文献精读 2022-04-29

文献信息

一共涉及三篇文献，分别如下

文章1

标题： TGMI: an efficient algorithm for identifying pathway regulators through evaluation of triple-gene mutual interaction

DOI(url): 10.1093/nar/gky210

发表日期： 20 March 2018

发表杂志： *Nucleic Acids Research*

关键词： #Network #TF

文章2

标题： Hierarchical transcription factor and regulatory network for drought response in *Betula platyphylla*

DOI(url): 10.1093/hr/uhac040

发表日期： 19 February 2022

发表杂志： *Horticulture Research*

关键词： #Network #TF

文章3

标题： Construction of a hierarchical gene regulatory network centered around a transcription factor

DOI(url): 10.1093/bib/bbx152

发表日期： 27 November 2017

发表杂志： *Briefings in Bioinformatics*

关键词： #Network #TF

概述

当时是先读的文章2，发现里面的基因调控网络(Gene Regulation Network, GRN)构建起来不是很难的样子，就算一些偏相关系数嘛，所以想拿自己的数据试一下，结果中间发现了一些问题(见文献笔记)，然后就写邮件问了一作，一作又推荐了文章3，说是按照这篇文章做的。其实文章3我之前读过，是卫海荣老师写的综述，卫海荣老师专攻GRN。之前读的很粗，主要原因是手上没有ChIP-Seq的数据，感觉复现

不了，就忽略了。但是文章2作者他们只用RNA-seq的数据也做了，那说明还是值得一读的。读文章3的时候发现卫海荣老师2019年发的文章1又提出了一种新的评估pathway regulator的有效算法。但只提出了算法和一个R脚本，没打成包，依赖又多，写的说实话是乱七八糟，其中还有个doMC包只能Unix平台的系统运行，意思是windows跑不了，再考虑到R的运行速度太慢（R的并行懂的都懂），因此精读改写一个python的脚本。地址https://github.com/Hua-CM/TGMI。总的来说读完这三篇文章，对GRN的构建基本上有了一些大概的框架。

文献笔记

Gene Regulation Network (GRN)

读完三篇文章，我认为构建GRN其实无非就是两件事：

- 使用已有数据评估三个基因(triple gene interference, TGI)之间的信息。
- 自底向上(bottom-up, BU)构建或自顶向下(top-down, TD)构建。

对于第一点，衡量TGI信息的方法包括偏相关系数、BWERF、TGMI等，综合考虑样本量和数据类型选择；对于第二点，我个人理解的是，有明确的目标转录因子时，使用自顶向下构建，有明确的目标通路时(比如干旱相关基因)，自底向上构建。

