



Chủ đề: Khai phá dữ liệu

Kho dữ liệu và khai phá dữ liệu - Lớp 01 -
Nhóm 05

Hoàng Hữu Đức - B21DCCN240

Nguyễn Việt Khiêm - B21DCCN458

Lương Việt Hùng - B21DCCN410

Phạm Văn Chiến - B19DCCN101

Nguyễn Đăng Quang - B21DCCN628

Bộ môn Hệ thống thông tin

Khoa Công nghệ thông tin 1

Kỹ sư Hệ thống thông tin

Hà Nội, May 2025



Chủ đề: Khai phá dữ liệu

Kho dữ liệu và khai phá dữ liệu - Lớp 01 -
Nhóm 05

Hoàng Hữu Đức - B21DCCN240

Nguyễn Việt Khiêm - B21DCCN458

Lương Việt Hùng - B21DCCN410

Phạm Văn Chiến - B19DCCN101

Nguyễn Đăng Quang - B21DCCN628

Giảng viên hướng dẫn: Trưởng nhóm: Nguyễn Quỳnh Chi
Hệ thống thông tin - CNTT1 - PTIT

Bộ môn Hệ thống thông tin
Khoa Công nghệ thông tin 1
Kỹ sư Hệ thống thông tin

Hà Nội, May 2025

Bảng phân chia công việc nhóm

STT	Họ tên	Mã SV	Công việc
1	Nguyễn Việt Khiêm	B21DCCN458	Phát biểu, kiểm thử, đánh giá hiệu quả và tính đúng đắn giả thuyết. Thủ nghiệm up/down scaling data EDA data. Data processing Feature Extracting Thử nghiệm, tinh chỉnh và kiểm thử mô hình
2	Lương Việt Hùng	B21DCCN410	Xây dựng và kiểm thử thuật toán MLP Xây dựng và kiểm thử thuật toán Tabnet Data processing EDA data
3	Hoàng Hữu Đức	B21DCCN240	Xây dựng và kiểm thử thuật toán Random Forest Xây dựng và kiểm thử thuật toán optimized Decision Tree by numpy, C++ Xây dựng và kiểm thử thuật toán Tabnet EDA data
4	Phạm Văn Chiến	B19DCCN101	Xây dựng và kiểm thử thuật toán Decision Tree by numpy Xây dựng và kiểm thử thuật toán MLP Thử nghiệm up/down scaling data
5	Nguyễn Đăng Quang	B21DCCN628	Xây dựng và kiểm thử thuật toán Decision Tree Xây dựng và kiểm thử thuật toán Tabnet

			EDA data
--	--	--	----------

I. Xây dựng thuật toán	3
1. Tiền xử lý dữ liệu	3
2. Thuật toán Decision Tree	4
2.1 Giới thiệu	4
2.2 Xây dựng decision tree	5
2.3 Tối ưu thuật toán Decision Tree:	9
2.3.1 Build thuật toán với C++	9
2.3.2 Tối ưu tạo cây quyết định ban đầu với python	18
2.4 So sánh hiệu năng và chứng minh thuật toán tự xây dựng là đúng với Decision Tree của thư viện Scikit-Learn	21
3. Thuật toán Random Forest	23
3.1 Giới thiệu	23
3.2 Xây dựng thuật toán Random Forest	24
3.3 So sánh và chứng minh thuật toán đã xây dựng là đúng với Random Forest trong Scikit-Learn	32
4. Thuật toán TabNet	34
4.1 Giới thiệu	34
4.2 Xây dựng thuật toán TabNet	34
4.3 So sánh và chứng minh thuật toán đã xây dựng là đúng với TabNet trong pytorch-tabnet	38
5. Thuật toán Multilayer Perceptron (MLP)	41
5.1 Giới thiệu	41
5.2 Xây dựng thuật toán MLP	42
5.3 So sánh và chứng minh thuật toán đã xây dựng là đúng với MLP trong pytorch-mlp	46
II. Khảo sát và đặt ra giả thiết trên bộ dữ liệu	47
1. Tìm hiểu investigate, muốn kiểm tra/khảo sát điều gì từ bộ dữ liệu (Investigating and Defining What to Examine)	47
a. Data Exploration (Một số biểu đồ phân bổ chính)	48
b. Data Analyzing	53
2. Đưa ra statement, phát biểu của nghiên cứu (Formulating a Statement/Research Hypothesis)	59
3. Thiết kế các thử nghiệm dựa trên bộ dữ liệu mình có (Designing Experiments)	59
III. Kết luận	69

Đề tài: Xây dựng và kiểm thử 4 mô hình Decision Tree, Random Forest, MLP, Tabnet. Chọn mô hình tốt nhất (Random Forest) để giải quyết bài toán Phân loại khách hàng có nhu cầu đăng ký gửi tiền tiết kiệm của bộ dữ liệu Bank Marketing, thu thập bởi ngân hàng Bồ Đào Nha (5/2008 - 11/2010)

Link bộ dữ liệu: <https://archive.ics.uci.edu/dataset/222/bank+marketing>

I. Xây dựng thuật toán

1. Tiền xử lý dữ liệu

```
> df.info()
[5]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45211 entries, 0 to 45210
Data columns (total 17 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   age          45211 non-null   int64  
 1   job          45211 non-null   object  
 2   marital      45211 non-null   object  
 3   education    45211 non-null   object  
 4   default      45211 non-null   object  
 5   balance      45211 non-null   int64  
 6   housing      45211 non-null   object  
 7   loan          45211 non-null   object  
 8   contact      45211 non-null   object  
 9   day           45211 non-null   int64  
 10  month         45211 non-null   object  
 11  duration     45211 non-null   int64  
 12  campaign     45211 non-null   int64  
 13  pdays        45211 non-null   int64  
 14  previous     45211 non-null   int64  
 15  poutcome     45211 non-null   object  
 16  y             45211 non-null   object  
dtypes: int64(7), object(10)
memory usage: 5.9+ MB
```

Kiểu dữ liệu

	age	balance	day	duration	campaign	pdays	previous
count	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000
mean	40.936210	1362.272058	15.806419	258.163080	2.763841	40.197828	0.580323
std	10.618762	3044.765829	8.322476	257.527812	3.098021	100.128746	2.303441
min	18.000000	-8019.000000	1.000000	0.000000	1.000000	-1.000000	0.000000
25%	33.000000	72.000000	8.000000	103.000000	1.000000	-1.000000	0.000000
50%	39.000000	448.000000	16.000000	180.000000	2.000000	-1.000000	0.000000
75%	48.000000	1428.000000	21.000000	319.000000	3.000000	-1.000000	0.000000
max	95.000000	102127.000000	31.000000	4918.000000	63.000000	871.000000	275.000000

Tóm tắt dữ liệu

- Xử lý giá trị null: trong bộ dữ liệu không có giá trị null nên nhóm không xử lý
- Giá trị khuyết thiêú: đối với giá trị liên tục là -1, đối với giá trị rời rạc là unknown. Thay vì fill bằng mean/mead, nhóm sẽ sử dụng như một trường hợp của bản ghi dữ liệu
- Scaling: Dữ liệu liên tục, nhóm sẽ normalize bằng Standard Scaler về khoảng
- Encoding: Dữ liệu rời rạc, nhóm sẽ thực hiện One Hot Encoder

Các tiêu chí đánh giá thuật toán

-Accuracy (Độ chính xác)

Tỉ lệ dự đoán đúng trên tổng số mẫu.

Phù hợp khi dữ liệu cân bằng.

$$\text{Accuracy} = (\text{TP} + \text{TN}) / (\text{TP} + \text{TN} + \text{FP} + \text{FN})$$

-Precision (Độ chính xác dương)

Trong các mẫu được dự đoán là positive, có bao nhiêu là đúng?

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

-Recall (Độ nhạy)

Trong các mẫu thật sự positive, mô hình thấy được bao nhiêu?

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$

-F1 Score

Trung bình điều hòa của precision và recall.

Dùng khi cần cân bằng precision & recall.

$$\text{F1} = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$$

-AUC (Area Under Curve - ROC)

Đo khả năng phân biệt giữa 2 lớp (positive/negative).

- AUC = 1: hoàn hảo
- AUC = 0.5: ngẫu nhiên

Phù hợp khi dữ liệu mất cân bằng.

2. Thuật toán Decision Tree

2.1 Giới thiệu

- Decision Tree (cây quyết định) là một thuật toán học máy (machine learning) được sử dụng cho cả bài toán phân loại (classification) và hồi quy (regression). Thuật toán hoạt động giống như cách con người ra quyết định dựa trên một loạt câu hỏi "nếu... thì...".

- Chi tiết cách mà cây quyết định được xây dựng:

- + Chọn thuộc tính tốt nhất để phân tách dữ liệu: Có nhiều phương pháp để chọn thuộc tính tốt nhất, phổ biến nhất là sử dụng các phép đo như Gini impurity hoặc Entropy để đánh giá mức độ tinh khiết của mỗi thuộc tính.
 - + Tách dữ liệu dựa trên thuộc tính được chọn: Dựa trên thuộc tính được chọn, dữ liệu được chia thành các nhóm con tại mỗi nút.
 - + Lặp lại quá trình cho các nhóm con: Quá trình này được lặp lại cho mỗi nhóm con, tiếp tục chia dữ liệu đến khi một điều kiện dừng được đáp ứng, như đạt đến độ sâu tối đa của cây, không còn dữ liệu phân chia hoặc không còn thay đổi đáng kể trong tinh khiết của các nhóm con.
 - + Xác định các lá của cây: Khi quá trình phân chia kết thúc, các nút cuối cùng của cây được gọi là lá, và chúng chứa các quyết định cuối cùng hoặc giá trị dự đoán.
- Ứng dụng:
1. Phân loại (Classification): Chẩn đoán y khoa, phân loại bệnh dựa trên triệu chứng, phân loại khách hàng, nhận diện nhóm khách hàng tiềm năng.
 2. Dự đoán số (Regression): Dự báo giá nhà/xe, dự đoán doanh thu, lợi nhuận theo yếu tố đầu vào.
 3. Trích xuất luật (Rule extraction): Tạo quy tắc dễ hiểu (nếu...thì...) → phù hợp cho hệ thống giải thích được.
 4. Tiền xử lý & trích chọn đặc trưng: Dùng tree để chọn feature quan trọng → giảm chiều dữ liệu.

- Ưu điểm:

1. Dễ hiểu và dễ diễn giải.
2. Có thể xử lý cả dữ liệu số và dữ liệu phân loại.
3. Độ phức tạp thấp trong quá trình phân tích và dự đoán.

- Nhược điểm:

1. Dễ bị overfitting nếu cây quá phức tạp.
2. Không đảm bảo chắc chắn cho việc tìm ra các mô hình tốt nhất.

2.2 Xây dựng decision tree

Hàm tính chỉ số Gini impurity

```

def gini_index(groups, classes):
    n_instances = float(sum(len(group) for group in groups))
    gini = 0.0
    for group in groups:
        size = float(len(group))
        if size == 0:
            continue
        score = 0.0
        labels = [row[-1] for row in group]
        for class_val in classes:
            p = labels.count(class_val) / size
            score += p ** 2
        gini += (1.0 - score) * (size / n_instances)
    return gini

```

+ gini càng thấp thì điểm được xét càng hiệu quả.

+ Công thức tính:

$$\text{Gini}(D) = 1 - \sum_{i=1}^C p_i^2$$

- Trong đó C là số lượng list của mỗi nút được xét.
- P là tỷ lệ phần tử có giá trị trùng khớp với giá trị của list(Giả sử list left được gán = 0)

Hàm này giúp chọn cách chia nào là tốt nhất, bằng cách tối thiểu hóa chỉ số Gini – tức là làm cho các nhóm sau chia càng tinh khiết càng tốt, nhiều mẫu cùng lớp hơn

- Hàm chia nhóm:

```

def test_split(index, value, dataset):
    left, right = [], []
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right

```

Tách tập dữ liệu thành hai phần (left và right) dựa trên điều kiện: Nếu giá trị ở cột index của dòng $<$ value \rightarrow cho vào left, ngược lại \rightarrow cho vào right

- Hàm get_split:

```
def get_split(dataset):
    class_values = list(set(row[-1] for row in dataset))
    best_index, best_value, best_score, best_groups = 999, 999, 999, None
    for index in range(len(dataset[0]) - 1):
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            if gini < best_score:
                best_index, best_value, best_score, best_groups = index, row[index], gini, groups
    return {'index': best_index, 'value': best_value, 'groups': best_groups}
```

Trích ra tất cả các giá trị nhãn (label) trong tập dữ liệu (row[-1] là nhãn). Dùng set để loại bỏ trùng lặp → class_values là danh sách các lớp duy nhất (VD: [0, 1]).

Tạo ra 4 biến:

- best_index: cột nào chia tốt nhất.
- best_value: giá trị tại cột đó dùng để chia.
- best_score: Gini index thấp nhất (càng thấp càng tốt).
- best_groups: cặp (left, right) tốt nhất.

Với mỗi cột, thử chia theo từng giá trị có trong tập dữ liệu.

Dùng test_split để chia thành 2 nhóm.

Tính Gini index cho cách chia này.

b_score được mặc định bằng 999, dùng để gán giá trị gini tốt nhất hiện tại. Nếu cách chia này cho Gini thấp hơn, lưu lại tất cả thông tin.

- Hàm tạo lá

```
def to_terminal(group):
    outcomes = [row[-1] for row in group]
    return max(set(outcomes), key=outcomes.count)
```

trả về nhãn phổ biến nhất trong nhóm → đây sẽ là giá trị dự đoán của cây tại nút lá
Kiểm tra các điều kiện dừng:

- Không có dữ liệu để chia: Nếu một trong các nhóm con (left hoặc right) rỗng sau khi get_split được gọi ở bước trước, nút hiện tại sẽ không được chia nữa, và cả hai con trỏ left và right của nó sẽ trỏ đến một nút lá chứa toàn bộ dữ liệu của nút hiện tại.
- Đạt độ sâu tối đa (max_depth): Nếu cây đã phát triển đến độ sâu tối đa cho phép, nút hiện tại sẽ trở thành nút cha của hai nút lá, mỗi nút lá chứa dữ liệu của nhóm left và right tương ứng.

Xử lý đệ quy cho nhánh con:

- Nhánh trái: Nếu số lượng mẫu trong nhóm left nhỏ hơn hoặc bằng min_size kích thước tối thiểu để chia, thì nhánh left sẽ trở thành một nút lá. Ngược lại,

hàm get_split(left) sẽ được gọi để tìm cách chia tốt nhất cho nhóm left, và sau đó hàm split lại được gọi đệ quy cho nút con left mới này.

- Nhánh phải:Tương tự như nhánh trái, nếu nhóm right quá nhỏ, nó sẽ trở thành nút lá.Ngược lại, get_split(right) được gọi, và hàm split lại được gọi đệ quy cho nút con right mới.

- Hàm xây cây:

```
def build_tree(train, max_depth, min_size):  
    root = get_split(train)  
    split(root, max_depth, min_size, 1)  
    return root  
  
def split(node, max_depth, min_size, depth):  
    left, right = node['groups']  
    del(node['groups'])  
    if not left or not right:  
        node['left'] = node['right'] = to_terminal(left + right)  
        return  
    if depth >= max_depth:  
        node['left'], node['right'] = to_terminal(left), to_terminal(right)  
        return  
    if len(left) <= min_size:  
        node['left'] = to_terminal(left)  
    else:  
        node['left'] = get_split(left)  
        split(node['left'], max_depth, min_size, depth + 1)  
    if len(right) <= min_size:  
        node['right'] = to_terminal(right)  
    else:  
        node['right'] = get_split(right)  
        split(node['right'], max_depth, min_size, depth + 1)
```

Dùng để xây dựng cây, biến đầu vào là bộ dữ liệu huấn luyện, giá trị max_depth, min_size. Max_depth là chiều sâu lớn nhất của cây, min_size là số bản ghi tối thiểu để tiếp tục phân nhánh.

-Hàm predict:

```

def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

```

Đầu vào là 1 đối tượng từ điển: node, bản ghi đang xét: row

Dùng để dự đoán nhãn các bản ghi trong tập dữ liệu test

Xét bản ghi nằm ở nhánh nào tại nút đang xét, nếu nhánh đó chưa đến nút

lá thì tiếp tục gọi hàm predict, nếu đã đến nút lá thì trả về nhãn của nhánh

-Hàm global_predict:

```

def global_predict(tree, test_data):
    predictions = []
    for row in test_data:
        predictions.append(predict(tree, row))
    return predictions

```

Hàm global_predict được thiết kế để đưa ra dự đoán cho toàn bộ một tập dữ liệu kiểm tra bằng cách sử dụng một cây quyết định đã được huấn luyện.

2.3 Tối ưu thuật toán Decision Tree:

2.3.1 Build thuật toán với C++

Struct Node:

```

struct Node {
    int index;
    double value;
    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;
    int prediction;
    double score;

    Node(int pred) : index(-1), value(0.0), left(nullptr), right(nullptr), prediction(pred), score(0.0) {}
    Node(int idx, double val, double sc) : index(idx), value(val), left(nullptr), right(nullptr), prediction(-1), score(sc) {}
};


```

index: Lưu chỉ số của đặc trưng được sử dụng để chia tại nút này. Nếu là nút lá, giá trị này thường không có ý nghĩa (ví dụ: -1).

value: Giá trị ngưỡng của đặc trưng index. Các mẫu có giá trị đặc trưng nhỏ hơn ngưỡng này sẽ đi sang con trái, ngược lại đi sang con phải.

left: Con trỏ đến nút con bên trái.

right: Con trỏ đến nút con bên phải.

iprediction: Nếu đây là nút lá, nó lưu giữ nhãn dự đoán cho các mẫu rơi vào nút này. Nếu không phải nút lá, giá trị này thường là -1.

score: Lưu trữ điểm Gini impurity của nút này *trước khi* nó được chia. Điểm này có thể được sử dụng để tính toán feature importance.

Constructors:

- Node(int pred): Dùng để tạo một nút lá.
 - Đầu vào: int pred (nhãn dự đoán).
 - Khởi tạo index = -1, value = 0.0, con trỏ left và right là nullptr, prediction = pred, và score = 0.0.
- Node(int idx, double val, double sc): Dùng để tạo một nút chia .
 - Đầu vào: int idx (chỉ số đặc trưng), double val (giá trị ngưỡng), double sc (điểm Gini).
 - Khởi tạo index = idx, value = val, con trỏ left và right là nullptr, prediction = -1, và score = sc.

Class Decision Tree:

Constructor DecisionTree():

```
DecisionTree(int max_depth = 5, int min_size = 1)
    : max_depth(max_depth), min_size(min_size), n_features_(-1) {}
```

Đầu vào:

max_depth (): Độ sâu tối đa của cây.

min_size (): Số lượng mẫu tối thiểu trong một nút để nút đó có thể được chia tiếp.

Hàm fit(): Huấn luyện cây

```
void fit(const std::vector<std::vector<double>>& X, const std::vector<int>& y) {
    if (X.empty() || X[0].empty()) {
        std::cerr << "Lỗi: Dữ liệu huấn luyện X rỗng." << std::endl;
        return;
    }
    n_features_ = X[0].size();

    flat_X.clear();
    flat_X.reserve(X.size() * n_features_);
    for (const auto& row : X) {
        flat_X.insert(flat_X.end(), row.begin(), row.end());
    }

    flat_y = y;

    std::vector<int> initial_indices(X.size());
    std::iota(initial_indices.begin(), initial_indices.end(), 0);

    root = split(flat_X, flat_y, initial_indices, 0);
}
```

Đầu vào:

- Dữ liệu huấn luyện. Là một vector các vector, mỗi vector con đại diện cho một mẫu dữ liệu, và các phần tử trong vector con là giá trị của các đặc trưng.
- Vector chứa các nhãn lớp tương ứng với từng mẫu trong X.

Đầu ra: Hàm này sửa đổi trạng thái của object DecisionTree bằng cách xây dựng cây và gán vào root

Hàm predict(): dự đoán nhãn cho 1 mẫu đơn lẻ

```
int predict(const std::vector<double>& row) const {
    if (!root) {
        std::cerr << "Lỗi: Cây chưa được huấn luyện." << std::endl;
        return -1;
    }
    auto node = root;
    while (node->left) {
        if (node->index >= row.size()) {
            std::cerr << "Lỗi: Chỉ số đặc trưng ngoài phạm vi trong quá trình dự đoán." << std::endl;
            return node->prediction != -1 ? node->prediction : 0;
        }
        if (row[node->index] < node->value)
            node = node->left;
        else
            node = node->right;
    }
    return node->prediction;
}
```

Đầu vào: Một vector chứa các giá trị đặc trưng của mẫu cần dự đoán.

Đầu ra: Nhãn lớp được dự đoán cho mẫu row. Trả về -1 hoặc giá trị lỗi khác nếu cây chưa được huấn luyện hoặc có lỗi.

Hàm predict(): Dự đoán nhãn cho 1 tập các mẫu dữ liệu

```
std::vector<int> predict(const std::vector<std::vector<double>>& X) const {
    std::vector<int> preds;
    preds.reserve(X.size());
    for (const auto& row : X)
        preds.push_back(predict(row));
    return preds;
}
```

Đầu vào: Vector chứa các mẫu dữ liệu cần dự đoán.

