DSCI553 Foundations and Applications of Data Mining

Fall 2020

Assignment 4

Deadline: Oct. 29th 11:59 PM PST

1. Overview of the Assignment

In this assignment, you will explore the spark GraphFrames library as well as implement your own **Girvan-Newman** algorithm using the Spark Framework to detect communities in graphs. You will use the ub_sample_data.csv dataset to find users who have a similar business taste, and sample_user_state.csv to find the users who have a similar travel history. The goal of this assignment is to help you understand how to use the Girvan-Newman algorithm to detect communities in an efficient way within a distributed environment.

2. Requirements

2.1 Programming Requirements

a. **You must use Python and Spark to implement all tasks.** There will be a 10% **bonus** for each task if you also submit a Scala implementation and both your Python and Scala implementations are correct.

b. **You can use the Spark DataFrame and GraphFrames library for task1, but for task2 you can ONLY use Spark RDD and standard Python or Scala libraries. (ps. For Scala, you can try GraphX, but for the assignment, you need to use GraphFrames.)**

2.2 Programming Environment

Python 3.6, Scala 2.11 and Spark 2.3.2

We will use Vocareum to automatically run and grade your submission. You must test your scripts on **the local machine** and **the Vocareum terminal** before submission.

2.3 Write your own code

Do not share code with other students!!

For this assignment to be an effective learning experience, you must write your own code! We emphasize this point because you will be able to find Python implementations of some of the required functions on the web. Please do not look for or at any such code!

TAs will combine all the code we can find from the web (e.g., Github) as well as other students' code from this and other (previous) sections for plagiarism detection. We will report all detected plagiarism.

2.4 What you need to turn in

You need to submit the following files on Vocareum: (all lowercase)

a. [REQUIRED] two Python scripts, named: **task1.py, task2.py**

b1. [REQUIRED FOR SCALA] two Scala scripts, named: **task1.scala, task2.scala**

b2. [REQUIRED FOR SCALA] one jar package, named: **hw4.jar**

c. [OPTIONAL] You can include other scripts called by your main program

d. You don't need to include your results. We will grade on your code with our testing data (data will be in the same format).

3. Datasets

You will continue to use Yelp dataset. We have generated a sub-dataset, ub_sample_data.csv, from the Yelp review dataset containing user_id and business_id. Also, we will use user state data that we use in the HW2. **Please note that you will use the two different for each task**. You can download them from Vocareum.

Task Name	Dataset
Task1	ub_sample_data.csv
Task2	Travel state match graph:user_state_synthetic.csv Similarity graph: sample_user_state.csv

4. Tasks

4.1 Task1: Graph Construction

To construct the social network graph, each node represents a user and there will be an edge between two nodes if the number of times that two users review the same business is **greater than or equivalent to** the filter threshold. For example, suppose user1 reviewed [business1, business2, business3] and user2 reviewed [business2, business3, business4, business5]. If the threshold is 2, there will be an edge between user1 and user2.

If the user node has no edge, we will not include that node in the graph.

In Task 1, we use filter threshold 7.

Use user_id directly as strings when constructing the graph, don't hash them to integers.

4.2 Task1: Community Detection Based on GraphFrames (3 pts)

In task1, you will explore the Spark GraphFrames library to detect communities in the network graph you constructed in 4.1. In the library, it provides the implementation of the Label Propagation Algorithm

(LPA) which was proposed by Raghavan, Albert, and Kumara in 2007. It is an iterative community detection solution whereby information "flows" through the graph based on underlying edge structure. For the details of the algorithm, you can refer to the paper posted on the Piazza. In this task, you do not need to implement the algorithm from scratch, you can call the method provided by the library. The following websites may help you get started with the Spark GraphFrames:

https://docs.databricks.com/spark/latest/graph-analysis/graphframes/user-guide-python.html

https://docs.databricks.com/spark/latest/graph-analysis/graphframes/user-guide-scala.html

4.2.1 Execution Detail

The version of the GraphFrames should be **0.6.0**.

