

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO THỰC TẬP CƠ SỞ

Giảng viên hướng dẫn	: TRẦN ĐÌNH QUẾ
Họ và tên sinh viên	: BÙI MẠU VĂN
Mã sinh viên	: B22DCCN887
Lớp	: D22CQCN11_B

Hà Nội – 2025

Yêu cầu:

Bài tập 1 (2 tuần)

Hạn Nộp: Trước 12g PM, Tối 11/03

1. Sinh viên cài đặt và chạy case study trong Chap 12 (Tài liệu 1).
2. Viết thành tài liệu những hiểu biết về
 - Kỹ thuật cleaning data
 - Các kỹ thuật Machine Learning trình bày trong Chap này
 - Giải thích code và Chụp ảnh demo

Mục lục

I. Kỹ thuật Cleaning data

- a. Khái niệm
- b. Các kỹ thuật xử lý

II. Các kỹ thuật Machine Learning trong Chapter 12

- 1. Logistic Regression
- 2. K-NN
- 3. SVM – linear kernel
- 4. SVM – RBF kernel

III. Chạy case study

- a. Tiền xử lý dữ liệu
- b. Đánh giá thuật toán
- c. Deploy model
- d. Creat client app

I. **Kỹ thuật Cleaning Data**

a. **Khái niệm**

- Đây là một bước rất quan trọng trong quá trình xử lý dữ liệu, giúp loại bỏ lỗi và cải thiện chất lượng dữ liệu trước khi đưa vào phân tích hoặc huấn luyện mô hình.
- Làm sạch dữ liệu là quá trình sửa chữa hoặc loại bỏ không chính xác, bị lỗi, được định dạng không chính xác, sao chép hoặc dữ liệu không đầy đủ trong bộ dữ liệu. Khi kết hợp nhiều nguồn dữ liệu, có nhiều cơ hội để dữ liệu được nhân đôi hoặc dán nhãn sai. Nếu dữ liệu không chính xác, kết quả và thuật toán là không đáng tin cậy, mặc dù chúng có thể trông chính xác.
- Các vấn đề khi cleaning data:
 - Dữ liệu bị thiếu
 - Dữ liệu bị trùng lặp
 - Dữ liệu không hợp lệ hoặc ngoại lệ
 - Dữ liệu không đúng định dạng
- Quy trình làm sạch dữ liệu:
 - Kiểm tra dữ liệu
 - Xử lý dữ liệu bị thiếu
 - Xử lý dữ liệu trùng lặp
 - Xử lý outliers
 - Chuẩn hóa dữ liệu
 - Chuyển đổi kiểu dữ liệu

Note: data cleaning khác với data transformation

b. **Các kỹ thuật xử lý**

1. **Xử lý dữ liệu bị thiếu (Handling Missing Values):**

- **Loại bỏ (Deletion):**
 - Loại bỏ các hàng hoặc cột chứa giá trị bị thiếu.
 - Phù hợp khi dữ liệu bị thiếu là ngẫu nhiên và chiếm tỷ lệ nhỏ.

- **Thay thế (Imputation):**

- Thay thế giá trị bị thiếu bằng giá trị khác.
- Các phương pháp thay thế phổ biến:
 - Giá trị trung bình (mean imputation).
 - Giá trị trung vị (median imputation).
 - Giá trị xuất hiện nhiều nhất (mode imputation).
 - Sử dụng các thuật toán học máy để dự đoán giá trị bị thiếu.
 - Sử dụng kỹ thuật nội suy hoặc ngoại suy

- **Giữ nguyên (Keeping):**

- Trong một số trường hợp, giữ nguyên giá trị thiếu và xử lý chúng trong quá trình phân tích.

2. Loại bỏ dữ liệu trùng lặp (Removing Duplicates):

- Tìm và loại bỏ các bản ghi trùng lặp trong tập dữ liệu.
- Có thể dựa trên toàn bộ bản ghi hoặc một số cột cụ thể.

3. Sửa chữa dữ liệu sai lệch (Correcting Inaccurate Data):

- Sửa chữa các lỗi chính tả, định dạng và các lỗi khác trong dữ liệu (viết hoa/thường, loại bỏ khoảng trắng).
- Kiểm tra tính hợp lệ của dữ liệu dựa trên các quy tắc và ràng buộc:
Chuyển đổi kiểu dữ liệu (string → numeric, date formatting)
- Xử lý các ký tự đặc biệt không mong muốn
- Sử dụng các công cụ và kỹ thuật như:
 - Biểu thức chính quy (regular expressions).
 - So sánh dữ liệu với các nguồn tham khảo.

4. Chuẩn hóa dữ liệu (Standardizing Data):

- Đảm bảo rằng dữ liệu được định dạng và đo lường một cách nhất quán.
- Các kỹ thuật chuẩn hóa phổ biến:
 - Chuẩn hóa Min-Max: Đưa dữ liệu về dải $[0,1]$
 - Chuẩn hóa Z-score: Chuyển về phân phối chuẩn ($\mu=0, \sigma=1$)
 - Log transformation cho dữ liệu có phân phối lệch

5. Xử lý dữ liệu ngoại lai (Handling Outliers):

- Xác định và xử lý các giá trị ngoại lai, là những giá trị khác biệt đáng kể so với phần còn lại của dữ liệu.
- Phát hiện outliers bằng biểu đồ Box Plot, Z-score, IQR
- Các phương pháp xử lý dữ liệu ngoại lai:
 - Loại bỏ.
 - Thay thế bằng giá trị biên.
 - Chuyển đổi dữ liệu.

c. Công Cụ Hỗ Trợ:

- Các thư viện Python: Pandas, NumPy.
 - Các công cụ phần mềm: OpenRefine, Trifacta Wrangler.
 - Các ngôn ngữ truy vấn cơ sở dữ liệu (SQL).
-

II. Thuật toán ML trong chapter 12

1. Logistic Regression

- Khái niệm:

“Logistic regression is a classification model that uses input variables (features) to predict a categorical outcome variable (label) that can take on one of a limited set of class values. A binomial logistic regression is limited to two binary output categories, while a multinomial logistic regression allows for more than two classes.”

Logistic Regression là một thuật toán **học có giám sát (supervised learning)** dùng để **phân loại (classification)**. Mặc dù có tên gọi “Regression” (hồi quy), nhưng Logistic Regression chủ yếu được sử dụng cho **bài toán phân loại nhị phân (binary classification)** và mở rộng cho **phân loại đa lớp (multiclass classification)**.

Mục tiêu của Logistic Regression là dự đoán xác suất một mẫu thuộc về một lớp cụ thể dựa trên các đặc trưng đầu vào.

Khác với hồi quy tuyến tính xuất ra giá trị liên tục, hồi quy logistic chuyển đổi kết quả bằng cách sử dụng hàm logistic (sigmoid) để trả về giá trị xác suất từ 0 đến 1.

Ví dụ: 0 - đại diện cho một lớp tiêu cực; 1 - đại diện cho một lớp tích cực. Hồi quy logistic thường được sử dụng trong các vấn đề phân loại nhị phân trong đó biến kết quả cho thấy một trong hai loại (0 và 1).

