TP1 + TP2 + TP3

Neo4j report : Tourism Circulation Dataset

Stella HU

Graph Mining DIA2

The dataset is used to study the circulation of tourists on a geographic territory

I – Dataset Importation

1) Bi-partite

The bi-partite dataset is made of 3 csv files :

- reviews.csv
- users.csv
- gadm36_4.csv.

The **reviews.csv** file is cut to 10 parts as it is very heavy. It contains columns such as :

Column	Description
user_id	Numbers that represent a user's id that has reviewed
gid_to	Area 4 geographical identifier, representing the destination city's geographical identifier
year	Year of the rating
rating	Rating of the location out of 5
NB	Number of ratings of the corresponding user

The **users.csv** file contains two columns :

Column	Description
user_id	Numbers that represent a user's id that has reviewed
country	The origin country of the user

Finally, the **gadm36_4.csv** file contains columns that store information according to administrative areas all around the world :

Column	Description
gid	Global geographic identifier of the area
gid_0	Geographic identifier of the country (area 0)
name_0	Name of the country

gid_1	Geographic identifier of the region (area 1)
name_1	Name of the region
gid_2	Geographic identifier of the department (area 2)
name_2	Name of the department
gid_3	Geographic identifier of the district (area 3)
name_3	Name of the district
gid_4	Geographic identifier of the city (area 4)
name_4	Name of the city

After putting all the files in the “import” repository for the bi-partite dataset, we must create indexes before importing the nodes.

Creation of indexes :

```
CREATE INDEX FOR (u:User) ON (u.id);
CREATE INDEX FOR (u:User) ON (u.country);
CREATE INDEX FOR (a:Area_4) ON (a.gid);
CREATE INDEX FOR (a:Area_4) ON (a.gid_4);
```

Once we have created indexed for each properties of each nodes, we need to import the corresponding nodes and relationship with our csv files.

Importation of users.csv and creation of the users nodes :

```
:auto LOAD CSV WITH HEADERS FROM "file:/users.csv" as l FIELDTERMINATOR "\t"
MERGE (user:User{id:toInteger(l.user_id), country:l.country});
```

MERGE function allows the creation of a `User` type node with the indicated properties and reuses it if it already exists.

Importation of gadm36_4.csv and creation of Area_4 nodes :

```
:auto LOAD CSV WITH HEADERS FROM "file:/gadm36_4.csv" AS l FIELDTERMINATOR "\
CREATE (loc:Area_4 {
  gid: toInteger(l.gid),
  name: l.nom,
  gid_0: l.gid_0,
  name_0: l.name_0,
  gid_1: l.gid_1,
  name_1: l.name_1,
  gid_2: l.gid_2,
  name_2: l.name_2,
  gid_3: l.gid_3,
  name_3: l.name_3,
  gid_4: l.gid_4,
```

```
  name_4: l.name_4
});
```

The function creates `Area_4` (city) type nodes with properties corresponding to the geographic data contained in the loaded file.

Finally, we can import the relationships with the reviews.csv file, cut in 10.

Importation of reviews.csv and creation of the relationships between an user and a city :

```
:auto LOAD CSV WITH HEADERS FROM "file:/reviews_0.csv" as l FIELDTERMINATOR "
MERGE (area:Area_4{gid_4:l.gid_to} )
MERGE (user:User{id:toInteger(l.user_id)} )
MERGE (user) -[:review{year:toInteger(l.year),rating:toFloat(l.rating),
NB:toInteger(l.NB)}]-> (area);
```

We do this query 10 times, replacing 0 by numbers from 1 to 9. These cypher queries create a `review` relationship between the `User` and the `Area_4` node, indicating that a user has left a review for a particular geographic area or city.

Now that we have imported all the nodes and relationship we needed, we can do the same for the Mono-partite dataset.

2) Mono-partite

The mono-partite dataset is made of 2 csv files :

- circulationGraph_4.csv
- gadm36.csv.

The **circulationGraph_4.csv** contains 7 columns :

Column	Description
gid_from	Area 4 geographical identifier, representing the origin city's geographical identifier
gid_to	Area 4 geographical identifier, representing the destination city's geographical identifier
year	Year of the review
month	Month of the review (empty)
country	Origin country of the user
age	Age of the user (empty)
NB	Number of reviews of the user

The **gadm36.csv** file contains columns that store information according to administrative areas all around the world with their latitude and longitude :

Column	Description
gid	Global geographic identifier of the area
gid_0	Geographic identifier of the country (area 0)
gid_1	Geographic identifier of the region (area 1)
gid_2	Geographic identifier of the department (area 2)
gid_3	Geographic identifier of the district (area 3)
gid_4	Geographic identifier of the city (area 4)
gid_5	Geographic identifier of the town (area 5)
name_0	Name of the country
name_1	Name of the region
name_2	Name of the department
name_3	Name of the district
name_4	Name of the city
name_5	Name of the town
centroidlat	Centroid latitude
centroidlong	Centroid longitude

The mono-partite dataset is a transformation of the bi-partite previously imported dataset. The process involved grouping locations from the GADM 3.6 dataset at level 4 (cities) and creating links between pairs of cities for each user review. These links are then aggregated by country of origin and year, resulting in weighted relationships that reflect user interactions. A Java program automates the extraction of this circulation graph connecting the cities based on user activity. Finally, the mono-partite dataset consists of `Area_4` (city) nodes and `circulation` relationships between them.

We must create indexes on the **gid** and **gid_4** properties of `Area_4` type nodes.

Creation of indexes :

```
CREATE INDEX FOR (a:Area_4) ON (a.gid);
CREATE INDEX FOR (a:Area_4) ON (a.gid_4);
```

We can now import the nodes and properties (here 1 type of node and 1 type of relationship) by loading the csv files.

Importation of gadm36.csv and creation of Area nodes:

```
:auto LOAD CSV WITH HEADERS FROM "file:/gadm36.csv" as l FIELDTERMINATOR "\t"
MERGE (loc:Area_4{
    gid:toInteger(l.gid),
  gid_0:l.gid_0,
  name_0:l.name_0,
  gid_1:l.gid_1,
```

```
  name_1:l.name_1,
  gid_2:l.gid_2,
  name_2:l.name_2,
  gid_3:l.gid_3,
  name_3:l.name_3,
  gid_4:l.gid_4,
  name_4:l.name_4});
```

We stop at level 4 to get the cities.

