

# 站点聚类实验报告

组队：顾天阳 曾世茂 韩世民

学号：2019211539 2019211532 2019211496

班级：2019211308

指导老师：潘维民 邓芳

## 目录

一、	运行环境与操作指令 .....	1
1.	运行环境: .....	1
2.	操作指令: .....	2
二、	代码运行截图 .....	4
1.	运行时间戳 .....	4
2.	绘制散点图 .....	6
3.	并行计算 .....	7
三、	预处理和算法分析 .....	8
1.	预处理 .....	8
2.	算法分析 .....	9
四、	结果分析 .....	14

### 一、 运行环境与操作指令

- 1. 运行环境:
  - (1) 操作系统: Windows 10 家庭中文版 20H2 19042.1348
  - (2) 硬件信息:
    - CPU: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz
    - RAM: 8.00 GB (7.88 GB 可用)



## 2. 操作指令:

### (1) 依赖包安装:

通过 cmd 命令安装 kmeans 算法包 sklearn 命令如下:

`pip install sklearn`

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.19042.1348]
(c) Microsoft Corporation。保留所有权利。

C:\Users\16576>pip install sklearn
Looking in indexes: http://mirrors.aliyun.com/pypi/simple/
Collecting sklearn
  Downloading http://mirrors.aliyun.com/pypi/packages/1e/7a/dbb3be0ce9bd5c8b7e3d87328e79063f8b263b2b1bfa4774cb1147bfd3f/sklearn-0.0.tar.gz (1.1 kB)
  Preparing metadata (setup.py) ... done
Collecting scikit-learn
  Downloading http://mirrors.aliyun.com/pypi/packages/46/cc/a39f26ae8e39f1c197192b582553c9edf88ddb7303b15b8bb6e695198c11/scikit_learn-1.0.1-cp39-cp39-win_amd64.whl (7.2 MB)
  |#####| 7.2 MB 3.3 MB/s
Requirement already satisfied: numpy>=1.14.6 in d:\python\lib\site-packages (from scikit-learn->sklearn) (1.21.2)
Collecting joblib>=0.11
  Downloading http://mirrors.aliyun.com/pypi/packages/3e/d5/0163eb0cfa0b673aa4fe1cd3ea9d8a81ea0f32e50807b0c295871e4aab2e/joblib-1.1.0-py2.py3-none-any.whl (306 kB)
  |#####| 306 kB 6.4 MB/s
Requirement already satisfied: scipy>=1.1.0 in d:\python\lib\site-packages (from scikit-learn->sklearn) (1.7.3)
Collecting threadpoolctl>=2.0.0
  Downloading http://mirrors.aliyun.com/pypi/packages/ff/fe/8aaca2a0db7fd80f0b2cf8a16a034d3eea8102d58ff9331d2aaf1f06766a/threadpoolctl-3.0.0-py3-none-any.whl (14 kB)
Using legacy 'setup.py install' for sklearn, since package 'wheel' is not installed.
Installing collected packages: threadpoolctl, joblib, scikit-learn, sklearn
  Running setup.py install for sklearn ... done
Successfully installed joblib-1.1.0 scikit-learn-1.0.1 sklearn-0.0 threadpoolctl-3.0.0

C:\Users\16576>
```

### (2) 预处理数据:

编写 python 脚本 pre.py 对数据进行预处理, 删除错误数据, 规范数据集。

通过 cmd 命令运行 python 脚本, 命令如下:

`python pre.py`

```
C:\Windows\System32\cmd.exe

C:\Users\16576\Desktop\xxmh\新建文件夹 (2)>python pre.py

C:\Users\16576\Desktop\xxmh\新建文件夹 (2)>
```

- (3) 使用自己实现的 kmeans 算法对预处理后的数据聚类, 之前安装的 kmeans 算法依赖包可以用于检验结果:

```
kmeans.cpp  x  Data.cpp  test.py

kmeans.cpp ?
1 #include <ctime>
2 #include <fstream>
3 #include <iostream>
4 #include <sstream>
5 #include <map>
6 #include <vector>
7 #include <time.h>
8 #include <windows.h>
9 #include "Data.cpp"
10 #define DBL_MAX 1.7976931348623158e+308
11 using namespace std;
12 double kMeansClustering(vector<Data> *points, int k);
13 DWORD WINAPI UpdateMinDist(LPVOID lpParameter);
14 typedef struct PARAMAR
15 {
16     vector<Data> *datas;
17     vector<Data> *centerPoint;
18     int left;
19     int right;
20 } PARAMAR;
```

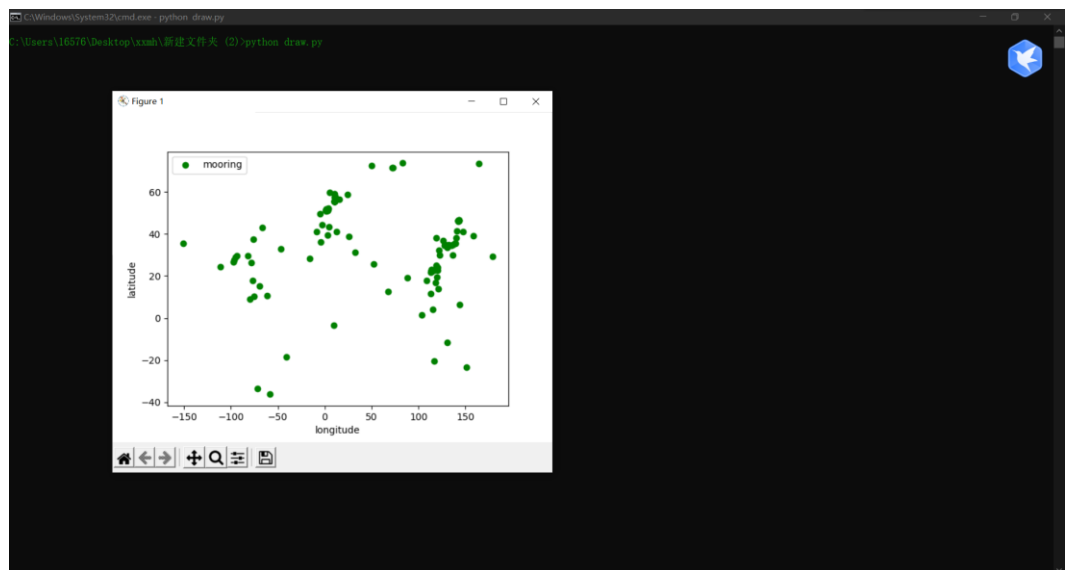
- (4) 使用 python 脚本对数据进行格式化处理, 并制作相应的图表:  
将结果转为 json 数组: python csvToJson.py

