**Multicollinearity (Đa cộng tuyến) => Double Errors**

Trong mô hình hồi quy, nếu các biến độc lập có quan hệ chặt với nhau, các biến độc lập có mối quan hệ tuyến tính, nghĩa là các biến độc lập có tương quan chặt, mạnh với nhau thì sẽ có hiện tượng đa cộng tuyến, đó là hiện tượng các biến độc lập trong mô hình phụ thuộc lẫn nhau và thể hiện được dưới dạng hàm số. Ví dụ có hai biến độc lập A và B, khi A tăng thì B tăng, A giảm thì B giảm…. thì đó là một dấu hiệu của đa cộng tuyến. Nói một cách khác là hai biến độc lập có quan hệ rất mạnh với nhau, đúng ra hai biến này nó phải là 1 biến nhưng thực tế trong mô hình nhà nghiên cứu lại tách làm 2 biến. Hiện tượng đa cộng tuyến vi phạm giả định của mô hình hồi qui tuyến tính cổ điển là các biến độc lập không có mối quan hệ tuyến tính với nhau.

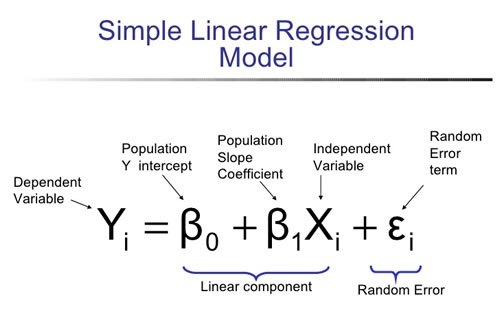
Đa cộng tuyến hoàn hảo:

|  |  |  |
| --- | --- | --- |
| **X2** | **X3** | **X4** |
| 10 | 50 | 52 |
| 15 | 75 | 75 |
| 18 | 90 | 97 |
| 24 | 120 | 129 |

**Nhận Xét:** **X2** và **X3** có mối quan hệ tuyến tính chính xác: **X3** = 5**X2**

Giả sử chúng ta ước lượng hàm tiêu dùng. **Y** = tiêu dùng, **X2** = thu nhập và **X3** = của cải

* **Y** = β1 + β2**X2** + β3**X3 +**
* **X3** = 5**X2**
* **Y** = β1 + β2**X2** + β35**X2 +**
* **Y** = β1 + (β2 + 5β3)**X3 +**



**Nguồn Gốc Của Multicollinearity**

Do phương pháp thu thập dữ liệu, các giá trị của các biến độc lập phụ thuộc lẫn nhau trong mẫu, nhưng không phụ thuộc lẫn nhau trong tổng thể.

Ví dụ: người có thu nhập cao hơn khuynh hướng sẽ có nhiều của cải hơn. Điều này có thể đúng với mẫu mà không đúng với tổng thể

Trong tổng thể sẽ có các quan sát về các cá nhân có thu nhập cao nhưng không có nhiều của cải và ngược lại.

**Cách phát hiện trường hợp đa cộng tuyến**

Có 2 cách: dựa vào hệ số phóng đại phương sai VIF, hoặc dựa vào ma trận hệ số tương quan. Tuy nhiên cách dùng ma trận hệ số tương quan ít được sử dụng, chủ yếu sửa dụng cách nhận xét chỉ số VIF.

**Cách 2:** Nhận dạng **Multicollinearity** dựa vào hệ số tương quan,có hay không tương quan tuyến tính mạnh giữa các biến độc lập. Cách làm: xây dựng ma trận hệ số tương quan cặp giữa các biến độc lập và quan sát để nhận diện độ mạnh của các tương quan giữa từng cặp biến số độc lập. Cũng có thể nhìn vào kết quả hồi quy, ta thấy R2 cao( tầm trên 0.8) và thống kê t thấp. Tuy nhiên như đã nói thì ít khi sử dụng cách hai này. Vì nó dựa vào phán đoán chủ quan hơn là công thức như cách 1.

