ỨNG DỤNG DỮ LIỆU LỚN

PROJECT 3: COMMUNITY DETECTION

Contents

1. Giới thiệu:....3
1.1 Mô tả đồ án:....3
1.2 Ngôn ngữ:....3
2. Nội dung thực hiện:....4
2.1 Lọc dữ liệu từ tập dữ liệu thô:....4
2.2 Task 1:....4
2.2.1. Tạo cấu trúc lưu đồ thị:....4
2.2.2. Các hàm đã tạo ra để phục vụ task 1:....5
2.2.3. Thực hiện task 1.1....6
2.2.4. Thực hiện task 1.2....7
2.3 Task 2:....7
2.4 So sánh kết quả task 1 và task 2:....8
So sánh kết quả task 1 và task 2:....8
So sánh kết quả task 1 với sự thay đổi modularity và task 2:....8
3. Tài liệu tham khảo:....8

1.Giới thiệu:

1.1 Mô tả đồ án:

Sử dụng thuật toán Girvan-Newman để phát hiện các công đồng trong mạng lưới mạng xã hội .

1.2 Ngôn ngữ:

Python .

1.3 Source code:

Task 1: UDDLL_3_1.ipynb

Task 2: UDDLL_3_2.ipynb

1.4Thông tin nhóm :

Họ và tên	MSSV
Đào Quốc Phong	18120505
Đoàn Thanh Quang	18120525
Nguyễn Như Quang	18120528
Dương Đoàn Bảo Sơn	18120533

2.Nội dung thực hiện:

2.1 Lọc dữ liệu từ tập dữ liệu thô:

Tập dữ liệu ub_sample_data.csv bao gồm 2 cột user_id và business_id là sở thích khinh doanh của user.

Tạo spark DataFrame từ tập dữ liệu rồi dùng sql để xử lý dữ liệu.

Kết hợp tập dữ liệu với chính nó(lấy tích Cartesian data1 và data2) gọi u1 là user_id của data1 và u2 là user_id của data2. Chỉ lấy các hàng có u1 < u2 (theo thứ tự từ điển, điều này tránh luôn cả việc u1 = u2, và đảm bảo yêu cầu đề bài), và lấy thêm điều kiện data1.business_id=data2.business_id. Sau đó dùng groupy u1, u2 và lấy count >= 7.

Sau lệch SQL như trên ta được DataFrame với mỗi dòng lưu một cạnh (u_i, u_j) của đồ thị. Dữ liệu này được dùng cho cả task 1 và task 2.

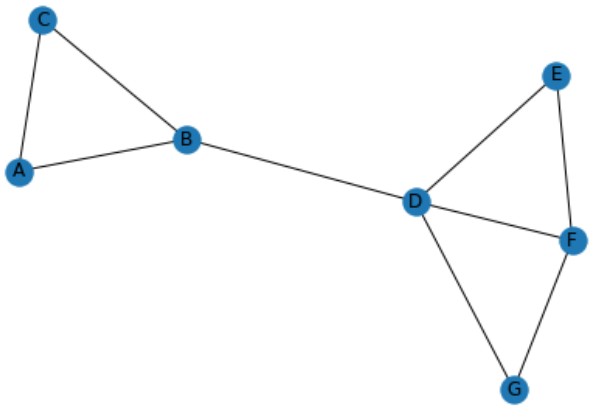
2.2 Task 1:

2.2.1. Tạo cấu trúc lưu đồ thị:

Cấu trúc lưu giữ dữ liệu đồ thị lấy tư tưởng từ danh sách kề.

Để thuật tiện cho việc truy xuất dữ liệu thì thay vì dùng list, nhóm đã dùng dictionary của python.

Việc lưu trữ này hoàn toàn giống với cách Networkx lưu trữ danh sách kề của đồ thị.(G.adj). Hình minh hoạ ở dưới.




```
{'A': {'B': 0, 'C': 0},
 'B': {'A': 0, 'C': 0, 'D': 0},
 'C': {'A': 0, 'B': 0},
 'D': {'B': 0, 'E': 0, 'F': 0, 'G': 0},
 'E': {'D': 0, 'F': 0},
 'F': {'D': 0, 'E': 0, 'G': 0},
 'G': {'D': 0, 'F': 0}}
```

Từ cách lưu trữ trên ta có thể lấy nhanh chóng tập các node kề với một node(gọi là tập A) trong graph G : A ={ G[node].keys() }. Từ đây để tạo tập cạnh kề với node là E = { (node, e) | e ∈ A }

Ta có thể lấy thuộc tính của cạnh bằng cách lấy G[u_i][u_j]. Trong đồ án này chúng ta chỉ quan tâm tới 1 thuộc tính đó là betweenness của cạnh, vì vậy cạnh trong đồ án chỉ có một thuộc tính duy nhất và có giá trị khởi tạo là 0.

Một cạnh trong đồ thị VD: cạnh (A,B) sẽ được lưu trữ 2 lần là cạnh (A,B) và (B,A).

Điều này làm cho việc duyệt đồ thị trở nên dễ dàng hơn, ta có thể đi từ đỉnh A sang đỉnh B và ngược lại(nếu như chỉ lưu một cạnh vd: (A,B) thì ta không thể đi từ đỉnh B ngược về đỉnh A). Vì vậy số cạnh khi lưu trữ đồ thị sẽ nhiều gấp đôi số cạnh thực tế, khi thêm vào, bớt đi hoặc cập nhật ta phải thực hiện trên cả 2 cạnh này.

2.2.2. Các hàm đã tạo ra để phục vụ task 1:

Từng bước cụ thể của các hàm được giải thích trong code.

- get_data(graph, edge_df): đưa dữ liệu các cạnh từ spark dataframe vào đồ thị.
- reset_weight(graph): đặt lại betweenness cho các cạnh của đồ thị về 0.
- BFS_gen_tree(graph, root): Tạo ra cây BFS từ đồ thị với node là nút gốc.

  Các node trong mỗi level được lưu vào một dictionary, với mỗi node có 2 thuộc tính là số đường ngắn nhất từ nút gốc tới node đó và credit của node. Cây BFS là danh sách các level này theo thứ tự từ level 0 tới level n.

```
[{'E': {'number_shortest_path': 1, 'credit': 1}},
 {'D': {'number_shortest_path': 1, 'credit': 1},
  'F': {'number_shortest_path': 1, 'credit': 1}},
 {'B': {'number_shortest_path': 1, 'credit': 1},
  'G': {'number_shortest_path': 2, 'credit': 1}},
 {'A': {'number_shortest_path': 1, 'credit': 1},
  'C': {'number_shortest_path': 1, 'credit': 1}}]
```

  Sau khi chạy hàm sẽ tính số đường ngắn nhất từ nút gốc tới node và giá trị khởi tạo credit cho các node là 1 (vì credit của tất cả node sẽ được cộng thêm 1 ở một thời điểm nào đó).

  Giá trị trả về là cây BFS.

- calc_betweeness(graph, tree): tính, và cập nhật betweenness cho các cạnh tực tiếp vào đồ thị. Betweenness được tính theo cây BFS được tạo ra bởi hàm trước.
- remove_between_edge(graph): Tìm và xoá cạnh có độ betweenness lớn nhất trong đồ thị.
- get_communities(graph): Từ đồ thị, tạo ra danh sách các cộng đồng trong đồ thị với mỗi cộng đồng là một danh sách gồm các node trong cộng đồng. Dưới dây là hình minh hoạ.

```
[['A', 'B', 'C'], ['D', 'E', 'F', 'G']]
```

  Dùng BFS duyệt đồ thị tới khi nào không thể duyệt tiếp được thì ta được một cộng đồng. Tiếp tục duyệt với một node khác(node chưa được duyệt) để được một cộng đồng khác. Tiếp tục cho tới khi duyệt hết tất cả các node trong đồ thị.

- modularity(graph, communities, A, m): Tính modularity của đồ thị, với danh sách các cộng đồng. Hàm tính modularity là hàm được đưa ra ở mục 2.2 Community Detection của đề bài.

Dữ liệu đầu vào là đồ thị, danh sách các cộng đồng dùng để tính modularity, A là danh sách kề của đồ thị gốc. m là số cạnh thực tế của đồ thị. Dưới đây là hình minh hoạ cho danh sách kề của đồ thị gốc.

```
{'A': ['B', 'C'],
 'B': ['A', 'C'],
 'C': ['A', 'B'],
 'D': ['E', 'F', 'G'],
 'E': ['D', 'F'],
 'F': ['D', 'E', 'G'],
 'G': ['D', 'F']}
```

Kết quả trả về là điểm modularity của cộng đồng đó trong đồ thị.

- modularity2(graph, communities, A, m): Tương tự như hàm trên. Ở hàm này nhóm mô phỏng lại cách tính modularity của Networkx. Hàm này để kiểm tra thử thuật toán Girvan-newman của nhóm có chạy giống với Girvan-newman của Networkx không, có đưa ra được kết quả chính xác mà Networkx đưa ra không.
- girvan_newman(ori_graph , modularity=modularity): Hàm chạy thuật toán Girvan_Newman.

  Trong mỗi vòng lặp tạo BFS tree với nút gốc lần lược là mỗi nút trong đồ thị, sau đó tính betweenness cho các cạnh của đồ thị theo từng BFS tree. Điều này dẫn đến việc các cạnh sẽ được tính betweenness 2 lần (từ root tới node, và khi node thành root thì từ node tới root cũ sẽ được tính lại một lần nữa) nên khi cập nhật betweenness cho cạnh ta chia đôi giá trị này. Thuật toán không ảnh hưởng khi trong đồ thị có nhiều cộng đồng, vì BFS tree chỉ tạo được cây cho một cộng đồng(không chứa tất cả các node trong đồ thị) thì khi tính betweenness chỉ những cạnh có trong cây mới được cập nhật.

  Sau khi cập nhật betweenness cho tất cả các cạnh. Tiến hành xoá cạnh có betweenness cao nhất.

  Tìm ra danh sách các cộng đồng trong đồ thị và tính modularity. Cập nhật đồ thị có modularity cao nhất.

  Đặt lại giá trị betweenness cho các cạnh về 0 để tiến hành chạy lần sau.

  Kết quả xuất ra là danh sách các cộng đồng có modularity cao nhất và modularity của nó. Dưới đây là ví dụ trả về của hàm.

```
([['A', 'B', 'C'], ['D', 'E', 'F', 'G']], 0.3641975308641976)
```

2.2.3. Thực hiện task 1.1

Tạo graph và gọi hàm get_data để đưa dữ liệu vào đồ thị.

Lấy danh sách các nút trong đồ thị.

Dùng hàm BFS_gen_tree để tạo BFS tree với nút gốc lần lược là các nút trong danh sách trên.

Cập nhật betweenness cho các cạnh dựa trên BFS tree bằng hàm calc_betweenness.

Xuất kết quả ra tập tin theo yêu cầu.

Gọi hàm reset_weight đặt lại giá trị betweenness cho graph, chuẩn bị cho task sau.

2.2.4. Thực hiện task 1.2

Gọi hàm Girvan_newman

Xuất ra giá trị modularity cao nhất: 0.6872550442734674

Xuất kết quả ra tập tin như yêu cầu.

2.3 Task 2:

Dùng thư viện NetworkX để thực hiện lại công việc trong Task 1.

Tải tập tin chứa danh sách các cạnh đã tạo ra ở công việc trước.

Đưa dữ liệu từ tập tin vào pandas DataFrame để thuận tiện cho việc đưa các cạnh vào graph của NetworkX.

Khởi tạo graph với hàm khởi tạo nx.Graph

Gọi hàm add_edges_from với biến đầu vào là giá trị trong pandas DataFrame chứ dữ liệu các cạnh.

Sau khi có được đồ thị với các cạnh. Tiến hành chạy thuật toán Girvan_newman.

Hàm tạo ra một generator, với mỗi lần generate sẽ tạo ra một danh sách các community khác nhau.

Tiến hành lặp generator(tương tự như list) tính modularity bằng hàm modularity của NetworkX, lưu lại giá trị modularity cao nhất và cộng đồng đó.

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

$$Q = \sum_{c=1}^{n} \left[\frac{L_c}{m} - \gamma \left(\frac{k_c}{2m} \right)^2 \right]$$

Đây là công thức NetworkX dùng để tính modularity. Công thức ở trên tương tự như công thức đưa ra ở mục 2.2 trong yêu cầu đồ án. Công thức dưới là công thức thu gọn từ công thức trên, hàm modularity2 ở task1 mô phỏng lại công thức này.

Xuất ra modularity cao nhất: 0.6872550442734798

Xuất kết quả vào tập tin như yêu cầu.

2.4 So sánh kết quả task 1 và task 2:

So sánh kết quả task 1 và task 2:

```
[True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, False]
Modulairy in task 1:  0.6872550442734674
Modulairy in task 2:  0.6872550442734798
```

Ở trên là kết quả so sánh từng dòng của 2 tập tin lưu kết quả của task 1 và task 2.

Ta thấy được tất cả các dòng của 2 tập tin đều giống nhau trừ dòng cuối cùng là dòng chứa modularity. Điều này thể hiện các community mà 2 task này đưa ra là hoàn toàn giống nhau.

Khác nhau ở modularity ở 2 task đưa ra là không đáng kể, khác nhau chỉ xuất hiện ở giá trị thập phân thứ 14.

So sánh kết quả task 1 với sự thay đổi modularity và task 2:

```
[True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True]
Modulairy in task 1:  0.6872550442734798
Modulairy in task 2:  0.6872550442734798
```

Hình trên cho thấy kết quả của thuật toán mà nhóm tạo ra giống hoàn toàn với thuật toán của NetworkX.

3.Tài liệu tham khảo:

Tài liệu môn học
Slide bài giảng.
Các trang hướng dẫn, chi tiết hàm, và source code của NetworkX
Wikipedia về modularity(network)
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.quality.modularity.html#modularity
https://en.wikipedia.org/wiki/Modularity_(networks)
https://networkx.org/documentation/stable/tutorial.html