Importation of circulationGraph_4.csv and creation of Circulation relationships :

```
:auto LOAD CSV WITH HEADERS FROM "file:/circulationGraph_4.csv" as l FIELDTER
MERGE (from:Area_4{gid:toInteger(l.gid_from)} )
MERGE (to:Area_4{gid:toInteger(l.gid_to)} )
MERGE (from) -[:trip{year:toInteger(l.year),
NB:toInteger(l.NB),country:l.country}]-> (to);
```

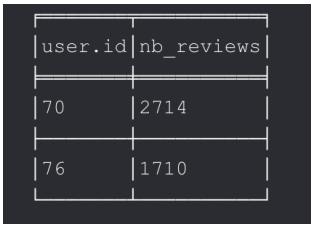
We have all our nodes and relationships to work on the datasets now.

II - Mining Bi-partite graphs

1) Similarity

1.1 We want to take the two French users who reviewed the most :

```
MATCH (user:User{country: 'France'})-[r:review]->(city:Area_4)
WITH user, SUM(r.NB) AS nb_reviews
ORDER BY total_reviews DESC LIMIT 2
RETURN user.id, nb_reviews
```



user.id	nb_reviews
70	2714
76	1710

This query matches all users (`User` node) that have reviewed (`review` relationship) a particular city (`Area_4` node), while applying a filter to take the French ones only. It aggregates the number of review for each user and returns the two bigger numbers and their associated user ID.

We can therefore see that the two French people that have reviewed the most are the user 76 and user 70, with more than 1000 reviews ahead of the second one.

1.2 Let's get their Jacquard similarity between distinct areas :

```
MATCH (user:User{country: 'France'})-[r:review]->(city:Area_4)
WITH user, SUM(r.NB) AS nb_reviews
ORDER BY nb_reviews DESC
LIMIT 2
WITH COLLECT(user.id) AS FrenchUsersMost

MATCH (user_1:User)-[:review]->(city_1:Area_4)
WHERE user_1.id = FrenchUsersMost[0]
WITH user_1, COLLECT(DISTINCT id(city_1)) AS user_1_city, FrenchUsersMost
MATCH (user_2:User)-[:review]->(city_2:Area_4)
WHERE user_2.id = FrenchUsersMost[1]
WITH user_1, user_1_city, user_2, COLLECT(DISTINCT id(city_2)) AS user_2_city

RETURN user_1.id, user_2.id, gds.similarity.jaccard(user_1_city, user_2_city)
AS Jaccard_similarity
```

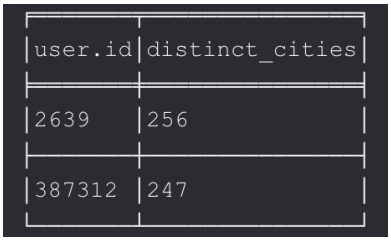
user_1.id	user_2.id	Jaccard_similarity
70	76	0.08359133126934984

We use the previous query to get the two top users' ID, then we collect each user's visited distinct cities. Finally, we compute their Jaccard similarity with `gds.similarity.jaccard()` function thanks to the Graph Data Science Library plug-in.

We therefore get a Jaccard similarity of **0.08** between the top two French reviewers. It is very close to 0, which means that their overlap in terms of cities they have both reviewed is minimal, so the two users have evaluated **very few** common cities.

1.3 We want to take the two French users who reviewed the most areas and give their similarity :

```
MATCH (user:User{country:'France'})-[r:review]->(city:Area_4)
RETURN user.id, count(DISTINCT city) AS distinct_cities ORDER BY distinct_cit
DESC LIMIT 2;
```



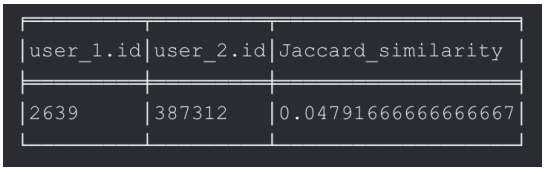
user.id	distinct_cities
2639	256
387312	247

This query matches all French users that have reviewed a particular city. It aggregates the number of distinct reviewed cities for each user and returns the two bigger numbers and their associated user ID.

> We can therefore see that the two French people that have reviewed the most distinct cities are user 2639 with 256 reviewed cities and user 387312, with 247 reviewed cities.

To get their similarity :

```
MATCH (user:User {country: 'France'})-[r:review]->(city:Area_4)
WITH user, COUNT(DISTINCT city) AS distinct_cities
ORDER BY distinct_cities DESC
LIMIT 2
WITH COLLECT(user.id) AS FrenchUsersMostAreas

MATCH (user_1:User)-[:review]->(city_1:Area_4)
WHERE user_1.id = FrenchUsersMostAreas[0]
WITH user_1, COLLECT(DISTINCT id(city_1)) AS user_1_city, FrenchUsersMostArea

MATCH (user_2:User)-[:review]->(city_2:Area_4)
WHERE user_2.id = FrenchUsersMostAreas[1]
WITH user_1, user_1_city, user_2, COLLECT(DISTINCT id(city_2)) AS user_2_city

RETURN user_1.id, user_2.id,
gds.similarity.jaccard(user_1_city, user_2_city) AS Jaccard_similarity
```



user_1.id	user_2.id	Jaccard_similarity
2639	387312	0.04791666666666667

We use the previous query to get the two top french users' ID who went to the most distinct cities, then we collect each user's visited distinct cities. Finally, we compute their Jaccard similarity with `gds.similarity.jaccard()` function thanks to the Graph Data Science Library plug-in.

💡 This low (**0.4**) Jaccard similarity indicates that despite the big number of distinct areas they reviewed, the number of cities **in common** they have visited is **very low**. It means that they **don't frequent the same areas**.