**Hệ quả của đa cộng tuyến:**

* Đa cộng tuyến hoàn hảo:
* Không ước lượng được mô hình
* Gặp các cảnh báo trong khi xây dựng model:
  + “Matrix singular”: ma trận khác thường mà máy tính không thể thực hiện được khi ước lượng các hệ số hồi qui
  + “Exact collinearity encounted”: trường hợp đa cộng tuyến hoàn hảo (chính xác)
* Đa cộng tuyến không hoàn hảo:

**Cách giải quyết đa cộng tuyến:**

Cách 1: Principal Component Analysis (PCA) => Demention Reduction (Decomposition)

Dùng kiến thức Eigenvector và Eigenvalue



Cách 2: Partial Least Square Regression (PLSQ) =>

**Feature Engineering**

**Giới thiệu:**

Khi làm việc với các bài toán Machine Learning thực tế, nhìn chung chúng ta chỉ có được dữ liệu thô (raw) chưa qua chỉnh sửa, chọn lọc. Chúng ta cần phải tìm một phép biến đổi để loại ra những dữ liệu nhiễu (noise), và để đưa dữ liệu thô với số chiều khác nhau về cùng một chuẩn (cùng là các vector hoặc ma trận). Dữ liệu chuẩn mới này phải đảm bảo giữ được những thông tin đặc trưng (features) cho dữ liệu thô ban đầu. Không những thế, tùy vào từng bài toán, ta cần thiết kế những phép biến đổi để có những features phù hợp. Quá trình quan trọng này được gọi là Feature Extraction, hoặc Feature Engineering, tiếng Việt gọi là trích chọn đặc trưng.

Trích một câu nói của thầy Andrew Ng (Giáo sư tại Standford) - Nguồn Feature Engineering - Wiki:

*Coming up with features is difficult, time-consuming, requires expert knowledge. “Applied machine learning” is basically feature engineering.*

*Tạm dịch:*

*Làm việc với features là một việc không dễ dàng, tốn nhiều thời gian, cần rất nhiều kiến thức, áp dụng máy học cơ bản chỉ là feature engineering.*

**Mô hình chung cho các bài toán Machine Learning**

Phần lớn các bài toán Machine Learning có thể được thể hiện trong hình vẽ dưới đây:



*Hình 1: Mô hình chung cho các bài toán Machine Learning.*

**TRAINING PHASE**

Có hai khối có nền màu xanh lục chúng ta cần phải thiết kế:

**Feature Extractor**

**ĐẦU RA**

Mục đích của Feature Engineering là tạo ra một Feature Extractor biến dữ liệu thô ban đầu thành dữ liệu phù hợp với từng mục đích khác nhau.