一般GRN包括三层：layer1是转录因子，layer2也是转录因子，layer3是目标基因。例如文章2中的图。

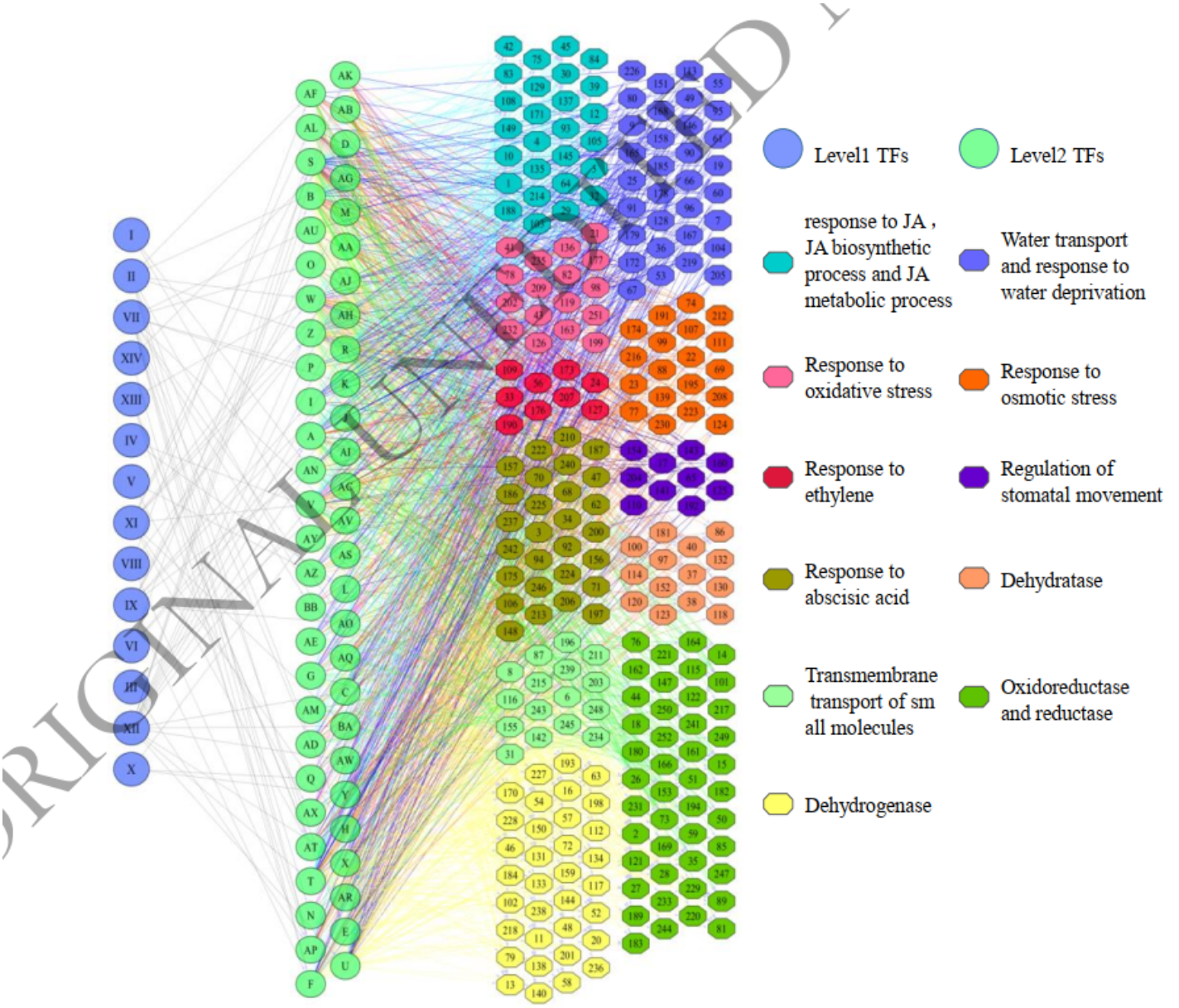

Figure 2. The gene regulatory network (GRN) of birch responding to PEG-induced

文章2中的构建方法如下：

The GRN was then constructed using the prefiltered DEGs from the bottom up. We divided the DEGs into two groups: (1) regulatory genes such as TF encoding genes, and (2) structural genes such as enzyme encoding genes. The structural genes were used as a bottom layer during the GRN construction. Since the expression of genes in the same biological process might be regulated by the same regulator (TF)20 we calculated correlation coefficients (CC) for each pair of the bottom genes. Co-expressed gene pairs (A and B) with CCs > 0.8 were considered as regulated by the same TF. Next, we calculated partial correlation coefficients (PCCs) for these co-expressed gene pairs by introducing each TF. If TF C significantly destroyed the co-expression relationship between A and B (PCC < 0.3), in other words, without the interference of C, A and B is not co-expressed, we assume that TF C regulates A and B simultaneously. Using such an algorithm, we identified a layer of TFs (level 2 TFs in Figure 2) that may directly regulate the bottom genes (Figure 2). We then used the layer 2 TFs as a new bottom layer to identify the top layer of TFs (layer 1 TFs in Figure 2). Finally, a GRN containing three layers was constructed

我之前没弄明白的点是：在构建Layer2和Layer3的时候，Layer2全是TFs，Layer3全是protein coding genes，所以没有重叠的问题。但是，对于Layer1和Layer2之间的网络，因为全部是TFs，在构建的时候结果中不可避免的会出现TF1同时调控TF2和TF3，同时发现TF2同时调控TF1和TF4的这样的情况，进而无法推断到底谁在Layer1，谁在Layer2。后来通过阅读文章3，我明白了他们的操作方法，他们首先按TFs在layer2-layer3的TGI中出现的频率挑选Layer2，然后所有在Layer2没有被选上的TF，当作Layer1的输入。

原文：$G_z^i(k) < -sig(G_Z^I)$ # keep most significant k genes to current

明白了这个，其实构建GRN就没有什么问题了。

TGMI构建步骤

python版TGMI地址:https://github.com/Hua-CM/TGMI

1. 对每个基因表达数据进行等频离散化（equal frequency discretization）
the gene expression data for each gene were discretized by using the equal frequency discretization algorithm