1.4 What is the difference ?

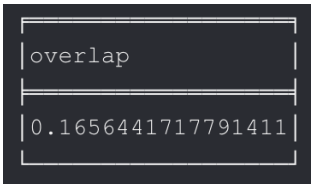
💡 The **difference** between the similarity of the two top French users who reviewed the most and the two top French users who reviewed the most areas is the fact that the first situation is **quantity-based** : the number of distinct areas reviewed might be smaller because **a user can review a same place multiple times**. On the other hand, the second situation is based on the **diversity of reviewed cities** and not the number of reviews.

Their Jaccard similarities show that in both cases, the **users don't frequent the same places**, showing that they must have different interests, especially for the two top French users who reviewed the most areas.

1.5 Overlaps for the two couples :

- For the two French users who reviewed the most :

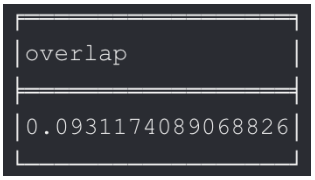
```
MATCH (user1:User {id: 70})-[:review]->(city1:Area_4)
WITH collect(DISTINCT id(city1)) AS user1Cities

MATCH (user2:User {id: 76})-[:review]->(city2:Area_4)
WITH user1Cities, collect(DISTINCT id(city2)) AS user2Cities

RETURN gds.similarity.overlap(user1Cities, user2Cities) AS overlap;
```



We already know the IDs of the two French users who reviewed the most. This cypher query gives us the overlap (using `gds.similarity.overlap()` function) between the distinct reviewed cities of each user. The result is a normalized overlap, so it's a value between 0 and 1. Here, the overlap of 0.17 indicates that 17% of the cities reviewed by the 2nd French user who reviewed the most are reviewed by both users.

- For the two French users who reviewed the most places :

```
MATCH (user1:User {id: 2639})-[:review]->(city1:Area_4)
WITH collect(DISTINCT id(city1)) AS user1Cities

MATCH (user2:User {id: 387312})-[:review]->(city2:Area_4)
WITH user1Cities, collect(DISTINCT id(city2)) AS user2Cities

RETURN gds.similarity.overlap(user1Cities, user2Cities) AS overlap;
```



overlap
0.0931174089068826

Like the previous case, here we are trying to compute the normalized overlap similarity between the distinct cities reviewed by the top two french users who reviewed the most unique places, using their ID and the `gds.similarity.overlap()` function. We get an overlap of 0.09, which is pretty low and shows that less than 10% of reviewed cities by the 2nd French user who reviewed the most unique cities are reviewed by both users.

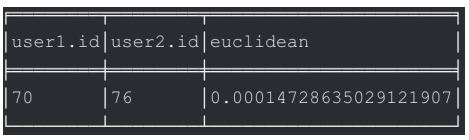
> 💡 The Jaccard similarity and the overlap similarity are **not the same**. While Jaccard similarity gives an idea about the s**hared neighbors by computing a score showing if the users reviews lots of cities in common or not**, the overlap similarity is calculated as **the ratio of the number of cities reviewed by both users to the size of the smaller set**. It tells in a normalized way, **how much their reviewed cities intersect compared to the smaller set of reviewed cities**.

1.6 Euclidean and cosine similarities, using the NB, for the two couples

Euclidian

- For the two French users who reviewed the most :

```
MATCH (user1:User {id: 70})-[r1:review]->(city1:Area_4)
MATCH (user2:User {id: 76})-[r2:review]->(city2:Area_4)

RETURN user1.id, user2.id,gds.similarity.euclidean(collect(r1.NB), collect(r2
AS euclidean;
```



user1.id	user2.id	euclidean
70	76	0.00014728635029121907

- For the two French users who reviewed the most places :

```
MATCH (user1:User {id: 2639})-[r1:review]->(city1:Area_4)
MATCH (user2:User {id:387312})-[r2:review]->(city2:Area_4)

RETURN user1.id, user2.id, gds.similarity.euclidean(collect(r1.NB), collect(r
AS euclidean;
```

user1.id	user2.id	euclidean
2639	387312	0.0003585250792079482

The two French users who reviewed the most places have a slightly bigger euclidian similarity than the two French users who reviewed the most. Both couples have a **very low (near zero) euclidian distance** (`gds.similarity.euclidean` function gives the euclidean distance and not similarity), it means that for both couples, their frequency of rating is very similar.

Cosine

- For the two French users who reviewed the most :

```
MATCH (user1:User {id: 70})-[r1:review]->(city1:Area_4)
MATCH (user2:User {id: 76})-[r2:review]->(city2:Area_4)

RETURN user1.id, user2.id, gds.similarity.cosine(collect(r1.NB), collect(r2.N
AS cosine;
```

user1.id	user2.id	cosine
70	76	0.17031072475815456

- For the two French users who reviewed the most places :

```
MATCH (user1:User {id: 2639})-[r1:review]->(city1:Area_4)
MATCH (user2:User {id:387312})-[r2:review]->(city2:Area_4)

RETURN user1.id, user2.id, gds.similarity.cosine(collect(r1.NB), collect(r2.N
AS cosine;
```

user1.id	user2.id	cosine
2639	387312	0.2823865293325399

The two French users who reviewed the most places have a bigger cosine similarity than the two French users who reviewed the most. Indeed, the two French users who reviewed the most places

have a cosine similarity of **0.28,** which is relatively low. It means that even though the number of places reviewed is similar, the places rated remain different.

> 💡 In both cases, we take into account the **weights** of the nodes. The main difference between euclidean and cosine similarities is what they represent.
>
> - **Euclidean distance** gives the **distance between the two vectors**. Applied to our cases, it means that the **less the distance is big, the more similarities there are in the intensity of rating for each couple**. When the euclidean distance **is very small**, it shows that **for the same places, the number of reviews are very similar**, which is the case for our two couples. However, to get the real euclidean similarity, we need to divide 1 by 1 + euclidean distance.
> - **Cosine similarity** gives an indication of the **direction** of the vector. Applied to our cases, it means that if the cosine similarity is near 1, the **proportion / distribution of given reviews is very similar.** In the cases of our both couples, their cosine similarity is rather low, which means that some of the cities have the same amount of reviews but not for the majority.