```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19042.1348]
(c) Microsoft Corporation. 保留所有权利。

C:\Users\16576\Desktop\xxmh\新建文件夹 (2)>python csvToJson.py

C:\Users\16576\Desktop\xxmh\新建文件夹 (2)>
```

绘制图表：python draw.py



## 二、 代码运行截图

### 1. 运行时间戳

入站：

```
kmeans.cpp X
kmeans.cpp > ...
#include <ctime>
#include <fstream>
#include <iostream>
#include <sstream>
#include <map>
#include <vector>
#include <time.h>
#include <windows.h>
#include "Data.cpp"
#define DBL_MAX 1.7976931348623158e+308
using namespace std;
double kMeansClustering(vector<Data> *points, int k);
DWORD WINAPI UpdateMinDist(LPVOID lpParameter);
typedef struct PARAMVAR
{
    vector<Data> *datas;
    vector<Data> *centerPoint;
    int left;
    int right;
} parameter;

int main()
{
    fstream fp("in.csv", ios::in);
    string temp;
    vector<Data> datas;
    if (!fp.is_open())
    {
        cout << "文件打开失败" << endl;
    }
}
```

D:\ShowMeTheCodeNo8B\ai>cd "d:\oBB\ai\" && g++ kmeans.cpp -o howMeTheCodeNo8B\ai\kmeans time cost: 0.883  
d:\ShowMeTheCodeNo8B\ai>

出站:

```
kmeans.cpp X out.csv
kmeans.cpp > main()
#include <ctime>
#include <fstream>
#include <iostream>
#include <sstream>
#include <map>
#include <vector>
#include <time.h>
#include <windows.h>
#include "Data.cpp"
#define DBL_MAX 1.7976931348623158e+308
using namespace std;
double kMeansClustering(vector<Data> *points, int k);
DWORD WINAPI UpdateMinDist(LPVOID lpParameter);
typedef struct PARAMVAR
{
    vector<Data> *datas;
    vector<Data> *centerPoint;
    int left;
    int right;
} parameter;