- 边界区重复值的处理方法上我和卫海荣老师不太一样。卫海荣用的`infotheo包`的`discretize`函数，更倾向于把重复值放到较小的区间，我写的函数更倾向于放到较大的区间，理由是编秩的时候重复值使用平均值编秩的话，离散化的结果应该是我这种方法才对。不过实际数据中只要样本数不是过少、重复的值不是过多，那么这个处理问题就不是问题（因为边界区不会有那么多重复值）。
- 实际上这步决定了TGMI算法根本不适用于小样本的情况，因为其进行等频离散化的函数的默认分bin的数量为$\sqrt[3]{n}$, 向下取整。 n=10时，只能分成两个bin进行离散化，n>27时才能分成3个bin进行离散化。我个人认为分成两个bins计算MIM是无意义的。那么至少要多少个样本才能进行TGMI算法呢？文章中作者使用了一个包含128个样本的microarray数据集，这时的bins=5，基于此我认为当样本数>125个时使用这个算法应当是没有问题的。当样本数小于125个时可以尝试手动指定bins=5。对于过小的样本数，就不建议使用TGMI算法了。小样本请参考文章2， 使用偏相关系数的方法。

2. The entropy (H) of each variable is then calculated using the following formula:对每个基因使用如下公式计算熵[1]

$$H(X) = -\Sigma p(x) log(p(x)) = -E[log(p(x))]$$

注: (1) p代表概率质量函数probability mass function，使用自然对数e作为底。
(2) 熵是对一个随机变量不确定度的度量(which is a measure of the uncertainty of a random variable [1])

3. 计算条件熵conditional entropy，联合熵joint entropy的公式如下

$$\begin{aligned} H(X;Y) &= -\Sigma p(x,y) log(p(x,y)) \\ &= -E[log(p(x,y))] \\ H(X|Y) &= H(X,Y) - H(Y) \end{aligned}$$

注意和贝叶斯公式不一样

4. 互信息(mutual information)定义如下

$$
\begin{aligned}
I(X;Y) &= H(X) - H(X|Y) \\
&= H(Y) - H(Y|X) \\
&= H(X) + H(Y) - H(X,Y)
\end{aligned}
$$

对于X,Y在Z出现时的条件互信息，定义如下：条件Z时, 由于Y引起的变量X的不确定性的下降程度。

`We now define the conditional mutual information as the reduction in the uncertainty of X due to knowledge of Y when Z is given.`

$$
I(X;Y|Z) = H(X|Z) - H(X|Y,Z)
$$

5. 对于两个基因Y1、Y2和一个转录因子X，做如下定义：
a.图中的S1、S2、S3定义为：在另外两个基因(变量)给定条件下，第三个基因(变量)的条件熵。
the conditional entropies of each variable given the other two variables.
因此对于S1，S2，S3有如下公式

$$
\begin{aligned}
S1 &= H(Y1|X,Y2) \\
S2 &= H(Y2|X,Y1) \\
S3 &= H(X|Y1,Y2)
\end{aligned}
$$

b.图中的S4、S4、S6定义为:在第三个基因给定的条件下，另两个基因的互信息。
因此对于S3，S5，S6有如下公式

$$
\begin{aligned}
S4 &= I(X;Y1|Y2) \\
S5 &= I(X;Y2|Y1) \\
S6 &= I(Y1;Y2|X)
\end{aligned}
$$

c.对于S7定义为：三个基因的互信息，有以下公式：

$$
S7 = I(Y1;Y2;X) = I(Y1;Y2) - I(Y1;Y2|X)
$$

定义 mutual interaction measure(MIM)如下：

$$
\begin{aligned}
MIM &= \frac{S7}{S1+S2+S3} \\
&= \frac{I(Y1;Y2;X)}{H(Y1|X,Y2)+H(Y2|X,Y1)+H(X|Y1,Y2)} \\
&= \frac{H(Y1)+H(Y2)+H(X)-H(X,Y1)-H(X,Y2)-H(Y1,Y2)+H(Y1,}{3H(Y1,Y2,X)-H(Y1,Y2)-H(X,Y1)-H(X,Y2)}
\end{aligned}
$$

