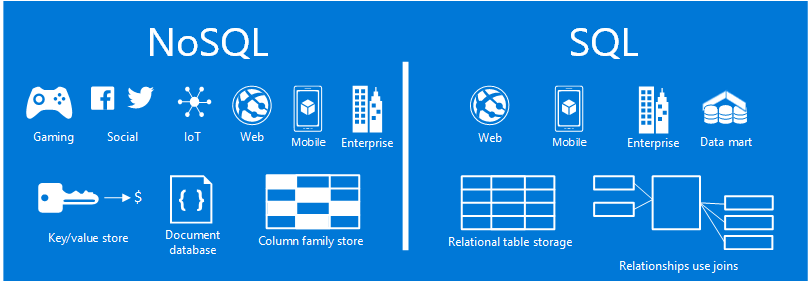
# NoSQL

Cơ Sở dữ liệu quan hệ được thiết kế cho những mô hình cần đảm bảo tính chặt chẽ và dữ liệu không quá lớn, trong khi các dịch vụ mạng xã hội lại có một lượng lớn dữ liệu và được cập nhật liên tục do số lượng người dùng quá nhiều. Do đó cơ sở dữ liệu NOSQL sinh ra để giải quyết các vấn đề mà RDBMS (A relational database management system) đã bộc lộ những yếu kém như: tốc độ thực thi, khả năng lưu trữ, các nghiệp vụ phức tạp (như phân trang, đánh chỉ mục …) Nhờ vậy giải pháp sử dụng cơ sở dữ liệu NOSQL sẽ mang lại một chi phí thấp hơn nếu so sánh với RDBMS truyền thống.



## Định Nghĩa

NoSQL là một xu hướng cơ sở dữ liệu mà không dùng dữ liệu quan hệ để quản lý dữ liệu trong lĩnh vực phần mềm. NOSQL có nghĩa là Non-Relational – không ràng buộc. Tuy nhên, thuật ngữ đó ít phổ biến hơn và ngày nay người ta thường dịch thành Not Only SQL – không chỉ SQL.

NOSQL được xem như thế hệ database kế tiếp của RDBMS, là một thế hệ cơ sở dữ liệu Non-relational (không ràng buộc), distributed (phân tán), open source (mã nguồn mở), horizontal scalable (khả năng mở rộng theo chiều ngang), có độ chịu tải, lỗi cao.

## Một Số Thuật Ngữ Trong NoSQL

* Tính ràng buộc (Relational): Thuật ngữ để mô tả tính ràng buộc giữa các bảng trong cơ sở dữ liệu quan hệ (MySQL, SqlSever, PostgreSQL…).
* Không ràng buộc (Non-relational): Dữ liệu sẽ không có các ràng buộc giữa các bảng nữa, mà dữ liệu sẽ ở dạng Json hoặc Bson.
* Khả năng mở rộng (High Scalability): Khi hệ thống lớn lên, ta có thể bổ sung thêm các Nodes mới, các Sever mới để chia tải hoặc chia dữ liệu. Để hệ thống không bị quá tải.
* Khả năng mở rộng theo chiều dọc (Vertical Scalable / Scale Up): Là việc nâng cấp phần cứng hệ thống bằng việc nâng cấp RAM, hay bộ nhớ.
* Khả năng mở rộng theo chiều ngang (Horizontal Scalable / Scale-Out): Là việc bổ sung phần cứng tránh sự hoạt động quá tải của hệ thống.
* Phân tán dữ liệu (Distributed Data): Là việc mô tả dữ liệu được phân tán ở các địa điểm khác nhau.
* Triển khai linh hoạt (Deployment Flexibilitty): Dễ dàng mở rộng thêm các nodes (Severs) mà không ảnh hưởng đến hoạt động của hệ thống.
* Tính sẵn sàng (High Availability): Hệ thống sẽ không bị ảnh hưởng khi một node bị trục trặc.
* Nhất quán cuối (Eventual Consistency): Khi ta đưa một dữ liệu mới vào một node của hệ thống, dữ lệu sẽ được lan truyền sang các node khác của hệ thống và cuối cùng tất cả node sẽ được đồng bộ.
* Lưu trữ tốt (Durability).

**Khi làm việc với Nosql ta sẽ gặp một số khái niệm sau:**

* FIELDS: Tương đương với khái niệm Columns trong SQL
* Documents: Thay thế khái niệm Rows trong SQL. Đây cũng chính là khái niệm làm nên sự khác biệt giữa NOSQL và SQL, 1 document chưa số cột (fields) không cố định trong khi 1 row thì số cột (columns) là định sẵn trước.
* Collection: Tương đương với khái niệm table trong SQL. Một Collection là tập hợp các documents.
* Key-Value: Cặp khóa - giá trị được dùng để lưu trữ dữ liệu trong NOSQL
* Cursor: Tạm dịch là con trỏ, sử dụng cursor để lấy dữ liệu từ database.

Trong các hệ cơ sở dữ liệu quan hệ, các cột được định nghĩa theo bảng, còn với hệ cơ sở dữ liệu không ràng buộc, các cột được định nghĩa ở mỗi document. Bởi thế, các document quản lý gần như tất cả, các collection không cần quản lý chặt chẽ những gì đang xảy ra trong nó nữa.

|  |  |
| --- | --- |
| RDBMS | NOSQL |
| Columns  Row  Table  Schema | Fields  Documents  Collection  Free Schema |

## Đặc Điểm

- NoSQL lưu trữ dữ liệu của mình theo dạng cặp giá trị “key – value”. Sử dụng số lượng lớn các node để lưu trữ thông tin.

- Chấp nhận dữ liệu bị trùng lặp do một số node sẽ lưu cùng thông tin giống nhau.

- Phi quan hệ – không có ràng buộc nào cho việc nhất quán dữ liệu.

- Có hiệu suất cao (High performance) và tính sẵn sàng cao (High availability).

## So Sánh Giữa SQL và NoSQL

|  |  |  |
| --- | --- | --- |
| Tính Năng | SQL | NoSQL |
| Hiệu xuất | Kém hơn NoSQL vì khi truy vấn nó phải tính toán, kiểm tra và xử lý mối quan hệ các bảng | Tốt hơn SQL do nó bỏ qua xử lý các ràng buộc giữa các bảng |
| Mở rộng theo chiều ngang | Có thể thực hiện được nhưng sẽ rất phức tạp nếu đã tồn tại dữ liệu trong database | Mở rộng dễ dàng mà không ảnh hưởng đến hệ thống |
| Tốc độ Đọc/Ghi | Kém hơn NoSQL do phải đảm bảo tính ràng buộc giữa các bảng | Tốc độ nhanh hơn SQL nhiều vì bỏ qua cơ chế ràng buộc giữa các bảng  Vì dữ liệu được lưu trong RAM ,sau đó đẩy xuống ổ cứng và cuối cùng là nhất quán cuối |
| Phần cứng | Đòi hỏi phần cứng cao | Không đòi hỏi cao về phần cứng |
| Thay đổi số node trong hệ thống | Vì tính nhất quán về dữ liệu nên khi thêm hoặc xóa một node thì hệ thống cần phải shutdown trong một khoảng thời gian | Vì tính nhất quán cuối nên không cần phải shutdown hệ thống |
| Truy vấn và Báo cáo | Dễ dàng, sử dụng ngôn ngữ SQL Query để truy vấn trực tiếp dữ liệu từ Database hoặc dùng công cụ hỗ trợ để lấy báo cáo | Việc lấy báo cáo dữ liệu trực tiếp từ NoSQL chưa được hỗ trợ tốt, thực hiện chủ yếu thông qua giao diện ứng dụng |
| Mở rộng dữ liệu | Khi muốn bổ sung thêm cột cho một bảng, cần phải khai báo trước | Không cần khai báo trước khi muốn bổ sung dữ liệu |
| Ứng Dụng | Sử dụng để xây dựng những hệ thống có quan hệ chặt chẽ và cần tính đồng nhất về dữ liệu như: tài chính, ngân hàng, chứng khoán… | Sử dụng xây dựng những hệ thống lưu trữ thông tin lớn, không quá quan trọng trong vấn đề đồng nhất dữ liệu trong 1 thời gian nhất định. |
| Ví dụ | **Facebook**: Sử dụng Hive(một dạng biến thể của SQL)  **Twitter:** sử dụng MySQL  **Stack Overflow**: sử dụng SQL Sever  **Youtube**: sử dụng MySQL  **Wikipedia:** sử dụng MySQL | **Facebook**: Sử dụng Cassandra(cơ sở dữ liệu đa chiều, lưu trữ phân tán)  **Twitter:** sử dụng cassandra |

## Ưu Và Nhược Điểm Của Cơ Sở Dữ Liệu NoSQL

**Hiệu suất hoạt động cao:** NoSQL có hiệu suất hoạt động cao, lưu trữ lượng lớn dữ liệu để đáp ứng nhu cầu lưu trữ ngày càng tăng hiện nay. Tuy nhiên để đạt được này, cần phải loại bỏ đi một số thứ như: các ràng buộc giữa các bảng, tính nhất quán dữ liệu, ngôn ngữ truy vấn SQL. Đồng thời NoSQL có một số cải tiến mới như sử dụng tốt index, khả năng phân tán dễ dàng đã giúp NoSQL có một hiệu suất hoạt động cao.

**Khả năng phân trang:** phân trang trong cơ sở dữ liệu khá khó khăn khi không có một phương pháp chính thống nào để phục vụ cho việc này. Người lập trình phải dùng các phương pháp khác nhau để có thể lấy đúng số item cần lấy.

**NoSQL là nguồn mở:** các sản phẩm nguồn mở đưa ra cho những người phát triển với nhiều lợi ích to lớn, ví dụ như việc sử dụng miễn phí. Ngoài ra, phần mềm mã nguồn mở có xu hướng tin cậy hơn, an ninh hơn và nhanh hơn để triển khai so với các lựa chọn thay thế sở hữu độc quyền. Ví dụ như các hệ quản trị cơ sở dữ liệu NoSQL như: Cassandra, CouchDB, Hbase, MongoDB, RavenDB và Redis.