For Python:

- In PyCharm, you need to add the sentence below into your code

  pip install graphframes

  os.environ["PYSPARK_SUBMIT_ARGS"] = (

    "--packages graphframes:graphframes:0.6.0-spark2.3-s_2.11")

- In the terminal, you need to assign the parameter "packages" of the spark-submit:

  --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11

For Scala:

- In Intellij IDEA, you need to add library dependencies to your project

  "graphframes" % "graphframes" % "0.6.0-spark2.3-s_2.11"

  "org.apache.spark" %% "spark-graphx" % sparkVersion

- In the terminal, you need to assign the parameter "packages" of the spark-submit:

  --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11

For the parameter "maxIter" of LPA method, you should set it to 5.

4.2.2 Output Result

In this task, you need to save your result of communities in a **txt** file. Each line represents one community and the format is:

'user_id1', 'user_id2', 'user_id3', 'user_id4', ...

Your result should be firstly sorted by the size of communities in the ascending order and then the first user_id in the community in **lexicographical** order (the user_id is a type of string). The user_ids in each community should also be in the **lexicographical** order.

If there is only one node in the community, we still regard it as a valid community.

```
'111', '681'
'1231', '142'
'2281', '283'
'2517', '2744'
'2862', '2985'
'359', '468'
'659', '661'
'102', '125', '54'
'166', '245', '58'
'119', '1615', '2543', '8'
'2', '216', '35', '6'
'1530', '1992', '2116', '497', '935'
'120', '183', '209', '60', '728', '74'
'1245', '1794', '1866', '2113', '2150', '2188', '2606', '2876', '2953', '2955', '640'
'1072', '1270', '1565', '1620', '1761', '1861', '2479', '2575', '2976', '30', '3280', '475', '713', '752'
'1136', '1197', '1206', '1355', '1408', '1418', '1498', '1508', '1648', '1723', '1913', '1918', '2005', '2097', '2332', '23
```

Figure 1: Sample community output file format

4.3 Task2: Community Detection Based on Girvan-Newman algorithm

In HW2, we studied the frequent pattern of the travel of the Yelp users. In this homework, we group users based on their travel history. For this aim, we are going to leverage the power of network science to cluster the users.

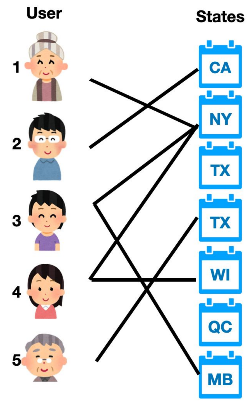
We first construct the two different graphs based on the users' travel history described in 4.3.1 Task2: Graph Construction. **Note that you will construct the different graphs from the different rules.** Then you will implement your own Girvan-Newman algorithm to detect the communities in the network graph. You can refer to the Chapter 10 of the textbook for the algorithm details. **For task2, you can ONLY use Spark RDD and standard Python or Scala libraries**. Remember to delete your code that imports graphframes.

4.3.1 Task2: Graph Construction

How could we represent the relationship between the yelp users in their travel history? In this task, you will try two graph representations; **Case1:** Travel state match graph (bipartite graph) and **Case2:** Similarity graph. You will use user_state_synthetic.csv for Travel state match graph, and sample_user_state.csv for Similarity graph.

- **Case1: Travel state match graph**

Travel state match graph is the bipartite graph where the nodes on the right side are the user nodes and the nodes on the other side are the states. Then if a user i travel state s, we will put the edge between i and s. Because this is a bipartite graph, there is no edge between users or between states. You can refer to the schematic example of Travel state match graph in Figure2.



User	State
1	NY
2	CA
3	NY
3	MB
4	NY
4	WI
5	TX

Figure2: Schematic of Travel state match graph

- **Case2: Similarity graph**

On the contrary, a similarity graph is the graph where the users who share similar travel patterns. The nodes on the similarity graph are the users. And there will be an edge between two users if their **Jaccard Similarity is equal or greater than 0.5 (>= 0.5)**. For example, if UserA traveled [state1, state2, state3] and UserB traveled [state2, state3, state4, state5], the Jaccard Similarity is $2/5 = 0.4$. Therefore, there will be an edge between UserA and UserB.

