网络分析基础 C12

胡俊峰 北京大学

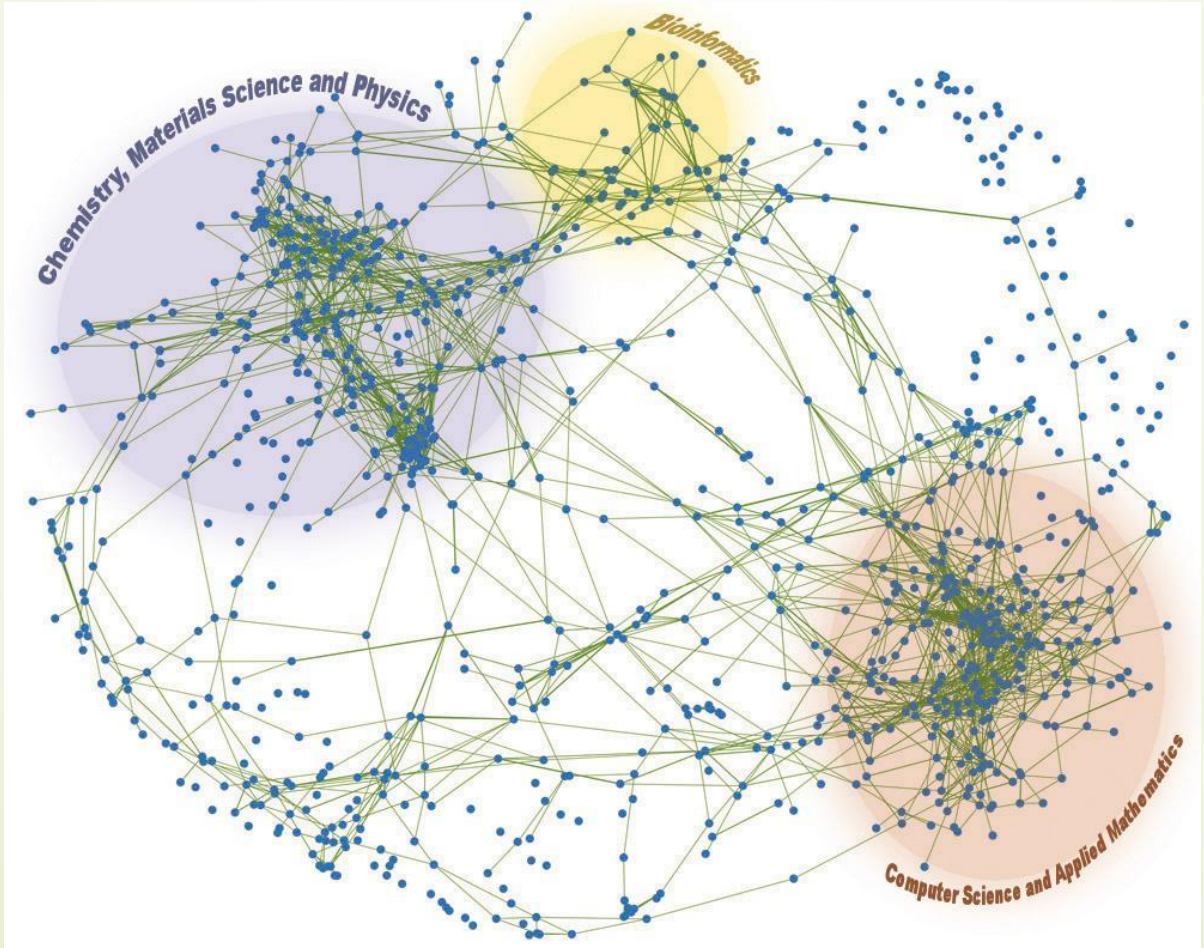
2023/04/13

网络(图)分析基础

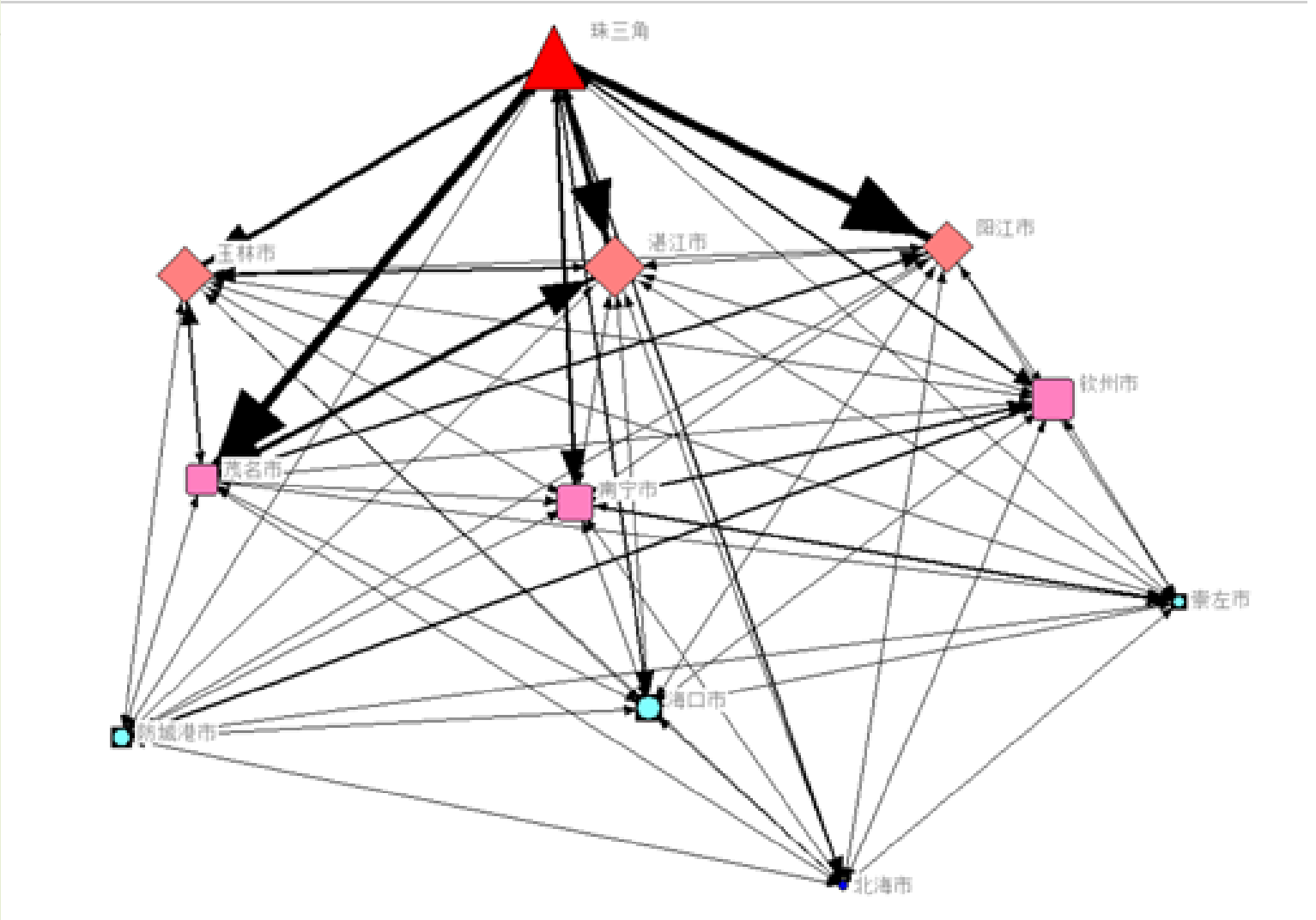
- 网络模型简介
- 网络（图）建模基础
- 图参数
- PageRank、HITS算法
- 图嵌入