int main()
{
    fstream fp("out.csv", ios::in);
    string temp;
    vector<Data> datas;
    if (!fp.is_open())
    {
        cout << "文件打开失败" << endl;
        return 50;
    }
    while (!fp.eof())
    {
        getline(fp, temp);
        if (temp == "")
        {
            continue;
        }
    }
}
```

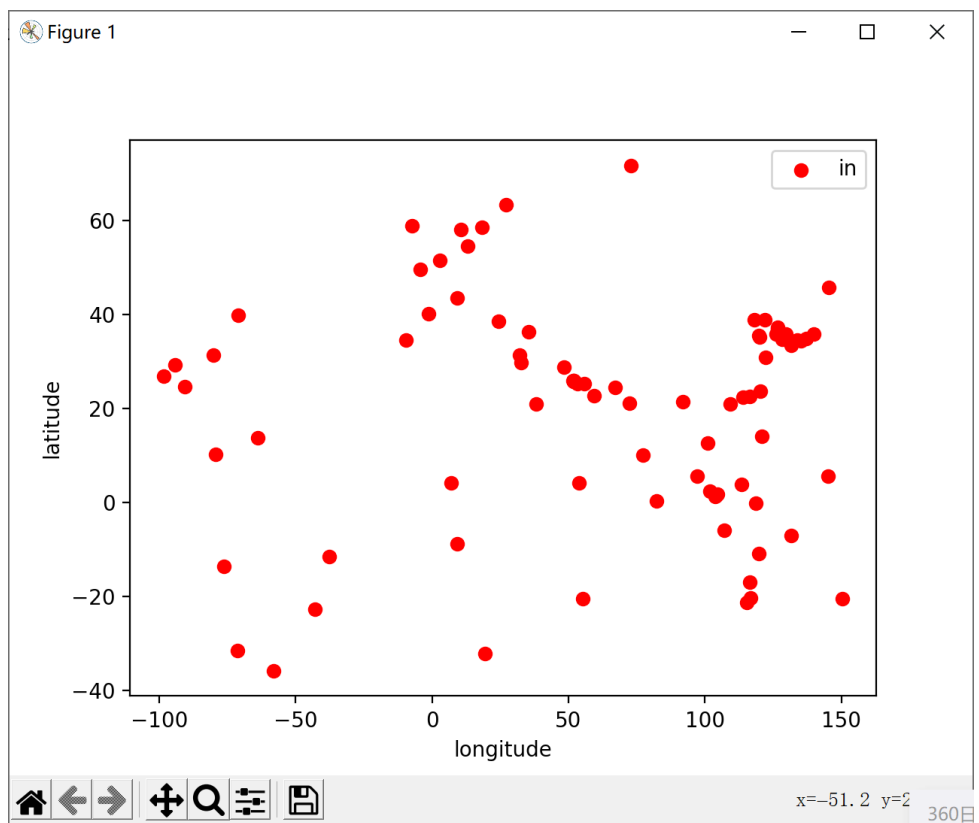
D:\ShowMeTheCodeNo8B\ai>cd "d:\oBB\ai\" && g++ kmeans.cpp -o howMeTheCodeNo8B\ai\kmeans time cost: 0.475  
d:\ShowMeTheCodeNo8B\ai>

锚地:

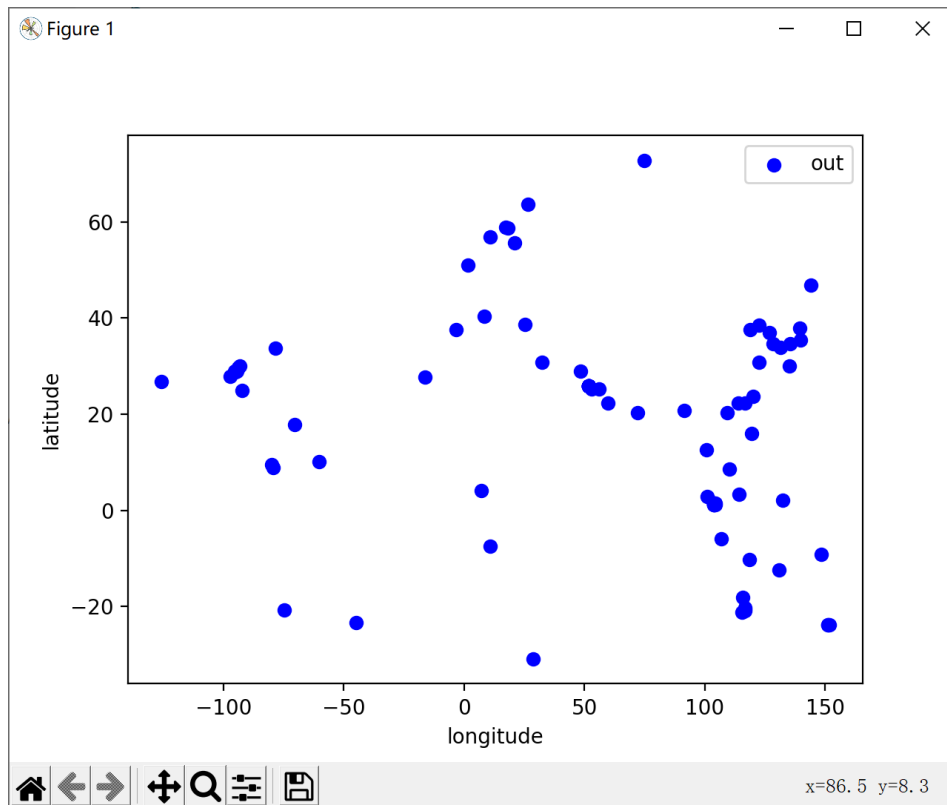
```
kmeans.cpp x Data.cpp letSeep.py
kmeans.cpp > ...
1 #include <ctime>
2 #include <fstream>
3 #include <iostream>
4 #include <sstream>
5 #include <map>
6 #include <vector>
7 #include <time.h>
8 #include <windows.h>
9 #include "Data.cpp"
10 #define DBL_MAX 1.7976931348623158e+308
11 using namespace std;
12 double kMeansClustering(vector<Data> *points, int k);
13 DWORD WINAPI UpdateMinDist(LPVOID lpParameter);
14 typedef struct PARAMAR
15 {
16     vector<Data> *datas;
17     vector<Data> *centerPoint;
18     int left;
19     int right;
20 } parameter;
21
22 int main()
23 {
24
25     fstream fp("mooring.csv", ios::in);
26     string temp;
27     vector<Data> datas;
28     if (!fp.is_open())
29     {
30         cout << "文件打开失败" << endl;
31         return 50;
32     }
33     while (!fp.eof())
34     {
35         getline(fp, temp);
36         if (temp == "")
37         {
38             continue;
39         }
40         Data data(temp);
41         datas.push_back(data);
42     }
```

## 2. 绘制散点图

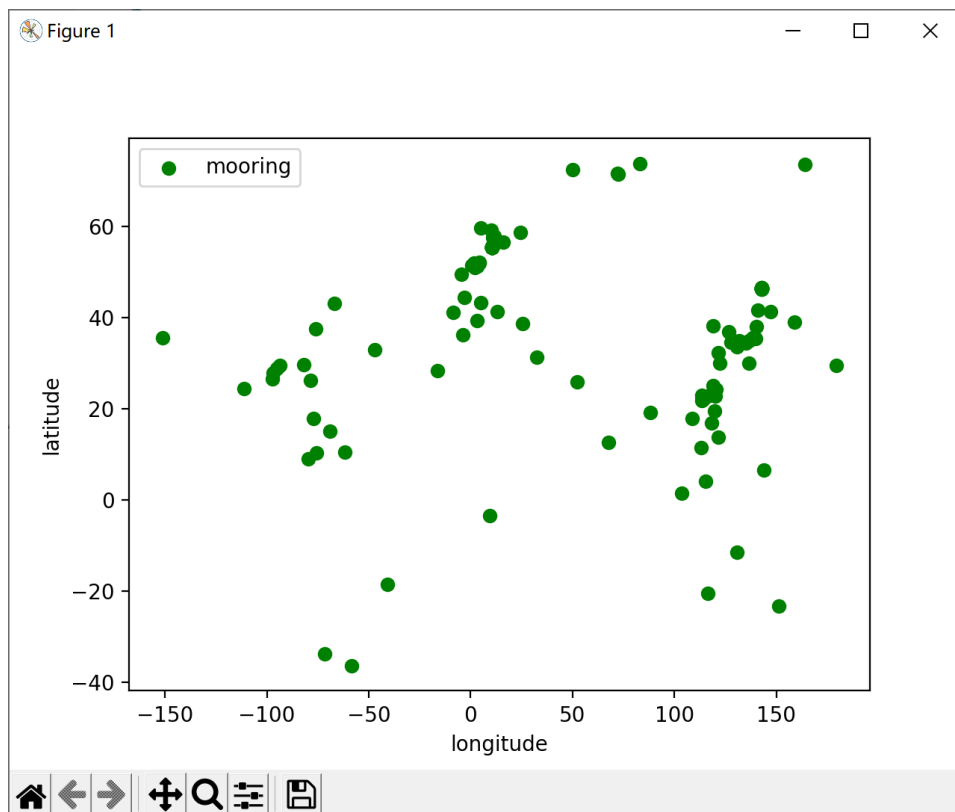
进站:



出站:



锚地:



### 3. 并行计算

大多数程序运行时间都消耗更新每个点最近的中心点的计算中。由于该计算需要

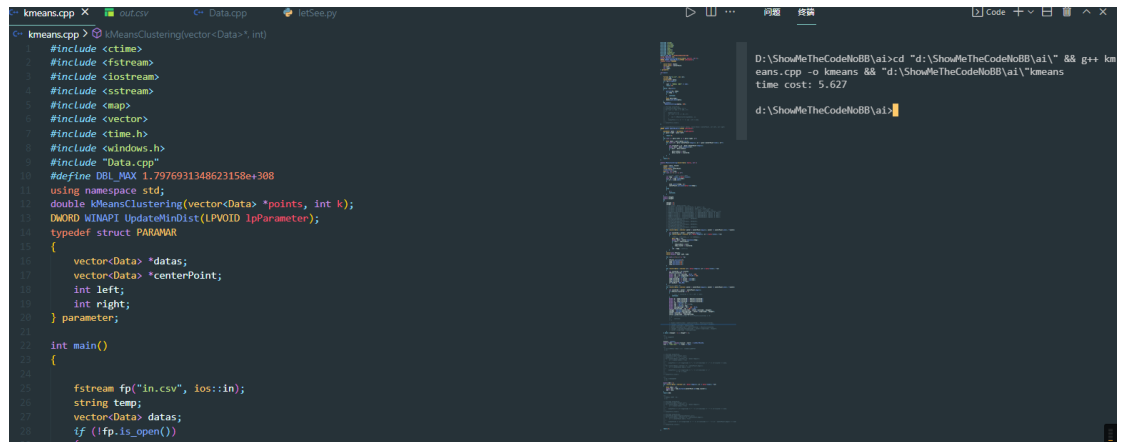
遍历所有的点，每个点均需遍历所有的中心并计算与该中心的距离，找出距离最小的中心点并将自己的类别设置成该中心所属的类。故该过程的时间复杂度为  $O(NK)$ 。而在比较各个点与各个中心最小距离的时候，各个点的计算过程是相互独立的，所以可以并行计算以加快处理过程。在该程序中，小组结合所使用设备 **cpu** 最大核心数为 4 的特点，使用了四线程并行计算处理该过程。成功将平均运行时间缩短了 4 倍。

运行时间截图：

数据：LNG 入站,点

数据量 1w

分类数：180