- Công thức của Logistic Regression:

+ Logistic Regression dự đoán xác suất một mẫu thuộc về một lớp cụ thể.

+ Công thức của Logistic Regression được xây dựng dựa trên hàm sigmoid:

$$h(x) = \frac{1}{1 + e^{-(w^T x + b)}}$$

Trong đó:

- x là vector đặc trưng (feature vector) của dữ liệu đầu vào
- w là vector trọng số (weights)
- b là hệ số bias
- $h(x)$ là xác suất mẫu thuộc về lớp 1

+ Nếu $h(x) > 0.5$, mẫu được phân loại vào lớp 1.

+ Nếu $h(x) \leq 0.5$, mẫu được phân loại vào lớp 0.

- Các loại Logistic Regression:

+ **Binary Logistic Regression:** Dùng cho bài toán phân loại hai lớp (0 hoặc 1).

+ **Multinomial Logistic Regression:** Dùng cho bài toán phân loại nhiều lớp không thứ tự.

+ **Ordinal Logistic Regression:** Dùng cho bài toán phân loại nhiều lớp có thứ tự.

- Cách huấn luyện mô hình Logistic Regression:

+ Tính toán hàm mất mát (Loss Function):

Sử dụng hàm mất mát log-loss (Binary Cross Entropy):

$$J(w, b) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)})) \right]$$

+ Tối ưu trọng số:

Dùng **Gradient Descent** hoặc các thuật toán tối ưu khác (SGD, Adam, v.v.) để cập nhật trọng số w .

- Quy trình huấn luyện

+ Khởi tạo các hệ số (trọng số)

+ Tính xác suất dự đoán bằng các hệ số hiện tại

+ Tính hàm mất mát (thường là log loss hoặc cross-entropy)

+ Cập nhật hệ số để giảm thiểu mất mát (thường sử dụng gradient descent)

+ Lặp lại cho đến khi hội tụ

- Các chỉ số đánh giá

+ Accuracy

+ Precision và Recall

+ F1 Score

+ ROC curve and AUC

+ Log loss

+ Confusion matrix

- Ứng dụng của Logistic Regression:

+ Phát hiện ung thư (có bệnh / không có bệnh)

+ Dự đoán khách hàng có mua hàng không

+ Phân loại email là spam hay không

+ Nhận diện chữ viết tay

Ưu điểm & Nhược điểm của Logistic Regression

- Ưu điểm:

- + Dễ hiểu, dễ triển khai
- + Không yêu cầu nhiều tài nguyên tính toán
- + Hoạt động tốt với dữ liệu tuyến tính

- Nhược điểm:

- + Không hoạt động tốt với dữ liệu phi tuyến tính
- + Nhạy cảm với dữ liệu mất cân bằng (imbalanced data)
- + Dễ bị ảnh hưởng bởi outliers

2. K-Nearest Neighbors (KNN)

- Khái niệm:

+ K-Nearest Neighbor là một trong những thuật toán supervised-learning đơn giản nhất trong Machine Learning. Khi training, thuật toán này gần như không học gì từ dữ liệu training (hay còn được biết tới tên gọi là lazy learning), mọi tính toán được thực hiện khi mô hình cần dự đoán kết quả của dữ liệu mới cần dự đoán. K-nearest neighbor có thể áp dụng được vào cả hai loại của bài toán Supervised learning là Classification và Regression. Nhưng thường được dùng cho bài toán Classification

- Cách tính khoảng cách:

+ Để xác định điểm dữ liệu nào gần nhất với một điểm truy vấn nhất định, khoảng cách giữa điểm truy vấn và các điểm dữ liệu khác sẽ cần được tính toán. Các số liệu khoảng cách này giúp hình thành các ranh giới quyết định, trong đó các phân vùng truy vấn chỉ vào các khu vực khác nhau.

+ **Euclidean distance (p=2)**

$$d(x,y) = \sqrt{\sum_{i=1}^n (y_i - x_i)^2}$$

+ **Manhattan distance (p=1)**

$$\text{Manhattan Distance} = d(x,y) = \left(\sum_{i=1}^m |x_i - y_i| \right)$$

+ **Minkowski distance:**

$$\text{Minkowski Distance} = \left(\sum_{i=1}^n |x_i - y_i| \right)^{1/p}$$

+ **Hamming distance:**

$$\text{Hamming Distance} = D_H = \left(\sum_{i=1}^k |x_i - y_i| \right)$$

$$\begin{array}{ll} x=y & D=0 \\ x \neq y & D \neq 0 \end{array}$$

- Cách xác định số cụm K:

+ Giá trị K trong thuật toán K-NN xác định có bao nhiêu hàng xóm sẽ được kiểm tra để xác định phân loại của một điểm truy vấn cụ thể. Ví dụ: nếu $k = 1$, trường hợp sẽ được gán cho cùng một lớp với hàng xóm gần nhất của nó.

+ Nhìn chung, nên có một số lẻ cho K để tránh các mối quan hệ trong phân loại và các chiến thuật xác thực chéo có thể giúp bạn chọn K tối ưu cho bộ dữ liệu của bạn.

- Ưu và nhược điểm:

+ Ưu điểm:

Dễ thực hiện: Với tính đơn giản và chính xác của thuật toán, đây là một trong những phân loại đầu tiên mà một nhà khoa học dữ liệu mới sẽ học.

Điều chỉnh dễ dàng: Khi các mẫu đào tạo mới được thêm vào, thuật toán điều chỉnh để tính đến bất kỳ dữ liệu mới nào vì tất cả dữ liệu đào tạo được lưu trữ vào bộ nhớ.

Ít hyperparameters: KNN chỉ yêu cầu giá trị k và số liệu khoảng cách, thấp khi so sánh với các thuật toán học máy khác.

+ Nhược điểm:

Không mở rộng quy mô tốt

Không hoạt động tốt với dữ liệu nhiều chiều

Dễ bị overfitting

3. Thuật toán SVM- linear kernel

- Khái niệm:

SVM (Máy vector hỗ trợ) là một thuật toán **học có giám sát** dùng để giải quyết bài toán **phân loại (classification)** và **hồi quy (regression)**. Tuy nhiên, SVM được sử dụng phổ biến nhất trong **phân loại nhị phân**.

Mục tiêu của SVM là tìm một **siêu phẳng (hyperplane)** để **phân tách dữ liệu** thành các lớp sao cho khoảng cách từ siêu phẳng đến các điểm gần nhất (gọi là **support vectors**) là lớn nhất.

+ **Linear Kernel (Nhân tuyến tính)** là một trong những hàm nhân đơn giản nhất được sử dụng trong SVM. Nó được dùng khi dữ liệu có thể **phân tách tuyến tính** (linearly separable), tức là có thể tách được bằng một đường thẳng (trong 2D) hoặc một siêu phẳng (trong không gian nhiều chiều).

Hàm nhân tuyến tính có dạng:

$$K(x_i, x_j) = x_i^T x_j$$

Với:

- x_i, x_j là các vector đầu vào
- $x_i^T x_j$ là tích vô hướng của hai vector

Note: khi sử dụng **Linear Kernel**, mô hình SVM chỉ là một **hồi quy tuyến tính với ràng buộc tối ưu khoảng cách giữa các lớp**.

