(self.X_test_scaled) #Dự đoán trên tập test đã đc chuẩn hóa
    y_proba = model.predict_proba(self.X_test_scaled)[:, 1] #Dự đoán xác xuất trên tập test đã đc chuẩn hóa

    metrics = {
        "accuracy": accuracy_score(self.y_test, y_pred),
        "precision": precision_score(self.y_test, y_pred),
        "recall": recall_score(self.y_test, y_pred),
        "f1": f1_score(self.y_test, y_pred),
        "roc_auc": roc_auc_score(self.y_test, y_proba),
        "y_pred": y_pred,
        "y_proba": y_proba
    }
    return model, metrics
```

Xây dựng MLPClassifier có fine-tunning:

```python
# ================== Fine-tuning ==================
def finetuning(self):

    clf = MLPClassifier(max_iter=500, random_state=42) #Khởi tạo với 500 vòng lặp học
    param_grid = {
        'hidden_layer_sizes': [(50,), (100,), (150,), (100, 50)], #Chọn các tầng ẩn
        'activation': ['relu'], #Chọn hàm kích hoạt relu
        'solver': ['adam'], #Chọn thuật toán tối ưu adam
        'learning_rate_init': [0.001, 0.0005], #Chọn Tốc độ học ban đầu
        'alpha': [0.0001, 0.01], #Chọn hệ số regularization
        'max_iter': [500, 1000], #Chọn vòng học epoch
        'early_stopping': [True, False]
    }
    #Tìm tham số tốt nhất bằng RandomizedSearchCV
    #chia tập train thành 3 phần để validate, dùng tất cả CPU cores để train
    rd_search = RandomizedSearchCV(clf, para  (variable) y_train: Any   ndom_state=42)
    #Huấn luyện
    rd_search.fit(self.X_train_scaled, self.y_train)

    #Lấy model tốt nhất, dự đoán xác suất lớp dương(nhị phân)
    best_model = rd_search.best_estimator_
    y_proba = best_model.predict_proba(self.X_test_scaled)[:, 1]

    # Tìm threshold tối ưu theo recall
    best_t, _ = find_best_threshold(self.y_test, y_proba, metric="recall")
    y_pred_custom = (y_proba >= best_t).astype(int) #dự đoán trên ngưỡng tốt nhất

    metrics = {
        "accuracy": accuracy_score(self.y_test, y_pred_custom),
        "precision": precision_score(self.y_test, y_pred_custom),
        "recall": recall_score(self.y_test, y_pred_custom),
        "f1": f1_score(self.y_test, y_pred_custom),
        "roc_auc": roc_auc_score(self.y_test, y_proba),
        "y_pred": y_pred_custom,
        "y_proba": y_proba
    }
    return best_model, metrics
```

Xây dựng MLPClassifier có ensemble. Voting với 2 mô hình RandomForest, LogisticRegression

```python
# ================== Ensemble ==================
def ensemble(self, n_runs=10):
    """Xây dựng hàm emsemble với sự kết hợp của RandomForest MLpClassifier
    LogisticRegression"""
    #Khởi tạo biến lưu kết quả tốt nhất
    best_acc = -1
    best_model = None
    best_y_pred = None
    best_y_proba = None

    #Vì có RandomForest nên kết quả sẽ có tính ngẫu nhiên-> Cho chạy ensemble 10 lần và lấy cái tốt nhất
    for _ in range(n_runs):
        clf1 = LogisticRegression(max_iter=1000)
        clf2 = RandomForestClassifier(random_state=None)
        #Thông số của MLPClassifier
        clf3 = MLPClassifier(
            activation='relu',
            solver='adam',
            max_iter=1000,
            learning_rate_init=0.0005,
            hidden_layer_sizes=(50,),
            early_stopping=True,
            random_state=None,
            alpha=0.1,
        )
        #tạo ensemble bằng VotingClassifier
        model = VotingClassifier(
            estimators=[('lr', clf1), ('rf', clf2), ('mlp', clf3)],
            voting='soft' #Dự đoán xác suất của từng base model-> tính trung bình xác suất
        )
        #Huấn luyện và chọn model tốt nhất dựa trên accuracy
        model.fit(self.X_train_scaled, self.y_train)
        acc = model.score(self.X_test_scaled, self.y_test)
        if acc > best_acc:
            best_acc = acc
            best_model = model
            best_y_pred = model.predict(self.X_test_scaled)
            best_y_proba = model.predict_proba(self.X_test_scaled)[:, 1]

