1 项目描述

1.1 加权排列熵

(1) 数学描述

(2) 分值：12 分

(3) 参考实现

```
import numpy as np
from math import factorial

def _embed(x, order=3, delay=1):
    N = len(x)
    Y = np.empty((order, N - (order - 1) * delay))
    for i in range(order):
        Y[i] = x[i * delay:i * delay + Y.shape[1]]
    return Y.T

def weighted_permutation_entropy(time_series, order=2, delay=1, normalize=
    False):
    x = _embed(time_series, order=order, delay=delay)
    weights = np.var(x, axis=1)
    sorted_idx = x.argsort(kind='quicksort', axis=1)
    motifs, c = np.unique(sorted_idx, return_counts=True, axis=0)
    pw = np.zeros(len(motifs))

    for i, j in zip(weights, sorted_idx):
        idx = int(np.where((j == motifs).sum(1) == order)[0])
        pw[idx] += i

    pw /= weights.sum()
    b = np.log2(pw)
    wpe = -np.dot(pw, b)
    if normalize:
        wpe /= np.log2(factorial(order))
    return wpe
```

(4) 测试用例

```
x = [4, 7, 9, 10, 6, 11, 3]
print(permutation_entropy(x, order=3, normalize=True))
结果：0.546995039859119
```

(5) Reference:https://github.com/nikdon/pyEntropy

2 项目报告

2.1 算法描述

排列熵 (permutation entropy,PE)，一种检测动力学突变和时间序列随机性的方法，能够定量评估信号序列中含有的随机噪声。排列熵无需考虑时间序列的数值大小, 而是对相邻样本点进行对比分析, 获取相应特征信息, 相较其他值方法更能捕获序列的微弱变化, 并且该算法具有理论简单、抗噪能力强等优势, 故在故障诊断领域应用较为广泛。

但 PE 算法仅利用时间序列的序数结构, 忽视其幅值信息, 因此,XIA 等在 PE 的基础上提出了加权排列熵 (weighted permutation entropy, WPE)，既考虑单一尺度上时间序列的复杂性也注重动力学突变。

以下为数学概念上具体计算步骤：

(1) 对时间序列 $X = (x_1, x_2, ..., x_N)$ 进行相空间重构, 得到一系列子序列 $X_i^{(M)} = \left(x_i, x_{i+T}, ..., x_{i+(M-1)T}\right)$，也即矩阵

$$Y = \begin{bmatrix} x_1 & x_{1+T} & \cdots & x_{1+(M-1)T} \\ x_2 & x_{2+T} & \cdots & x_{2+(M-1)T} \\ x_j & x_{j+T} & \cdots & x_{j+(M-1)T} \\ \cdots & \cdots & \cdots & \cdots \\ x_K & x_{K+T} & \cdots & x_{K+(M-1)T} \end{bmatrix}$$

式中 T 为时延，M 为嵌入维数，K=N-(M-1)T，矩阵 Y 中的每一行都是一个重构分量，共有 K 个重构分量。

(2) 计算出每个子序列 $X_i^{(M)}$ 的权重值 w_i

$$w_i = \frac{1}{M} \sum_{l=1}^{M} \left(x_{i+(l-1)T} - X_l^{(M)} \right)^2$$

$$X_l^{(M)} = \frac{1}{M} \sum_{l=1}^{M} x_{i+(l-1)T}$$

(3) 任意子序列 $X_i^{(M)}$ 的特征信息用权重值 w_i 和排列模式 π_q 表示。排列模式 π_q 是将每一个重构分量按照升序重新排列，得到向量中各元素位置的列索引构成的一组符号序列。对于该时间序列 X 共有 Q 种排列模式, 每种排列模式 π_q 的加权概率值为

$$P_w(\pi_q) = \frac{\sum \left\{ w_i \mid 1 \le i \le N - (M-1)T, i \in Z^+, N\left(X_i^M\right) \right\}}{\sum w_i}$$

式中 $N\left(X_i^M\right)$ 为 X_i^M 具有的排列模式 π_q

(4) 计算时间序列 X 的加权排列熵 WPE 值

$$WPE\left(X,M,T\right)=-\sum_{q=1}^{Q}P_w\left(\pi_q\right)\log_2 P_w\left(\pi_q\right)$$