- Công thức toán học của SVM với Linear Kernel

SVM cố gắng tìm một **siêu phẳng** tối ưu có dạng:

$$w^T x + b = 0$$

Trong đó:

- w là vector trọng số
- x là vector đặc trưng
- b là bias

Bài toán tối ưu trong SVM tìm w và b sao cho khoảng cách giữa hai lớp là lớn nhất, tức là:

$$\min_{w,b} \frac{1}{2} ||w||^2$$

Với điều kiện ràng buộc:

$$y_i(w^T x_i + b) \geq 1, \quad \forall i$$

Trong đó $y_i \in \{-1, 1\}$ là nhãn của dữ liệu.

Nếu dữ liệu không hoàn toàn phân tách tuyến tính, ta dùng **Soft Margin SVM** với tham số C để điều chỉnh mức độ cho phép lỗi:

$$\min_{w,b,\xi} \frac{1}{2} ||w||^2 + C \sum \xi_i$$

Với ξ_i là biến slack để cho phép một số điểm dữ liệu nằm sai phía của siêu phẳng.

Linear Kernel hoạt động tốt khi:

- + Dữ liệu có thể phân tách tuyến tính hoặc gần tuyến tính.
- + Số lượng đặc trưng (features) lớn so với số lượng mẫu dữ liệu.

- **Ưu điểm:**

- + Nhanh hơn so với các kernel phi tuyến tính (Polynomial, RBF)
- + Hiệu quả với dữ liệu có số chiều cao
- + Dễ hiểu và dễ triển khai

- **Nhược điểm:**

- + Không hoạt động tốt với dữ liệu phi tuyến tính
- + Không linh hoạt như RBF Kernel
- **Linear Kernel SVM** là một lựa chọn mạnh mẽ khi dữ liệu có thể phân tách tuyến tính.
- Khi số chiều lớn, Linear Kernel vẫn có thể hoạt động tốt ngay cả khi dữ liệu không hoàn toàn tuyến tính.

4. SVM RBF kernel

RBF Kernel (Radial Basis Function Kernel) là một trong những hàm kernel phổ biến nhất của SVM, giúp mô hình có thể xử lý dữ liệu **không phân tách tuyến tính** bằng cách ánh xạ dữ liệu lên không gian cao hơn.

Hàm kernel RBF có dạng:

$$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$$

Trong đó:

- x_i, x_j là hai điểm dữ liệu
 - $\|x_i - x_j\|^2$ là bình phương khoảng cách Euclidean giữa hai điểm
 - γ (gamma) là một **tham số quan trọng**, kiểm soát mức độ ảnh hưởng của một điểm dữ liệu đến các điểm khác.
-

Vai trò của tham số γ trong RBF Kernel

- **γ lớn** \rightarrow Mỗi điểm dữ liệu chỉ ảnh hưởng đến vùng gần nó \rightarrow Mô hình có thể bị **overfitting**.
- **γ nhỏ** \rightarrow Các điểm dữ liệu ảnh hưởng đến nhiều điểm khác \rightarrow Mô hình có thể bị **underfitting**.

Note: Giá trị γ cần được tinh chỉnh bằng cách thử nghiệm nhiều giá trị khác nhau, thường dùng phương pháp **Grid Search** hoặc **Random Search** để tìm giá trị tối ưu.

4. Công thức toán học của SVM với RBF Kernel

SVM với RBF Kernel tìm nghiệm của bài toán tối ưu:

$$\min_{w,b,\xi} \frac{1}{2} \|w\|^2 + C \sum \xi_i$$

Với điều kiện ràng buộc:

$$y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i, \quad \forall i$$

Trong đó:

- C là tham số điều chỉnh độ phạt khi có điểm dữ liệu nằm sai phía siêu phẳng.
- $\phi(x)$ là ánh xạ dữ liệu từ không gian gốc lên không gian có số chiều cao hơn thông qua RBF Kernel.

Nên sử dụng RBF kernel khi:

- + Khi dữ liệu **không thể phân tách tuyến tính**.
- + Khi có mối quan hệ **phi tuyến tính** giữa các đặc trưng.

+ Khi số chiều của dữ liệu không quá lớn (nếu số chiều lớn, **Linear Kernel** có thể hoạt động tốt hơn).

- **Ưu điểm:**

- + Có khả năng xử lý dữ liệu **phi tuyến tính** tốt.
- + Phù hợp với nhiều dạng dữ liệu khác nhau.
- + Không yêu cầu chọn số chiều ánh xạ như Polynomial Kernel.

- **Nhược điểm:**

- + Cần tinh chỉnh tham số **C** và γ để tránh overfitting.
- + Tốn nhiều tài nguyên tính toán hơn so với Linear Kernel.

III. Chạy casestudy trong chapter 12:

a. Tiền xử lý dữ liệu

```
: df = pd.read_csv('diabetes.csv')
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
#   Column                Non-Null Count  Dtype
---  -
0   Pregnancies            768 non-null    int64
1   Glucose                768 non-null    int64
2   BloodPressure          768 non-null    int64
3   SkinThickness          768 non-null    int64
4   Insulin                768 non-null    int64
5   BMI                   768 non-null    float64
6   DiabetesPedigreeFunction 768 non-null    float64
7   Age                   768 non-null    int64
8   Outcome                768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

- Đọc dữ liệu từ file csv tên 'diabetes.csv' và chuyển nó thành một DataFrame.

- Hiện thị thông tin về các cột trong DataFrame, gồm:

- Số lượng dòng & cột
- Tên cột
- Kiểu dữ liệu của từng cột (int64, float64, object, v.v.)
- Số lượng giá trị không bị thiếu (non-null) của từng cột

```
print("Nulls")
print("=====")
print(df.isnull().sum())

Nulls
=====
Pregnancies            0
Glucose                0
BloodPressure          0
SkinThickness          0
Insulin                0
BMI                   0
DiabetesPedigreeFunction 0
Age                   0
Outcome                0
dtype: int64
```

- print("Nulls")

- In tiêu đề "Nulls" để báo hiệu phần kiểm tra dữ liệu bị thiếu.

- df.isnull().sum()

- `df.isnull()`: Trả về DataFrame boolean, trong đó:
 - True (1) nếu giá trị bị thiếu (NaN).
 - False (0) nếu có giá trị.
- `sum()`: Tính tổng số giá trị bị thiếu của từng cột.

```
[24]: #Dem so Luong phan tu 0
print("0s")
print("==")
print(df.eq(0).sum())

0s
==
Pregnancies      111
Glucose           5
BloodPressure     35
SkinThickness    227
Insulin          374
BMI              11
DiabetesPedigreeFunction  0
Age              0
Outcome          500
dtype: int64
```

- `df.eq(0).sum()`

- `df.eq(0)`: Trả về một DataFrame Boolean, trong đó:
 - True (1) nếu giá trị trong bảng bằng 0.
 - False (0) nếu khác 0.
- `sum()`: Đếm số lượng giá trị 0 trong từng cột.