Đầu ra: Vector chứa các nhãn dự đoán tương ứng cho từng mẫu trong X

Hàm get_feature_importances(): Tính toán và trả về feature_importances dựa trên giá trị Gini impurity tại các điểm chia.

```

std::vector<double> get_feature_importances() const {
    if (n_features_ == -1) {
        return {};
    }
    std::vector<double> importances(n_features_, 0.0);
    if (!root) return importances;

    std::vector<const Node*> stack;
    stack.push_back(root.get());

    while (!stack.empty()) {
        auto node = stack.back();
        stack.pop_back();

        if (node->left) {
            if (node->index >= 0 && node->index < n_features_) {
                importances[node->index] += node->score;
            }
            stack.push_back(node->left.get());
            stack.push_back(node->right.get());
        }
    }

    double sum = std::accumulate(importances.begin(), importances.end(), 0.0);
    if (sum > 0)
        for (auto& v : importances) v /= sum;
    return importances;
}

```

Đầu vào: Không có

Đầu ra: Vector chứa feature importances point được chuẩn hóa để tổng bằng 1 cho từng đặc trưng. Thứ tự tương ứng với thứ tự đặc trưng trong dữ liệu huấn luyện ban đầu.

Hàm tính giá trị Gini:

```

std::shared_ptr<Node> root;
int max_depth;
int min_size;
int n_features_;

std::vector<double> flat_X;
std::vector<int> flat_y;

double gini_index(const std::vector<std::vector<int>>& groups_indices) const {
    double gini = 0.0;
    size_t n_instances_in_node = 0;
    for (const auto& group_indices : groups_indices) {
        n_instances_in_node += group_indices.size();
    }

    if (n_instances_in_node == 0) return 0.0;

    for (const auto& group_indices : groups_indices) {
        if (group_indices.empty()) continue;
        size_t size = group_indices.size();
        double score = 0.0;
        std::unordered_map<int, int> class_counts;

        for (int idx : group_indices) {
            class_counts[flat_y[idx]]++;
        }

        for (const auto& pair : class_counts) {
            double proportion = static_cast<double>(pair.second) / size;
            score += proportion * proportion;
        }
        gini += (1.0 - score) * (static_cast<double>(size) / n_instances_in_node);
    }
    return gini;
}

```

Đầu vào: Một vector chứa các vector chỉ mục. Mỗi vector con (group_indices) chứa các chỉ mục (trong flat_X và flat_y) của các mẫu thuộc về một nhóm con sau khi chia. Thường thì groups_indices sẽ có 2 phần tử trái và phải

Đầu ra: Giá trị Gini impurity của cách chia đó. Giá trị càng thấp thì cách chia càng đúng

Hàm chia các mẫu thành 2 tập khác nhau dựa trên 1 đặc trưng nào đó:

```

std::pair<std::vector<int>, std::vector<int>> test_split(int feature_index, double value, const std::vector<int>& current_indices) const {
    std::vector<int> left_indices, right_indices;
    left_indices.reserve(current_indices.size());
    right_indices.reserve(current_indices.size());

    for (int sample_idx : current_indices) {
        if (feature_index >= n_features_) {
            std::cerr << "Lỗi: Chỉ số đặc trưng ngoài phạm vi trong test_split." << std::endl;
            continue;
        }
        if (flat_X[sample_idx * n_features_ + feature_index] < value)
            left_indices.push_back(sample_idx);
        else
            right_indices.push_back(sample_idx);
    }
    return {left_indices, right_indices};
}

```

Đầu vào:

- Chỉ số của đặc trưng dùng để chia.
- Giá trị ngưỡng để chia.
- Vector chứa các chỉ mục của các mẫu hiện tại cần được chia.

Đầu ra: Một cặp vector. Vector đầu tiên chứa chỉ mục của các mẫu thuộc nhóm con bên trái, vector thứ hai chứa chỉ mục của các mẫu thuộc nhóm con bên phải.

Hàm tìm feature và giá trị Gini thấp nhất để chia tập các dữ liệu:

```

std::shared_ptr<Node> get_split(const std::vector<int>& current_indices) const {
    if (current_indices.empty()) {
        return std::make_shared<Node>(-1);
    }

    double b_score = std::numeric_limits<double>::infinity();
    int b_index = -1;
    double b_value = 0.0;

    for (int index = 0; index < n_features_; index++) {
        std::vector<std::pair<double, int>> feat_label_pairs(current_indices.size());
        for (size_t i = 0; i < current_indices.size(); ++i) {
            int sample_idx = current_indices[i];
            feat_label_pairs[i] = {flat_X[sample_idx * n_features_ + index], flat_y[sample_idx]};
        }

        std::sort(feat_label_pairs.begin(), feat_label_pairs.end());

        for (size_t i = 1; i < feat_label_pairs.size(); ++i) {
            if (feat_label_pairs[i-1].second != feat_label_pairs[i].second ||
                feat_label_pairs[i-1].first != feat_label_pairs[i].first) {

                double split_value = (feat_label_pairs[i-1].first + feat_label_pairs[i].first) / 2.0;

                auto groups_indices = test_split(index, split_value, current_indices);

                if (groups_indices.first.empty() || groups_indices.second.empty()) {
                    continue;
                }

                std::vector<std::vector<int>> groups_vec = {groups_indices.first, groups_indices.second};
                double gini = gini_index(groups_vec);

                if (gini < b_score) {
                    b_index = index;
                    b_value = split_value;
                    b_score = gini;
                }
            }
        }
    }

    return std::make_shared<Node>(b_index, b_value, b_score);
}

```

Đầu vào: Vector chứa các chỉ mục của các mẫu trong nút hiện tại.

Đầu ra: Một con trỏ đến một node mới. Nút này chứa thông tin về đặc trưng tốt nhất, ngưỡng tốt nhất và Gini impurity tốt nhất của điểm chia đó. Con trỏ left và right của nút này chưa được gán sẽ được gán trong hàm split đệ quy. Nếu không tìm được điểm chia hợp lệ, có thể trả về một nút với index = -1.

Hàm tạo nút lá từ một nhóm các mẫu dữ liệu:

```

std::shared_ptr<Node> to_terminal(const std::vector<int>& current_indices) const {
    if (current_indices.empty()) {
        return std::make_shared<Node>(-1);
    }
    std::unordered_map<int, int> counts;
    for (int idx : current_indices) {
        counts[flat_y[idx]]++;
    }
    int max_count = 0;
    int prediction = -1;
    if (!counts.empty()) {
        for (const auto& pair : counts) {
            if (pair.second > max_count) {
                max_count = pair.second;
                prediction = pair.first;
            }
        }
    }
    return std::make_shared<Node>(prediction);
}

```

Đầu vào: Vector chứa các index của các mẫu trong nhóm sẽ trở thành nút lá.

Đầu ra: Một con trỏ đến một node mới được khởi tạo là nút lá. prediction của nút này là nhãn xuất hiện nhiều nhất trong current_indices.

Hàm đệ quy xây dựng cây:

Đầu vào:

- Dữ liệu flat_X (không thay đổi trong quá trình đệ quy).
- Dữ liệu flat_y (không thay đổi).
- Vector chứa các chỉ mục của các mẫu trong nút hiện tại đang được xử lý.
- Độ sâu hiện tại của nút.

Đầu ra: Con trỏ đến nút (hoặc cây con) đã được xây dựng từ current_indices.

Các interface để python sử dụng:

Hàm dt_create(): Tạo 1 Decision Tree mới:

```

DLL_EXPORT DecisionTreeHandle dt_create(int max_depth, int min_size) {
    return new DecisionTree(max_depth, min_size);
}

```

Đầu vào:

- max_depth: Số nguyên, độ sâu tối đa cho phép của cây.
- min_size: Số nguyên, số lượng mẫu tối thiểu trong một nút để nút đó có thể được chia tiếp.

Đầu ra:

- DecisionTreeHandle Một con trỏ đến đối tượng DecisionTree vừa được tạo trong bộ nhớ.

Hàm dt_destroy(): Giải phóng bộ nhớ vừa tạo cho cây

```
DLL_EXPORT void dt_destroy(DecisionTreeHandle handle) {
    delete static_cast<DecisionTree*>(handle);
}
```

Hàm dt_fit(): Huấn luyện cây quyết định dựa trên dữ liệu đầu vào.

```
DLL_EXPORT void dt_fit(DecisionTreeHandle handle, double* X_flat, int* y, int n_samples, int n_features) {
    auto* tree = static_cast<DecisionTree*>(handle);
    std::vector<std::vector<double>> X_vec_temp(n_samples, std::vector<double>(n_features));
    for (int i = 0; i < n_samples; ++i) {
        for (int j = 0; j < n_features; ++j) {
            X_vec_temp[i][j] = X_flat[i * n_features + j];
        }
    }
    std::vector<int> y_vec_temp(y, y + n_samples);
    tree->fit(X_vec_temp, y_vec_temp);
}
```

Đầu vào:

- handle: Con trỏ đến đối tượng DecisionTree cần huấn luyện.
- X_flat: Con trỏ đến một mảng dữ liệu đặc trưng đã được làm phẳng
- y: Con trỏ đến một mảng chứa các nhãn (labels) tương ứng với từng mẫu trong X_flat.
- n_samples: Số lượng mẫu trong tập huấn luyện.
- n_features: Số lượng đặc trưng cho mỗi mẫu.

Đầu ra:

- void. Hàm này sửa đổi trạng thái object DecisionTree

Hàm dt_predict(): Dự đoán nhãn cho 1 hoặc nhiều mẫu dữ liệu:

```

DLL_EXPORT void dt_predict(DecisionTreeHandle handle, double* X_flat, int n_samples, int n_features, int* out) {
    auto* tree = static_cast<DecisionTree*>(handle);
    std::vector<std::vector<double>> X_vec_temp(n_samples, std::vector<double>(n_features));
    for (int i = 0; i < n_samples; ++i) {
        for (int j = 0; j < n_features; ++j) {
            X_vec_temp[i][j] = X_flat[i * n_features + j];
        }
    }
    auto preds = tree->predict(X_vec_temp);
    for (int i = 0; i < n_samples; ++i)
        out[i] = preds[i];
}

```

Đầu vào:

- handle: Con trỏ đến đối tượng DecisionTree đã được huấn luyện.
- X_flat: Con trỏ đến mảng dữ liệu đặc trưng của các mẫu cần dự đoán, đã được làm phẳng.
- n_samples: Số lượng mẫu cần dự đoán.
- n_features: Số lượng đặc trưng cho mỗi mẫu.
- out: Con trỏ đến một mảng để lưu trữ các nhãn dự đoán.

Đầu ra:

- void. Kết quả dự đoán được ghi vào mảng out.

Hàm dt_feature_importances(): Tính toán và trả về feature_importances của từng feature trong cây quyết định đã huấn luyện.

```

DLL_EXPORT void dt_feature_importances(DecisionTreeHandle handle, double* out_importances, int n_features) {
    auto* tree = static_cast<DecisionTree*>(handle);
    auto importances = tree->get_feature_importances();
    for (int i = 0; i < n_features; ++i)
        out_importances[i] = (i < (int)importances.size() ? importances[i] : 0.0);
}

```

Đầu vào:

- handle: Con trỏ đến đối tượng DecisionTree đã được huấn luyện.
- out_importances: Con trỏ đến một mảng (đã được cấp phát bộ nhớ từ phía Python) để lưu trữ điểm số quan trọng của từng đặc trưng.
- n_features: Số lượng đặc trưng (kích thước của mảng out_importances).

Đầu ra: feature importance của các đặc trưng được ghi vào mảng out_importances.

2.3.2 Tối ưu tạo cây quyết định ban đầu với python

Các hàm ban đầu sẽ được chuyển qua sử dụng với numpy vì được built-in bằng C nên tốc độ sẽ nhanh hơn rất nhiều so với ban đầu

Hàm split ban đầu được viết với ý tưởng đệ quy thì được sử dụng thêm stack để tránh tràn bộ nhớ

```
def split(node, max_depth, min_size, depth):
    stack = [(node, depth)]

    while stack:
        current_node, current_depth = stack.pop()
        left, right = current_node['groups']
        del(current_node['groups'])

        if not left or not right:
            current_node['left'] = current_node['right'] = to_terminal(left + right)
            continue

        if current_depth >= max_depth:
            current_node['left'], current_node['right'] = to_terminal(left), to_terminal(right)
            continue

        if len(left) <= min_size:
            current_node['left'] = to_terminal(left)
        else:
            current_node['left'] = get_split(left, current_depth + 1)
            stack.append((current_node['left'], current_depth + 1))

        if len(right) <= min_size:
            current_node['right'] = to_terminal(right)
        else:
            current_node['right'] = get_split(right, current_depth + 1)
            stack.append((current_node['right'], current_depth + 1))
```

hàm này duyệt qua các nút cây xử lý bằng một stack, quyết định chia nút hoặc biến nó thành nút lá dựa trên các điều kiện. Các nút con mới được tạo ra sẽ được đưa trở lại stack để tiếp tục quá trình xây dựng cây. Ưu điểm chính của cách này là tránh lõi tràn bộ nhớ đẻ quy khi xây dựng các cây rất sâu.

Hàm predict sử dụng vòng lặp thay vì đệ quy như trước:

```
def predict(node, row):
    current = node
    while isinstance(current, dict):
        if row[current['index']] < current['value']:
            current = current['left']
        else:
            current = current['right']
    return current
```

Hàm global_predict tối ưu bằng vectorize

```
def global_predict(tree, test_data):
    test_data = np.array(test_data)
    predictions = np.zeros(len(test_data), dtype=int)

    for i in range(len(test_data)):
        predictions[i] = predict(tree, test_data[i])

    return predictions
```

Hàm cắt tỉa cây:

```
def prune_tree(node, X_val, y_val, min_impurity_decrease=0.0):
    if not isinstance(node, dict):
        return node

    y_pred = global_predict(node, X_val)
    acc_before = accuracy_score(y_val, y_pred)

    left = prune_tree(node['left'], X_val, y_val, min_impurity_decrease)
    right = prune_tree(node['right'], X_val, y_val, min_impurity_decrease)

    if not isinstance(left, dict) and not isinstance(right, dict):
        merged_node = {'index': node['index'], 'value': node['value'],
                      'left': left, 'right': right}
        y_pred = global_predict(merged_node, X_val)
        acc_after = accuracy_score(y_val, y_pred)

        if acc_after >= acc_before - min_impurity_decrease:
            return to_terminal(np.column_stack((X_val, y_val)))

    node['left'] = left
    node['right'] = right
    return node
```

Hàm prune_tree này thực hiện kỹ thuật **cắt tỉa cây quyết định** để giảm độ phức tạp của cây và có khả năng cải thiện hiệu suất trên dữ liệu mới bằng cách tránh overfitting.

Hàm xây cây giờ có áp dụng post-pruning nếu có validation set

```

def build_tree(train, max_depth, min_size, max_features=None, X_val=None, y_val=None, min_impurity_decrease=0.0):
    root = get_split(train, 0, max_features)
    split(root, max_depth, min_size, 1)

    # Áp dụng post-pruning nếu có validation set
    if X_val is not None and y_val is not None:
        root = prune_tree(root, X_val, y_val, min_impurity_decrease)

    return root

```

2.4 So sánh hiệu năng và chứng minh thuật toán tự xây dựng là đúng với Decision Tree của thư viện Scikit-Learn

```

● (venv) PS C:\Users\ASUS\Desktop\Techclub\data_mining> python .\test_performance.py
ROOT: C:\Users\ASUS\Desktop\Techclub\data_mining

Distribution of classes in combined dataset:
Class 0: 3405 samples (33.5%)
Class 1: 3392 samples (33.3%)
Class 2: 3381 samples (33.2%)

[C++ ctypes Version]
Running cross-validation for C++ ctypes version...
- Build time: 56.4254 seconds
- Cross-validation scores: [0.74312377 0.74901768 0.74459725 0.74348894 0.74987715]
- Mean accuracy: 74.60%
- Std accuracy: 0.29%

[Optimized Version]
Running cross-validation for optimized version...
Depth 0, Gini score: 0.5982
Depth 0, Gini score: 0.6011
Depth 0, Gini score: 0.6011
Depth 0, Gini score: 0.6045
Depth 0, Gini score: 0.5992
- Build time: 39.0411 seconds
- Cross-validation scores: [0.74066798 0.74901768 0.72937132 0.75331695 0.74545455]
- Mean accuracy: 74.36%
- Std accuracy: 0.82%

[Scikit-learn Version]
Running cross-validation for scikit-learn version...
- Build time: 0.4100 seconds
- Cross-validation scores: [0.74459725 0.74852652 0.74607073 0.74152334 0.75233415]
- Mean accuracy: 74.66%
- Std accuracy: 0.37%
Depth 0, Gini score: 0.5994

[Detailed Comparison on Test Set]

```

Optimized Implementation:

	precision	recall	f1-score	support
0	0.84	0.82	0.83	682
1	0.77	0.72	0.74	702
2	0.66	0.72	0.69	652
accuracy			0.75	2036
macro avg	0.76	0.75	0.76	2036
weighted avg	0.76	0.75	0.76	2036

Scikit-learn Implementation:

	precision	recall	f1-score	support
0	0.84	0.79	0.82	682
1	0.79	0.70	0.74	702
2	0.63	0.75	0.69	652
accuracy			0.75	2036
macro avg	0.75	0.75	0.75	2036
weighted avg	0.76	0.75	0.75	2036

C++ ctypes Implementation:

	precision	recall	f1-score	support
0	0.84	0.78	0.81	682
1	0.78	0.70	0.74	702
2	0.63	0.75	0.69	652
accuracy			0.74	2036
macro avg	0.75	0.74	0.75	2036
weighted avg	0.75	0.74	0.75	2036

Phiên bản	Độ chính xác trung bình	Thời gian build	Ưu điểm
Scikit-learn	74.66%	0.41s	Nhanh, tối ưu cao, tiện dụng

C++ types	74.60%	56.42s	Tự triển khai từ đầu, học tập tốt
Python Optimized	74.36%	39.04s	Tối ưu thuật toán tốt hơn C++ gốc

Chi tiết kết quả test set f1-score và recall tương đối giống:

- Precision/Recall/F1-score giữa các phiên bản rất sát nhau.
- Phiên bản sklearn vẫn cho kết quả nhanh nhất và ổn định nhất.

Kết luận:

- Các phiên bản đều hoạt động ổn định và hiệu năng gần tương đương (~74–75%). -> thuật toán build from scratch là đúng so với thư viện
- Viết bằng C++ hoặc tối ưu hóa chủ yếu giúp kiểm soát nội bộ và học thuật, chưa vượt hiệu năng sklearn.

3. Thuật toán Random Forest

3.1 Giới thiệu

- Random forest là một thuật toán học máy phô biến dùng để giải quyết các bài toán phân loại và dự đoán. Random Forest xây dựng một tập hợp cây quyết định ngẫu nhiên, mỗi cây được huấn luyện trên một tập dữ liệu con được lấy mẫu ngẫu nhiên từ tập dữ liệu huấn luyện ban đầu. Các cây trong tập hợp được sử dụng để thực hiện việc phân loại hoặc dự đoán cho các mẫu mới bằng cách đưa ra quyết định bằng cách "bầu chọn" kết quả của từng cây.

- Ứng dụng:

1. Chẩn đoán y tế: Chẩn đoán bệnh, dự đoán nguy cơ mắc bệnh, v.v.
2. Phân loại: Phân loại email rác, phân loại khách hàng tiềm năng, phân loại ảnh, v.v.
3. Hồi quy: Dự đoán giá nhà, dự đoán doanh thu bán hàng, dự đoán xu hướng thị trường

- Ưu điểm

1. Chống lại việc overfitting bởi vì nó kết hợp nhiều máy học yếu
2. Có thể xử lý cả dữ liệu phân loại và dữ liệu số.
3. Cung cấp các ước lượng về mức độ quan trọng của đặc trưng, có thể hữu ích cho việc lựa chọn đặc trưng.
4. Tương đối không nhạy cảm với việc chọn các siêu tham số.

- Nhược điểm

1. Có thể tốn nhiều tài nguyên tính toán, đặc biệt là đối với các tập dữ liệu lớn.

2. Có thể không hoạt động tốt trên các tập dữ liệu mất cân bằng.
3. Là một mô hình hộp đen, làm cho việc giải thích khó

3.2 Xây dựng thuật toán Random Forest

Class Node:

```
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, *, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf_node(self):
        return self.value is not None
```

feature: Thuộc tính (đặc trưng) được sử dụng để chia dữ liệu tại nút này.

threshold: Ngưỡng giá trị của feature để quyết định rẽ nhánh.

left: Nút con bên trái

right: Nút con bên phải

value: Giá trị dự đoán nếu nút này là một nút lá

is_leaf_node(self): Phương thức này kiểm tra xem một nút có phải là nút lá hay không.

Class Decision Tree:

Hàm init:

```
def __init__(self, min_samples_split=2, max_depth=100, n_features=None):
    self.min_samples_split = min_samples_split
    self.max_depth = max_depth
    self.n_features = n_features
    self.root = None
```

min_samples_split: Số lượng mẫu tối thiểu cần có để một nút có thể được chia tách.

max_depth: Độ sâu tối đa của cây.

n_features: Số lượng đặc trưng sẽ được xem xét khi tìm cách chia tốt nhất tại mỗi nút.
Nếu None, tất cả đặc trưng sẽ được dùng.

self.root: Lưu trữ nút gốc của cây, ban đầu là None.