    metrics = {
        "accuracy": accuracy_score(self.y_test, best_y_pred),
        "precision": precision_score(self.y_test, best_y_pred),
        "recall": recall_score(self.y_test, best_y_pred),
        "f1": f1_score(self.y_test, best_y_pred),
        "roc_auc": roc_auc_score(self.y_test, best_y_proba),
        "y_pred": best_y_pred,
        "y_proba": best_y_proba
    }
    return best_model, metrics
```

Tìm ngưỡng tối ưu phù hợp với dữ liệu thực tế:

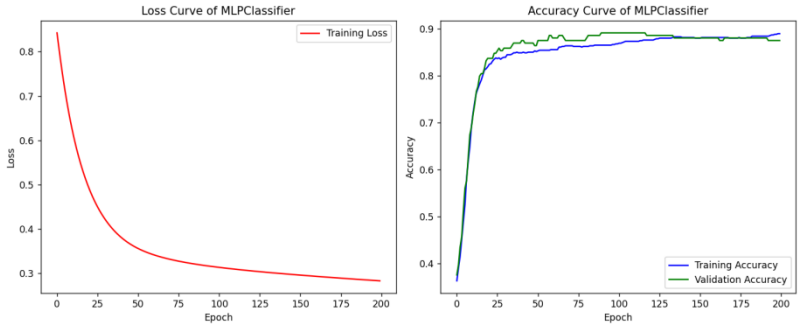
```python
#Tìm ngưỡng tốt nhất của model
def find_best_threshold(y_true, y_proba, metric="recall"):
    thresholds = np.linspace(0.1, 0.9, 81)
    best_t, best_score = 0.5, -1
    for t in thresholds:
        y_hat = (y_proba >= t).astype(int)
        score = recall_score(y_true, y_hat) if metric == "recall" else f1_score(y_true, y_hat)
        if score > best_score:
            best_score, best_t = score, t
    return best_t, best_score
```

```python
# ================== Hiệu chuẩn và tìm threshold ==================
def calibrate_and_find_threshold(self, model, X_val_scaled, y_val, metric="recall"):
    # Hiệu chuẩn xác suất
    calibrated = CalibratedClassifierCV(model, method='isotonic', cv=3)
    calibrated.fit(self.X_train_scaled, self.y_train)

    # Dự đoán xác suất trên tập validation
    y_proba_val = calibrated.predict_proba(X_val_scaled)[:, 1]

    # Tìm ngưỡng tối ưu theo metric
    best_t, best_score = find_best_threshold(y_val, y_proba_val, metric=metric)
    print(f"Ngưỡng tối ưu theo {metric}: {best_t:.2f} | {metric} đạt: {best_score:.4f}")

    # Lưu lại mô hình đã hiệu chuẩn và ngưỡng
    self.calibrated_model = calibrated
    self.best_threshold = best_t
    return calibrated, best_t
```

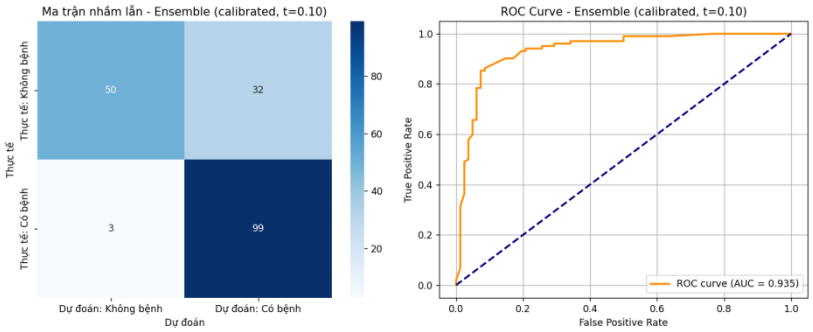
So sánh chỉ số của 3 mô hình MLPCLassifier và lấy mô hình tốt nhất theo ROC:

```
MLP cơ bản:
  accuracy: 0.8424
  precision: 0.8614
  recall: 0.8529
  f1: 0.8571
  roc_auc: 0.9006

MLP fine-tune:
  accuracy: 0.8152
  precision: 0.7698
  recall: 0.9510
  f1: 0.8509
  roc_auc: 0.9036