**Việc mở rộng phạm vi là mềm dẻo:** Thay vì bổ sung thêm các máy chủ lớn hơn để điều khiển nhiều dữ liệu hơn, thì NoSQL cho phép một công ty phân tán dữ liệu qua nhiều máy chủ khi mà tải gia tăng.

**Cassandra**

Cassandra là một cơ sở dữ liệu hướng cột, phân tán mã nguồn mở được thiết kế để xử lý một khối lượng lớn dữ liệu giàn trải trên nhiều node mà vẫn đảm bảo tính sẵn sàng cao (Highly Availability), khả năng mở rộng hay thu giảm số node linh hoạt (Elastic Scalability) và chấp nhận một số lỗi (Fault Tolerant). Nó được phát triển bởi Facebook và vẫn còn tiếp tục phát triển và sử dụng cho mạng xã hội lớn nhất thới giới này. Năm 2008, Facebook chuyển nó cho cộng đồng mã nguồn mở và được Apache tiếp tục phát triển đến ngày hôm nay. Cassandra được coi là sự kết hợp của Amazon’s Dynamo và Google’s BigTable. Các nút máy chủ trong cụm Cassandra là đồng nhất theo thiết kế ngang hàng (peer-to-peer), không có bất cứ thành phần nào trong hệ thống là điểm hỏng thắt cổ chai (bottle-neck).



*Hình 1:*

Thiết kế của Cassandra là thiết kế phân tán trên hàng ngàn máy chủ mà không có bất cứ điểm chết tập trung nào. Cassandra có thiết kế dựa trên kiến trúc mạng ngang hàng (Peer - to - Peer) tất cả các nút máy chủ trong hệ thống đều có vai trò như nhau và không có nút máy chủ nào đóng vai trò là máy chủ trung tâm mà việc hỏng hóc của máy chủ này có thể kéo theo đánh sập hoàn toàn hệ thống như các kiến trúc chủ - khách truyền thống. Các nút máy chủ của Cassandra là độc lập và tham gia vào kết nối với các nút máy chủ khác trong hệ thống. Mỗi nút đều có thể xử lý các thao tác ghi và đọc dữ liệu, không phân biệt là dữ liệu được lưu trữ một cách vật lý trên máy chủ nào trong hệ thống. Khi một nút trong hệ thống bị hỏng hóc và dừng hoạt động, các thao tác đọc ghi dữ liệu có thể được xử lý bởi các nút khác trong hệ thống. Quá trình này hoàn toàn trong suốt với ứng dụng cho phép ẩn đi hỏng hóc của hệ thống đối với các ứng dụng đó. Trong Cassandra, mỗi đối tượng dữ liệu có thể được nhân bản và lưu giữ trên nhiều máy chủ. Nếu một trong các máy chủ lưu một phiên bản dữ liệu bị lỗi hoặc không phải là phiên bản được cập nhật dữ liệu mới nhất, Cassandra có cơ chế đồng bộ để luôn đảm báo các thao tác đọc sẽ luôn trả về dữ liệu mới nhất. Cơ chế này được thực thi trong quá trình đọc dữ liệu (read repair) thay vì đồng bộ ngay trong thao tác ghi dữ liệu, điều này cho phép tăng hiệu năng cho thao tác ghi dữ liệu.

**Phân tán dữ liệu trong Cassandra**

Cassandra sử dụng cơ chế hàm băm nhất quán phân tán (Distributed consistent hashing) tổ chức các nút máy chủ thành cụm theo định dạng vòng tròn và dữ liệu được phân tán theo vòng tròn này theo hàm băm nhất quán. Mỗi vòng tròn được coi là một Datacenter.



*Hình 1: phân tán dữ liệu trên cassandra*

Các nút trong một cụm Cassandra sẽ được phân bố trên một vòng tròn gọi là **ring** *(như hình trên)*. Mỗi nút sẽ được gán với 1 giá trị **key**, Cassandra dùng 127 bit để tạo ra key này. Mỗi nút trong **ring** sẽ quản lý một phạm vi giá trị của các **key**. Phạm vi của **key** được xác định trải đều từ giá trị của chính nút đó nắm giữ, đi ngược lại chiều kim đồng hồ cho đến khi gặp nút đầu tiên thì dừng lại. Đối chiếu lên hình, ta sẽ thấy rằng phạm vi các **key** mà nút ***T-1*** quản lý nằm trong vùng ***(T-0; T-1].*** Khi một bản ghi được ghi vào cụm Cassandra. Trường khóa của bản ghi đó sẽ được đi qua một hàm băm nhất quán, trả về một giá trị **key** 127bit, giá trị **key** này nằm trong vùng kiểm soát của nút nào thì bản ghi đó sẽ được ghi vào nút đấy. Ví dụ ta có giá trị trên các trường name được băm ra như bảng sau:

|  |  |
| --- | --- |
| Name | Hash Value |
| A | -2245452657672322382 |
| B | 7723358928203680754 |
| C | -6756552657672322382 |
| D | 1168658928203680754 |

Và ta có một cụm Cassandra với 4 nút được phân bố trên **ring** như hình sau:



*Hình 2: Cụm Cassandra với 4 nút được phân bố trên ring*

Với Cassandra, chúng ta có hai chiến lược phân mảnh (xác định vị trí của từng nút trong ***ring***)

**Random partitioning:** Đây là chiến lược mặc định và được đề xuất của Cassandra, vị trí của các node được xác định hoàn toàn thông qua mảng băm MD5. Phạm vi khóa nằm trong khoảng từ **0** tới **.**

**Ordered partitioning:** Chiến thuật phân mảnh đảm bảo các nút được sắp xếp theo thứ tự và phạm vi key mà mỗi nút sở hữu là như nhau.

Với chiến lược phân mảnh thứ nhất, nếu như các giá trị băm xuất ra giúp cho việc đặt các nút trong vòng phù hợp thì tất cả các bản ghi sẽ được phân bố đều trên toàn cụm. Việc thêm hay bớt mỗi nút ra khỏi cụm cũng dễ dàng hơn do không phải phân bố lại vị trí các nút khác. Với chiến lược phân mảnh thứ hai, khi mà các nút được phân bố đều vả phạm vi quản lý **key** là như nhau, những điều đó lại mang lại những bất lợi khá rõ ràng: Khó cân bằng trong cụm: Mỗi khi thêm hay bớt một nút khỏi cụm, người quản trị sẽ phải tự tái cân bằng cụm lại một cách thủ công để đảm bảo các nút phân bố đều. Nếu dữ liệu được ghi tuần tự, có thể xảy ra trường hợp hàng loạt dữ liệu được ghi vào một nút. Gây mất cân bằng trong cụm.

**Nhận xét:** Với cả hai chiến lược phân mảnh trên, vẫn để lộ ra những điểm yếu, khi số lượng nút trong vòng quá ít, hoặc các nút phân bố không đều theo giá trị băm của các bảng ghi đưa vào, rất dễ đưa đến hiện tượng mất cân bằng, quá tải trong cụm. Ngoài ra, khi thêm hay xóa một nút khỏi vòng, thì sẽ phải mất công tái cân bằng lại cụm.

**Nút ảo**

Để giải quyết vấn đề này, ta có một giải pháp đó là sử dụng ***nút ảo***. Nút ảo trông giống như một thành phần của vòng tròn trong hệ thống, nhưng bản chất nút ảo chỉ là ánh xạ của một nút vật lý đến một địa chỉ khác trong vòng. Khi dữ liệu đi vào vùng quản lý của nút ảo, nó sẽ được đưa về lưu trữ tại nút vật lý của nút ảo đó. Mỗi nút vật lý khi tham gia vào vòng sẽ được gán một vị trí của chính nút đó và gán thêm một số lượng các vị trí khác (được coi như là nút ảo của nút đó). Cassandra cấu hình mặc định mỗi một nút tham gia vòng sẽ được gán 256 nút ảo trong vòng.



*Hình 3:*

Hình trên thể hiện một vòng trong có 4 nút vật lý, mỗi nút được gán thêm 7 nút ảo, như vậy tổng cộng trên vòng tròn sẽ có 32 phân vùng ***key***. Vậy tác dụng của nút ảo là gì, khi việc phân tán đều các nút ảo ra khắp vòng, số lượng nút tăng lên khiến cho các phân vùng key bé lại, việc phân vùng key bé lại mang ý nghĩa rất lớn trong việc phân bổ dữ liệu của cụm Cassandra, việc phân vùng nhỏ lại và các nút sát nhau hơn đưa hệ thống càng gần đến với việc tất cả dữ liệu sẽ được phân bổ đều khắm các nút, xác suất dữ liệu được đưa vào các nút là cân bằng nhau khi mà trên một khoảng key nhỏ ta có đầy đủ các nút ảo hoặc nút vật lý. Trường hợp hoàn hảo nhất là các nút vật lý đều có thành phần hiện diện của mình đêu khắp trên vòng. Phân vùng key quản lý khi có và không có nút ảo.