Hàm fit:

```

def fit(self, X, y):
    self.n_features = X.shape[1] if not self.n_features else min(X.shape[1], self.n_features)
    self.root = self._grow_tree(X, y)

```

X: Ma trận đặc trưng

y: Vector nhãn

Gọi _grow_tree để bắt đầu xây dựng cây và gán kết quả cho self.root.

Hàm growtree:

```

def _grow_tree(self, X, y, depth=0):
    n_samples, n_feats = X.shape
    n_labels = len(np.unique(y))

    if (depth >= self.max_depth
        or n_labels == 1
        or n_samples < self.min_samples_split):
        leaf_value = self._most_common_label(y)
        return Node(value=leaf_value)

    feat_idxs = np.random.choice(n_feats, self.n_features, replace=False)

    best_feature, best_thresh = self._best_split(X, y, feat_idxs)
    if best_feature is None:
        leaf_value = self._most_common_label(y)
        return Node(value=leaf_value)

    left_idxs, right_idxs = self._split(X[:, best_feature], best_thresh)

    if len(left_idxs) == 0 or len(right_idxs) == 0:
        leaf_value = self._most_common_label(y)
        return Node(value=leaf_value)

    left = self._grow_tree(X[left_idxs, :], y[left_idxs], depth + 1)
    right = self._grow_tree(X[right_idxs, :], y[right_idxs], depth + 1)
    return Node(best_feature, best_thresh, left, right)

```

Là hàm để quy xây dựng cây

Điều kiện dừng

- Đã đạt max_depth.
- Tất cả các mẫu trong nút hiện tại thuộc cùng một lớp (`n_labels == 1`).
- Số lượng mẫu `n_samples` nhỏ hơn `min_samples_split`.
- Nếu một trong các điều kiện trên đúng, tạo một nút lá

Hàm _best_split: Tìm đặc trưng và ngưỡng chia tốt nhất trong số các đặc trưng được cung cấp

```
def _best_split(self, X, y, feat_idxs):
    best_gain = -1
    split_idx, split_thresh = None, None

    for feat_idx in feat_idxs:
        X_column = X[:, feat_idx]
        thresholds = np.unique(X_column)

        for thr in thresholds:
            gain = self._information_gain(y, X_column, thr)

            if gain > best_gain:
                best_gain = gain
                split_idx = feat_idx
                split_thresh = thr

    if best_gain == -1:
        return None, None

    return split_idx, split_thresh
```

Lặp qua từng feat_idx trong feat_idxs.

Với mỗi đặc trưng, lặp qua tất cả các giá trị thresholds để dùng làm ngưỡng tiềm năng.

Tính gain = self._information_gain(y, X_column, thr) cho mỗi cặp (đặc trưng, ngưỡng).

Lưu lại đặc trưng và ngưỡng nào cho best_gain

Nếu không có cách chia nào cải thiện trả về None, None.

Trả về split_idx và split_thresh

Hàm _information_gain:

```

def _information_gain(self, y, X_column, threshold):
    parent_entropy = self._entropy(y)

    left_idxs, right_idxs = self._split(X_column, threshold)

    if len(left_idxs) == 0 or len(right_idxs) == 0:
        return 0

    n = len(y)
    n_l, n_r = len(left_idxs), len(right_idxs)
    e_l, e_r = self._entropy(y[left_idxs]), self._entropy(y[right_idxs])
    child_entropy = (n_l / n) * e_l + (n_r / n) * e_r

    ig = parent_entropy - child_entropy
    return ig

```

Tính toán lượng thông tin thu được khi chia dữ liệu dựa trên một cột đặc trưng và một ngưỡng

Hàm _split:

```

def _split(self, X_column, split_thresh):
    left_idxs = np.argwhere(X_column <= split_thresh).flatten()
    right_idxs = np.argwhere(X_column > split_thresh).flatten()
    return left_idxs, right_idxs

```

Chia một cột dữ liệu thành hai nhóm chỉ số dựa trên ngưỡng

Hàm _entropy:

```

def _entropy(self, y):
    if len(y) == 0:
        return 0

    hist = np.bincount(y.astype(int))
    ps = hist / len(y)
    ps = ps[ps > 0]
    if len(ps) == 0:
        return 0
    return -np.sum(ps * np.log2(ps))

```

Tính entropy của một tập nhãn y.

Hàm _most_common_label:

```

def _most_common_label(self, y):
    if len(y) == 0:
        return None
    counter = Counter(y)
    if not counter:
        return None
    most_common = counter.most_common(1)[0][0]
    return most_common

```

Tìm nhãn xuất hiện nhiều nhất trong tập y. Đây là giá trị được gán cho nút lá:

- Nếu y rỗng, trả về None.
- Sử dụng collections.Counter để đếm tần suất các nhãn và trả về nhãn có tần suất cao nhất.

Hàm predict: Dự đoán nhãn cho một hoặc nhiều mẫu mới X:

```

def predict(self, X):
    if X.ndim == 1:
        X_processed = X.reshape(1, -1)
    else:
        X_processed = X
    return np.array([self._traverse_tree(x, self.root) for x in X_processed])

```

Xử lý X để đảm bảo nó là mảng 2 chiều.

Với mỗi mẫu x trong X_processed, gọi _traverse_tree(x, self.root) để lấy dự đoán.

Trả về một mảng NumPy chứa các dự đoán.

Hàm _traverse_tree: Duyệt cây từ một nút node để dự đoán cho một mẫu x

```

def _traverse_tree(self, x, node):
    if node.is_leaf_node():
        return node.value

    if x[node.feature] <= node.threshold:
        return self._traverse_tree(x, node.left)
    return self._traverse_tree(x, node.right)

```

Nếu node là nút lá (node.is_leaf_node()), trả về node.value.

Nếu giá trị đặc trưng x[node.feature] của mẫu nhỏ hơn hoặc bằng node.threshold, đi tiếp xuống cây con bên trái (node.left).

Ngược lại, đi tiếp xuống cây con bên phải (node.right).

Lặp lại cho tới khi tìm được nút lá

Class Random Forest

-Hàm init

```
def __init__(self, n_trees=10, max_depth=10, min_samples_split=2, n_features=None):
    self.n_trees = n_trees
    self.max_depth = max_depth
    self.min_samples_split = min_samples_split
    self.n_features = n_features
    self.trees = []
```

- n_trees: Số lượng cây quyết định sẽ được tạo trong rừng.
- max_depth: Độ sâu tối đa cho mỗi cây quyết định
- min_samples_split: Số lượng mẫu tối thiểu để chia một nút
- n_features: Số lượng đặc trưng sẽ được xem xét ngẫu nhiên tại mỗi lần chia nút trong mỗi cây. Nếu None, nó sẽ được tính mặc định trong hàm fit.
- self.trees: Một danh sách rỗng để lưu trữ các cây quyết định sau khi train

Hàm fit: Train model Random Forest

```
def fit(self, X, y):
    self.trees = []

    num_total_features = X.shape[1]
    if self.n_features is None:
        self.n_features_for_tree = int(np.sqrt(num_total_features))
    else:
        self.n_features_for_tree = min(num_total_features, self.n_features)

    for _ in range(self.n_trees):
        tree = DecisionTree(max_depth=self.max_depth,
                             min_samples_split=self.min_samples_split,
                             n_features=self.n_features_for_tree)

        X_sample, y_sample = self._bootstrap_samples(X, y)
        tree.fit(X_sample, y_sample)
        self.trees.append(tree)
```

X : Ma trận dữ liệu huấn luyện, trong đó mỗi hàng là một mẫu và mỗi cột là một đặc trưng.

y : Vector nhãn tương ứng với các mẫu trong X.

Đầu ra: Hàm này huấn luyện mô hình bằng cách tạo và lưu trữ các cây quyết định đã huấn luyện vào self.trees.

Hàm _bootstrap_samples: Tạo một tập dữ liệu bootstrap từ X và y

```
def _bootstrap_samples(self, X, y):
    n_samples = X.shape[0]
    idxs = np.random.choice(n_samples, n_samples, replace=True)
    return X[idxs], y[idxs]
```

n_samples = X.shape[0]: Lấy số lượng mẫu trong tập dữ liệu gốc.

idxs = np.random.choice(n_samples, n_samples, replace=True): Tạo ra một mảng các chỉ số ngẫu nhiên từ 0 đến n_samples - 1. replace=True đảm bảo rằng các chỉ số có thể được chọn nhiều lần. Kích thước của mảng chỉ số này bằng với n_samples.

return X[idxs], y[idxs]: Trả về các mẫu và nhãn tương ứng với các chỉ số đã chọn.

Hàm predict

```
def predict(self, X):
    if X.ndim == 1:
        X_processed = X.reshape(1, -1)
    else:
        X_processed = X

    tree_predictions = np.array([tree.predict(X_processed) for tree in self.trees])

    if self.n_trees == 1:
        return tree_predictions.flatten()

    try:
        valid_predictions = []
        for i in range(tree_predictions.shape[1]):
            sample_preds = tree_predictions[:, i]
            cleaned_sample_preds = [p for p in sample_preds if p is not None]
            if not cleaned_sample_preds:
                valid_predictions.append(None)
            else:
                counter = Counter(cleaned_sample_preds)
                valid_predictions.append(counter.most_common(1)[0][0])
        final_predictions = np.array(valid_predictions)

    except ImportError:
        predictions_per_sample = np.swapaxes(tree_predictions, 0, 1)
        final_predictions = np.array([Counter(row_preds[row_preds != np.array(None)]).most_common(1)[0][0]
                                     if len(row_preds[row_preds != np.array(None)]) > 0 else None
                                     for row_preds in predictions_per_sample])

    return final_predictions.flatten()
```

Đầu vào: X (mảng NumPy hoặc tương tự, 1D hoặc 2D): Dữ liệu mới cần dự đoán. Nếu là 1D, nó sẽ được coi là một mẫu duy nhất.

Đầu ra: final_predictions (mảng NumPy, 1D): Một mảng chứa các nhãn dự đoán cho mỗi mẫu trong X.

3.3 So sánh và chứng minh thuật toán đã xây dựng là đúng với Random Forest trong Scikit-Learn

```
Distribution of classes in combined dataset:  
Class 0: 392 samples (33.3%)  
Class 1: 404 samples (34.3%)  
Class 2: 382 samples (32.4%)  
  
[Manual RandomForest Version]  
Running cross-validation for manual RandomForest version...  
ROOT: C:\Users\ASUS\Desktop\Techclub\data_mining  
- Build time (CV): 38.7732 seconds  
- Cross-validation scores: [0.82188295 0.82697201 0.80867347]  
- Mean accuracy: 81.92%  
- Std accuracy: 0.77%  
  
[Scikit-learn RandomForest Version]  
Running cross-validation for scikit-learn RandomForest version...  
- Build time (CV): 0.2043 seconds  
- Cross-validation scores: [0.82188295 0.84223919 0.83928571]  
- Mean accuracy: 83.45%  
- Std accuracy: 0.90%
```

[Detailed Comparison on Test Set]

Manual RandomForest Implementation:

Fit time: 43.9065s

Predict time: 0.0090s

	precision	recall	f1-score	support
0	0.92	0.86	0.89	79
1	0.82	0.89	0.85	81
2	0.84	0.82	0.83	76
accuracy			0.86	236
macro avg	0.86	0.86	0.86	236
weighted avg	0.86	0.86	0.86	236

Scikit-learn RandomForest Implementation:

Fit time: 0.0427s

Predict time: 0.0147s

	precision	recall	f1-score	support
0	0.91	0.86	0.88	79
1	0.85	0.89	0.87	81
2	0.80	0.80	0.80	76
accuracy			0.85	236
macro avg	0.85	0.85	0.85	236
weighted avg	0.85	0.85	0.85	236

Phiên bản	CV Accuracy	Test Accuracy	Fit time	Lưu ý
Manual Tabnet	82.68%	85%	44.6s	Kết quả tốt nhưng rất chậm
Pytorch	83.45%	85%	0.044s	Kết quả tương đương, rất nhanh

Kết luận:

Cả hai model cho kết quả giống nhau về độ chính xác. Tuy nhiên, Scikit-learn hiệu quả vượt trội về tốc độ (gấp hơn 1000 lần nhanh hơn).

-> Thuật toán tự phát triển là đúng so với thư viện

4. Thuật toán TabNet

4.1 Giới thiệu

- TabNet là một kiến trúc học sâu được thiết kế đặc biệt cho dữ liệu dạng bảng.
- Luồng hoạt động:
 1. **Feature Selection:** Tại mỗi bước quyết định, một thành phần gọi là "Attentive Transformer" sẽ chọn ra một tập hợp con các đặc trưng quan trọng từ dữ liệu đầu vào.
 2. **Feature Processing:** Các đặc trưng được chọn sau đó được xử lý bởi một "Feature Transformer" để tạo ra các biểu diễn
 3. **Tổng hợp quyết định:** Kết quả từ mỗi bước quyết định được tổng hợp lại để đưa ra dự đoán cuối cùng.
 4. **Mask Update:** Mặt nạ lựa chọn đặc trưng được cập nhật để các bước tiếp theo tập trung vào các đặc trưng khác hoặc các khía cạnh khác của các đặc trưng đã sử dụng.
- TabNet được ứng dụng rộng rãi trong nhiều lĩnh vực yêu cầu phân tích dữ liệu dạng bảng

4.2 Xây dựng thuật toán TabNet

-**Ứng dụng:**

Class GLU_Block - Gate Linear Unit: tăng biểu diễn phi tuyến, chọn lọc thông tin mạnh mẽ hơn.

Có thể hiểu theo ý sau: GLU_Block = Fully Connected + BatchNorm + Dropout + Cơ chế Gating.

-Hàm init:

```
def __init__(self, input_dim, output_dim, dropout_rate=0.2):  
    super().__init__()  
    self.fc = nn.Linear(input_dim, output_dim * 2)  
    self.bn = nn.BatchNorm1d(output_dim * 2)  
    self.dropout = nn.Dropout(dropout_rate)  
    self.output_dim = output_dim
```

Tạo một fully connected layer, đầu ra gấp đôi để chia thành 2 phần: values và gate.

Chuẩn hóa batch để ổn định huấn luyện.

Giảm overfitting.

-Hàm forward:

```
def forward(self, x):
    x_dropped = self.dropout(x)
    x_lin = self.fc(x_dropped)
    x_bn = self.bn(x_lin)
    values, gate = torch.chunk(x_bn, 2, dim=-1)
    return values * torch.sigmoid(gate)
```

Áp dụng dropout cho input.

Đưa qua linear + batch norm.

Tách làm 2: values và gate mỗi cái có size output_dim

Áp dụng sigmoid lên gate rồi nhân element-wise với values \Rightarrow cho phép thông tin đi qua có kiểm soát.

Class FeatureTransformer: Giúp mô hình học biểu diễn đặc trưng sâu và phi tuyến

-Hàm init:

```
def __init__(self, input_dim, output_feature_dim, n_glu_layers=2, dropout=0.2):
    super().__init__()
    layers = []
    current_dim = input_dim
    for _ in range(n_glu_layers):
        layers.append(GLU_Block(current_dim, output_feature_dim, dropout_rate=dropout))
        current_dim = output_feature_dim

    self.net = nn.Sequential(*layers)
```

Tạo nhiều lớp GLU_Block liên tiếp, để:

- Học các biểu diễn đặc trưng sâu hơn
- Mỗi lớp xử lý và lọc thông tin từ đầu vào \rightarrow đầu ra

Sau đó gộp các GLU_Block lại thành 1 mạng tuần tự.

-Hàm forward:

```
def forward(self, x):
    return self.net(x)
```

Truyền input x qua chuỗi các GLU_Block để trích xuất đặc trưng đã chọn lọc.

Class AttentiveTransformer: lớp tính attention mask

-Hàm init:

```

def __init__(self, input_dim_attention, num_output_features):
    super().__init__()
    self.fc = nn.Linear(input_dim_attention, num_output_features)
    self.bn = nn.BatchNorm1d(num_output_features)

```

Biến đổi đầu vào thành số lượng đặc trưng cần chọn

Sau đó ổn định giá trị attention để train hiệu quả hơn.

-Hàm forward:

```

def forward(self, x_processed_for_attention, prior):
    x = self.fc(x_processed_for_attention)
    x = self.bn(x)
    x = x * prior
    return F.softmax(x, dim=1)

```

Biến đổi đầu vào attention

Nhân với prior (ưu tiên chọn những đặc trưng ít được chọn trước đó)

Chuẩn hóa thành xác suất chọn từng feature.

Class TabNet:

-Hàm init:

```

def __init__(self, input_dim, output_dim, n_d=64, n_steps=5, gamma=1.5, dropout=0.2, n_glu_shared=2, n_glu_decision=2):
    super().__init__()
    self.n_d = n_d
    self.n_steps = n_steps
    self.gamma = gamma
    self.output_dim = output_dim

    self.shared_feat_trans = FeatureTransformer(input_dim, n_d, n_glu_layers=n_glu_shared, dropout=dropout)

    self.att_transformers = nn.ModuleList([
        AttentiveTransformer(n_d, input_dim) for _ in range(n_steps)
    ])

    self.decision_steps = nn.ModuleList([
        FeatureTransformer(input_dim, n_d, n_glu_layers=n_glu_decision, dropout=dropout) for _ in range(n_steps)
    ])

    self.final_fc = nn.Linear(n_d, self.output_dim)

```

Các tham số:

n_d: kích thước đầu ra mỗi bước quyết định.
n_steps: số bước lặp để chọn đặc trưng.
gamma: hệ số kiểm soát "penalty" khi chọn đặc trưng lặp lại.
n_glu_shared: số lớp GLU dùng chung cho tất cả bước.
n_glu_decision: số lớp GLU cho mỗi bước quyết định.

Luồng hoạt động:

1. Biến đổi đầu vào ban đầu – dùng dùng chung cho tất cả bước.
2. Tính attention mask cho từng bước
3. Dùng các đặc trưng được chọn để sinh output từng bước.

-Hàm forward:

```
def forward(self, x):  
    B, D = x.shape  
    prior = torch.ones(B, D).to(x.device)  
    output_agg = torch.zeros(B, self.n_d).to(x.device)  
    mask_values = []  
  
    x_shared_processed = self.shared_feat_trans(x)  
  
    for step in range(self.n_steps):  
        mask = self.att_transformers[step](x_shared_processed, prior)  
        mask_values.append(mask)  
  
        prior = prior * (self.gamma - mask)  
  
        x_masked = x * mask  
  
        step_out = self.decision_steps[step](x_masked)  
  
        output_agg += step_out  
    final_output_logits = self.final_fc(output_agg)  
    return final_output_logits, mask_values
```

Lặp qua n_steps:

mask: attention chọn đặc trưng ở bước step
prior: giảm ưu tiên với đặc trưng đã được chọn quá nhiều
x_masked: input sau khi chọn đặc trưng

step_out: kết quả học từ đặc trưng đó
output_agg: cộng dồn kết quả

final_output_logits = self.final_fc(output_agg) là kết quả cuối

-Hàm sparse_loss: Đây là hàm regularization cho attention mask trong TabNet giúp chọn ít đặc trưng nhất có thể

Ý tưởng chính:

1. Nếu mask chọn ít đặc trưng rõ ràng (gần 1 hoặc 0), entropy sẽ thấp → loss thấp.
2. Nếu chọn đều (nhiều feature có xác suất ~ đều), entropy cao → loss cao.

```
def sparse_loss(mask_values):  
  
    loss = 0.0  
    for mask in mask_values:  
        entropy_per_sample = -torch.sum(mask * torch.log(mask + 1e-15), dim=1)  
        loss += torch.mean(entropy_per_sample)  
    return loss
```

Với mỗi mask:

- Tính entropy cho từng sample: $-\sum p \cdot \log(p)$
- Trung bình các entropy đó → tăng dần vào loss

4.3 So sánh và chứng minh thuật toán đã xây dựng là đúng với TabNet trong pytorch-tabnet

Test Data:

Phiên bản	Thời gian	CV scores	Mean Accuracy	Std Accuracy
Manual TabNet Implementation	4.0871	[0.92111959, 0.96692112, 0.95663265]	94.82%	1.96%
Library	4.9476	[0.92111959,	90.07%	2.04%

pytorch-tabnet		0.87277354, 0.90816327]		
----------------	--	----------------------------	--	--

Detailed Comparison on Test Set

```
- Build and CV time: 4.9476 seconds
- Cross-validation scores: [0.92111959 0.87277354 0.90816327]
- Mean accuracy: 90.07%
- Std accuracy: 2.04%
```

[Detailed Comparison on Test Set]

Training Manual TabNet on split...