```
kmeans.cpp X out.csv Data.cpp letSee.py
kmeans.cpp > kMeansClustering(vector<Data>*, int)
#include <ctime>
#include <fstream>
#include <iostream>
#include <sstream>
#include <map>
#include <vector>
#include <time.h>
#include <windows.h>
#include "Data.cpp"
#define DBL_MAX 1.7976931348623158e+308
using namespace std;
double kMeansClustering(vector<Data> *points, int k);
DWORD WINAPI UpdateMinDist(LPVOID lpParameter);
typedef struct PARAMAR
{
    vector<Data> *datas;
    vector<Data> *centerPoint;
    int left;
    int right;
} parameter;

int main()
{
    fstream fp("In.csv", ios::in);
    string temp;
    vector<Data> datas;
    if (!fp.is_open())
```

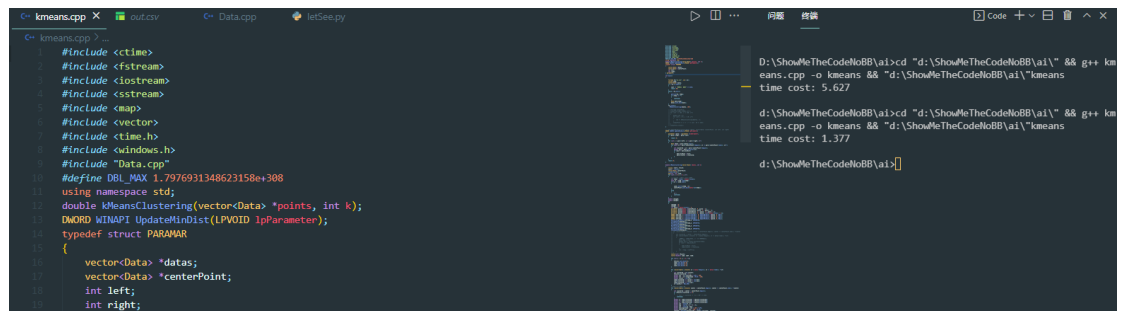
D:\ShowMeTheCodeNoBB\ai>cd "d:\ShowMeTheCodeNoBB\ai" && g++ kmeans.cpp -o kmeans && "d:\ShowMeTheCodeNoBB\ai\kmeans"
time cost: 5.627
d:\ShowMeTheCodeNoBB\ai>

不使用并行计算加速：

运行时间 5.627s

使用 4 线程并行计算加速：

运行时间：1.377s



```
kmeans.cpp X out.csv Data.cpp letSee.py
kmeans.cpp > ...
#include <ctime>
#include <fstream>
#include <iostream>
#include <sstream>
#include <map>
#include <vector>
#include <time.h>
#include <windows.h>
#include "Data.cpp"
#define DBL_MAX 1.7976931348623158e+308
using namespace std;
double kMeansClustering(vector<Data> *points, int k);
DWORD WINAPI UpdateMinDist(LPVOID lpParameter);
typedef struct PARAMAR
{
    vector<Data> *datas;
    vector<Data> *centerPoint;
    int left;
    int right;
} parameter;

int main()
{
    fstream fp("In.csv", ios::in);
    string temp;
    vector<Data> datas;
    if (!fp.is_open())
```

D:\ShowMeTheCodeNoBB\ai>cd "d:\ShowMeTheCodeNoBB\ai" && g++ kmeans.cpp -o kmeans && "d:\ShowMeTheCodeNoBB\ai\kmeans"
time cost: 5.627
d:\ShowMeTheCodeNoBB\ai>cd "d:\ShowMeTheCodeNoBB\ai" && g++ kmeans.cpp -o kmeans && "d:\ShowMeTheCodeNoBB\ai\kmeans"
time cost: 1.377
d:\ShowMeTheCodeNoBB\ai>

加速比：  $5.627 / 1.377 \approx 4$

与预想中的加速效果以及使用线程数一致，也印证了该算法主要时间消耗点位于计算更新中心点处，将多线程加速用于该处理过程是正确的选择。

### 三、预处理和算法分析

#### 1、预处理

1. # 读取 lng2.csv 中的数据，经过预处理分类输出到各个文件中
2. f = open('lng2.csv', 'r+')
3. f1 = open('in.csv', 'w+')
4. f2 = open('out.csv', 'w+')



```

5. f3 = open('mooring.csv', 'w+')
6.
7. is_first = True # 是否为第一个有效数据
8. draft, last_draft = 0, 0 # 当前吃水深度和上一个吃水深度
9. for line in f.readlines():
10.     line = line.split()
11.     draft = int(line[6])
12.     if draft == 0 or draft > 300:
13.         continue
14.
15.     if is_first == True:
16.         is_first = False
17.         f3.write(' '.join(line) + '\n')
18.     # 如果小于上一个吃水深度, 则加入入站
19.     elif draft < last_draft:
20.         f1.write(' '.join(line) + '\n')
21.     # 如果大于上一个吃水深度, 则加入出战
22.     elif draft > last_draft:
23.         f2.write(' '.join(line) + '\n')
24.     # 如果等于上一个吃水深度, 则加入锚地
25.     elif draft == last_draft:
26.         f3.write(' '.join(line) + '\n')
27.     # 更新上一个吃水深度
28.     last_draft = draft
29.
30. f.close()
31. f1.close()
32. f2.close()
33. f3.close()

```

结果中锚地数据约为百万级别, 入站和出战为万级别, 符合实际中锚地较大导致站点个数过多的事实。

## 2、算法分析

### (1) 算法步骤

kmeans 算法的主要核心思想很简单。主要的步骤如下:

- 1、首先根据给出的簇数目  $k$  选出  $k$  个初始中心点。在该实验中, 小组选择了随机设置初始中心点。
- 2、随后遍历每一个点, 找出与其距离最短的中心点, 并将自己所属的类设为该中心点的类。
- 3、遍历完所有的点之后, 每个中心点的坐标更改成自己所在类成员的中心点。
- 4、重复步骤 2 和 3, 直到满足退出条件

在该实验中, 由于考虑到各个站点之间的距离不会靠的太接近, 故小组将退出条件设置成更新完中心点的坐标后, 每个新坐标与锚坐标的经度纬度变化均不超过 1, 则认为所有中心点已经稳定, 聚类结束退出。

## (2) 算法实现:

全局变量以及类定义:

类定义:

```
1. class Data
2. {
3.     string msi;
4.     long long int time;
5.     int state;
6.     int volicity;
7.     int load;
8.     double latitude; //纬度
9.     double longitude; //经度
10.    int cluster; //所属的类
11.    double minDist; //到所属中心的距离
12. }
```

结构体定义:

```
1. typedef struct PARAMAR //用于多线程传递参数
2. {
3.     vector<Data> *datas; //所有数据的集合
4.     vector<Data> *centerPoint; //所有中心点的集合
5.     int left; //该线程的任务划分起始下标
6.     int right; //该线程任务划分结束下标, 左闭右闭
7. } parameter;
```

全局变量定义:

```
1. #define DBL_MAX 1.7976931348623158e+308 // double 最大值
2. #define EARTH_RADIUS 6378.137 //地球半径
3. #define M_PI 3.14159265358979323846 //圆周率
```

## Kmeans 算法

```
1. void kMeansClustering(vector<Data> *datas, int k)//输入样本集以及 k 值
2. {
3.     vector<Data> centerPoint;
4.     for (int i = 0; i < k; i++)
5.     {
6.         ramdonlySelectKpoint();
7.     }
8.     double changeX, changeY; //更新中心点后每个中心点的经纬度变化最大值
9.     do
10.    {
11.        changeX = 0;
12.        changeY = 0;
```

```

13.     updateMinDist();
14.     vector<int> pointsNum;
15.     vector<double> sumX, sumY, sumZ;
16.     //改变每个中心点位置
17.     for (vector<Data>::iterator center = centerPoint.begin(); center != centerPoint.end(); ++center)
18.     {
19.         int clusterId = center - centerPoint.begin();
20.         if (pointsNum[clusterId] == 0)
21.         {
22.             //说明该类出现了空簇
23.             continue;
24.         }
25.         calculateNewCenterPoint();
26.         changeX = max(abs(newLatitude - center->latitude), changeX);
27.         changeY = max(abs(newLongitude - center->longitude), changeY);
28.         center->latitude = newLatitude;
29.         center->longitude = newLongitude;
30.     }
31. } while (changeX > 1 || changeY > 1); //只要有一个中心经纬度变化超过 1 则继续聚类
32. }

```

计算每个点到每个中心的最短距离，并更新所属的类

```

1.  DWORD WINAPI UpdateMinDist(LPVOID lpParameter) //多线程实现
2.  {
3.      /* 原包含参数:
4.      vector<Data> *datas; 数据集
5.      vector<Data> *centerPoint; //中心集
6.      int left; //该线程的任务划分起始下标
7.      int right; //该线程任务划分结束下标, 左闭右闭类型
8.      */
9.      parameter *para = (parameter *)lpParameter;
10.     if (para->right < para->left)
11.     {
12.         return 0;
13.     }
14.     for (int i = para->left; i <= para->right; i++)
15.     {
16.         Data &data = para->datas->at(i); //遍历每一个该线程的划分节点
17.         for (auto it = para->centerPoint->begin(); it != para->centerPoint->end(); it++) //遍历每一个中心
18.         {
19.             int clusterId = it - para->centerPoint->begin(); //得到中心 id
20.             double dist = data.distance(*it); //计算距离

```

```

21.         if (dist < data.minDist)
22.         {
23.             data.minDist = dist;
24.             data.cluster = clusterId;//更新中心
25.         }
26.     }
27. }
28. return 0;
29. }

```

根据经纬度计算两点距离