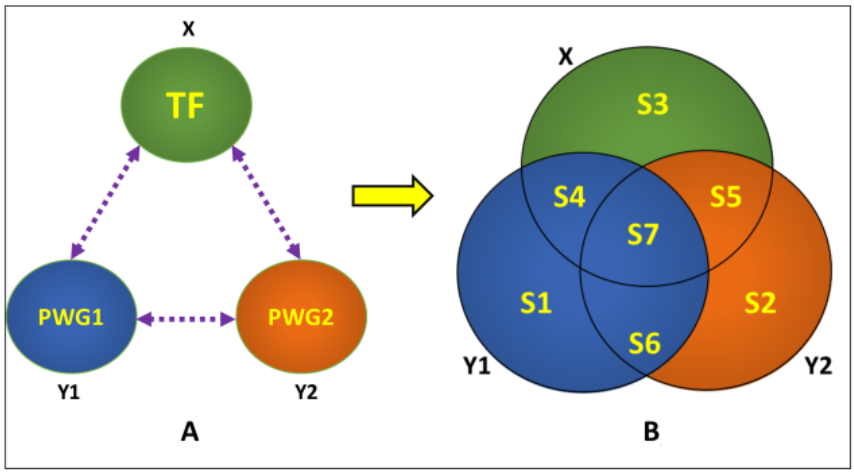

Figure 1. Dissection of interactive components for a given triple gene block. A. Pathway Gene 1 (PWG1) and 2 (PWG2), and the TF are represented by Y1, Y2 and X, respectively. B. Dissected components of interaction among triple genes.

注：注意定义与推论的区别！图中的S1-S7都是定义出来的，计算方法才是根据定义推出来的。

6. 如果MIM小于0的话则说明该三基因对无意义，舍去；如果MIM大于0，则进入步骤7计算P值。
7. 使用置换法(permutation method)计算每个三基因模块(triple gene block)的MIM是否具有显著性(P<0.01).

- 首先从数据集中随机抽选1000个TF-gene1-gene2模块生成随机数据集
  注意原作者代码中抽样的`sample`函数。在R中sample函数未指定抽样数时，实际上是起到了一种重排序的作用，由此产生*随机数据集*。因此实际上作者所说的**抽选生成随机数据集**就**不完全是抽选**了，还包括了**随机排序**

```R
> sample(seq(1,10))
 [1]  9  4  3  2  6  7 10  5  1  8
> sample(seq(1,10))
 [1]  5  8  6  9  2  7 10  4  3  1
> sample(seq(1,10))
 [1]  1  4  2  6  5 10  3  7  8  9
```

- 计算随机数据集的MIM值
- 基于随机数据集的MIM计算P值
- 使用FDR法对P值进行校正
  原作者的代码如下：

```R
randomized_3GI_2PW_1TF<-c()
i1 <- floor(runif(1,1,dim(pathway.data)[2]))
j1 <- floor(runif(1,1,dim(pathway.data)[2]))
k1 <- floor(runif(1,1,dim(tf.data)[2]))
for(i in 1:1000){
  randomized_3GI_2PW_1TF<-c(randomized_3GI_2PW_1TF,mi3(sample(pathway.data[,i1])
}
# I3 was the MIM value
z_score<-(I3-mean(randomized_3GI_2PW_1TF))/sd(randomized_3GI_2PW_1TF)
```

```
  #pnorm(z_score,lower.tail=FALSE)
I3_pval <- pnorm(z_score,lower.tail=FALSE)
```

性能调优

注： 性能调优的时候对permutation理解有误，对$X, Y1, Y2$同时进行了permutation，正式代码中已经改为了只对X进行permutation。不影响性能调优的分析过程。

因为第一次计算一个三基因模块的MIM的p-value的时候发现要耗时1.22s，按这个时间计算的话，以指定通路有50个基因计算，计算一个TF与这个通路中两两基因的关系需要20分钟左右，即使使用40核的CPU，一小时也只能计算120个TF的关系。这速度太慢了，因此必须想办法进行性能调优，即使能把性能提升20%也是巨大进步。具体步骤如下：
初始代码：