网络模型的问题提出

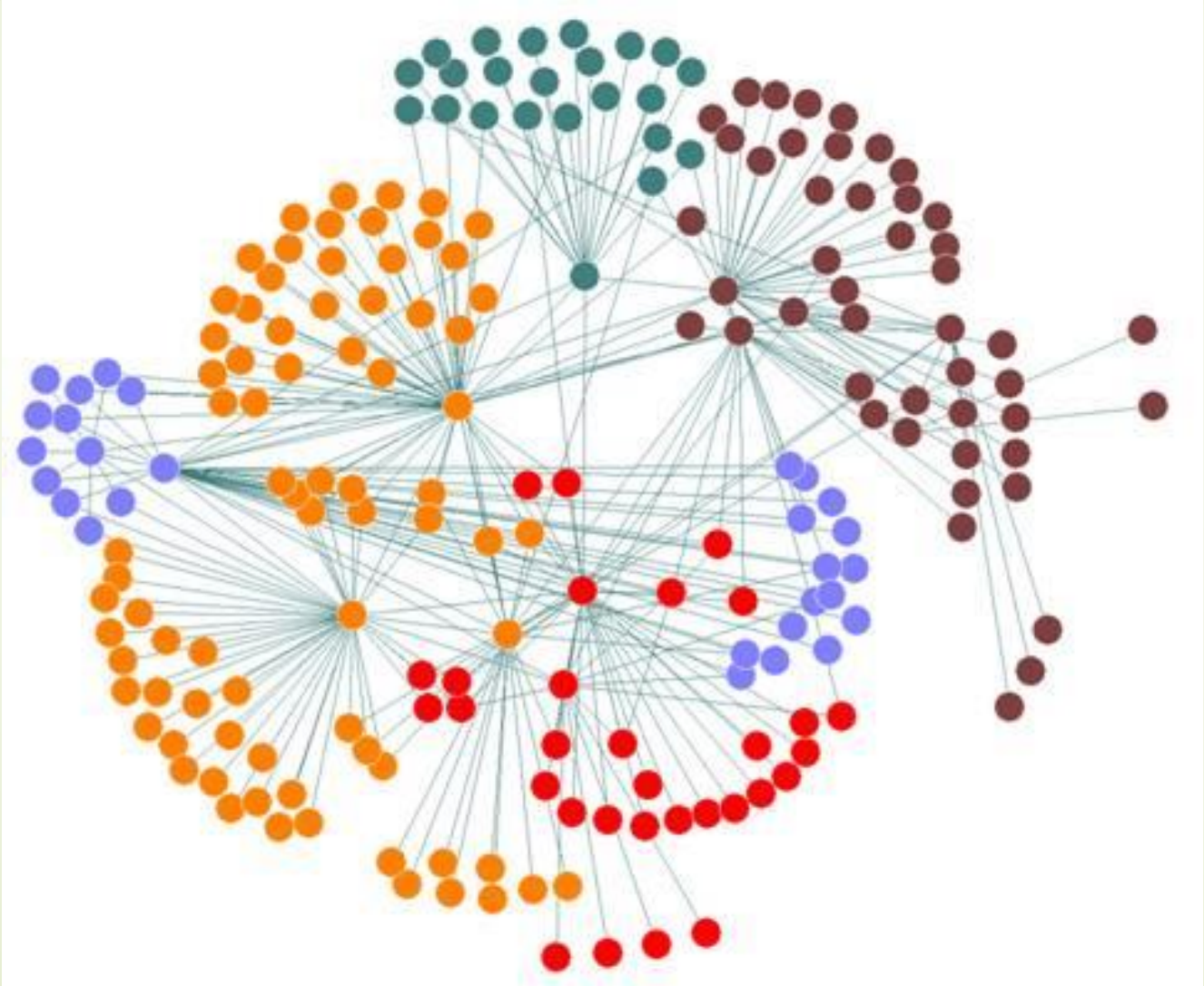
- 关联及属性：网络链接（WWW），社交网路（朋友圈），合作网络



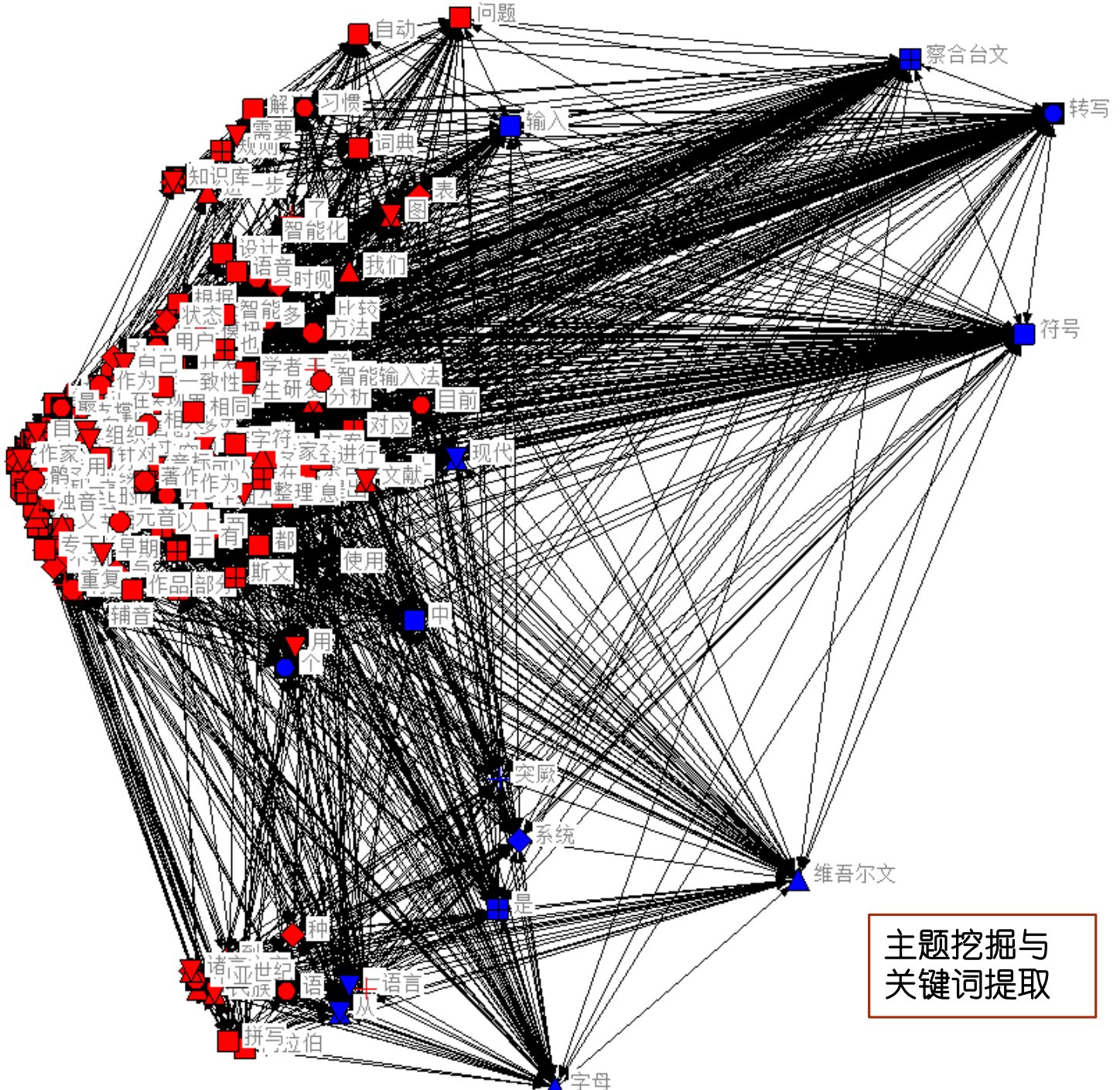
网络架构是一种重要信息：



社区、中心、桥、层次都可以是属性



为传统问题提供新的网络化视角

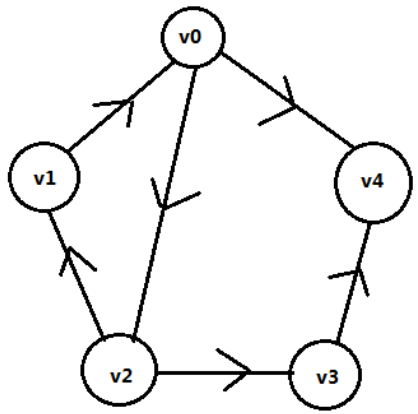


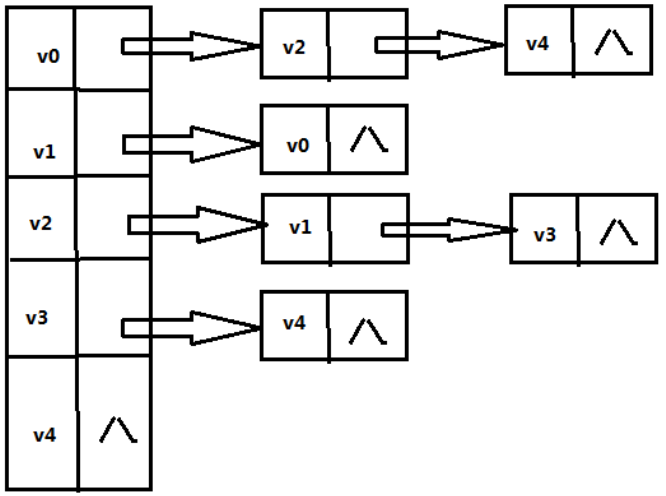
研究目标：

- 提供一种社会化的视角对网络的总体特征，社区划分、中心性、关联性等进行定量-定性的分析
- 对网络演进、关联推荐给出可验证的方案
- 提供一种多模态非空间化的建模与解决方案

图分析数据结构方案

- 关联矩阵模型
- 邻接表模型





图分析软件包：Install networkX

```
(base) C:\Users\hujf>pip install networkx
Requirement already satisfied: networkx in c:\users\hujf\anaconda3\lib\site-packages (2.5)
Requirement already satisfied: decorator>=4.3.0 in c:\users\hujf\anaconda3\lib\site-packages (from networkx) (5.0.6)

(base) C:\Users\hujf>pip install --upgrade networkx
Requirement already satisfied: networkx in c:\users\hujf\anaconda3\lib\site-packages (2.5)
Collecting networkx
  Downloading networkx-2.7.1-py3-none-any.whl (2.0 MB)
     |████████████████████████████████| 2.0 MB 819 kB/s
Installing collected packages: networkx
  Attempting uninstall: networkx
    Found existing installation: networkx 2.5
    Uninstalling networkx-2.5:
      Successfully uninstalled networkx-2.5
Successfully installed networkx-2.7.1

(base) C:\Users\hujf>
```

Reference — NetworkX 2.7.1 documentation

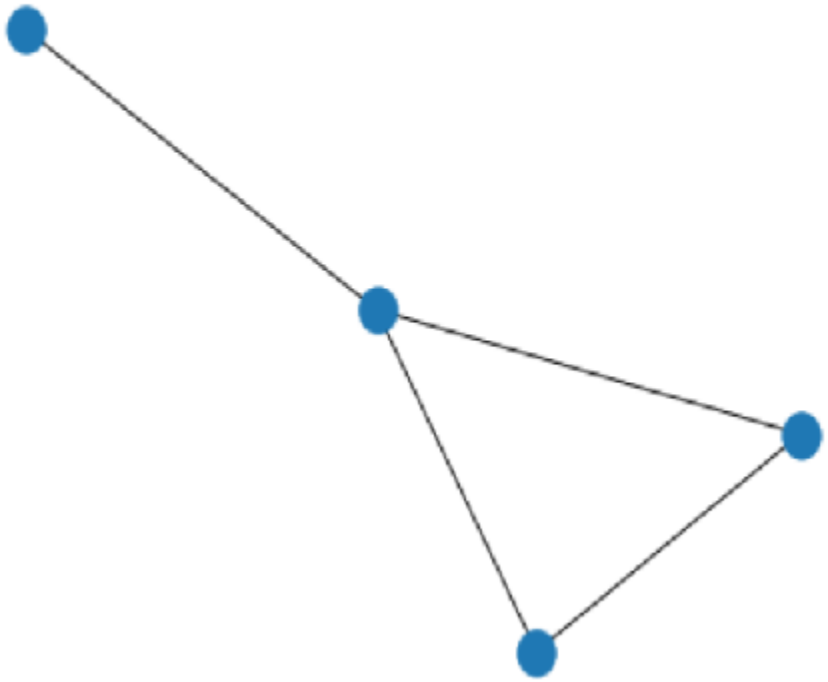
创建一个图：

```
G = nx.Graph()
elist = [(1, 2), (2, 3), (1, 4), (4, 2)] # 边list，定点对的tuple
G.add_edges_from(elist)                  # 从list导入边集

print(G)
display(G)
G.adjacency()  # 返回一个节点连边的迭代器
for eg in G.adjacency():
    print(eg)
```

创建一个无向图对象

```
Graph with 4 nodes and 4 edges

<networkx.classes.graph.Graph at 0x18df97cd100>

(1, {2: {}, 4: {}})
(2, {1: {}, 3: {}, 4: {}})
(3, {2: {}})
(4, {1: {}, 2: {}})
```

```
nx.draw(G)
```

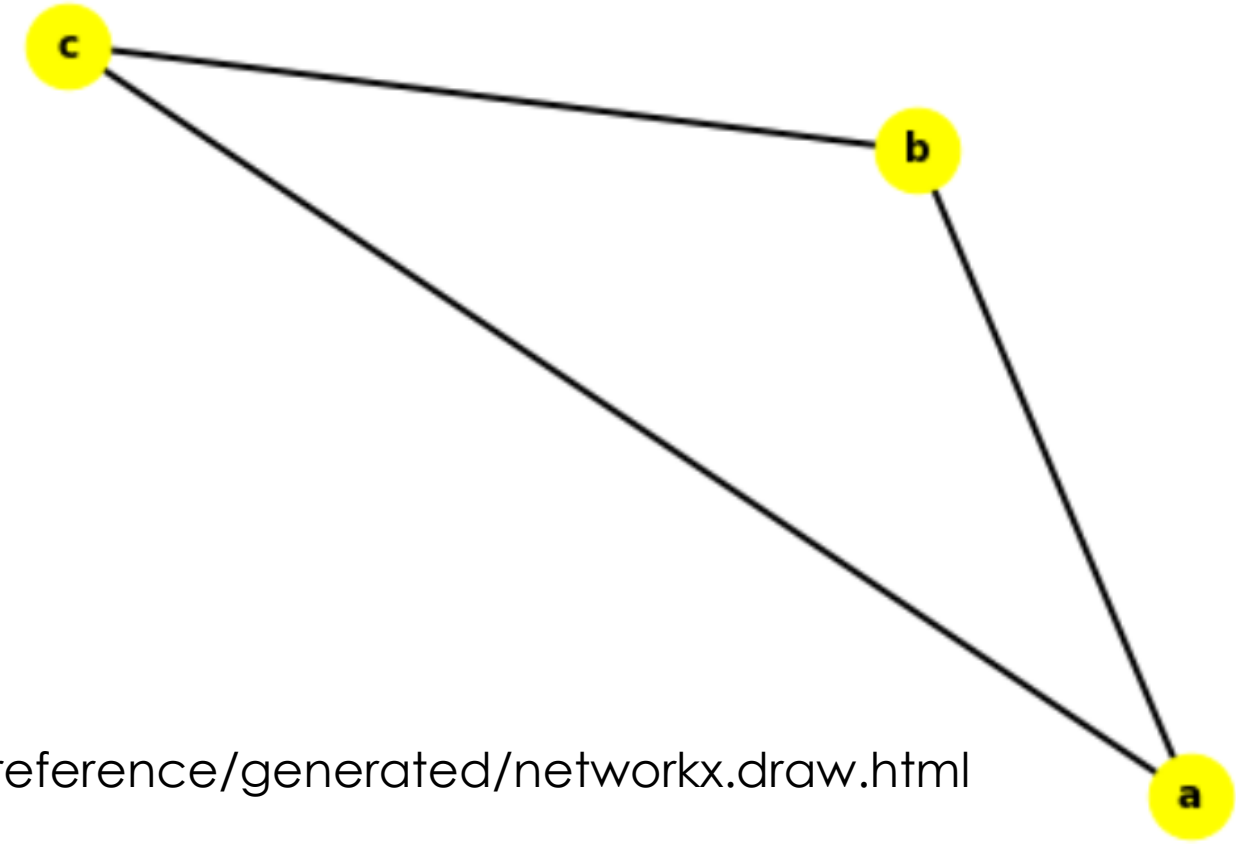


nx.draw(G, pos=None, ax=None, hold=None, **kwds)