**Nhân bản dữ liệu trong Cassandra**

Để thỏa mãn tính sẵn sàng và liên tục trong Cassandra, mỗi đối tượng dữ liệu có thể được nhân bản và lưu giữ trên nhiều máy chủ. Nếu một trong các máy chủ lưu một phiên bản dữ liệu bị lỗi hoặc là phiên bản cũ, không phải là phiên bản được cập nhật dữ liệu mới nhất, Cassandra có cơ chế đồng bộ để luôn đảm báo các thao tác đọc sẽ luôn trả về dữ liệu mới nhất. Đồng thời với việc này Cassandra tiến hành thao tác sửa lỗi đọc ***(read repair)*** là tiến trình ngầm để cập nhật trạng thái mới nhất cho tất cả các máy chủ lưu trữ nhân bản của dữ liệu. Cassandra tổ chức các nút máy chủ thành cụm theo định dạng vòng tròn và dữ liệu được phân tán theo vòng tròn này theo bảng hàm băm nhất quán ***(Distributed consistent hashing)***. Nếu mỗi dữ liệu của Cassandra được sao lưu trên **N** nút, khi một khóa **k** được quyết định sẽ lưu vào một nút nào đó, nút đó sẽ được coi là nút điều phối. Nút điều phối có nhiệm vụ phân phối bản ghi đấy cho **N-1** nút còn lại theo nguyên tắc: từ nút điều phối, đi theo chiều kim đồng hồ, dữ liệu sẽ được ghi lên 2 nút tiếp theo được gặp. Hình trên mô tả khi khóa **k** được xác định là sẽ ghi vào nút **B**, nút **B** sẽ đóng vai trò điều phối, luân chuyển khóa đấy cho 2 nút tiếp theo là nút **C** và nút **D**. như vậy, nút **D** sẽ lưu trữ các khóa nằm trong vùng **[A; D]**. Danh sách các khóa trong vùng này được gọi là danh sách liên kết của nút D. Việc đưa các giá trị của khóa k sang các nút khác áp dụng cho tất cả các tác vụ ghi, cập nhật hay xóa. Vì việc việc quyết định số lượng nút được luân chuyển ngay lập tức mỗi khi có tác vụ ghi diễn ra ảnh hưởng trực tiếp đến mức độ nhất quán của hệ thống.

Trong cấu hình của Cassandra Apache ta có một chỉ số "***replicatioon\_factor***" và "***w***". Chỉ số "***replication\_factor***" sẽ được cài đặt ngay khi khởi tạo một ***key\_space***, đó là số lượng nút trong vòng sẽ được dùng để sao lưu dữ liệu. Chỉ số "***w***" khi cấu hình Cassandra là số lượng nút trả về kết quả khi thực hiện tác vụ ghi bắt buộc để tác vụ đấy được coi là thành công.

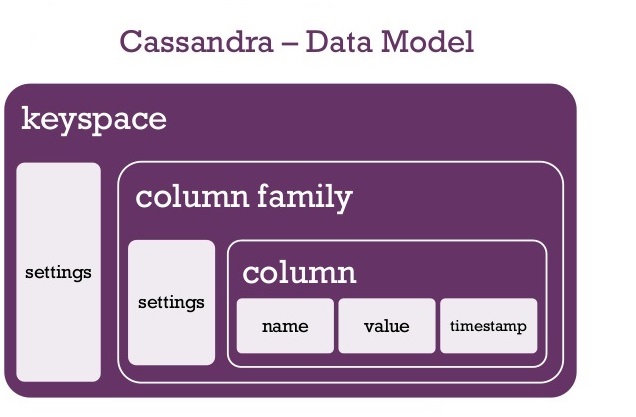
**Giao tiếp giữa các nút trong Cassandra**

****

Mỗi khi cụm Cassandra bổ sung hoặc loại bỏ một nút ra khỏi cụm, dữ liệu trong cụm sẽ phải được phân bố lại. Khi bổ sung một nút, nút đó sẽ lấy đi 1 phần dữ liệu của các nút khác khi nó được cấp cho 256 nút ảo. Khi một nút bị loải khỏi cụm, dữ liệu của nút đó sẽ phải được rải đều cho các nút khác. Trong Cassandra, các nút giao tiếp với nhau thông qua giao thức ***Gossip***. Gossip là một giao thức dùng để cập nhật thông tin về trạng thái của các node khác đang tham gia vào cluster. Đây là một giao thức liên lạc dạng peer-to-peer trong đó mỗi node trao đổi định kỳ thông tin trạng thái của chúng với các node khác mà chúng có liên kết. Tiến trình gossip chạy mỗi giây và trao đổi thông tin với nhiều nhất là ba node khác trong cluster. Các node trao đổi thông tin về chính chúng và cả thông tin với các node mà chúng đã trao đổi, bằng cách này toàn bộ những node có thể nhanh chóng hiểu được trạng thái của tất cả các node còn lại trong cluster. Một gói tin gossip bao gồm cả version đi kèm với nó, như thế trong mỗi lần trao đổi gossip, các thông tin cũ sẽ bị ghi đè bởi thông tin mới nhất ở một số node. Khi một node được khởi động, nó sẽ xem file cấu hình *cassandra.yaml* để xác định tên cluster chứa nó và các nút khác trong cluster được cấu hình trong file, được biết với tên là seed node. Để ngăn chặn sự đứt đoạn trong truyền thông gossip, tất cả các nút trong cluster phải có cùng 1 danh sách các seed node được liệt kê trong file cấu hình. Bởi vì, phần lớn các xung đột được sinh ra khi 1 node được khởi động. Mặc định, 1 node sẽ phải nhớ những node mà nó đã từng gossip kể cả khi khởi động lại và seed node sẽ không có mục đích nào khác ngoài việc cập nhật 1 node mới khi nó tham gia vào cluster. Tức là, khi một node tham gia vào cluster, nó sẽ liên lạc với các seed node để cập nhật trạng thái của tất cả các node khác trong cluster. Trong những cluster có nhiều data center, danh sách seed node nên chứa ít nhất một seed node trên mỗi data center, nếu không thì khi có 1 nút mới tham gia vào cluster, thì nó sẽ liên lạc với một seed node nằm trên data center khác. Cũng không nên để mọi node đều là seed node vì nó sẽ làm giảm hiệu năng của gossip và gây khó duy trì. Việc tối ưu gossip là không quan trọng như khuyến khích nên sử dụng một danh sách nhỏ các seed node, thông thường 3 seed node trên một data center.

**Mô hình dữ liệu của Cassandra**

Mô hình dữ liệu của Cassandra tuân theo cách tiếp cận hệ thống cột, mà có thể dễ dàng được hiểu tương tự như một cấu trúc bảng quan hệ nhưng theo cách NoSQL. Mô tả dưới đây sẽ làm rõ ràng hơn:



*Hình: mô hình dữ liệu trên cassandra*

**Keyspace:** Một khóa có thể được xem như là kho chứa ngoài cùng nhất cho dữ liệu trong Cassandra. Tất cả dữ liệu trong Cassandra sẽ đặt trong một keyspace. Nó có thể được xem như là một cơ sở dữ liệu trong RDBMS, nó là một tập hợp các bảng. Trong trường hợp của Cassandra, một keyspace là một tập hợp hệ thống cột.

**Column family (Hệ thống Cột):** Một hệ thống cột có thể được xem như là một bộ sưu tập các hàng, và mỗi hàng có một tập hợp các cột. Nó tương tự như một bảng trong RDBMS nhưng có một số khác biệt. Hệ thống cột được xác định, nhưng không cần thiết mỗi hàng phải có tất cả các cột và các cột có thể được thêm vào hoặc xoá bỏ khỏi một hàng khi cần thiết.

**Column (Cột):** Cột là đơn vị dữ liệu cơ bản trong Cassandra. Nó có 3 giá trị*: khoá hoặc tên cột, giá trị cột, và một mốc thời gian.*

**Quy trình khoa học dữ liệu**

Quy trình tiêu chuẩn liên ngành cho khai thác dữ liệu, thường được biết đến với từ viết tắt CRISP-DM (Cross-industry standard process for Data Mining) là một quy trình lặp, có khả năng quay lui (backtracking) gồm 6 giai đoạn:



*Hình 3: Mô hình CRISP-DM*

* Tìm hiểu nghiệp vụ (Business understanding)
  + Hiểu mục tiêu nghiệp vụ
  + Đánh giá tình huống
  + Quy đổi từ mục tiêu nghiệp vụ sang mục tiêu khai phá dữ liệu
  + Xây dựng kế hoạch dự án
* Tìm hiểu dữ liệu (Data understanding)
  + Xem xét các yêu cầu về dữ liệu
  + Thu thập, thăm dò và đánh giá chất lượng ban đầu
* Chuẩn bị dữ liệu (Data preparation)
  + Lựa chọn dữ liệu
  + Thu thập dữ liệu
  + Tích hợp và định dạng dữ liệu (ETL)
  + Làm sạch dữ liệu
  + Chuyển đổi dữ liệu
* Mô hình hoá (Modeling)
  + Lựa chọn kỹ thuật mô hình thích hợp
  + Tách tập dữ liệu thành tập huấn và thử nghiệm tập hợp con cho mục đích đánh giá
  + Phát triển và kiểm tra các thuật toán mô hình thay thế và cài đặt tham số
  + Tinh chỉnh cài đặt mô hình theo đánh giá ban đầu về hiệu suất của mô hình
* Đánh giá mô hình (Evaluation Model)
  + Đánh giá mô hình trong bối cảnh ứng với các tiêu chí nghiệp vụ
  + Phê duyệt mẫu
* Triển khai (Deployment)
  + Tạo báo cáo kết quả
  + Lập kế hoạch và phát triển quy trình triển khai
  + Triển khai mô hình
  + Phân phối kết quả mô hình và tích hợp trong hệ thống của tổ chức
  + Lập kế hoạch bảo trì / cập nhật
  + Đánh giá dự án

**Các loại phân tích dữ liệu:**

1. **Thống kê mô tả (Descriptive Statistics):**

Là các phương pháp liên quan đến việc thu thập số liệu, tóm tắt, trình bày, tính toán và mô tả các đặc trưng khác nhau để phản ánh một cách tổng quát đối tượng nghiên cứu.

* Các nguyên tắc định lượng mô tả các features chính của một tập dữ liệu. Về bản chất, nó mô tả một bộ dữ liệu.
* Quy trình mô tả và diễn dịch là các bước khác nhau
* Đơn biến và 2 biến là 2 loại phân tích thống kê mô tả

1. **Thống kê suy luận (Inferential Statistics):**

Là bao gồm các phương pháp ước lượng các đặc trưng của tổng thể, phân tích mối liên hệ giữa các hiện tượng nghiên cứu, dự đoán hoặc ra quyết định trên cơ sở thu thập thông tin từ kết quả quan sát mẫu. Nghĩa là, thống kê từ một lượng sample để dự đoán về population.