PYTHON

```python
def triple_interaction(xvector
    n_sample = len(xvector)
    mim = MIM(discretize(xvector)
    # generate randomsized dataset
    randomzied_dataset = []
    for i in range(permutation):
        tmp_x = discretize(np.random.choice(xvector, size=n_sample, replace=Fals
        tmp_y1 = discretize(np.random.choice(y1vector, size=n_sample, replace=Fa
        tmp_y2 = discretize(np.random.choice(y2vector, size=n_sample, replace=Fa
        randomzied_dataset.append(MIM(tmp_x, tmp_y1, tmp_y2))
    # calculate P-value
    if np.std(randomzied_dataset) == 0:
        z_score = 0
    else:
        z_score = (mim - np.mean(randomzied_dataset)) / np.std(randomzied_datase
    p_val = 1 - norm.cdf(z_score)
    return p_val
```

测试结果：

```
Total time: 1.16917 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    84                                           def triple_interaction(xvector,
    92         1         26.0     26.0      0.0      n_sample = len(xvector)
    93         1      24311.0  24311.0      0.1      mim = MIM(discretize(xvecto
    94                                               # generate randomsized data
    95         1          8.0      8.0      0.0      randomzied_dataset = []
    96      1001      10061.0     10.1      0.0      for i in range(permutation)
    97      1000    3893113.0   3893.1     16.7          tmp_x = discretize(np.r
```

98	1000	3796558.0	3796.6	16.3	tmp_y1 = discretize(np.
99	1000	3883792.0	3883.8	16.7	tmp_y2 = discretize(np.
100	1000	11710315.0	11710.3	50.2	randomzied_dataset.appe
101					# calculate P-value
102	1	1422.0	1422.0	0.0	if np.std(randomzied_datase
103	1	7.0	7.0	0.0	z_score = 0
104					else:
105					z_score = (mim - np.mea
106	1	2054.0	2054.0	0.0	p_val = 1 - norm.cdf(z_scor
107	1	7.0	7.0	0.0	return p_val

很明显时间全耗在了抽样和计算MIM两个环节上，考虑到这边我是先抽样，再离散化的（就是1000次permutation，每次都先抽样再离散化），我就考虑是否是离散化耗时过长，当时这么做的原因主要是担心先离散化再抽样可能对离散化的结果有影响。这边仔细想了下，先离散化再抽样应该是不影响的，所以改成先离散化，再抽样(直接给triple_interaction传入离散化后的数据)。然后这边就重写了一下，重新测试发现影响速度的关键环节不在离散化（测试略）。所以我接着想优化MIM。
MIM代码如下：

PYTHON

```python
def MIM(xvector, y1vector, y2vector):
    """
    :param xvector: TF's expression discretized value    :param y1vector: gene1'
    s7 = entropy(xvector) + entropy(y1vector) + entropy(y2vector) + jentropy(xve
    s456 = 3 * jentropy(xvector, y1vector, y2vector) - jentropy(xvector, y1vecto
    mim = s7 / s456
    if mim < 0:
        mim = 0
    return mim
```

非常明显的，其中有几处重复计算的地方，因此直接改写成如下代码：

PYTHON

```python
def MIM(xvector, y1vector, y2vector):
    """
    :param xvector: TF's expression discretized value    :param y1vector: gene1'
    je1 = jentropy(xvector, y1vector, y2vector)
    je2 = jentropy(xvector, y1vector)
    je3 = jentropy(xvector, y2vector)
    je4 = jentropy(y1vector, y2vector)
    s7 = entropy(xvector) + entropy(y1vector) + entropy(y2vector) + je1 - je2 -
    s456 = 3 * je1 - je2 - je3 - je4
    mim = s7 / s456
    if mim < 0:
        mim = 0
    return mim
```

测试结果：

```
Timer unit: 1e-07 s
Total time: 0.0010544 s
File: <ipython-input-80-a28857f7c136>
Function: MIM at line 1
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def MIM(xvector, y1vector,
y2vector):
     2                                               # accelerate
     3         1       3088.0   3088.0     29.3      je1 = jentropy(xvector,
y1vector, y2vector)
     4         1       1625.0   1625.0     15.4      je2 = jentropy(xvector,
y1vector)
     5         1       1457.0   1457.0     13.8      je3 = jentropy(xvector,
y2vector)
     6         1       1382.0   1382.0     13.1      je4 = jentropy(y1vector,
y2vector)
     7         1       2945.0   2945.0     27.9      s7 = entropy(xvector) +
entropy(y1vector) + entropy(y2vector) + je1 - je2 - je3 - je4
     8         1         24.0     24.0      0.2      s456 = 3 * je1 - je2 - je3 -
je4
     9         1          7.0      7.0      0.1      mim = s7 / s456
    10         1         11.0     11.0      0.1      if mim < 0:
    11         1          3.0      3.0      0.0          mim = 0
    12         1          2.0      2.0      0.0      return mim
```