常用参数：

- **with_labels**：是否绘制节点标签
- **nodelist/edgelist**：绘制的节点集、边集
- **node_size**：节点大小**(default=300)**
- **node_color**：节点颜色
- **node_shape**：节点形状
- **cmap**：**colormap**
- **width**：边宽
- **edge_color**：边颜色
- **font_size**：标签中字的大小
- **font_color**：标签中字的颜色

```
G2 = nx.Graph()
elist = [('a', 'b', 5.0), ('b', 'c', 5.0), ('a', 'c', 1.0)]
G2.add_weighted_edges_from(elist)
option={
        'with_labels':True, 'font_weight':'bold',
        'node_color':'yellow','node_size':700,'width':2
       }

nx.draw(G2, **option)
```



https://networkx.org/documentation/networkx-1.0/reference/generated/networkx.draw.html

Graph Reporting

```
for nodes in G2:
    print(nodes)  # iter all nodes

for ne in G2.neighbors('a'): # iter, neighbers
    print(ne)

print(G2.order()) # number of nodes
```

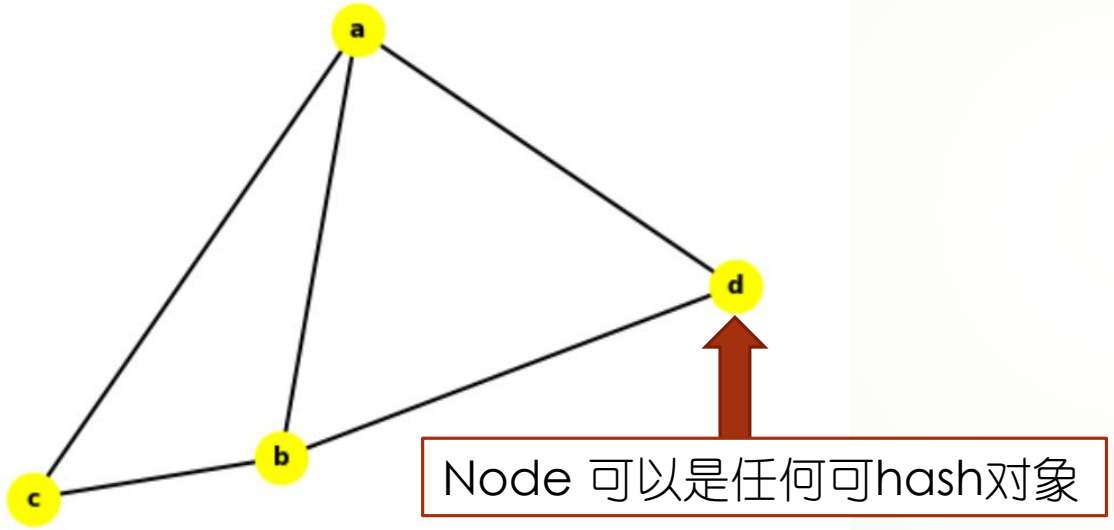
```
a
b
c
b
c
3
```

Reporting nodes edges and neighbors

`Graph.nodes`	A NodeView of the Graph as G.nodes or G.nodes().
`Graph.__iter__`()	Iterate over the nodes.
`Graph.has_node`(n)	Returns True if the graph contains the node n.
`Graph.__contains__`(n)	Returns True if n is a node, False otherwise.
`Graph.edges`	An EdgeView of the Graph as G.edges or G.edges().
`Graph.has_edge`(u, v)	Returns True if the edge (u, v) is in the graph.
`Graph.get_edge_data`(u, v[, default])	Returns the attribute dictionary associated with edge (u, v).
`Graph.neighbors`(n)	Returns an iterator over all neighbors of node n.
`Graph.adj`	Graph adjacency object holding the neighbors of each node.
`Graph.__getitem__`(n)	Returns a dict of neighbors of node n.
`Graph.adjacency`()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
`Graph.nbunch_iter`([nbunch])	Returns an iterator over nodes contained in

元素操作函数：

```
G2.add_edges_from([('a', 'd'), ('d', 'b')])
nx.draw(G2, **option)
```

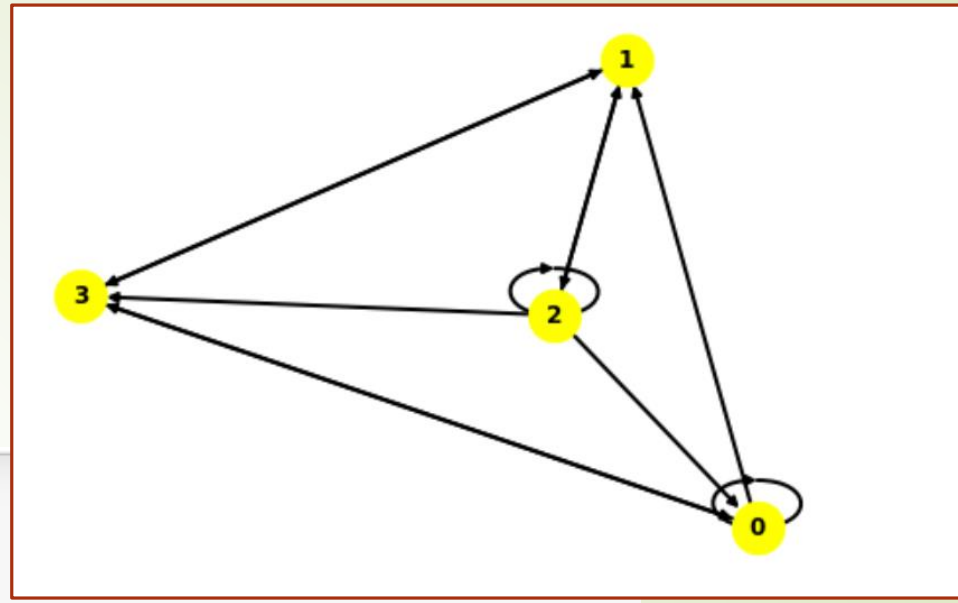


Node 可以是任何可hash对象

https://networkx.org/documentation/latest/tutorial.html

Graph.__init__([incoming_graph_data])	Initialize a graph with edges, name, or graph attributes.
Graph.add_node(node_for_adding, **attr)	Add a single node node_for_adding and update node attributes.
Graph.add_nodes_from(nodes_for_adding, **attr)	Add multiple nodes.
Graph.remove_node(n)	Remove node n.
Graph.remove_nodes_from(nodes)	Remove multiple nodes.
Graph.add_edge(u_of_edge, v_of_edge, **attr)	Add an edge between u and v.
Graph.add_edges_from(ebunch_to_add, **attr)	Add all the edges in ebunch_to_add.
Graph.add_weighted_edges_from(ebunch_to_add)	Add weighted edges in ebunch_to_add with specified weight attr
Graph.remove_edge(u, v)	Remove the edge between u and v.
Graph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
Graph.update([edges, nodes])	Update the graph using nodes/edges/graphs as input.
Graph.clear()	Remove all nodes and edges

从其他数据源接入数据
—— from Numpy




```
import numpy as np
ar = np.array([[1,1,0,1],[0,0,1,1],[2,2,2,4],[1,2,0,0]])
```

```
DG = nx.from_numpy_array(ar, create_using=nx.DiGraph)
for i in DG.adjacency():
    print(i)
```

邻接表方式

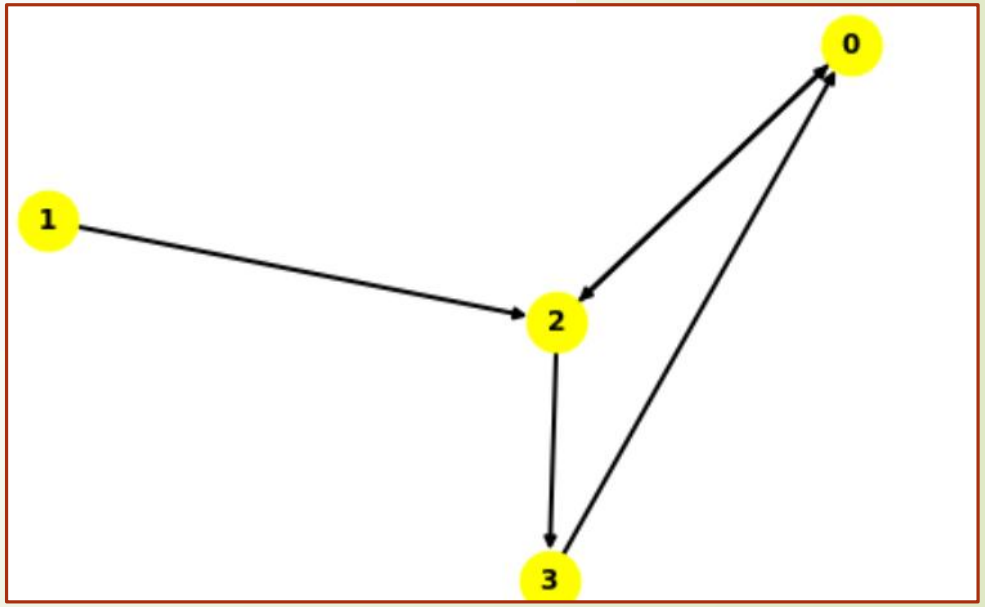
```
(0, {0: {'weight': 1}, 1: {'weight': 1}, 3: {'weight': 1}})
(1, {2: {'weight': 1}, 3: {'weight': 1}})
(2, {0: {'weight': 2}, 1: {'weight': 2}, 2: {'weight': 2}, 3: {'weight': 4}})
(3, {0: {'weight': 1}, 1: {'weight': 2}})
```

```
nx.draw(DG, **option)
```

Converting to and from other data formats — NetworkX 2.7.1 documentation

From pandas dataframe

```
import pandas as pd
edges = pd.DataFrame(                    # 起点、终点、权重 3列数据
    {
        "source": [0, 1, 2, 2, 3],
        "target": [2, 2, 3, 0, 0],
        "weight": [3, 4, 5, 6,7]
    }
)
MG = nx.from_pandas_edgelist(
    edges,
    create_using=nx.DiGraph(),
)
nx.draw(MG, **option)
```



```
adjacency_dict = {0: (1, 2,3,4), 1: (0, 2,3), 2: (0, 1,4),4:(1,3,4)}
H = nx.Graph(adjacency_dict)  # create a Graph dict mapping nodes to nbrs
#list(H.edges())
a = np.array(H)
a
```

```
array([0, 1, 2, 4, 3])
```