Manual TabNet Implementation (Test Set):

	precision	recall	f1-score	support
0	0.92	0.99	0.95	98
1	0.97	0.91	0.94	102
2	1.00	0.98	0.99	95
accuracy			0.96	295
macro avg	0.96	0.96	0.96	295
weighted avg	0.96	0.96	0.96	295

Training Library TabNet on split...

```
Stop training because you reached max_epochs = 30 with best_epoch = 27 and best_val_0 accuracy = 0.90816327
C:\Users\ASUS\Desktop\Techclub\data_mining\venv\Lib\site-packages\pytorch_tabnet\callbacks.py:105: UserWarning: Early stopping did not reach max_epochs = 30. Best epoch = 27 and best_val_0 accuracy = 0.90816327
  warnings.warn(wrn_msg)
```

Library pytorch-tabnet Implementation (Test Set):

	precision	recall	f1-score	support
0	0.94	0.93	0.93	98
1	0.95	0.95	0.95	102
2	0.98	0.99	0.98	95
accuracy			0.96	295
macro avg	0.96	0.96	0.96	295
weighted avg	0.96	0.96	0.96	295

Real data:

```
[Detailed comparison on Test Set]

Training Manual TabNet on split...
Manual TabNet Implementation (Test Set):
      precision    recall   f1-score   support
          0         0.93     0.96     0.95     9981
          1         0.62     0.49     0.55     1322
   accuracy                           0.90     11303
macro avg       0.78     0.72     0.75     11303
weighted avg    0.90     0.90     0.90     11303

Training Library TabNet on split...
Early stopping occurred at epoch 32 with best_epoch = 17 and best
C:\Users\ASUS\Desktop\Techclub\data_mining\venv\Lib\site-packages
warnings.warn(wrn_msg)

Library pytorch-tabnet Implementation (Test Set):
      precision    recall   f1-score   support
          0         0.94     0.96     0.95     9981
          1         0.63     0.50     0.56     1322
   accuracy                           0.91     11303
macro avg       0.78     0.73     0.75     11303
weighted avg    0.90     0.91     0.90     11303
```

Kết luận:

Manual TabNet đang hoạt động tốt, đúng kỳ vọng.

-> Phiên bản manual được phát triển đúng

```
--- Training and Evaluating: Custom TabNet ---
```

```
Training time: 52.47 seconds
```

```
Prediction time: 0.02 seconds
```

```
Accuracy: 0.8482
```

```
Classification Report:
```

	precision	recall	f1-score	support
no	0.98	0.85	0.91	9981
yes	0.43	0.87	0.57	1322
accuracy			0.85	11303
macro avg	0.70	0.86	0.74	11303
weighted avg	0.92	0.85	0.87	11303

```
ROC AUC Score: 0.9288
```

5. Thuật toán Multilayer Perceptron (MLP)

5.1 Giới thiệu

Multilayer Perceptron (MLP) là một loại mạng nơ-ron nhân tạo phổ biến trong học máy, thuộc nhóm mô hình mạng nơ-ron sâu (Deep Learning). MLP bao gồm nhiều lớp neuron (lớp đầu vào, các lớp ẩn và lớp đầu ra), trong đó mỗi neuron thực hiện phép biến đổi tuyến tính và phi tuyến trên dữ liệu đầu vào. Mỗi lớp kết nối đầy đủ với lớp kế tiếp, cho phép mô hình học các đặc trưng phức tạp và mối quan hệ phi tuyến giữa các biến đầu vào. MLP thường được sử dụng để giải quyết các bài toán phân loại và hồi quy.

Ứng dụng:

- Nhận dạng hình ảnh, giọng nói, và xử lý ngôn ngữ tự nhiên.
- Dự đoán chuỗi thời gian như dự báo tài chính, dự báo thời tiết.
- Phân loại văn bản, dự đoán nhãn trong bài toán phân loại.
- Các bài toán hồi quy phức tạp trong kinh tế, y tế và kỹ thuật.

Ưu điểm:

- Có khả năng học các mối quan hệ phi tuyến phức tạp trong dữ liệu.
- Linh hoạt, có thể áp dụng cho nhiều loại bài toán khác nhau.
- Mô hình có thể mở rộng với nhiều lớp ẩn để tăng khả năng biểu diễn dữ liệu.
- Hỗ trợ các kỹ thuật tối ưu và regularization để tránh overfitting.

Nhược điểm:

- Cần lượng dữ liệu lớn và thời gian huấn luyện lâu hơn so với các mô hình đơn giản.
- Là mô hình “hộp đen”, khó giải thích trực tiếp các quyết định của mô hình.
- Dễ bị quá khớp nếu không được điều chỉnh đúng cách hoặc không có đủ dữ liệu.

Luồng hoạt động của MLP:

1. Nhận dữ liệu đầu vào: Mỗi mẫu dữ liệu được biểu diễn dưới dạng một vector đặc trưng và đưa vào lớp đầu vào của mạng.
2. Lan truyền tiến (Forward propagation): Dữ liệu được truyền qua từng lớp ẩn. Ở mỗi neuron, dữ liệu đầu vào được nhân với trọng số, cộng thêm bias, sau đó áp dụng hàm kích hoạt phi tuyến (ví dụ ReLU, sigmoid, tanh) để tạo ra đầu ra của neuron đó.
3. Tính toán lỗi: Kết quả đầu ra cuối cùng được so sánh với nhãn thực (trong bài toán có giám sát) bằng hàm mất mát (loss function) để tính toán sai số dự đoán.
4. Lan truyền ngược (Backpropagation): Sai số được truyền ngược từ đầu ra về các lớp trước, tính toán gradient của hàm mất mát theo các trọng số trong mạng.
5. Cập nhật trọng số: Sử dụng thuật toán tối ưu (như Gradient Descent hoặc Adam) để điều chỉnh trọng số mạng nhằm giảm lỗi dự đoán.
6. Lặp lại quá trình: Quá trình trên được lặp lại qua nhiều epoch cho đến khi mô hình hội tụ hoặc đạt được hiệu suất mong muốn.

5.2 Xây dựng thuật toán MLP

Class MLP:

Hàm init:

input_size: kích thước của lớp đầu vào neural

hidden_size: kích thước của lớp ẩn

output_size: kích thước lớp đầu ra

learning_rate: tốc độ học

Khởi tạo luôn cái trọng số và độ lệch cho mạng neuron, các trọng số được khởi tạo ngẫu nhiên giúp tăng hiệu quả huấn luyện

Khởi tạo weight theo cách này giúp giảm thiểu vấn đề vanishing gradient hoặc exploding gradient, và cũng phù hợp cho các hàm kích hoạt như ReLU hoặc Sigmoid

vì đảm bảo các trọng số không quá lớn hoặc quá nhỏ

```
def __init__(self, input_size, hidden_size, output_size, learning_rate=0.01):
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size
    self.learning_rate = learning_rate

    self.weights1 = np.random.randn(self.input_size, self.hidden_size) * np.sqrt(2.0 / (input_size + hidden_size))
    self.weights2 = np.random.randn(self.hidden_size, self.output_size) * np.sqrt(2.0 / (hidden_size + output_size))

    self.bias1 = np.zeros((1, self.hidden_size))
    self.bias2 = np.zeros((1, self.output_size))
```

Hàm sigmoid và sigmoid_derivative:

```
def sigmoid(self, z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(self, z):
    s = self.sigmoid(z)
    return s * (1 - s)
```

Chọn hàm sigmoid thay vì softmax vì kết quả của bài toán cuối cùng chỉ cho ra 2 kết quả là yes hoặc no

Hàm sigmoid_derivative có tác dụng tính đạo hàm của hàm sigmoid, giúp điều chỉnh trọng số bằng cách tính gradient

Hàm fit:

```

def fit(self, X, y, epochs=1000):
    # Tính class weights
    n_samples = len(y)
    n_class_1 = np.sum(y)
    n_class_0 = n_samples - n_class_1
    class_weight = {0: 1.0, 1: n_class_0/n_class_1}

    for epoch in range(epochs):
        # feedforward
        layer1 = X.dot(self.weights1) + self.bias1
        activation1 = self.sigmoid(layer1)
        layer2 = activation1.dot(self.weights2) + self.bias2
        activation2 = self.sigmoid(layer2)

        # backpropagation với class weights
        error = activation2 - y
        # Thêm class weights vào gradient
        sample_weights = np.array([class_weight[int(yi[0])] for yi in y])
        error = error * sample_weights.reshape(-1, 1)

        d_weights2 = activation1.T.dot(error * self.sigmoid_derivative(layer2))
        d_bias2 = np.sum(error * self.sigmoid_derivative(layer2), axis=0, keepdims=True)
        error_hidden = error.dot(self.weights2.T) * self.sigmoid_derivative(layer1)
        d_weights1 = X.T.dot(error_hidden)
        d_bias1 = np.sum(error_hidden, axis=0, keepdims=True)

        # update weights and biases
        self.weights2 -= self.learning_rate * d_weights2
        self.bias2 -= self.learning_rate * d_bias2
        self.weights1 -= self.learning_rate * d_weights1
        self.bias1 -= self.learning_rate * d_bias1

```

Có 2 bước trong hàm Fit:

Bước đầu là tính toán trọng số lớp trong đó:

n_samples: Tổng số mẫu dữ liệu

n_class_1: Số mẫu thuộc lớp 1 (tính bằng tổng các giá trị y vì y là 0 hoặc 1)

n_class_0: Số mẫu thuộc lớp 0 (tính bằng tổng số mẫu trừ đi n_class_1)

class_weight: Từ điển lưu trọng số của mỗi lớp

Bước 2 là vòng lặp huấn luyện theo epoch(số lần lặp qua dữ liệu):

Trong bước 2 có 3 ý chính:

- Thứ nhất là lan truyền tiến(feedforward):

Layer 1: Sử dụng trọng số và bias của lớp đầu tiên để tính đầu ra sau đó dùng activation 1 để kích hoạt hàm sigmoid để biến đổi đầu ra

Layer 2: Cũng giống như cách layer 1 hoạt động

- Thứ 2 là lan truyền ngược(backpropagation):

error có tác dụng để tính lỗi sau đó áp dụng trọng số mẫu để đảm bảo lỗi của lớp thiểu số được tính toán với mức độ ưu tiên cao hơn

Sau đó gradient cập nhật layer 2 và lan truyền lỗi về lớp ẩn

Cuối cùng là tính gradient cho lớp ẩn

- Thứ 3: Cập nhật trọng số và độ lệch để giảm thiểu lỗi cho lần học sau

Hàm predict:

```
def predict(self, x):  
    layer1 = x.dot(self.weights1) + self.bias1  
    activation1 = self.sigmoid(layer1)  
    layer2 = activation1.dot(self.weights2) + self.bias2  
    activation2 = self.sigmoid(layer2)  
    return (activation2 > 0.5).astype(int)
```

Trước tiên là tính tổng có trọng số (weighted sum) tại lớp ẩn và áp dụng hàm sigmoid để đưa tổng có trọng số về giá trị giữa 0 và 1

Tiếp đó, tính tổng có trọng số tại lớp đầu ra áp dụng hàm sigmoid để đưa giá trị về khoảng [0, 1], đại diện cho xác suất dự đoán

Cuối cùng là chuyển đổi xác suất thành nhãn (Trả về một mảng các nhãn dự đoán (0 hoặc 1) tương ứng với từng mẫu đầu vào.)

Hàm predict_proba:

```
def predict_proba(self, x):  
    layer1 = x.dot(self.weights1) + self.bias1  
    activation1 = self.sigmoid(layer1)  
    layer2 = activation1.dot(self.weights2) + self.bias2  
    return self.sigmoid(layer2) |
```

Giống như hàm predict nhưng kết quả trả về là một mảng chứa xác suất dự đoán cho từng mẫu, thay vì nhãn cụ thể

5.3 So sánh và chứng minh thuật toán đã xây dựng là đúng với MLP trong pytorch-mlp

```
Custom MLP Implementation (Test Set):
precision    recall    f1-score   support
          0       0.95      0.80      0.87     7952
          1       0.33      0.71      0.45     1091

   accuracy                           0.79      9043
  macro avg       0.64      0.76      0.66     9043
weighted avg       0.88      0.79      0.82     9043

accuracy: 0.79
AUC: 0.83
Build & Train Time: 94.76 seconds
Cross-validation scores: [0.88526403 0.57036218 0.71067183 0.88193004 0.7153325 ]
Mean accuracy: 0.7527
Std accuracy: 0.1189
CV Time: 496.89 seconds
```

```
Torch MLP Implementation (Test Set):
precision    recall    f1-score   support
          0       0.92      0.97      0.94     7952
          1       0.62      0.37      0.47     1091

   accuracy                           0.90      9043
  macro avg       0.77      0.67      0.70     9043
weighted avg       0.88      0.90      0.89     9043

accuracy: 0.90
AUC: 0.90
Build & Train Time: 484.23 seconds
Cross-validation scores: [0.89216987 0.89681486 0.89407764]
Mean accuracy: 0.8944
Std accuracy: 0.0019
CV Time: 1003.29 seconds
```

Phiên bản	CV Accuracy	Test Accuracy	Lưu ý
MLP	75.26%	79%	Viết tay được, kết quả không được bằng pytorch

Pytorch	89.40%	90%	Kết quả ổn định hơn, nhưng thời gian chạy lâu
---------	--------	-----	---

Kết luận:

Model cho kết quả về độ chính xác cao hơn viết tay.

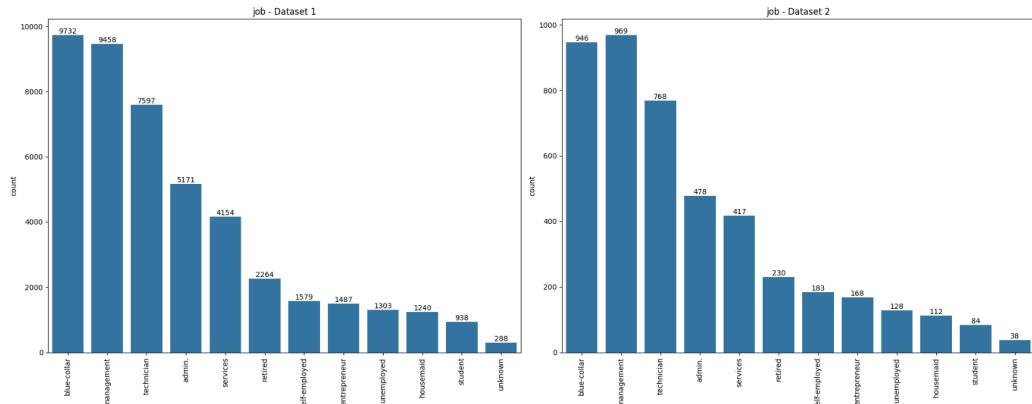
-> Thuật toán tự phát triển là chưa hoàn toàn đúng so với thư viện

II. Khảo sát và đặt ra giả thiết trên bộ dữ liệu

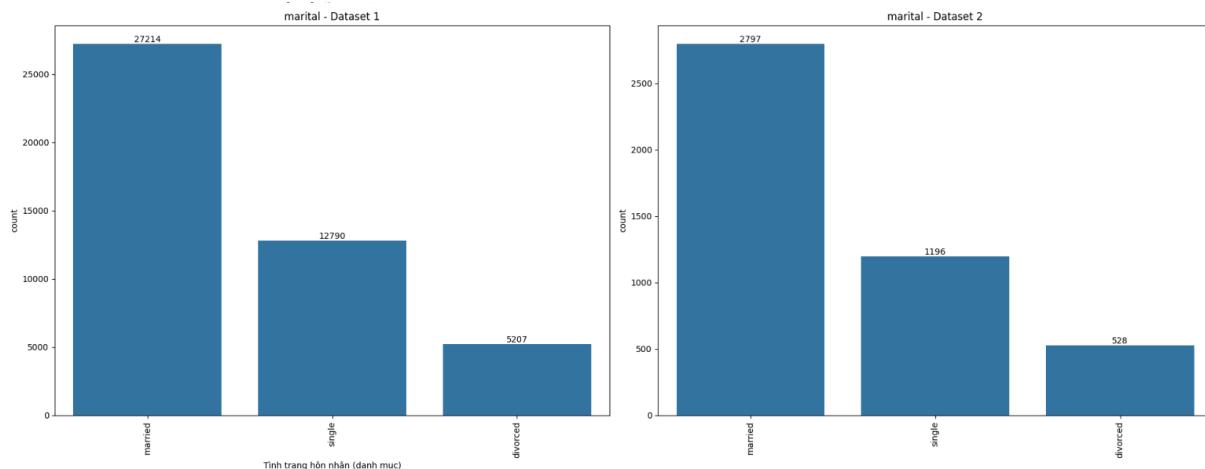
1. Tìm hiểu investigate, muôn kiểm tra/khảo sát điều gì từ bộ dữ liệu (Investigating and Defining What to Examine)

- Bộ dữ liệu sử dụng: **Bank Marketing** - *Dữ liệu có liên quan đến các chiến dịch tiếp thị trực tiếp (cuộc gọi điện thoại) của một tổ chức ngân hàng Bồ Đào Nha. Mục tiêu phân loại là dự đoán nếu khách hàng sẽ đăng ký một khoản tiền gửi có kỳ hạn (biến y).*
- Thuộc tính
 - + "age": "Tuổi của khách hàng (numeric)"
 - + "job": "Nghề nghiệp (category: student, manager, housemaid...)"
 - + "marital": "Tình trạng hôn nhân (category)"
 - + "education": "Trình độ học vấn (category)"
 - + "default": "Khách hàng có nợ xấu không? (yes/no)"
 - + "balance": "Số dư trung bình hàng năm (euro) (numeric)"
 - + "housing": "Khách hàng có vay mua nhà không? (yes/no)"
 - + "loan": "Khách hàng có vay tiêu dùng không? (yes/no)"
 - + "contact": "Loại phương thức liên hệ (unknown/telephone/cellular)"
 - + "day": "Ngày gọi gần nhất trong tháng (numeric)"
 - + "month": "Tháng gọi (category: jan, feb, ...)"
 - + "duration": "Thời lượng cuộc gọi (giây)"
 - + "campaign": "Số lần liên hệ trong chiến dịch hiện tại (bao gồm cả lần này)"
 - + "pdays": "Số ngày kể từ lần gọi trước trong chiến dịch trước (-1 nếu chưa gọi)"
 - + "previous": "Số lần liên hệ trước đó trong các chiến dịch khác"
 - + "poutcome": "Kết quả của chiến dịch trước (unknown/other/failure/success)"
 - + "y": "Khách hàng có đăng ký gửi tiết kiệm không? (yes/no)"

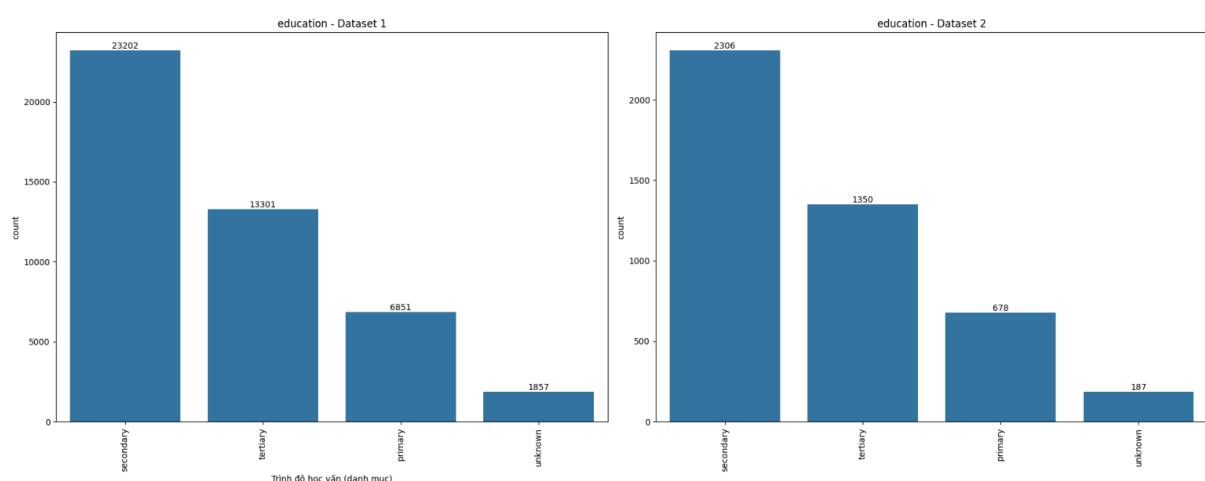
a. Data Exploration (Một số biểu đồ phân bố chính)



