Practice of Social Media Analytics

CS5128701

工管產碩數據一

M11101T04

Hsiang-Jen Li 李享紝

Table of Contents

graph_algo

Practice of Social Media Analytics CS5128701

This is a code written for the ***Practice of Social Media Analytics*** course.

The code was written by @李享紜 - Hsiang-Jen Li, but there is no guarantee that all algorithms are error-free. Therefore, users need to assume their own risk when using these code.

Project structure

```
.
├── ./README.md
├── ./by_cn.ipynb # 測試
├── ./by_deep_cn.ipynb # 測試
├── ./hw02.ipynb # 最終版
├── ./networkX.ipynb # networkX 直接使用套件
├── ./data
│   ├── ./data/sampleSubmission.csv
│   ├── ./data/test.csv
│   └── ./data/train.csv
├── ./docs # 文件檔
├── ./gen.sh # 建立文件檔的 shell script
└── ./graph_algo
    ├── ./graph_algo # 本次撰寫的演算法
    │   ├── ./graph_algo/graph_algo/base.py
    │   ├── ./graph_algo/graph_algo/graph.py
    │   ├── ./graph_algo/graph_algo/pipeline.py
    │   ├── ./graph_algo/graph_algo/score_func.py
    │   └── ./graph_algo/graph_algo/sparsification.py
```

Create documentation from source code

```
IMAGE=https://hsiangjenli.github.io/hsiangjenli/static/image/ntust.png
pdoc graph_algo/graph_algo -o ./docs --favicon "$IMAGE" --logo "$IMAGE" --docformat "numpy"
```

Report

What is new?

1. deep_common_neighbors
2. deep_neighbors
3. deep_jaccard_coefficient

10 Experiment

E1-networkX networkX.ipynb

Concept

- I tested the `greedy_modularity_communities` function in networkX to address my concerns about the effectiveness of my own algorithm implementation.
- The actual results were unsatisfactory, leading me to speculate that using this method for community detection may result in overly fine-grained partitions.

E2 ~ E4 hw02.ipynb

1. E2-deep_common_neighbor-level-1

2. E3-deep_common_neighbor-level-2
3. E4-deep_common_neighbor-level-3

   Concept If two nodes recursively search for common neighbors within (1, 2, 3) levels, it implies that they are likely to be in the same community, as they will be connected within six steps.

E5 ~ E7 hw02.ipynb

1. E5-deep_jaccard_coefficient-level-1-threshold-mean
2. E6-deep_jaccard_coefficient-level-2-threshold-mean
3. E7-deep_jaccard_coefficient-level-3-threshold-mean

   Concept

   - Search for common neighbors between two nodes and **consider the common neighbors and the size of their union neighbors**, the Jaccard coefficient (JC) formula denominator, which is the union neighbor size, may cause the JC to decrease if both nodes have a large number of friends
   - Additionally, I made a mistake in **only calculating the average JC for the test data**, rather than considering the **average JC for all node combinations**. However, due to computational limitations, I was unable to perform the latter calculation :)

E8 ~ E10 hw02.ipynb

1. E8-deep_jaccard_coefficient-level-1-threshold-mean-minus-var
2. E9-deep_jaccard_coefficient-level-2-threshold-mean-minus-var
3. E10-deep_jaccard_coefficient-level-3-threshold-mean-minus-var

   Concept

   - Same as above, the threshold is the average JC minus variance
   - Their still have mistake in my implementation.

Summary and Conclusion

Experiment	Accuracy
E1-networkX	0.49
E2-deep_common_neighbor-level-1	0.7
E3-deep_common_neighbor-level-2	0.79333
E4-deep_common_neighbor-level-3	0.78
E5-deep_jaccard_coefficient-level-1-threshold-mean	0.67166
E6-deep_jaccard_coefficient-level-2-threshold-mean	0.69166
E7-deep_jaccard_coefficient-level-3-threshold-mean	0.705
E8-deep_jaccard_coefficient-level-1-threshold-mean-minus-var	0.68333
E9-deep_jaccard_coefficient-level-2-threshold-mean-minus-var	0.71666
E10-deep_jaccard_coefficient-level-3-threshold-mean-minus-var	0.75666

I initially intended to use an algorithm similar to "greedy_modularity_communities" in networkX for my homework.

However, after actually doing it, I found that the results were not satisfactory and the compute time was too long (about 12 hours).

Later on, I realized that this assignment only focuses on node1 and node2. So, I changed my approach and considered only node1 and node2, which greatly improved the final result.

- View Source

graph_algo.graph

- View Source

class Graph(graph_algo.base.BaseGraph):

- View Source

Examples

```
from core import Graph
graph = Graph()

graph.add_edge(1, 2)
graph.add_edge(1, 3)

print(graph.get_nodes)
>>> [1, 2, 3]

print(graph.get_average_degree)
>>> 2.0

print(graph.get_neighbors(1))
>>> [2, 3]

print(graph.get_neighbors_size(1))
>>> 2

print(graph.edges)
>>> {1: [2, 3]}
```

def common_neighbors(*args, **kwargs):

- View Source

Common Neighbors

Calculate the common neighbors score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **int**: The common neighbors score between two nodes.