* Suy luận thường là mục tiêu của các mô hình thống kê (Statistical models)
* Suy luận phụ thuộc rất nhiều vào population và sample

1. **Phân tích dữ liệu thăm dò (Exploratory Data Analysis - EDA)**

Là một cách tiếp cận để phân tích các tập dữ liệu để tóm tắt các đặc điểm chính của chúng, tìm các mối quan hệ chưa được biết trước đó, thường với các phương pháp visual. EDA khác với phân tích dữ liệu ban đầu (Initial Data Analysis - IDA) tập trung hẹp hơn trong việc kiểm tra các giả định cần thiết cho việc kiểm tra mô hình và giả thuyết, và xử lý các giá trị thiếu và thực hiện các biến đổi khi cần. EDA bao gồm IDA.

* + Khám phá các model tốt cho việc khám phá các kết nối mới.
  + Có ích cho việc định nghĩa các nghiên cứu và câu hỏi trong tương lai.
  + Không nên sử dụng phân tích thăm dò để khái quát hoặc dự đoán.

1. **Phân tích dự đoán (Predictive Analysis):**

Phân tích dự đoán là nhánh của phân tích nâng cao được sử dụng để đưa ra dự đoán về các sự kiện không xác định trong tương lai. Phân tích dự báo sử dụng nhiều kỹ thuật từ khai phá dữ liệu, thống kê, mô hình hóa, học máy và trí thông minh nhân tạo để phân tích dữ liệu hiện tại để đưa ra dự đoán về tương lai. Nó sử dụng một số khai thác dữ liệu, mô hình tiên đoán và các kỹ thuật phân tích để tập hợp quản lý, công nghệ thông tin và quy trình nghiệp vụ mô hình hóa để đưa ra dự đoán về tương lai. Các mẫu tìm thấy trong dữ liệu lịch sử và giao dịch có thể được sử dụng để xác định các rủi ro và cơ hội cho tương lai. Các mô hình phân tích dự đoán nắm bắt các mối quan hệ giữa nhiều yếu tố để đánh giá rủi ro với một tập hợp các điều kiện cụ thể để gán điểm hoặc trọng số. Bằng cách áp dụng thành công các phân tích dự báo, các doanh nghiệp có thể giải thích một cách hiệu quả các dữ liệu lớn vì lợi ích của họ.

1. **Phân tích nguyên nhân (Causal analysis)**

Tìm hiểu chuyện gì sẽ xảy ra đối với một biến khi thay đổi một biến khác

1. **Phân tích cơ chế (Mechanistic analysis)**

Hiểu được chính xác những thay đổi ở các biến dẫn tới những thay đổi các biến khác

* Khó để suy luận, trừ những tình huống đơn giản.
* Thường mô hình hóa bởi một tập các phương trình xác định.
* Nếu các phương trình được biết nhưng các thông số thì không, chúng có thể được suy ra cùng với phân tích dữ liệu.

**Cách xác định bài toán trong Machine Learning**

Trong thực tế, trước khi giải bất kỳ bài toán nào, việc đầu tiên chúng ta cần làm đó là xác định vấn đề. Đặc biệt khi làm trong lĩnh vực Machine Learning (ML), nhiều khi bài toán do các doanh nghiệp đặt ra khá mơ hồ và không cụ thể khiến cho quá trình xây dựng mô hình dự đoán đi đến ngõ cụt hoặc không đáp ứng được yêu cầu của khách hàng.

**Feature Engineering**

1. **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. Feature engineering cố gắng biểu diễn tốt nhất tập dữ liệu ban đầu sao cho tương thích với mô hình dự đoán đang sử dụng.

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.*

1. **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:

1. **Feature Extractor**

* **Đầu vào**

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 ra**
  + **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.

* + **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.

* + **Prior knowledge about data (kiến thức về 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.

1. **Main Algorithms (các thuật toán chính)**

Khi có được extracted features rồi, chúng ta sử dụng những thông tin này cùng với training output và 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.

**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.

1. **Một số ví dụ về Feature Engineering**

* **Feature Selection**

Giả sử rằng các điểm dữ liệu có số features khác nhau (do kích thước dữ liệu khác nhau hay do một số feature mà điểm dữ liệu này có nhưng điểm dữ liệu kia lại không thu thập được), và số lượng features là cực lớn. Chúng ta cần chọn ra một số lượng nhỏ hơn các feature phù hợp với bài toán.

**Một số cách trong Feature Selection như:**

**Cách 1: Loại bỏ các features có giá trị phương sai thấp:**

Ngưỡng phương sai (*VarianceThreshold*) là một phương pháp cơ bản của Feature Selection. Nó loại bỏ tất cả các features có phương sai không đáp ứng giá trị ngưỡng. Theo mặc định, nó loại bỏ tất cả các tính năng zero-variance (phương sai bằng 0), tức là các features có cùng giá trị trong tất cả các sample.

Trong đó: **p** là số phần trăm muốn loại bỏ của một sample, như ví dụ dưới đây, ta chọn 80% = 0.8

**from** sklearn.feature\_selection **import** VarianceThreshold  
**import** numpy **as** np  
X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]  
sel = VarianceThreshold(threshold=(.8 \* (1 - .8)))  
print(**"Ma trận ban đầu:\n{}\n\nMa trận sau khi transform:\n{}\n"**.format(X, sel.fit\_transform(X)))**def** column(matrix, i):  
 **return** [row[i] **for** row **in** matrix]  
print(**"Giá trị của varianceThreshold là: {}"**.format(sel))  
print(**"Giá trị variance của cột 0 là: {}\nGiá trị variance của cột 1 là: {}"**.format(np.var(column(X,0)),np.var(column(X,1))))

*Kết Quả:*

Ma trận ban đầu:

[[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]

Ma trận sau khi transform:

[[0 1] [1 0] [0 0] [1 1] [1 0] [1 1]]

#Ta có thể thấy VarianceThreshold đã xóa cột đầu tiên của ma trận vì variance của cột 0 < giá trị VarianceThreshold

#để kiểm chứng, ta sẽ thử xem giá trị của cột 0 và cột 1 của ma trận

Giá trị của varianceThreshold là:

VarianceThreshold(threshold=0.15999999999999998)

Giá trị variance của cột 0 là: 0.13888888888888892

Giá trị variance của cột 1 là: 0.22222222222222224

**Cách 2: Dimentionality reduction (giảm chiều dữ liệu)**

Một phương pháp khác là làm giảm số chiều của dữ liệu để giảm bộ nhớ và khối lượng tính toán. Việc giảm số chiều này có thể được thực hiện bằng nhiều cách, trong đó random projection là cách đơn giản nhất. Tức chọn một ma trận chiếu (projection matrix) ngẫu nhiên rồi nhân nó với từng điểm dữ liệu (giả sử dữ liệu ở dạng vector cột) để được các vector có số chiều thấp hơn.

Ví dụ, vector ban đầu có số chiều là 784, chọn ma trận chiếu có kích thước (100x784), khi đó nếu nhân ma trận chéo này với vector ban đầu, ta sẽ được một vector mới có số chiều là 100, nhỏ hơn số chiều ban đầu rất nhiều. Lúc này, có thể ta không có tên gọi cho mỗi feature nữa vì các feature ở vector ban đầu đã được trộn lẫn với nhau theo một tỉ lệ nào đó rồi lưu và vector mới này. Mỗi thành phần của vector mới này được coi là một feature (không tên).

Việc chọn một ma trận chiếu ngẫu nhiên đôi khi mang lại kết quả tệ không mong muốn vì thông tin bị mất đi quá nhiều.

Một phương pháp được sử dụng nhiều để hạn chế lượng thông tin mất đi có tên là Principle Component Analysis:



*Hình: Sử dụng phương pháp PCA để giảm chiều dữ liệu 3D xuống 2D*

**Chú ý:** Feature learning không nhất thiết phải làm giảm số chiều dữ liệu, đôi khi feature vector còn có số chiều lớn hơn raw data. Random projection cũng có thể làm được việc này nếu ma trận chiếu là một ma trận cao (số cột ít hơn số hàng).

* **Bag-of-word**

Bag-of-word (BoW) là phương pháp đưa các từ, các câu, đoạn văn ở dạng text trong các văn bản về một vector mà mỗi phần tử là một số. Ý tưởng của BoW là phân tích và phân nhóm dựa theo "Bag of Words". Với test data mới, tiến hành tìm ra số lần từng từ của test data xuất hiện trong "bag". Tuy nhiên BoW vẫn tồn tại khuyết điểm, nên TF-IDF là phương pháp khắc phục. Có thể ứng dụng BoW + TF-IDF vào việc tìm kiếm, phân loại tài liệu, lọc mail spam xác định ý định của người dùng…

**BoW hoạt động như thế nào?**

Trong hầu hết các ngôn ngữ, có một số từ có xu hướng xuất hiện thường xuyên như trong tiếng anh có "is", "the"... tương tự tiếng việt có các từ như "là", "của", "cứ"... Chính vì vậy nếu chỉ xét theo tần số xuất hiện của từng từ thì việc phân loại văn bản rất có thể cho kết quả sai dẫn tỷ lệ chính xác sẽ thấp.

Giải pháp phổ biến là sử dụng một phương pháp thống kê có tên là **Term Frequency– Inverse Document Frequency (TF-IDF)**, giá trị **TF-IDF** của một từ là một con số thu được qua thống kê thể hiện mức độ quan trọng của từ này trong một văn bản, mà bản thân văn bản đang xét nằm trong một tập hợp các văn bản.

Đầu tiên, **TF**(Term Frequency) là tần số xuất hiện của 1 từ trong 1 văn bản có cách tính như sau:

*Trong đó:*

* ***f(t,d)*** *số lần xuất hiện từ t trong văn bản d*
* *Mẫu số là tổng số từ trong văn bản d*

Tiếp theo, là **IDF** (Inverse Document Frequency): Tần số nghịch của 1 từ trong tập văn bản.

Mục đích của việc tính IDF là giảm giá trị của các từ thường xuyên xuất hiện như "is", "the"... Do các từ này không mang nhiều ý nghĩa trong việc phân loại văn bản.

*Trong đó:*

* ***N:*** *là tổng các văn bản trong một tập văn bản* ***N =* |D|**
* *số lượng văn bản khi mà t xuất hiện.*

Cuối cùng, **TF-IDF** những từ có giá trị TF-IDF cao là những từ xuất hiện nhiều trong văn bản này, và xuất hiện ít trong các văn bản khác. Việc này giúp lọc ra những từ phổ biến và giữ lại những từ có giá trị cao (từ khoá của văn bản đó). Được tính bởi công thức:

**Áp dụng:**

Khởi tạo 2 văn bản, tính số lần xuất hiện của mỗi từ trong văn bản:

*# Khởi tạo 2 văn bản text1, text2*

text1 = **"Phúc thích xem phim , Trâm cũng thích xem phim"**text2 = **"Ngoài ra , Phúc còn thích bơi lội"**bowA = text1.split(**" "**) *# tách từ ở văn bản 1*bowB = text2.split(**" "**) *# tách từ ở văn bản 2*

*#Tạo một dictionary*word\_dict = set(bowA).union(set(bowB))  
wordDictA = dict.fromkeys(word\_dict, 0)  
wordDictB = dict.fromkeys(word\_dict, 0)  
  
*#Đếm số lượng từ***for** word **in** bowA:  
 wordDictA[word]+=1  
**for** word **in** bowB:  
 wordDictB[word]+=1  
  
print(**"Các từ trong 2 văn bản là:\n {}"**.format(word\_dict))  
print(**"Số từ xuất hiện trong văn bản 1 là:\n {}\n\nSố từ xuất hiện trong văn bản 2 là:\n {}"**.format(wordDictA,wordDictB))

***Kết quả:***

Các từ trong 2 văn bản là:

{'thích', 'phim', ',', 'lội', 'cũng', 'ra', 'Trâm', 'còn', 'Ngoài', 'Phúc', 'xem', 'bơi'}

Số từ xuất hiện trong văn bản 1 là:

{'thích': 2, 'phim': 2, ',': 1, 'lội': 0, 'cũng': 1, 'ra': 0, 'Trâm': 1, 'còn': 0, 'Ngoài': 0, 'Phúc': 1, 'xem': 2, 'bơi': 0}

Số từ xuất hiện trong văn bản 2 là:

{'thích': 1, 'phim': 0, ',': 1, 'lội': 1, 'cũng': 0, 'ra': 1, 'Trâm': 0, 'còn': 1, 'Ngoài': 1, 'Phúc': 1, 'xem': 0, 'bơi': 1}

***Tính TF*:**

**def** compute\_TF(word\_dict, bow):  
 tf\_dict = {}  
 bow\_count = len(bow)  
 **for** word, count **in** word\_dict.items():  
 tf\_dict[word] = count / float(bow\_count)  
 **return** tf\_dict

print(**"\nKết quả TF:\n văn bản 1: {}\n văn bản 2 {}"** .format(compute\_TF(wordDictA,bowA),compute\_TF(wordDictB,bowB)))

***Kết quả:***

Kết quả TF:

văn bản 1: {'xem': 0.2, 'thích': 0.2, 'còn': 0.0, 'Ngoài': 0.0, 'ra': 0.0, 'Trâm': 0.1, ',': 0.1, 'bơi': 0.0, 'lội': 0.0, 'Phúc': 0.1, 'phim': 0.2, 'cũng': 0.1}

văn bản 2: {'xem': 0.0, 'thích': 0.125, 'còn': 0.125, 'Ngoài': 0.125, 'ra': 0.125, 'Trâm': 0.0, ',': 0.125, 'bơi': 0.125, 'lội': 0.125, 'Phúc': 0.125, 'phim': 0.0, 'cũng': 0.0}

**Tính IDF:**

**def** compute\_IDF(doc\_list):  
 **import** math *#import thư viện math* idf\_dict = {} *#tạo một dictionary rỗng* N = len(doc\_list) *#gán độ dài của list cho biến N* idf\_dict = dict.fromkeys(doc\_list[0].keys(), 0) *#tạo dictionary lưu các keys với value = 0  
 #lọc ra thành 1 list gồm các từ xuất hiện >=1 lần* **for** doc **in** doc\_list:  
 **for** word, count **in** doc.items():  
 **if** count > 0:  
 idf\_dict[word] += 1  
 **for** word, count **in** idf\_dict.items():  
 idf\_dict[word] = math.log(N / float(count))  
 **return** idf\_dict

print(**"Kết Quả IDF:\n {}"**.format(compute\_IDF([wordDictA, wordDictB])))

*Kết quả:*

Kết Quả IDF:

{'cũng': 0.6931471805599453, 'Phúc': 0.0, 'bơi': 0.6931471805599453, 'Trâm': 0.6931471805599453, 'ra': 0.6931471805599453, 'phim': 0.6931471805599453, 'lội': 0.6931471805599453, 'còn': 0.6931471805599453, 'xem': 0.6931471805599453, 'Ngoài': 0.6931471805599453, ',': 0.0, 'thích': 0.0}

Từ kết quả trên có thể nhìn thấy những từ có trọng số càng cao thì những từ đó càng có giá trị phân loại và ngược lại ví dụ như từ "thích", dấu “,” xuất hiện nhiều nên sẽ không có giá trị phân loại các văn bản với nhau. Tuy nhiên bộ train data lần này ít nên hiệu quả không rõ rệt, nếu thử trên lượng data lớn chắc chắn sẽ rất hiệu quả.

**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:

**Mô hình hóa (Modeling)**

**Regression Algorithms**

* Linear Regression
* Logistic Regression
* Stepwise Regression

**Gradient Decent**

**Giới Thiệu:**

Trong Machine Learning nói riêng và Toán Tối Ưu nói chung, chúng ta thường xuyên phải tìm giá trị nhỏ nhất (hoặc đôi khi là lớn nhất) của một hàm số nào đó. Ví dụ như các hàm mất mát trong hai bài Linear Regression và K-means Clustering. Nhìn chung, việc tìm global minimum của các hàm mất mát trong Machine Learning là rất phức tạp, thậm chí là bất khả thi. Thay vào đó, người ta thường cố gắng tìm các điểm local minimum, và ở một mức độ nào đó, coi đó là nghiệm cần tìm của bài toán.



*Hình 1: biểu diễn các điểm local và global*

Các điểm local minimum là nghiệm của phương trình đạo hàm bằng 0. Nếu bằng một cách nào đó có thể tìm được toàn bộ (hữu hạn) các điểm cực tiểu, ta chỉ cần thay từng điểm local minimum đó vào hàm số rồi tìm điểm làm cho hàm có giá trị nhỏ nhất (đoạn này nghe rất quen thuộc, đúng không?). Tuy nhiên, trong hầu hết các trường hợp, việc giải phương trình đạo hàm bằng 0 là bất khả thi. Nguyên nhân có thể đến từ sự phức tạp của dạng của đạo hàm, từ việc các điểm dữ liệu có số chiều lớn, hoặc từ việc có quá nhiều điểm dữ liệu.

Hướng tiếp cận phổ biến nhất là xuất phát từ một điểm mà chúng ta coi là gần với nghiệm của bài toán, sau đó dùng một phép toán lặp để tiến dần đến điểm cần tìm, tức đến khi đạo hàm gần với 0. Gradient Descent – Đạo hàm ngược (viết gọn là GD) và các biến thể của nó là một trong những phương pháp được dùng nhiều nhất.

**Gradient Decent cho hàm một biến:**

****

*Hình 2: parapol của hàm f(x)*

Giả sử là điểm ta tìm được sau vòng lặp **t**. Ta cần tìm một thuật toán để đưa về càng gần càng tốt.

Từ hình trên ta thấy:

Nếu đạo hàm của hàm số tại thì nằm về bên phải so với (và ngược lại). Để điểm tiếp theo gần với hơn, chúng ta cần di chuyển về phía bên trái, tức về phía âm. Nói cách khác, ta cần di chuyển ngược dấu với đạo hàm

Trong đó: là một đại lượng ngược dấu với đạo hàm

* càng xa về phía bên phải thì càng lớn hơn 0 (và ngược lại). Vậy, lượng di chuyển , một cách trực quan nhất, là tỉ lệ thuận với

Từ 2 nhận xét trên cho ta một cách cập nhật đơn giản:

Trong đó: (đọc là eta) là một số dương được gọi là learning rate (tốc độ học). Dấu trừ thể hiện việc chúng ta phải đi ngược với đạo hàm (Đây cũng chính là lý do phương pháp này được gọi là Gradient Descent - descent nghĩa là đi ngược). Các quan sát đơn giản phía trên, mặc dù không phải đúng cho tất cả các bài toán, là nền tảng cho rất nhiều phương pháp tối ưu nói chung và thuật toán Machine Learning nói riêng.

**Ví dụ đơn giản với Python:**

Xét hàm số với đạo hàm  **2x + 5cos(x).** Giả sử bắt đầu từ một điểm nào đó, tại vòng lặp thứ **,** chúng ta sẽ cập nhật như sau:

Các hàm số:

* **Grad\_Fx** để tính đạo hàm
* **Cost** để tính giá trị của hàm số. Hàm này không sử dụng trong thuật toán nhưng thường được dùng để kiểm tra việc tính đạo hàm của đúng không hoặc để xem giá trị của hàm số có giảm theo mỗi vòng lặp hay không.
* Hàm **Gradient\_Decent** là phần chính thực hiện thuật toán Gradient Desent nêu phía trên. Đầu vào của hàm số này là learning rate () và điểm bắt đầu. Thuật toán dừng lại khi đạo hàm có độ lớn đủ nhỏ

Các điểm khởi tạo khác nhau là và

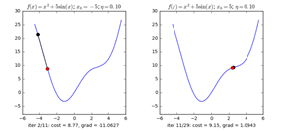
**import** numpy **as** np  
  
**def** grad\_Fx(x):  
 **return** 2\*x+ 5\*np.cos(x)**def** cost(x):  
 **return** x\*\*2 + 5\*np.sin(x)  
**def** Gradient\_Decent(eta, x0):  
 x = [x0]  
 **for** it **in** range(100):  
 x\_new = x[-1] - eta\*grad\_Fx(x[-1])  
 **if** abs(grad\_Fx(x\_new)) < 1e-3: *#Thuật toán dừng lại khi đạo hàm có độ lớn đủ nhỏ (0.001)* **break** x.append(x\_new)  
 **return** (x, it)  
(x1, it1) = Gradient\_Decent(.1, -5) *# đầu vào, learning rate = 0.1, điểm bắt đầu bằng -5*(x2, it2) = Gradient\_Decent(.1, 5) *# đầu vào, learning rate = 0.1, điểm bắt đầu bằng 5*print(**'với x1 = %f, cost = %f, hội tụ sau %d bước lặp'**%(x1[-1], cost(x1[-1]), it1))  
print(**'với x2 = %f, cost = %f, hội tụ sau %d bước lặp'**%(x2[-1], cost(x2[-1]), it2))

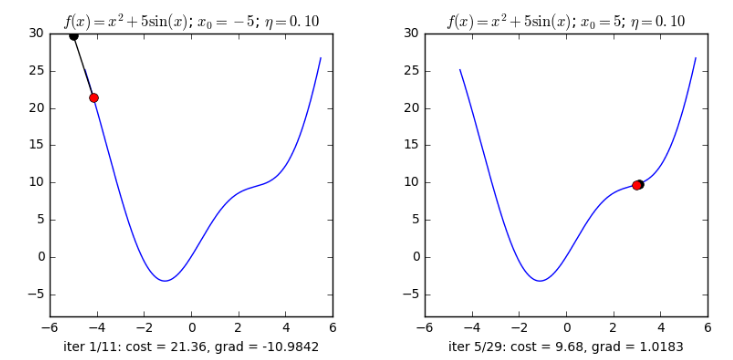
*Kết quả:*

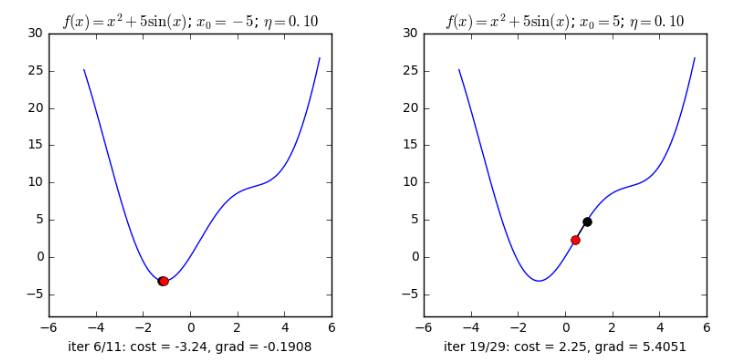
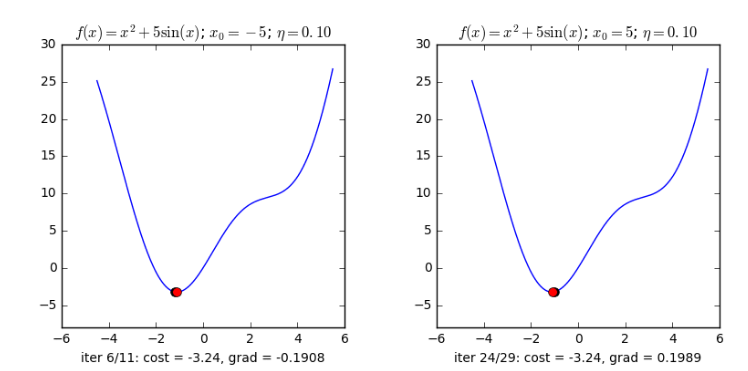
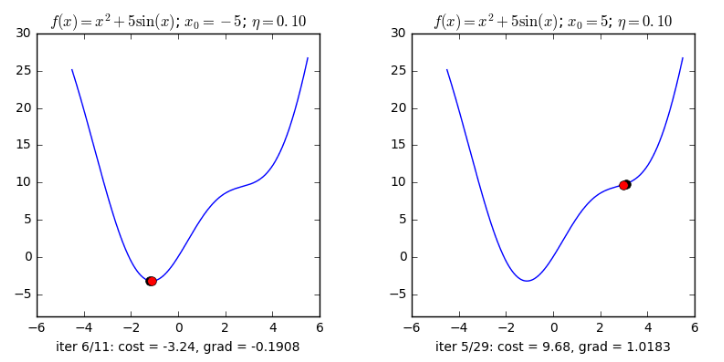
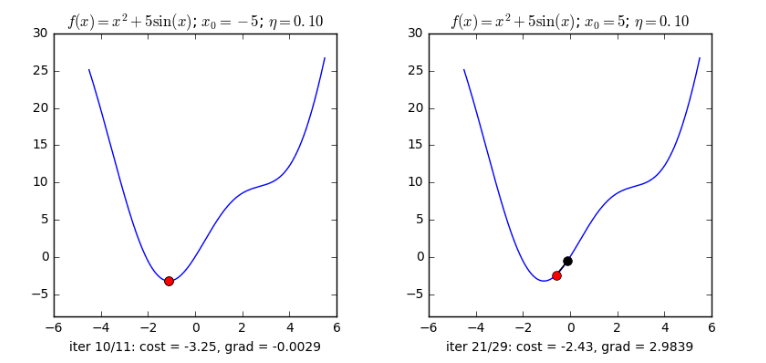
với x1 = -1.110667, cost = -3.246394, hội tụ sau 11 bước lặp

với x2 = -1.110341, cost = -3.246394, hội tụ sau 29 bước lặp

Vậy là với các điểm ban đầu khác nhau, thuật toán của chúng ta tìm được nghiệm gần giống nhau, mặc dù với tốc độ hội tụ khác nhau. Dưới đây là hình ảnh minh họa thuật toán GD cho bài toán này.





*** Hình 1: Hình 2:*

*Hình 3: Hình 4:*

*Hình 5: Hình 6:*

Từ hình minh họa trên ta thấy rằng ở hình bên trái, tương ứng với , nghiệm hội tụ nhanh hơn, vì điểm ban đầu gần với nghiệm hơn. Hơn nữa, với ở hình bên phải, đường đi của nghiệm có chứa một khu vực có đạo hàm khá nhỏ gần điểm có hoành độ bằng 2. Điều này khiến cho thuật toán la cà ở đây khá lâu. Khi vượt qua được điểm này thì mọi việc diễn ra rất tốt đẹp.

Nhận xét:

* Với các điểm ban đầu khác nhau, thuật toán tìm được nghiệm gần giống nhau, mặc dù với tốc độ hội tụ khác nhau.
* Tốc độ hội tụ của GD không những phụ thuộc vào điểm khởi tạo ban đầu mà còn phụ thuộc vào learning rate.

Việc lựa chọn learning rate rất quan trọng trong các bài toán thực tế. Việc lựa chọn giá trị này phụ thuộc nhiều vào từng bài toán và phải làm một vài thí nghiệm để chọn ra giá trị tốt nhất. Ngoài ra, tùy vào một số bài toán, GD có thể làm việc hiệu quả hơn bằng cách chọn ra learning rate phù hợp hoặc chọn learning rate khác nhau ở mỗi vòng lặp

**Gradient Decent cho hàm nhiều biến:**

Giả sử ta cần tìm global minimum cho hàm trong đó **θ** (theta) là một vector, thường được dùng để ký hiệu tập hợp các tham số của một mô hình cần tối ưu (trong Linear Regression thì các tham số chính là hệ số). Đạo hàm của hàm số đó tại một điểm **θ** bất kỳ được ký hiệu là (hình tam giác ngược đọc là *nabla*). Tương tự như hàm 1 biến, thuật toán GD cho hàm nhiều biến cũng bắt đầu bằng một điểm dự đoán **,** sau đó, ở vòng lặp thứ , quy tắc cập nhật là:

**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.

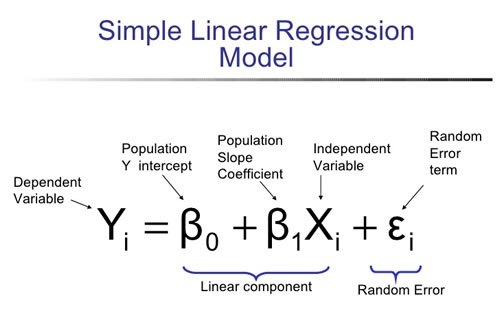
Đa cộng tuyến hoàn hảo:

|  |  |  |
| --- | --- | --- |
| 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) =>

**Classification Algorithms**

* Linear Classifier
* Support Vector Machine (SVM)
* Kernel SVM
* Sparse Representation-based classification (SRC)

**Instance-based Algorithms**

* k-Nearest Neighbor (kNN)
* Learning Vector Quantization (LVQ)

**Regularization Algorithms**

* Ridge Regression
* Least Absolute Shrinkage and Selection Operator (LASSO)
* Least-Angle Regression (LARS)

**Bayesian Algorithms**

* Naive Bayes
* Gaussian Naive Bayes

**Clustering Algorithms**

* k-Means clustering
* k-Medians
* Expectation Maximization (EM)

**Đánh giá mô hình (Evaluation Model)**

**Giới Thiệu:**

Khi xây dựng một mô hình Machine Learning, chúng ta cần một phép đánh giá để xem mô hình sử dụng có hiệu quả không và để so sánh khả năng của các mô hình. Trong bài viết này, tôi sẽ giới thiệu các phương pháp đánh giá các mô hình classification.

Có rất nhiều cách đánh giá một mô hình phân lớp. Tuỳ vào những bài toán khác nhau mà chúng ta sử dụng các phương pháp khác nhau. Các phương pháp thường được sử dụng là: Accuracy score, Confusion matrix, ROC curve, Area Under the Curve, Precision and Recall, F1 score, Top R error.

**Accuracy**

Cách đơn giản và hay được sử dụng nhất là accuracy (độ chính xác). Cách đánh giá này đơn giản tính tỉ lệ giữa số điểm được dự đoán đúng và tổng số điểm trong tập dữ liệu kiểm thử.

Trong ví dụ này, ta có thể đếm được có 6 điểm dữ liệu được dự đoán đúng trên tổng số 10 điểm. Vậy ta kết luận độ chính xác của mô hình là 0.6 (hay 60%). Để ý rằng đây là bài toán với chỉ 3 class, nên độ chính xác nhỏ nhất đã là khoảng 1/3, khi tất cả các điểm được dự đoán là thuộc vào một class nào đó.

**import** numpy **as** np *#numpy để làm việc với mảng***from** sklearn.naive\_bayes **import** GaussianNB *#thư viện dùng naive-bayes***from** sklearn.metrics **import** accuracy\_score *#hàm accuracy\_score trong sklearn  
# Dữ liệu huấn luyện đầu vào*training\_points = [[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]] *# các điểm*training\_labels = [1, 1, 1, 2, 2, 2] *# các nhãn*X = np.array(training\_points) Y = np.array(training\_labels)  
*# Tạo Naive Bayes classifier*clf = GaussianNB()  
clf.fit(X, Y) *#train model với naive bayes  
# dữ liệu test*test\_points = [[1, 1], [2, 2], [3, 3], [4, 3]]  
test\_labels = [2, 2, 2, 1]  
predicts = clf.predict(test\_points) *#dựa vào model sau khi đã huấn luyện, dự đoán dữ liệu test  
# Kết quả Accuracy tính bằng tay*count = len([**"ok" for** idx, label **in** enumerate(test\_labels) **if** label == predicts[idx]])  
print (**"Kết quả Accuracy được tính bằng tay là: %f"** % (float(count) / len(test\_labels)))  
*# Calculate Accuracy Rate by using accuracy\_score()*print (**"Kết quả Accuracy dùng hàm accuracy\_score() trong sklearn là: %f"** % accuracy\_score(test\_labels, predicts))

*Kết quả:*

Kết quả Accuracy được tính bằng tay là: 0.750000

Kết quả Accuracy dùng hàm accuracy\_score() trong sklearn là: 0.750000

**Confution Matrix (CM)**

Cách tính sử dụng Accuracy như ở trên chỉ cho chúng ta biết được bao nhiêu phần trăm lượng dữ liệu được phân loại đúng mà không chỉ ra được cụ thể mỗi loại được phân loại như thế nào, lớp nào được phân loại đúng nhiều nhất, và dữ liệu thuộc lớp nào thường bị phân loại nhầm vào lớp khác. Để có thể đánh giá được các giá trị này, chúng ta sử dụng một ma trận được gọi là confusion matrix.

Về cơ bản, confusion matrix thể hiện có bao nhiêu điểm dữ liệu thực sự thuộc vào một class, và được dự đoán là rơi vào một class. Để hiểu rõ hơn, hãy xem bảng dưới đây:



Có tổng cộng **165** điểm dữ liệu. Chúng ta xét ma trận tạo bởi các giá trị tại vùng **2x2** trung tâm của bảng. Ma trận thu được được gọi là Confusion Matrix. Nó là một ma trận vuông với kích thước mỗi chiều bằng số lượng lớp dữ liệu. Giá trị tại hàng thứ **i**, cột thứ **j** là số lượng điểm lẽ ra thuộc vào class **i** nhưng lại được dự đoán là thuộc vào class **j**. Như vậy, nhìn vào hàng thứ nhất (**0**), ta có thể thấy được rằng trong số **60** điểm thực sự thuộc lớp **0**, chỉ có một điểm được phân loại đúng, điểm còn lại bị phân loại nhầm vào lớp **1**.

Chúng ta có thể suy ra ngay rằng tổng các phần tử trong toàn ma trận này chính là số điểm trong tập kiểm thử. Các phần tử trên đường chéo của ma trận là số điểm được phân loại đúng của mỗi lớp dữ liệu. Từ đây có thể suy ra **accuracy** chính bằng tổng các phần tử trên đường chéo chia cho tổng các phần tử của toàn ma trận. Đoạn code dưới đây mô tả cách tính confusion matrix:

**import** numpy **as** np  
**def** confusion\_matrix(y\_true,y\_pred): *# xây dựng hàm tính ma trận* N= np.unique(y\_true).shape[0] *#tính số lớp của ma trận* cm = np.zeros((N,N)) *#khởi tạo ma trận 0 NxN lớp* **for** i **in** range(y\_true.shape[0]): *# lấy từng phần tử trong ma trận y\_true* cm[y\_true[i],y\_pred[i]] +=1  
 **return** cm  
  
y\_true = np.array([0, 0, 0, 0, 1, 1, 1, 2, 2, 2])  
y\_pred = np.array([0, 1, 0, 2, 1, 1, 0, 2, 1, 2])  
cnf\_matrix = confusion\_matrix(y\_true, y\_pred) *#gọi hàm*print(**'Confusion matrix là :'**)  
print(cnf\_matrix)  
*#Độ chính xác Accuracy được tính bằng tổng các thành phần trên đường chéo chính chia cho tổng các thành phần trên ma trận*print(**'\nĐộ chính xác Accuracy: {}%'**.format(int(np.diagonal(cnf\_matrix).sum()/cnf\_matrix.sum()\*100)))

*Kết quả:*

Confusion matrix là:

[[ 2. 1. 1.]

[ 1. 2. 0.]

[ 0. 1. 2.]]

Độ chính xác Accuracy: 60%

**True/False Positive/Negative**

Cách biểu diễn trên đây của **confusion matrix** còn được gọi là **unnormalized confusion matrix**, tức **CM** chưa chuẩn hoá. Để có cái nhìn rõ hơn, ta có thể dùng **normalized confuion matrix**, tức **CM** được chuẩn hoá. Để có **normalized confusion matrix**, ta lấy mỗi hàng của **unnormalized confusion matrix** sẽ được chia cho tổng các phần tử trên hàng đó. Như vậy, ta có nhận xét rằng tổng các phần tử trên một hàng của **normalized confusion matrix** luôn bằng **1**. Điều này thường không đúng trên mỗi cột. Dưới đây là cách tính **normalized confusion matrix**:

normalized\_confusion\_matrix = cnf\_matrix/cnf\_matrix.sum(axis=1,keepdims=**True**)  
print(**"Normalized confusion matrix là:\n {}"**.format(normalized\_confusion\_matrix))

*Kết quả:*

Normalized confusion matrix là:

[[ 0.5 0.25 0.25 ]

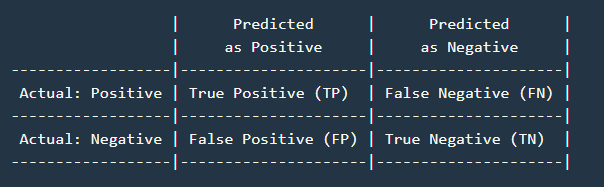
[ 0.33333333 0.66666667 0. ]

[ 0. 0.33333333 0.66666667]]

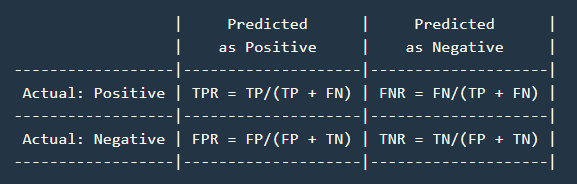
**True/False Positive/Negative**

Cách đánh giá này thường được áp dụng cho các bài toán phân lớp có hai lớp dữ liệu. Cụ thể hơn, trong hai lớp dữ liệu này có một lớp nghiêm trọng hơn lớp kia và cần được dự đoán chính xác. Ví dụ, trong bài toán xác định có bệnh ung thư hay không thì việc không bị sót (miss) quan trọng hơn là việc chẩn đoán nhầm âm tính thành dương tính. Trong bài toán xác định có mìn dưới lòng đất hay không thì việc bỏ sót nghiêm trọng hơn việc báo động nhầm rất nhiều. Hay trong bài toán lọc email rác thì việc cho nhầm email quan trọng vào thùng rác nghiêm trọng hơn việc xác định một email rác là email thường.

Trong những bài toán này, người ta thường định nghĩa lớp dữ liệu quan trọng hơn cần được xác định đúng là lớp Positive (P-dương tính), lớp còn lại được gọi là Negative (N-âm tính). Ta định nghĩa True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN) dựa trên confusion matrix chưa chuẩn hoá như sau:



Người ta thường quan tâm đến **TPR, FNR, FPR, TNR** (R - Rate) dựa trên normalized confusion matrix như sau:



**False Positive Rate** còn được gọi là **False Alarm Rate** (tỉ lệ báo động nhầm), **False Negative Rate** còn được gọi là **Miss Detection Rate** (tỉ lệ bỏ sót). Trong bài toán dò mìn, thà báo nhầm còn hơn bỏ sót, tức là ta có thể chấp nhận **False Alarm Rate** cao để đạt được **Miss Detection Rate** thấp.

**Chú ý:**

* Việc biết một cột của **confusion matrix** này sẽ suy ra được cột còn lại vì tổng các hàng luôn bằng 1 và chỉ có hai lớp dữ liệu.
* Với các bài toán có nhiều lớp dữ liệu, ta có thể xây dựng bảng **True/False Positive/Negative** cho mỗi lớp nếu coi lớp đó là lớp Positive, các lớp còn lại gộp chung thành lớp Negative, giống như cách làm trong **one-vs-rest**.

**Receiver operating characteristic curve**

Trong một số bài toán, việc tăng hay giảm **FNR**, **FPR** có thể được thực hiện bằng việc thay đổi một ngưỡng (***threshold***) nào đó. Lấy ví dụ khi ta sử dụng thuật **toán Logistic Regression**, đầu ra của mô hình có thể là các lớp cứng **0** hay **1**, hoặc cũng có thể là các giá trị thể hiện xác suất để dữ liệu đầu vào thuộc vào lớp **1**. Khi sử dụng thư viện ***sklearn Logistic Regression***, ta có thể lấy được các giá trị xác xuất này bằng phương thức ***predict\_proba()***. Mặc định, ngưỡng được sử dụng là **0.5**, tức là một điểm dữ liệu **x** sẽ được dự đoán rơi vào lớp **1** nếu giá trị ***predict\_proba(x)*** lớn hơn **0.5** và ngược lại.