转出为其他格式数据

```
b = np.array(H.edges)
b
```

```
array([[0, 1],
       [0, 2],
       [0, 3],
       [0, 4],
       [1, 2],
       [1, 3],
       [1, 4],
       [2, 4],
       [4, 3],
       [4, 4]])
```

基于关联的网络结构相似度 — connection-based Similarity

- 基于邻居的相似度 Similarity based on neighborhoods.
  - 公共邻居数（向量点积）
    - $score(x,y) = |\Gamma(x) \cap \Gamma(y)|$
  - Jaccard 系数 Jaccard's coefficient
    - $score(x,y) = |\Gamma(x) \cap \Gamma(y)| / |\Gamma(x) \cup \Gamma(y)|$
  - Adamic/Adar系数
    - $score(x,y) = \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{log|\Gamma(z)|}$
      - 1/ log | Γ (z) | 类似节点关联的IDF加权

基于多重路径的相似度（距离）方案

- Katz距离：
  - 两点间所有路径的加权和

$$\mathsf{score}(x,y) := \sum_{\ell=1}^{\infty} \beta^{\ell} \cdot |\mathsf{paths}_{x,y}^{\langle \ell \rangle}|,$$

Katz中心度（A为关联阵，）：

$$C_{\mathrm{Katz}}(i) = \sum_{k=1}^{\infty} \sum_{j=1}^{n} \alpha^k (A^k)_{ji}$$

- SimRank：Similarity based Katz score + degree regularization（点度正则化）
  - 关联节点全连接路径的Katz相似度之加正则化

$$\mathsf{similarity}(x,y) := \gamma \cdot \frac{\sum_{a \in \Gamma(x)} \sum_{b \in \Gamma(y)} \mathsf{similarity}(a,b)}{|\Gamma(x)| \cdot |\Gamma(y)|}$$

递归定义

节点的静态特征（中心度）

- 中心度（ Centrality ）
  - 点度中心度（degree centrality）
  - 接近中心度（closeness centrality）* N-1
  - $$CC(x) = \frac{1}{\sum_y d(x, y))}$$
  - 介度中心度（betweeness centrality）

图算法：Centrality：

- 点度中心度
- 特征向量中心度
- 接近中心度
- 介度中心度（betweeness）

Introduction
Graph types
Algorithms
Approximations and Heuristics
Assortativity
Asteroidal
Bipartite
Boundary
Bridges
Centrality
Chains
Chordal
Clique
Clustering
Coloring
Communicability
Communities
Components
Connectivity
Cores
Covering
Cycles
Cuts
D-Separation
Directed Acyclic Graphs

Centrality

Degree

`degree_centrality`(G)	Compute the degree centrality for nodes.
`in_degree_centrality`(G)	Compute the in-degree centrality for nodes.
`out_degree_centrality`(G)	Compute the out-degree centrality for nodes.

Eigenvector

`eigenvector_centrality`(G[, max_iter, tol, ...])	Compute the eigenvector centrality for the graph `G`.
`eigenvector_centrality_numpy`(G[, weight, ...])	Compute the eigenvector centrality for the graph G.
`katz_centrality`(G[, alpha, beta, max_iter, ...])	Compute the Katz centrality for the nodes of the graph G.
`katz_centrality_numpy`(G[, alpha, beta, ...])	Compute the Katz centrality for the graph G.

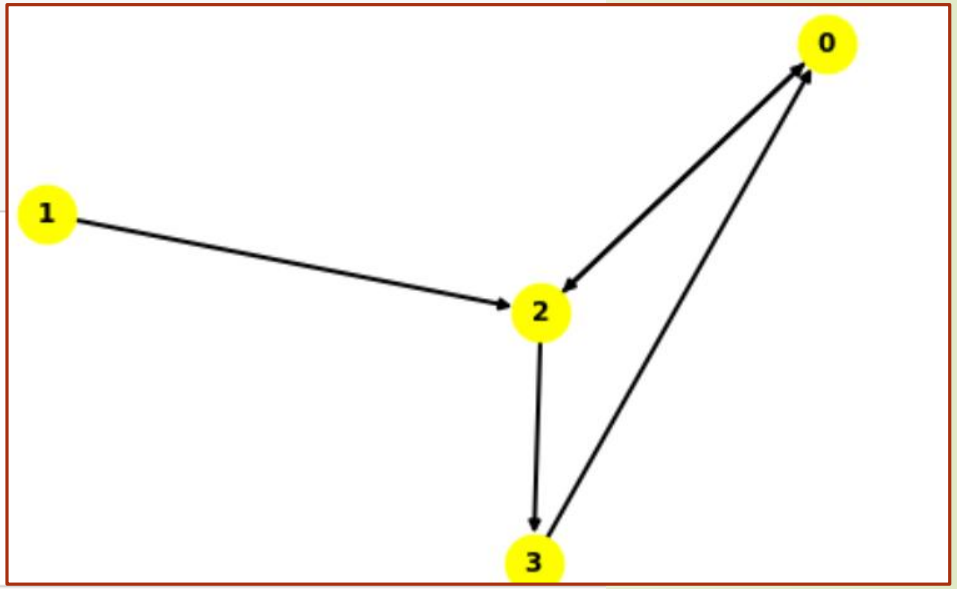
Closeness

```python
@not_implemented_for("undirected")
def in_degree_centrality(G):

    if len(G) <= 1:
        return {n: 1 for n in G}

    s = 1.0 / (len(G) - 1.0)
    centrality = {n: d * s for n, d in G.in_degree()}
    return centrality
```

点度中心度

```python
print(len(MG))
for n,d in MG.in_degree():
    print(n,d)
```

```
4
0 2
2 2
1 0
3 1
```




```python
id = nx.in_degree_centrality(MG)  # 输出为一个词典
id
```

```
{0: 0.6666666666666666, 2: 0.6666666666666666, 1: 0.0, 3: 0.3333333333333333}
```

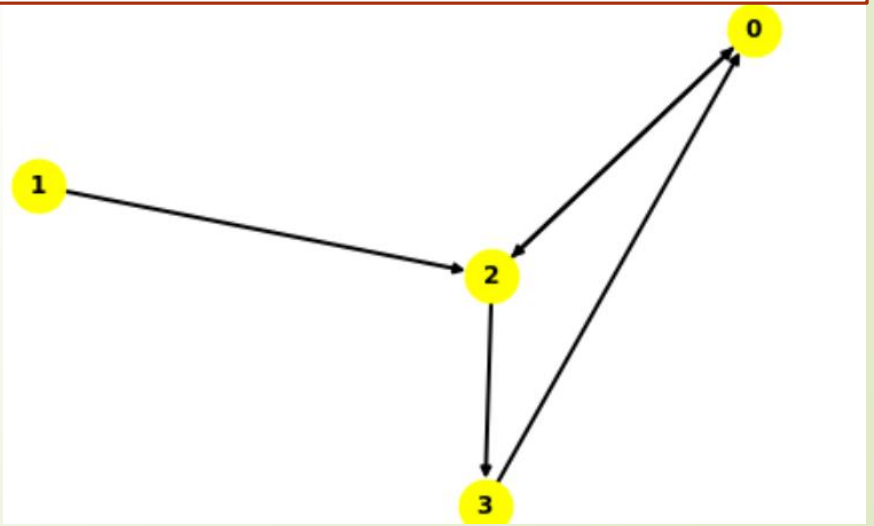
特征向量中心度 eigenvector_centrality
介度中心度 betweenness_centrality

```
centrality = nx.eigenvector_centrality(MG)

sorted((v, f"{c:0.2f}") for v, c in centrality.items())
```

```
[(0, '0.73'), (1, '0.00'), (2, '0.55'), (3, '0.41')]
```

```
bt_centrality = nx.betweenness_centrality(MG)
sorted(bt_centrality.items(), key = lambda kv:(kv[1], kv[0]), reverse = True)
```

```
[(2, 0.5), (0, 0.16666666666666666), (3, 0.0), (1, 0.0)]
```



PageRank算法

- PageRank，网页排名，又称网页级别、Google左侧排名或佩奇排名，是一种由搜索引擎根据网页之间相互的超链接计算的技术，作为网页排名的要素之一。

PageRank算法

- 简单的理解相当于一个投票过程。
- 可以假设每个人一开始都有1票，然后根据页面的链接指向进行投票。
- step 1：$PR_i = 1$，$1 <= i <= n$
- step 2：$PR_i = \sum_{j=1}^{n} \frac{PR_j}{C_j}$，$j$到$i$有边
- step 3：迭代进行step 2，直至PR值稳定。

PageRank算法

- 把上面的公式写成矩阵形式
- $R_k = AR_{k-1}$
- $A_{i,j} = 0$ j到i无边
- $\frac{1}{c_j}$，j到i有边
- A的每一列的和都是1

PageRank算法

- 最终求的就是$R=AR$，R其实就是A在$\lambda = 1$下的特征向量（HITS）
- 一般使用的时候会加入一个阻尼系数d *(0.85 ?)*
- $R_k = (1-d)\alpha + dAR_{k-1}$ (α为全是1的列向量)

- 随机冲浪模型：
- $R_k = (\frac{1-d}{N})\alpha + dAR_{k-1}$ (N为网页总数)

- 随机冲浪模型与排名推荐

PageRank算法

- 公式中网页与网页间的边权值为1或0
- 红楼梦按有向带权图实现
- 修改公式：
- $PR_i = (1-d) + d\sum_{j=1}^{n} \frac{E_{j,i} PR_j}{C_j}$
- $E_{j,i}$是两个人的共现次数（在同一句 或 同一场景）
- C_j是j出现的次数(加权, 保证 $\sum_{i=1}^{n} E_{j,i} = C_j$)

红楼梦人物Rank排序

pagerank_order.txt

1	贾宝玉-宝玉-宝二爷-怡红公子	43.380625
2	贾母	25.721786
3	王熙凤-熙凤-凤姐-凤哥-凤哥儿-凤辣子-凤丫头	25.247519
4	王夫人-王氏	17.730632
5	林黛玉-黛玉-颦儿-潇湘妃子	14.572775
6	贾政	13.534319
7	薛宝钗-宝钗-蘅芜君	11.925397
8	袭人-花袭人	11.585154
9	贾琏	11.246868
10	尤氏	9.608205
11	贾珍-宁国公	9.601215
12	平儿	8.014226
13	贾探春-探春-蕉下客	6.863918
14	金鸳鸯-鸳鸯	6.275644
15	薛姨妈	5.983160
16	李纨-稻香老农	5.683295
17	邢夫人	5.592737
18	贾蓉	4.841630
19	小丫头	4.818685
20	贾赦-荣国公	4.806279
21	香菱-甄英莲-英莲	4.767188
22	周瑞	4.644967
23	薛蟠	4.582727
24	秦钟	4.468754
25	晴雯	4.292686
26	史湘云-湘云	4.290791

图算法：最短路

- （点对）最短路
- 单源最短路
- 图最短路（All shortest path）

Link Analysis
Link Prediction
Lowest Common Ancestor
Matching
Minors
Maximal independent set
non-randomness
Moral
Node Classification
Operators
Planarity
Planar Drawing
Reciprocity
Regular
Rich Club
Shortest Paths
Similarity Measures
Simple Paths
Small-world
s metric
Sparsifiers
Structural holes
Summarization
Swap
Threshold Graphs
Tournament

Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

`shortest_path`(G[, source, target, weight, ...])	Compute shortest paths in the graph.
`all_shortest_paths`(G, source, target[, ...])	Compute all shortest simple paths in the graph.
`shortest_path_length`(G[, source, target, ...])	Compute shortest path lengths in the graph.
`average_shortest_path_length`(G[, weight, method])	Returns the average shortest path length.
`has_path`(G, source, target)	Returns *True* if *G* has a path from *source* to *target*.