1.7 Euclidean and cosine similarities, using the rating, for the two couples

Euclidian

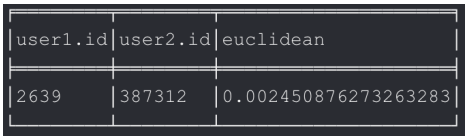
- For the two French users who reviewed the most :

```
MATCH (user1:User {id: 70})-[r1:review]->(city1:Area_4)
MATCH (user2:User {id: 76})-[r2:review]->(city2:Area_4)

RETURN user1.id, user2.id, gds.similarity.euclidean(collect(r1.rating),
collect(r2.rating)) AS euclidean;
```

user1.id	user2.id	euclidean
70	76	0.003046827759226419

- For the two French users who reviewed the most places :

```
MATCH (user1:User {id: 2639})-[r1:review]->(city1:Area_4)
MATCH (user2:User {id:387312})-[r2:review]->(city2:Area_4)

RETURN user1.id, user2.id, gds.similarity.euclidean(collect(r1.rating),
collect(r2.rating)) AS euclidean;
```



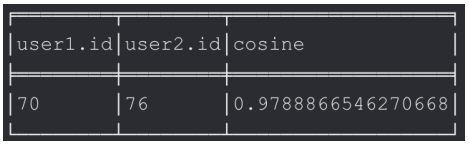
user1.id	user2.id	euclidean
2639	387312	0.002450876273263283

The two French users who reviewed the most have a slightly bigger euclidian distance than the two French users who reviewed the most places. Both couples have a **very low (near zero) euclidian distance :** it means that for both couples, the rating points (from 1 to 5) for each common city are very similar between the two users.

Cosine

- For the two French users who reviewed the most :

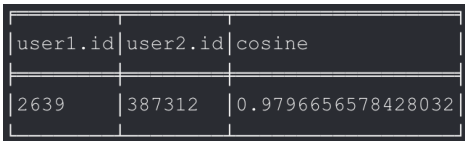
```
MATCH (user1:User {id: 70})-[r1:review]->(city1:Area_4)
MATCH (user2:User {id: 76})-[r2:review]->(city2:Area_4)

RETURN user1.id, user2.id, gds.similarity.cosine(collect(r1.rating),
collect(r2.rating))
AS cosine;
```



user1.id	user2.id	cosine
70	76	0.9788866546270668

- For the two French users who reviewed the most places :

```
MATCH (user1:User {id: 2639})-[r1:review]->(city1:Area_4)
MATCH (user2:User {id:387312})-[r2:review]->(city2:Area_4)

RETURN user1.id, user2.id, gds.similarity.cosine(collect(r1.rating),
collect(r2.rating)) AS cosine;
```



user1.id	user2.id	cosine
2639	387312	0.9796656578428032

The two couples have a **very similar** cosine similarity, and they are both **close to 1**. It means that the two users of each couple have similar schemes of rating, so they ratting patterns are **very aligned**. It could mean that both users in each couple, no matter if they rate highly or strictly, have the **same criterias** for rating.

We saw that for both users of each couple, the euclidian distances between them is so close to 0 and the cosine similarites are close to 1. These information are a good indicator of each couple's preferences and types of rating.

We can interpret that as :

- Both users in each couple **like and dislike the same cities**
- Both users in each couple have the **same rating criterias**
- Both users in each couple have **similar interests and preferences** when it comes to cities they rated in common

Overall, we can make the following statement : **Both users in both couples tend to almost give the same ratings, but they don't give the same amount of reviews for the same cities.**

1.8 Give the average jaccard and overlap similarities for Spanish users who visited at least 5 places per area (NB >= 5)

```
MATCH (user:User {country: 'Spain'})-[r:review]->(city:Area_4)
WHERE r.NB >= 5
WITH user, COLLECT(DISTINCT id(city)) AS cities
WITH COLLECT(cities) AS review_cities

UNWIND range(0, SIZE(review_cities) - 1) AS i
UNWIND range(i + 1, SIZE(review_cities) - 1) AS j
WITH review_cities[i] AS cities1, review_cities[j] AS cities2
WITH gds.similarity.jaccard(cities1, cities2) AS Jaccard_similarity,
     gds.similarity.overlap(cities1, cities2) AS Overlap_similarity

RETURN avg(Jaccard_similarity) AS Average_Jaccard_similarity,
       avg(Overlap_similarity) AS Average_Overlap_similarity;
```

Average_Jaccard_similarity	Average_Overlap_similarity
0.03827563217896215	0.050964844334701456

This query first filters the **users who rated a minimum of 5 times a same city**. It then proceeds to make a list of all users and the cities (uniquely) they rated at least 5 times. With the function `UNWIND` it creates pairs of lists of these reviewed cities and the indexes i and j go through the lists to make unique combinations of cities sets. We then use `gds.similarity.jaccard()` and `gds.similarity.overlap()` functions to get the Jaccard similarity and overlap similarity for each `cities1 cities2` pair. Finally, we get the average **Jaccard similarity and overlap similarity** by averaging the values previously obtained with `avg()`.

💡 Both values indicates that even among users that review multiple places in a city, all the pairs of users usually don't review the same cities.

- An average Jaccard similarity of 0.04 shows that all the selected users share very few common cities when it comes to reviewing.
- An average overlap of 0.05 shows that there is a small amount of overlapping in the set of cities reviewed by the selected users but it is still not really significant.