Nếu bây giờ ta coi lớp **1** là lớp **Positive**, lớp **0** là lớp **Negative**, câu hỏi đặt ra là làm thế nào để tăng mức độ báo nhầm (**FPR**) để giảm mức độ bỏ sót (**FNR**)? Chú ý rằng tăng **FNR** đồng nghĩa với việc giảm **TPR** vì tổng của chúng luôn bằng 1.

Một kỹ thuật đơn giản là ta thay giá trị ***threshold*** từ 0.5 xuống một số nhỏ hơn. Chẳng hạn nếu chọn ***threshold*** = 0.3, thì mọi điểm được dự đoán có xác suất đầu ra lớn hơn 0.3 sẽ được dự đoán là thuộc lớp Positive. Nói cách khác, tỉ lệ các điểm được phân loại là **Positive** sẽ tăng lên, kéo theo cả **False Positive Rate** và **True Positive Rate** cùng tăng lên (cột thứ nhất trong ma trận tăng lên). Từ đây suy ra cả **FNR** và **TNR** đều giảm.

Ngược lại, nếu ta muốn bỏ sót còn hơn báo nhầm, tất nhiên là ở mức độ nào đó, như bài toán xác định email rác chẳng hạn, ta cần tăng ***threshold*** lên một số lớn hơn 0.5. Khi đó, hầu hết các điểm dữ liệu sẽ được dự đoán thuộc lớp **0**, tức **Negative**, và cả **TNF** và **FNR** đều tăng lên, tức **TPR** và **FPR** giảm xuống.

Như vậy, ứng với mỗi giá trị của ***threshold***, ta sẽ thu được một cặp (**FPR**, **TPR**). Biểu diễn các điểm (**FPR**, **TPR**) trên đồ thị khi thay đổi ***threshold*** từ 0 tới 1 ta sẽ thu được một đường được gọi là **Receiver Operating Characteristic curve** hay **ROC curve**. (Chú ý rằng khoảng giá trị của ***threshold*** không nhất thiết từ 0 tới 1 trong các bài toán tổng quát. Khoảng giá trị này cần được đảm bảo có trường hợp **TPR/FPR** nhận giá trị lớn nhất hay nhỏ nhất mà nó có thể đạt được).

Dưới đây là một ví dụ với hai lớp dữ liệu. Lớp thứ nhất là lớp Negative có 20 điểm dữ liệu, 30 điểm còn lại thuộc lớp Positive. Giả sử mô hình đang xét cho các đầu ra của dữ liệu (xác suất) được lưu ở biến *scores*.

**import** numpy **as** np  
**from** sklearn.metrics **import** roc\_curve, auc  
n0, n1 = 20, 30  
score0 = np.random.rand(n0)/2  
label0 = np.zeros(n0, dtype = int)  
score1 = np.random.rand(n1)/2 + .2  
label1 = np.ones(n1, dtype = int)  
scores = np.concatenate((score0, score1))  
y\_true = np.concatenate((label0, label1))  
fpr, tpr, thresholds = roc\_curve(y\_true, scores, pos\_label = 1)  
print(**"True lables: \n{}\nScores: \n{}\nThresholds:\n{}\nFPR:\n{}\nTPR: \n{}\n"**.format(y\_true,scores,thresholds,fpr,tpr))

*Kết quả:*

True lables:

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

Scores:

[ 0.00399151 0.25826574 0.16858168 0.03329907 0.37570748 0.20519163

0.14683173 0.33808756 0.30511688 0.34295643 0.42260856 0.10433163

0.2062915 0.04314333 0.22564941 0.30344634 0.41291515 0.14160504

0.30188107 0.27875212 0.2734653 0.49132647 0.3026426 0.30907854

0.52039594 0.64291785 0.56946258 0.651503 0.68220519 0.47964084

0.63036443 0.45834015 0.22527505 0.24574169 0.47388309 0.23264289

0.33615167 0.46375379 0.45881991 0.55189626 0.49818998 0.34390935

0.5318267 0.65937194 0.27529899 0.20633514 0.47168178 0.6369788

0.61766553 0.26678961]

Thresholds:

[ 0.68220519 0.45834015 0.37570748 0.34390935 0.33808756 0.30907854

0.30344634 0.3026426 0.27875212 0.26678961 0.25826574 0.23264289

0.22564941 0.20633514 0.00399151]

FPR:

[ 0. 0. 0.15 0.15 0.25 0.25 0.35 0.35 0.45 0.45 0.5 0.5 0.55 0.55 1. ]

TPR:

[ 0.03333333 0.63333333 0.63333333 0.66666667 0.66666667 0.73333333

0.73333333 0.76666667 0.76666667 0.86666667 0.86666667 0.93333333

0.93333333 1. 1. ]

***Nhận xét:***

Như vậy, ứng với *threshold = 0.69637251*, *fpr = 0* và *tpr = 0.03*. Đây không phải là một ngưỡng tốt vì mặc dụ **False Positive Rate** (**FPR**) thấp, **True Positive Rate (TPR)** cũng rất thấp. Chúng ta luôn muốn rằng **FPR** thấp và **TPR** cao.

Dựng hình đường **ROC:**

**import** matplotlib.pyplot **as** plt  
**from** itertools **import** cycle  
plt.figure()  
lw = 2  
plt.plot(fpr, tpr, color=**'darkorange'**,  
 lw=lw, label=**'Đường cong ROC (area = %0.2f)'** % auc(fpr, tpr))  
plt.plot([0, 1], [0, 1], color=**'navy'**, lw=lw, linestyle=**'--'**)  
plt.xlim([0.0, 1.0])  
plt.ylim([0.0, 1.05])  
plt.xlabel(**'False Positive Rate'**)  
plt.ylabel(**'True Positive Rate'**)  
plt.title(**'Ví dụ về đường cong Receiver operating characteristic'**)  
plt.legend(loc=**"lower right"**)  
plt.show()



***Hình 1: Mô tả đường ROC***

**Area Under the Curve**

Dựa trên đường cong **ROC**, ta có thể chỉ ra rằng một mô hình có hiệu quả hay không. Một mô hình hiệu quả khi có **FPR** thấp và **TPR** cao, tức tồn tại một điểm trên đường cong **ROC** gần với điểm có toạ độ **(0, 1)** trên đồ thị (góc trên bên trái). Curve càng gần thì mô hình càng hiệu quả.

Có một thông số nữa dùng để đánh giá mà tôi đã sử dụng ở trên được gọi là **Area Under the Curve** hay **AUC**. Đại lượng này chính là diện tích nằm dưới **ROC** **curve** màu cam. Giá trị này là một số dương nhỏ hơn hoặc bằng 1. Giá trị này càng lớn thì mô hình càng tốt.

**Precision và Recall**

**Định nghĩa:**

Với bài toán phân loại mà tập dữ liệu của các lớp là chênh lệch nhau rất nhiều, có một phép đó hiệu quả thường được sử dụng là **Precision-Recall**.

Trước hết xét bài toán phân loại nhị phân. Ta cũng coi một trong hai lớp là **positive**, lớp còn lại là **negative**.

Xét Hình 3 dưới đây:



***Hình 2: Cách tính Precision và Recall***

Với một cách xác định một lớp là positive, Precision được định nghĩa là tỉ lệ số điểm true positive trong số những điểm được phân loại là positive (TP + FP).

Recall được định nghĩa là tỉ lệ số điểm true positive trong số những điểm thực sự là positive (TP + FN).

Một cách toán học, Precison và Recall là hai phân số có tử số bằng nhau nhưng mẫu số khác nhau:

Bạn đọc có thể nhận thấy rằng TPR và Recall là hai đại lượng bằng nhau. Ngoài ra, cả Precision và Recall đều là các số không âm nhỏ hơn hoặc bằng một.

Precision cao đồng nghĩa với việc độ chính xác của các điểm tìm được là cao. Recall cao đồng nghĩa với việc True Positive Rate cao, tức tỉ lệ bỏ sót các điểm thực sự positive là thấp.

Ví dụ nhỏ dưới đây thể hiện cách tính Precision và Recall dựa vào Confusion Matrix cho bài toán phân loại nhị phân.

**from** \_\_future\_\_ **import** print\_function  
**import** numpy **as** np  
*# confusion matrix to precision + recall***def** cm2pr\_binary(cm):  
 p = cm[0,0]/np.sum(cm[:,0])  
 r = cm[0,0]/np.sum(cm[0])  
 **return** (p, r)  
  
*# example of a confusion matrix for binary classification problem*cm = np.array([[100., 10], [20, 70]])  
p,r = cm2pr\_binary(cm)  
print(**"precition = {0:.2f}, recall = {1:.2f}"**.format(p, r))

*Kết quả:*

precition = 0.83, recall = 0.91

Khi Precision = 1, mọi điểm tìm được đều thực sự là positive, tức không có điểm negative nào lẫn vào kết quả. Tuy nhiên, Precision = 1 không đảm bảo mô hình là tốt, vì câu hỏi đặt ra là liệu mô hình đã tìm được tất cả các điểm positive hay chưa. Nếu một mô hình chỉ tìm được đúng một điểm positive mà nó chắc chắn nhất thì ta không thể gọi nó là một mô hình tốt.

Khi Recall = 1, mọi điểm positive đều được tìm thấy. Tuy nhiên, đại lượng này lại không đo liệu có bao nhiêu điểm negative bị lẫn trong đó. Nếu mô hình phân loại mọi điểm là positive thì chắc chắn Recall = 1, tuy nhiên dễ nhận ra đây là một mô hình cực tồi.

Một mô hình phân lớp tốt là mô hình có cả Precision và Recall đều cao, tức càng gần một càng tốt. Có hai cách đo chất lượng của bộ phân lớp dựa vào Precision và Reall: Precision-Recall curve và F-score.

Khi kích thước các lớp dữ liệu là chênh lệch (imbalanced data hay skew data), precision và recall thường được sử dụng.