```
[26]: df[['Glucose', 'BloodPressure', 'SkinThickness',
        'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age']] = \
        df[['Glucose', 'BloodPressure', 'SkinThickness',
        'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age']].replace(0, np.NaN)

[28]: df.fillna(df.mean(), inplace = True)

[30]: print(df.eq(0).sum())

Pregnancies      111
Glucose           0
BloodPressure     0
SkinThickness     0
Insulin           0
BMI               0
DiabetesPedigreeFunction  0
Age              0
Outcome          500
dtype: int64
```

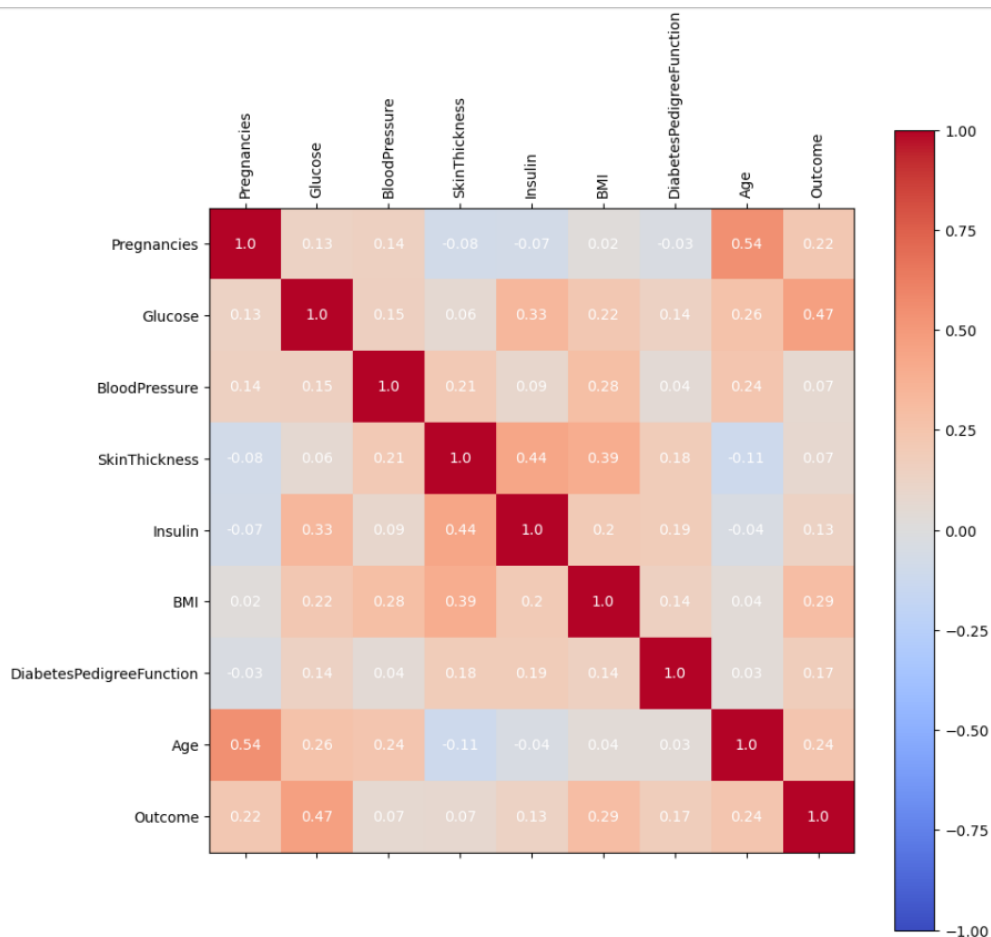
- Thay thế 0 bằng NaN để dễ xử lý dữ liệu bị thiếu.
- Điền các giá trị bị thiếu (NaN) bằng trung bình (mean) của từng cột.

- inplace=True: Thay đổi trực tiếp df, không cần gán lại.
- df.eq(0): Tạo DataFrame Boolean với giá trị True (1) nếu bằng 0, False (0) nếu khác 0.
- .sum(): Đếm số lượng giá trị bằng 0 trong từng cột.

```
[9]: corr = df.corr()
```

```
[11]:
```

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(10, 10))
cax = ax.matshow(corr, cmap='coolwarm', vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = np.arange(0, len(df.columns), 1)
ax.set_xticks(ticks)
ax.set_xticklabels(df.columns)
plt.xticks(rotation = 90)
ax.set_yticklabels(df.columns)
ax.set_yticks(ticks)
#---print the correlation factor---
for i in range(df.shape[1]):
    for j in range(9):
        text = ax.text(j, i, round(corr.iloc[i][j], 2),
                        ha="center", va="center", color="w")
plt.show()
```



- `corr = df.corr()`

+ Mục đích: Tính ma trận tương quan (correlation matrix) giữa các cột trong DataFrame `df`.

+ Kết quả: Lưu vào biến `corr` dưới dạng một DataFrame chứa hệ số tương quan giữa các cột.

- `matplotlib.pyplot`: Dùng để vẽ biểu đồ.

- `numpy`: Dùng để tạo mảng số (cho trục `xticks`, `yticks`).

- `fig, ax = plt.subplots(figsize=(10, 10))`

+ `plt.subplots(figsize=(10,10))`: Tạo một figure với kích thước 10x10

+ `ax`: Trục để vẽ ma trận tương quan.

- `cax = ax.matshow(corr, cmap='coolwarm', vmin=-1, vmax=1)`

+ ax.matshow(corr, cmap='coolwarm', vmin=-1, vmax=1):

- Vẽ ma trận tương quan bằng matshow().
- Màu coolwarm: Đỏ (hệ số tương quan âm), Xanh (hệ số tương quan dương).
- vmin=-1, vmax=1: Định nghĩa giá trị nhỏ nhất và lớn nhất cho màu sắc.

- fig.colorbar(cax)

+ colorbar(cax): Hiển thị thanh màu để dễ đọc giá trị.

- np.arange(0, len(df.columns), 1): Tạo danh sách số thứ tự cho trục.

- set_xticks(ticks): Đặt vị trí nhãn trên trục X.

- set_xticklabels(df.columns): Đặt tên cột cho trục X.

- plt.xticks(rotation=90): Xoay nhãn X 90 độ.

- set_yticks(ticks) và set_yticklabels(df.columns): Đặt tên cho trục Y

```
for i in range(df.shape[1]):  
    for j in range(9):  
        text = ax.text(j, i, round(corr.iloc[i][j],2),  
                        ha="center", va="center", color="w")
```

- Duyệt qua từng ô trong ma trận.
- Hiển thị giá trị hệ số tương quan (round(corr.iloc[i][j],2)) tại mỗi ô.
- Căn chỉnh ha="center", va="center" để giá trị nằm giữa ô.
- Màu chữ color="w" (màu trắng).