1.9 Give the one for British, Americans and Italians

```
UNWIND ['Spain', 'United Kingdom', 'United States', 'Italy'] AS country
MATCH (user:User {country: country})-[r:review]->(city:Area_4)
WHERE r.NB >= 5
WITH country, user, COLLECT(DISTINCT id(city)) AS cities
WITH country, COLLECT(cities) AS review_cities

UNWIND range(0, SIZE(review_cities) - 1) AS i
UNWIND range(i + 1, SIZE(review_cities) - 1) AS j
WITH country, review_cities[i] AS cities1, review_cities[j] AS cities2
WITH country,
     gds.similarity.jaccard(cities1, cities2) AS Jaccard_similarity,
     gds.similarity.overlap(cities1, cities2) AS Overlap_similarity

RETURN country,
       avg(Jaccard_similarity) AS Average_Jaccard_similarity,
       avg(Overlap_similarity) AS Average_Overlap_similarity;
```

country	Average_Jaccard_similarity	Average_Overlap_similarity
"Spain"	0.03825683999537706	0.050952723416420966
"United Kingdom"	0.03291728561828005	0.03735612202951054
"United States"	0.03423585090035582	0.04943728793842629
"Italy"	0.034288562529204696	0.04413901365216344

We used the same previous query but we used `UNWIND` to iterate over each country in a list.

💡 There are **very few differences** between the average Jaccard and overlap similarities of all the countries. The biggests ones are the ones from Spain but there are not significantly superiors to the ones in the other countries. We can interpret the stability of theses results by the fact that all the users around the world rate **various cities**, so the **touristic traffic is pretty homogenous**.

2) Link prediction

2.1 We want the number of common neighbors between the two French who reviewed the most (seen before)

```
MATCH (u1:User{id:70}),(u2:User{id:76})
RETURN gds.alpha.linkprediction.commonNeighbors(u1,u2,{relationshipQuery: "re
```

score
27.0

`gds.alpha.linkprediction.commonNeighbors` assigns a score based on the number of common nodes connected to both users, here through the `review` relationship.

> 💡 A score of 27 means that the 2 top French users that reviewed the most have **27 common neighbors.** In other words, they have reviewed 27 of the same cities `Area_4`, meaning that there is some similarity between their choice of places to review.

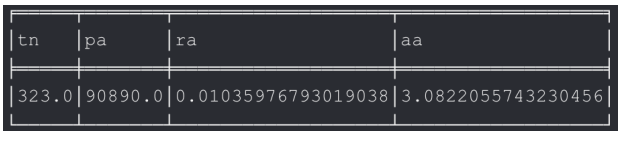
Reminder : Common neighbors : $CN(x,y) = |N(x) \cap N(y)|$

To get each users' number of neighbors :

```
MATCH(u:User)-->(l) WHERE u.id IN [70,76]
RETURN u.id, COUNT(l) AS visits
```

2.2 Give the link prediction on total neighbors, preferential attachment, resource allocations and Adamic Adar

```
MATCH (u1:User{id:70}), (u2:User{id:76})
RETURN gds.alpha.linkprediction.totalNeighbors(u1, u2, {relationshipQuery:"re
gds.alpha.linkprediction.preferentialAttachment(u1, u2, {relationshipQuery:"r
gds.alpha.linkprediction.resourceAllocation(u1, u2, {relationshipQuery:"revie
gds.alpha.linkprediction.adamicAdar(u1, u2, {relationshipQuery:"review"}) as
```



tn	pa	ra	aa
323.0	90890.0	0.01035976793019038	3.0822055743230456

We can see here that the resource allocation is near 0 while the Adamic adar is moderately bigger. These two indexes measure the **closeness of nodes on shared neighbors** and are **complementary**. They both use the degree of common neighbors and, while resource allocation take into account the whole pack of shared neighbors and ponderate their importance in the link prediction, Adamic adam will lessen the weight of neighbors with lots of connections to give more importance to the rarest ones using the logarithmic function.

It is difficult to give a clear interpretation with these information only but we can suppose that in our case, the two French users who reviewed the most have **few global connections** but they still share rare neighbors.

- The resource allocation is **very low** : it means that they either have very few common neighbors, or/and the common neighbors they share have high degrees. Their common neighbors are therefore not very significative in the global graph.
- The Adamic adar is **moderate**. They share some neighbors that are specific and weakly connected.

Overall, considering their total neighbors, their high preference attachment and the metrics we just saw, we can tell that their have **similar intensity of activities** (as they are the two French users who reviewed the most) but they are **not very linked** by the places they review. They could likely meet in **popular places** but they **don't share the same interest in niche communities**.

2.3

Resource Allocation :

$$RA(x,y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{|N(u)|}$$

Adamic adar :

$$A(x,y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{\log |N(u)|}$$

Preferential attachment :

$$PA(x,y) = |N(x)| * |N(y)|$$

Total neighbors :

$$TN(x,y) = |N(x) \cup N(y)|$$

They all indicate in different ways if two nodes are close.

Metric	Significance	Interpretation
Resource allocation	Sum of one over the degree of common neighbors	Penalizes pairs of nodes that have common neighbors that themselves have lots of other neighbors.
Adamic adar	Sum of one over the logarithm of the degree of common neighbors	Mitigate high degrees nodes to give more importance to the rarest neighbors.
Preferential attachment	Product of degrees of two nodes	The more connected a node is, the more likely it is to receive new links
Total neighbors	Sum of neighbors of two nodes (not necessarily shared)	Computes the closeness of nodes, based on the number of unique neighbors that they have

2.4 Let's compute the top 10 shared neighbors between the top 10 spanish reviewers (sum of NB) and get all similarities (total neighbors, preferential attachment, resource allocations and Adamic Adar) ordered by Adamic Adar

```
MATCH (u1:User{country:"Spain"})-[r1:review]->(a1:Area_4)
WITH u1, SUM(r1.NB) as nb_reviews
ORDER BY nb_reviews DESC LIMIT 10
MATCH (u2:User{country:"Spain"})-[r2:review]->(a2:Area_4) WHERE u1.id < u2.id
WITH u1,u2, SUM(r2.NB) as nb_reviews
ORDER BY nb_reviews DESC LIMIT 10
RETURN u1.id, u2.id,
gds.alpha.linkprediction.commonNeighbors(u1,u2,{relationshipQuery: "review"})
gds.alpha.linkprediction.totalNeighbors(u1,u2,{relationshipQuery: "review"})
gds.alpha.linkprediction.preferentialAttachment(u1,u2,{relationshipQuery: "re
gds.alpha.linkprediction.resourceAllocation(u1,u2,{relationshipQuery: "review
gds.alpha.linkprediction.adamicAdar(u1,u2,{relationshipQuery: "review"}) AS a
ORDER BY aa DESC LIMIT 10;
```