Examples

```
from core import Graph

graph = Graph()
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(2, 3)

# print(graph.common_neighbors(1, 2))
# >>> 1
```

def deep_neighbors(*args, **kwargs):

- View Source

★ [new!!!] Deep Neighbors

Using recursion to get all neighbors of a node in a specific level.

Parameters

- **node** (nodeId): A node id of the node.
- **cur_level** (int, optional): Current level of the node, used in the recursion, DO NOT SET THIS ARG, by default 1
- **stop_level** (int, optional): The level to stop the recursion, by default 3

Returns

- **list**: A list of node ids of the neighbors.

```
def deep_common_neighbors(
 self,
 node1: <function NewType.<locals>.new_type>,
 node2: <function NewType.<locals>.new_type>,
 stop_level: int = 3
) -> list:
```

- View Source

★ [new!!!] Deep Common Neighbors

Return the common neighbors of two nodes in a specific level.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.
- **stop_level** (int, optional): The level to stop the recursion, by default 3, by default 3

Returns

- **list**: Return the common neighbors of two nodes in a specific level.

```
def jaccard_coefficient(*args, **kwargs):
```

- View Source

Jaccard Coefficient

Calculate the Jaccard coefficient score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **float**: The Jaccard coefficient score between two nodes.

Examples

```
from core import Graph

graph = Graph()
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(2, 3)

print(graph.jaccard_coefficient(1, 2))
>>> 0.5
```

def deep_jaccard_coefficient(*args, **kwargs): • View Source

def adamic_adar(*args, **kwargs): • View Source

Adamic-Adar

Calculate the Adamic-Adar score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **float**: The Adamic-Adar score between two nodes.

Examples

```
from core import Graph

graph = Graph()
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(2, 3)

print(graph.adamic_adar(1, 2))
>>> 0.7213475204444817
```

def shortest_path(*args, **kwargs): • View Source

Shortest Path

Calculate the shortest path score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **int**: The shortest path score between two nodes.

Examples

```
from core import Graph

graph = Graph()
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(2, 4)

print(graph.shortest_path(1, 4))
>>> 2
```

def katz_score(*args, **kwargs): • View Source

Katz Score

Calculate the Katz score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **float**: The Katz score between two nodes.

Examples

```
graph = Graph()
graph.add_edge(1, 2)
graph.add_edge(1, 4)
graph.add_edge(4, 3)
graph.add_edge(4, 2)

print(graph.katz_score(1, 4, alpha=0.8))
>>> 1.6
```

```
def preferential_attachment(
 self,
 node1: <function NewType.<locals>.new_type>,
 node2: <function NewType.<locals>.new_type>
) -> int:
```

- View Source

Preferential Attachment

Calculate the preferential attachment score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **int**: The preferential attachment score between two nodes.

Examples

```python graph = Graph()

graph.add_edge(1, 2) graph.add_edge(1, 4) graph.add_edge(4, 3) graph.add_edge(4, 2)

print(graph.preferential_attachment(1, 4))
```

Inherited Members

graph_algo.base.BaseGraph add_edge(), get_nodes, get_average_degree, get_neighbors_size(), get_neighbors()

graph_algo.pipeline

- View Source

class ScoreFuncPipeline:

- View Source

ScoreFuncPipeline(**socre_func: dict)

- View Source

Score Function Pipeline

A pipeline for calculating the score of the graph

Parameters

- ****socre_func** (dict): The key is the name of the score function, and the value is the score function.

Example

```
def cal_neighbors_size(row, **kwargs):
    return kwargs['graph'].get_neighbors_size(row["node1"])

def cal_common_neighbors(row, **kwargs):
    return kwargs['graph'].common_neighbors(row["node1"], row["node2"])

socre_func = {
    "out": cal_neighbors_size,
    "common_neighbors": cal_common_neighbors
}

pipeline = ScoreFuncPipeline(**socre_func)
sc_train, sc_test = pipeline.transform(graph=graph, df_train=train, df_test=test)
```

def transform(self, graph: dict, **kwargs) -> list:

- View Source

Parameters

- **graph** (dict): _description_
- ****kwargs** (dict):
    1. If the key of kwargs starts with `df_` , it will be considered as a dataframe, and will be transformed by the score function.
    2. Otherwise, it will be passed as a parameter to the score function.