```

1. double Data::GetDistance2(double lng1, double lat1, double lng2, double lat2)
2. {
3.     double radLat1 = rad(lat1);
4.     double radLat2 = rad(lat2);
5.     double radLng1 = rad(lng1);
6.     double radLng2 = rad(lng2);
7.     double s = acos(cos(radLat1) * cos(radLat2) * cos(radLng1 -
        radLng2) + sin(radLat1) * sin(radLat2));
8.     s = s * EARTH_RADIUS;
9.     s = round(s * 1000 * 1000000) / 1000000;
10.    return s;
11. }
12.
13. double Data::rad(double LatOrLon)
14. {
15.     return LatOrLon * M_PI / 180.0;
16. }

```

(3) 运行测试以及效果截图：

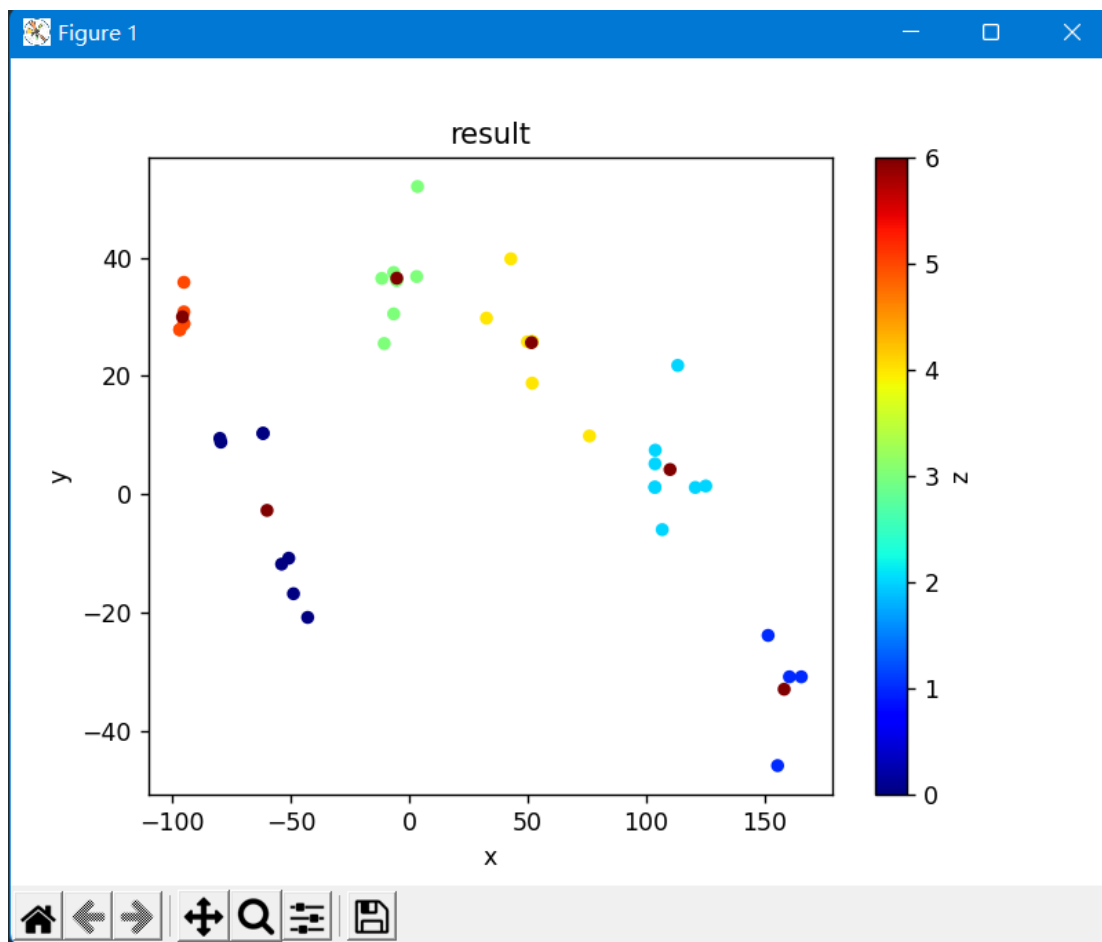
数据源：随机挑选的 41 条数据