u1.id	u2.id	cn	tn	pa	ra	aa
134651	164748	7.0	97.0	3872.0	0.0298284802810642	1.084206466600212
164748	211248	4.0	138.0	7128.0	0.001113179615120749	0.4757948735437447
134651	211248	2.0	112.0	3564.0	0.0006391303070042378	0.23320052921128373
79	164748	1.0	72.0	616.0	0.0013966480446927375	0.15212179093014075
67654	211248	1.0	96.0	1944.0	0.0001208313194780087	0.11085104177163535
19623	211248	0.0	100.0	2349.0	0.0	0.0
1504	211248	0.0	93.0	1539.0	0.0	0.0
79	211248	0.0	83.0	567.0	0.0	0.0
19623	164748	0.0	90.0	2552.0	0.0	0.0
1504	164748	0.0	83.0	1672.0	0.0	0.0

It is more interesting to compare the users 2 by 2 rather than taking the whole array of results.

2.5 Discussing the results

It seems like the more common neighbors there are for a user, the more their Adamic adar is big. To estimate the closeness of two nodes, it is also pertinent to take into account the ratio between the number of common neighbors and the total neighbors.

💡 Interesting insights among the 10 Spanish users who reviewed the most :

- The user 134651 and user 164748 have the **closest nodes**. Indeed, they have 7 common neighbors out of 97 total neighbors, giving a rate of 7,2% of common neighbors, which is the **best common neighbor rate of the list**. These two users also have the highest resource allocation and Adamic adar, reinforcing the thought that this pair of users have the **highest** probability to be **linked**.
- The pair of users with an Adamic adar of 0 have 0 common neighbors. They are not linked at all, even though some of the pairs have a high preferential attachment. It means that even if they have a similar activity intensity, **they don't review the same places at all**.

III - Mining Mono-partite graphs

1) Cypher Projection

1.1 Create a Cypher Projection named "French2019" where you extract the graph for the French in 2019 population with NB

```
CALL gds.graph.project.cypher(
    "French2019",
    "MATCH (a1:Area_4) RETURN id(a1) AS id",
    "MATCH (n:Area_4)-[r:trip{country:'France', year:2019}] -> (m)
    RETURN id(n) as source, id(m) as target, SUM(r.NB) as NB")
    YIELD graphName AS graph, nodeCount AS nodes, relationshipCount as rels;
```

graph	nodes	rels
"French2019"	36612	237543

Two filters are used for the relationship : **country and year**. They allow us to select particular relationships for the sub-graph. We can see that there are 36612 nodes and 237543 relationships that correspond to French trips and year 2019 simultaneously.

To delete graphs : CALL gds.graph.drop("graph_name")

(to make an undirected graph)

```
CALL gds.graph.project(
    "French2019",
    "Area_4",
    {
        relationshipType: {
            type: "trip",
            orientation : "UNDIRECTED"
        }
    })
YIELD graphName, nodeCount, relationshipCount;
```

graphName	nodeCount	relationshipCount
"French2019"	36612	1145404

The number of relationships is different as the graph is undirected, so the relations are symmetrical.

1.2 Idem with “French2020”, “British2019”, “British2020”, “US2019”, “US2020”

French2020 :

```
CALL gds.graph.project.cypher(
    "French2020",
    "MATCH (a1:Area_4) RETURN id(a1) AS id",
    "MATCH (n:Area_4)-[r:trip{country:'France', year:2020}] -> (m)
    RETURN id(n) as source, id(m) as target, SUM(r.NB) as NB")
    YIELD graphName AS graph, nodeCount AS nodes, relationshipCount as rels;
```

graph	nodes	rels
"French2020"	36612	243970

British2019 :

```
CALL gds.graph.project.cypher(
    "British2019",
    "MATCH (a1:Area_4) RETURN id(a1) AS id",
    "MATCH (n:Area_4)-[r:trip{country:'United Kingdom', year:2019}] -> (m)
    RETURN id(n) as source, id(m) as target, SUM(r.NB) as NB")
    YIELD graphName AS graph, nodeCount AS nodes, relationshipCount as rels;
```

graph	nodes	rels
"British2019"	36612	17740

British2020 :

```
CALL gds.graph.project.cypher(
    "British2020",
    "MATCH (a1:Area_4) RETURN id(a1) AS id",
    "MATCH (n:Area_4)-[r:trip{country:'United Kingdom', year:2020}] -> (m)
    RETURN id(n) as source, id(m) as target, SUM(r.NB) as NB")
    YIELD graphName AS graph, nodeCount AS nodes, relationshipCount as rels;
```

graph	nodes	rels
"British2020"	36612	5628

US2019 :

```
CALL gds.graph.project.cypher(
    "US2019",
    "MATCH (a1:Area_4) RETURN id(a1) AS id",
    "MATCH (n:Area_4)-[r:trip{country:'United States', year:2019}] -> (m)
    RETURN id(n) as source, id(m) as target, SUM(r.NB) as NB")
    YIELD graphName AS graph, nodeCount AS nodes, relationshipCount as rels;
```

graph	nodes	rels
"US2019"	36612	11340

US2020 :

```
CALL gds.graph.project.cypher(
    "US2020",
    "MATCH (a1:Area_4) RETURN id(a1) AS id",
    "MATCH (n:Area_4)-[r:trip{country:'United States', year:2020}] -> (m)
    RETURN id(n) as source, id(m) as target, SUM(r.NB) as NB")
    YIELD graphName AS graph, nodeCount AS nodes, relationshipCount as rels;
```

graph	nodes	rels
"US2020"	36612	2211

We can see interesting insights when we compare the number of relationships through the time (2019 to 2020) and based on the country :

- for French users, there is a slight increasing of 6427 trips from 2019 to 2020
- for British users, there is a significant decreasing of 12112 trips from 2019 to 2020
- for American users, there is a very significant decreasing of 9129 trips from 2019 to 2020, representing more than 80% of decreasing (versus 68% for British users)