Ensemble:
  accuracy: 0.9185
  precision: 0.9143
  recall: 0.9412
  f1: 0.9275
  roc_auc: 0.9455

==================================================
Mô hình tốt nhất theo ROC AUC: Ensemble với roc_auc = 0.9455
==================================================
```

Biểu đồ Loss curve và Accuracy curve của mô hình:



Biểu đồ ma trận nhầm lẫn và đường cong ROC của mô hình:



5. Kết quả và đánh giá mô hình

a. Kết quả

Nhập với 2 giá trị như sau:

- Người thứ nhất: age(30), sex(0), ChestPainType(ATA), RestingBP(115), Cholesterol(180), FastingBS(0), RestingECG(normal), MaxHR(185), ExerciseAngina(0), Oldpeak(0.5), ST_Slope(0) => Người khả năng mắc bệnh về tim thấp
- Người thứ hai: age(65), sex(1), ChestPainType(ASY), RestingBP(170), Cholesterol(280), FastingBS(1), RestingECG(LVH), MaxHR(100), ExerciseAngina(1), Oldpeak(3.0), ST_Slope(2) => Người có vấn đề về tim.

❖ MLPCLassifier:

- Kết quả người thứ nhất:

```
Mô hình: MLP
KẾT QUẢ: KHÔNG CÓ BỆNH TIM
Xác suất: 2.22%
```

- Kết quả người thứ 2:

```
Mô hình: MLP
KẾT QUẢ: CÓ BỆNH TIM
Xác suất: 36.63%
```

❖ GradientBoosting:

- Kết quả người thứ nhất:

```
Mô hình: GradientBoosting
KẾT QUẢ: KHÔNG CÓ BỆNH TIM
Xác suất: 3.36%
```

- Kết quả người thứ 2:

```
Mô hình: GradientBoosting
KẾT QUẢ: CÓ BỆNH TIM
Xác suất: 88.22%
```

❖ Decision Tree

- Kết quả người thứ nhất:

```
========================================
Mô hình: DecisionTree
KẾT QUẢ: KHÔNG CÓ BỆNH TIM
Xác suất: 47.76%
```

- Kết quả người thứ 2

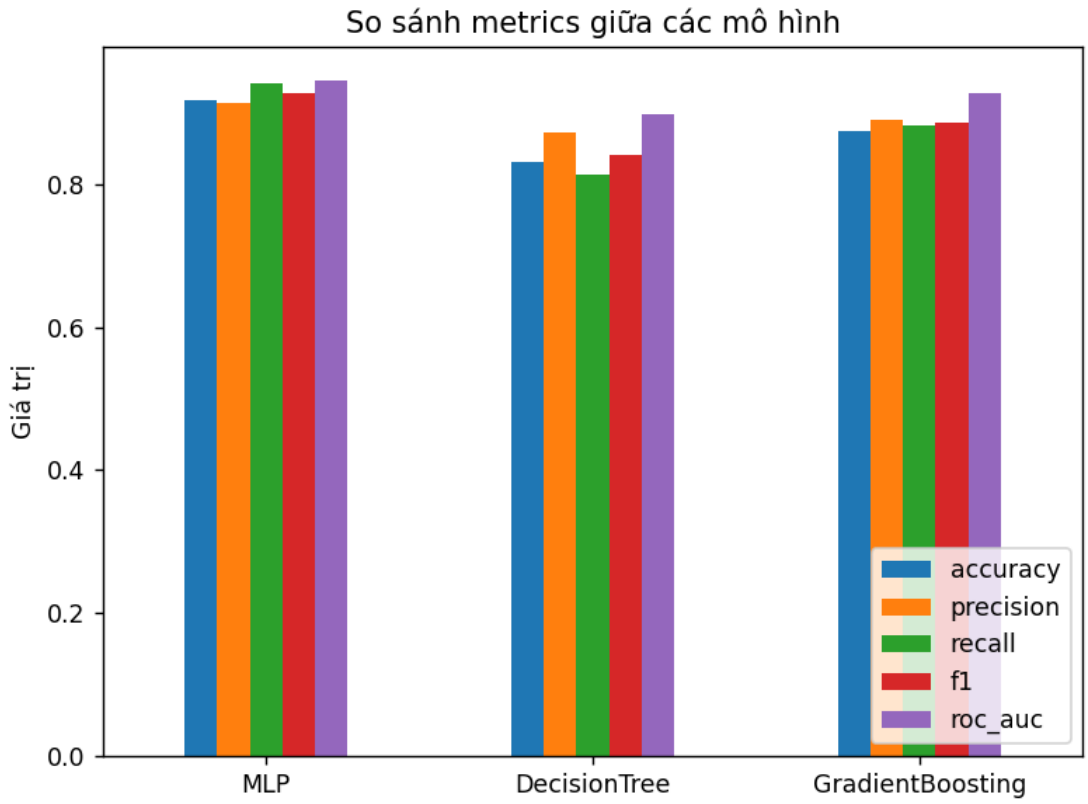
```
Mô hình: DecisionTree
KẾT QUẢ: CÓ BỆNH TIM
Xác suất: 83.33%
```

➢ Xác suất phát hiện người có bệnh tim của mô hình GradientBoosting chiếm % cao nhất trong 3 mô hình.

b. Đánh giá mô hình

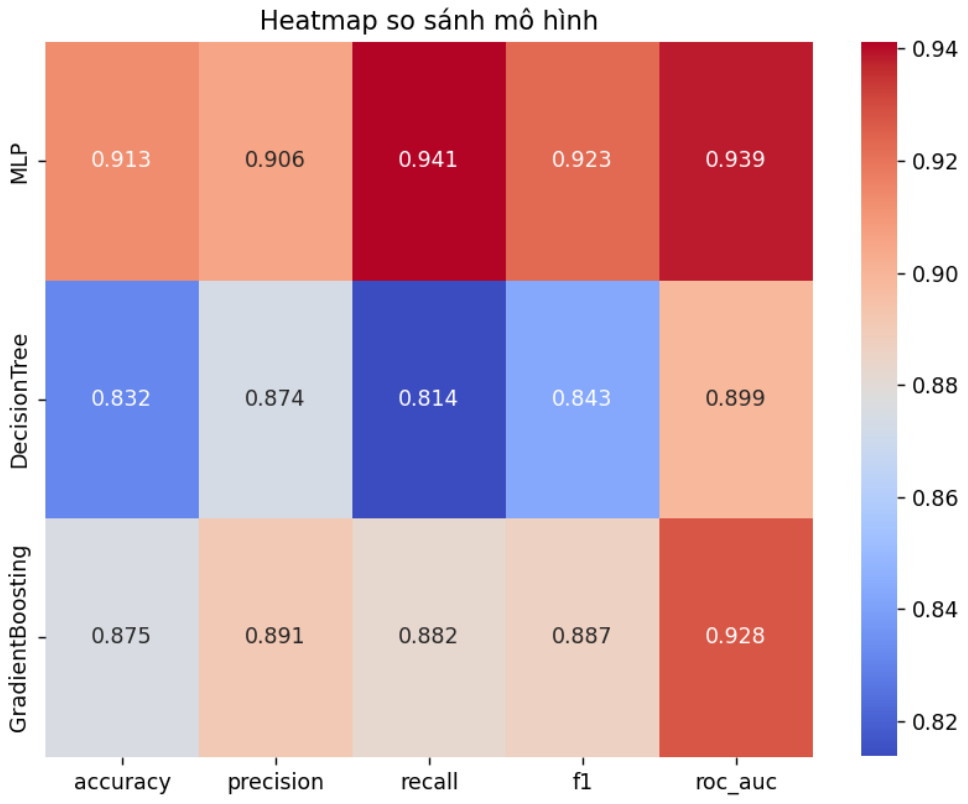
Các chỉ số pecision/recall/f1-score/roc-auc của 3 mô hình MLP Classifier, Decision Tree, GradientBoosting. Trong đó có thể nhận thấy MLP Classifier và GradientBoosting có

giá trị khá tương đương nhau. Các chỉ số của 3 model đều trên 0.8.



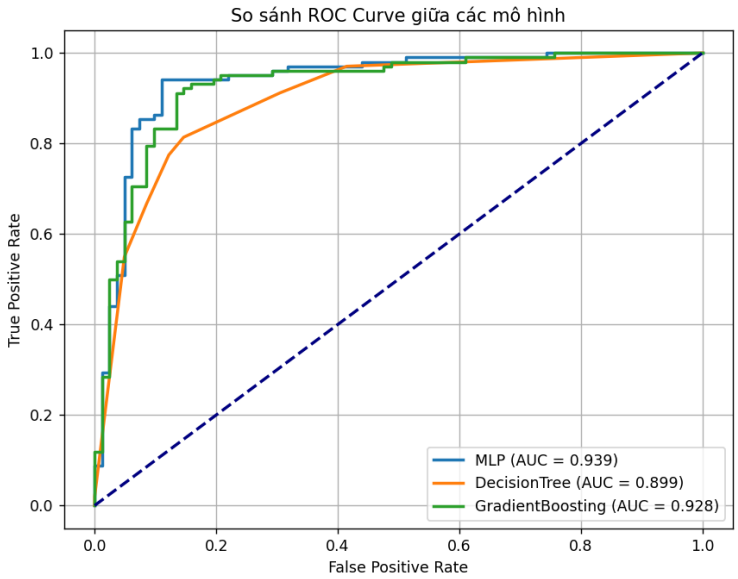
Hình 7 So sánh metrics giữa các mô hình

Chỉ số mô hình MLPClassifier cao nhất, đến GradientBoosting, cuối cùng là Decision Tree.



Hình 8 Heatmap so sánh mô hình

Dựa biểu đồ ROC, chỉ số AUC MLPClassifier có hiệu năng cao nhất, GradientBoosting xấp xỉ MLP nhưng chạy ổn định và mạnh hơn MLP, và chỉ số Decision Tree có phần thấp hơn nên khả năng phân tích sẽ yếu hơn so với 2 mô hình còn lại.



Hình 9 So sánh ROC Curve giữa các mô hình

III.Tổng kết:

1. Kết luận

Sau khi tiến hành dự đoán thông qua ba mô hình khác nhau gồm MLPClassifier, Decision Tree và Gradient Boosting, kết quả cho thấy rằng Gradient Boosting mang lại hiệu năng ổn định và chính xác hơn so với hai mô hình còn lại. Mặc dù MLPClassifier đạt giá trị AUC cao nhất trên đường cong ROC, nhưng mô hình này khá nhạy cảm với tham số huấn luyện và dễ bị ảnh hưởng bởi dữ liệu nhiễu. Decision Tree thì đơn giản, dễ hiểu nhưng lại có xu hướng