Advanced Interface

Shortest path algorithms for unweighted graphs.

`single_source_shortest_path`(G, source[, cutoff])	Compute shortest path between source and all other nodes reachable from source.
`single_source_shortest_path_length`(G, source)	Compute the shortest path lengths from

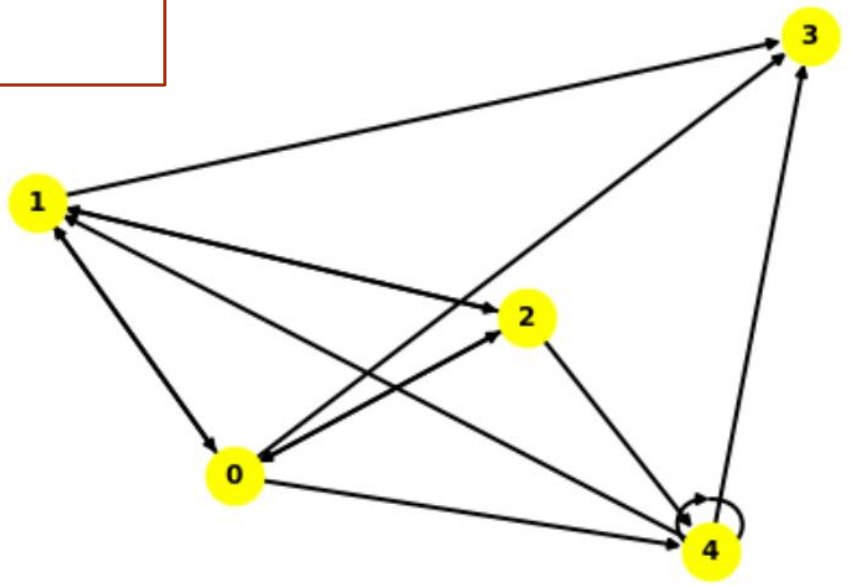
单源最短路：

```
pa = nx.shortest_path(H, source=4)
pa
```

```
{4: [4], 1: [4, 1], 3: [4, 3], 0: [4, 1, 0], 2: [4, 1, 2]}
```

```
pa = nx.shortest_path(H, target=4)
pa
```

```
{4: [4], 0: [0, 4], 2: [2, 4], 1: [1, 0, 4]}
```



jaccard_coefficient

`jaccard_coefficient(G, ebunch=None)` [source]

Compute the Jaccard coefficient of all node pairs in ebunch.

Jaccard coefficient of nodes `u` and `v` is defined as

$$\frac{|\Gamma(u) \cap \Gamma(v)|}{|\Gamma(u) \cup \Gamma(v)|}$$

where $\Gamma(u)$ denotes the set of neighbors of u.

Parameters: **G** : *graph*

A NetworkX undirected graph.

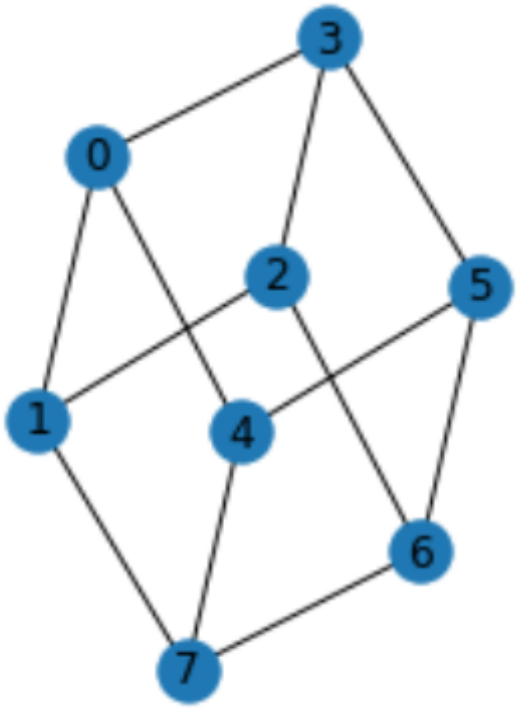
ebunch : *iterable of node pairs, optional (default = None)*

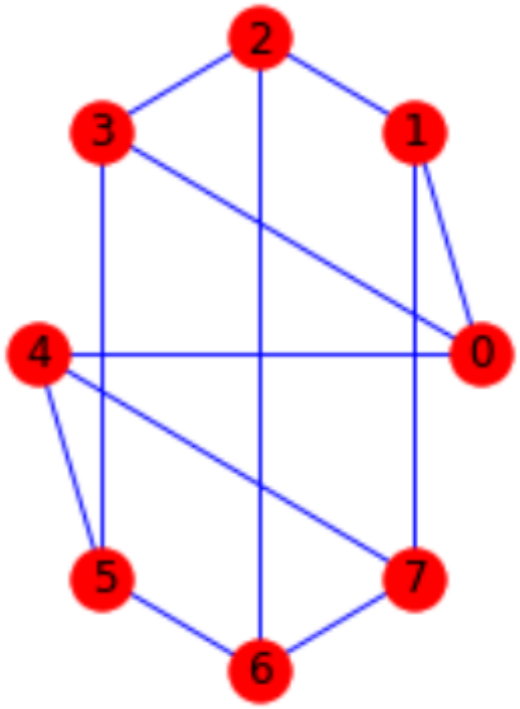
Jaccard coefficient will be computed for each pair of nodes given in the iterable. The pairs must be given as 2-tuples (u, v) where u and v are nodes in the graph. If ebunch is None then all non-existent edges in the graph will be used. Default value: None.

Returns: **piter** : *iterator*

An iterator of 3-tuples in the form (u, v, p) where (u, v) is a pair of nodes and p is their Jaccard coefficient.

```python
G = nx.cubical_graph()      # 立方图
subax1 = plt.subplot(121) # 划分子图，1行两列，目前是1号子图
nx.draw(G, with_labels=True)                    # default spring_layout 弹簧力学模型
subax2 = plt.subplot(122) # 2号子图
nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b', with_labels=True)
```






```python
preds = nx.jaccard_coefficient(G, [(0, 2), (1, 5)])
for u, v, p in preds:
    print(f"({u}, {v}) -> {p:.8f}")
```

```
(0, 2) -> 0.50000000
(1, 5) -> 0.00000000
```

社区发现与网络的凝聚子群分析

K-核分析（社群影响力中心）

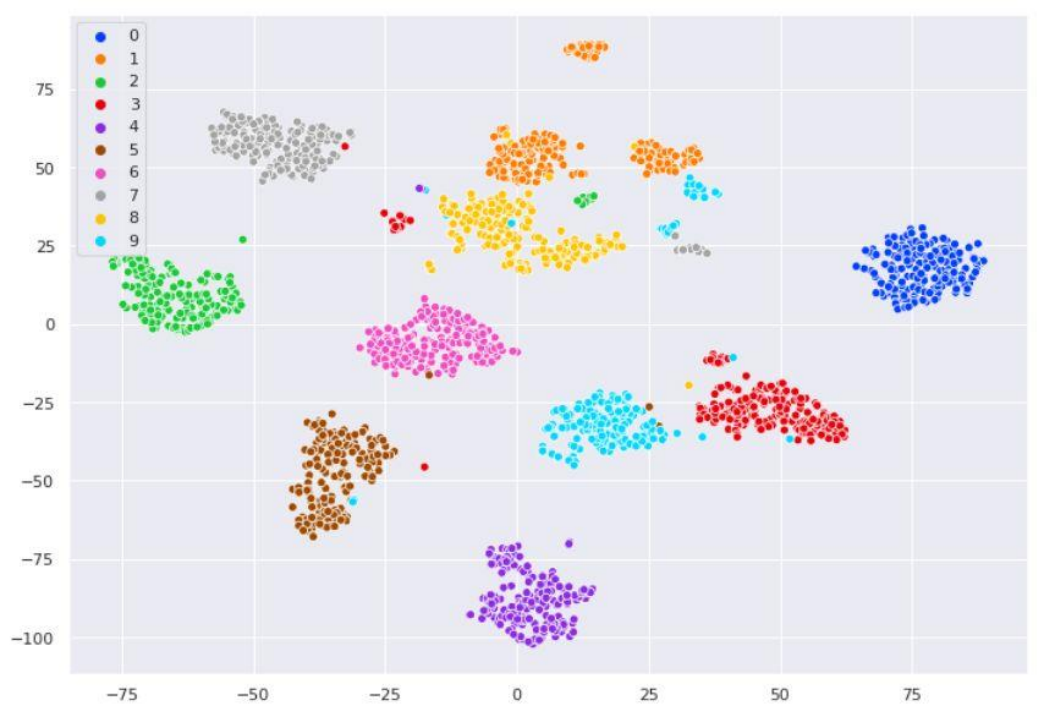
- 选择K值为8，即得到的子图之中的点至少与该子图的8个其他点邻接，来对以上连通子图继续分析。最后，得到了14个顶点的连通图（贾宝玉、王熙凤、贾母、王夫人、薛宝钗、林黛玉、袭人、平儿、贾探春、薛姨妈、贾迎春、贾惜春、李纨、史湘云）；
- 结果子图基本上包含了点度中心度较大的节点，说明了在红楼梦人物中，越是处于核心的人物，越是倾向于和其他核心人物发生联系。
- 如何计算得到网络的K-core ？

```
<level3 id = "春雷 春风 喜讯 捷报 火山 赞歌 云霄 颂歌 战歌" weight = "0.569667">
    <level2 id = "春雷 春风 喜讯 捷报 火山" weight = "0.707107">
        <level1 id = "春雷 春风" weight = "0.685398">
            <word weight = "0.707107" freq = "00000371" pos = "n" >春风</word>
            <word weight = "0.707107" freq = "00000411" pos = "n" >春雷</word>
        </level1>
        <level1 id = "喜讯 捷报" weight = "0.590778">
            <word weight = "0.707107" freq = "00000425" pos = "n" >捷报</word>
            <word weight = "0.707107" freq = "00002794" pos = "n" >喜讯</word>
        </level1>
        <level1 id = "火山" weight = "0.425689">
            <word weight = "1.000000" freq = "00000343" pos = "n" >火山</word>
        </level1>
    </level2>
    <level2 id = "赞歌 云霄 颂歌 战歌" weight = "0.707107">
        <level1 id = "赞歌 云霄" weight = "0.707107">
            <word weight = "0.707107" freq = "00000309" pos = "n" >云霄</word>
            <word weight = "0.707107" freq = "00000387" pos = "n" >赞歌</word>
        </level1>
        <level1 id = "颂歌 战歌" weight = "0.707107">
            <word weight = "0.610537" freq = "00000385" pos = "n" >颂歌</word>
            <word weight = "0.609174" freq = "00000340" pos = "n" >战歌</word>
            <word weight = "0.506114" freq = "00002418" pos = "n" >歌</word>
        </level1>
    </level2>
</level3>
```

Figure 6: Part of 1995-1999 ontology showing “春风” (spring wind) and its semantically similar words.