这个结果比较符合预期，证明主要耗时是在计算上了，因此进一步考虑如何提高计算速度。这边我从两个角度考虑

1. 首先是考虑`entropy(xvector)`，`entropy(y1vector)` 和 `entropy(y2vector)` 三个值。因为`MIM`是要计算1000次的，然而permutation只影响$P(X,Y1)$, $P(X,Y2)$, $P(Y1,Y2)$, 并不影响$X, Y1, Y2$， 所以其实$X, Y1, Y2$ 的熵只要算一次就行了，所以我考虑直接搞一个permutation特供版`MIM`函数，把这三个熵的值直接传进来
2. 直接优化`entropy`函数加快速度。entropy函数非常简单，所以可以优化的地方非常有限
原代码如下:

PYTHON

```python
def entropy(vector):
    """
    Calculate entropy use discretized values
    :param vector: discretized values
    :return: entropy
    """
```

```
    site, count = np.unique(vector, return_counts=True)
    probs = count / len(vector)
    return np.sum((-1) * probs * np.log(probs))
```

直觉告诉我最有的地方可能就是求概率这步，经过google，发现用numpy的另一个函数会更快一点，经过测试确实如此[2],快了10倍。。。。。。
测试结果：

```
In [86]: %timeit np.unique(gene1, return_counts=True)
14.7 µs ± 241 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
In [87]: %timeit np.bincount(gene1)
846 ns ± 5.62 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

但是因为`np.bincount`只能对整数进行计数，而$H(X,Y1)$这种的组合之前是用数组表示的(1,2)，这样的话使用bincount会报错，因此需要对进行jentropy也进行一些处理，我的方案是把(1,2)变成12，(2,1)变成21，如果是三个数组，则将变成(1,2,3)变成123，以此类推
所以改写如下

PYTHON

```python
def entropy(vector):
    """
    Calculate entropy use discretized values
    :param vector:
    :return:
    """
    count = np.bincount(vector)
    count = count[count != 0]
    probs = count / len(vector)
    return np.sum((-1) * probs * np.log(probs))

def jentropy(*vectors):
    power = 0
    join_vector = np.zeros(len(vectors[0]), dtype=np.int32)
    for vector in vectors:
        join_vector += vector * pow(10, power)
        power += 1
    return entropy(join_vector)
```

然后再改写一个特供版的`MIM`,并且对应着改写下`triple_interaction`

PYTHON

```python
def MIM_permutation(xvector, y1vector, y2vector, ex, ey1, ey2):
    # accelerate
    je1 = jentropy(xvector, y1vector, y2vector)
    je2 = jentropy(xvector, y1vector)
```

```
    je3 = jentropy(xvector, y2vector)
    je4 = jentropy(y1vector, y2vector)
    s7 = ex + ey1 + ey2 + je1 - je2 - je3 - je4
    s456 = 3 * je1 - je2 - je3 - je4
    mim = s7 / s456
    if mim < 0:
        mim = 0
    return mim


def triple_interaction(xvector, y1vector, y2vector, permutation=1000):
    """
    :param xvector: TF's expression discretized value
    :param y1vector: gene1's expression discretized value
    :param y2vector: gene2's expression discretized value
    :param permutation: permutation numbers
    :return: MIM for TF-gene1-gene2
    """
    n_sample = len(xvector)
    mim = MIM(xvector, y1vector, y2vector)
    if mim == 0:
        return 2 # return an impossible p-value for filter
    ex = entropy(xvector)
    ey1 = entropy(y1vector)
    ey2 = entropy(y2vector)
    # generate randomsized dataset
    randomzied_dataset = []
    for i in range(permutation):
        tmp_x = np.random.choice(xvector, size=n_sample, replace=False)
        tmp_y1 = np.random.choice(y1vector, size=n_sample, replace=False)
        tmp_y2 = np.random.choice(y2vector, size=n_sample, replace=False)
        randomzied_dataset.append(MIM_permutation(tmp_x, tmp_y1, tmp_y2, ex, ey1
    # calculate P-value
    if np.std(randomzied_dataset) == 0:
        z_score = 0
    else:
        z_score = (mim - np.mean(randomzied_dataset)) / np.std(randomzied_datase
    p_val = 1 - norm.cdf(z_score)
    return p_val
```

再测试下速度