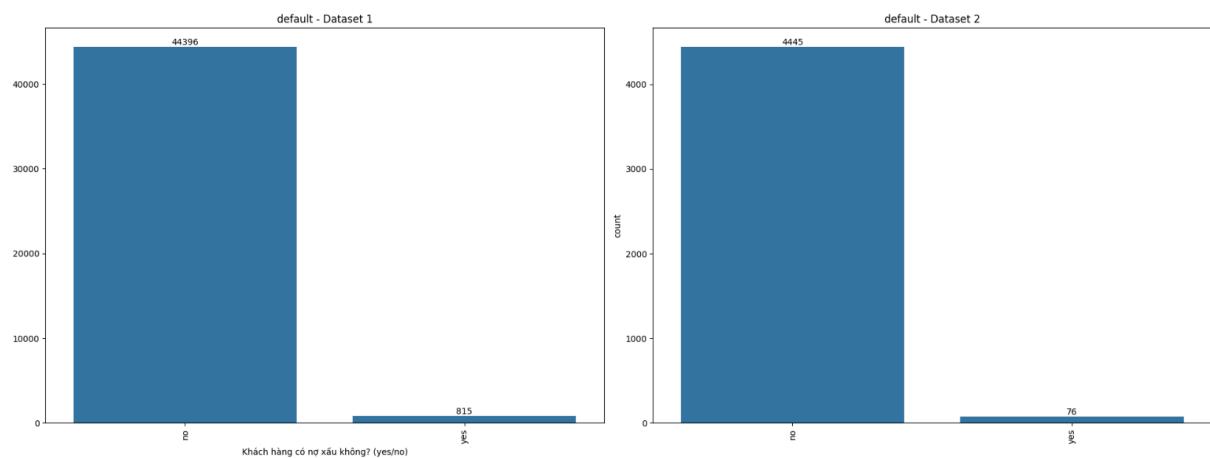
Job của Train và Test



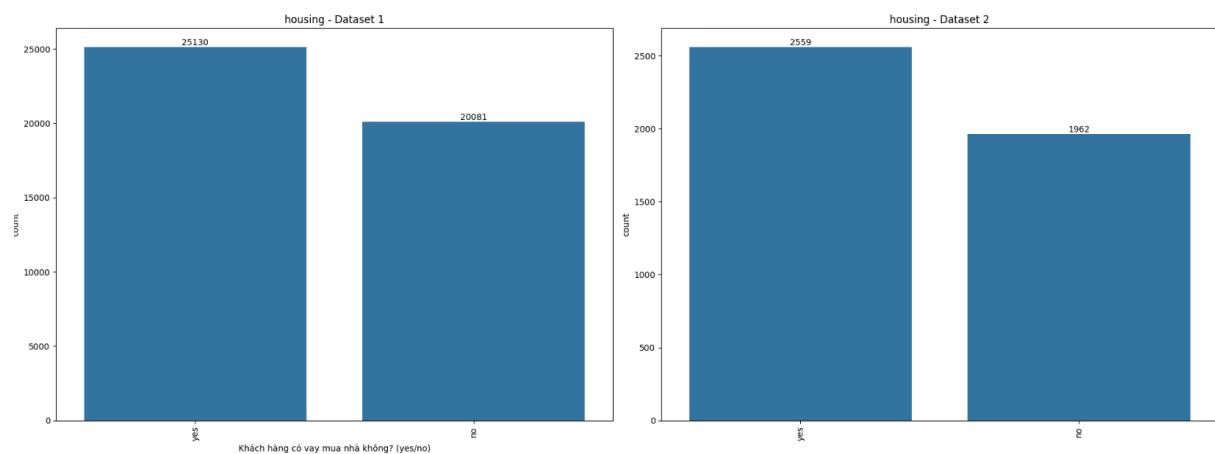
Tình trạng hôn nhân của Train và Test



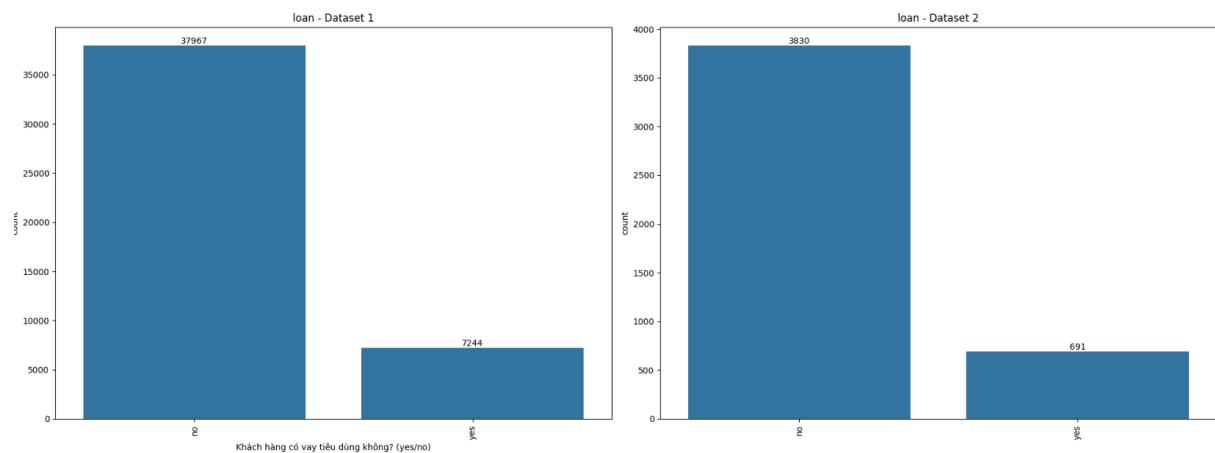
Tình trạng học vấn của Train và Test



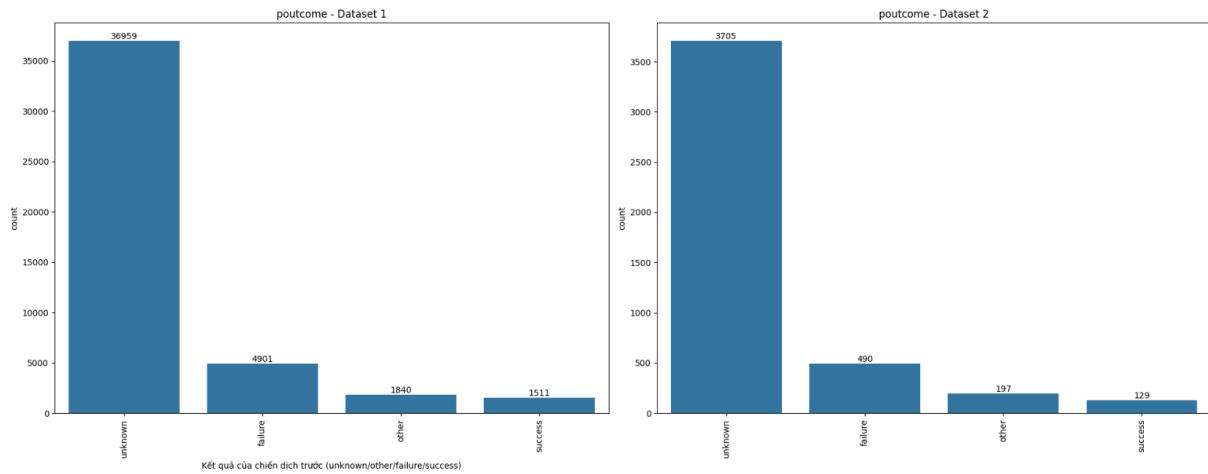
Tình trạng nợ xấu của khách hàng trong Train và Test



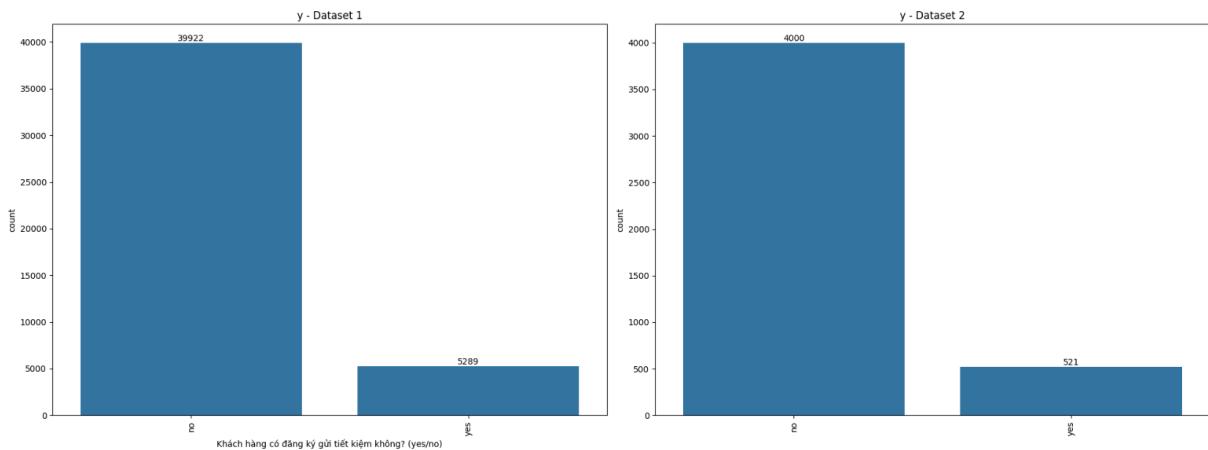
Tình trạng vay mua nhà của khách hàng



Tình trạng vay tiêu dùng của khách hàng



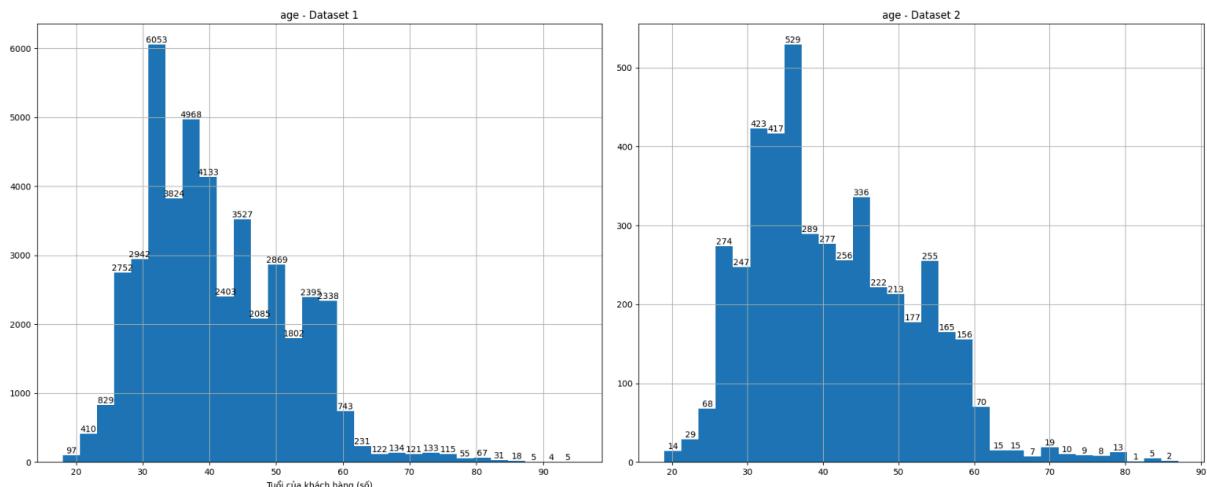
Trạng thái của chiến dịch trước



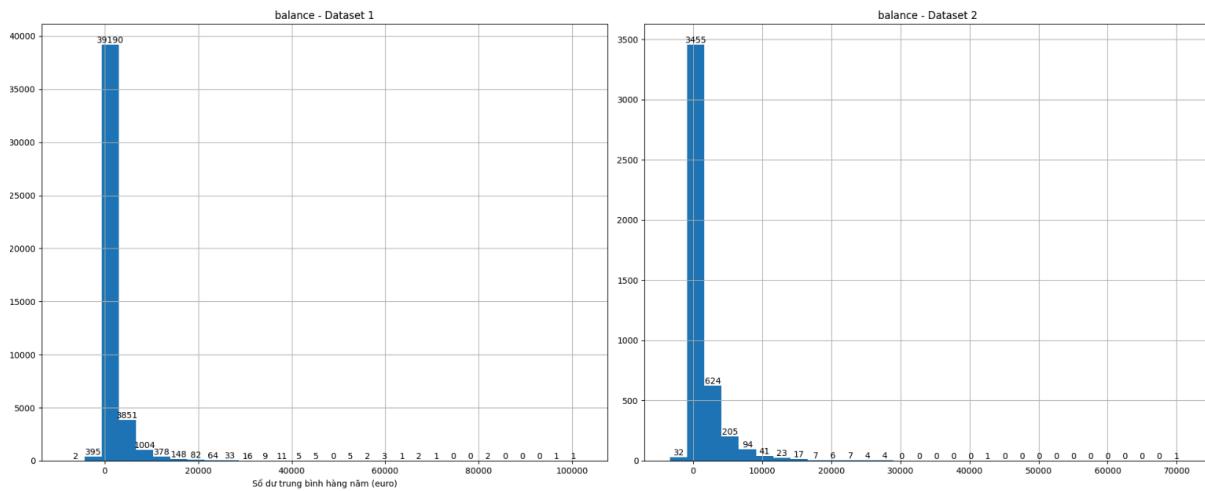
Khách hàng có đăng ký gửi tiết kiệm không?

- **Kết luận 1:**

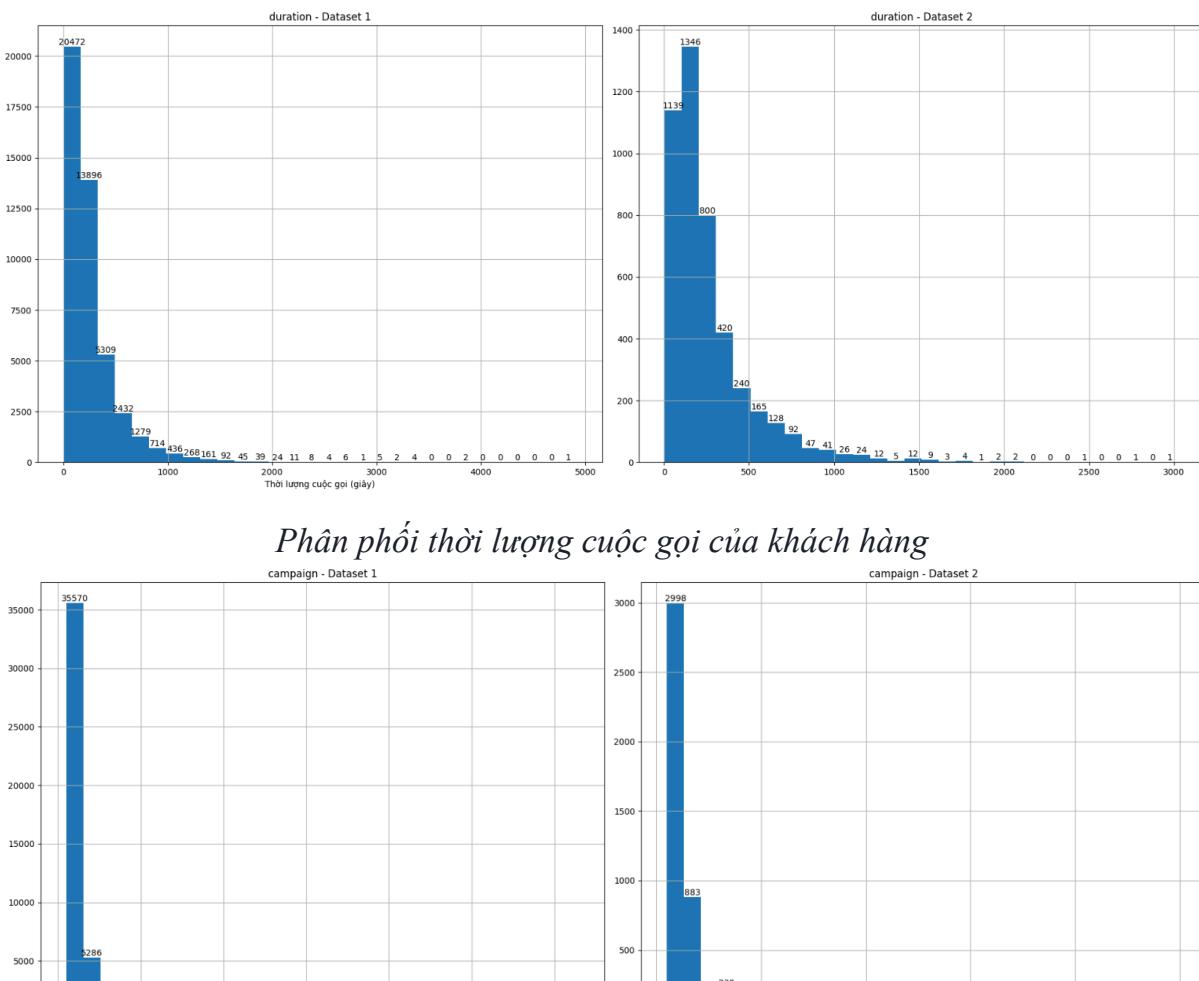
- + Các giá trị như “unknown” là các missing value, nhưng ta sẽ coi đó là một trường hợp của bản ghi dữ liệu
- + Dữ liệu nhãn phân phối không đồng đều
- + Tỉ lệ nhãn Yes/No TARGET phân bố không đều (mất cân bằng nhãn)

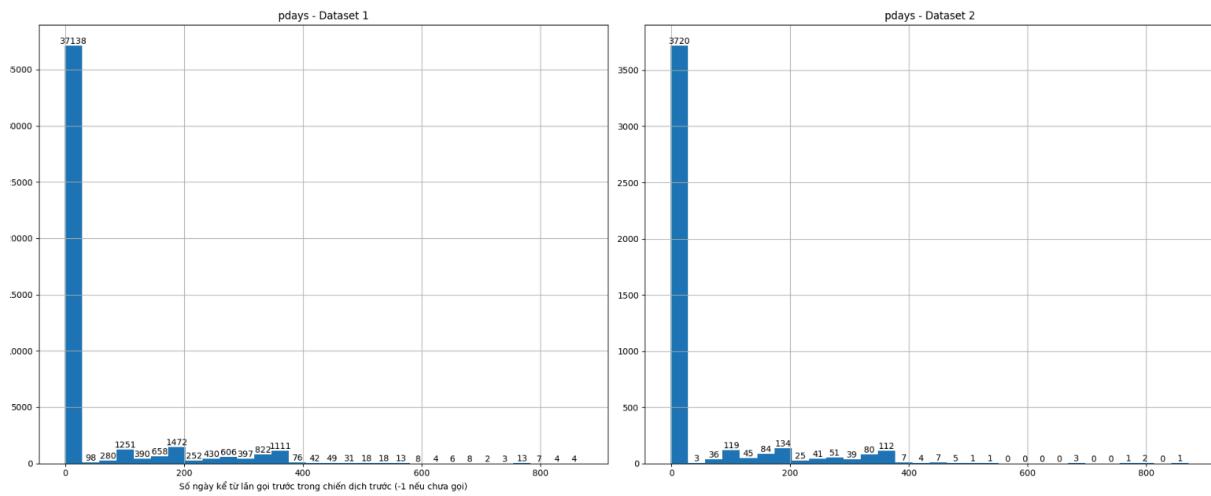


Phân phối tuổi của khách hàng

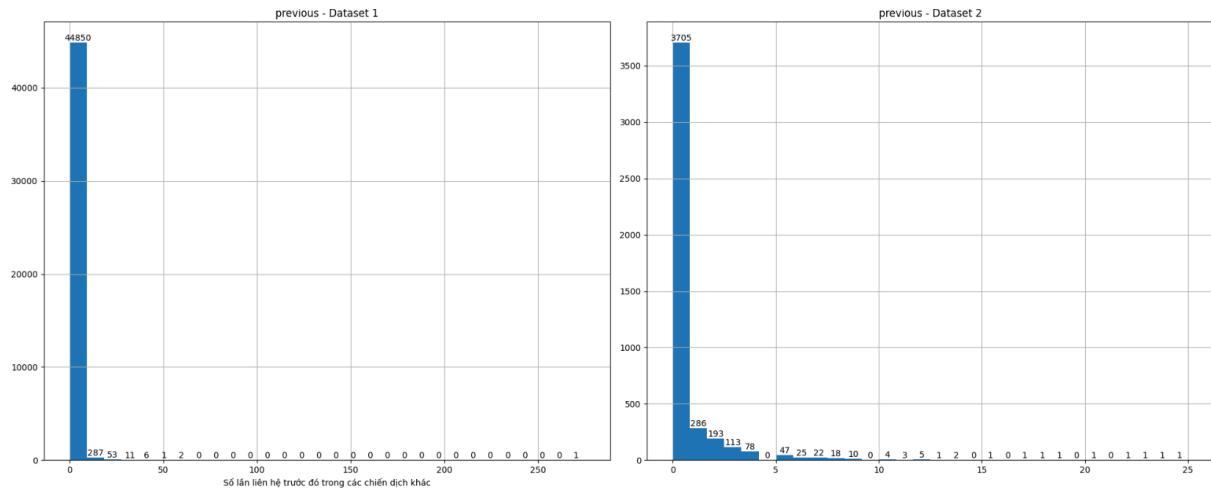


Phân phối số lần liên hệ trong chiến dịch hiện tại





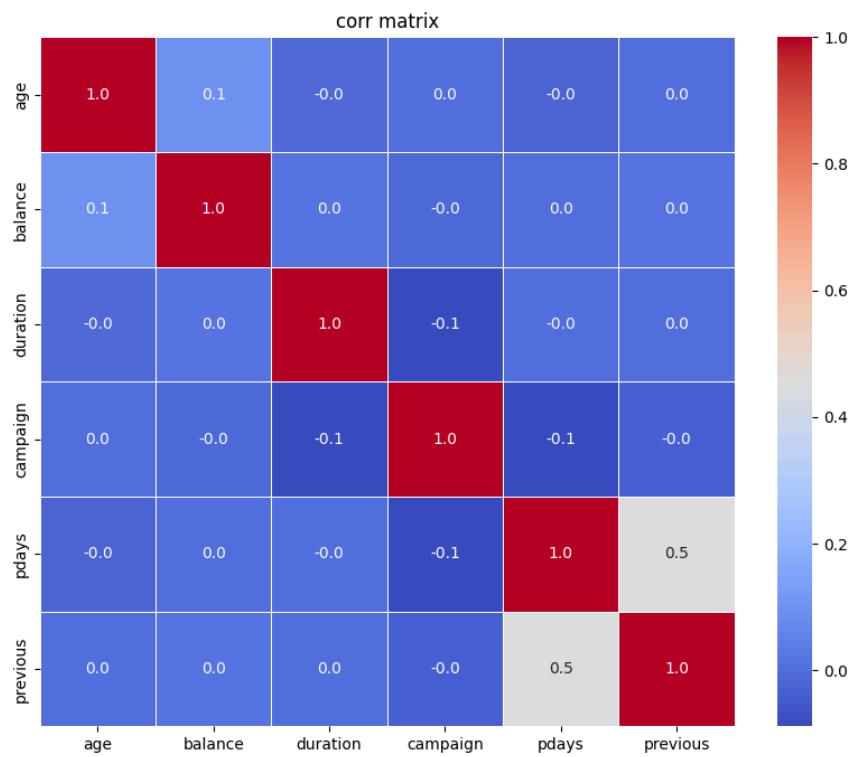
Phân phối số ngày kể từ lần gọi trước trong chiến dịch trước (-1 nếu chưa gọi)