- plt.show(): hiển thị hình ảnh lên màn hình

```
: #Khai bao thu vien seaborn de ve bieu do
```

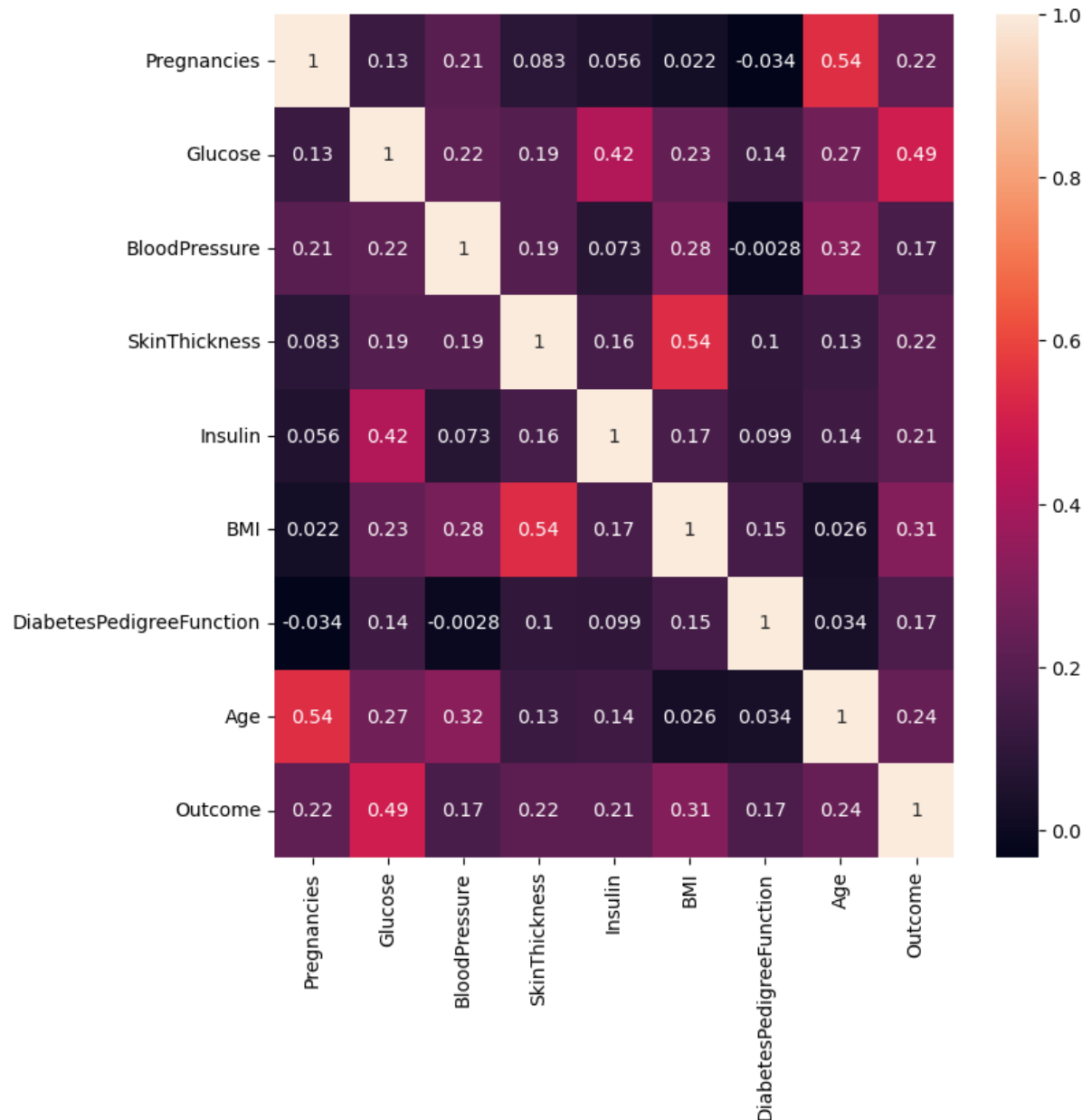
```
import seaborn as sns
```

```
sns.heatmap(df.corr(),annot=True)
```

```
#---get a reference to the current figure and set its size--
```

```
fig = plt.gcf()
```

```
fig.set_size_inches(8,8)
```



- `df.corr()`:
 - Tính hệ số tương quan giữa các cột trong `df`.
 - Hệ số tương quan (Pearson) có giá trị từ -1 đến 1:
 - 1: Tương quan dương hoàn hảo (càng tăng thì càng tăng).
 - -1: Tương quan âm hoàn hảo (càng tăng thì càng giảm).

- 0: Không có mối quan hệ.
- sns.heatmap():
 - Hiển thị ma trận tương quan dưới dạng bản đồ nhiệt.
 - annot=True: Hiển thị giá trị số trong từng ô.
- plt.gcf(): Lấy đối tượng figure hiện tại.
- set_size_inches(8,8):
 - Đặt kích thước biểu đồ 8x8 inches để hiển thị rõ hơn.

```
print(df.corr().nlargest(4, 'Outcome').index)
```

```
Index(['Outcome', 'Glucose', 'BMI', 'Age'], dtype='object')
```

Tìm 4 biến có tương quan cao nhất với Outcome để chọn đặc trưng quan trọng cho mô hình.

```
: print(df.corr().nlargest(4, 'Outcome').values[:,8])
```

```
[1.          0.49292767  0.31192439  0.23835598]
```

```
:
```

```
:
```

```
:
```

```
:
```

```
:
```

```
:
```


Trích xuất giá trị tương quan của 4 biến có tương quan cao nhất với Outcome.

b. Đánh giá thuật toán

Linear Regression

```
[56]: from sklearn import linear_model
      from sklearn.model_selection import cross_val_score
      #---features---
      X = df[['Glucose', 'BMI', 'Age']]
      #---label---
      y = df.iloc[:,8]
      log_regress = linear_model.LogisticRegression()
      log_regress_score = cross_val_score(log_regress, X, y, cv=10,
      scoring='accuracy').mean()
      print(log_regress_score)

0.7669856459330144
```

```
[ ]:
[ ]:
[ ]:
```

- linear_model: Chứa thuật toán hồi quy logistic.
- cross_val_score: Dùng để đánh giá mô hình bằng cross-validation (cv).
- Chọn 3 đặc trưng quan trọng nhất (Glucose, BMI, Age) từ dataset.
- Đây là các biến có tương quan cao nhất với Outcome.
- Chọn cột thứ 8 của df làm nhãn (Outcome).
- Cột này có giá trị 0 (không tiểu đường) hoặc 1 (có tiểu đường).
- Tạo một mô hình Hồi quy Logistic (Logistic Regression).
 - cross_val_score():
 - Huấn luyện & đánh giá mô hình bằng 10-Fold Cross Validation (cv=10).
 - Thước đo đánh giá: accuracy (độ chính xác).
 - Tính trung bình của 10 lần đánh giá.
- In ra độ chính xác trung bình sau khi đánh giá.
- Mô hình hồi quy logistic đạt 78% độ chính xác.

```
result = []
result.append(log_regress_score)
```

Lưu kết quả lại để so sánh sau

K-Nearest Neighbors

```
[60]: # K-Nearest Neighbors

[64]: from sklearn.neighbors import KNeighborsClassifier
      #---empty list that will hold cv (cross-validates) scores--
      cv_scores = []
      #---number of folds--
      folds = 10
      #---creating odd list of K for KNN--
      ks = list(range(1,int(len(X) * ((folds - 1)/folds)), 2))
      #---perform k-fold cross validation--
      for k in ks:
          knn = KNeighborsClassifier(n_neighbors=k)
          score = cross_val_score(knn, X, y, cv=folds, scoring='accuracy').mean()
          cv_scores.append(score)
      #---get the maximum score--
      knn_score = max(cv_scores)
      #---find the optimal k that gives the highest score--
      optimal_k = ks[cv_scores.index(knn_score)]
      print(f"The optimal number of neighbors is {optimal_k}")
      print(knn_score)
      result.append(knn_score)

The optimal number of neighbors is 19
0.7721462747778537
```

```
[ ]:
```

- Dùng KNeighborsClassifier để xây dựng mô hình KNN.
- Tạo Danh sách rỗng để lưu độ chính xác của KNN với từng giá trị k.
- Dùng 10-Fold Cross-Validation để đánh giá mô hình.
- Chỉ chọn số lẻ (2, 4, 6, ...) để tránh trường hợp đồng số phiếu bầu khi phân loại.