Returns

- **list[pd.DataFrame]**: return a list of transformed dataframe.

def cal_func_score(
self,
df: pandas.core.frame.DataFrame,
**kwargs
) -> pandas.core.frame.DataFrame:

- View Source

graph_algo.score_func

- View Source

class CommonNeighbors:

- View Source

Common neighbors

```
@staticmethod
def func(
 self,
 node1: <function NewType.<locals>.new_type>,
 node2: <function NewType.<locals>.new_type>
) -> list:
```

- View Source

Calculate the common neighbors score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **int**: The common neighbors score between two nodes.

class JaccardCoefficient:

- View Source

Jaccard coefficient

```
@staticmethod
def func(
 self,
 node1: <function NewType.<locals>.new_type>,
 node2: <function NewType.<locals>.new_type>
) -> float:
```

- View Source

Calculate the Jaccard coefficient score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **float**: The Jaccard coefficient score between two nodes.

class DeepJaccardCoefficient:

- View Source

Deep Jaccard coefficient

```
@staticmethod
def func(
 self,
 node1: <function NewType.<locals>.new_type>,
 node2: <function NewType.<locals>.new_type>,
 max_depth: int = 2
) -> float:
```

- View Source

summary

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.
- **max_depth** (int, optional): The maximum depth of the search, by default 2

Returns

- **float**: Jaccard coefficient score between two nodes.

class AdamicAdar:

- View Source

Adamic-Adar

```
@staticmethod
def func(
 self,
 node1: <function NewType.<locals>.new_type>,
 node2: <function NewType.<locals>.new_type>
) -> float:
```

- View Source

Calculate the Adamic-Adar score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **float**: The Adamic-Adar score between two nodes.

class ShortestPath:

- View Source

Shortest path

```
@staticmethod
def func(
 self,
 node1: <function NewType.<locals>.new_type>,
 node2: <function NewType.<locals>.new_type>,
 max_depth: int = 6
) -> int:
```

- View Source

Calculate the shortest path score between two nodes.

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.

Returns

- **int**: The shortest path score between two nodes.

class KatzScore:

- View Source

Katz score

```
@staticmethod
def func(
 self,
 node1: <function NewType.<locals>.new_type>,
 node2: <function NewType.<locals>.new_type>,
 alpha: float = 1.0,
 beta: float = 1.0,
 max_length: int = 1000
):
```

- View Source

Calculate the Katz score between two nodes.

1. Measure the relative degree of influence of an actor (or node) within a social network
2. Measures influence by taking into account the **total number of walks between a pair of actors**

Similar

1. PageRank
2. Eigenvector centrality

Parameters

- **node1** (nodeId): A node id of the first node.
- **node2** (nodeId): A node id of the second node.
- **alpha** (float): Attenuation factor. 衰減係數，用來控制遠近的影響力，通常介於 0~1 之間 Connections made with distant neighbors are, however, penalized by an attenuation factor *alpha*. Each path or connection between a pair of nodes is assigned a weight determined by *alpha* and the distance between nodes
- **beta** (float, optional): Weight attributed to the immediate neighborhood, by default 1
- **max_length** (int, optional):

References

1. Katz Centrality (Centrality Measure) - GeeksforGeeks
2. katz和eigenvector 中心性

```
@staticmethod
def get_all_possible_path(
 self,
 start: <function NewType.<locals>.new_type>,
 max_length: int
) -> list:
```

- View Source

Get all possible path from start node to other nodes.

Parameters

- **start** (nodeId): A node id of the start node.
- **max_length** (int): The max length of the path.

Returns

- **list**: A list of all possible path.

graph_algo.sparsification

- View Source

class DegreeBased:

- View Source

DegreeBased

DegreeBased(graph: dict, degree: int, operation=<built-in function gt>)

- View Source

summary

Parameters

- **graph** (dict): The graph to be sparsified
- **degree** (int or list): The degree to be sparsified. The type of degree can be int or list. If the type is int, then the graph will be sparsified by the degree. If the type is list, then the graph will be sparsified by the degree in the list.
- **operation** (_type_, optional): The operation to be used to remove the node by degree, by default operator.gt

Example

```
from core import DegreeBased
from core import Graph

graph = Graph()
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(1, 4)
graph.add_edge(2, 1)
graph.add_edge(2, 3)
graph.add_edge(2, 4)
graph.add_edge(2, 5)
graph.add_edge(3, 1)

sparsified_graph = DegreeBased(graph=graph, degree=2).fit()
print(sparsified_graph.edges)

>>> {1: [2], 2: [1]}
```

def sparsify(self):

- View Source

Sparsify the graph by degree.

def fit(self) -> dict:

- View Source

return the sparsified graph

class RandomWalk:

- View Source

RandomWalk

```
@staticmethod
def fit(
 graph: dict,
 node1_dropout: float = 0.1,
 neighbor_dropout: float = 0.1
) -> dict:
```

- View Source

Parameters

- **graph** (dict): The graph to be sparsified
- **node1_dropout** (float, optional): The percentage of node1 to be dropped out, by default 0.1
- **neighbor_dropout** (float, optional): The percentage of neighbor to be dropped out, if the dropout size is 0, then make it 1, by default 0.1

Returns

- **dict**: The sparsified graph

Example

```
from core import RandomWalk
from core import Graph

graph = Graph()
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(1, 4)
graph.add_edge(2, 1)
graph.add_edge(2, 3)
graph.add_edge(2, 4)
graph.add_edge(3, 1)

sparsified_graph = RandomWalk.fit(graph=graph, node1_dropout=0.1, neighbor_dropout=0.1)
print(sparsified_graph.edges)

>>> {1: [3, 4], 2: [3, 4], 3: [1]}
```