Phân phối số lần liên hệ trước đó trong chiến dịch trước

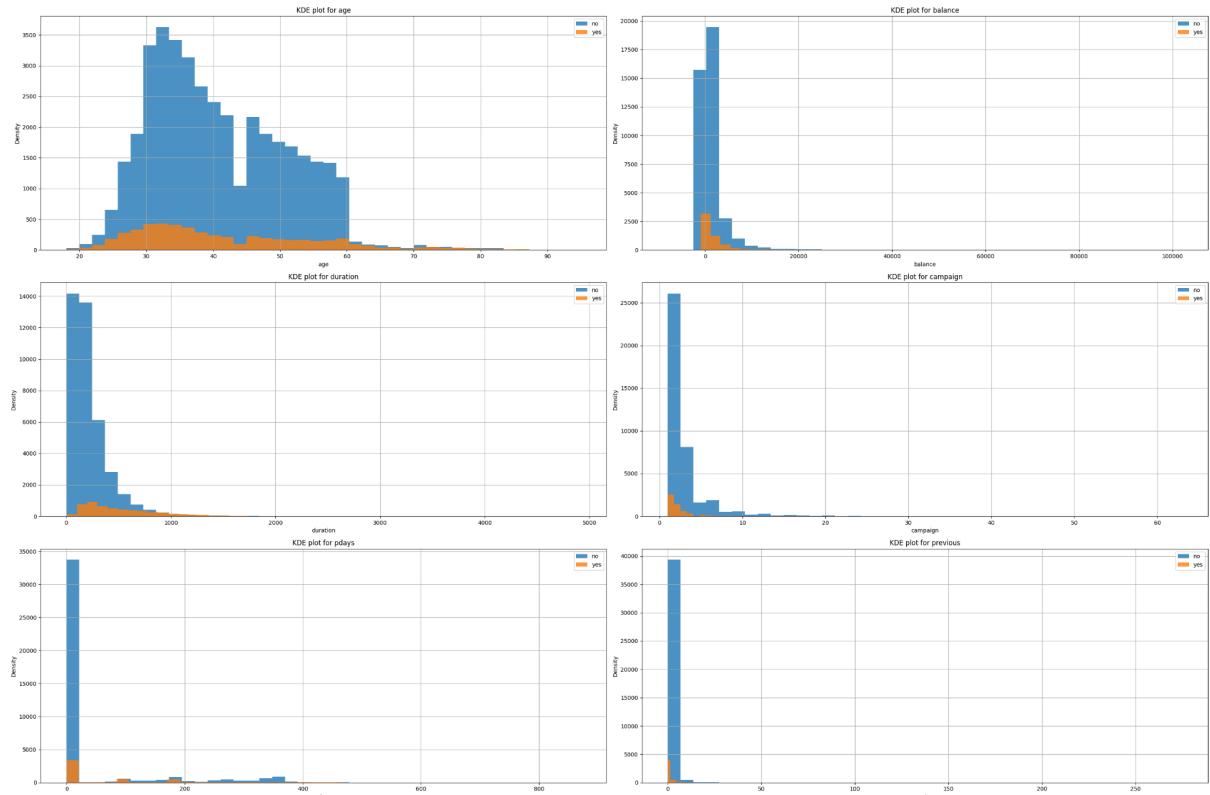
- Kết luận 2:

- + Missing value có các giá trị như pdays=-1 (ta có thể hiểu là khách hàng này chưa được liên hệ lần nào)



Ma trận tương quan giữa các thuộc tính liên tục

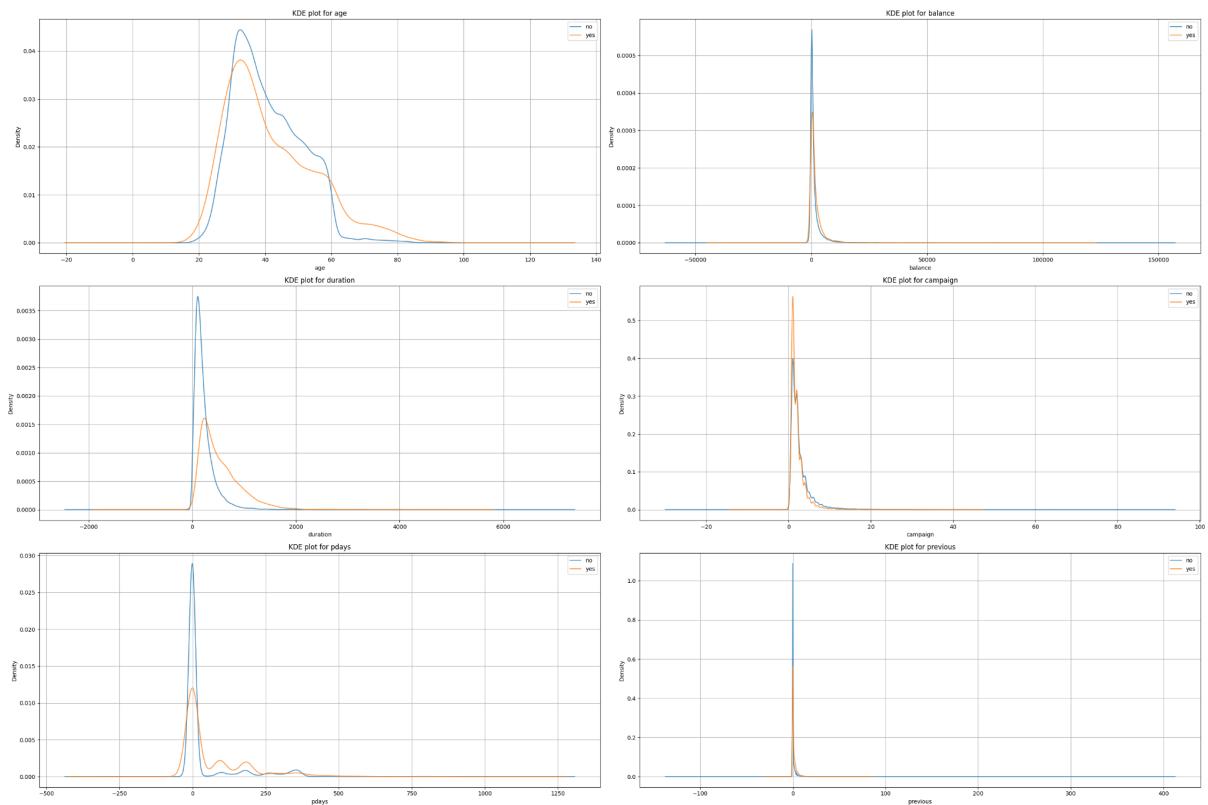
b. Data Analyzing



Histogram tương quan tỉ lệ giữa dữ liệu nhãn và TARGET khách hàng

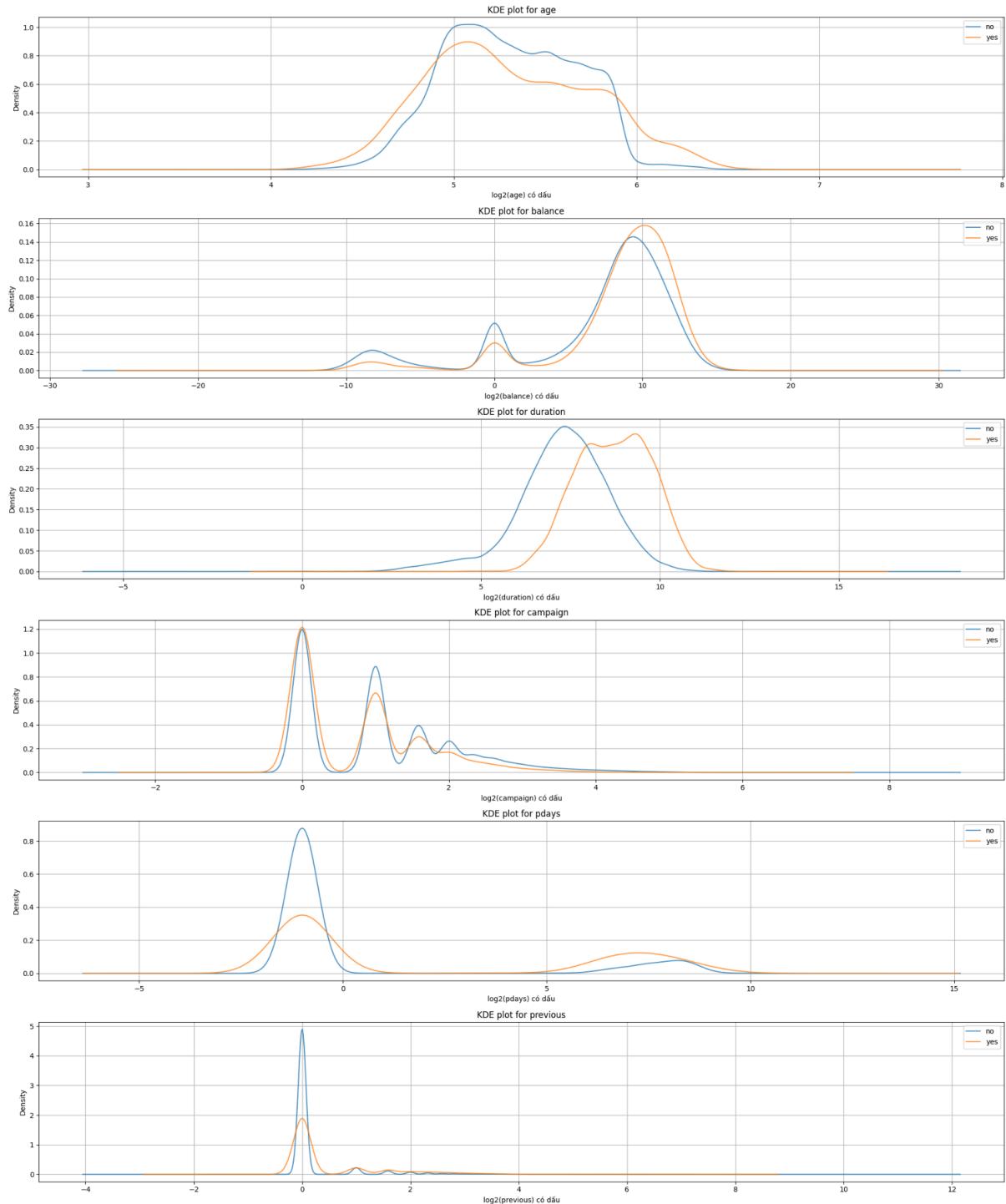
- Vì dữ liệu mất cân bằng nhăn, nên thay vì sử dụng histogram, nhóm sẽ sử dụng biểu đồ KDE (Kernel Density Estimation) - Ước lượng mật độ hạt nhân

KDE là một phương pháp phi tham số để ước lượng hàm mật độ xác suất (Probability Density Function - PDF) của một biến ngẫu nhiên. Nói cách khác, nó giúp vẽ một đường cong mềm mại để biểu diễn sự phân bố của dữ liệu.



KDE của các giá trị rời rạc tương quan TARGET

- Vì miền giá trị của balance, campaign... là rất lớn khiến cho việc quan sát dữ liệu khó khăn, nhóm sẽ phi tuyến bằng $\log_2(x + 1)$

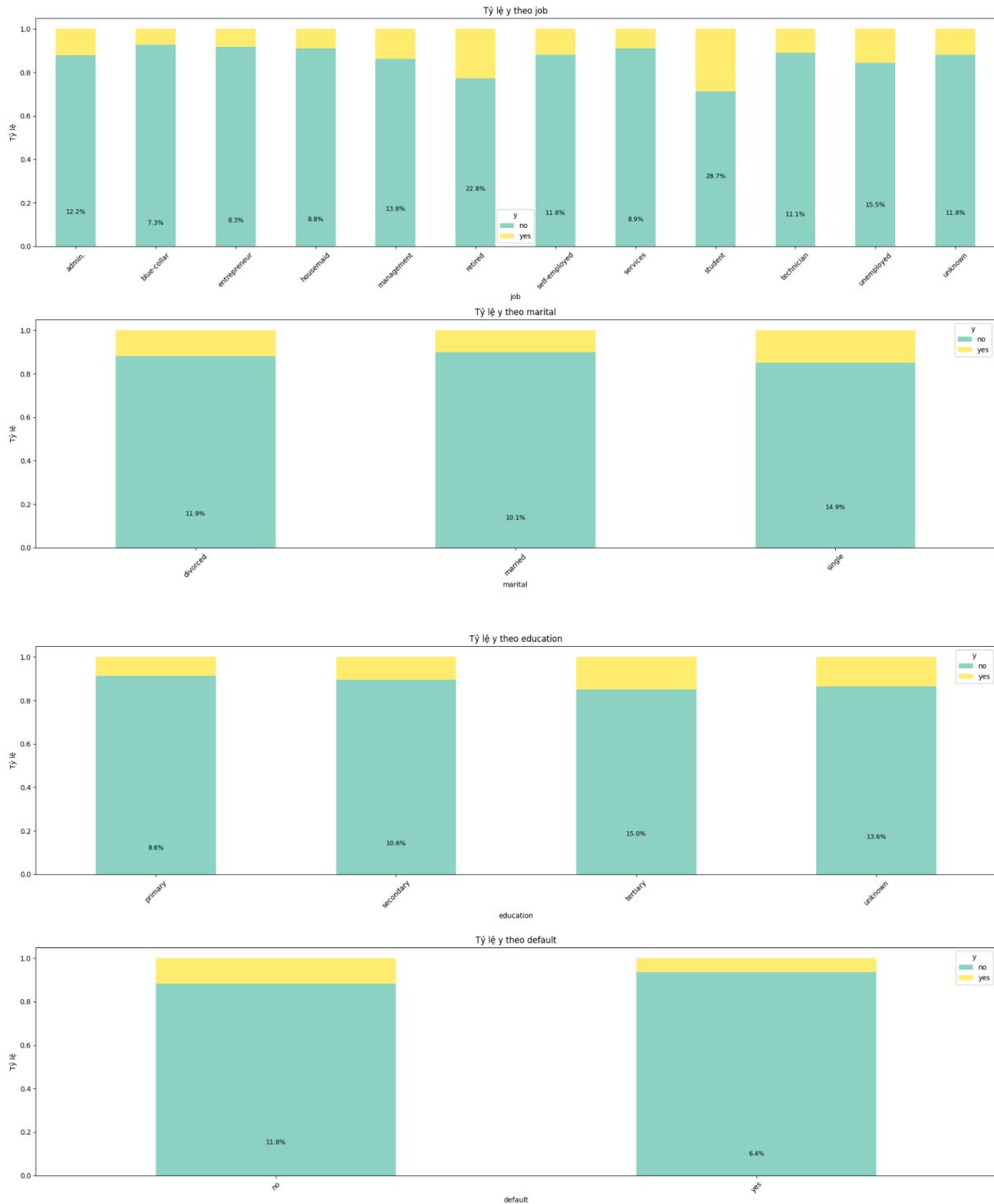


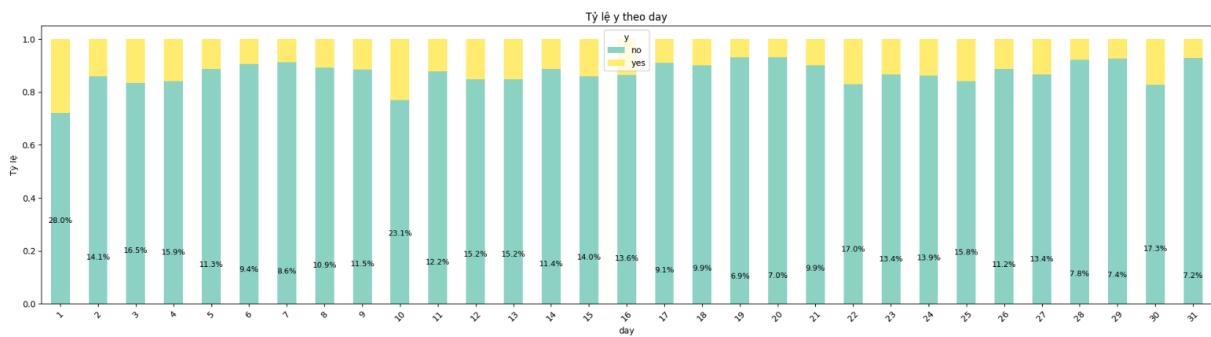
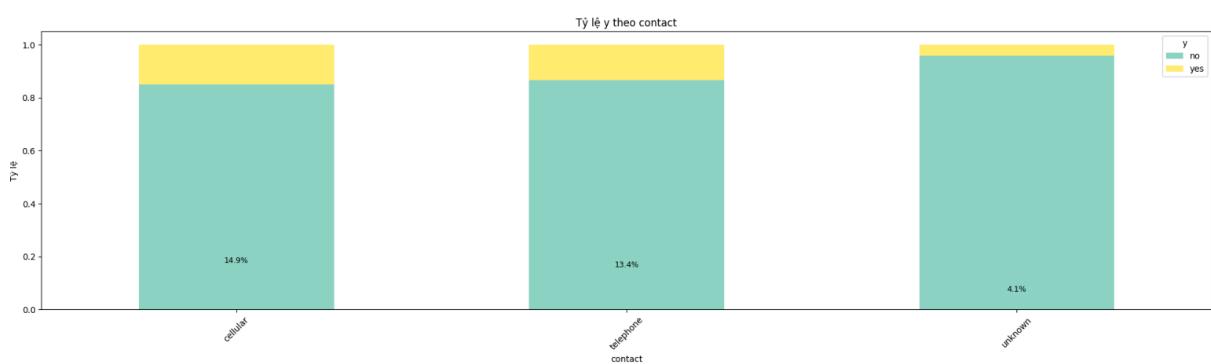
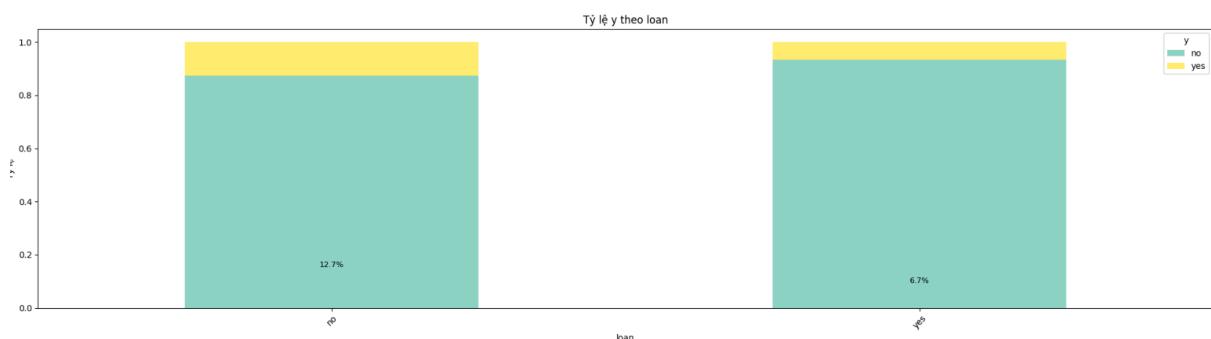
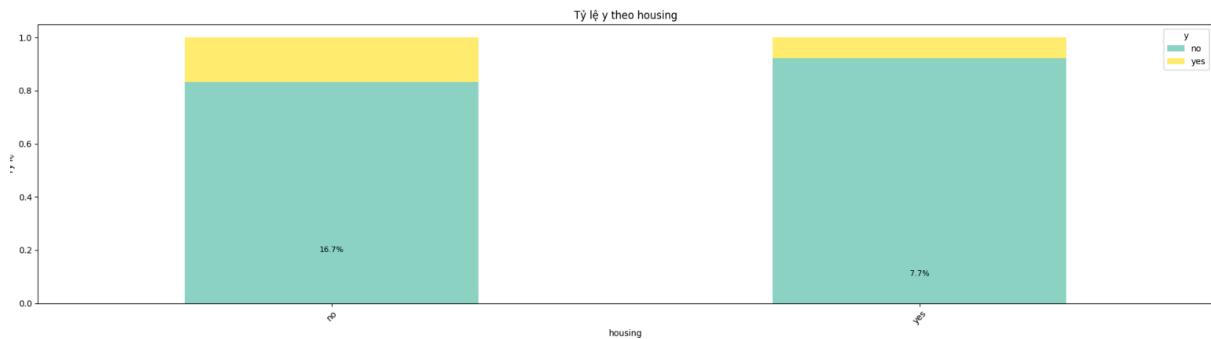
KDE sau khi logarit trục x cho miền giá trị