图的空间嵌入（embedding）：

- 关联作为特征，目标是空间距离尽量反应关联特征（t-SNE）



https://zhuanlan.zhihu.com/p/148170862

PMI（jaccard）关联的PCA近邻嵌入

7]:
```python
#using PMI to calculate the similarity matrix
def similarity_matrix(rating_matrix):
    N = len(rating_matrix)
    S = np.zeros((N,N))
    for i in range(N):
        if (i % 100 == 99): # 算前100测试一下，这个算法太慢
            break
        for j in range(i,N):
            S[i,j] = pmi(i,j,rating_matrix) # i-j pmi
            S[j,i] = S[i,j]
    return S

# user_similarity = similarity_matrix(train_data)

data_mat_freq = np.where(data_matrix>0, 1, 0)    # 评分 ==> 频率
user_similarity_test = similarity_matrix(data_mat_freq)
```

1]:
```python
a = data_mat_freq @ data_mat_freq.T  # 20分钟
np.save("data/movie_coRating_count",a ) #保存数组
a[0]
```

1]:
```
array([53,  7,  6, ...,  0, 15, 17])
```

Jaccard关系

```
1  person_moive_count = np.sum(data_mat_freq,axis = 1)
```

```
1  ja_mat= np.zeros(a.shape)
2  ja_mat[0]
```

```
array([0., 0., 0., ..., 0., 0., 0.])
```

```
1  for i in range(len(a[0])):
2      for j in range(i):
3          ja_mat[i,j] = a[i,j]/ (person_moive_count[i]+person_moive
4          ja_mat[j,i] = ja_mat[i,j]
5  ja_mat[0]
```

```
array([0.        , 0.04      , 0.06122449, ..., 0.        , 0.09316
77 ,
       0.04509284])
```

关联做属性；PCA

```
1  Y = ja_mat.copy()
2  Y -= np.mean(Y,axis = 0) # 按列中心化
3
4  # 理论上应该把自己相似的维度去掉，形成n-1维属性，然后中心化，PCA
5  # 简单化处理，把对角线置为均值
6  row,col = np.diag_indices_from(Y)
7  Y[row,col] = np.mean(Y,axis = 0)
8  Y[0]
```

```
array([ 4.37977469e-17, -3.02279187e-02,  3.40340726e-03, ...,
       -2.06387864e-02,  4.35564673e-02, -4.03041313e-02])
```

```
1  %%time
2  cov = Y * Y.T
3  W,V = np.linalg.eig(cov)
```

```
Wall time: 1min 3s
```

```
1  W[:50]
```

```
array([18.94189217,  7.73621972,  6.16587462,  5.7410379 ,  5.42087
92 ,
        4.54280448,  3.94646283,  3.405892  ,  3.16000759,  2.75831
```

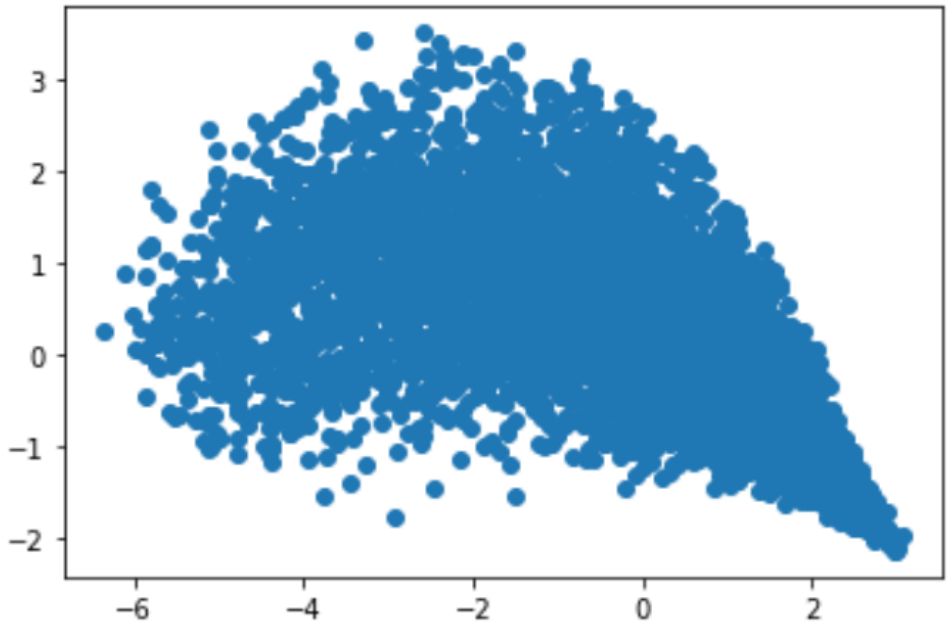
```python
# 把特征降到2维
base = V[:,:2].copy()
#base.shape
Y_2 = Y @ base
Y_2.shape
```

```
(6040, 2)
```

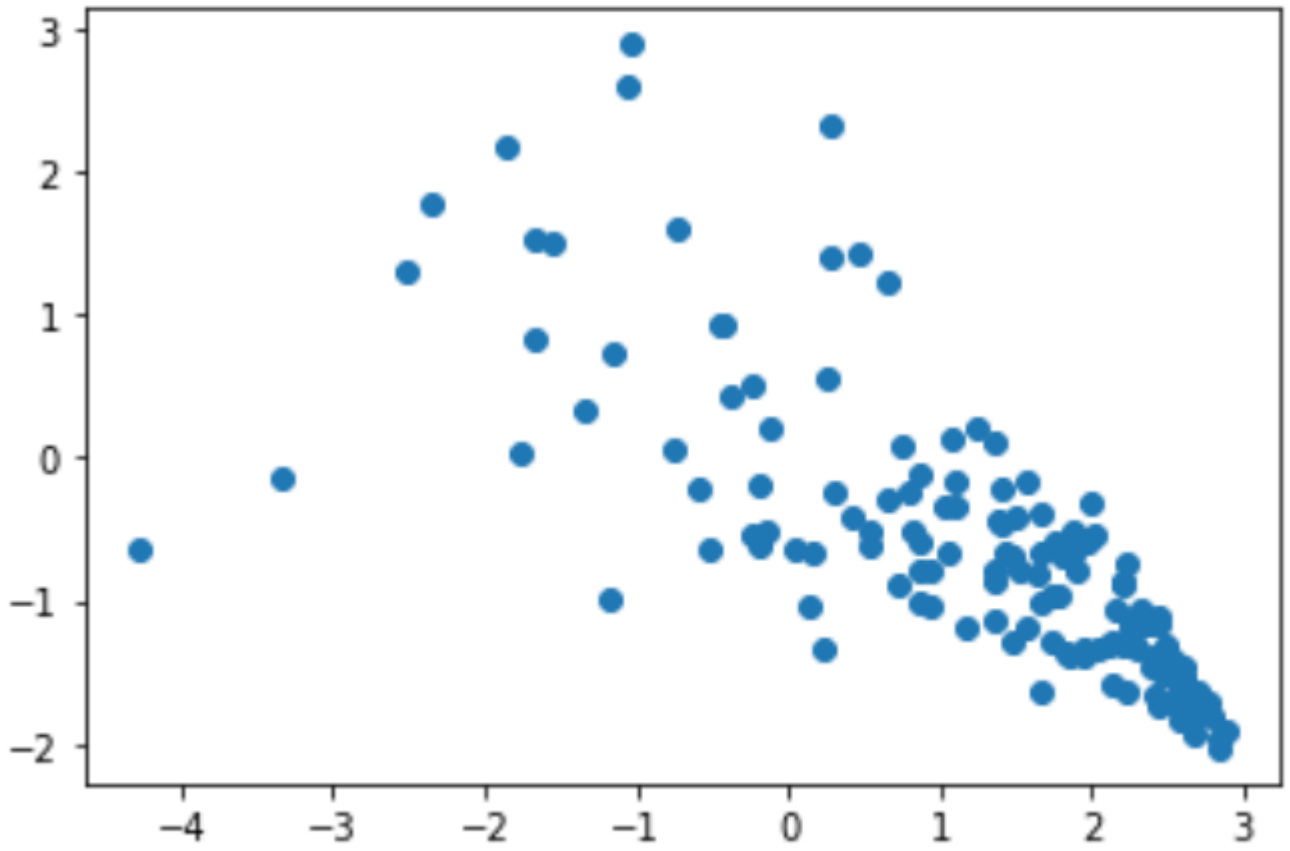
```python
Y_2[:,1]
```

```
array([-0.6094386 ,  0.78913409,  0.18056299, ..., -1.59870039,
       -0.7038668 ,  0.33787549])
```

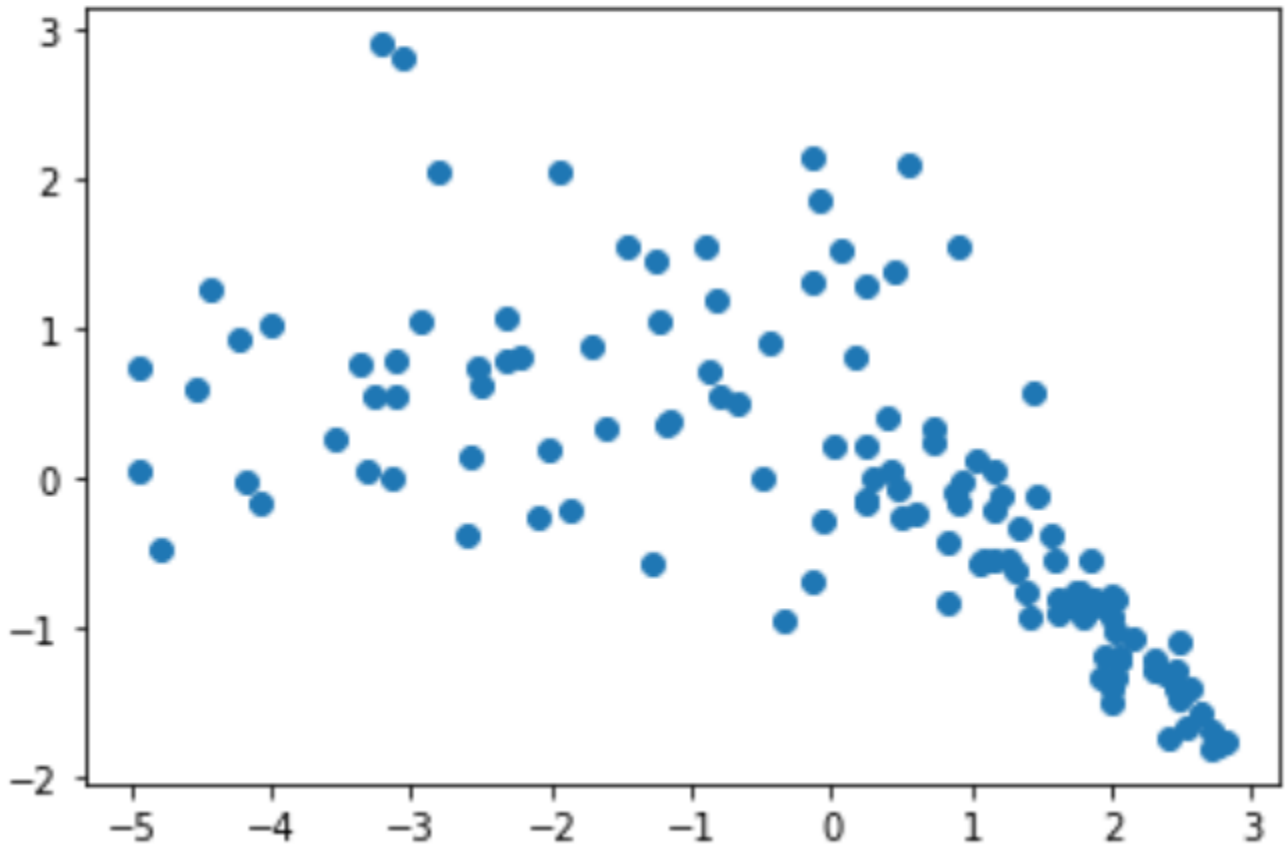
```python
import matplotlib.pyplot as plt
plt.scatter(Y_2[:,0],Y_2[:,1])
plt.show()
```



```
retired_distri = users[users['occ_desc'] == 'retired']
indices_ret = users['occ_desc'] == 'retired'
indices = indices_ret.to_numpy()
#indices
retired = Y_2[indices]
plt.scatter(retired[:,0],retired[:,1])
plt.show()
```