- Giới hạn k theo $\text{len}(X) * ((\text{folds} - 1) / \text{folds})$ để phù hợp với số lượng mẫu huấn luyện.
- Lặp qua từng giá trị k.
- Huấn luyện và đánh giá mô hình bằng Cross-Validation (cv=10).
- Lưu độ chính xác trung bình vào cv_scores.
- Tìm giá trị k có độ chính xác cao nhất ($\max(\text{cv_scores})$).
- Lấy k tương ứng từ danh sách ks.

Kết luận: KNN hoạt động tốt với giá trị k = 19

Độ chính xác là: 0.7721462747778537

Thuật toán SVM

1. SVM linear

```
[66]: # Support Vector Machines (SVM)–Linear and RBF Kernels

[68]: from sklearn import svm
      linear_svm = svm.SVC(kernel='linear')
      linear_svm_score = cross_val_score(linear_svm, X, y, cv=10, scoring='accuracy').mean()
      print(linear_svm_score)
      result.append(linear_svm_score)

0.7656527682843473
```

```
[ ]:
```

```
[ ]:
```

- Dùng svm.SVC để xây dựng mô hình Support Vector Machine (SVM).
- Dùng kernel='linear' để áp dụng SVM tuyến tính.
- Hữu ích khi dữ liệu có thể phân tách tuyến tính (ví dụ: có thể vẽ đường thẳng để phân biệt giữa hai lớp).
- Huấn luyện & đánh giá mô hình bằng 10-Fold Cross-Validation (cv=10).

- Sử dụng "accuracy" làm thước đo đánh giá.
- Tính trung bình độ chính xác qua 10 lần đánh giá
- Lưu kết quả vào mảng Result để so sánh

Kết luận: SVM tuyến tính đạt độ chính xác 76.57%.

2. SVM RBF

```
[70]: rbf = svm.SVC(kernel='rbf')
      rbf_score = cross_val_score(rbf, X, y, cv=10, scoring='accuracy').mean()
      print(rbf_score)
      result.append(rbf_score)
```

0.765704032809296

[]:

- Dùng kernel='rbf' để áp dụng SVM với Radial Basis Function (RBF).
- Phù hợp với dữ liệu phi tuyến tính (dữ liệu không thể phân tách bằng đường thẳng).
- Huấn luyện & đánh giá mô hình bằng 10-Fold Cross-Validation (cv=10).
- Sử dụng "accuracy" làm thước đo đánh giá.
- Tính trung bình độ chính xác qua 10 lần đánh giá

Kết quả: độ chính xác 76.57%

```
[72]: algorithms = ["Logistic Regression", "K Nearest Neighbors", "SVM Linear Kernel", "SVM RBF Kernel"]
cv_mean = pd.DataFrame(result, index = algorithms)
cv_mean.columns=["Accuracy"]
cv_mean.sort_values(by="Accuracy", ascending=False)
```

```
[72]:
```

	Accuracy
K Nearest Neighbors	0.772146
Logistic Regression	0.766986
SVM RBF Kernel	0.765704
SVM Linear Kernel	0.765653

```
[ ]:
```

Lưu tên của các thuật toán tương ứng với giá trị độ chính xác (result)

Chuyển danh sách result thành DataFrame, trong đó:

- Chỉ mục (index) là tên các thuật toán.
- Cột Accuracy chứa giá trị độ chính xác.

Sắp xếp mô hình theo độ chính xác từ cao đến thấp.

Kết quả KNN đạt độ chính xác cao nhất

```
[74]: knn = KNeighborsClassifier(n_neighbors=19)
knn.fit(X, y)
```

```
[74]:
```

▼ KNeighborsClassifier ⓘ ?

KNeighborsClassifier(n_neighbors=19)

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

-
- Dùng thuật toán K-Nearest Neighbors (KNN).
 - Thiết lập số láng giềng (n_neighbors=19).
 - Giá trị k=19 có thể được chọn dựa trên Cross-Validation trước đó.

- Huấn luyện mô hình với tập dữ liệu X (features) và y (labels).
- Sau khi huấn luyện, mô hình có thể dự đoán nhãn mới.

```
[76]: import pickle
      #---save the model to disk--
      filename = 'diabetes.sav'
      #---write to the file using write and binary mode--
      pickle.dump(knn, open(filename, 'wb'))
```

```
[ ]:
```

```
[ ]:
```

- pickle giúp lưu trữ (serialize) và nạp lại (deserialize) mô hình Machine Learning.
- Dữ liệu được lưu dưới dạng nhị phân (.sav hoặc .pkl).
- Mô hình KNN sẽ được lưu vào file diabetes.sav.
- Có thể đặt tên khác như knn_model.pkl nếu cần.
- pickle.dump(): Lưu mô hình knn vào file.
- Chế độ 'wb': Ghi file ở dạng nhị phân.
- Sau khi lưu, mô hình có thể được nạp lại để sử dụng mà không cần huấn luyện lại

```
[78]: #---load the model from disk--
      loaded_model = pickle.load(open(filename, 'rb'))
```

```
[ ]:
```

- Nạp lại mô hình đã lưu (diabetes.sav) từ ổ đĩa.

- Sau khi tải, có thể sử dụng mô hình để dự đoán mà không cần huấn luyện lại

```
[82]: Glucose = 65
      BMI = 70
      Age = 50
      prediction = loaded_model.predict([[Glucose, BMI, Age]])
      print(prediction)
      if (prediction[0]==0):
          print("Non-diabetic")
      else:
          print("Diabetic")

[0]
Non-diabetic
```