```
636092943 1635438946 1 2 103.651054 1.218048 112
605076060 1630962335 1 0 3.121508 36.806938 92
565515000 1638092574 1 1 -96.905754 27.817961 92
565515000 1633827602 1 0 -95.158806 28.839363 93
565513000 1634401327 1 9 -96.861557 27.825245 93
538003361 1629497421 1 0 -48.875149 -16.791197 95
565513000 1623412118 1 4 3.467202 52.015274 113
564988000 1633905914 1 6 32.581390 29.772842 94
564988000 1628946380 1 0 113.316536 21.785908 113
564988000 1620434595 1 0 -79.517883 8.821743 98
538003361 1629497421 1 0 -42.875149 -20.791197 95
563574000 1632091793 1 1 -5.308000 36.115414 100
563574000 1626989699 1 0 -61.716339 10.306280 102
563574000 1622482981 1 0 -61.715492 10.308493 91
563214000 1634151035 1 0 125.143547 1.414750 90
563214000 1627597238 1 0 103.823280 7.464917 20
563214000 1621537620 1 0 106.769981 -5.961800 89
563178000 1626844951 1 1 76.046867 9.868615 92
563178000 1624411991 1 0 51.810036 25.826332 90
563126800 1629241449 1 0 -79.941978 9.456280 110
563121700 1634414838 1 0 103.693237 1.183985 92
538003361 1629497421 1 0 51.875149 25.791197 95
538003361 1629497421 1 0 -50.875149 -10.791197 95
538003361 1629497781 1 0 51.875103 25.791178 95
538003361 1629498141 1 0 51.875149 18.791155 95
538003361 1629498319 1 0 49.875256 25.791283 95
538003361 1629498679 1 0 42.875259 39.791288 95
538002609 1638062710 1 1 -11.589705 36.490543 93
538002609 1638063781 1 3 -6.589617 37.490292 93
538002609 1638064202 1 9 -10.587525 25.490269 93
538002609 1638064509 1 4 -6.587072 30.490528 93
477157300 1631202450 1 0 165.462234 -30.831299 94
477157300 1631202811 1 0 155.462265 -45.831284 94
538003361 1629497421 1 0 -53.875149 -11.791197 95
477157300 1631203172 1 0 151.462326 -23.831314 94
477157300 1631203236 1 0 160.462326 -30.831314 94
355998000 1637516291 1 2 103.725571 5.163348 72
355998000 1637516823 1 1 120.725975 1.162060 72
311000968 1634918517 1 0 -95.124405 30.826157 97
311000968 1634918519 1 0 -95.124405 28.826157 97
311000968 1634918707 1 0 -95.124405 35.826141 97
```

聚类 k 数: 6

聚类结果: (不同颜色代表不同的簇, 深红色的点代表该簇的中心位置)

运行时间: 0.003s