(5) 归一化处理

$$WPE=\frac{WPE\left(X,M,T\right)}{\log_2 M!}$$

算法思路：用二维数组存放重构分量、排列模式，用一维数组存放其他变量，通过 5 个函数逐步完成计算。

(1) F1 相空间重构数组，得重构分量矩阵

(2) F2 计算出每个重构分量对应的权重值

(3) F3 归类每个重构分量对应的排列模式

(4) F4 计算每种排列模式 π_q 的加权概率值

(5) F5 计算时间序列 X 的加权排列熵 WPE 值

2.2 算法代码

Python 原型代码

```
import numpy as np
from math import factorial

def _embed(x, order=3, delay=1):
    N = len(x)
    Y = np.empty((order, N - (order - 1) * delay))
    for i in range(order):
        Y[i] = x[i * delay:i * delay + Y.shape[1]]
    return Y.T

def weighted_permutation_entropy(time_series, order=2, delay=1, normalize=False):
    x = _embed(time_series, order=order, delay=delay)
    weights = np.var(x, axis=1)
    sorted_idx = x.argsort(kind='quicksort', axis=1)
    motifs, c = np.unique(sorted_idx, return_counts=True, axis=0)
    pw = np.zeros(len(motifs))

    for i, j in zip(weights, sorted_idx):
        idx = int(np.where((j == motifs).sum(1) == order)[0])
        pw[idx] += i
```

```
    pw /= weights.sum()
    b = np.log2(pw)
    wpe = -np.dot(pw, b)
    if normalize:
        wpe /= np.log2(factorial(order))
    return wpe

x = [4, 7, 9, 10, 6, 11, 3]
print(weighted_permutation_entropy(x, order=3, normalize=True))
```

Cpp 实现代码

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

#define N 7 // 矩阵时间序列大小
#define M 3 // 嵌入维数
#define T 1 // 延迟时间
#define K N - (M - 1) * T // 重构分量个数

// F1 相空间重构数组，得重构分量矩阵
int function1(int X[N], int Y[K][M]);
// F2 计算出每个重构分量对应的权重值
int function2(int Y[K][M], double weight[K]);
// F3 归类每个重构分量对应的排列模式
int function3(int pattern[K][M], int Y[K][M], int links[K]);
// F4 计算每种排列模式πk的加权概率值
int function4(double poss[K], double weight[K], int links[K]);
// F5 计算时间序列X的加权排列熵WPE值
double function5(double poss[K]);

int main()
{
    int X[N] = {4, 7, 9, 10, 6, 11, 3};
    int links[K] = {0}, Y[K][M] = {0}, pattern[K][M] = {0};
    double result = 0, weight[K] = {0}, poss[K] = {0};

    function1(X, Y);
    function2(Y, weight);
    function3(pattern, Y, links);
    function4(poss, weight, links);
    result = function5(poss);
    cout << "PWE=" << setprecision(16) << result << endl;

    return 0;
}
// F1 相空间重构数组，得重构分量矩阵
```

```
int function1(int X[N], int Y[K][M]) // X[N]时间序列，Y[K][M]相空间重构分量的集合
{
   for (int i = 0; i < K; i++)
      for (int j = 0; j < M; j++)
         Y[i][j] = X[i + j * T];
   return 0;
}
// F2 计算出每个重构分量对应的权重值
int function2(int Y[K][M], double weight[K]) // Y[K][M]相空间重构分量的集合，weight[K]
    重构分量的对应权重
{
   // 每个重构分量的平均值，计算过程值
   double avg[K];
   for (int i = 0; i < K; i++)
   {
      avg[i] = 0;
      for (int j = 0; j < M; j++)
         avg[i] += Y[i][j];
      avg[i] /= M;
   }
   // 每个重构分量的权重值
   for (int i = 0; i < K; i++)
   {
      weight[i] = 0;
      for (int j = 0; j < M; j++)
         weight[i] += (Y[i][j] - avg[i]) * (Y[i][j] - avg[i]);
      weight[i] /= M;
   }
   return 0;
}
// F3 归类每个重构分量对应的排列模式
int function3(int pattern[K][M], int Y[K][M], int links[K]) // pattern[K][M]排列模
    式，Y[K][M]相空间重构分量的集合，links[K]重构分量与排列模式对应关系
{
   int k = 0; // 重构分量计数
   int row = 0; // 排列模式计数
   while (k < K)
   {
      // 复制一份重构分量，便于排序处理；准备序号数组，记录排列模式
      int raw[M], order[M]; // raw[M]复制分量，order[M]排序后对应序号
      for (int i = 0; i < M; i++)
      {
         raw[i] = Y[k][i];
         order[i] = i;
      }
      // 对复制的重构分量排序（冒泡排序），并记录改变后序号
      for (int i = 0; i < M - 1; i++)
      {
```