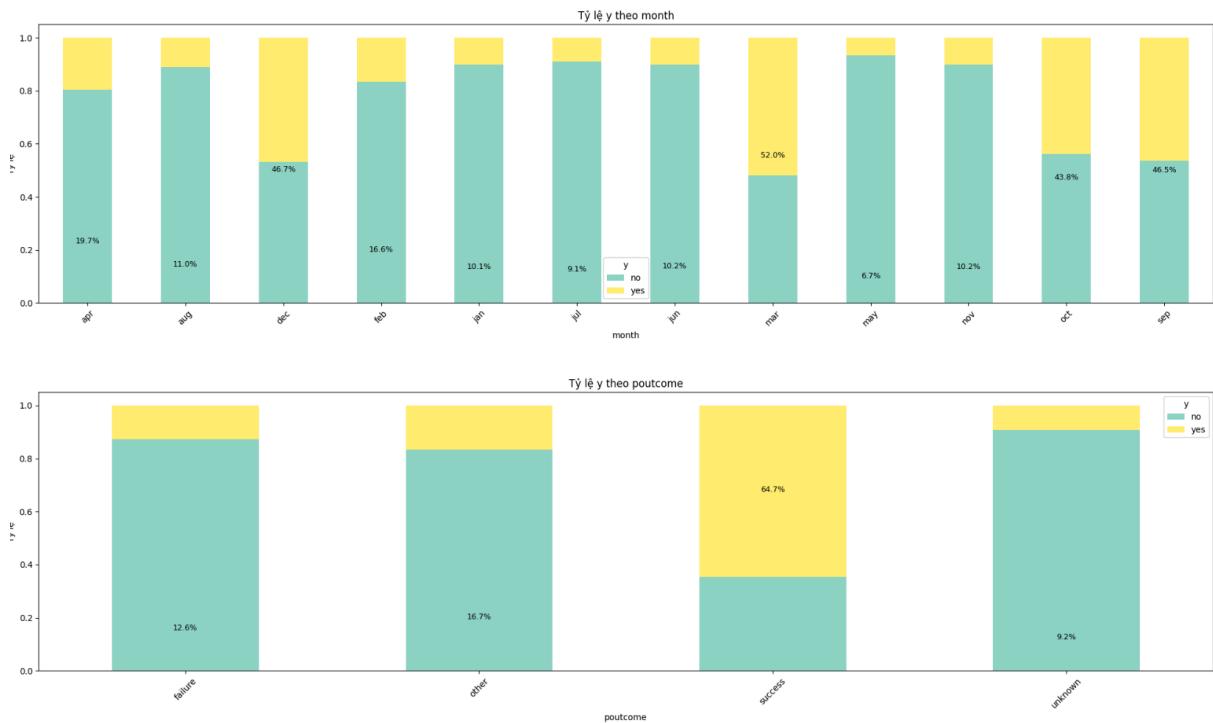
- Kết luận 1:

- + duration (thời lượng cuộc gọi) có ảnh hưởng tới quyết định khá nhiều, thời lượng cuộc gọi càng nhiều dẫn tới xác suất quyết định mua càng lớn
- + previous (số lần liên hệ trước đó) có ảnh hưởng mạnh tới quyết định, số lần liên hệ ít hơn có xác suất no cao hơn
- + pdays(số ngày từ lần cuối gọi) ảnh hưởng mạnh, nếu chưa từng gọi khả năng cao là no, nếu cuộc gọi gần đây thì xác suất cao hơn

- + age (tuổi) ảnh hưởng nhẹ. Tuổi trẻ ($<=30$) và về hưu ($>=60$) thì xác suất yes cao hơn, $30 < \text{độ tuổi} < 60$ có xác suất no cao hơn
- + balance(số dư trung bình năm) có ảnh hưởng nhẹ, số dư cao thì xác suất yes cao hơn, số dư âm thì xác suất no cao hơn
- + campaign(số lần liên hệ mời/giới thiệu) ảnh hưởng rất nhẹ, số lần liên hệ ít xác suất yes cao hơn







Biểu đồ cột tương quan tỉ lệ giữa các giá trị nhãn

- **Kết luận 2:** mặc dù số lượng y=no chiếm đa số, nhưng những giá trị sau đây ảnh hưởng nhiều tới quyết định y=yes (giá trị mạnh)
 - + poutcome=success (nếu trước đây từng đăng ký thì lần này xác suất cao sẽ lại đăng ký)
 - + month=dec, mar, oct, sep (tháng 3, 9, 10, 12)
 - Tháng 3: tuần Thánh - lễ Phục Sinh
 - Tháng 9: hội chợ truyền thống
 - Tháng 10: hội chợ + lễ hành hương
 - Tháng 12: Giáng sinh + Năm mới
 - + job=student; job=management (số lượng khá nhiều); job=retire (đã nghỉ hưu, tương ứng với độ tuổi >60 như phân tích trên luôn); blue-collar, entre, housemaid xác suất no cao
 - + day=1 (đầu tháng)
 - + housing (no xác suất cao yes, yes xác suất cao no)
 - + default=no (không có nợ xấu)
 - + loan (no xác suất cao yes, yes xác suất cao no)
 - + contact=unknown (unknown khá nhiều, nhưng xác suất yes lại rất thấp, nhưng xác suất no rất cao, khả năng phân vùng cao)

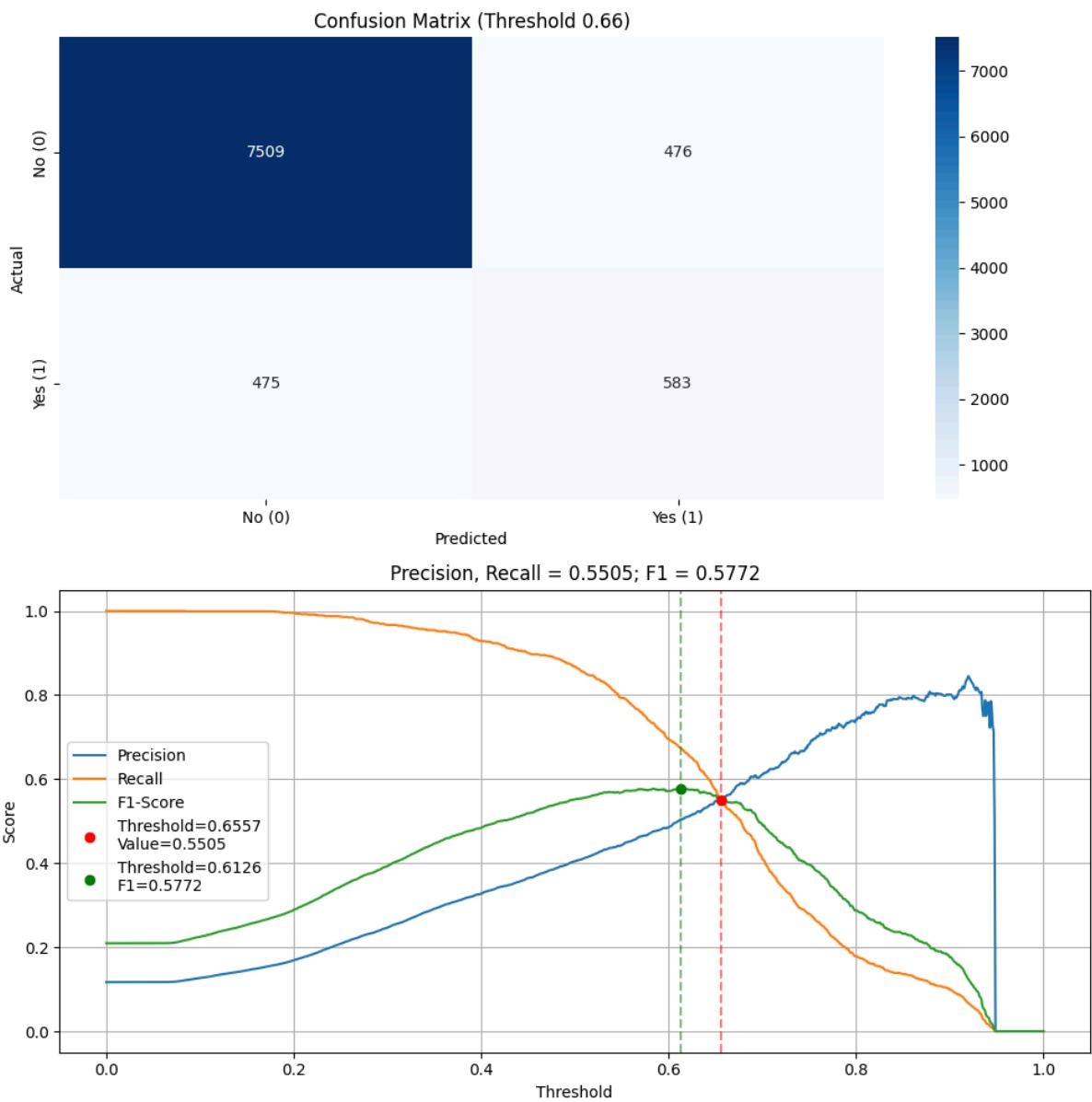
Trong phạm vi báo cáo, nhóm sẽ chỉ phân tích và triển khai các thuộc tính ảnh hưởng trực tiếp tới mục tiêu của bộ dữ liệu. Chi tiết về phân tích tương quan giữa những feature ở https://github.com/nguyenvietkhiemm/data_mining/tree/main

2. Đưa ra statement, phát biểu của nghiên cứu (Formulating a Statement/Research Hypothesis)

- **Phát biểu 1:** "Thời lượng cuộc gọi cuối cùng (duration) là yếu tố có ảnh hưởng mạnh nhất đến việc khách hàng đăng ký tiền gửi có kỳ hạn, và một mô hình dự đoán sử dụng chủ yếu yếu tố này cùng với số lần liên hệ trong chiến dịch (campaign) có thể đạt được độ chính xác (AUC) trên 90%."
- **Phát biểu 2:** "Số dư trung bình hàng năm của khách hàng nằm trong miền giá trị [-8019; 10217], tức là sẽ có những khách hàng âm số dư.Thêm nữa, những khách hàng có balance ≤ 0 thường tập trung ở ngành nghề housemaid (giúp việc), student (học sinh). Do vậy, việc một khách hàng có số dư âm và kết hợp với nghề nghiệp nào tương quan ảnh hưởng mạnh tới quyết định đăng ký"
- **Phát biểu 3:** "Khách hàng ở độ tuổi trung niên (ví dụ: 30-50 tuổi) sẽ có tỷ lệ đăng ký tiền gửi thấp hơn đáng kể so với các nhóm khác."
- **Phát biểu 4:** "Có thể xây dựng một mô hình phân loại dự đoán khách hàng đăng ký tiền gửi có kỳ hạn với chỉ số F1-score cho lớp 'yes' (khách hàng đăng ký) ít nhất là 0.5, bằng cách sử dụng kết hợp thông tin nhân khẩu học của khách hàng và chi tiết các liên hệ trong chiến dịch trước đó."
- **Kết luận:** Nhóm sẽ tạo thêm các bộ đặc trưng tĩnh và thử nghiệm
 - + balance > 0
 - + pdays > 0
 - + campaign > 2 (mean)
 - + previous > 0
 - + bins số tuổi (0 - 30, 30 - 60, >60)
 - + duration > 3000 (threshold)
 - + Tạo các tổ hợp
 - job + ["marital", "education", "default", "housing", "loan", "contact", "balance > 0"]
 - default + ["balance > 0", "campaign > 2"]
 - housing + ["balance > 0", "campaign > 2", "contact"]
 - loan + ["balance > 0", "campaign > 2", "pdays > 0", "contact"]

3. Thiết kế các thử nghiệm dựa trên bộ dữ liệu mình có (Designing Experiments)

- *Mô hình sử dụng: Random Forest và LightGBM*
- **Chưa thêm bất kỳ đặc trưng nào. Chuẩn hóa giá trị numeric và one hot encode các giá trị cat**



Precision, Recall tối ưu khoảng 0.5505

- **Kiểm thử với down/up sampling dữ liệu**

Vì Precision, Recall khá thấp. Điều này có thể đến từ việc mất cân bằng nhãn nên nhóm kiểm thử Cross-validation trên tập dữ liệu đã được up/down scale

	Original Data	Upsampled Data	Down Sampled Data
CV Precision	0.4664	0.8789	0.8507
CV Recall	0.8360	0.9570	0.8877
CV F1	0.5987	0.9163	0.8688
CV AUC	0.9299	0.9644	0.9292

Test Precision	0.4546	0.4590	0.4332
Test Recall	0.8422	0.8469	0.8913
Test F1	0.5905	0.5953	0.5830
Test AUC	0.9288	0.9285	0.9282

- **Kết luận:** Như ta có thể thấy, CV của up/down sampling tăng vọt. Điều này là dấu hiệu của việc data leakage. Tức là tồn tại các bản ghi dữ liệu được sampling trong các fold test. Mặc dù các giá trị CV của up/down sampling cao hơn, nhưng tổng thể thì lại không khác dữ liệu ban đầu, AUC có cao hơn đôi chút. Do vậy, việc precision, recall thấp đến từ việc chênh lệch nhãn là chính. Nhóm quyết định vẫn sẽ sử dụng bộ data ban đầu.

- **Bins tuổi**

BINS TUỔI

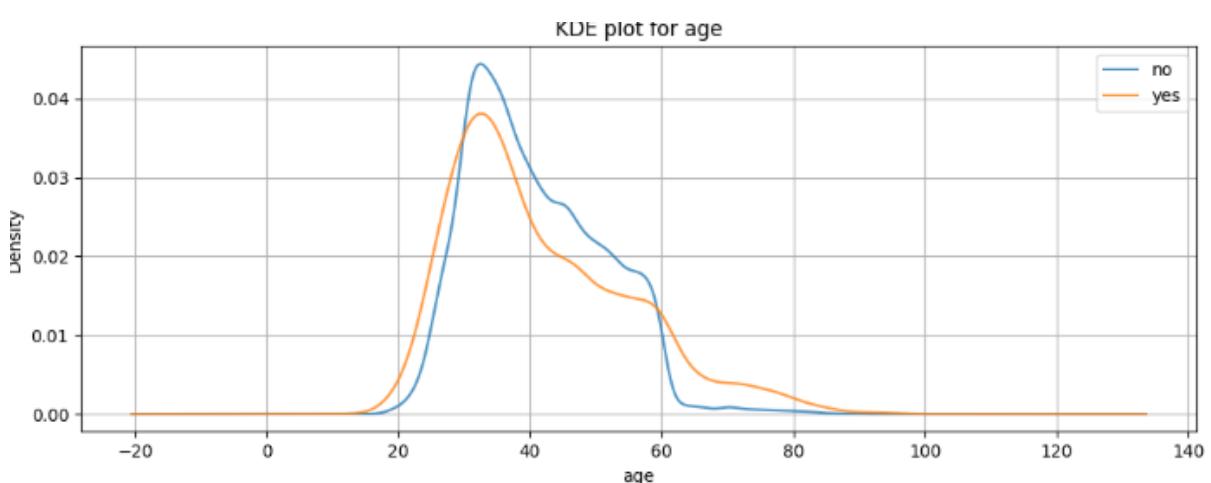
```

bins = [0, 30, 60, np.inf]
labels = ['young', 'ad', 'old']

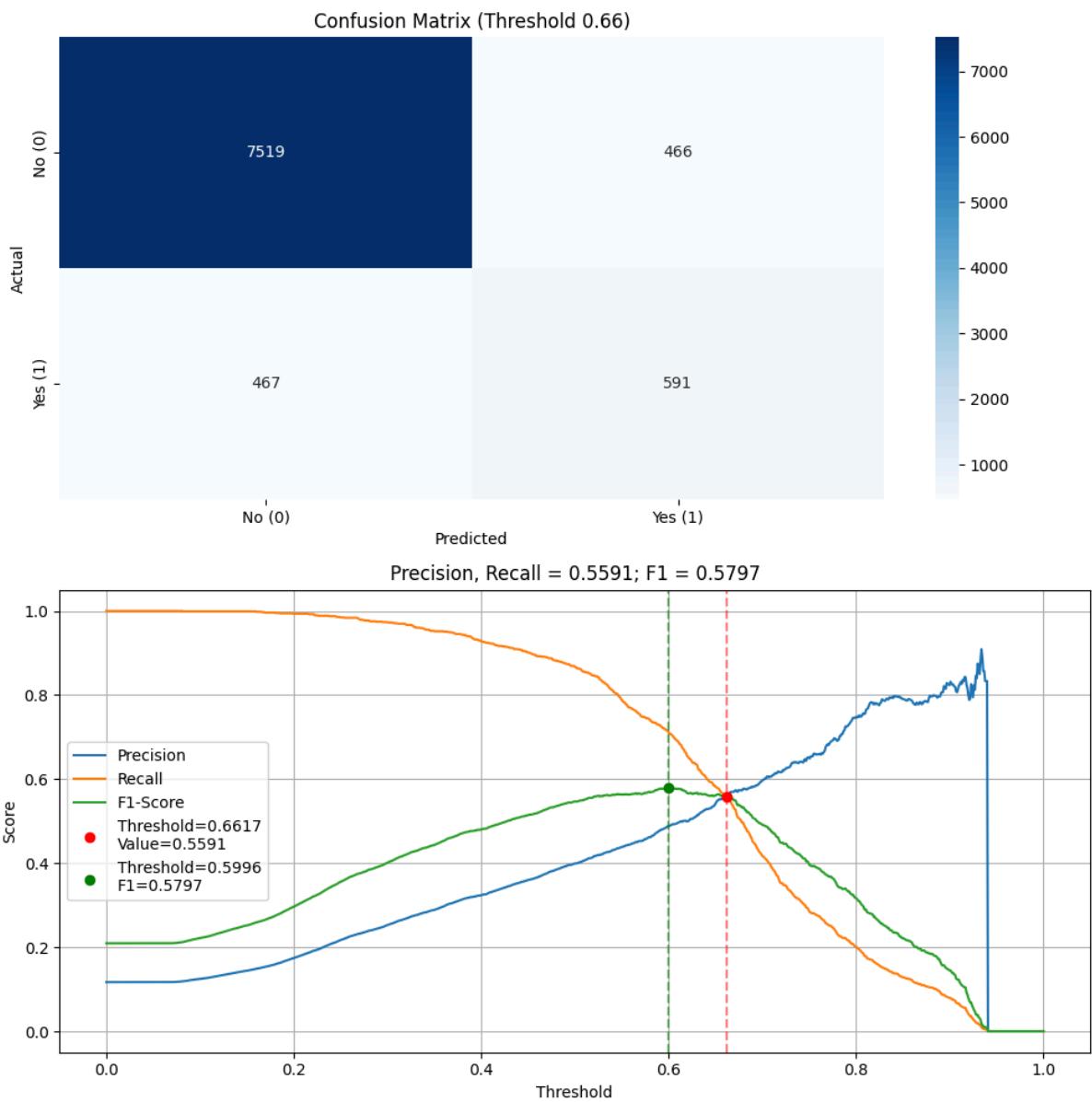
df['age'] = pd.cut(df['age'], bins=bins, labels=labels, right=False)

```

Bins tuổi (nhóm các khoảng tuổi tác)