聚类结果分析：

分析运行结果图可以看出，各个簇之间形状较为规整，没有与其他簇相互交叉，各个簇的成员间距离也较为接近，每个簇的质心均位于所在簇的中心处，由以上分析可得，该算法的聚类效果较好。

#### 四、结果分析

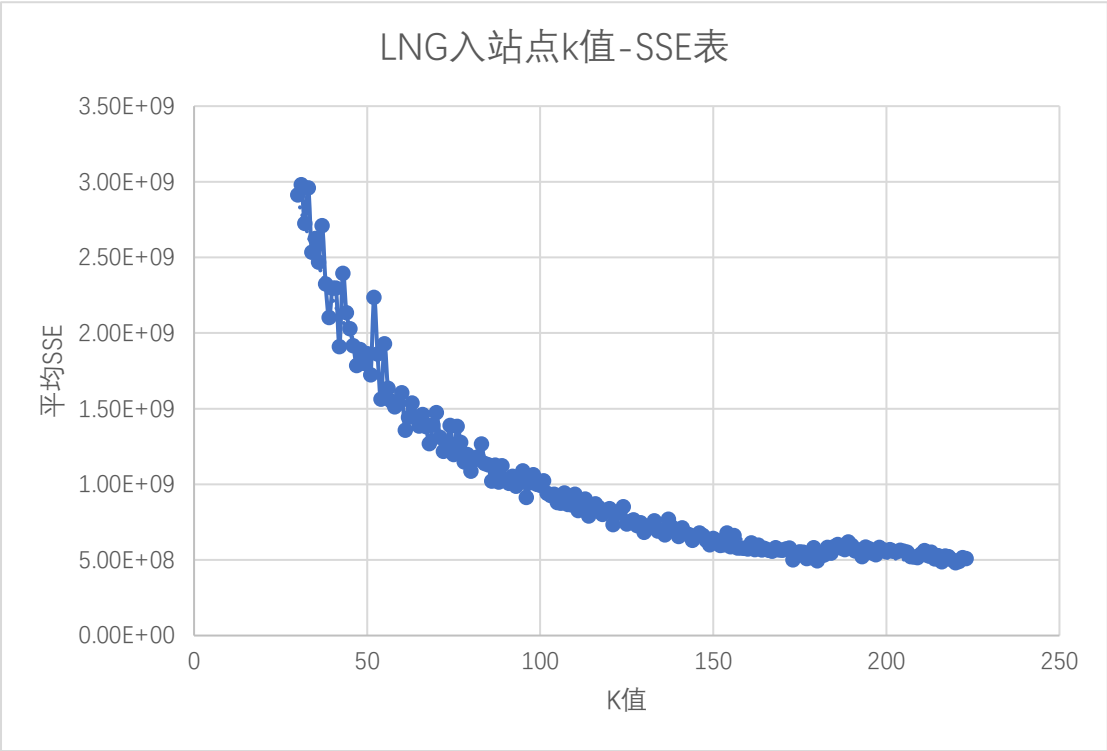
将  $k$  值从 30~230 分别运行 `kmeans`，每次运行都计算该次运行结果的误差平方和  $SSE$ ， $SSE$  由以下公式给出：其中  $k$  为聚类数量， $p$  为每一个样本， $m_k$  为第  $k$  个聚类的中心点。 $SSE$  随着  $k$  增大而减少，说明样本聚合程度随着  $k$  增大而增高。

$$SSE = \sum_{k=1}^k \sum_{p \in C_k} |p - m_k|^2$$

当  $k$  小于真实聚类数时，由于  $k$  的增大会大幅增加每个簇的聚合程度，故  $SSE$  的下降幅度会很大，而当  $k$  到达真实聚类数时，再增加  $k$  所得到的聚合程度回报会迅速变小，所以  $SSE$  的下降幅度会骤减，然后随着  $k$  值的继续增大而趋于平缓，这个最先趋于平缓的点就是合适的  $K$  值。

在该实验中，为了避免由于 `Kmeans` 初始选择中心引起的  $SSE$  偶然性突变，每个  $k$  均运行 10 次，并求平均值得出该  $k$  值对应的  $SSE$ 。结果如下：

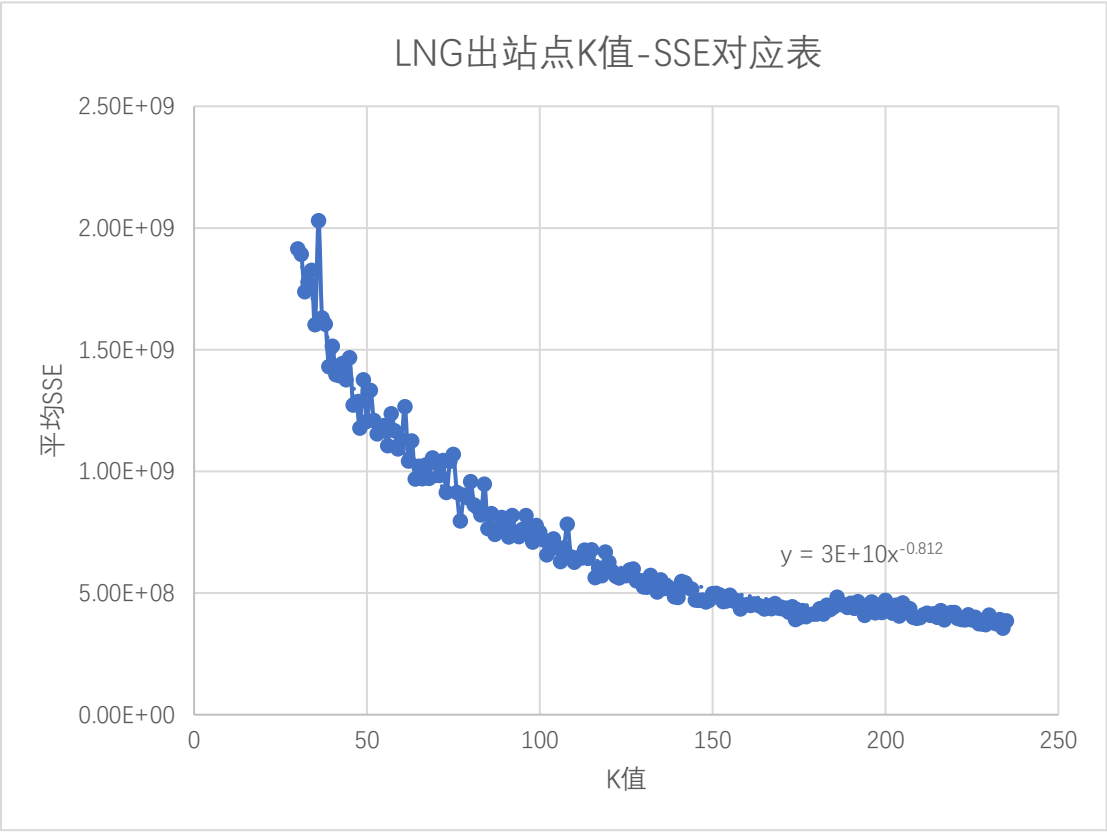
LNG 入站点：



得到的拟合图像函数为

$$y = 6E + 10x^{-0.908}$$

LNG 出站点:



得到的拟合图像函数为

$$y = 3E + 10x^{-0.812}$$

于是这两个图像的肘点——即最大曲率点对应的  $k$  值即为对应的最佳  $k$  值。《Finding a “Kneedle” in a Haystack: Detecting Knee Points in System Behavior》<sup>i</sup> 中指出了一种快速找出  $k$  值的方法：将曲线首尾相连得到一条直线，则与该直线距离最大的点即为肘点的一个近似取值。

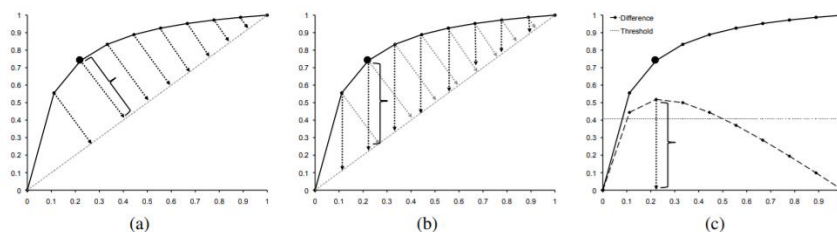
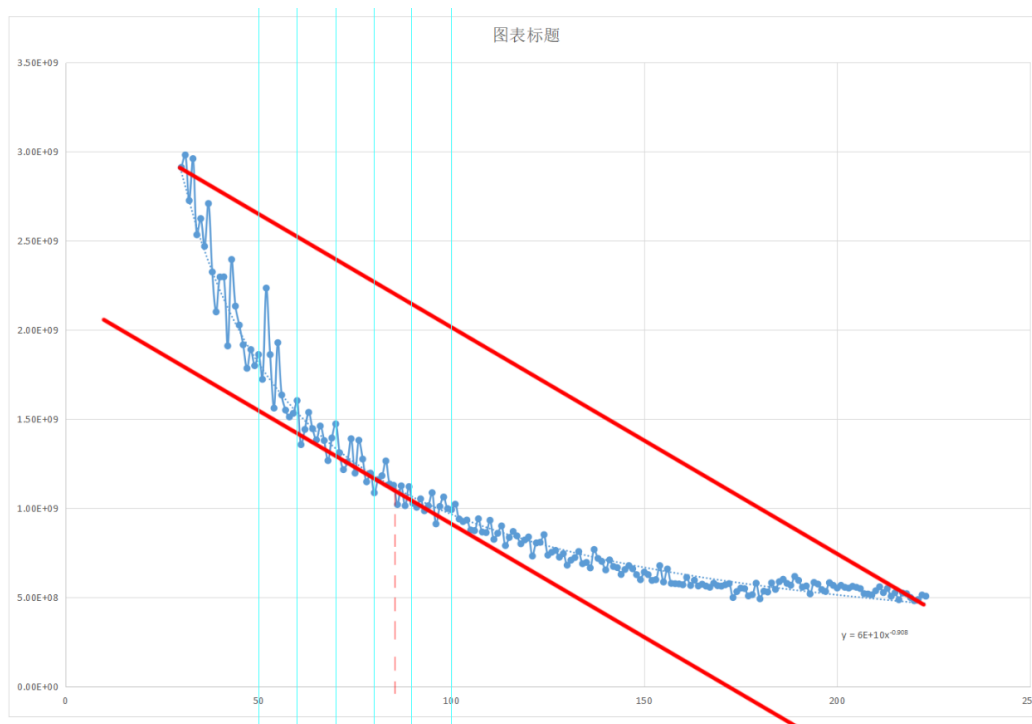


Fig. 2: Kneedle algorithm for online knee detection. (a) depicts the smoothed and normalized data, with dashed bars indicating the perpendicular distance from  $y = x$  with the maximum distance indicated. (b) shows the same data, but this time the dashed bars are rotated 45 degrees. The magnitude of these bars correspond to the difference values used in Kneedle. (c) shows the plot of these difference values and the corresponding threshold values (with  $S = 1$ ). The knee is found at  $x = 0.22$  and is detected after receiving the point  $x = 0.55$ .

结果如下所示：

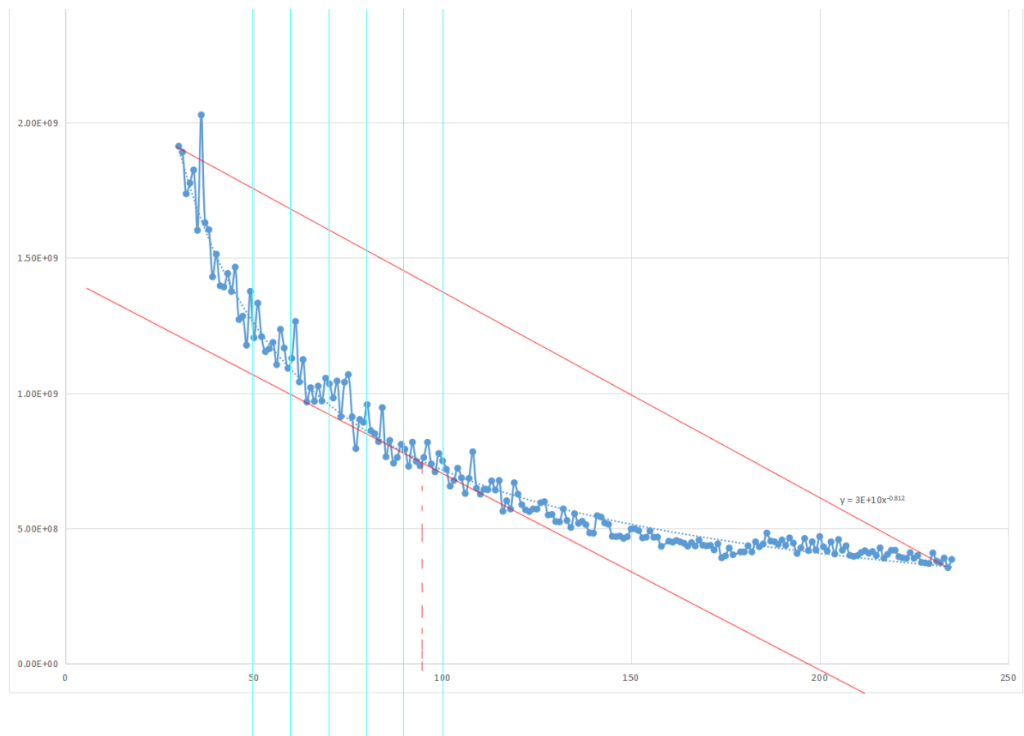
LNG 入站点：



相交处对应的  $K$  值约为 85，故在 LNG 入站点聚类中，选择  $K$  值 85。

LNG 出站点：





相交处对应的 K 值为 95，故在 LNG 入站点聚类中，选择 k 值为 95。出入站点之和为 180，与给出的 182 个站点相吻合

---

<sup>i</sup> Satopaa V , Albrecht J , D Irwin, et al. Finding a Kneedle in a Haystack: Detecting Knee Points in System Behavior[C]// International Conference on Distributed Computing Systems Workshops. IEEE Computer Society, 2011.