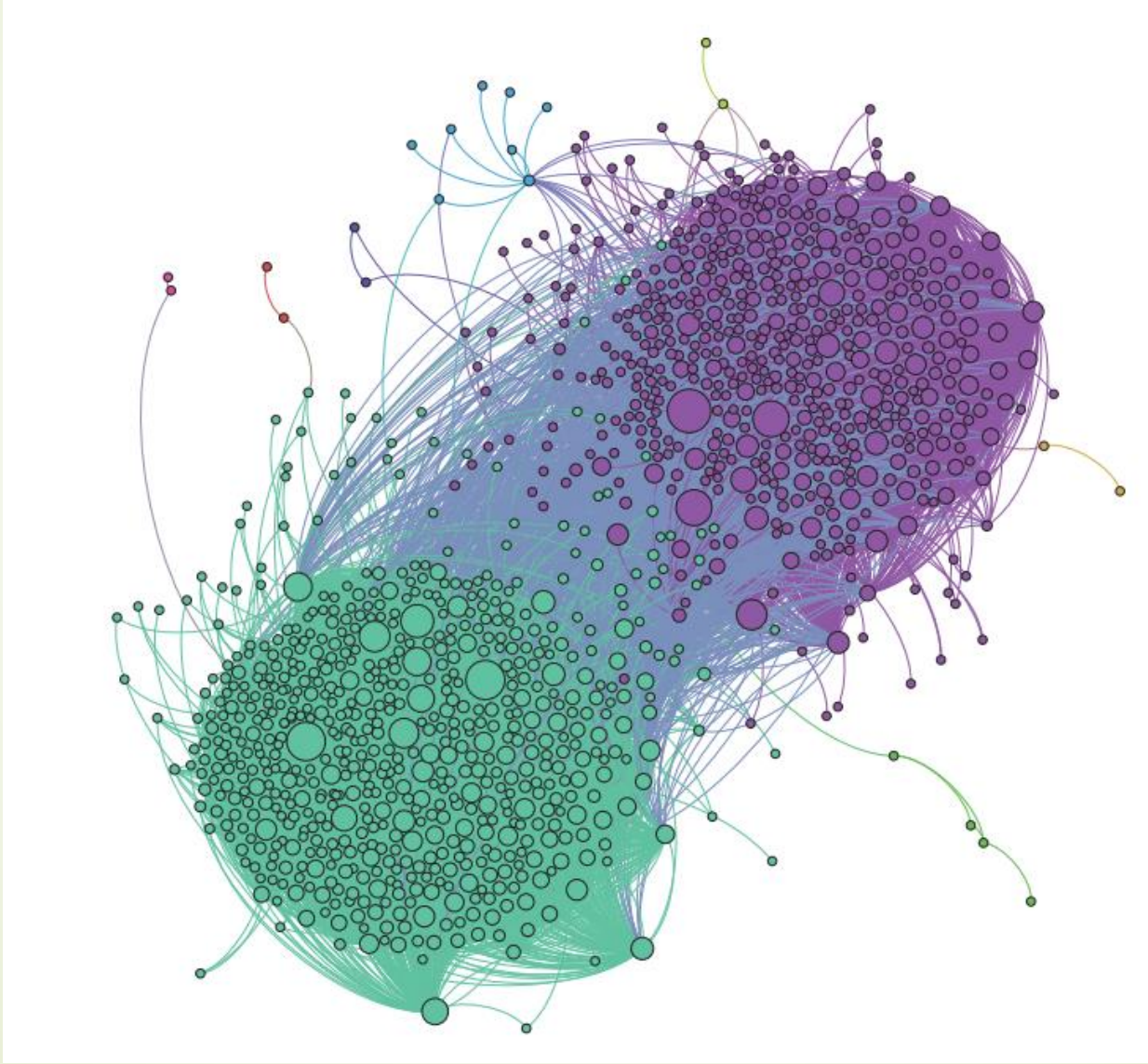


```
indices_get = users['occ_desc'] == 'lawyer'
indices = indices_get.to_numpy()
#indices
selected = Y_2[indices]
plt.scatter(selected[:,0],selected[:,1])
plt.show()
```



复杂网络与社区发现

社区发现



标签传播算法

- LPA[1]
  - 1. 给每个节点一个唯一的标签
  - 2. 对于每个节点，选择它邻居中出现最多的标签
  - 3. 重复第二步直到所有节点的标签不再变化
- 同步传播 & 异步传播
- 强社区(社区内每个节点连向社区内部的点比连向社区外部的点多)

异步算法流程

1）初始化网络中所有节点的标签，对于给定节点x，Cx(0)=x

2）设置 t=1

3）以随机顺序排列网络中的节点，并按排列序号将其设置为x

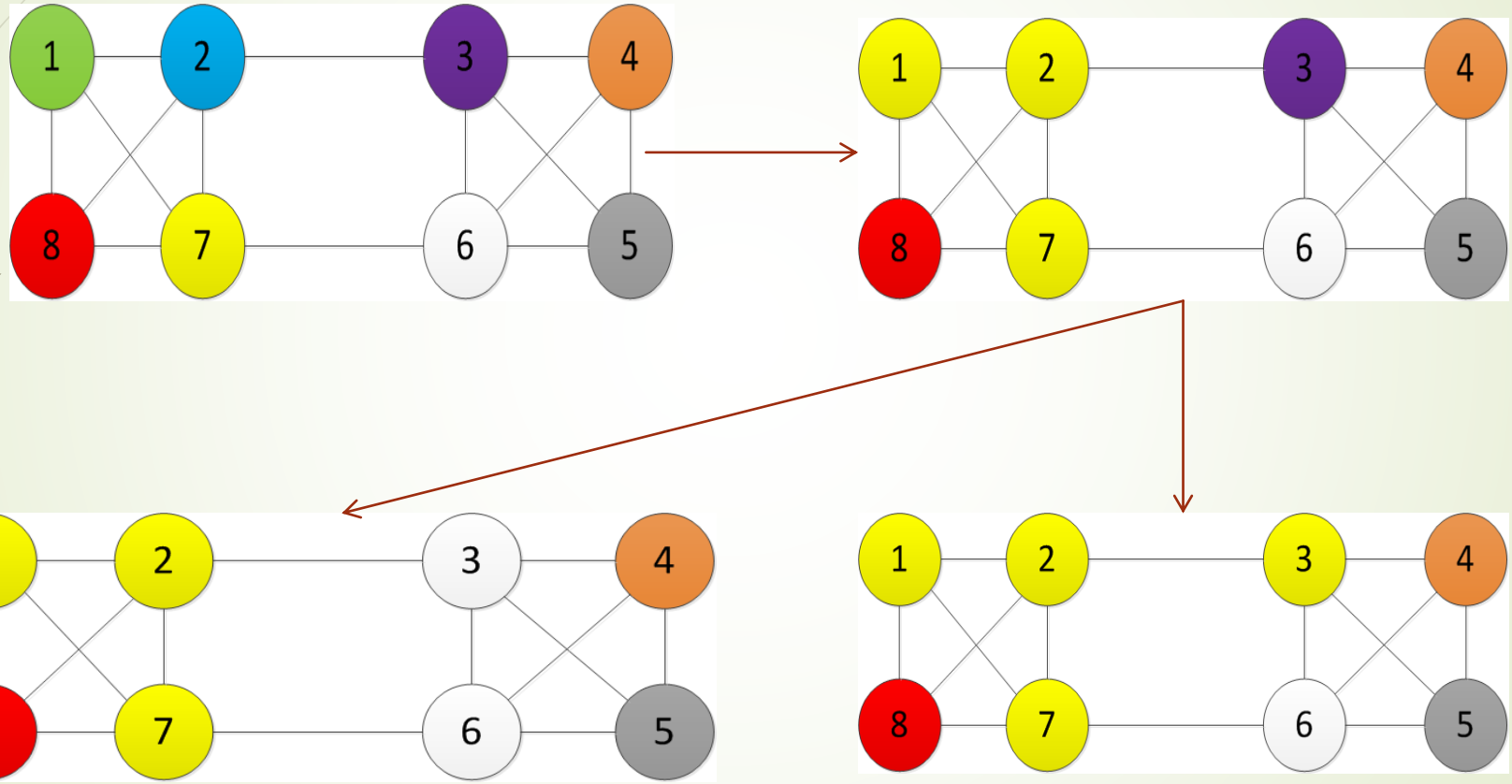
4）对于特定顺序选择的每个x∈X，让Cx(t)=f(Cxi1(t)，...，Cxim(t)，...。返回相邻标签中出现频率最高的标签。如果有多个最高频率的标签，就随机选择一个标签返回

5）如果每个节点都有其邻居节点中数量最多的标签，没有更新，则停止算法，否则，设置t=t+1并转到3

（计算过程结果不唯一且不保证收敛）

参考：https://blog.csdn.net/fengdu78/article/details/124811892

标签传播算法

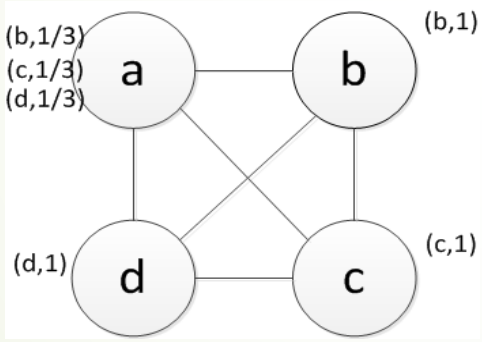


标签传播算法

- 重叠的社区发现：
- 允许一个节点同时属于多个社区。这种现象在真实网络中非常常见，比如社交网络中一个人可以有多个交际圈。

标签传播算法

- COPRA[2]
- 节点的标签为一个向量(多标签)
- 标签形式: $((c_1,b_1),(c_2,b_2),\ldots,(c_v,b_v))$



$$b_t(c,x) = \frac{\sum_{y \in N(x)} b_{t-1}(c,y)}{|N(x)|},$$

- 传递更多信息
- 重叠社区发现

reference

- LPA[1] : near linear time algorithm to detect community structures in large-scale networks
- COPRA[2] : finding overlapping communities in networks by label propagation
- LPAm[3]: detecting network communities by propagating labels under constraints
- BGLL[4]: fast unfolding of communities in large networks
- CM[5]: detect communities in networks by merging cliques
- MSG[6]: efficient modularity optimization by multistep greedy algorithm and vertex mover refinement
- LPAm+[7]: Advanced modularity-specialized label propagation algorithm for detecting communities in networks
- DPA[8]: unfolding communities in large complex networks: combining defensive and offensive label propagation for core extraction