We can link these results as the consequences of **COVID-19** in 2020 and its resulting trip restrictions, encouraging French people to stay in their country to travel, allowing an increase of trips in their local country. These same restrictions explain the significant decreasing of trips from British people, who could still go to France by train but were **strictly limited in crossing borders**. Finally, the spectacular decreasing of American travelers in France is easily explained by the **important difficulty to cross borders outside of their continent and by plane due to the pandemic crisis.**

2) Community detection

2.1 Give the number of triangles per node for French2019 and French2020, in decreasing order

```
CALL gds.triangleCount.stream("French2019")
YIELD nodeId, triangleCount
RETURN gds.util.asNode(nodeId).name_4, triangleCount
ORDER BY triangleCount DESC;
```

gds.util.asNode(nodeId).name_4	triangleCount
"Bordeaux"	237960
"Paris, 9e arrondissement"	219132
"Toulouse"	211443
"Saint-Malo"	209340
"La Rochelle"	203697
"Nice"	202329
"Lille"	201100
"Strasbourg"	197332
"Agde"	192933
"Lyon, 2e arrondissement"	192089

This query gives the **number of triangles** (a triangle is a relationship where three nodes are interconnected) of each node with `gds.triangleCount.stream()` function. The triangles are important because they indicate **direct paths** (as squares are alternative paths) between locations.

	Rang parmi les villes européennes	Nombre de nuitées					Part des nuitées de non-résidents (en %)
		Résidents	Non-résidents	Ensemble	Moyenne par jour	Moyenne par jour pour 1 000 habitants	
		(en milliers)					
Paris	1	3 662	11 388	15 051	41	4	76
Nice	9	1 471	3 880	5 351	15	22	73
Marseille	27	1 243	805	2 048	6	6	39
Lyon	31	1 126	791	1 916	5	5	41
Bordeaux	33	945	805	1 750	5	7	46
Toulouse	43	689	453	1 141	3	5	40
Strasbourg	46	483	625	1 109	3	8	56
Montpellier	50	664	348	1 012	3	9	34
Bayonne	53	661	302	963	3	23	31
Lille	70	500	240	741	2	2	32
Nantes	71	530	203	733	2	4	28
La Rochelle	77	526	118	645	2	21	18
Toulon	81	434	151	586	2	5	26
Annecy	82	357	223	580	2	13	38
Aix-en-Provence	87	267	285	552	2	11	52
Fréjus	92	297	207	504	1	16	41

Number of nights booked via Internet in cities in France during 2019, source : Eurostat

The French cities that have the biggest number of triangles are mostly in the top French cities that have the most booked nights via the internet in 2019. The triangles are therefore **an indicator of the popularity of a city**, in which the connexions are dense. We can also suppose that the number of triangles of a city can also depend on the geographical situation, as we can see that cities on the seaside have a big number of triangles (Agde, Saint-Malo ...).

2.2 Idem but grouped by department (Area_2)

```
CALL gds.triangleCount.stream("French2019")
YIELD nodeId, triangleCount
```

```
RETURN gds.util.asNode(nodeId).name_2, triangleCount
ORDER BY triangleCount DESC;
```

gds.util.asNode(nodeId).name_2	triangleCount
"Gironde"	237960
"Paris"	219132
"Haute-Garonne"	211443
"Ille-et-Vilaine"	209340
"Charente-Maritime"	203697
"Alpes-Maritimes"	202329
"Nord"	201100
"Bas-Rhin"	197332
"Hérault"	192933
"Rhône"	192089

The results are coherent with the results we got on the previous query (for cities). Indeed, the department nodes with the most triangles include the cities with the most triangles.

2.3 Idem with the clustering coefficient. Discuss the result (Infinity and different results)

```
CALL gds.localClusteringCoefficient.stream("French2019")
YIELD nodeId, localClusteringCoefficient
RETURN gds.util.asNode(nodeId).name_2,
SUM(localClusteringCoefficient) as localClusteringCoefficient ORDER BY localC
```

gds.util.asNode(nodeId).name_2	localClusteringCoefficient
"Nord"	36.44136620714438
"Pas-de-Calais"	31.617493090283745
"Isère"	25.895517309868783
"Seine-Maritime"	24.42086706843141
"Bouches-du-Rhône"	24.157201268581815
"Gironde"	23.9702672436514
"Côtes-d'Armor"	23.23477113161134
"Saône-et-Loire"	22.65767350430486

This query gives the **local clustering coefficient of a node** that is an indicator of the probability t**hat the node forms a triangle with its neighbors** with `gds.localClusteringCoefficient.stream()` function.

💡 We can see here that in the Nord department (and globally in the north of France) , people tend to go to a city (probably Lille as it's the biggest one in the department) and visit the other cities around, resulting **in many connected cities**.

An interesting remark is the fact that **Gironde is lower in the ranking**, while Bordeaux was the city (excluding Paris or Lyon as a whole) with the most triangles and part of the Gironde department. We can suppose that **the city acts more as a single destination point rather than an intermediary for tourist circuits**.

2.4 / 2.5 Extract communities with "Label Propagation" and give the list of communities per department

```
CALL gds.labelPropagation.stream("French2019")
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name_2, collect(distinct communityId);
```

	gds.util.asNode(nodeId).name_2	collect(distinct communityId)
32	"Seine-Maritime"	[251]
33	"Charente-Maritime"	[251]
34	"Aveyron"	[251]
35	"Vendée"	[251]
36	"Yvelines"	[251]
37	"Haute-Saône"	[251, 1788]

Results from the practical work course professor

This query gives the list of communities for each department.

💡 It is curious to see that there is, for each department, an unique big cluster, which means that our graph is so densely connected that **every department belongs to a big cluster**. Few departments have local clusters, such as Haute-Saône but it seems to be a minority.