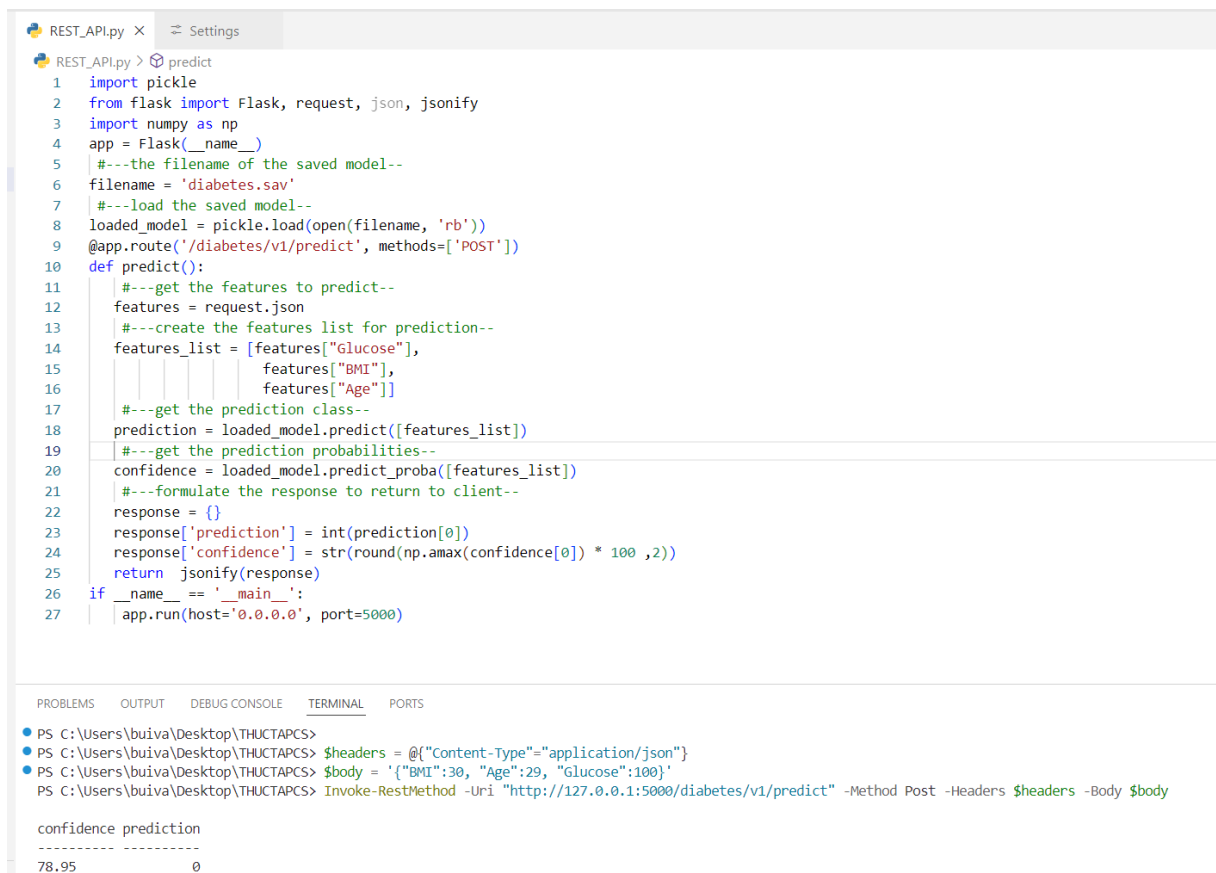
- Sử dụng mô hình loaded_model (KNN) để dự đoán trên dữ liệu mới.
- Kết quả dự đoán là 0 (Không tiểu đường).

```
[84]: proba = loaded_model.predict_proba([[Glucose, BMI, Age]])
      print(proba)
      print("Confidence: " + str(round(np.amax(proba[0]) * 100 ,2)) + "%")

[[0.94736842 0.05263158]]
Confidence: 94.74%
```

- predict_proba() trả về xác suất dự đoán cho mỗi lớp (0 hoặc 1).
- proba chứa một mảng có hai giá trị:
 - proba[0][0]: Xác suất không mắc tiểu đường (0): 94.73%.
 - proba[0][1]: Xác suất bị tiểu đường (1): 5.27%.
- np.amax(proba[0]): Lấy giá trị xác suất cao nhất.
- Nhân với 100 và làm tròn 2 chữ số thập phân.
- Hiện thị độ tin cậy của mô hình cho quyết định cuối cùng là: 94.74%.

c. Deploying the Model



```
REST_API.py x Settings
REST_API.py > predict
1 import pickle
2 from flask import Flask, request, jsonify
3 import numpy as np
4 app = Flask(__name__)
5 #---the filename of the saved model---
6 filename = 'diabetes.sav'
7 #---load the saved model---
8 loaded_model = pickle.load(open(filename, 'rb'))
9 @app.route('/diabetes/v1/predict', methods=['POST'])
10 def predict():
11     #---get the features to predict---
12     features = request.json
13     #---create the features list for prediction---
14     features_list = [features["Glucose"],
15                     features["BMI"],
16                     features["Age"]]
17     #---get the prediction class---
18     prediction = loaded_model.predict([features_list])
19     #---get the prediction probabilities---
20     confidence = loaded_model.predict_proba([features_list])
21     #---formulate the response to return to client---
22     response = {}
23     response['prediction'] = int(prediction[0])
24     response['confidence'] = str(round(np.amax(confidence[0]) * 100 ,2))
25     return jsonify(response)
26 if __name__ == '__main__':
27     app.run(host='0.0.0.0', port=5000)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\buiva\Desktop\THUCTAPCS>
PS C:\Users\buiva\Desktop\THUCTAPCS> $headers = @{"Content-Type"="application/json"}
PS C:\Users\buiva\Desktop\THUCTAPCS> $body = '{"BMI":30, "Age":29, "Glucose":100}'
PS C:\Users\buiva\Desktop\THUCTAPCS> Invoke-RestMethod -Uri "http://127.0.0.1:5000/diabetes/v1/predict" -Method Post -Headers $headers -Body $body

confidence prediction
-----
78.95              0
```

Tạo file REST_API.py

Giải thích code:

- pickle: Dùng để lưu và tải model đã train.
- Flask: Framework giúp tạo API web.
- request: Lấy dữ liệu từ HTTP request (từ client gửi đến).
- jsonify: Chuyển kết quả thành JSON response.
- numpy: Hỗ trợ tính toán số học (ở đây dùng để tìm giá trị xác suất cao nhất).
- Flask(__name__): Khởi tạo ứng dụng Flask.
- diabetes.sav: File chứa model đã được huấn luyện.
- pickle.load(open(filename, 'rb')):
 - Mở file dưới dạng read binary (rb).

- Tải model đã lưu vào biến `loaded_model`.
- `@app.route('/diabetes/v1/predict', methods=['POST']):`
 - API có đường dẫn `/diabetes/v1/predict`.
 - Chỉ chấp nhận HTTP POST request.
 - + `def predict():` Định nghĩa hàm `predict()` để xử lý request.
- `request.json:`
 - Lấy dữ liệu từ request dưới dạng JSON.
 - Lưu vào biến `features` (dạng dictionary)
- Tạo danh sách đặc trưng (`features_list`) từ JSON đầu vào.
- Dự đoán kết quả dựa trên mô hình đã train.
 - + `loaded_model.predict([features_list])` nhận danh sách 3 giá trị đầu vào và trả về 0 (không bị tiểu đường) hoặc 1 (bị tiểu đường).
- Tính toán độ tin cậy của dự đoán
 - + `predict_proba()` trả về xác suất của từng lớp
 - + Lấy giá trị xác suất cao nhất (độ tự tin của mô hình) làm tròn đến 2 chữ số thập phân
- Tạo dictionary chứa:
 - + `prediction: 0 hoặc 1 (non-diabetic hoặc diabetic).`
 - + `confidence: Xác suất dự đoán.`
- Dùng `jsonify(response)` để trả kết quả dạng JSON.