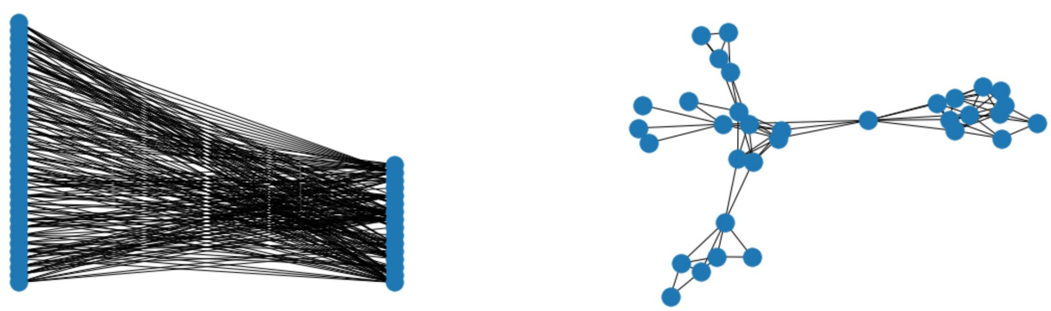


Figure 3

Case1: travel state match graph

Bipartite graph; the left side nodes are users and the right side nodes are states.

Case2: similarity graph

Nodes are users and users are connected if their travel history similarity calculated by Jaccard similarity is greater than 0.5.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Jaccard Similarity (from the Week6 lecture slide)

4.3.1 Betweenness Calculation (2 pts)

Similarity graph betweenness is only graded.

In this part, you will calculate the betweenness of each edge in the graph we constructed in 4.3.1. Then you need to save your result in a **txt** file. The format of each line is

('user_id1', 'user_id2'), betweenness value

Your result should be firstly sorted by the betweenness values in the descending order and then the first user_id in the tuple in **lexicographical** order (the user_id is a type of string). The two user_ids in each tuple should also be in **lexicographical** order. You do not need to round your result.

```
('12', '74'), 243.36111111111106
('24', '74'), 237.93253968253967
('3', '36'), 215.63333333333333
('12', '50'), 199.18067210567193
('2113', '640'), 189.00000000000003
('2188', '640'), 189.00000000000003
('2606', '640'), 189.00000000000003
```

Figure 4: Sample betweenness output file format

4.3.2 Community Detection (4.6 pts = 2.3 pts (Case1) + 2.3 pts (Case2))

Please check *7. Additional grading criteria for Task2-Case1* below

You are required to divide the graph into suitable communities, which reaches the global highest modularity. We will use the two slightly different modularities for Travel state match graph (Case1) and Similarity graph (Case2). The formula of modularity is shown below:

Modularity of partitioning S of graph G:

- $Q = \sum_{s \in S}$ [(# edges within group s) – (expected # edges within group s)]
- $Q(G,S) = \underbrace{\frac{1}{2m}}_{\text{Normalizing cost.: -1<Q<1}} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} \left(A_{ij} - \frac{k_i k_j}{m}\right)$

A_{ij} = 1 if i connects j, 0 else

Case1: The modularity for Travel state match graph [1]

Modularity of partitioning S of graph G:

- $Q = \sum_{s \in S}$ [(# edges within group s) – (expected # edges within group s)]
- $Q(G,S) = \underbrace{\frac{1}{2m}}_{\text{Normalizing cost.: -1<Q<1}} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} \left(A_{ij} - \frac{k_i k_j}{2m}\right)$

A_{ij} = 1 if i connects j, 0 else

Case2: The modularity for Similarity graph [2]

According to the Girvan-Newman algorithm, after removing one edge, you should re-compute the betweenness. The “m” in the formula represents the edge number of the **original graph**. The “A” in the formula is the adjacent matrix of the **original graph**. (Hint: In each remove step, “m” and “A” should not be changed).

If the community only has one user node, we still regard it as a valid community.

You need to save your result in a **txt** file. The format is the same with the output file from task1. Note that the output file of Case1 contains the state since the states are also nodes in the Travel state match graph.

4.4 Execution Format

Execution example:

Python:

spark-submit --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11 task1.py <filter threshold> <input_file_path> <community_output_file_path>

spark-submit task2.py <case number> <input_file_path> <betweenness_output_file_path> <community_output_file_path>

Scala:

spark-submit --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11 –-class task1 hw4.jar <filter threshold> <input_file_path> <community_output_file_path>

spark-submit –-class task2 hw4.jar <case number> <input_file_path> <betweenness_output_file_path> <community_output_file_path>

Input parameters:

1. <filter threshold>: the filter threshold to generate edges between user nodes (Task1).