**ĐẦU VÀO**

* **Raw training input**: Raw input là tất cả các thông tin ta biết về dữ liệu. Ví dụ: với ảnh thì là giá trị của từng pixel; với văn bản thì là từng từ, từng câu; với file âm thanh thì nó là một đoạn tín hiệu; với cơ sở dữ liệu Iris thì nó là độ dài các cánh hoa và đài hoa,…Dữ liệu thô này thường không ở dạng vector, không có số chiều như nhau. Thậm chí có thể có số chiều như nhau nhưng số chiều quá lớn, như một bức ảnh màu 1000 pixel **x** 1000 pixel thì số elements đã là **3x**(3 vì ảnh màu thường có 3 channels: Red, Green, Blue). Đây là một con số quá lớn, không lợi cho lưu trữ và tính toán.
* **(Optional) output của training set**. Trong các bài toán Unsupervised learning, ta không biết output nên hiển nhiên sẽ không có đầu vào này. Trong các bài toán Supervised learning, có khi dữ liệu này cũng không được sử dụng. Ví dụ: nếu raw input đã có cùng số chiều rồi nhưng số chiều quá lớn, ta muốn giảm số chiều của nó thì cách đơn giản nhất là chiếu vector đó xuống một không gian có số chiều nhỏ hơn bằng cách lấy một ma trận ngẫu nhiên nhân với nó. Ma trận này thường là ma trận béo (số hàng ít hơn số cột, tiếng Anh - fat matrices) để đảm bảo số chiều thu được nhỏ hơn số chiều ban đầu. Việc làm này mặc dù làm mất đi thông tin, trong nhiều trường hợp vẫn mang lại hiệu quả vì đã giảm được lượng tính toán ở phần sau. Đôi khi ma trận chiếu không phải là ngẫu nhiên mà có thể được học dựa trên toàn bộ raw input, ta sẽ có bài toán tìm ma trận chiếu để lượng thông tin mất đi là ít nhất. Trong nhiều trường hợp, dữ liệu output của training set cũng được sử dụng để tạo ra Feature Extractor. Ví dụ: trong bài toán classification, ta không quan tâm nhiều đến việc mất thông tin hay không, ta chỉ quan tâm đến việc những thông tin còn lại có đặc trưng cho từng class hay không. Ví dụ, dữ liệu thô là các hình vuông và hình tam giác có màu đỏ và xanh. Trong bài toán phân loại đa giác, các output là tam giác và vuông, thì ta không quan tâm tới màu sắc mà chỉ quan tâm tới số cạnh của đa giác. Ngược lại, trong bài toán phân loại màu, các class là xanh và đỏ, ta không quan tâm tới số cạnh mà chỉ quan tâm đến màu sắc thôi.
* **(Optional) Prior knowledge about data**: Đôi khi những giả thiết khác về dữ liệu cũng mang lại lợi ích. Ví dụ, trong bài toán classification, nếu ta biết dữ liệu là (gần như) linearly separable thì ta sẽ đi tìm một ma trận chiếu sao cho ở trong không gian mới, dữ liệu vẫn đảm bảo tính linearly separable, việc này thuận tiện hơn cho phần classification vì các thuật toán linear, nhìn chung, đơn giản hơn.

Sau khi học được feature extractor thì ta cũng sẽ thu được extracted features cho raw input data. Những extracted features này được dùng để huấn luyện các thuật toán Classification, Clustering, Regression… ở phía sau.

**Main Algorithms**

Khi có được extracted features rồi, chúng ta sử dụng những thông tin này cùng với (optional) training output và (optional) prior knowledge để tạo ra các mô hình phù hợp, điều mà chúng ta đã làm ở những bài trước.

Chú ý: Trong một số thuật toán cao cấp hơn, việc huấn luyện feature extractor và main algorithm được thực hiện cùng lúc với nhau chứ không phải từng bước như trên.

Một điểm rất quan trọng: khi xây dựng bộ feature extractor và main algorithms, chúng ta không được sử dụng bất kỳ thông tin nào trong tập test data. Ta phải giả sử rằng những thông tin trong test data chưa được nhìn thấy bao giờ. Nếu sử dụng thêm thông tin về test data thì rõ ràng ta đã ăn gian! Tôi từng đánh giá các bài báo khoa học quốc tế, rất nhiều tác giả xây dựng mô hình dùng cả dữ liệu test data, sau đó lại dùng chính mô hình đó để kiểm tra trên test data đó. Việc ăn gian này là lỗi rất nặng và hiển nhiên những bài báo đó bị từ chối (reject).

**TESTING PHASE**

Bước này đơn giản hơn nhiều. Với raw input mới, ta sử dụng feature extractor đã tạo được ở trên (tất nhiên không được sử dụng output của nó vì output là cái ta đang đi tìm) để tạo ra feature vector tương ứng. Feature vector được đưa vào main algorithm đã được học ở training phase để dự đoán output.

**Feature Scaling and Normalization:**

Các điểm dữ liệu đôi khi được đo đạc với những đơn vị khác nhau, m và feet chẳng hạn. Hoặc có hai thành phần (của vector dữ liệu) chênh lệch nhau quá lớn, một thành phần có khoảng giá trị từ 0 đến 1000, thành phần kia chỉ có khoảng giá trị từ 0 đến 1 chẳng hạn. Lúc này, chúng ta cần chuẩn hóa dữ liệu trước khi thực hiện các bước tiếp theo.