2.6 Idem with "Louvain"

```
CALL gds.louvain.stream("French2019")
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name_2, collect(distinct communityId);
```

gds.util.asNode(nodeId).name_2	collect(distinct communityId)
"Haute-Loire"	[3614, 251]
"Manche"	[1520, 107]
"Orne"	[1520, 251]
"Pas-de-Calais"	[2221]
"Moselle"	[2261]
"Hérault"	[251]

Results from the practical work course professor

Using Louvain `gds.louvain.stream()` seems **more effective** than Label propagation for our graph. Indeed, instead of obtaining a single big community, we get results that are pretty coherent : we can see that we get d**ifferent communities depending on the department** but some departments have communities in common. For example, Orne and Manche have the **same community,** which seems logical as they are **neighbor departments** and belong to the **same region** (Normandie).

> 💡 Louvain is more adapted to densely connected graphs than Label propagation.

2.7 Group previous result per communityId. Discuss the result

```
CALL gds.louvain.stream("French2019")
YIELD nodeId, communityId
RETURN communityId, collect (distinct gds.util.asNode(nodeId).name_2)
```

	communityId	collect(distinct gds.util.asNode(nodeId).name_2)
1	3614	["Haute-Loire", "Alpes-Maritimes", "Alpes-de-Haute-Provence", "Bouches-du-Rhône", "Hérault", "Vaucluse", "Aveyron", "Lozère", "Ardèche", "Var", "Loire", "Gard", "
2	1520	["Manche", "Orne", "Finistère", "Côtes-d'Armor", "Marne", "Eure", "Sarthe", "Eure-et-Loir", "Loire-Atlantique", "Ille-et-Vilaine", "Seine-Maritime", "Vendée", "Calvados
3	2458	["Pas-de-Calais", "Somme", "Marne", "Nord", "Ardennes", "Aisne", "Oise", "Haute-Marne", "Seine-Maritime", "Aube", "Tarn", "Côte-d'Or", "Allier", "Meuse"]
4	359	["Moselle", "Jura", "Vosges", "Doubs", "Meurthe-et-Moselle", "Côte-d'Or", "Haute-Saône", "Haute-Marne", "Meuse", "Saône-et-Loire", "Bas-Rhin", "Aube", "Nièvre",
5	251	["Hérault", "Creuse", "Gers", "Haute-Vienne", "Deux-Sèvres", "Haute-Garonne", "Corrèze", "Ariège", "Pyrénées-Atlantiques", "Charente", "Tarn", "Pyrénées-Orienta
6	1004	["Corse-du-Sud", "Haute-Corse"]

Results from the practical work course professor

This query is very similar to the previous one : we aggregate the departments by cluster (communityId) instead of aggregating the communities by department.

💡 We can see that most of the clusters groups departments that are from the **same regions or neighbor departments**. The biggest clusters are very **expanded** through the country, by grouping many diverse departments that could be geographically far away from each other, while there are **very local clusters**, such as the communityId 1004 for example, that includes only Corse du Sud and Corse du Nord (Corse region).

The clusters are a good indicators of the way people tend to travel in France depending on the department they visit : following which part of France they choose to visit, they tend to move across many cities/departments or on the other hand, only stay in a zone (for example Corse, which is easily explainable as it is an island).

3) Path finding

3.1 Give all pairs of shortest paths in French2019 based on NB properties. Need to use a Map configuration instead on CypherProjection (with "relationshipWeightProperty")

```
CALL gds.alpha.allShortestPaths.stream("French2019"), {relationshipWeightProp
YIELD sourceNodeId, targetNodeId, distance
RETURN gds.util.asNode(sourceNodeId).name_3, SUM(distance) as sum_dist
ORDER BY sum_dist DESC limit 10;
```

gds.util.asNode(sourceNodeId).name_3	sum_dist
"Grenoble"	311787.0
"Lyon"	284425.0
"Rodez"	222377.0
"Albi"	218027.0
"Lille"	216202.0
"Pau"	215402.0
"Castres"	203234.0
"Troyes"	199254.0
"Rouen"	197384.0
"Caen"	197130.0

3.2 Extract the Minimum Spanning Tree starting from "Paris 1° arrondissement"

```
CALL gds.spanningTree.minimum.write("French2019",
{
    startNodeId:873,
```

```
    relationshipWeightProperty: "NB",
    writeProperty: "MinFrench2019",
    weightWriteProperty: "writeCost"
}) YIELD effectiveNodeCount
```

```
MATCH p-(Area_4) - [r:MinFrench2019]
```

4) Centrality

4.1 Extract PageRank centralities from nodes in different various cypher projection

```
CALL gds.pageRank.stream("French2019", {
    relationshipWeightProperty : "NB",
    maxIterations : 20,
    dampingFactor : 0.85
    })
YIELD nodeId, score
RETURN
  gds.util.asNode(nodeId).name_4 AS name_4,
  score
ORDER BY score DESC, name_4 ASC
```

4.2 Give the average, min and max PageRank score for corresponding departments. Explain the differences

```
CALL gds.pageRank.stream("French2019", {
    relationshipWeightProperty : "NB",
    maxIterations : 20,
    dampingFactor : 0.85
})
YIELD nodeId, score
RETURN
    avg(score) AS averageScore,
    min(score) AS minScore,
    max(score) AS maxScore
```

Bordeaux seems to be the most popular.

4.3 Let's compute degree and closeness centralities for those graphs and explain differences

```
CALL gds.degree.stream("French2019")
YIELD nodeId, score
```

```
RETURN
     nodeID,
     gds.util.asNode(nodeId).name_4 AS name_4,
     score AS degreeCentrality
ORDER BY degreeCentrality DESC, name_4 ASC
```

- Degree centrality measures the number of direct connections a city has, which corresponds to the number of trips that start or end in that city. A high degree centrality indicates that the city is a major hub with many direct connections to other cities. It can identify cities that are popular travel destinations

```
CALL gds.alpha.closeness.harmonic.stream("French2019")
YIELD nodeId, centrality
RETURN
    nodeId,
    gds.util.asnode(nodeId).name_4 AS name_4,
    centrality AS harmonicClosenessCentrality
```

- Closeness centrality reflects how quickly a city can connect to all other cities in the network. It is inversely proportional to the sum of the shortest path distances from a city to all others. This is useful for identifying cities that serve as strategic locations for logistics, regional connectivity, or travel efficiency.