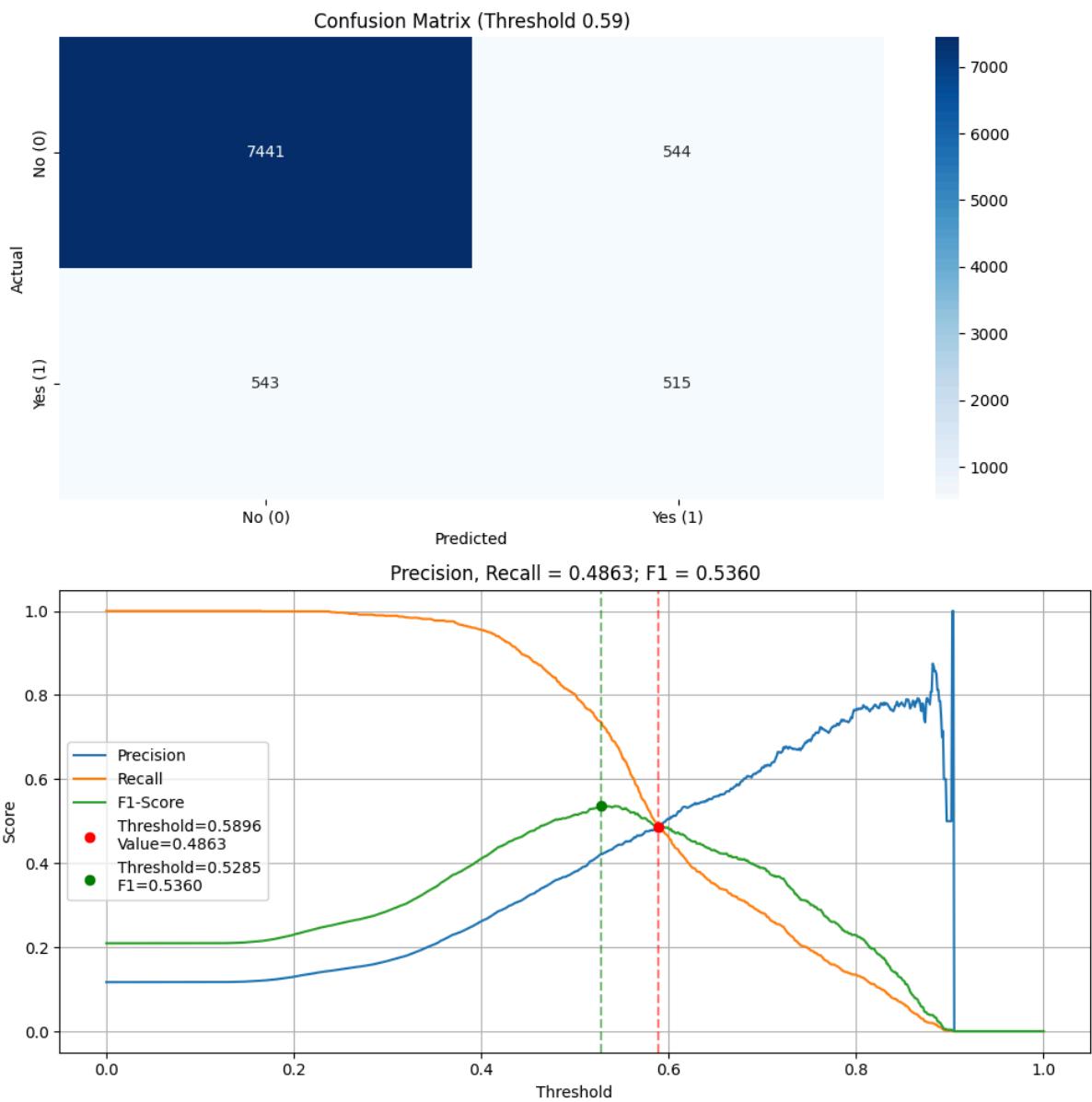


Ta thấy độ tuổi có xác suất TARGET=yes cao hơn nằm trong khoảng ≤ 30 và ≥ 60 tuổi



Precision, Recall tối ưu khoảng 0.5591

- Kết quả khi ta bins tuổi, cả precision và recall đều tăng
- **Balance > 0 kết hợp job**

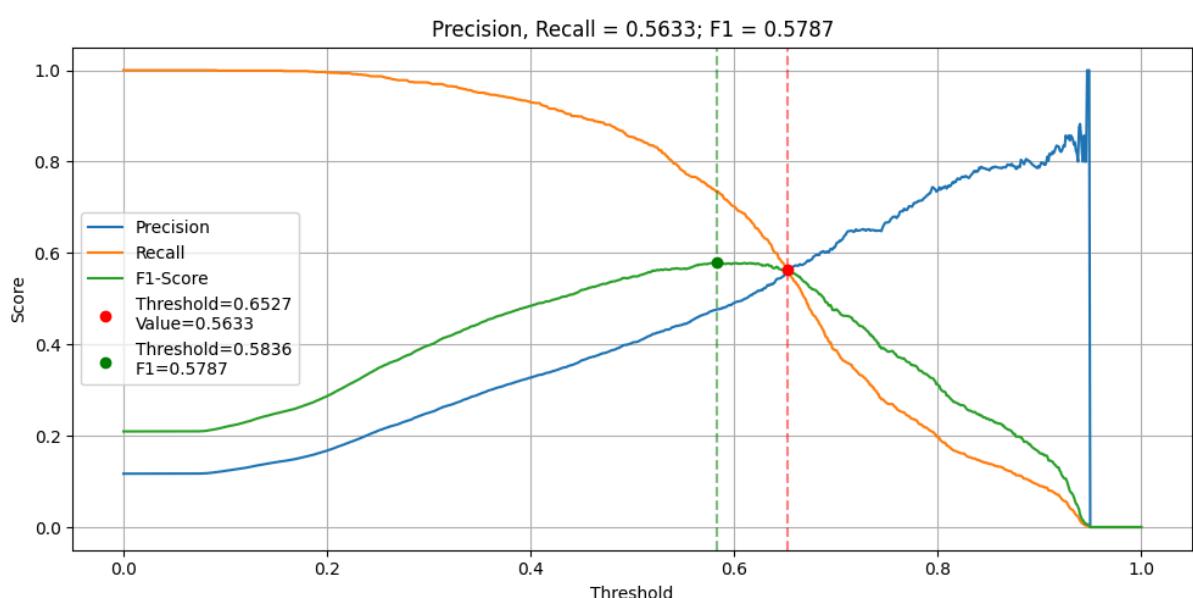
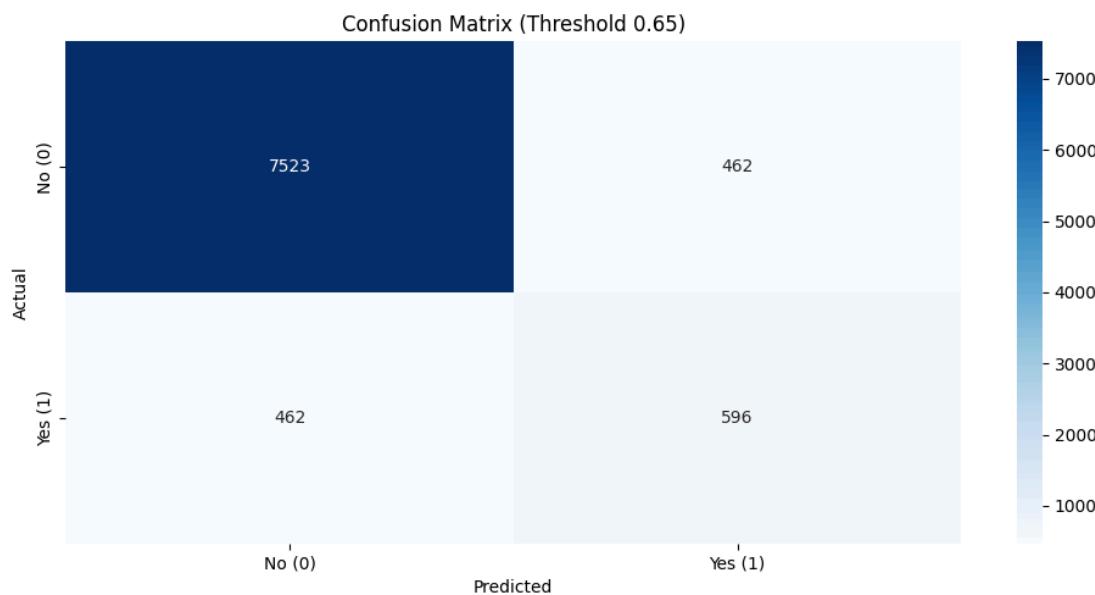
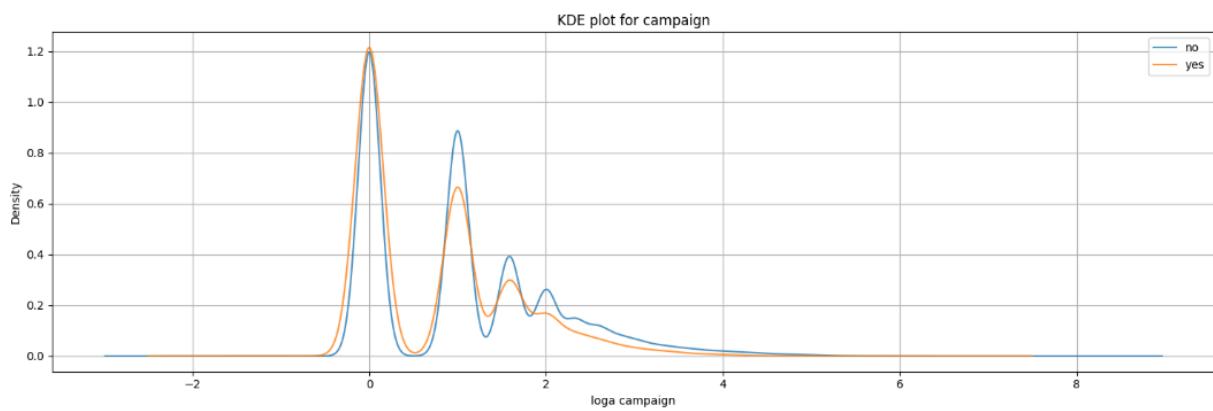


Precision, Recall tối ưu khoảng 0.4863

- Kết quả khi ta thêm balance > 0 và tổ hợp job, kết quả bị giảm xuống

- **Campaign > 2 (mean của campaign)**

Campaign là số lần liên hệ trong chiến dịch hiện tại, bao gồm cả lần này



Precision, Recall khoảng 0.5633

- Thêm biến phân loại **campaign > 2**, precision và recall tăng

- Extracting + lightGBM

```
# # # --- Feature Engineering ---

# df['balance_per_campaign'] = df['balance'] / df['campaign']
# df['balance_per_previous'] = np.where(df['previous'] > 0, df['balance'] / df['previous'], 0)
df['duration_per_campaign'] = df['duration'] / df['campaign']
# df['age_from_mean'] = df['age'] - df['age'].mean()
# df['campaign_minus_previous'] = df['campaign'] - df['previous']
# df['total_contacts'] = df['campaign'] + df['previous']

# df['balance_gt_0'] = (df['balance'] >= 0).astype(object)

df['campaign_gt_3'] = (df['campaign'] > 3).astype(int)
df['duration_gt_mean'] = (df['duration'] > 3000).astype(object)
```

Các phép tính như ratio, delta trên dữ liệu liên tục tương quan giữa duration với campaign...

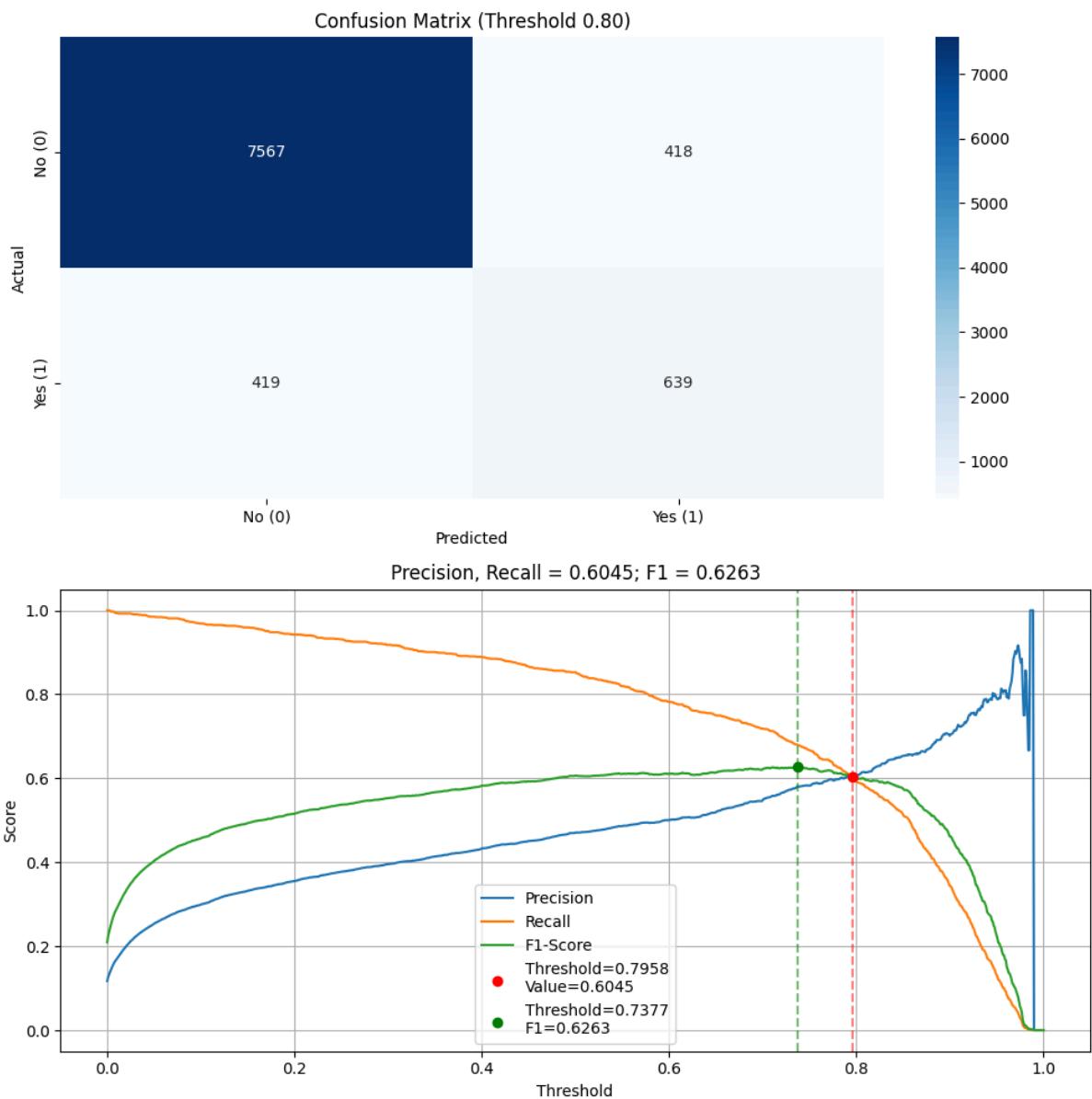
Best cross-val F1: 0.6067141592353338

Test ROC AUC: 0.9318 (cao nhất trong số các thử nghiệm)

Test F1: 0.6062

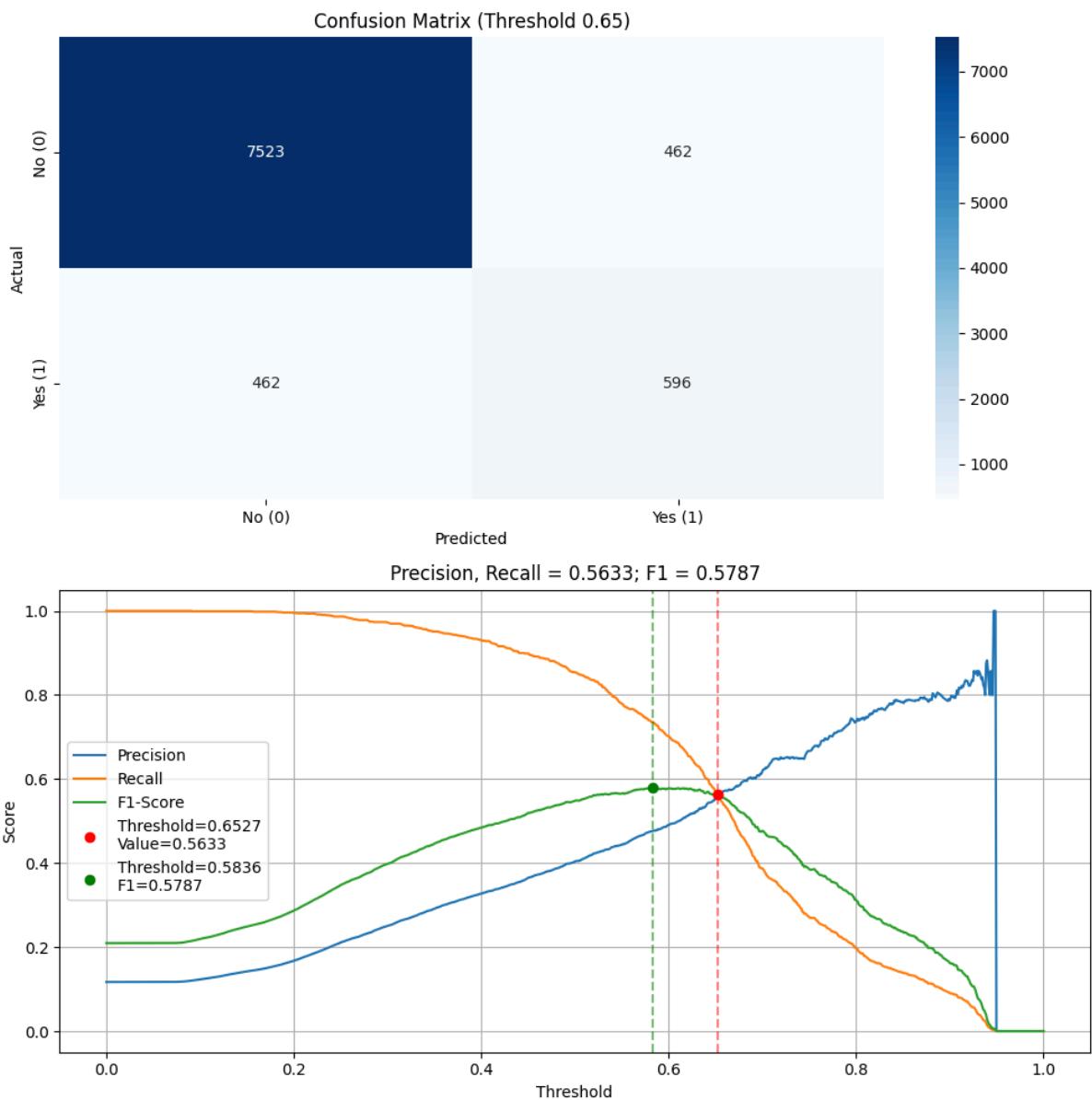
Test Precision: 0.4703

Test Recall: 0.8526



Precision, Recall khoảng 0.6045

- Extracting + Random Forest



Precision, Recall khoảng 0.5633

- **Kết luận:**

- + Giả thiết campaign > 2 (mean) sẽ phân biệt mạnh mẽ khách hàng hơn là đúng
- + Giả thiết về nhóm các độ tuổi sẽ phân biệt mạnh mẽ khách hàng hơn là đúng
- + Giả thiết các khách hàng có duration > 3000 (thời lượng cuộc gọi ngắn) sẽ phân biệt mạnh mẽ khách hàng hơn là đúng
- + Giả thiết phân biệt khách hàng có số dư âm và nghề nghiệp tương ứng có sự phân biệt mạnh mẽ là sai

Bank Marketing UCI
The data is related with direct marketing campaigns of a Portuguese banking institution

Overview Data Code Models Discussion Leaderboard Rules Team Submissions

Leaderboard

[Raw Data](#) [Refresh](#)

YOUR RECENT SUBMISSION
submission.csv Submitted by Khiêm Nguyễn - Submitted 31 minutes ago
[Jump to your leaderboard position](#)

Score: 0.97945
Public score: 0.87009

Search leaderboard

[Public](#) [Private](#)

The private leaderboard is calculated with approximately 20% of the test data.
This competition has completed. This leaderboard reflects the final standings.

#	Team	Members	Score	Entries	Last	Solution
1	kaggle bank solution.csv		1.00000			
2	Eyangya		0.96575	24	7y	
3	Nicolas Maignan		0.95205	10	7y	
4	Chrisma SSN		0.94520	11	7y	
5	Sherry Francis		0.94099	15	7y	
5	Vishnu R Nambiar		0.94099	19	7y	

Top 1 Private Score (tập dữ liệu nhỏ) trên Kaggle và Top 8 Public Score (tập dữ liệu lớn) với lightGBM chứng minh giả thiết là đúng đắn

YOUR RECENT SUBMISSION
submission_RF.csv Submitted by Khiêm Nguyễn - Submitted 3 minutes ago
[Jump to your leaderboard position](#)

Score: 0.90602
Private score: 0.97945

Search leaderboard

[Public](#) [Private](#)

This leaderboard is calculated with approximately 80% of the test data. The final results will be based on the other 20%, so the final standings may be different.

#	Team	Members	Score	Entries	Last	Solution
1	kaggle bank solution.csv		1.00000			
2	Nicolas Maignan		0.91196	10	7y	
2	Vishnu R Nambiar		0.91021	19	7y	
3	Chrisma SSN		0.90863	11	7y	
4	Eyangya		0.90024	24	7y	
5	RahulVarghese		0.88870	7	7y	
6	afeen aroob		0.88870	8	7y	
7	vidya		0.87105	5	7y	

Top 1 Private Score (tập dữ liệu nhỏ) trên Kaggle và Top 4 Public Score (tập dữ liệu lớn) với Random Forest chứng minh giả thiết là đúng đắn

III. Kết luận

- Sau khi chạy 4 thuật toán với các cách trích xuất đặc trưng khác nhau của bộ dữ liệu Bank Marketing, nhóm thấy mô hình Random Forest cùng cách trích xuất đặc trưng tinh mang lại kết quả tốt nhất.
- Trong bài toán phân loại, Random Forest sẽ cho ra kết quả tốt nhất với các đặc trưng được chọn lọc. Không phải cứ có thật nhiều feature đặc trưng để phân loại là kết quả sẽ chính xác hơn mà trong báo cáo này, ta phải thử một cách thủ công các đặc trưng gây nhiễu cho dự đoán
- Đối với việc khảo sát dữ liệu, việc đưa ra các giả thiết là vô cùng quan trọng, đặc biệt là các giả thiết liên quan trực tiếp đến sự tương quan ý nghĩa của đặc trưng dựa trên domain knowledge. Mặc dù LightGBM là một mô hình tốt để lựa chọn đặc trưng phù hợp Random Forest, nhưng qua thử nghiệm trên, ta vẫn thấy được tầm quan trọng của feature selection.