**Rescaling**

Phương pháp đơn giản nhất là đưa tất cả các thành phần về cùng một khoảng, [0,1] hoặc [-1,1]. Nếu muốn đưa một thành phần (feature) về khoảng [0,1] công thức sẽ là:

Trong đó, **x** là giá trị ban đầu, **x’** là giá trị sau khi chuẩn hóa. **min(x), max(x)** được tính trên toàn bộ dữ liệu training data ở cùng một thành phần. Việc này được thực hiện trên từng thành phần của vector dữ liệu **x**.

Ở Python, thư viện sklearn cho phép thực hiện scale khoảng [0, 1] đơn giản với hàm: min\_max\_scaler()

*#thêm thư viện xử lý ma trận***import** numpy **as** np  
*#thư viện sklearn để lấy hàm min\_max\_scaler***from** sklearn **import** preprocessing  
*#Tạo một ma trận tên X\_Train*X\_train = np.array([[1.,-1.,2.],[2.,0.,0.],[0.,1.,-1.]])  
*#gán biến min\_max\_scaler từ hàm MinMaxScaler() cho khoảng [0,1]  
#gán biến max\_abs\_scaler từ hàm MaxAbsScaler() cho khoảng [-1,1]*min\_max\_scaler = preprocessing.MinMaxScaler()  
max\_abs\_scaler = preprocessing.MinMaxScaler()  
*#Thực hiện chuyển đổi về khoảng [0,1] và [-1,1] bằng hàm fit\_transform*X\_train\_minmax = min\_max\_scaler.fit\_transform(X\_train)  
X\_train\_maxabs = max\_abs\_scaler.fit\_transform(X\_train)  
*#in ra màn hình ma trận sau khi scaled!*print(**"Ma trận ban đầu:\n{}\n\nMa trận sau khi scaling:\nĐối với khoảng [0,1]:\n{}\n\n""Đối với khoảng [-1,1]:\n{}"** .format(X\_train,X\_train\_minmax,X\_train\_maxabs))

Kết Quả:

Ma trận ban đầu:

[[ 1. -1. 2.]

[ 2. 0. 0.]

[ 0. 1. -1.]]

Ma trận sau khi scale:

Đối với khoảng [0,1]:

[[ 0.5 0. 1. ]

[ 1. 0.5 0.33333333]

[ 0. 1. 0. ]]

Đối với khoảng [-1,1]:

[[ 0.5 -1. 1. ]

[ 1. 0. 0. ]

[ 0. 1. -0.5]]

**Standardization**

Một phương pháp nữa cũng hay được sử dụng là giả sử mỗi thành phần đều có phân phối chuẩn với kỳ vọng là 0 và phương sai là 1. Khi đó, công thức chuẩn hóa sẽ là:

Trong đó: **,**  lần lượt là kỳ vọng và phương sai (standard deviation) của thành phần đó trên toàn bộ training data.

*#thêm thư viện xử lý ma trận***import** numpy **as** np  
*#thư viện sklearn để lấy hàm min\_max\_scaler***from** sklearn **import** preprocessing  
*#Tạo một ma trận tên X\_Train*X\_train = np.array([[1.,-1.,2.],[2.,0.,0.],[0.,1.,-1.]])  
*#gán biến scaler từ hàm StandardScaler() sau khi fit ma trận*scaler = preprocessing.StandardScaler().fit(X\_train)  
print(**"Ma trận ban đầu:\n{}\n\nMa trận sau khi scaling:\n{}"** .format(X\_train,scaler.transform(X\_train)))

Kết Quả:

Ma trận ban đầu:

[[ 1. -1. 2.]

[ 2. 0. 0.]

[ 0. 1. -1.]]

Ma trận sau khi scaling:

[[ 0. -1.22474487 1.33630621]

[ 1.22474487 0. -0.26726124]

[-1.22474487 1.22474487 -1.06904497]]

**Scaling to unit length**

Một lựa chọn khác nữa cũng được sử dụng rộng rãi là chuẩn hóa các thành phần của mỗi vector dữ liệu sao cho toàn bộ vector có độ lớn (**Euclid**, tức norm 2) bằng 1. Việc này có thể được thực hiện bằng: