Practice of Social Media Analytics

CS5128701

工管產碩數據一

M11101T04

Hsiang-Jen Li 李享紝

Table of Contents

core

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

```
.
├── ./00_demo_usage.ipynb # 使用演算法範例
├── ./01_degree_random.ipynb # 資料前處理 + 特徵工程 + Ensemble Voting Predict
├── ./Data
│   ├── ./Data/new_test_data.csv  # 訓練用資料
│   ├── ./Data/new_train_data.csv # 測試用資料
│   └── ./Data/sample_submit.csv
├── ./README.md
├── ./core
│   ├── ./core/__init__.py
│   ├── ./core/base.py # 基本 Graph 的 attributes
│   ├── ./core/graph.py # 主要 Graph + scoring function
│   ├── ./core/pipeline.py # 方便建立 scoring function(feature engineering) 的 PIPELINE
│   ├── ./core/score_func.py # 所有 scoring function (JC, CN, PA....)
│   └── ./core/sparsification.py # 稀疏化的模組 (Degree-Based, Random-Walk)
```

Create documentation from source code

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

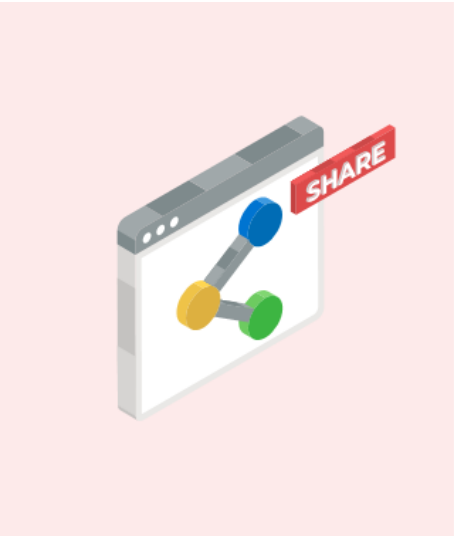
Report

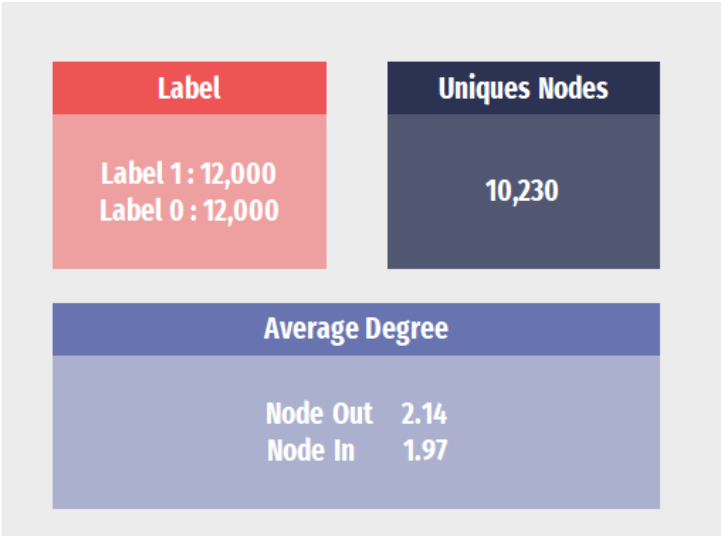
Implementation of 6 Algorithms

1. CommonNeighbors
2. JaccardCoefficient
3. AdamicAdar
4. ShortestPath
5. KatzScore
6. PreferentialAttachment

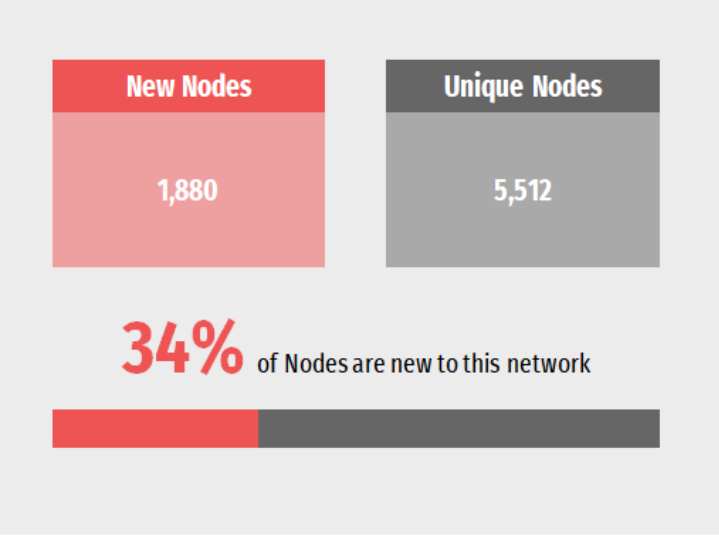
Basic Understand of Dataset

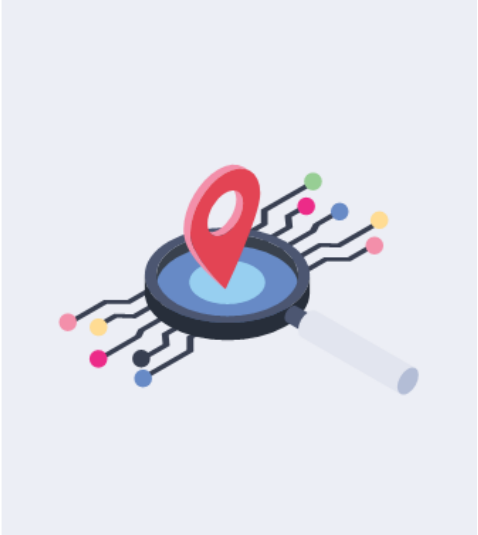
Train set





Test Set





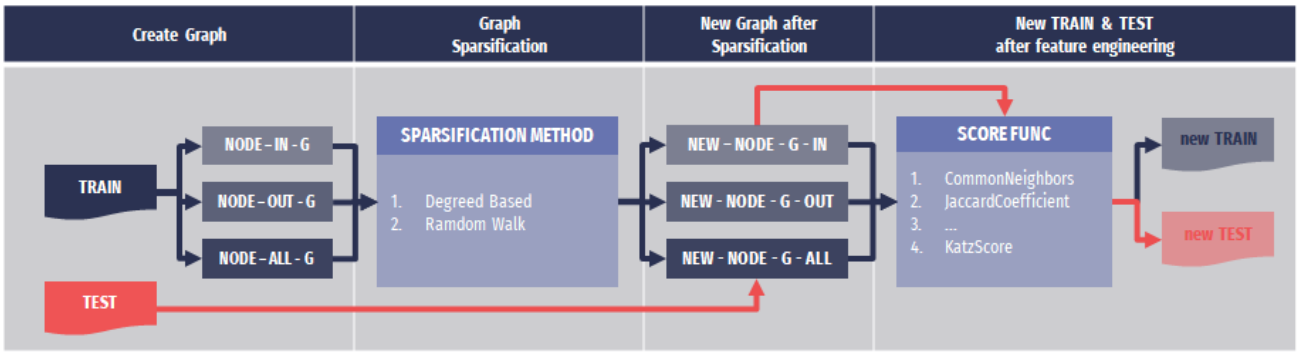
Flow Chart

pseudocode

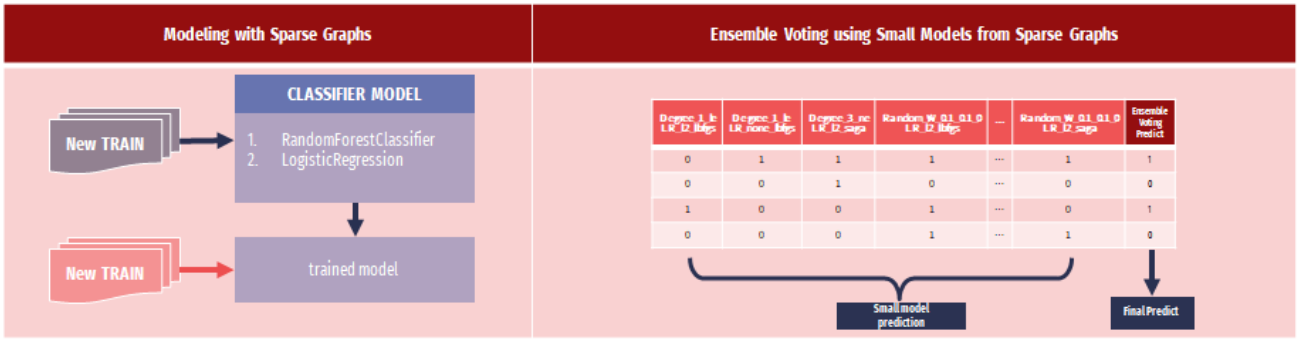
1. Create Graph using train data where label is 1
2. Sparsify the Graph using Degree-Based and RandomWalk methods
3. Generate new features (CN, JC, AA, PA) using ScoreFunc based on new graphs and train/test data
4. Train a small model on the new train data (excluding node1 and node2)
5. Use the trained models to predict test data (excluding node1 and node2) and perform voting

Preprocessing

- **Node Out** : The number of edges pointing from Node1 to other nodes.
- **Node In** : The number of edges pointing to Node1 from other nodes.
- **Node All** : Refers to an undirected graph.



Modeling



Result

Test Name	Node Direction	ScoreFunc	Model	Accuracy	Note