2. <case number>: the case number for Task2.

3. <input file path>: the path to the input file including path, file name and extension.

4. <betweenness output file path>: the path to the betweenness output file including path, file name and extension.

5. <community output file path>: the path to the community output file including path, file name and extension.

Execution time:

The overall runtime limit of your task1 (from reading the input file to finishing writing the community output file) is **200** seconds.

The overall runtime limit of your task2 (from reading the input file to finishing writing the community output file) is **600** seconds.

If your runtime exceeds the above limit, there will be no point for this task.

5. About Vocareum

a. You can use the provided datasets under the directory resource: /asnlib/publicdata/

b. You should upload the required files under your workspace: work/

c. You must test your scripts on both the local machine and the Vocareum terminal before submission.

d. During the submission period, the Vocareum will automatically test task1.py and task2.py

e. During the grading period, the Vocareum may use another dataset that has the same format for testing.

f. We do not test the Scala implementation during the submission period.

g. Vocareum will automatically run both Python and Scala implementations during the grading period.

h. Please start your assignment early! You can resubmit any script on Vocareum. We will only grade on your last submission.

6. Grading Criteria

(% penalty = % penalty of possible points you get)

1. You can use your free 5-day extension separately or together. You must submit a late-day request via https://forms.gle/vWZPAujxsHc2w3QB7. This form is recording the number of late days you use for each assignment. By default, we will not count the late days if no request submitted.
2. There will be 10% bonus for each task (i.e., 0.3pts, 0.2pts, 0.23pts, and 0.23pts) if your Scala implementations are correct. Only when your Python results are correct, the bonus of using Scala will be calculated. There is no partial point for Scala.
3. There will be no point if your submission cannot be executed on Vocareum.
4. There is no regrading. Once the grade is posted on the Blackboard, we will only regrade your assignments if there is a grading error. No exceptions.
5. There will be 20% penalty for the late submission within one week and no point after that.
6. If you use your late days, there wouldn't be the 20% penalty but the final deadline wouldn't be extended.

7. Additional grading criteria for Task2-Case1 (added by Akira Matsui, Oct 25)

I found some implementation returns with different results for sample_user_state.csv, even if it follows the description (it needs some detailed description). Given this situation, I made a synthetic dataset (user_state_synthetic.csv) for the Travel state match graph, now your code could return the expected results.

I also investigated some submitted codes. Here are the common mistakes that I found so far. Your output for Task2-Case1 should contain states (e.g., "CA"), not just user_id because both are nodes in the

Travel-state match graph. You need to apply the same functions of the Girvan-Newman algorithm except for the modularity calculation. For example, your adjacency matrix have to be symmetric.

Also, I will give the credits for the code that follows the description of the Task2-Case1. That is, I will give full points (2.3 points) for Task2-Case1 if as long as your Girvan-Newman algorithm returns the right results for your Task2-Case2, and it clusters the Travel-state match graph, regardless of its output.

More precisely, to get full points for Task2-Case1, your implementation will need to satisfy the following criteria:

- Criterion 1: You got the full points (2.3 pts) for Task2-Case2 (Similarity graph).
- Criterion 2: Your code constructs the bipartite graph (Travel-state match).
- Criterion 3: The modularity calculation for the Travel state match graph is correct (its denominator is one, while the denominator for the Similarity graph is two, see Figs in 4.3.2).

You will get fill points (2.3 pts) for Task2-Case1 as long as your implementation satisfies those three criteria, even if you get 0 pts in the submission report. I will manually check Criterion 2, 3 during grading. Since we do not require any strict format of the graph, you can use any data type (e.g., adjacency matrix or dictionary of the nodes) as long as your code will cluster the Travel-state match graph.

If you want to check if your code satisfies Criterion 2 and 3, please visit my Office-hour (Thursday morning). I will also hold an additional office-hour this coming Tuesday, 11:00-12:00 (Oct 27). If both time slots do not work for you, please contact me via email.

Reference

[1] Barber, Michael J. "Modularity and community detection in bipartite networks." *Physical Review E* 76.6 (2007): 066102.

[2] Newman, Mark EJ, and Michelle Girvan. "Finding and evaluating community structure in networks." *Physical review E* 69.2 (2004): 026113.