```
Timer unit: 1e-07 s
Total time: 0.227071 s
File: <ipython-input-99-83c10dc75c34>
```

```
Function: triple_interaction at line 15
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    15                                           def triple_interaction(xvector,
    16                                               """
    17                                               :param xvector: TF's expres
    18                                               :param y1vector: gene1's ex
    19                                               :param y2vector: gene2's ex
    20                                               :param permutation: permuta
    21                                               :return: MIM for TF-gene1-g
    22                                               """
    23         1         19.0     19.0      0.0      n_sample = len(xvector)
    24         1       6431.0   6431.0      0.3      mim = MIM(xvector, y1vector
    25         1        157.0    157.0      0.0      ex = entropy(xvector)
    26         1        144.0    144.0      0.0      ey1 = entropy(y1vector)
    27         1        140.0    140.0      0.0      ey2 = entropy(y2vector)
    28                                               # generate randomsized data
    29         1          5.0      5.0      0.0      randomzied_dataset = []
    30      1001       7291.0      7.3      0.3      for i in range(permutation)
    31      1000     243370.0    243.4     10.7          tmp_x = np.random.choic
    32      1000     224222.0    224.2      9.9          tmp_y1 = np.random.choi
    33      1000     215888.0    215.9      9.5          tmp_y2 = np.random.choi
    34      1000    1564534.0   1564.5     68.9          randomzied_dataset.appe
    35                                               # calculate P-value
    36         1       2482.0   2482.0      0.1      if np.std(randomzied_datase
    37                                                   z_score = 0
    38                                               else:
    39         1       1940.0   1940.0      0.1          z_score = (mim - np.mea
    40         1       4084.0   4084.0      0.2      p_val = 1 - norm.cdf(z_scor
    41         1          7.0      7.0      0.0      return p_val
```

提速5倍左右，成功。

其他

1. 写完我算是知道为啥只用这个方法去推断TF和**指定通路基因**的网络关系了，因为这个方法的计算实在是太慢了。卫海荣老师的代码并没有像我一样进行优化，搞个什么特供版的`MIM`出来。他就是按正常逻辑写的，再加上他用的是R写的，只会比我更慢（即使大家底层都是C++）。
2. Pathos的使用。因为众所周知python自带的`multiprocessing`是基于`pickle`的，所以我更喜欢使用基于`dill`的`pathos`进行多进程，然而Pathos引入进程池的写法五花八门，这次写`TGMI`的过程中搜到了一个回答，系统的解释了进程池引入方法的区别。总结一下就是很多是历史遗留问题，以后用`from pathos.pools import ProcessPool`就行了。`ParallelPool`主要是为分布式机器的多节点准备的，单机的多进程没必要用这个(用这个要考虑别的问题)。

```python
from pathos.pools import ProcessPool as Pool0
from pathos.multiprocessing import Pool as Pool1
from pathos.pools import ParallelPool as Pool2
from pathos.parallel import ParallelPool as Pool3
```

名词解释

1. 概率公式中分号、逗号和竖线的含义
   a. 分号：$P(X;\theta)$ x代表确定的采样值，θ代表待估参数。需要指出的是θ是一个确定的值，只是待估。所以$P(X;\theta)$实际上写成$P(X)$也是可以的。在这篇文献中，我感觉理解为$I(X1;X2)$ 需要两个参数更合理。
   b. 逗号：表示 A和B事件同时发生的概率，是联合概率分布
   c. 竖线：表示条件概率
   优先级： 分号>逗号>竖线

和我相关

1. 掌握了如何利用permutation法计算P值
2. 对概率论的理解加深了一些
3. 掌握了TGMI方法
4. 根系分泌物对微生物的调控是否也可以使用这种方法进行推断呢？因为数据反正都是先进行了离散化的。

参考文献

[1] Cover, T.M. and Thomas, J.A. (2005). Entropy, Relative Entropy, and Mutual Information. In Elements of Information Theory (eds T.M. Cover and J.A. Thomas). https://doi.org/10.1002/047174882X.ch2
[2] https://stackoverflow.com/questions/10741346/numpy-most-efficient-frequency-counts-for-unique-values-in-an-array
[3] https://stackoverflow.com/questions/48990688/pathos-parallel-processing-options-could-someone-explain-the-differences