Chạy API Flask

- `app.run()` khởi chạy Flask server trên port 5000.
- `host='0.0.0.0':` Chấp nhận request từ tất cả thiết bị trong mạng cục bộ.

Kết quả

```
• PS C:\Users\buiva\Desktop\THUCTAPCS>
• PS C:\Users\buiva\Desktop\THUCTAPCS> $headers = @{"Content-Type"="application/json"}
• PS C:\Users\buiva\Desktop\THUCTAPCS> $body = '{"BMI":30, "Age":29, "Glucose":100}'
PS C:\Users\buiva\Desktop\THUCTAPCS> Invoke-RestMethod -Uri "http://127.0.0.1:5000/diabetes/v1/predict" -Method Post -Headers $headers -Body $body

confidence prediction
-----
78.95                0
```

- \$headers = @{"Content-Type"="application/json"} → Định nghĩa headers cho request.

- \$body = '{"BMI":30, "Age":29, "Glucose":100}' → Chuyển JSON thành chuỗi.

- Invoke-RestMethod → Lệnh dùng để gửi request trong PowerShell.

- -Uri → Chỉ định đường dẫn API cần gọi.

(<http://127.0.0.1:5000/diabetes/v1/predict>) đã khởi tạo

- -Method Post → Chỉ định phương thức POST.

- -Headers \$headers → Gửi headers dưới dạng dictionary.

- -Body \$body → Gửi dữ liệu JSON làm request body.

- Kết quả trả về:

+ confidence: 78.95

+prediction: 0

d. Creating the Client Application to Use the Model

```
import json
import requests
def predict_diabetes(BMI, Age, Glucose):
    url = 'http://127.0.0.1:5000/diabetes/v1/predict'
    data = {"BMI":BMI, "Age":Age, "Glucose":Glucose}
    data_json = json.dumps(data)
    headers = {'Content-type':'application/json'}
    response = requests.post(url, data=data_json, headers=headers)
    result = json.loads(response.text)
    return result
if __name__ == "__main__":
    predictions = predict_diabetes(30,40,100)
    print("Diabetic" if predictions["prediction"] == 1 else "Not Diabetic")
    print("Confidence: " + predictions["confidence"] + "%")
```

✓ 1.2s

Not Diabetic
Confidence: 68.42%

- json: Dùng để chuyển đổi dữ liệu giữa Python dictionary và JSON string.
- requests: Thư viện dùng để gửi HTTP requests (ở đây là POST request đến API).
- url = 'http://127.0.0.1:5000/diabetes/v1/predict': Xác định URL của API (Flask backend) mà client sẽ gửi dữ liệu để dự đoán bệnh tiểu đường.
- data: Dictionary chứa dữ liệu đầu vào gồm BMI, Age, Glucose.
- json.dumps(data): Chuyển dictionary data sang chuỗi JSON (vì API yêu cầu định dạng JSON).
- headers = {'Content-type': 'application/json'}: Xác định loại dữ liệu gửi là JSON.
- requests.post(url, data=data_json, headers=headers):
 - Gửi POST request đến API tại url.
 - Dữ liệu data_json (JSON) được gửi trong request body.
 - headers đảm bảo server hiểu dữ liệu này là JSON.
- response.text: Lấy nội dung phản hồi từ API (thường là chuỗi JSON).

- `json.loads(response.text)`: Chuyển phản hồi JSON thành dictionary Python để xử lý.
- `return result`: Trả về kết quả dự đoán từ API.
- Gọi hàm `predict_diabetes(30, 40, 100)` để dự đoán bệnh tiểu đường cho một bệnh nhân có:
 - BMI = 30
 - Age = 40
 - Glucose = 100

Kết quả: Not Diabetic

Confidence: 68.42% (Mức độ tự tin (%) của mô hình với kết quả dự đoán.)

Tạo file: `Predict_Diabetes.py`

```

21 import json
22 import requests
23
24 def predict_diabetes(BMI, Age, Glucose):
25     url = 'http://127.0.0.1:5000/diabetes/v1/predict'
26     data = {"BMI": float(BMI), "Age": int(Age), "Glucose": int(Glucose)}
27     headers = {'Content-Type': 'application/json'}
28
29     response = requests.post(url, json=data, headers=headers)
30
31     if response.status_code == 200:
32         return response.json()
33     else:
34         return {"error": "Request failed"}
35
36 if __name__ == "__main__":
37     BMI = input("BMI? ")
38     Age = input("Age? ")
39     Glucose = input("Glucose? ")
40
41     predictions = predict_diabetes(BMI, Age, Glucose)
42
43     if "error" in predictions:
44         print("Error:", predictions["error"])
45     else:
46         print("Diabetic" if predictions["prediction"] == 1 else "Not Diabetic")
47         print("Confidence: " + predictions["confidence"] + "%")
48

```

PROBLEMS 35 OUTPUT DEBUG CONSOLE TERMINAL PORTS JUPYTER

```

PS C:\Users\buiva\Desktop\THUCTAPCS> python Predict_Diabetes.py
BMI? 55
Age? 29
Glucose? 120
Not Diabetic
Confidence: 52.63%

```

- Xác định URL API mà chương trình sẽ gửi yêu cầu (http://127.0.0.1:5000/diabetes/v1/predict).
- API Flask này đã được triển khai trên cổng 5000 của localhost.
- Chuyển đổi BMI sang kiểu float (vì BMI có thể có số thập phân).
- Chuyển đổi Age và Glucose sang kiểu int.
- Điều này giúp đảm bảo dữ liệu có đúng kiểu khi gửi lên server.
- headers = {'Content-Type': 'application/json'}: Xác định rằng dữ liệu gửi lên là JSON.
- requests.post(url, json=data, headers=headers):
 - Gửi POST request đến API Flask.
 - json=data: requests tự động chuyển đổi dictionary data thành JSON.

- headers=headers: Đảm bảo API biết rằng dữ liệu được gửi dưới dạng JSON.
-
- response.status_code == 200: Kiểm tra nếu API phản hồi thành công (200 OK).
 - response.json(): Chuyển đổi phản hồi JSON thành dictionary Python.
 - Nếu lỗi, trả về {"error": "Request failed"}.
 - Trong hàm main yêu cầu:
 - + người dùng nhập vào BMI, Age, Glucose từ bàn phím.
 - Gửi dữ liệu và hiển thị kết quả
 - + Gọi hàm predict_diabetes() để gửi dữ liệu đến API Flask và nhận kết quả.

Kết quả

```
PS C:\Users\buiva\Desktop\THUCTAPCS> python Predict_Diabetes.py
BMI? 55
Age? 29
Glucose? 120
Not Diabetic
Confidence: 52.63%
PS C:\Users\buiva\Desktop\THUCTAPCS> █
```

Với dữ liệu BMI = 55, Age = 29, Glucose = 120 mô hình dự đoán kết quả là không có bệnh, với độ tự tin là 52.63%

END