```
            for (int j = 0; j < M - 1 - i; j++)
            {
                if (raw[j] > raw[j + 1])
                {
                    // 重构分量排序
                    int temp1 = raw[j];
                    raw[j] = raw[j + 1];
                    raw[j + 1] = temp1;
                    // 序号同步排序
                    int temp2 = order[j];
                    order[j] = order[j + 1];
                    order[j + 1] = temp2;
                }
            }
        }
        // 判断是否已存在相同排列模式，对排列模式归类
        int flag = 1; // 1代表不存在，0代表已存在
        for (int i = 0; i < row; i++)
        {
            int cnt = 0;
            for (int j = 0; j < M; j++)
            {
                if (pattern[i][j] == order[j])
                    cnt++;
                else
                {
                    cnt = 0;
                    break;
                }
            }
            if (cnt == M) // 如果已存在，记录重构分量对应模式
            {
                flag = 0;
                links[k] = i;
                break;
            }
        }
        // 未存在则对排列模式数组添加新模式，并记录重构分量对应模式
        if (flag)
        {
            for (int i = 0; i < M; i++)
                pattern[row][i] = order[i];
            links[k] = row;
            row++;
        }
        k++;
    }
    return 0;
```

```
}
// F4 计算每种排列模式πk的加权概率值
int function4(double poss[K], double weight[K], int links[K]) // poss[K]排列模式的加
    权概率值，weight[K]重构分量的对应权重，links[K]重构分量与排列模式对应关系
{
   double sum = 0;
   for (int i = 0; i < K; i++)
   {
      sum += weight[i];
      poss[i] = 0;
   }
   for (int i = 0; i < K; i++)
   {
      poss[links[i]] += weight[i];
   }
   for (int i = 0; i < K; i++)
   {
      poss[i] = poss[i] / sum;
   }
   return 0;
}
// F5 计算时间序列X的加权排列熵WPE值
double function5(double poss[K]) // poss[K]排列模式的加权概率值
{
   // 此处选用参考的python文件log2，而非论文中ln
   double res = 0;
   for (int i = 0; i < K; i++)
   {
      if (poss[i] != 0)
         res -= poss[i] * log(poss[i]) / log(2);
   }
   // 归一化处理，除以log2(维数阶乘)
   int factorial = 1;
   for (int i = M; i >= 1; i--)
   {
      factorial *= i;
   }
   res = res / (log(factorial) / log(2));
   return res;
}
```

2.3 测试结果

2.3.1 原型测试结果

图 2.1: 原型代码运行截图

2.3.2 Cpp 实现结果

```
X[N]=
4 7 9 10 6 11 3

Y[K][M]=
4 7 9
7 9 10
9 10 6
10 6 11
6 11 3

avg[K]=
6.66667 8.66667 8.33333 9 6.66667

weight[K]=
4.22222 1.55556 2.88889 4.66667 10.8889

第0个重构分量排列模式: 0 1 2
第1个重构分量排列模式: 0 1 2
第2个重构分量排列模式: 2 0 1
第3个重构分量排列模式: 1 0 2
第4个重构分量排列模式: 2 0 1

pattern[K][M]=
[0]  0 1 2
[1]  2 0 1
[2]  1 0 2

links[K]=
重构分量0: pattern[0]
重构分量1: pattern[0]
重构分量2: pattern[1]
重构分量3: pattern[2]
重构分量4: pattern[1]

sum=24.2222

poss[K]=
0: 0.238532
1: 0.568807
2: 0.192661

PWE=0.546995039859119
```

图 2.2: Cpp 代码运行截图

2.4 问题及解决方案

(1) 问题：数学概念理解。
解决方案：广泛阅读相关文献，列思维导图，手算加权排列熵。

(2) 问题：排列模式统计。
解决方案：拆分为两步，先记录重构分量的排列模式，后比较归类统计。

(3) 问题：记录排列模式。
解决方案：另开序号数组，将序号数组和重构分量同步排序。

2.5 总结

本文主要介绍了加权排列熵的概念，并用 Cpp 实现了相关表达式的计算。

实际应用中，建议对本实现方式的输入输出再做改进，对本实现方式的代码再做精简。

参考文献

[1] Zhi Zhao. 排列熵、模糊熵、近似熵、样本熵的原理及 *MATLAB* 实现.https://blog.csdn.net/weixin_45317919/article/details/109254213.

[2] 马弄一下. 排列熵 *(permutation entropy)*.https://blog.csdn.net/fws920910fws/article/details/87890412.

[3] 丁嘉鑫, 王振亚, 姚立纲, 蔡永武. 广义复合多尺度加权排列熵与参数优化支持向量机的滚动轴承故障诊断 *[J]*. 中国机械工程,*2021,32(02):147-155.*