2023_04_22 test_01-test_jc_pa	Node Out	JC & PA	RandomForest Classifier	0.52333	
2023-04-28-test-02	Node Out	Neighbor Size & CN & JC & AA	RandomForestClassifier, XGBClassifier, LogisticRegression	0.52916	Use a voting method to make predictions based on the predicted results from RFC, XGB, LR
2023-04-30 test-08	Node Out & In	Neighbor Size & CN & JC & PA	Logistic Regression	0.67194	
2023_05_07 test_02_rw_db_lr	Node Out & In	Neighbor Size & CN & JC & PA	Logistic Regression	**0.67305**	• Degree-based Sparsification • Random-Walk Sparsification • Train logistic regression models with different parameters on the training set • Use a voting method to make predictions based on the predicted results (if more than half predict 1, then it is considered as 1)

1. Here are some results from a few tests, as we ran too many tests and are uncertain about which parameters were used
2. In terms of model performance, LogisticRegression clearly performed better than RandomForestClassifier
3. Regarding the direction of nodes, using all three types of direction (In, Out, All) did not result in better performance than using just In and Out

4. For FuncScore, using all values did not improve model accuracy, whereas using Neighbor Size, CN, JC, and PA yielded relatively better results
5. The accuracy on the final test set was about 67%

- View Source

core.graph

- View Source

class Graph(core.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 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 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

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

core.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

core.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>
) -> int:
```

- 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 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.

core.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

```python 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]}
```