Network science

- Newman:
- http://www-personal.umich.edu/~mejn/

考察贝叶斯公式的一个变形

$$P(spam|text) = \frac{P(text|spam)P(spam)}{P(text|spam)P(spam) + P(text|nonspam)P(nonspam)}$$

$$= \frac{1}{1 + \exp\{-(\alpha_1(text) - \alpha_0(text))\}}$$

其中$\alpha_1(text) = \log(P(text|spam)P(spam))$

以及$\alpha_0(text) = \log(P(text|nonspam)P(nonspam))$

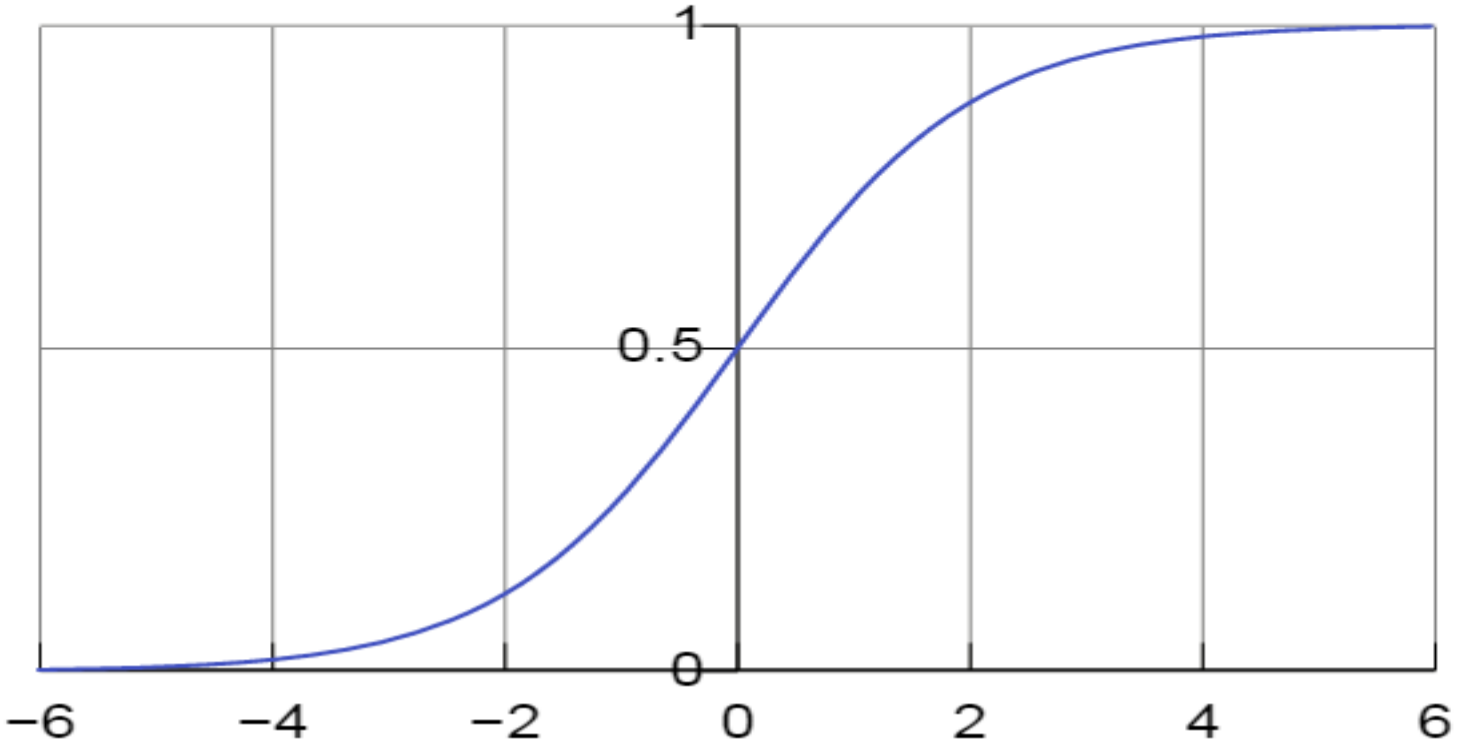
$$= \sigma(\alpha(x))$$

这里$\sigma(x) = \frac{1}{1+\exp(-x)}$, $\alpha(x) = \alpha_1(x) - \alpha_0(x)$

Sigmoid函数

$$g(x) = \frac{1}{1 + e^{-x}}$$

将线性回归值变换到$(0, 1)$，将其理解为x对应的y为1的概率



分类问题转化为网络参数回归问题

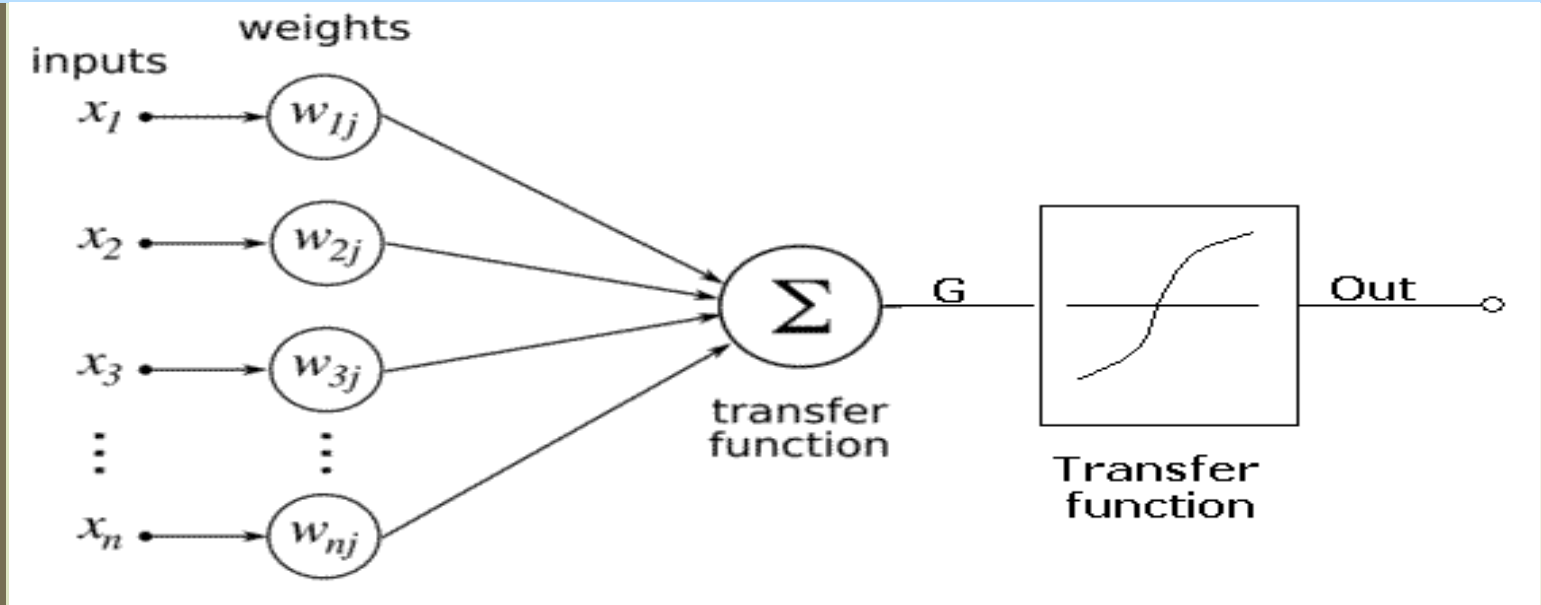
$$\alpha(text) = \log P(text|spam) + \log P(spam) - \log P(text|nonspam) - \log P(spam)$$

$$= x_1 \log P(w_1|spam) + \ldots + x_{|V|} \log P(w_{|V|}|spam) + \log P(spam)$$

$-\ldots$(对应的nonspam的项)

(x_i表示词典中第i个词在text中出现的次数)

$$= k_0 + x_1 k_1 + x_2 k_2 + \ldots x_{|V|} k_{|V|}$$



因此邮件分类问题转化为回归问题

$$\alpha(text) = \log P(text|spam) + \log P(spam) - \log P(text|nonspam) - \log P(nonspam)$$

$$= x_1 \log P(w_1|spam) + \ldots + x_{|V|} \log P(w_{|V|}|spam) + \log P(spam)$$

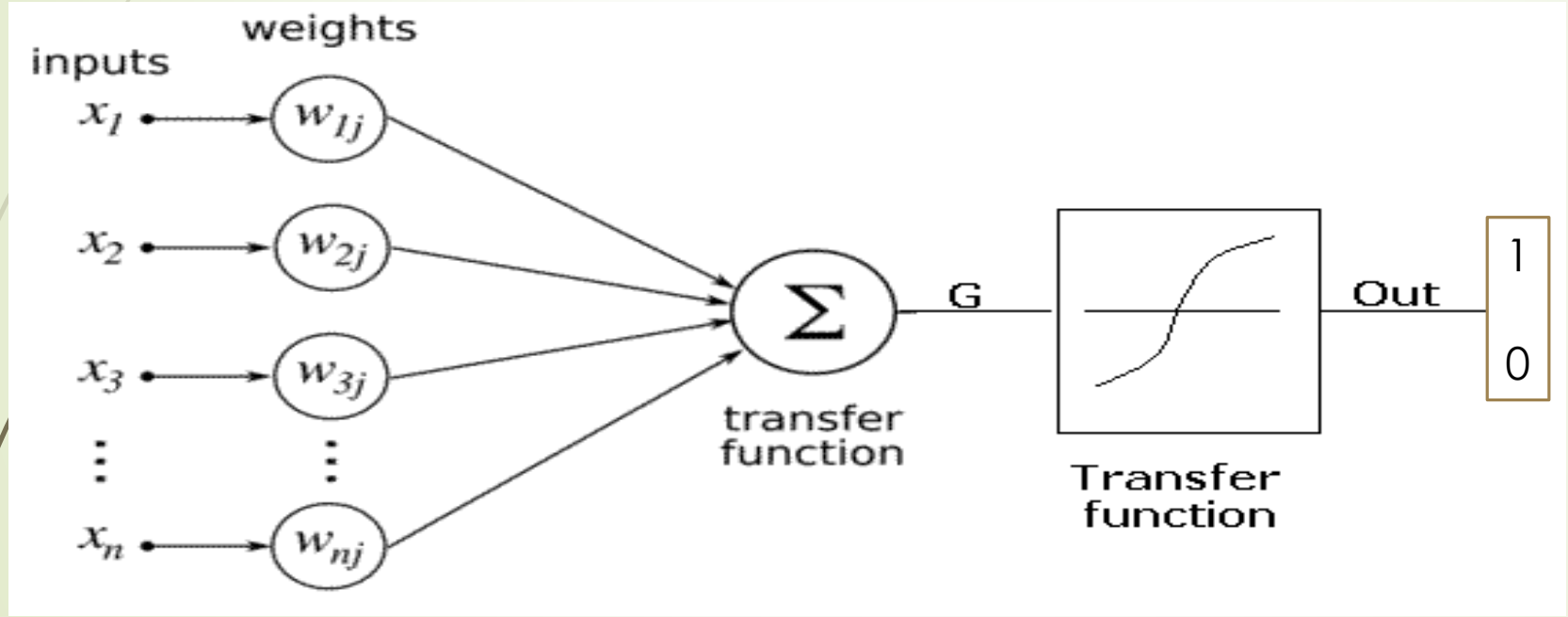
$-\ldots$(对应的nonspam的项)

(x_i表示词典中第i个词在text中出现的次数)

$$= k_0 + x_1 k_1 + x_2 k_2 + \ldots x_{|V|} k_{|V|}$$

找一组**K**，最大化**α**

单元神经元网络模型



最小化loss，最大似然