overfit và cho kết quả phân loại kém hơn. Ngược lại, Gradient Boosting với cơ chế kết hợp nhiều cây quyết định nhỏ đã giúp giảm thiểu overfit, duy trì sự cân bằng giữa precision và recall, đồng thời cho khả năng dự đoán ổn định hơn trong thực tế. Chính vì vậy, xét trên tổng thể, Gradient Boosting được đánh giá là mô hình phù hợp hơn để áp dụng trong bài toán dự đoán bệnh tim.

2. Hướng phát triển

Sau khi phân loại lựa chọn model tốt nhất nhằm áp dụng thuật toán dự đoán bệnh nhân có bệnh hay không với xác suất cao nhất, Bayesian optimization: Sử dụng mô hình xác suất để tìm ra các tổ hợp giá trị có khả năng cao mang lại hiệu quả tốt.

3. Tài liệu tham khảo và phân công

Nguồn tham khảo:

https://www.kaggle.com/code/shaimamedini/classification-the-salary-level-with-knn-and-svm

https://www.kaggle.com/code/vasylmoisieiev/dl-final-project

https://www.kaggle.com/code/fangya/machine-learning-survival-for-heart-failure

https://www.kaggle.com/code/surajjha101/heart-failure-prediction-svm-and-ann

https://www.kaggle.com/code/surajjha101/heart-failure-prediction-svm-and-ann

https://www.kaggle.com/code/rude009/heart-failure-model-prediction-comparisons-95

https://www.kaggle.com/code/tanmay111999/heart-failure-prediction-eda-model-comparison

Nguyễn Minh Thiện: tiền xử lý, Encode , mô hình Gradient Boosting

Nguyễn Ngọc Châu Giang: tiền xử lý, Biểu đồ phân tích ngoại lai, ma trận tương quan, phân tích theo độ tuổi, đánh giá mô hình, hàm nhập, mô hình MLPClassifier.

Nguyễn Thị Yến Ngọc: Biểu đồ phân phối dữ liệu, phân phối theo mục tiêu, phân tích theo giới tính, mô hìnhDecision Tree